Module 5

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 5 Study Guide and Deliverables

Background Concepts Readings:	• Coronel & Morris, chapter 10 and chapter 11 (only sections 11.1 to 11.7)
Optional SQL Readings:	• *12th Edition:* Coronel & Morris, section 8.2 of chapter 8 • *13th Edition:* Coronel & Morris, section 7.8 of chapter 7
Assignments:	• Term Project Iteration 5 due **Tuesday, February 21 at 6:00 AM ET** • Lab 5 due **Tuesday, February 21 at 6:00 AM ET**
Live Classroom:	• **Tuesday, February 14 from 8:00-9:30 PM ET** • **Wednesday, February 15 from 8:00-9:30 PM ET**

Which First?

Read the book chapters before reading the online lectures.

Lecture 13 - Database Performance Tuning and Query Optimization

Introduction

metcs669_module11 video cannot be displayed here

It is fairly easy to obtain good performance from a small database such as we use in this class, but it is difficult to obtain good performance from large databases with high transaction rates. Because small successful databases often grow and attract more users, it is important to design our databases and SQL so that they perform well.

We begin this lecture by introducing the basic relational database performance tuning concepts. We then move to tuning SQL statements and how to use indexes to improve performance. Finally we introduce tuning techniques that affect both the SQL and the schema, such as stored procedures; these techniques are not described in the text.

Database performance tuning and query optimization are covered in more detail in MET CS779 Advanced Database Management.

Basic Relational Database Performance Tuning Concepts

There are three areas where we work to tune our SQL database applications:

1. **SQL tuning.** We tune the requests that the application sends to the DBMS.
2. **DBMS tuning.** We tune the DBMS so that it executes the requests as fast as possible. The only aspect of this complex area which we cover here is basic indexing.
3. **Combined tuning.** We make changes that affect both the requests and the schema. This includes migrating data intensive operations into the DBMS. This kind of tuning is not covered in the text.

How fast is fast enough? Sometimes it is obvious that our RDBMS application is too slow, such as when our users complain that the application is slow, or the system can't keep up with the transactions. Sometimes we also need a good way to know when we have made it fast enough before our users

complain or have their performance degraded by a slow application. Fortunately there is a simple basis for deciding, based on human performance criteria. When a user waits longer than about three seconds for the initial response to a request, the user's attention shifts to something else. After the response the user's attention then needs to return to the application. This attention shift away and then back takes about ten seconds, so if the computer doesn't get back to the user with something interesting within about three seconds, it wastes ten seconds or more of the user's time. The conclusion is that our applications should get back to the user as fast as feasible, and within three seconds whenever possible.

Test Yourself 5.1

Select all that are true of database performance tuning. (Choose all that apply.)

Adding indexes to the database can improve performance.

This is true. 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.

The larger the database the easier it is to maintain.

This is false. A small database is easier to manage than a large one.

Altering SQL statements can improve performance.

This is true. SQL statements can often be changed to improve the performance of the request.

Tuning the database instance can improve performance.

This is true. Monitoring and tuning the database instance is an ongoing responsibility of the database administrator, and this monitoring and tuning may need to be performed frequently, such as daily.

The Block Buffer Cache and Performance

The *block buffer cache* is the area in shared random access memory (RAM) containing database data, upon which all database operations are performed. With modern processor and disk speeds operations on data in the block buffer cache are at least 100 times faster than operations requiring I/O to disk, so that 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. SQL requests that can be processed using data in the cache (cache hits) are much faster than SQL requests that require I/O (cache misses).

Scalable performance

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 application SQL. This can be very expensive, so it is important that database designs be scalable, particularly the database schema.

Scalable designs are those where the DBMS does not operate on more data for each SQL as the database and load scale. Scans of tables or indexes are not scalable, because the time required to scan a table or index is proportional to the size of the table or index. 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. An intermediate result set which is too large to fit in the RAM allocated for it will need to be processed on storage, which is very slow compared to having the same operation performed in 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. When a database become very large, avoiding large intermediates and disk sorts requires careful schema and query design, careful indexing, and adequate RAM.

Size, throughput, and session scalability

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

1. **The DB grows.** This is *size scalability*. Software factors affecting size scalability include the application SQL, schema, and indexing. Hardware factors include I/O and storage type and configuration, and the speed and number of processors.
2. **The transaction rate increases.** This is *throughput scalability*. Throughput scalability is primarily determined by lock contention, DB server concurrency, and platform performance.
3. **The number of simultaneous sessions increases.** This is user or *session scalability*. As the number of concurrent sessions increases there are more sessions accessing the same data, and lock contention increases. Session scalability is driven by schema design, DBMS concurrency support, and lock granularity.

Test Yourself 5.2

Which of the following are true regarding database performance? (Please check all of the following that are true.)

A SQL request that requires data that is only stored on disk takes much longer to process than a SQL request that only requires data stored in shared random access memory.

This is true. The block buffer cache is the area in shared random access memory (RAM) containing database data, upon which all database operations are performed. It takes thousands of times

---

longer to retrieve data from the hard disk than when the data is in the block buffer cache.

Scans of tables or indexes are not scalable.

This is true. The time required to scan a table or index is proportional to the size of the table or index. Operations against tables that may grow in size must use indexes or other means to limit the number of rows accessed.

Session scalability is a measure of how well the database performs as the transaction rate increases.

This is false. This is a description of throughput scalability. Session scalability is a measure of how well the database performs as the number of concurrent sessions increases.

Size scalability is a measure of how well the database performs as the database grows, as measured by the storage required.

This is true. Size scalability, throughput scalability, and session scalability are the three main dimensions of database scalability.

Lock granularity can influence database performance and scalability.

This is true. For example, suppose that a transaction that needs to update one row of a table and it needs to acquire a block level lock on that table. This could prevent other transactions from accessing different rows of the table and result in reduced concurrency and database performance. Lock granularity is one factor that drives session scalability.

SQL Tuning

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 they are driven mainly by the application needs. However, there may be many different ways of describing what the application needs, and some of these typically are much faster for the DBMS to execute than others.

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 focus on SELECT because finding the data or a subquery is often the portion of an UPDATE, DELETE or INSERT which dominates the performance. For example, the most expensive part of most UPDATE operations is finding the rows to update.

Execution Plans

Because SQL is a declarative language in which we describe the logic for the results we want, without describing the exact steps for how to complete the logic, the DBMS must decide on its own precisely what steps it will take to obtain the requested results. Therefore the DBMS always creates an *execution plan* for every query, which is an outline of every step it will execute to obtain the results. Each DBMS provides us with a way to view the execution plan, and graphical SQL clients usually provide us with enhanced views of the plan. Database developers regularly refer to the execution plan of a query when they are trying to improve its performance.

When we give a valid query to a DBMS, we have described the precise results we want to see by specifying specific durable objects and operations on those objects. The DBMS must *guarantee* that the results it produces precisely matches the query's specification. However, it is very important that you understand that there are many execution plans that would retrieve the correct results, and the database *does not* guarantee that it will use any particular plan. The DBMS does its best to choose the execution plan that results in the best performance.

One common point of confusion for students and database developers alike, is that oftentimes database texts describe the logic of a query in terms of how the database will carry it out, but in truth these explanations are subtly incorrect because there are many ways the database can carry it out. For example, one could describe this query:

```
SELECT  Name.first_name, Name.last_name
FROM    Customer
JOIN    Name ON Customer.name_id = Name.name_id
WHERE   Customer.customer_id = 7;
```

by stating that the DBMS will create the Cartesian product of all possible row combinations between Customer and Name, then identify all rows where the name_id values are equal, then further identify all rows where the customer_id is 7, then extract the first_name and last_name from those rows. Unfortunately, this statement is subtly incorrect, because it does not distinguish between the *logical,* or *theoretical*, definition of the query, and the *physical* execution of the query. And in fact, most DBMS will not execute the steps in this order because of the poor performance that would result.

The logical definition of a query rests in the definitions found in relational algebra. To describe the logical definition of the query, we need to refer back to the relational algebraic definitions of each statement. The JOIN statement does mandate a PRODUCT operation between Customer and name to derive a new relation, followed by a SELECT operation to create yet another new relation by retrieving the specific rows where the name_id values are equal. The WHERE statement mandates an additional SELECT operation which creates a new relation where the customer_id value is 7. The column selection specification mandates a PROJECT operation whereby a new relation is created with only the first_name and last_name columns present. Thus, logically we have correctly described what this query means.

However, it is a fallacy to directly translate this logical definition of the query into steps that the database must or will execute. One reason is that relational algebra does not take into account real-world storage and memory characteristics and constraints. Another reason is that the database can retrieve the correct results in many different ways. So it is most important that you understand:

- There is a distinction between the logical definition of a query and the physical execution plan used to retrieve its results.
- The database does not guarantee any particular execution plan but will do its best to choose the plan that will perform the best.
- You can view the chosen execution plan to help you tweak your query or add indexes to improve a query's performance.

A Large Table Join 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. Then 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.

Tuning joins

Joining many tables can degrade performance dramatically. Joining a small table won't slow a SQL very much, because a small table is usually mainly in RAM, so little I/O is required. Joining two or more large tables takes great care. Make sure there are indexes covering the joined columns of both tables, particularly on DBMS such as Oracle that support index joins

WHERE clause tuning

Tuning WHERE clauses can reap great performance improvements, often with little effort. One of the most important principles is to avoid functions in WHERE clauses. The problem with functions in where clauses is that you can't easily index the result of the function call, so that 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.

Example

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, but just part of SQL syntax that the optimizer can handle.

Use equality tests in WHERE clauses when possible. Equality tests are usually a little faster than inequality tests.

Test against literals in WHERE clauses when possible.

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 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 with a DBMS such as MySQL we need to hand tune every query that has more than one WHERE clause. With a modern DBMS such as Oracle with a cost-based optimizer the order of the WHERE clauses doesn't matter, because the query optimizer will optimize the query based on the statistics of the database.

ORDER BY clause tuning

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, so don't use ORDER BY unless you need a sorted result set.

GROUP BY clause tuning

GROUP BY is powerful, but it can also be slow if the GROUP BY operates on large intermediate result sets. This is one of the situations where denormalization can improve performance. You can precomputed aggregates that are computed by the GROUP BY clause and store them as part of the database, often in a different table. This is a common denormalization. You need to weigh the performance improvement provided by the denormalization against the cost to maintain the denormalized aggregates.

Test Yourself 5.3

Which of the following are true regarding SQL tuning? (Please check all of the following that are true.)

When joining large tables, a full table scan may be avoided by adding an index that covers one column.

This is true. Indexing the column of a table that is used in joining that table can avoid a full table scan. If joining based on the primary key of one table and a foreign key of another, the primary key already has an index. The presence of an index covering the foreign key can make a significant performance difference.

There is an index covering the last name column of a customer table. There are no other indexes associated with this table except the primary key. The SQL statement "SELECT * FROM Customer WHERE UPPER(cust_lname) = 'SMITH';" may take significantly longer to provide a user with results than the SQL statement "SELECT * FROM Customer WHERE cust_lname IN ('SMITH', 'Smith', 'smith');"

This is true. The first SQL statement has a function in the where clause. The function will be applied to the value (cust_lname) in each row of the table. The second SQL statement is checking an indexed column for literal values. With a large table this can make a significant difference.

Sorts of larger result sets will slow performance.

This is true. This is true even if the sorts will fit in RAM. Use of the ORDER BY clause causes a sort of the result set.

The SQL statement "SELECT * FROM Pet WHERE pet_legs > 6;" will probably be a little faster than "SELECT * FROM Pet WHERE pet_legs = 8;"

This is false. Equality tests are usually a little faster than inequality tests such as greater than.

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 a table. Indexes have no role in the relational model or SQL; their only consequence is speeding requests. Unique indexes also enforce uniqueness constraints. There are many different uses for indexes, including primary and alternate keys, support indexes, and indexes that make index joins possible.

Different DBMS support different types of indexes. The following table summarizes the common types of indexes. You are not responsible for understanding these index types; the list is just here so that you are aware that there are different types. We will study bitmap, hash, and function-based indexes and index-organized tables in CS779 Advanced Database Management.

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. 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. 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 table. Eliminates the I/O to fetch the row. Index organized tables do not support other indexes besides primary key, because there is no rowID. Supported in Oracle.

B-tree 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}.

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}

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}

> The order of the columns in the index matters, 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.

Support indexes

The DBMS provides primary key (PK) 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 may 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 on the SSN column of the 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, because there are millions of different SSN values, but usually only two genders. Primary key and unique indexes are the extreme case 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.

Test Yourself 5.4

Which of the following are true regarding indexes? (Please check all of the following that are true.)

A customer support center gets frequent calls from customers that do not know there account or customer number. When this happens, the customer accounts are found with queries of a customer table using the customer's phone number. An index added to the phone number column of the customer table to speed this activity is an example of a support index.

This is true. Support indexes are used to speed the restrictions of frequently run queries when the WHERE clauses are against columns other than the primary key.

There is a column in a customer table with two possible values ('Y' or 'N') indicating whether the customer wants information about new products sent to them. An index on this column would be more selective than an index on the customer phone number column.

This is false. If the values in an index help narrow the number of rows, then the index is more selective and useful to the optimizer. Generally, most of the values in a phone number column of a customer table would be different. With a two value column, many rows can have the same value. This would make a phone number column better at narrowing the number of rows.

Tuning That Affects Both the SQL and DBMS

Some common techniques for improving performance involve changing both the application SQL and the DBMS. Because these changes alter the applications SQL and the DBMS they are not as easy to implement as many of the previous techniques.

> The row length is usually measured in bytes.

One of the most basic techniques is to minimize the lengths of the rows in the database tables. You do this by eliminating unneeded columns and by choosing the most compact data types for the columns and by eliminating unneeded data such as data values that represent null. (Modern DBMS 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 multi-terabyte tables.

Perhaps the most powerful performance optimizing technique that changes both the SQL and database is migrating data intensive computations from the application to the database itself as stored procedures and triggers. (We studied stored procedures and triggers in Week 4, and RobCor Chapter 8.) 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 then external computations that require the data to be returned to the application. Stored procedures are commonly used in critical enterprise databases, because they improve security and performance.

> ### Domain
> The domain of a variable is the set of legal values that the variable can take on.

Another basic general technique for improving performance is to constrain the domain 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 to look for a particular value requires converting it to standard casing, using SQL similar to

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

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

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

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

There are usually many more SELECTs than INSERTs or UPDATEs, so take time during changes to constrain the data, because this will help the queries, which are much more common.

Test Yourself 5.5

Which of the following statements regarding performance tuning are true? (Please check all of the following that are true.)

Removing a column that contains a value derived mathematically from the value in another column and is never used in a query can improve performance.

This is true. Reducing the lengths of rows in a database table can improve performance. 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.

When large amounts of data are needed for computations, performing the computations on the application side of a network connection to the database generally improves performance and provides the added benefits of improved security and reduced network traffic.

This is false. Performing the computations on the database side can improve performance, improve security, and reduce network traffic. This can be accomplished through the use of stored procedures or triggers.

Limiting the possible values that can be stored in a column can improve query performance.

This is true. Constraining the domain of the data can eliminate performing operations on the data to convert it to some standard form at query time.

Summary

Design relational database applications for scalable performance. Scalable performance requires coordinated design of hardware, the 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 14 - Distributed Database Management Systems (DDBMS)

Introduction

metcs669_module12 video cannot be displayed here

A distributed database management system (DDBMS) governs the storage and processing of logically related data over interconnected computer systems in which both data and processing functions are distributed among several sites. A DDBMS offers several advantages over traditional systems: the data are located nearer to the sites with greatest demand; data access is faster; data processing is faster; growth is facilitated; communications are improved; operating costs are reduced; interfaces are user-friendly; danger of single-point failure is reduced; and the data is processor independent. The DDBMS also suffers from several disadvantages: the complexity of management and control; security; lack of standards; increased storage requirements; greater difficulty in managing the data environment; and increased training costs. A DDBMS is constructed from several components: computer workstations; network hardware and software components; communications media; a transaction processor (TP), also called an application manager (AP) or transaction manager (TM); and a data processor (DP), also called a data manager (DM). Together these provide a platform that enables users to interact with distributed data in a variety of ways.
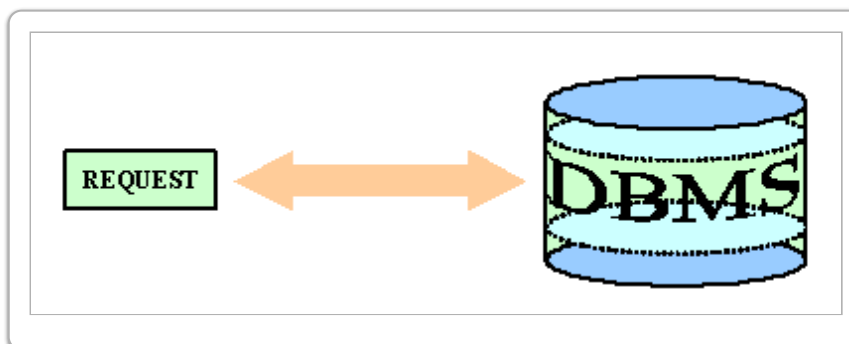
The most primitive and least effective of the distributed database scenarios is based on a single SQL statement (a "request" or "unit of work") that is directed to a single remote DBMS. Such a request is known as a *remote request*. Remember the distinction between a request and a transaction:

- A *request* uses a single SQL statement to request or change data.
- A *transaction* is a collection of one or more SQL statements executed as an atomic business operation.

Test Yourself 5.6

Which of the following are true regarding distributed database management systems (DDBMS)?
(Please check all of the following that are true.)

Data can be split up among multiple storage sites, but all processing must occur at a single central location.

This is false. Processing functions can be distributed among multiple sites as well.

Suppose that there is a nation-wide business organization with regional offices and that most of the access to data for a region is from the regional offices. A distributed database could probably be designed so that the regional data is located in data processors at the regional offices.

This is true. The ability of distributed databases to locate data near sites that most frequently access the data has many advantages. Region-based distributed database designs are common, because they provide superior performance for the common local access to local data, and also because these distributed databases can be designed so that local transactions can be performed without connections to the larger distributed database, improving availability in the event of network or other outages. Region-based designs can also reduce wide area network traffic and costs.

As compared to a centralized (non-distributed) database, a failure at one point in a distributed database is more likely to shut down the operations of an entire organization.

This is false. Distributed databases can be designed so that they have very high availability. This is one reason that distributed databases are used for banking and other high-availability applications. The consequences of single-point failure are reduced with a well designed distributed database.

Transactions that involve changes to data at different data processors are not possible with a distributed database, because it is very difficult to coordinate the different data processors to ensure atomicity of the distributed transactions.

This is false. Managing distributed transactions is more complex than single-site transactions, but distributed transactions are key to distributed databases, and are the primary distinguishing feature of a DDBMS.

How Database Implementation is Affected

Database systems can be classified on the basis of how process distribution and data distribution are supported. In the single-site processing, single-site data (SPSD) scenario, all processing is done on a single CPU or host computer and all data are stored on the host computer's local disk. Processing cannot be done on the end user's side of the system. This scenario is typical of mainframe computers. With the multiple-site processing, single-site data (MPSD) scenario, multiple processes run on different computers using a single data repository. Typically the MPSD scenario requires a network file server running conventional applications accessed through a LAN. The multiple-site, multiple-data scenario describes a fully distributed database management system with support for multiple data processors and transaction processors at multiple sites. *Homogeneous* distributed database systems integrate one type of DBMS such as Oracle or Microsoft SQL Server over a network. In contrast, a *heterogeneous* distributed database supports different types of DBMSs that may even be based on different data models on different computer platforms. Homogeneous distributed database systems are more common than heterogeneous.

Multiple requests accessing a single remote database
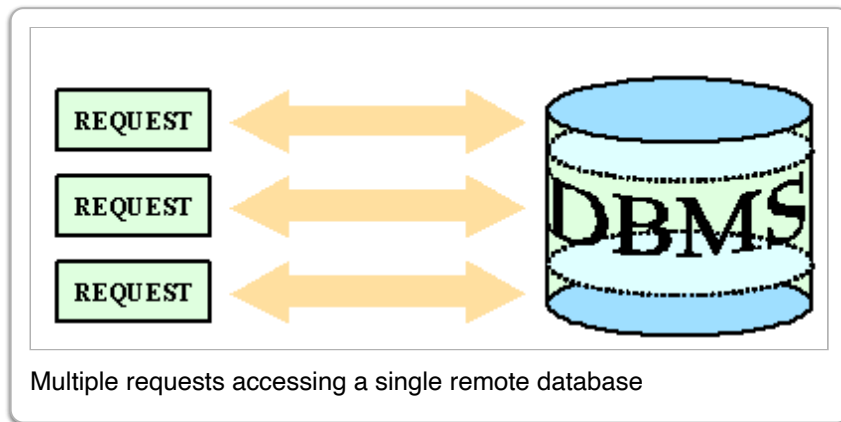
A unit of work now consists of multiple SQL statements directed to a single remote DBMS. The local user defines the start/stop sequence of the units of work, using COMMIT, but the remote DBMS *manages* the unit of work's processing.

Multiple requests accessing multiple remote databases
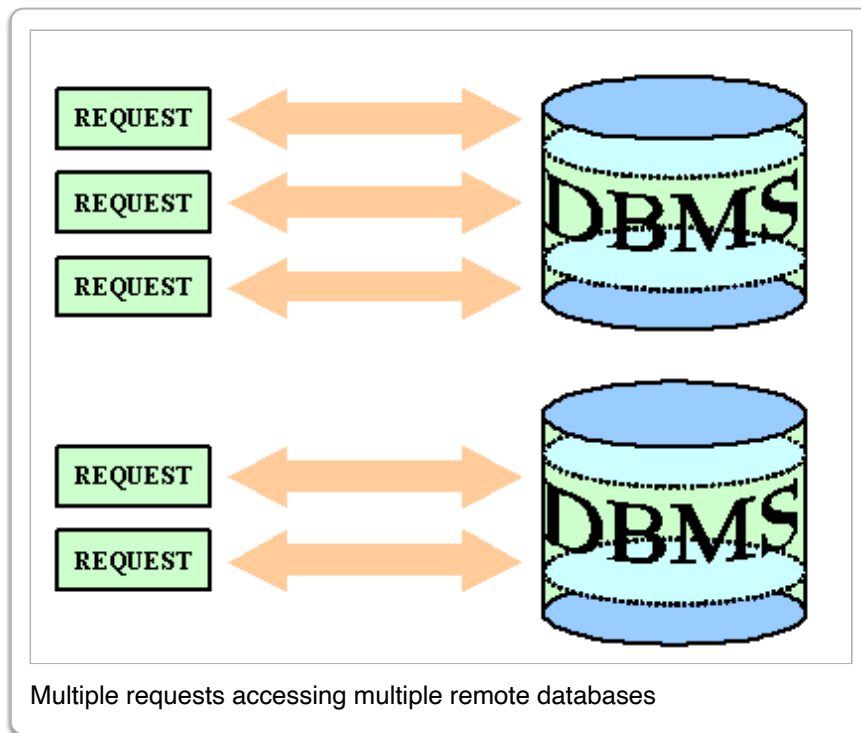
Test Yourself 5.7

Please select all of the following that are correct.

In the single-site processing, single-site data (SPSD) scenario, all processing must be done at a single end user site.

This is false. All processing is done on a host computer. Processing does not occur on the end user side.

In a fully distributed database system processing can occur at multiple sites.

This is true. Also data is stored at multiple sites.

An Oracle DBMS and a Microsoft SQL Server DBMS can both be part of the same DDBMS.

This is true.This is referred to as a heterogeneous distributed database system.

Heterogeneous distributed database systems are more common than homogeneous.

This is false. Homogeneous systems are more common. A homogeneous system contains only one type of DBMS.

Managing Transactions in Distributed Environments

A transaction is composed of one or more database requests. An undistributed transaction updates or requests data from a single site. A distributed transaction may request or update data from multiple sites. The issue of distributed concurrency becomes important in a network of distributed databases. A two-phase COMMIT protocol ensures that all parts of a transaction are completed. It requires a DO-UNDO-REDO protocol and a write-ahead protocol. If part of a transaction cannot be completed, then all changes made at other sites must be completely undone to maintain a consistent database. A distributed DBMS must evaluate every data request to find the optimum access path. The DDBMS must optimize the query to reduce access communications and CPU costs associated with the query.

A unit of work now may be composed of multiple SQL statements directed to multiple remote database management systems. Moreover, any of these SQL statements can join or otherwise access any combination of local and remote database management systems. As was true in the second scenario, the local user defines the start/stop sequence of the units of work, using COMMIT, but the remote DBMS to which the SQL statement was directed *manages* the unit of work's processing. In this scenario, a two-phase COMMIT must be used to coordinate COMMIT processing for the multiple locations.

Multiple requests accessing any combination of multiple remote DBMSs
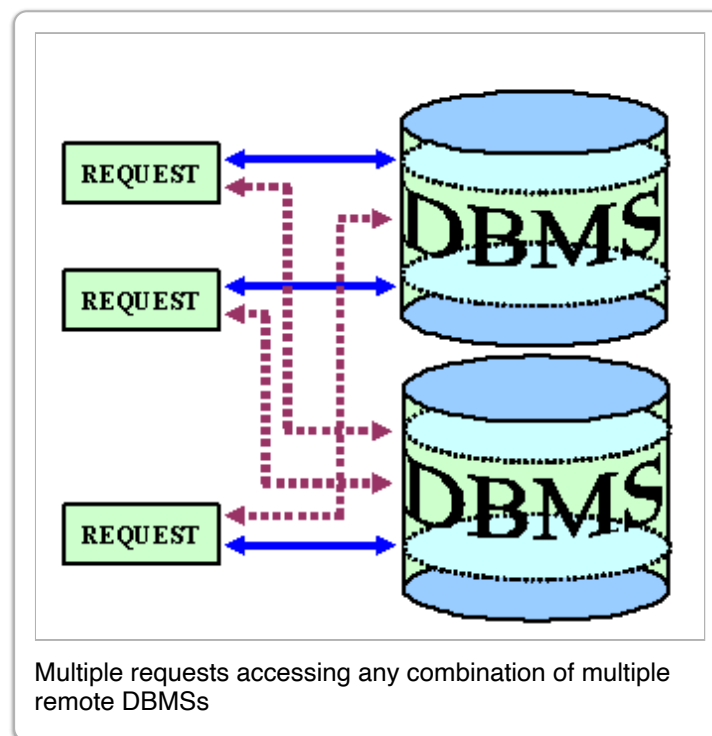
Test Yourself 5.8

Which of the following are true regarding distributed database management systems (DDBMS)?
(Please check all of the following that are true.)

With a distributed database, a single transaction may update data at multiple sites.

This is true. This is referred to as a distributed transaction.

Suppose that a distributed transaction has two SQL statements, both of which are UPDATE statements. If changes made by the second SQL statement of the transaction cannot be committed, then any changes made by the first SQL statement must be undone.

This is true. This is the atomicity requirement for transactions, including distributed transactions. Atomicity of distributed database transactions is ensured by the two-phase COMMIT protocol.

With a write-ahead protocol, data update operations are performed and saved (written ahead) to permanent storage before transaction logs are updated.

This is false. With a write-ahead protocol, transaction logs are revised to contain the updates and saved to permanent storage before the updates are made. Write-ahead is necessary so that if it is necessary to undo the updates, then the information needed to undo the changes is in the transaction logs, and to ensure recovery in the event that the database instances crashes before the distributed transaction is completed.

How Database Design is Affected by Distributed Environments

Database design in a distributed environment must consider fragmentation and replication of data. In addition to concerns common to all database design efforts, designers of distributed database must decide how to fragment the data and how to allocate each fragment or replica to obtain better overall response time and to ensure data availability to the end user. A database can be replicated over several different sites on a computer network. This puts important data closer to the applications and users requiring it. This can improve availability and decrease access time. A database can be not replicated, fully replicated, or partially replicated. Data fragmentation allows a designer to break a single object into two or more fragments. Horizontal, vertical, and mixed fragmentation strategies can be used. Data allocation strategies are designed to determine where database fragments or replicas should be located. Centralized, partitioned, and replicated data allocation are all options.

Let's look at a database design problem that can be best understood by looking at the following exercise. The following data structure and constraints exist for a magazine publishing company. Click here to review the Self-Paced Database Design Exercise. Given the exercise scenario, consider your answers to the questions.

Test Yourself 5.9

Please select all of the following that are correct.

Data in a distributed environment may be fragmented or replicated, but not both.

This is false. The use of both fragmentation and replication is commonly part of a good design. For example, a Sales table may be horizontally fragmented by sales region, allocated to the regional

offices, and replicated at a headquarters site.

If a table is vertically fragmented each fragment has the same number of rows.

This is true. Each fragment will have fewer columns, but the same number of rows.

If a table is horizontally fragmented each fragment has the same number of columns.

This is true. Except in the special case where all but one fragment has no rows, after horizontal fragmentation each fragment will have fewer rows than the unfragmented table. Each horizontal fragment will have the same number of columns as the unfragmented table.

If a table is fragmented using mixed fragmentation each fragment could have a different number of rows.

This is true. Mixed fragmentation is a combination of horizontal and vertical fragmentation. Because horizontal fragmentation can result in fragments with different numbers of rows, mixed fragmentation can result in fragments with different numbers of rows. For example, suppose that an Employee table is fragmented with mixed fragmentation. Suppose that the table is horizontally fragmented by region. Suppose further that the table is fragmented vertically, with the salary columns at a headquarters HR database, and the other columns in the regional databases. Then the number of rows in the regional fragments would probably different, unless the number of employees in each reagion is the same.

Lecture 15 - Subqueries

Subqueries and Correlated Queries

A *subquery* is based on the use of the SELECT statement to return one or more values to another query. A subquery can return a single value, a list of values, or a virtual table. A subquery can also return no values at all. The most common type of subquery uses an inner SELECT subquery on the right side of a WHERE comparison expression. When you want to compare a single attribute to a list of values, you use the IN operator. Just as you can use subqueries with the WHERE clause, you can use a subquery with a HAVING clause.

The IN subquery uses an equality operator; that is, it only selects those rows that match (are equal to) at least one of the values in the list. To perform inequality comparisons you must use either ANY or ALL. The ALL operator allows you to compare a single value with a list of values returned by the first subquery by using a comparison operator other than equals. A subquery can also be placed in the attribute list where it is known as an inline subquery.

A *correlated subquery* executes the inner query once for each row in the outer query. In a correlated subquery, the inner query references a column of the outer query and is therefore related to the outer query. Correlated subqueries can also be used with the EXISTS special operator.

A *subquery* is a query (expressed as a SELECT statement) that is located inside another query. The first SQL statement is known as the *outer query*, the second is known as the *inner query* or *subquery*. The inner query or subquery is normally executed first. The output of the inner query is used as the input for the outer query. A subquery is normally expressed inside parenthesis and can return zero, one, or more rows and each row can have one or more columns.

A subquery can appear in many places in a SQL statement:

- as part of a FROM clause,
- to the right of a WHERE conditional expression,
- to the right of the IN clause,
- in an EXISTS operator,
- to the right of a HAVING clause conditional operator,
- in the attribute list of a SELECT clause.

Examples of subqueries are:

```
INSERT INTO PRODUCT
SELECT * FROM P;

DELETE FROM PRODUCT
WHERE V_CODE IN (SELECT V_CODE FROM VENDOR
WHERE V_AREACODE = '615');

SELECT V_CODE, V_NAME
FROM VENDOR
WHERE V_CODE NOT IN (SELECT V_CODE FROM PRODUCT);
```

A *correlated subquery* is subquery that executes once for each row in the outer query. This process is similar to the typical nested loop in a programming language. Contrast this type of subquery to the typical subquery that will execute the innermost subquery first, and then the next outer query until the execution of the last outer query is completed. That is, the typical subquery will execute in serial order, one after another, starting with the innermost subquery. In contrast, a correlated subquery will run the outer query first, and then it will run the inner subquery once for each row returned in the outer subquery. For this reason, some queries with correlated subqueries can be slow.

For example, the following subquery will list all the product line sales in which the "units sold" value is greater than the "average units sold" value for *that* product (as opposed to the average for *all* products.)

```
SELECT INV_NUMBER, P_CODE, LINE_UNITS
FROM LINE LS
```

```
WHERE LS.LINE_UNITS > (SELECT AVG(LINE_UNITS) FROM LINE LA
WHERE LA.P_CODE = LS.P_CODE);
```

Note that the correlated subquery references the table alias LS defined in the outer query. The previous nested query will execute the inner subquery once to compute the average sold units for each set of LINE attributes returned by the outer query.

Test Yourself 5.10

Which of the following is correct? (Please check all of the following that are correct.)

A subquery is a query located inside another query.

This is true. The first SELECT statement would be referred to as the outer query. The second SELECT statement would be referred to as the inner query or subquery.

A subquery can return one and only one value.

This is false. A subquery can return a single value, a list of values, no values, or a result set, which is like a table, with potentially multiple records, each with potentially multiple columns.

A subquery can be used with the IN operator.

This is true. Subqueries can be used with multiple operators including IN, ALL, and EXISTS.

A correlated subquery will run the outer query first.

This is true. As opposed to a typical subquery, that will run the innermost query first. In programming terms, a correlated subquery is like nested loops, where the outer query runs and then runs the inner query for each value of the outer query.

The TEAM table has 12 rows. The PLAYER table has 288 rows. Consider the following SQL command:

```
SELECT team_id AS TID, team_name
FROM Team
WHERE EXISTS (SELECT * FROM Player WHERE Player.team_id = TID
AND Player.player_salary > 1000000)
```

The inner correlated subquery SELECT * FROM Player WHERE Player.team_id = TID AND Player.player_salary > 1000000 runs once.

This is false. It should run 12 times. (Once for each row of the Team table.)

Nested Queries

SQL allows us to nest one query inside another. ISO SQL permits a subquery only on the right hand side of an operator. The operators available are IN, EXISTS, and the relational operators =, <>, >, >=, <, <=. When a subquery will return just a single row (normally a single field value) SQL allows the use of a relational operator. When a subquery returns more than one row (a set of rows) a relational operator must be used with ALL or ANY.

Use of IN

The IN operator is used to determine if some value is present in a list of values. That list can be explicitly specified (covered previously) or generated by a subquery.

Example List names and ages of patrons who have books out on loan. We shall proceed by giving two separate queries to accomplish this and then we specify it as one nested query. Consider the query:

```
SELECT userid
FROM loan
WHERE dateret IS NULL;

USERID
20
30
40
```

We can now specify and execute the query:

```
SELECT name, age
FROM patron
WHERE userid IN (20,30,40);

NAME AGE
Jones 41
Niall 17
King 21
```

These two queries can easily be combined; we replace the list (100,250,400) with the first subquery:

```
SELECT name, age
FROM patron
WHERE userid IN
(SELECT userid FROM loan
WHERE dateret IS NULL) ;

NAME            AGE
-------------- ---------------------
King            21
```

```
Niall          17

Jones          41
```

Example List names of patrons who have borrowed a computing or history book.

```
SELECT name
FROM patron
WHERE userid IN
(SELECT userid FROM loan
WHERE callno IN
(SELECT callno FROM book
WHERE subject IN
('Business','Computing')));

NAME
Wong
King
Niall
Jones
```

Let's analyze the above SELECT; it is composed of three queries. The innermost query is a simple one and we can consider it separately. That is, we can consider that the database system replaces it by a list of call numbers that have been retrieved from the BOOK table. The "middle" query retrieves the user ID for any patron who has borrowed a book in the computing or history categories. The outer query lists the names of these people.

Simple Comparison Operators: =, <, ...

We use two examples to illustrate the use of the comparison operators with subqueries. To the beginning SQL user, the need for subqueries in these cases is not obvious.

Example Who has paid the largest fine? This is such a simple question, but unfortunately it is not trivial to formulate the correct SQL query. One may be first tempted to express the query as

```
SELECT userid, MAX(fine)
FROM loan;
```

However, this is not correct since MAX is a function which operates on the whole table and hence is single-valued and userid is multi-valued—there are multiple values, one for each row. We cannot mix these two types of expressions.

To explain the correct formulation of the query, we begin by constructing the query in two parts. First (and this corresponds to the innermost query we shall use) we find the largest fine:

```
SELECT MAX(fine)
FROM loan ;
```

This query becomes our nested query and all that we need to do is compare each fine to this maximum value:

```
SELECT userid
FROM loan
WHERE fine = (SELECT MAX(fine)
FROM loan) ;

USERID
30
```

The 'equals' is permitted for testing with respect to the subquery since it is known that the subquery returns just one value.

Example What is the name of the oldest patron?

```
SELECT name
FROM patron
WHERE age =
(SELECT MAX(age)
FROM patron);

NAME
Smith
```

Boston University Metropolitan College