Module 2

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 2 Study Guide and Deliverables

Readings: Required Reading:

- CB6 chapters 19, 20, and 23.6

Recommend Reading:

- Loney chapters 46–48 and 36

Assignments:

- Assignment 2.0 due Wednesday, July 22 at 5:00 PM ET
- Programming Part 2 due Sunday, July 26 at 6:00 AM ET

Assessments: Quiz 2 due Wednesday, July 22 at 5:00 PM ET

Term Project Note: Term Project Update #2, is due Sunday, July 26 at 6:00 AM ET.

During this module you should finalize the definition of your project, working with your instructor. You should develop a project definition document with a project plan, and should obtain approval for this project. Your updated term project concept can be different than the concept submitted for the Module 1 Term Project Proposal. Still, it is risky to change your term project concept after this module, because

	you may not have sufficient time to complete your project.
Live Classrooms:	Supplementary Live Session, Wednesday July 15, 8:00 PM- 10:00 PM ET
	Current week's assignment review and examples, Thursday July 16, 8:00 PM- 9:00 PM ET
	Live office help, Saturday, July 18, 11:00 AM - 12:00 PM ET
	Live office help, Monday, July 20, 8:00 PM - 8:45 PM ET

Lecture 2A - Relational DB Performance Tuning

Overview - Tuning Relational Databases

This lecture describes DBMS-independent techniques for designing relational databases and SQL statements that perform well as databases scale. Many performance tuning techniques are DBMS-specific. Consult your DBMS vendor's tuning documentation and independent DBMS-specific tuning guides. The goal of this lecture is to provide you with an opportunity to learn how to design scalable relational database applications systems, to determine what is causing performance problems, and to fix performance problems. These are critical skills for enterprise-scale production databases, data marts, data warehouses, and many other commercial database applications.

We begin this lecture by introducing the basic relational database performance tuning concepts, including an introduction to RDBMS software architecture. We then move to tuning SQL statements, including the tuning process and tuning rules. Next we describe how to tune the database itself, including tuning the server hardware, tuning the instance parameters, designing fast relational schemas, indexing for performance, and hash and bitmap indexes. Finally we discuss tuning techniques that affect both the SQL and database, such as stored procedures.

Performance tuning is introduced in Connolly and Begg sixth edition Chapter 19 *Methodology—Monitoring and Tuning the Operational System*. This lecture builds on that introduction, including more advanced material. For students who are interested, general and Oracle-specific tuning is introduced in Loney 12c Chapter 46 *The Hitchhiker's Guide to Tuning Applications and SQL* and Chapter 48 *Case Studies in Tuning*. You are not responsible for the Oracle- specific techniques, though you may find some of this material useful for a term project.

Learning Objectives

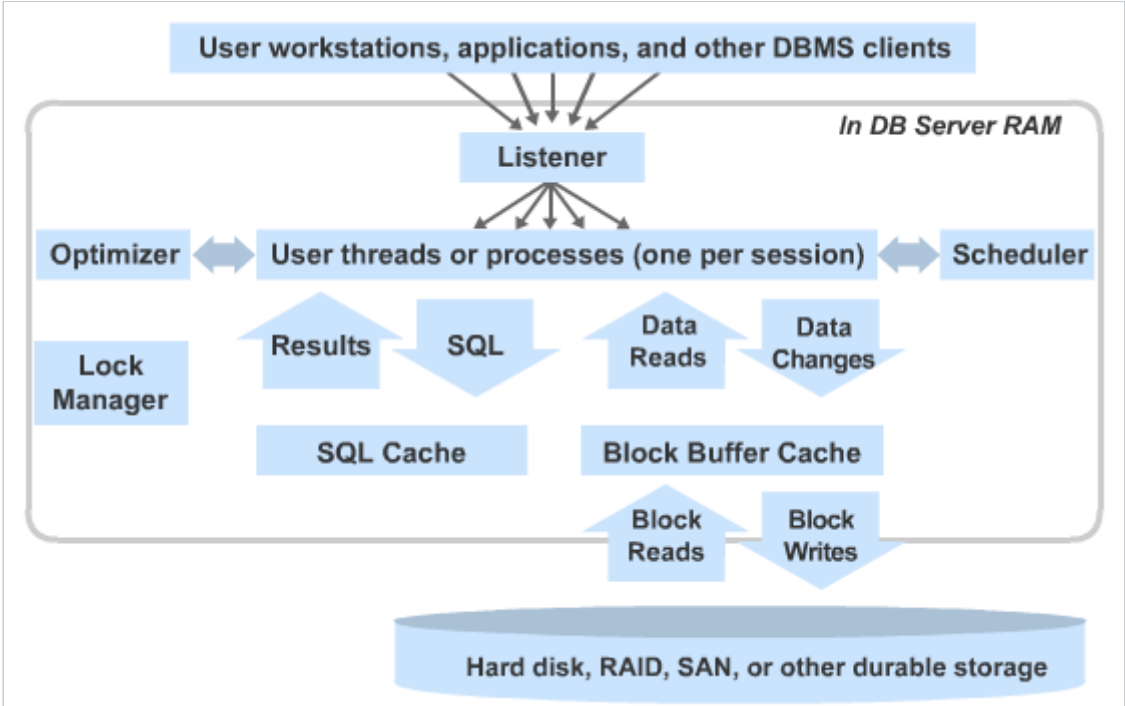
By completing the readings, participating in discussions, and completing the assignments, you will be able to:

- Describe why it is important to design scalable databases.
- Describe the three dimensions of scalable performance.
- Design efficient scalable relational schemas.
- Tune SELECT and other SQL requests.
- Design indexes for efficient request execution.
- Explain and use B-tree, hash, and bitmap indexes.
- Identify and correct common scalability problems.

SQL DBMS Data Flow Architecture

We begin by examining what goes on inside a SQL RDBMS, to help us understand how to help an RDBMS execute our requests faster. The following figure illustrates the basic components of an RDBMS.

Relational DBMS Data Flow Architecture



Test Yourself

Where is the SQL Cache located in the database server?

RAM

This is true. SQL Cache is located in the RAM.

Durable Storage

This is false. SQL Cache would be slower on durable storage.

Both RAM and Durable Storage

This is false. SQL Cache would be slower even partially on durable storage.

The Block Buffer Cache and the SQL Cache

The Block Buffer Cache and Performance

With modern processors and disk speeds, operations on data in the block buffer cache are at least 100 times faster than operations requiring I/O to disk. As a result much of what we do to optimize our RDBMS applications is based on minimizing the amount of I/O required to disk or other storage. Requests that can be processed using data in the cache (cache hits) are much faster than SQLs that require I/O (cache misses). The *cache hit ratio* (hits/(hits+misses)) is a key performance metric, which should usually be above 90%. We can maximize the cache hit ratio by providing lots of RAM for a large block buffer cache. *DBMS love RAM.* When there is more RAM, the DBMS can keep more data in the caches—and this means more cache hits and better performance. Smaller, frequently accessed tables and indexes are normally in the block buffer cache. Not much of a large table or index will fit in the cache, so processing a significant fraction of a large table or large index costs lots of I/O and time.

The SQL Cache

Sometimes applications will send the same SELECT statement to the DBMS over and over again. The SQL cache makes this reasonably efficient by caching the SQL and the result set for SELECT statements. The SQL cache monitors database changes to determine when the same SQL would produce the same cached result set. If a query would produce the same result set as a previous query, then the SQL cache returns that cached result set, and the SQL isn't actually executed. The SQL cache thus shunts the entire RDBMS processing of the SQL. Because the cache eliminates the need to produce or execute a query plan, the SQL cache can make a significant difference in SELECT performance.

In most modern commercial RDBMS the SQL cache also caches the query plan, eliminating the cost and delay of running the query optimizer, which can be computationally expensive. The cached query plan may be valid even though the cached result set is no longer valid, because data upon which the query depends may have changed. Query plan caching is particularly helpful with Microsoft SQL Server, which has an unusually computationally expensive query optimizer. The SQL cache holds result sets for SELECT statements; it may also hold the query plan for other SQL statement types.

Often repeated SELECTs are due to immature software, where the application developers didn't save the result set and just reran the SELECT to fetch it again. Sometimes these repeated SELECTs are necessary for correct functioning of an application. For example, a financial application may need to query an interest rate each time a transaction is performed, to be sure that the interest rate is the correct one to use for the application at that time.

Test Yourself

Select all that are true about improving performance with cache optimization.

The cache hit ratio should be high and can be maximized with lots of RAM.

This is true.

The SQL cache can cache large result sets if the cache size is configured properly.

This is true. A SQL cache ("SQL Result Cache" in Oracle) keeps query plans for SQL statement types like triggers and procedures, and keeps the result sets for SELECT statements.

The block buffer cache keeps all the information about the SQL operation, including data and SQL statements.

This is false. The block buffer cache only contains data.

The Scalable Performance Problem

When the amount of data in a database is smaller than the physical RAM on the computer system, the database performs well regardless of scalable design problems. With un-scalable designs, when the database scales many requests take much longer—often unacceptably longer. Obtaining scalable performance is often the most difficult task faced by database designers and application developers. Fixing scalable performance problems often requires major redesign and rework, such as schema changes and rewriting thousands or millions of lines of application SQL. This can be very expensive, so it is important that database designs be scalable, particularly the database schema.

An Easy Scalability Fix

A two terabyte constellation data warehouse for which I was a design consultant didn't have an index that it needed, so a new query took an hour. When I added the index the query took a few seconds. This fix took me about thirty minutes, most of which was the runtime to create the new index. We didn't even need to interrupt the users; all that they knew was that the query results suddenly became much faster.

A Difficult Scalability Fix

With an un-scalable design, a report that completed in three seconds with a small database took over two hours with a production database. I designed a new database architecture and data management system and supervised a two-year effort to incrementally redevelop the application. With the new architecture, the same queries took less than three seconds with much larger databases.

Truly scalable designs are those where the DBMS does not operate on more data for each request as the database and load scale. Only a small fraction of the rows of a large table or index can be processed for each SQL. Scans are not scalable. Thus operations against tables that may grow in size must use indexes or other means to limit the number of rows accessed.

A second important consideration is minimizing the size of intermediate results sets that the DBMS produces as part of executing the query plan. Large intermediate result sets need to be processed on storage (e.g., disk sorts), which is very slow compared to RAM. Good scalable performance requires query plans with intermediate result sets that fit in RAM. The query optimizer tries to sequence the restrictions so the intermediates are as small as possible. Avoiding large intermediates and disk sorts requires careful schema and query design and adequate RAM.

Test Yourself

What is the most important consideration when improving scalable performance?

It is easy to rewrite applications SQL.

This is false. It can take years to rewrite millions of lines of application SQL.

RAM is faster than durable storage.

This is true. If a result set can fit into RAM it can be processed much faster than durable storage.

Scans are always scalable.

This is false. Scans are not scalable.

Result sets can be processed efficiently even if they do not fit into RAM.

This is false. While result sets can be processed even if they do not fit into RAM it does not help scalable performance.

Size, Throughput, and Session Scalability

The three main measures of database scalability are how well the DB maintains performance as

- **The DB grows**—This is ***size scalability***. Software factors affecting size scalability include the application SQL, schema, and indexing. Hardware factors include I/O, storage type and configuration, and CPU speed and number.
- **The transaction rate increases**—This is ***throughput scalability***. Throughput scalability is primarily determined by lock contention, DB server concurrency, and platform performance.
- **The number of simultaneous sessions increases**—This is user or ***session scalability***. More concurrent sessions means more opportunities for lock contention. Session scalability is driven by schema design and DBMS concurrency support.

Of these three size scalability is often the most difficult to obtain. Size scalability requires a good scalable schema, a well-configured DBMS, appropriate hardware, tuned SQL, careful indexing, good storage hardware and configuration, and adequate speed and number of processors. Table and index scans are slow on large databases, so size scalability requires indexing for all queries against large tables. There are many kinds of indexing.

Good throughput scalability requires a good design and a fast scalable DBMS. If you require scalable throughput be sure to investigate throughput scalability when choosing the DBMS and architecture. You can sometimes obtain scalability by factoring and/or distributing the database, but be aware that many database applications do not factor well.

When there are many concurrent users or other sessions and locks are being procured, there may be lock contention with sessions waiting for locks. This is normally a problem only for busy transactional applications and not for well-designed OLAP systems. Each session requires its own thread or process and takes considerable DBMS and operating system resources, which may need to be configured. Creating and destroying sessions is expensive. DBMS that reuse sessions usually have better session scalability.

If you can't factor your database and need high scalability verify that your DBMS supports scalable platforms such as large multiprocessor servers and/or clustering or gridding. Oracle and DB2 have excellent scalability and they run on very fast platforms. MSSQL runs on midsize but not large multiprocessors. Thus MSSQL works for almost all databases, but not for very large un-factorable databases. MySQL is fast, free, and clusterable. MySQL clusters support very high throughput websites such as Yahoo. MySQL is maturing rapidly but it not yet suitable for very large un-factorable databases or very critical applications.

Test Yourself

What are some of the main considerations for scalability in database design? (Select all that apply.)

Number of people or applications concurrently accessing the database, particularly when this may increase over time.

This is true. This is session scalability.

The rate at which requests must be processed, particularly when this may increase over time.

This is true. This is throughput scalability.

Whether users access the database remotely.

This is false. Database scalability is not affected by the origin of the requests.

The SQL Design and Tuning Process

This section describes how to tune application SQL so that the DBMS can run it efficiently. SQL describes the data sought or the results of the operation requested, so it is driven mainly by the application needs. However, there may be different ways of describing what the application needs and some of these typically are faster for the DBMS to execute than others. Writers of scalable application SQL must therefore understand the schema and the *request patterns.* The application, schema, indexing, and request patterns are defined in concert. New designs may have bugs and some requests will be slow. You should perform timing tests and tune with realistic databases and workload.

Request Pattern

A *request pattern* is a particular pattern of SQL statements. In an OLTP database, requests with different tables in the table list or different tables or columns in WHERE clauses are normally considered different patterns, because they require different indexes. Different select lists do not make the patterns different. In a ROLAP schema, some patterns may be very general, because everything is indexed. For example, a constellation schema may efficiently support SELECTs where any one of a constellation of fact tables is joined against any number of dimension tables, with WHERE clauses that correspond to the attributes of the dimensions.

We will now describe how to tune each of the clauses of a SELECT statement. We choose a SELECT statement because there are typically many more SELECTs than other DML statements. We also choose SELECT because a subquery is often the performance-critical part of an UPDATE, DELETE or INSERT.

Test Yourself

Which of the following are true about the tuning process? (Check all that are true.)

Different columns in a WHERE clause can require different indexes.

This is true. Different columns in the WHERE clause could require different indexes.

In a ROLAP schema nothing is indexed.

This is false. In a ROLAP schema everything is indexed.

UPDATE, DELETE, and INSERT are more performance-critical than SELECT and finding the data because changes can take longer.

This is false for two reasons. While changes can take longer the most difficult operation are usually finding the data to change. The other reaon that this is false is because there are so many more SELECT statements and subqueries that SELECT and finding the data is more performance-critical.

SELECT and FROM Tuning

SELECT List Tuning	Most of the cost of a query is fetching the row; the DBMS doesn't need to do much additional work to return additional items from a row. Nonetheless you should avoid selecting things that you don't really need, because the extra columns in the result set will need to be converted to character strings and sent from the DBMS to the application. In fine-tuned applications it is best to avoid selecting literals, constants or other data that is known to the application. Eliminating unneeded items will not have a large effect on performance unless the result sets are large, but

	we can often obtain a few percent improvement by eliminating unnecessary items in the select list. Unnecessary items in the select list appear in the intermediate result sets, so they can sometimes force disk sorts and otherwise cause significant slowdowns. Beware of the overhead of DISTINCT, ORDER BY and other operations on the SELECT list. These force sorts to identify duplicates. In some DBMS, such as MySQL, you may need to examine the query plan to determine if such operations are taking place. For analysis and reporting applications it is often useful to compute the SQL at runtime to omit unneeded items from the SELECT list. This is called *dynamic SQL*. Dynamic SQL permits the application to determine at runtime exactly the items that are needed in the select list, usually based on the information that the user requested.
FROM List Tuning and Joins	Be careful when joining tables, because joining many tables can degrade performance dramatically. A single table in the FROM list is fastest. Joining a small table won't slow a SQL query very much, because small tables are usually mainly in the block buffer cache, so little I/O is required to access them. Dimensional data warehouse dimension tables are usually present in the block buffer cache—and that is one reason why well-designed dimensional data warehouses perform well. (Dimension tables are tables that represent times, locations, customers and similar discrete non-event entities in dimensional data warehouses and data marts.) The number of ordinary-sized tables that can be joined with reasonable efficiency depends on the DBMS, indexing, etc. It is hard to generalize, but Oracle performance usually falls off around eight tables, while MySQL performance usually falls off when more than about three larger tables are joined. Joining two or more large tables takes great care. Make sure there are *join indexes* covering the joined columns of both tables, particularly on DBMS such as Oracle that support index joins of the indexes instead of the tables. When joining large tables also check index sparsity and beware of large intermediates. Check the query plan, time realistically, and help the optimizer if necessary.

An Example

Suppose that you have a foreign key from an order number column of a line_item table referencing the primary key order_number column of the order table. If you are joining these two tables with WHERE line_item.order_number = order.order_number then you should create an index covering line_item.order_number. There is already the primary key index on order.order_number.

Test Yourself

Select some of the ways you can optimize SQL. (Choose all that apply.)

You can optimize SQL by using dynamic SQL to select only the needed fields at runtime.

This is true.

You can optimize SQL by joining fewer tables when that meets the application needs.

This is true. Sometimes an application doesn't need information that might be derived by joining additional tables.

Select all of the attributes so that all the data is retrieved in case you need it.

This is false. It is better for performance to only select the data that you need.

WHERE Clause Tuning

Tuning WHERE clauses can reap great performance improvements. One of the most important principles is to avoid functions in WHERE clauses such as `WHERE UPPER(name) = 'GEORGE'`. The problem with functions in WHERE clauses is that in most DBMS (except Oracle) you can't easily index the result of the function call, so the only way that the DBMS can test the WHERE clause is to scan every row of the table, run the function on each row, and then test the result. Table scans are very slow.

Function Calls

A function call is something like `WHERE UPPER(name) = 'GEORGE'`, where UPPER is a built-in function. IN and NOT IN are not function calls, just part of SQL syntax that the optimizer can handle.

Tips for using Function-Based Indexes

Oracle supports function-based indexes. A function-based index indexes the result of the function call. When you INSERT a row into a table with a function-based index, Oracle automatically calls the function and inserts the result of the function call in the index. A function-based index is better than the traditional approach of adding a column to the table to store the function result such as UPPER(name), and then indexing that new column. It's better because the function-based index stores the function result only in the index, while the traditional (and portable) approach stores the function result both in the table and the index.

Use equality tests in WHERE clauses when possible. Equality tests are usually a little faster with B-tree indexes and are the only type of tests supported by hash and bitmap indexes.

Test against literals in WHERE clauses when possible. For example, consider the WHERE clause `WHERE person.firstname = 'Sam'`. The string 'Sam' is a literal in this WHERE clause. The number 6 is a literal in

the clause `WHERE line_item.cost > 6`. A cost-based optimizer can use the sparsity of the literal (for example, the fraction of person.firstname which are 'Sam') to improve optimization.

Some DBMS, including MySQL, take the SQL order of the WHERE clauses as a hint for their execution order. This allows us to implicitly specify the order of the restrictions. This also means that for good performance we need to hand-tune every query that has more than one WHERE clause. With a cost-based optimizer the order of the WHERE clauses shouldn't matter, because the query optimizer will optimize the query based on the statistics of the database, including sparsity.

Functions in WHERE clauses

Avoid functions in WHERE clauses, such as:

```
WHERE LOWER(city) = 'boston'
WHERE SQRT(area) > 3.0
WHERE my_function(hour) = 5
```

There are many ways to eliminate functions, such as:

Store `city` in lower case and eliminate `LOWER`

Transform the function: `WHERE area > 9.0`

Add a column with the function result, index it, (or use a function-based index in Oracle), and query it:

```
WHERE my_function_col= 5
```

LIKE in WHERE clauses

Avoid LIKE clauses, particularly when the % is anywhere other than at the end, because these clauses can force index or table scans.

E.g., `WHERE name LIKE 'geo%'` may be supported by a B-tree index on name, but `WHERE name LIKE '%rge'` commonly results in an index or table scan.

LIKE is handy for exploring a database or prototyping, but a LIKE clause in a production system is usually not the most efficient design.

Test Yourself

Select all that are true about tuning WHERE clauses for improved performance.

Including a function in a WHERE clause usually requires the DBMS to scan every row of the table, applying the function to each row.

This is true. Including the function requires the DBMS to scan every row of the table, whether or not it will be returned as a result of the query.

A function-based index will be as efficient as storing the results of the function for each row in its own column, and then indexing that column.

This is false. A function-based index stores the result just in the index, instead of both in the table and in the index, so it takes up less space.

A WHERE clause can be optimized by using equality operators rather than inequality operators when that meets the application needs.

This is true.

ORDER BY and GROUP BY Clause Tuning

ORDER BY Clause Tuning	Don't use ORDER BY unless you really need to. The ORDER BY clause forces a sort of the result set, and sorts of larger result sets will slow performance, even if the sorts will fit in RAM. It doesn't help performance if you are only fetching the first n rows of the result set, because the whole result set must be sorted to determine which rows to fetch. Some DBMS do an implicit ORDER BY so that the result set order doesn't depend on the storage order. Suppress this sort for optimal performance. In MySQL you do this with `ORDER BY NULL`. If the ORDER BY clause columns are on one table you can also add a B-tree or other sorted index on the columns in the ORDER BY clause, keeping the order of the columns the same in the index and query. Not all DBMS are able to use an index to avoid the sort, so time the query with and without the index, and check the query plan.
GROUP BY Clause Tuning	GROUP BY is powerful, but it can also be slow if the GROUP BY operates on large intermediate result sets. If GROUP BY queries are slow, it is usually because there is a large intermediate result set that is being grouped. A GROUP BY clause may be computing aggregates that could be pre-computed and stored in an aggregate table. A good solution is to pre-compute the results of the GROUP BY and store them in a new aggregate table, which is queried without the GROUP BY clause. This is a common denormalization. There are costs to maintain the aggregates, including the additional complexity of the triggers or other mechanism and the additional storage and processing to maintain the aggregates. Materialized views are often a good design for the aggregates, because the DBMS maintains the materialized views when the underlying tables change. If the columns that appear in the GROUP BY clause are all in one table the grouping operation can be speeded if there is an index on the columns in the GROUP BY clause. Not all DBMS can use an index to speed grouping, so time the queries with and without the indexes, and check the query plan.

Test Yourself

In the following SQL statement, only the first five rows are sorted by deptno before the requested 5 rows are returned (True or false): [Rownum indicates the number of the row in the result set, so this query is requesting the first five rows.]

```
SELECT * from employee
WHERE rownum <=5
ORDER BY deptno;
```

False. All rows in the result set are sorted before the first 5 rows are returned. This is necessary to assure that the first five rows returned are the first five rows of the entire sorted result set.

Principles of Query Optimization

Request runtime depends primarily on the number of input/output operations that must be performed to execute the query plan. To estimate the number of required I/O operations you need to understand which tables and indexes will normally be memory resident, and which will be brought in by I/O operations from storage. This means that you can't search for anything in large tables, but must always use an index.

Returning lots of data from a query is expensive, involving DBMS server resources, network resources, and client resources. SQL can usually be designed to reduce data in the database, and return only the results. When SQL can't easily reduce the data within the database, stored procedures can always reduce it.

Test Yourself

True or False: Should DML operations against large tables be supported by indexes to obtain scalable performance?

This is true. The alternative to index support is scanning, which is very slow when tables are large.

Relational DBMS Server Tuning

In the previous section we described what you can do on the client side to tune the SQL that the application sends to the server. We now describe what you can do on the server side so that the SQL runs quickly, including tuning the hardware, database instance parameters, schema, and indexing.

Tuning DBMS Server Hardware

The following table summarizes the hardware requirements for a platform to host the DBMS.

Hardware	Requirement for good performance

CPU	Enough so that some CPU is normally available under load. Scalable DBMS can use many CPUs. Check the DBMS doc and configure the DBMS for the number of CPUs.
RAM	Sufficient so that the block buffer cache can be made large enough to obtain a good cache hit ratio. Allow plenty of RAM for the operating and file systems. Check your DBMS and OS docs. Memory-resident databases can benefit from a great deal of RAM. For servers with more than a few gigabytes of RAM or for critical servers it is prudent to use error-correcting RAM. Error-correcting RAM costs very little more and it is much more reliable.
Storage	Large enough to obtain the required database performance. Size storage to at least three times the DB size for backup, growth and administration. Check vendor documentation. Consider solid-state (flash) storage if storage performance is critical.
Network	As fast as feasible between clients and web server. Many gigabits may be needed between database server and application servers. Consider a Storage Area Network (SAN). Check vendor docs. Keep network traffic off the general LAN. Protect the DBMS with firewalls.

Database platforms are often configured and tuned by a systems administrator working with or supporting a DBA.

Test Yourself

Which of the following are true regarding the size of the storage for a database and the size of the database? (Check all that are true.)

The size of the storage must be at least as large as the size of the database.

This is true. To be durable at checkpoints the database must all fit on durable storage. In practice much more durable storage is required, for example to hold the transaction logs.

In practice the available storage should be several times larger than the size of the database, to support administrative operations such as copying the database.

This is true. A database should have storage about three times larger than the size of the data for backup, growth, and administration.

Server Storage

Other than schema and SQL design, storage has the greatest effect on DBMS performance. There are many kinds of storage and tuning depends on the kind that you have available. A slow hard disk will hurt database performance. How database and other storage is physically allocated to hard disks can have a large effect on performance. High-end storage, such as that from the EMC Symmetrix family or a Hitachi Data Systems high-end product, is generally the best, particularly for nonstop databases, but it is also the most expensive option.

Database backup media—The consequences of a failure to recover all of a database are usually much more severe than failure to recover all of the files in a file system, so it is critical that the database backup medium be very reliable. Tape is an exposed contacting magnetic medium prone to wear, loss of magnetic material, dirt, and other failures, so it is not very reliable. Backing up to tape is much slower than backing up to hard disk or RAID. When you backup a database to tape (for example, for an offsite or archival backup) it is a good practice to make two interchangeable copies of each tape and verify the copies by testing them. Even with care a backup to more than about ten tapes is likely to fail during restore. Backup large databases to fast reliable media such as RAID or SAN or NAS storage.

SAN means *Storage Area Network*

SAN is the standard very high speed fiber channel network used for database storage.

NAS means *Network Attached Storage*

This is fine for file systems, but because it is optimized for serial retrieval, it is not very good for databases.

Spread the storage load—Disk seeks take milliseconds. Many operations involve sequential fetches, so if the HD arm is left where it is, the next fetch will be significantly faster. Thus, if two or more tables, indexes, or other objects are mapped to the same hard drive and operations are being carried out simultaneously, the hard disk arm will spend a lot of time moving back and forth between the storage areas for these objects. For the same reason, try to avoid using the same drive for the swapping store and any database store. Similarly try to put the transaction logs and rollback segments on different drives than the database tables and indexes. If you have multiple RAID devices on a server it is also desirable to spread the load between the RAID controllers.

Put tables and their indexes on separate drives—Some operations such as index-aided queries of large tables involve I/O alternating between an index and a table. If the table and index are on the same hard drive, there will be much head motion. If the index and table are on separate drives, the heads can move much less, and the index-aided search will be much faster. This can make an order of magnitude difference in the performance of index-aided queries.

I/O system tuning — Use the operating system utilities to determine the number of I/O operations against each drive. Identify the database objects that are responsible for any I/O hot spots. Move the objects to spread the load and improve performance. Pay attention to the rules above about co-locating tables, indexes, and transaction logs.

Test Yourself

Which of the following correctly describe the relationship between hardware and DBMS performance? (Check all that are true.)

The CPU requirements of different DBMS are significantly different.

This is true. Some DBMS such as Oracle and DB2 are very CPU efficient while others such as Microsoft SQL Server can require considerably more CPU resources.

The amount of RAM available to the DBMS can have a major effect on database performance.

This is true. If a DBMS has more RAM then more of the data will fit in the block buffer and SQL caches and other caches such as the sort cache can be larger. All of these improve performance.

The performance of storage has a major effect on the performance of all DBMS except memory-resident DBMS.

This is true. Storage typically has the most effect on DBMS performance.

Network performance is typically a major consideration in scalable performance.

This is false. While network performance must be high enough, this is usually fairly easy to arrange, so network performance is rarely limiting or a major consideration.

Tuning and RAID Storage

Definition of RAID

RAID stands for Redundant Array of Independent (or Inexpensive) Disks. Disk drives today are almost always Winchester technology, which is the technology used in PCs.

Performance Note: The highest-performance hard disks are much faster than ordinary hard disks. For example, the 15K RPM server disks from Hewlett Packard have a 3.5 millisecond average seek time.

As discussed before, high-end storage systems, such as those from venders like EMC or Hitachi Data Systems, are generally the best, particularly for nonstop databases, but they are also the most expensive. Large cache RAID such as the EMC Clarion family is good for most large databases, though not ideal for large nonstop databases. Ordinary RAID is inexpensive, even with good caching controllers, and it works fine for most databases. Fast Hard Disk (e.g., 15K RPM SCSI) is fine for databases where HD fault tolerance is not needed. Ordinary HD (e.g., 7200 RPM synchronous ATA) is fine for most small non-critical databases. Older slow hard disks can reduce database performance substantially. If you need to run a database instance on a notebook computer consider the new 7200 RPM notebook disks, which are available as an option from leading vendors.

RAID storage—There are many different RAID configurations, with different cost and performance levels. Most RAID controllers can be configured for different applications. Almost all large storage is based on RAID,

sometimes with durable RAM and other supporting hardware to improve performance. Hard drives fail fairly often, so RAID configurations used for DB storage should tolerate HD failure. The following table summarizes the common RAID configurations.

Configuration	Characteristics
RAID 0	Uses striping to increase throughput, but has no fault tolerance, so it should not be used with databases where data loss risk is important.
RAID 0+1	Combines striping and mirroring for good throughput and fault tolerance.
RAID 1	Is mirroring of paired drives. It is fast and fault-tolerant, but requires twice the HD storage. Has about half the rotational latency of a single drive, because the result can be returned when the data comes under the head on the first drive. Software mirroring is much slower than a mirroring hardware controller, so it should not normally be used on databases where performance matters.
RAID 3	Uses a parity disk that stores the XOR of the other disks for fault tolerance. Not commonly used because of problems with the load on the parity disk.
RAID 5	Distributes the redundancy information over the drives and also blocks the data. This is the most common configuration for large RAID, because it is fault-tolerant and makes efficient use of the hard disks. RAID 5 requires a good caching controller to obtain good performance, because all but one drive must be read before returning the result.
RAID 6	RAID 5 plus additional error-correcting codes to protect against multiple disk failure.
RAID 10	Combines striping and mirroring for good throughput and fault tolerance. RAID 10 is common for small database storage where the efficiency of the hard disk usage is not important, because RAID 10 provides both good throughput and fault tolerance.

The performance of a RAID cache depends on how well it caches the data on the hard disks. The following table summarizes the common RAID controller cache types.

Cache type	Characteristics
Read Cache	Holds additional data beyond what was requested, e.g., the whole cylinder under the head; then if the additional data is requested it is immediately available from the cache. This is an example of pre-fetching.
Write-Back Cache	Durable RAM that supports immediate completion of write requests before data is written to HD.
A RAID 5	Gathers all the blocks in a stripe and computes the redundancy codes, reducing the reads

Write Gather Cache	and permitting all blocks to be written to HD in parallel.

RAID 5 spreads the data over many hard drives using redundancy codes that permit reconstruction of the data even if one HD is lost. This makes the most efficient use of HD resources. With RAID 5 the data must be fetched from all but one spindle before the controller can reconstruct and return the data, so the overall latency is about one full rotation plus arm motion latency.

RAID 1 can return the results as soon as any HD returns the results, so RAID 1 or 0+1 perform better for DB storage than RAID 5 when the RAID doesn't have cache. RAID 5 or 6 with a small or no controller cache is fine as a backup device. RAID 5 or 6 works fine for database storage when used with a good caching controller. The performance of the controllers, the amount of cache, and the price vary quite widely.

Test Yourself

Which RAID levels support drive failure? (Check all that are true.)

0

This is false. Raid 0 does not support redundancy.

1

This is true. Raid 1 supports failure of 1 drive in a set of two.

3

This is true. Raid 3 supports single drive failure.

5

This is true. Raid 5 supports failure of 1 drive in a minimum of 3 disks.

6

This is true. Raid 6 supports failure of 2 drives in a minimum of 4 disks.

10

This is true. Raid 10 supports failure of up to half the drives but only 1 from each group.

Indexes

An index is a durable auxiliary data structure associated with a table that is usually used to efficiently identify and access the rows of the table. Indexes have no role in the relational model or SQL; their only consequence is speeding requests. The exception is unique indexes, which also enforce uniqueness constraints. There are

many different uses for indexes, including primary and alternate keys, supporting indexes, join indexes, and indexes to support index joins. Different DBMS support different types of indexes. DBMS also differ in their support for different uses of indexes, such as using them for dimensional query plans, etc. The following table summarizes the common types of indexes.

Index Type	Properties
B-tree	Flexible. Supports equality, inequality, range searches, and index joins. Supported by all common SQL DBMS.
Bitmap	Very fast for binary joins with other bitmap indexes. Only useful for low cardinality columns. Very compact, because there is no rowID in the index. Does not support theta queries. Supported in Oracle.
Hash	Fast. Access time does not increase as the size of the index increases. Does not support theta queries. Supported in Oracle.
Function-Based Indexes	Supports function calls in WHERE clauses efficiently. Compact, because the function result is not stored in the table, but only in the index. Useful when most queries use the same function(s). Supported in Oracle.
Index Organized Tables	B-tree without the table, with all columns in the index. Eliminates the I/O to fetch the row. Supported in Oracle.

Test Yourself

What is the most compact index for a column that only has three possible values?

B-Tree

This is false. While this may be more efficient for some operations such as index joins with other B-tree indexes, the B-tree index requires a ROWID for each entry plus other overhead, so it is much larger than a bitmap index.

Bitmap

This is true. Bitmap uses bit arrays and works well for low-cardinality columns.

Hash

This is false. While a hash index would be the fastest and most scalable for equality keys, a hash index requires a ROWID for each entry, so it will be much larger than a bitmap index.

B-tree and Bitmap Indexes

B-tree indexes are the most common. B-tree indexes support equality, inequality, range, partial-key, index scan, fast index sorting, and query resolution from the index. B-tree indexes are self-balancing trees. B-tree indexes on multiple columns support queries on any prefix of the columns; for example, an index on {A,B,C} supports queries that need indexes on {A}, {A,B} or {A,B,C}. The technical way of saying this is that B-tree indexes support any prefix of the columns in the index. The most common type of B-tree index is the B+ tree index.

Sometimes you need more than one index on the same column. For example, if you need to support frequent queries that need coverage of {A}, {B}, and {A,B}, then you need at least two indexes:

{A,B} and {B}
-or-
{B,A} and {A}

Index Column Order Matters

The order of the columns matters in B-tree and other sorted indexes, because the order of the columns in the index is the order of the sort keys. The order of the WHERE clauses should not matter, but it can make a difference with DBMS such as MySQL that don't have a cost-based optimizer.

Details about Index Scans

An index scan involves more CPU overhead than a table scan. An index scan is generally faster when the amount of data (in bytes) in the indexed columns is much less than the length of the row. The reason is that bringing in the index from storage is faster than bringing in the rows. If only a small fraction of the rows are fetched from the table, then it is faster to scan the index than to scan the table. It can happen that all required data is in the index, in which case the index scan can be significantly faster than the table scan.

The optimizer can transpose the order of the WHERE clauses, so coverage of {A,B} is the same as coverage of {B,A}, though an index on {A,B} covers {A}, but not {B}. Sometimes a DBMS will do an index scan when the index is faster to scan than the table, so an index may speed a query that it doesn't actually cover.

Bitmap indexes are implemented with bit vectors with one bit for each row in a table. There is one bit vector for each value of the indexed column, so bitmap indexes are useful for indexing low-cardinality columns such as gender, location, and year. The bit in the bit vector is one if the column for that bitmap index has the value associated with that bitmap. Because there is one bit position for each row in the table, the row locations can be computed from the bit number, and no row pointers are required in the index. If the cardinality is low this makes bitmap indexes smaller than other index types such as B-trees and hash indexes. Bitmap indexes must be

updated if rows are moved on disk. Sparse bit vectors can be compressed. Bitmap indexes do not support inequalities or ranges, so they are not useful for queries containing WHERE clauses such as WHERE cost > 10.

Bitmap Indexes

Bitmap indexes are very efficient for WHERE clauses that combine AND, OR, NOT and other Boolean or set operators. They can be combined very efficiently by applying the corresponding operations to the bit strings. For example, bitmap indexes would help:

```
SELECT * FROM customer
WHERE gender='F'
AND (last_degree = 'MS') OR (last_degree = 'PhD')
AND marital_status = 'M';
```

Note that `gender`, `last_degree`, and `marital_status` all have low cardinality.

Test Yourself

Select all that are true about B-tree and bitmap indexes.

A bitmap index would work well for an attribute like Social Security number in a customer table, because all values are distinct.

This is false. Bitmap indexes work best with attributes that have low cardinality, or a few possible values. Social Security number would have an almost infinite number of values, so it would not be a good choice for a bitmap index.

Bitmap indexes do not require as much storage space as B-tree indexes.

This is true. Bitmap indexes are the most compact of all external index types. The only more compact type is the clustered index, which is sorting by the index key.

In a table with a B-tree index which indexes the first two fields, an entry of {one,two} will also find rows that begin with {two, one}.

This is false. Order matters, so for the row starting with {two,one} will only be located if the index entry is {two, one}.

Hash Indexes

Hash indexes use a hash function over the indexed columns to identify and retrieve a bucket. The bucket is scanned for the key. If the key is not present the row with the key is not present. If the key is present the entry will be in the page, or potentially in an overflow page. Hash indexing can require about one I/O operation independent of the size of the table or key. Hash indexes do not support inequalities or range searches.

Dynamic hash indexing supports tables whose size is unknown or varying over time. In dynamic hashing the size of the space hashed into varies as the size of the table varies. It depends on sequences of hash functions with varying bucket counts. The data migrates to higher order members of the family when overflow chains grow and it migrates to lower order members of the family when there are too many empty buckets.

Test Yourself

True or False: An appropriate hash index would greatly improve the scalable performance of a query that retrieves all of the customers in a large Customer table that have a DateOfBirth before January 1, 2000.

This is false. Hash indexes do not support range or inequality (theta) queries. If the Customer table has long rows then an index scan of the hash index may be faster than a table scan of Customer, but this is not a scalable design.

The Roles of Indexes

Primary Key and Support indexes—The DBMS provides primary key indexes, which are commonly used in WHERE clauses. *Support indexes* are used to speed the restrictions of frequently run queries when the WHERE clauses are against columns other than the PK. For example:

```
WHERE customer.SSN = 123456789
```

The SSN is not likely the primary key of customer, because not all customers will have social security numbers. Yet queries similar to the above may run frequently, so they need to be efficient. The solution is to add a support index to customer:

```
Create index customer_ssn_sx on customer(SSN);
```

The suffix ***_sx*** is a conventional way to identify support indexes.

Selectivity and sparsity—If the values in an index help narrow the number of rows, then the index is more selective and useful to the optimizer. For example, an index on SSN is more useful than an index on gender. Primary key and unique indexes are extreme cases of sparsity, because each value can occur only once.

Indexes to support joins—If two columns are frequently joined with a WHERE clause, then it is usually desirable to index both of the joined columns. When the referencing columns refer to a foreign key or unique column, then the PK or unique columns of the referenced table are already indexed, so we don't need to index them again. We do need to explicitly create the indexes on the *referencing* columns of foreign keys that are used in joins.

Index joins—When both column sets involved in a join are indexed, the DBMS can do an *index join* to identify the rows that satisfy the WHERE clause without even using the tables. Index joins can be much faster than un-indexed joins, because the indexes are usually much smaller than the tables, so less I/O is required. In sorted

indexes such as B+ tree indexes, the traversal with comparison is fast. With B+tree indexes an index join can be done for inequalities (theta joins) as well as for equijoins.

Indexes to support ORDER BY clauses—A B+ tree or other sorted index on the columns in an ORDER BY clause eliminates the need to sort the rows; they can just be fetched in order, based on the index. Thus a B+ tree index on the columns in an ORDER BY clause can greatly speed queries that return large result sets that must be ordered. The columns in the ORDER BY clause must be on the same table in order to use indexes to support ORDER BY clauses. Time your queries with and without the indexes and check the query plan to determine whether your DBMS can use indexes to speed ORDER BY clauses.

Test Yourself

Select all that are true about indexing.

It is useful to add an index on the primary key.

This is false. The DBMS provides an index on the primary key, so an additional index is not necessary.

It is useful to add indexes on attributes that are frequently used together in a join statement.

This is true.

It can be useful to add an index on a foreign key.

This is true. Most DBMS do not automatically add an index on a foreign key. It can be useful to add an index if the attribute is used frequently in join statements.

Tuning Relational DBMS Initialization Parameters

DBMS have 100 or more initialization parameters that adjust the sizes of internal buffers, caches, and establish various characteristics of the operation of the instance. The specifics of these tuning parameters depend on the DBMS; since there are no standards for database instance parameters, you should study the documentation for your particular DBMS to determine what these parameters are and how to tune them to obtain the best performance. The following steps describe the general tuning process.

1. **Start with the appropriate size configuration (typically named small, medium, large or huge)** from the DBMS vendor and the type of database (OLTP, general purpose or data warehouse). Check that your hardware will support it. If it will not, downsize the configuration or upsize the hardware. Configure storage. At this time you should adjust any particular configuration parameters that you know should have different values than their defaults. Instructions for setting these parameters are included with the DBMS configuration documentation or as comments in the default configuration files.
2. **Bring up the database instance** and load or migrate the current production database if you have it.
3. **Load test** by running load test scripts consisting of realistic application SQL streams.

- Check key DBMS performance parameters such as the cache hit ratios. If any are out of line correct them as suggested in the DBMS tuning documentation. You may need to tune storage.
- Adjust parameters to improve performance. Repeat until you are satisfied with the DBMS performance. Understand the effect of each parameter and don't try to tune all the parameters using blind search.
- Check the SQL logs produced by the DBMS and look for SQLs that take longer than you expect. Look for poorly designed (*rogue*) requests and correct them. At this time you can also fix any remaining problems with indexes.

4. **Proceed to full-scale test and production**. Continue with the tuning operations in step 3, monitoring the logs, SQL performance, and storage. Check on a regular basis (perhaps daily or weekly) and whenever there are performance issues or significant changes. You may wish to use a DBMS monitoring tool to reduce the effort required and more quickly identify problems.

General performance tuning rules—As a general rule of thumb, on a dedicated database server you should size the shared global memory to about half the physical RAM on the server. Watch the block buffer cache hit ratio and increase the block buffer cache size limit if the block buffer cache hit ratio is under about 90% during routine ongoing operations. Watch the SQL cache replacement and hit statistics and increase the SQL cache size if there is much replacement in the cache, the cache hit ratios are low, or analysis of the SQL logs indicates that there are repeated SQLs. Other cache and RAM setting heuristics are DBMS-dependent. In DBMS such as Oracle 10g and MSSQL, the system will automatically adjust the RAM allocations within configuration limits.

Tuning the database blocksize—Mainstream DBMS such as Oracle, MSSQL, and DB2 support variable blocksize; the database blocksize should be a multiple of the virtual memory page size and the storage block size; powers of two multiples of 1024 bytes (1 kilobyte) are the rule, e.g. 2KB, 4KB, 8KB, 16KB, 32KB, or 64KB. The blocksize for a table should be at least four times the average row length for the table, to minimize fragmentation and block chaining. The optimal blocksize for an OLTP database without very long rows is usually 2KB to 16KB. The optimal blocksize for a warehouse is typically 32KB to 64KB; the larger blocksize helps reduce the number of I/O operations required to bring in large tables or indexes. There are variations in blocksize configuration flexibility. Oracle supports a different blocksize for each tablespace. To change the blocksize of MySQL you need to recompile the DBMS from source.

Test Yourself

What should the blocksize be for a table with an average row length of 16KB?

4KB

This is false. The blocksize should be 4 times the average row length.

16KB

This is false. The blocksize should be 4 times the average row length.

32KB

This is false. The blocksize should usually be 4 times the average row length.

64KB

This is true. The blocksize should almost always be at least 4 times the average row length, to minimize block chaining.

Indexing for Performance

I use a three-phase approach for designing indexes. I begin by identifying the application request patterns. A request pattern is like a template for the kinds of SQL statements that you expect the database to efficiently support. For example, we may know that sales requests will equijoin the line_item table with the sales_order table and customer table, so the request pattern will be:

```
SELECT <select list>
FROM sales_order
INNER JOIN  line_item ON
line_item.sales_order_id = sales_order.sales_order_id
INNER JOIN customer ON
sales_order.customer_id = customer.customer_id
```

Notice that we don't care what items are selected from these three tables, but only that something is needed from each table.

The indexing for queries with this pattern would include a supporting index with coverage of line_item.sales_order_id and an index with coverage of sales_order.customer_id. It would be normal that sales_order_id is the synthetic primary key of the sales_order table. Primary keys are implemented as unique indexes, so there is no need to index sales_order.sales_order_id. Similarly it would be normal that customer_id is the synthetic primary key of the customer table, so no additional index is required.

Suppose that there is an additional request pattern:

```
SELECT <select list>
FROM line_item, customer, product
INNER JOIN sales_order ON
line_item.sales_order_id = sales_order.sales_order_id
INNER JOIN customer ON
sales_order.customer_id = customer.customer_id
INNER JOIN product ON
line_item.product_id = product.product_id;
```

Requests with this pattern would require the two indexes defined above, plus an additional index covering line_item.product_id. We now have a need for index coverage on two columns (sales_order_id and product_id) of the line_item table. We could create two separate indexes on the sales_order_id and product_id tables or create a two-column composite index on (sales_order_id, product_id). The choice between these options depends on the particular mix of queries.

After designing the indexes to cover the expected request patterns, I run the applications or SQL stream against the indexed database, and collect request timing data. I analyze slow SQLs to understand why they are slow. Sometimes the problem is just a request pattern that you didn't expect; this can usually be corrected by adding an index or adding a column to an existing index. Occasionally the slowness can't be corrected by indexing, so changes to the requests or schema are required.

Only create the indexes required to efficiently execute the intended plans. Do not create unneeded indexes, because unused indexes just increase database size and slow update, insert, and delete performance.

Choose the appropriate kinds of indexes. Particularly for large databases and DBMS such as Oracle that support more advanced index types the selection of the appropriate index type can make up to an order of magnitude difference in query performance.

Test each request pattern against production databases and verify that the query plans are as intended and that they are fast. If the optimizer produces a plan different than the one you intended, then understand the plan the optimizer produced. Sometimes good optimizers find faster plans than the ones we intended to use. Sometimes a surprising plan means that we forgot an index. If the optimizer's plan is not fast enough and the optimizer is not very good, restructure the SQL or use hints or other techniques to guide the optimizer to a faster plan. Tune based on realistic SQLs against realistic databases. Request performance depends on the database statistics.

Sometimes you may encounter slow requests that cannot be made fast enough by indexing or changing the requests, and this slowness is due to scalable performance problems in the schema. No amount of tuning or indexing will make a poorly designed schema perform well. Incompatible schema changes "break" existing SQL, so it is important to get a good stable scalable schema early in development. Mature databases such as Oracle have many features that DBAs use to adapt the DB to changing SQL or workloads without changing the schema in ways that break existing application's SQL.

Test Yourself

What is the most efficient index for the query: `SELECT * FROM Assignment INNER JOIN Student ON Student.student_id = Assignment.student_id`? Assume that there is a B-tree primary key index on Student.student_id.

A B-tree index on Assignment.student_id

This is true. This index would support an index join between the primary key index of Student and Assignment.student_id.

A bitmap index on Student.student_id

This is false. There is already an index on this and most DBMS would not even allow a second index.

Tuning that Affects both the SQL and DBMS

Some of the most effective techniques for improving performance involve changing both the application and the DBMS. Because these changes alter the application's SQL and the DBMS they are not as easy to implement as many of the previously discussed techniques.

One of the most basic techniques is to minimize the lengths in bytes of the rows in the database tables. You do this by eliminating unneeded columns, by choosing the most compact data types for the columns, and by eliminating unneeded data such as data values that represent null. (Modern DBMS such as Oracle use special algorithms to compress null values, so they store much more compactly than application-specified null values.) Operations against tables with shorter rows run faster, because more rows are brought in with each I/O operation and because the block buffer cache is better utilized. I have obtained a thirty percent performance improvement by carefully optimizing the data types in large tables.

Another less obvious technique is to vertically partition tables that have very different access patterns for different columns, and very large columns. You do this by moving infrequently accessed large columns to another table. When you do this the frequently accessed columns will be in much shorter rows, so they will run much faster and cause less traffic in the block buffer cache. Using this technique I obtained a five-fold speedup of the frequent operations against a large telephone company warehouse table that contained one column that was most of the size of the table. What I did was to create a new table that shared the synthetic primary key of the previous table, and move this large VARCHAR column to the new table. The few queries that accessed the large VARCHAR column were rewritten to join the shorter table and the new longer-row table; these queries ran no more slowly than previously, because most of the time was spent accessing the long rows of the new table. Fortunately queries that accessed the large VARCHAR column were infrequent, so average performance was improved dramatically with no slowing of the worst case.

Advanced Topic

You may notice that this is the opposite of what is done when tables that are frequently accessed together are physically clustered using a common cluster key; this reduces the number of I/O operations required to join the clustered tables. In this case what we are doing is physically separating the storage for parts of a table that are infrequently accessed together, rather than physically clustering the parts of different tables that are frequently accessed together.

*See Loney, *Oracle Database 12c: The Complete Reference*, Chapter 17, pages 338–340 for details on clustered tables.

Perhaps the most powerful performance optimizing technique that changes both the SQL and the database is migrating data-intensive computations from the application to the database itself as stored procedures and triggers. We studied this in Week 1. Moving computations into stored procedures has a number of advantages, including improving security, reducing network traffic, and providing a more stable application interface in spite of schema changes. Stored procedures have direct access to the data in RAM, so they are faster than external computations for data-intensive operations. This is a common approach for larger enterprise systems. Some databases are designed so that applications can only access the database using stored procedures. Stored procedures and triggers are not as portable as ANSI SQL 92, but vendors are slowly catching up with SQL 99, which includes stored procedure and trigger standards.

Performance Note

Most DBMS such as Oracle support executing Java in a Java Virtual Machine in the database kernel or linking application C code into the stored procedures kernel. Because Java is compiled on the fly and C is precompiled, this is the most CPU-efficient approach for computation-intensive operations. It is also the most difficult to achieve, because the native data types may not correspond directly to the data types in C or Java, making data conversion necessary. This can slow performance, so there is a design tradeoff when using Java or C.

Another basic general technique for improving performance is to constrain the domain (the domain of a variable is the set of legal values that the variable can take on) of the data at the time that you insert or update it rather than doing operations on the data to convert it to some standard form at query time. For example, if text is stored with uncontrolled casing, then querying it requires converting it to standard casing, using something like:

```
WHERE UPPER(name) = 'FRED'
```

The result UPPER function call isn't indexable, except with an Oracle function-based index, so it forces a scan, and it is much slower than:

```
WHERE name = 'FRED'
```

Consider CHECK constraints or triggers to enforce data domain constraints such as casing.

Even in OLTP databases there are usually many more queries than inserts or other change requests, so take time during changes to help the queries.

Test Yourself

What are some of the ways to optimize DBMS performance? (Select all that apply.)

Performance can be optimized by eliminating data values that represent null.

This is true.

Performance can be optimized by minimizing the length of bytes in each row.

This is true.

Performance can be optimized by moving infrequently accessed columns to separate tables.

This is true.

Performance can be optimized by moving computations to stored procedures and triggers.

This is true.

Summary

Design relational database applications for scalable performance. Scalable performance requires coordinated design of hardware, DBMS, DBMS configuration, schema, SQL, indexing and applications software architectures. Scalable performance requires coordinated tuning of both the SQL and the DB server with realistic databases and load. SQL changes may require DB server changes such as adding an index. With scalable performance best practices you will obtain good scalable performance and minimize the life cycle costs to maintain good performance, because any production performance tuning will be local and not disruptive.

Lecture 2B - Database Security

Overview

This lecture introduces Database Security. Database security is a large topic; it is covered in detail in the course MET CS 674, *Database Security*. We begin this lecture by introducing the basic concepts of database security and then introduce the different kinds of threats, both internal and external. We then introduce the countermeasures and practices that can be used to help protect against different threats. We introduce the different security options and features available in Oracle. Finally, we introduce the special database security considerations for applications accessible over the web.

Database Security is introduced in Chapter 20 of the sixth edition of Connolly and Begg. You are responsible for the material in this lecture and in Connolly and Begg. You are not responsible for techniques specific to Oracle or any other DBMS, but you may find some of these useful for your term project.

Learning Objectives

By completing the readings, participating in discussions, and completing the assignments, you will be able to:

- Explain the basics of database security
- Explain why database security is important
- Protect a database from internal and external threats
- Describe authentication and authorization
- Understand security measures used in Oracle
- Describe DBMS and Web Security

Portions of this lecture were prepared by Matthew Harris, a MET MSCIS graduate and DBA.