Database Architecture: & configuration

Learning Outcomes : At the end of this section the student will be able to

- Describe and explain selected concepts relating to the architecture and configuration of a multi user database system.
- Describe how CPU scheduling and memory allocation are linked.
- Explain the fundamental techniques used to implement SQL Joins efficiently.
- Explain how/why different DBMS have different approaches to the physical implementation of logical tables.
- Explain why SQL Dialects and portability are related in DBMS.
- Explain the difference between Embedded SQL and Embedded DBMS?
- Describe the following terms/concepts applied to database systems:
  load balancing; relation between fault tolerance and multi-threaded systems; an n-tier system; load balance and multi-threaded systems; how multi-threaded systems relate to 'good' database configuration; database memory management.
- Eexplain the following concepts/terms compiler, open source, modular design, transactions, B-tree Index, Query execution plan, Client Server, Embedded, Scalability, Connectivity, Localization, Database Tools

We will first examine a list of features from a popular database system, and then select aspects that are relevant for this module for more detailed examination.

The Main Features of MySQL (Version 4)

Internals and Portability:

- Written in C and C++. Tested with a broad range of different compilers.
- Works on many different platforms.
- Uses GNU Automake, Autoconf, and Libtool for portability.
- The MySQL Server design is multi-layered with independent modules.
- Fully multi-threaded using kernel threads. It can easily use multiple CPUs if they are available.
- Provides transactional and non-transactional storage engines.
- Uses very fast B-tree disk tables (**MyISAM**) with index compression.
- Relatively easy to add other storage engines.
- A very fast thread-based memory allocation system.
- Very fast joins using an optimized one-sweep multi-join.
- In-memory hash tables, which are used as temporary tables.
- SQL functions are implemented using a highly optimized class library and should be as fast as possible. Usually there is no memory allocation at all after query initialization.
- The MySQL code is tested with Purify (a commercial memory leakage detector) as well as with Valgrind, a GPL tool
- The server is available as a separate program for use in a client/server networked environment. It is also available as a library that can be embedded (linked) into standalone applications i.e. used in isolation or in environments where no network is available.

Data Types:

- Many data types: signed/unsigned integers 1, 2, 3, 4, and 8 bytes long, **FLOAT**, **DOUBLE**, **CHAR**, **VARCHAR**, **TEXT**, **BLOB**, **DATE**, **TIME**, **DATETIME**, **TIMESTAMP**, **YEAR**, **SET**, **ENUM**, and OpenGIS spatial types. Fixed-length and variable-length records.

Statements and Functions:

- Full operator and function support in the **SELECT** list and **WHERE** clause of queries. For example:

```
mysql> SELECT CONCAT(first_name, ' ', last_name)
    -> FROM citizen
    -> WHERE income/dependents > 10000 AND age > 30;
```

- Support for **LEFT OUTER JOIN** and **RIGHT OUTER JOIN** with both standard SQL and ODBC syntax.
- Support for aliases on tables and columns as required by standard SQL.
- **DELETE**, **INSERT**, **REPLACE**, and **UPDATE** return the number of rows that were changed (affected).
- The MySQL-specific **SHOW** statement can be used to retrieve information about databases, storage engines, tables, and indexes. MySQL 5.0 adds support for the **INFORMATION_SCHEMA** database, implemented according to standard SQL.
- The **EXPLAIN** statement can be used to determine how the optimizer resolves a query.

Security:

- A privilege and password system that is very flexible and secure, and that allows host-based verification. All password traffic is encrypted when you connect to a server.

Scalability and Limits:

- Handles large databases. We use MySQL Server with databases that contain 50 million records. We also know of users who use MySQL Server with 60,000 tables and about 5,000,000,000 rows.
- Up to 64 indexes per table are allowed. Each index may consist of 1 to 16 columns or parts of columns. The maximum index width is 1000 bytes (767 for **InnoDB**);

Connectivity:

- Clients can connect to MySQL Server using several protocols:
  - Clients can connect using TCP/IP sockets on any platform.
  - On Windows systems clients can connect using named pipes if the server is started with the --enable-named-pipe option. In MySQL 4.1 and higher, Windows servers also support shared-memory connections
  - On Unix systems, clients can connect using Unix domain socket files.
- MySQL client programs can be written in many languages. A client library written in C is available for clients written in C or C++, or for any language that provides C bindings.
- APIs for C, C++, Eiffel, Java, Perl, PHP, Python, Ruby, and Tcl are available, allowing MySQL clients to be written in many languages.
- The Connector/ODBC (MyODBC) interface provides MySQL support for client programs that use ODBC (Open Database Connectivity) connections.
- The Connector/J interface provides MySQL support for Java client programs that use JDBC connections. Clients can be run on Windows or Unix. Connector/J source is available.
- MySQL Connector/NET enables developers to easily create .NET applications that require secure, high-performance data connectivity with MySQL. It implements the required ADO.NET interfaces and integrates into ADO.NET aware tools. Choice of .NET languages. MySQL Connector/NET is a fully managed ADO.NET driver written in 100% pure C#.

Localization:

- The server can provide error messages to clients in many languages.
- Full support for several different character sets, including Unicode, **latin1, german**, **big5**, **ujis**, and more. All data is saved in the chosen character set.

Clients and Tools:

- MySQL AB provides several client and utility programs. These include both command-line programs such as **mysqldump** and **mysqladmin**, and graphical programs such as MySQL Administrator and

MySQL Query Browser. A **mysqlcheck** client for SQL statements to check, optimize, and repair tables.

Features of a database system explained

Complier:
A compiler is a program that typically transforms program source code into some other target language. In most cases, this is to translate high level program code into a lower level code (assembler or machine code) that is executable on a specific system.

Q. Why would you ever need to compile database code?
Once you have the DBMS, you'd expect it to come in an executable form only, so how and why would you ever recompile it? The answer to this question is linked to other terms such as 'open source', modular, embedded etc. See below.

Open source means that the source code of the DBMS itself is available to you. This is generally not the case with proprietary system (where you pay for a licence to use, not edit the program). It means that you can program (or reprogram) aspects of the system to suit your particular installation. We are informed that the source code for MySql is written in C/C++ so once you can program in that language, you can program the DBMS itself (and recompile it into an executable). See Modular later.

Data Manipulation:

Database processing (data manipulation) is be in two forms. Individual operations (e.g. select) or transactions. The standard database language is Structured Query Language (SQL).

Transaction: a logical sequence of SQL statements

For a DBMS to manage transactions takes a lot of work. An installation that requires the DBMS to handle transactions must handle multiple users accessing the shared data simultaneously. In addition to ensuring that the shared data is kept consistent while users read and update, the DBMS is also responsible for the recovery of the system if failure occurs in the middle of transactions.

Some databases are basic, and do not handle transactions. MySql is designed to allow you the option of using transactions or not. Because it is modular, you can tailor an installation to suit the specific needs of your organization.

Database Transactions

'Provides transactional and non-transactional storage engines'

A transaction in a database, called a **database transaction**, usually refers to a unit block of operations performed on behalf of a user e.g. a number of updates to a patients record system.

A transaction is the database unit of concurrency (the unit to be protected from others). So we have multiple transactions competing to read and/or modify shared data items. The system must protect the entire transaction not just individual operations inside a transaction. The transaction is the unit of recovery (i.e. it defines what needs to be recovered to bring the DB back to a correct state after a failure. Note transaction management is a major section of this module (recovery, concurrency).

MySQL 'storage engine' is a non standard DBMS concept, and has no equivalence in other DBMS.

Since the transaction is a sequence of related operations, the DBMS must protect or recover the entire transaction.

Note the save/auto save options in MS-Word, which acts to commit a block(group) of modifications. Also the Undo function to reverse an action in case of error. We can think of the Word text .doc file as the database in this example. Undo is an example of recovery back to a correct state.

Transaction Management: This term can be used to define the capability of a Database System. Smaller databases are usually designed for a single user and do not implement/cater for transaction management; however large multi-user systems do. Transaction management needs concurrency & recovery sub systems in the DBMS.

Even where a database system allows multiple users, they may not implement full transaction protection for multiple users operating simultaneously

The concurrency & recovery sections of this module deal with multi-user transaction systems.

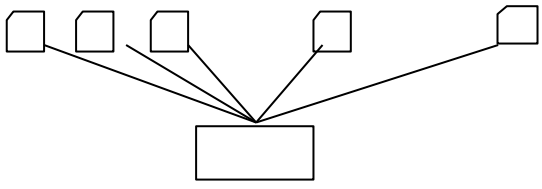
CPU scheduling & threads:

Processes are in competition for CPU time slices. In relation to the CPU we should understand that firstly the database system may be in competition with other programs/processes running on the system including the operating system, web servers, name servers etc. Moving from one process to another is called 'context switching' as the context that the CPU is working in is changed to another program. This switching is a delay cost, and al processes are treated equally i.e. there is no way to prioritise important processes (a disadvantage).

Multi-threading relates to **CPU scheduling**. A thread is a management unit of work for running on the CPU. Let's say you have 3 processes running on the computer: an email client, an internet browser and a DBMS. All three have equal importance so they are all scheduled for CPU time equally.

However, what if one of these processes is very important and it has a lot of work to do?
How can you allocate more CPU time to it? Threads allow you do just that. The system administrator can configure the system so that the DBMS is allocated more threads than the others e.g. we give 3 threads to the DBMS. **The CPU is then managed by scheduling the threads not the full process**. So the DBMS will get 3 times more CPU than each of the others (Email, Browser)

	DBMS	Internet Browser	Email	
Old Process-	1	1	1	DBMS gets CPU 1 in 3(33%)
With Threads-	3	1	1	DBMS gets CPU 3 in 5(60%)



When a process/thread is executing, it uses RAM for the data that it is processing. Therefore, we should note, that **threads and memory allocation are linked**. A thread based system is efficient as it

conserves global memory for the overall process, as threads are swopped. In other words, the parent process can manage shared memory between its threads. So in general, memory management is better and the context switching for threads are lower cost while maintaining the same parent address space and control. Because of this threads are called 'light weight',

Note that in many systems the sharing of session data is desirable for efficient performance (e.g. data generated during a client connection). Since the parent process controls the address space for all child threads, the database (in this case) can implement the sharing of important data between its child threads easily.

Threads also enable a more **robust system** essential where demands of **high availability** are becoming a standard feature of many business systems. Usually this demand is prompted by web interfaces for transaction systems such as sales, ordering etc. Reliability means that the system is as close to constantly available as is possible. The threads being relatively autonomous makes the system **fault tolerant**. This means that individual threads can fail without affecting the system as a whole. (important for recovery). A multi threaded database server can create and manage many simultaneous client connections, one connection can fail without affecting the others & general DBMS function.

The number of threads may be a configurable parameter, and allocation may be on a module/ component basis e.g. the DBA may be responsible for setting the number of threads for the recovery system. http://dev.mysql.com/doc/refman/5.0/en/connection-threads.html

The ability to configure the system with regard to allocation of the number of threads means that e.g. priority can be given to the more important database functions e.g. recovery log manager. Note how MS-Word has the ability to background print (a slow write to an external device), while the user can still use the other normal functions of Word. The DBS recovery log manager is a slow write to a disk. Multi threaded systems allow multi tasking (multi processing) within the one application (the database system in this case).

Query Processing and optimization:

QEP: Query Execution Plan

When an SQL statement is run, the DBMS must decide on a **query execution plan** (QEP) to implement that SQL. The DBMS may run an optimizer program to implement the SQL statement in an efficient way i.e. it develops the QEP.

Many systems implement a command called 'Explain'. http://dev.mysql.com/doc/refman/5.0/en/explain.html

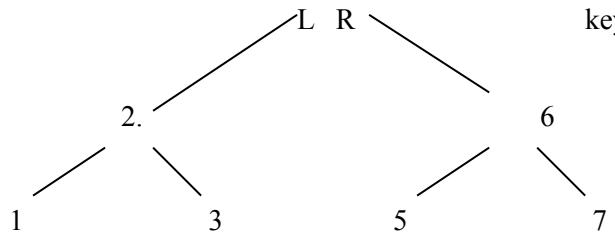
The Syntax is: **Explain** 'SQL query text' e.g. Explain Select * from Students

The Explain function takes the query given as parameter and executes it, but returns some of the QEP details for executing that query e.g. info on what access mechanism(s) were used or available.

Access mechanism: an access mechanism is the way (mechanism) the DBMS finds (accesses) the records it needs out on the disk. The users interact with the system using SQL, structured query language. The system may use available access mechanisms to create an efficient QEP. An Index is an example of an access mechanism, Hashing (based on the key value) is another (see hash tables later).

Index: a type of access mechanism to enable faster searching(access) to the data file on disk.
B-Tree.
Instead of a sequential list (array) of record key values 1,2,3,4,6,7 a balanced tree is organized as follows (note:irrespective of the order of the inserts of the key values)

Root: ——— 4 This example just displays the



Select * From Tree Where Search_Key_value = 6

Start at Root, compare Search_key_value with Tree_node_key_value
If = then record found, else if > go right else if < go left.

This is a **balanced tree** and it gives fast but importantly uniform cost retrieval for any search key.

A balanced Btree is a dynamic file organization. It is a self organizing file organization in that it dynamically reorganizes itself as inserts, deletes occur to maintain the balance. The DBMS does not stop user activity to reorganize. **This is essential in high availability systems.**

Why is balance important? A balanced Tree gives **consistent and uniform** performance i.e. gives uniform retrieval for any key value 'selected' by the user. This is critical for the query optimizer to produce efficient and repeatable query execution plans. The query processor works by estimating the cost of a query. When estimating, a uniform cost is preferable to an average cost, because an average covers a range of different query costs (i.e the actual cost can vary wildly). Therefore, uniform gives more reliable/accurate estimates of real execution costs.

Compare this to a linear search of a list 1,2,3,4,5,6,7,8 search for 2 is fast but search 6 is slow. (average search time is N/2, half the list, N = number of items(records) in the list).

A Btree can be found in a number of forms. The form depends on what is in the nodes of the tree. We must store any pointer(s) (required for Tree structure), we won't display these, we'll just assume they are there. In a Primary Index, we must store the key of the database record. However, what else is stored in each node? i.e. besides the key and the tree pointers themselves.

Key=3	?????

Option: 1. Key, + rest of the data record e.g. name, address, phone etc.
Option: 2. Key + pointer to the data record stored elsewhere on disk (in a different file?)

Option 1 is an Index and a Data file **combined**. This is where the nodes of the tree contain actual data records. This is a **primary indexed sequential file** i.e. the index is a btree and the data file is sequentially ordered dictated by the index elements.

Alternatively, in option 2 a DBMS might split the Index from the Data File. In this form the nodes of the Btree contain pointers into the location of the records in the separate Data file i.e. stored elsewhere on the disk.

Most DBMS also allow independent Btree index files i.e. the data file is not bound to the index. In this way you can have multiple Btree indexes on different attribute(s) or column(s) of the file. Indexes defined on non primary key are called **secondary indexes.** These are all separate index files to the main data file(table) i.e. option 2 above.

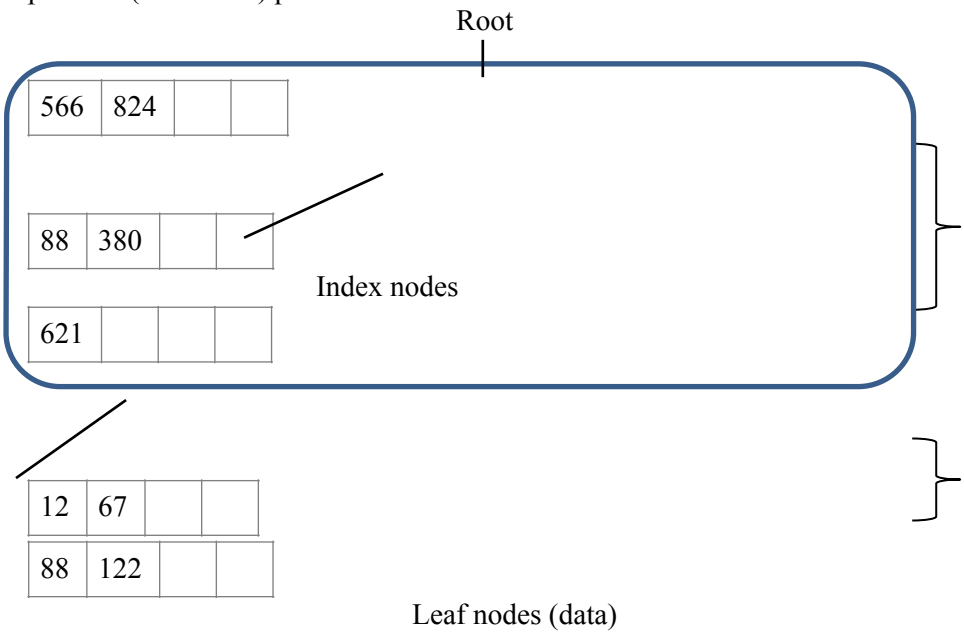
So, Btrees can be used as an Index or as an Indexed Sequential File Organisation.

An Index is an extra cost in space and maintenance. The Index gives fast lookups for a random search e.g. on the key value; however it is not as fast as a hash (as the index must be read and processed first then the data file is accessed once you know where to look in it). Note: for application

with large batch inserts, it may be faster to drop indexes before the batch insert and then reform the index afterwards. Maintenance is work done to change the index based on modifications to the main data file. **This is a critical performance issue as a simple data record delete (or update, insert) may result in major index re-organisation requiring multiple reads/writes which blocks other users.**

In general, Btrees give the best 'all round' performance, i.e. it can handle a range of different queries well, even though it may not be optimum for any of these e.g. search on key value, order, range of key

Rather than just having two branches, B+ trees use the same principle as above but have more pointers (or records) per node.



One variation of this type of BTree, is that only the leaf nodes of the tree contain the data records. The Index nodes 'pack' more keys per node, so the index section of the Indexed Seq file is smaller (for faster searching). Notice that you can search the index section of the tree to find individual records, or start at the first leaf and search sequentially across the leaves of the tree (i.e. avoid index) to access just the ordered data in the file. So this is a version of an Indexed Sequential file.
Question? Why is this version of an Indexed Sequential file more efficient?

Answer: **the non key data attributes are the one that take up most space in a record**(e.g. name, address etc). In most searches (retrievals) you need only check the key and decide if you're interested in the full record? So, for searches that only result in a small number of records to be returned to the user, there is a significant cost in storing all the data values with the key in the Btree nodes. You are better off (more efficient) to only store the key with the pointers required to maintain the Btree structure. **The leaf nodes of the tree store the data record**. This reduces the size of the index section of the file to be small that you can process in Main memory quickly. Once you process this index to find what actual data records you want, then go to the main data and extract the specific records you want. The root node of the tree index (and as many higher level index nodes) should be memory resident to avoid reading the disk for parts of the index. For a standalone pure Index, just replace the data leaf node with pointers to a remote data file.

Question: Why not generate indexes on every attribute?

In a book, the index is an integral part of the book. In a Database System, an index is stored in a file. Some DBMS do store the Index with the main data file (i.e. the index uses pages with the file

allocated to the table). However, only one index can be integrated (called the Primary Index i.e the index on the primary key). Any other indexes would be stored in a separate file. Indexes should be generated on columns for fast retrieval. We note:

- Indexes take up storage space
- Indexes are used to speed up retrieval (but not guaranteed to do so for all queries)
- The DBMS reading the index then the data file is a cost and may incur Disk Seek costs
- Indexes must be maintained for updates, so an application with frequent updates will incur index maintenance costs (sometimes significant performance cost). See concurrency later.
- Not all columns are searched frequently enough to justify index costs (maintenance and storage)
- Some files/tables are small enough to read into memory; So no need for an index.
- Index management is one of the top performance issues in a DBMS.

Learning objectives: You should be able to

- Explain whether a Btree is an index or a datafile organsiation?
- Explain why Btrees are recommended for high availability applications?
- Describe how Btrees are dynamic (or self organizing) & Explain what is meant by a balanced Btree?
- Explain why Balanced Btree offer good performance?
- Explain what types of applications are Btrees suitable for?
- Compare binary search and balanced BTrees.
- What is the difference between a BTree and a B+Tree?
- Explain how indexes might be managed for large batch inserts/updates?

Optimising a query

'Very fast joins using an optimized one-sweep multi-join'

Fast Joins and the notion of one sweep:

We should understand that all non procedural SQL code is transformed into procedural code to execute on the CPU (i.e. the CPU executes instruction at a time in sequence i.e. procedural). If we look at an SQL join of two tables to find the suppliers that currently supply:

Tables: Supplier (Sno, Sname, Address, City etc) and Shipment (Sno, Pno, Date, Qty)

SQL Query: find the name and address of Suppliers London that currently supply

```
Select sname, address
From Supplier S Join Shipment SPJ On S.Sno = Spj.Sno
Where S.City = 'London'
```

This non-procedural SQL is transformed into procedural code that needs to perform two major tasks

1. Find rows in Supplier with London in the city field.
2. Join rows in the Supplier file with rows in the Shipment file that have the same Sno value.

A Query processor (optimiser) could perform these two operations in either order, (1-2) or (2-1).

Which is more efficient and why? This is the work of a database optimiser

---

Let's examine the join operation in isolation to understand the processing of the file(s). **Went you process a file from beginning to end, we call it making a pass of the file. A For loop that read a file is a pass of the file.**

Sno	Pno	Qty

Supplier SPJ

Sno	Sname	Status	Address

For every row in first table, check all rows of the other.

Once, the first pass of the second table is performed, you move on to the next row/record/tuple of the first table and repeat, and so on for every row combination. Note: database tables are suitable for 100,000's of rows, so this is a lot of processing.

Computer programmers in C, C++ or JAVA, may understand that a Join is effectively implemented by using what is called 'nested loops'; i.e.

A query optimiser can reorganise the SQL code and implement it in different ways.

So let us rewrite the SQL query

Select Sname, Address

From Supplier S

Where Exists or use In

(select Sno

From Shipment Spj

Where Sno = S.Sno)

So, is there any 'trick' the DBMS can use to reduce the cost of processing all the rows in both tables?

You should understand that 'Exists' only needs to know if a record appears in the nested table. If an index has been created on Sno field in the Shipment table then use it to answer the query, and avoid the Shipment table Disk file I/O altogether. That is, search the index to find if the key of the record exists. If you find it in the index, then you know the full data record also exists in the main data file. You therefore can save on reading (passing) the large data file out on the disk. NB: an index is small as it contains a small number of attributes. Data files are large because they usually contain many large attributes e.g. name, address, comments etc.

This is simplistic just to indicate the point: the optimiser attempts to reduce the number of full 'sweeps' or 'passes' of the disk file.

Another technique of implementing Joins to save on a full Pass of a file is to use 'order' in data.

Note in an unordered list (file) or heap file.

1 ,9, 4, 8, 3, 5

If we search for 3, we may need to process most of the list (file); also, what if the value is not unique? Then we must search all the file to ensure we don't have this situation.

1 ,9, 4, 8, 3, 5, 3

But if we have the list (file) ordered (as in Indexed Sequential files such as ISAM and Btree), then the search does not need to process the entire list (file)

1, 3, 4, 5, 8 ,9 etc

Search for 3, stop when found. Search for 6? Stop after you find 8, as you now know that the record does not exist in the file (return 'not found' message) hence you prevent a full pass/sweep of the file. Here we have used uniqueness and order to limit the search (i.e. pass/sweep)

So, in general, query optimisers use any available techniques to avoid full sweeps/passes of a disk file.

Query Optimisation: 2 Types of Optimisers: Rule Based V's Cost Based

Optimisers can be defined as being of two types:

- Rule based optimisers follow a preset optimisation scheme using factors such as: operator precedence (And, Or, Join, etc) and rules for using available access paths to produce a final query e.g. if an index is defined on the attribute specified in the query then the processor must use it; Or, always implement a 'select of rows' before a Join of tables.
- Cost based optimiser follow similar rules but may work out a number of different query plans and compare costs of each. This is done in real time (i.e. at run time)

Note: Oracle gives a choice of rule or cost based therefore this is a configurable parameter.

Note that an available index doesn't mean efficient execution. A full file scan might be optimal if more than approx 20% is required. **So analysing different paths can be beneficial (cost based). But note extra work by the cost based optimiser (note the benefit if the query is re-executed).**

Recall how a non procedural SQL Select statement is transformed into a procedural sequence of operations to be performed. The optimiser can **manipulate the procedure to obtain different execution orders. This is the basis of query optimisation.**

Note, there may be temporary tables generated during a session (stored in a data cache). Join operations in SQL generate internal temporary tables.

The term **hit rate** refers to the proportion of successful accesses in the total number of accesses on a particular item of storage. It is a critical indicator of performance. Successful access means that the required data was already available in a cache when it was requested. If you store anything in a cache/buffer, to justify its position there it should have a high hit rate. Good hit rates = good performance.

Think about the difference between a DB application in CIT that selects

- where CIT_No = R0001134 or
- where dept = 'Computing' or
- where dept = 'Business and dept = 'Engineering'

Rule V's Cost based optimisers, Indexing and the role of statistics (extra note)

Take an SQL statement:

```
Select *
From Student
Where Dept = '????'
```

A rule for the optimiser when making the QEP might be:

However, if you force a QEP to use an index and you end up reading a large percentage of the records of the table in from the disk, then the cost of reading both the Index and the data file becomes a performance issue due to excessive disk seeks. Think of a wind screen wiper (back and forth) motion.

You should note that an Indexed Sequential file might be physically stored as a B-Tree where the Index and Data are together i.e. the nodes of the Index contains data records. This is usually an indexed sequential file organisation on the primary key column(s).

However, how about any non-primary key column(s) with Indexes defined? All of these are physical Index files separate to the data file. Hence, the dual disk seeks to both Index and then Data files.

We might refine the rule:

So in CIT, Where Dept = 'Business' might result in a different QEP to Where Dept = 'Computing' because the numbers of records for each of the values is different (far more Business than Computing students in CIT). Therefore, it might be worthwhile using the Dept index for Where Dept_code = 'Computing' but not for Where Dept_code = 'Business'.

The problem for rule based systems that they do not reflect the exact cost for the given query at the time of execution. The cost is based on statistics about the records in the tables.
The problem for stats is that they are costly to run as they must access and process the entire data file. Stats are therefore updated periodically. In a dynamic system (lots of updates, inserts and deletes), the stats may be feeding the optimiser 'bad' out of date data resulting in poor query performance. For example, what if 1000 new students Computing students are inserted into the file, the stats will not reflect this change until they are run again. (ref import practical lab).

Cost based optimisers do more work at execution time examining the exact cost of the query. This results in a 'better' QEP, but you must ask as DBA is it worth the extra cost? We should now realise that a complex query involves a long sequence of sub operations (to be performed procedurally, one after the other).

Cost based optimisers examine the interim results of the QEP. If the projected (estimated costs) are different to the actual costs in real time (at run time), then an alternative execution plan can be made.

Important queries, that are run frequently (e.g. by triggers or embedded in application) should be well optimised.

Ad hoc queries (typed in from a terminal) that may never be run again, may not justify the extra work.

Other Access mechanisms:

Besides Indexes, are there any other forms of access mechanism? Yes, for example, Hashing.

'In-memory hash tables, which are used as temporary tables'

Hash tables.

In general file theory there is a Hash file (implemented in some DBMS e.g. Ingres). A hash file uses a mathematical function to take a key value and compute a disk address for the corresponding full data record for that key value.

This is a CPU operation (very fast), so it is highly efficient for key lookup applications such as bank ATM. Note, using an index, you must read the index first (cost), then the data file.

However, hashing as a general system can also be used as an in memory data structure process i.e. the hash function generates a value used in some way by the application code. In MySQL, hash tables are used. Tables are usually files (as in disk files), however, not in this case. In MySQL, they have implemented In-memory (RAM) hash tables. This makes them volatile, and therefore only used for temporary storage purposes.

SQL Dialects, Portability and Migration

'SQL functions are implemented using a highly optimized class library and should be as fast as possible'

SQL functions: many DBMS implement their own functions (i.e. extra to the standard SQL functions like Max, Min, Avg etc); these are optimised to run on the native DBMS;. So the advantage to the user is user friendly, efficient availability of functions, however the problem is that they may not port to (run on) a different DBMS. **This is an SQL dialect and therefore a code portability issue.**

Although SQL is both an ANSI and an ISO standard, many database products support SQL with proprietary extensions to the standard language i.e. non standard elements to the language. Each vendor e.g. MySQL, Oracle etc therefore are said to use different SQL dialects. This gives rise to a common criticism of SQL: lack of cross-platform portability between vendors.
Portability and data migration are closely related. SQL code that is portable means that it'll work on any DBMS, so you can port or migrate your programs to a new system without encountering SQL dialect errors.

Programming trade off: writing portable SQL usually means writing SQL programs that adhere to the SQL standard only (i.e. do not use the proprietary extensions). However in many cases these non standard SQL extensions are very useful (e.g. some custom maths or string function) or good for efficiency (optimised for that DBMS, and hence not using them makes the SQL programming more difficult/inefficient. Hence using non-standard code is a tradeoff between efficiency and portability.

Embedded SQL: Why does it occur? What are the requirements on a system to allow embedded SQL? Aside: what is the difference between embedded SQL and an embedded database?

Dialects/migration from a configuration/installation perspective:

DBMS may have a setting that relates to the version of SQL supported. Many older programs have code that does not adhere to significant SQL standard releases. The administrator must liaison/agree with software developers that all programs that interact with the database adhere to a SQL standard before configuring/installing the DBMS.

Migration/import/export will be dealt with in detail in a later section.

Modular

Modular or layered program refers to the design approach for the software itself. Modular usually is beneficial as it allows a 'plug n' play' approach to the software subsystems of the program. So, in an Open source DBMS, you might be **able to reprogram** the Index subsystem, or the optimiser, or the recovery system etc to suit your requirements; or alternatively you may wish to **remove sections to minimise DBMS code** (see Embedded later). If you edit a C++ program, you will need to recompile it and then link it to the other modules of the DBMS. **Modular software design enables customisation, and tailored DBMS installations.**

The server is available as a separate program for use in a client/server networked environment. It is also available as a library that can be embedded (linked) into standalone applications i.e. used in isolation or in environments where no network is available.
Client/Server networked environment V's Standalone Application with Embedded DBMS

The server is available as a separate program for use in a client/server networked environment. It is also available as a library that can be embedded (linked) into standalone applications i.e. used in isolation or in environments where no network is available.

Client Server V's Embedded?

Standard software architecture is client server. So what is meant by Embedded?

Do not confuse Embedded SQL and embedded DBMS.

In embedded systems, such as in machine controllers (e.g. cars), there is no network, so no client server. The DBMS becomes a library of data management functions included in the actual code of the application. You cannot 'see' the tables on any server; they are hidden 'in the application' e.g. Xampp

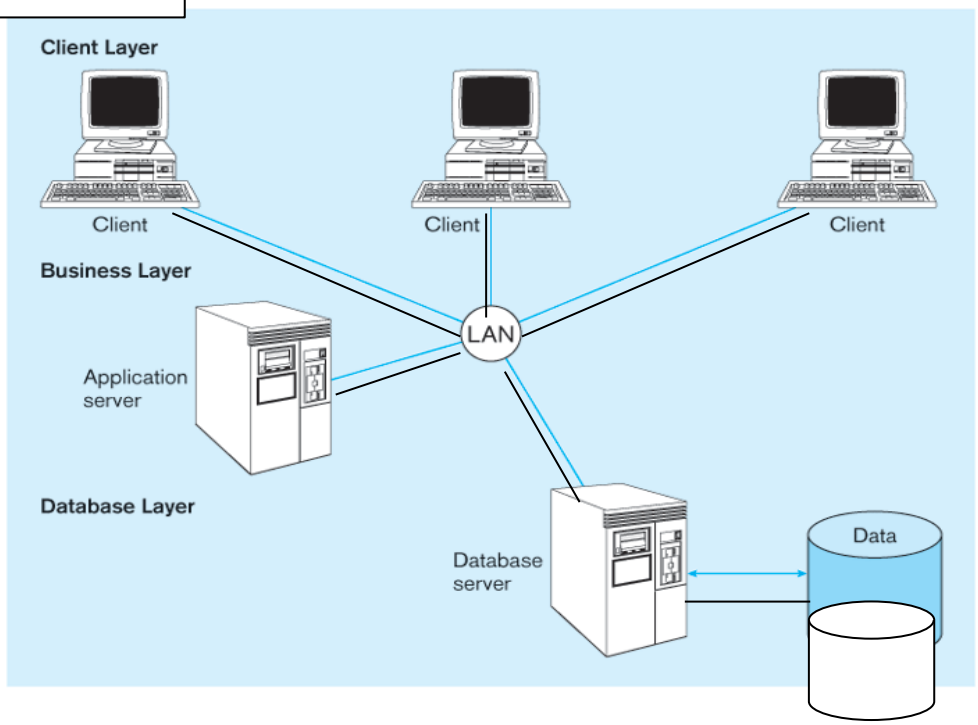
This relates to open source and modular design, where you can edit the code, remove sections of unwanted code, and recompile for a smaller resource demanding DBMS.

This depends on what application the database is being used for. If the DBMS is on a dedicated 'backend' server, then you need a network to allow connection to it from remote 'client' programs. This relates to another term called 'tiered' systems, e.g. 2 or 3 tier system architecture.

In this system, you'd install the database as a separate system; programmers would then have to connect to that DBMS over the network (note, using services such as sockets or APIs such as ODBC). This type of system typically requires a Database Administrator.

However, what about a dedicated system where no other software is required, just the application code and some database services to store data. In this type of application, a trimmed down version of the database code itself is linked and 'embedded' into the application source code. This combined (integrated) piece of software is then sold as a product e.g. an application on a PDA for stock management in a large retail store. Database administrators are not required in this type of system; the

Three-tier architecture



An alternative description of the 3 tier architecture is

Presentation Tier (Client)
The user interface, typically a GUI: responsible for services such as, data input and display e.g. browsing, purchasing, and shopping cart contents. Data input is communicated to the middle layer for processing. In a Web or E-Commerce application, the Presentation Tier is a Web browser, accepting input and displaying static HTML for output display.

Application Tier (Business Logic/Middleware Tier)
The actual processing of the data is controlled here e.g. decisions, complex calculations and sorting etc. Essentially the main processing code for the application. Output formatting may be done here.

Data Tier (Database)
At least one database management system running on at least one server machine. All data management is performed here. Data is 'served to the middleware for processing as required; or accepted for storage. This tier keeps data independent which enables the advantages of scalability and efficient performance.
Modular software with well defined interfaces have the advantages of (plug n' play flexibility, scalability, ease of development/testing etc); Tier architecture (just like modular programming) is intended to allow any of the tiers/modules to be upgraded or replaced independently as requirements or technology change. **An advantage of separation is that each tier can be optimised for its function and operating environment**.

A three tier architecture is effectively spreading the overall processing load over 3 machines. Any tier can be expanded but in data intensive applications the data tier is commonly more complex. This an N-tier architecture. Within the data server element of the overall architecture you may for instance be able to run multiple instances of the database on the same machine (or on different machines). Each instance can be configured specifically for the particular data it stores. This is moving into the area of distributed databases (note the terms federated databases, clouds or clustered databases relating to a co-ordinated set of data servers).

In recent years, many companies have opted for scaling by adding many smaller 'commodity' servers rather than replacing a central large 'top of the market' server. **This is called horizontal scaling**

versus vertical. You scale out, not up. However, you must take full cost into account e.g. electricity, admin, location.

Separation of Application and DBMS might also suit depending on the licensing terms & conditions e.g. if software is charged by CPU then one large server with many CPUs is costly
However, it should be noted that the fastest response and throughput is obtained when a DBMS and application are co-located, thereby removing network issues and protocols (e.g. TCP/IP).

How to handle varied configuration demands? Multiple instances.

Different types of applications require different configurations of the database to run efficiently. Rather than tuning the single database to give acceptable performance to all applications, one solution is to install multiple instances of the DB server, configuring each specifically to suit a limited subset of applications. **Multiple instances are used for load balancing and targeted configuration for optimum performance; Instances can also be used in development and test environments.**

Learning outcomes: Explain Client-Server; N-tier architecture, Adj * disadvantages, Scale out V scale up (horizontal V's vertical)

Database capacity : limits and scalability.
2 ways to compare or gauge capabilities and scalability of a database system:

- **data capacity - limits on number of records (amount of data), number of tables; number of columns, number and range of access mechanisms e.g. how many indexes.**

- **Activity : how dynamic is the system**
  - **How many users can the system handle (in total)?**
  - **How many active users can the system handle (in current sessions)?**
  - **What type of activity is being done: read, write, transactions?**

When you compare database systems, you can look to **whether they are transactional or not**. This is a key indicator of the capabilities of the database. Note, transactions can be a sequence of modifications V's analysis/read only). Another aspect that indicates the capabilities of the database is **the amount of data it can handle.** This is important as the data always grows over time so an organization must plan for future data requirements.

Scalability refers to how much a database can grow (how much data it can handle). A database is made up of tables, each table has many fields (columns) and each table can store many rows. So the **limits and scalability of a database is usually described by the number of tables it can handle, or the amount of storage the table can take up, or the number of rows etc.**

Connectivity:

This refers to how client programs (e.g. written in C++ or Java) can connect and interact with the database system. **Client programs use concepts like sockets, or APIs to connect to a remote server that runs the DBMS**. The database administrator would only be interested in these only from the point of view that they must cater for the needs of the programmers. That is, the DBA needs to know the programmers requirements.

Localization: support for many languages e.g. for error messages; or support for character sets. Note, some character sets are bigger/larger than other. Bigger means a larger set of characters (e.g. A-Z, a-z, 1-9, special characters (!"£$%) etc. **The more characters needed, the bigger the set and the more storage space required for each character stored in the database records**. If there are millions of records all containing a lot of text then choice of character set might be important from an efficiency point of view. **Having a choice of character sets is advantageous as it adds flexibility and scalability** (it allows an organization to grow but using the same base technology e.g. multinational company using particular database system such as MySql).

Therefore the choice of character set is a trade off: efficiency V's flexibility&scalability.
Tools:
A database is used by people e.g. administrators or programmers. Tools provide services to the users to make interaction with the database easier. The more tools the better! Tools include

- SQL program development (i.e. a runtime environment for developing and running SQL code.
- DBMS administration : managing data, users, recovery etc.

Command Line V's Graphic User Interface utilities: Advantages/ Disadvantages:

Adj: User friendly, Powerful: use of drop down menus, icons, window panes gives control over a wide range of DBMS functions. Full screen editing allows faster program development / debugging

Disadj: hidden from native commands, Non transferable skills (from one DBMS to another). SQL direct to system tables is transferable, Resource hungry: graphics

Note: some Database Administrators (DBA) may not allow GUI on the server machine; so you may need to use the command line. Developers/programmers (i.e. not DBA on production server) prefer the GUI

Other tools may include: Modelling ; Data Migration (import/export) etc.

Memory management:

Recall two type of memory

1. External: 2nd long term storage, Disk : slow, non volatile, long term storage, I/O bottleneck.
2. Internal : RAM, Main memory, Cache, buffers : fast, volatile, limited capacity

Commonly, memory management refers to internal memory; while management of 2nd storage is referred to as File or Data management.
External Memory/Storage Management: Disk, Data, File Management.

Tables in many database systems can have different physical organisations (or implementations) on the disk e.g. Btree (a dynamic auto adjusting index structure effectively load balancing the index across the tree), ISAM (fixed tree structure for static lookup tables) or hash (fast direct lookup using the record key value) and others. In MySql these are called storage engines, but in general they are called file organisations. A database administrator might find it useful to change the file organisation and process the table in a different way i.e re-organise the table and therefore process it using a different internal (sub) program e.g. change from a Btree to a Hash file. Note: Ingres uses a hash table on disk, Mysql uses In Memory hash tables: same theoretical concept, two implementations.

In addition, in large data systems, it is beneficial to separate the data by its function for better management e.g. recovery data, log data, user data etc

Why & how might we manage a relatively small amount of RAM?

As we work through this section, you should keep in mind that particular pieces of data are important, e.g. for the recovery system or for concurrency system. The data may be in main memory buffers or memory caches for I/O, for reuse of critical data (i.e. by concurrent users or repeat processing).

Note that optimising use of critical data or maximising sharing of data can result in improved performance and throughput i.e. increased concurrency. e.g. Root node of index.

A given database system may have its own way of dealing with memory (Oracle, Ingres, MySql etc)

Basically we might start by thinking we can have both

- system data and
- application data.

Once we read data into main memory it is available to running processes until it gets swapped out if the memory get full. The unit of disk I/O is a page, and the page replacement algorithm employed by the system may have an effect on performance. Least recently used (LRU) i.e. remove the page(s) that haven't been accessed in a while, or Least frequently used (LFU) remove page(s) that aren't used a lot. The DBA should be aware that data in memory is stored in pages; data on disks is stored in files.

Database systems will also categorise data according whether it is

- local per connection/session : NB memory is allocated PER connection
- shared or global, to all connections/sessions

From an administrators point of view it is critical to determine what memory is fixed and preallocated and what memory is dynamic and allocated per session.

It is crucial therefore that the administrator is aware of the number of possible connections (concurrency) and the type of activity generated that requires memory. Note it is memory to be allocated per user/connection multiplied by the number of connections.

Too many connections, each taking up memory, results in the database using up all available memory. Once this happens the swapping algorithm is called excessively resulting in system thrashing, i.e. more cpu time spent executing the page swaps than work for the database. Effectively the database system slows to an unacceptable speed of processing and possibly crashes.

Most databases refer to this memory as internal caches.

We can examine memory/caches in more detail on a functional basis. What types of data do we have by function?

- Catalog cache aka meta data / data dictionary. Data held in RAM for fast, frequent lookup
- Data cache : application data, i.e. rows of table(s) e.g. student data
- Log buffer (or log cache): data for recovery system
- I/O buffer, data temp store in channel to database on disk

Data Dictionary or Database Catalog.

The data dictionary is not one thing. It is a collection (set) of database tables that store all the data required by the DBMS to manage the database. There are tables for Users, Connections, Transactions, Databases, Tables, Indexes, etc. This is called **meta data** i.e. it is data that describes other data.

This is where all the information required by the system is stored. Each system component may need to access this Dictionary to function e.g. security needs lists of users, list of database, tables within a database, column definition for each table, indexes and operations allowed by users on data etc Integrity needs data types and constraints e.g. Salary < 100,000. DML processors need info on how tables are physically stored, their structure etc.

Every database program that references a table name requires work in the back ground to check the existence of that table name, and then check the attribute names and data types referenced by the user program e.g Insert into Student (Sno, Sname, DOB) values ('S22', 'J.Murphy', '1/1/76')

Think about the background work required to handle tables related via a referential integrity rule? (discuss in class)

Accessing the Dictionary/catalogue is a potential critical point for system efficiency.

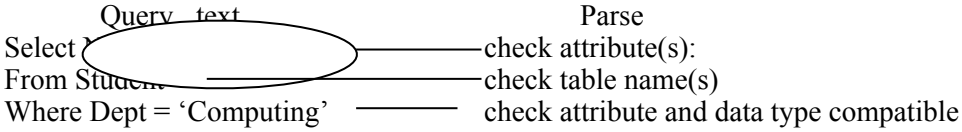
Catalog: some examples of the use of meta data stored in the data dictionary

Security: each user connection is checked initially, but some data must be retained for checking for each individual data access issues during the connection lifetime e.g. note a transaction is a sequence of SQL operations.

Query Parsing: table headers i.e. name, column names, data types. See below.

Query Optimisation: read catalog for what indexes exist, stats on size of tables etc

Query Parsing:



In the Data cache, we might store the actual data set returned by a query, and or we might store the SQL query itself. This also is a factor in query optimisation, e.g. If a query is rerun (e.g. by a refresh) or an similar query received, the server will retrieve the results from the query cache rather than parsing and executing the same query again.

Recall the terms B-Tree index, Access mechanism and QEP from earlier.

The system generates a Query Execution Plan, possibly optimised, for every access. The system may store the QEP or the Query text or both. The DBA can view the QEP using a utility(program) called 'Explain' implemented in most DBS. Saving the QEP rather than the query text will save on running the optimiser again, however it takes no account of changes in the condition of the database e.g. after deletes, inserts etc.

Please note the following web site: Please note the following web site: http://www.synametrics.com/SynametricsWebApp/WPTop10Tips.jsp

Our notes and labs should help explain some (or all) of the issues raised in this article. That is, you should be developing an understanding of how logical table design (normalisation) and SQL have implications for how the DBMS organises the data (physical design).

We can think of this as, Database 3 examines the DBMS administrators perspective of Database 1 and database 2 modules (table design & SQL)

Our notes and labs should help explain some (or all) of the issues raised in this article. That is, you should be developing an understanding of how logical table design (normalisation) and SQL have implications for how the DBMS organises the data (physical design).
Learning outcome: Have you attained the learning outcomes detailed at the start of this section?

Can you

- Discuss the link between memory management and concurrency (multiple users accessing shared data)
- Explain what is meant by load balancing
- Describe the term hit rate and comment on it as a factor in performance.
- Storing Query data (or QEP) may be useful for improved throughput, explain?
- A DBA faced with a choice of optimizer type faces a dilemma, explain?
- Explain how the same query may have different executions (or costs of executing) .
- Explain, using an example, the basis of query optimisation.
- What is meta data, and why is accessing meta data in the data dictionary critical for system performance?

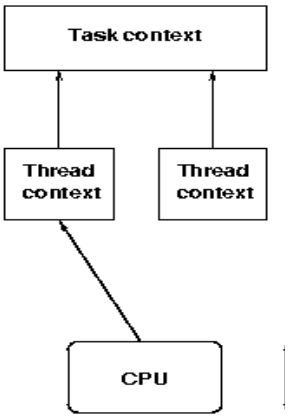
- What is an SQL dialect? Why it is important for issues such as portability and data migration?
- Explain using examples why a DBA must know about software developer (programmer) requirements during install/config.
- Describe an N-tier architecture for a database system? IS this recommended in all cases?
- What is thread and what is its role in making a database robust (or fault tolerant) (or load balanced)
- In a database there is far more data that can fit in RAM (processing memory); Describe how 'good' memory management is essential for efficient DBMS.
- Explain why multiple instances of a database might be used?
- Explain what is the role of the administrator in Data/File Management on 2nd storage devices.
- Why not create indexes on all attributes? Should a QEP always use an index if available?

MySQL Top 10+ SQL Performance Tips

https://wikis.oracle.com/pages/viewpage.action?pageId=27263381
http://ronaldbradford.com/blog/
http://www.mysqlperformanceblog.com/2008/11/24/how-percona-does-a-mysql-performance-audit/
http://dev.mysql.com/doc/refman/5.0/en/server-system-variables.html
http://dev.mysql.com/doc/refman/5.0/en/explain.html

Appendix:

Extra notes (reference only, not for examination): In a basic multitasking OS, the single CPU must switch its entire context from one program code to another. This gives the illusion of simultaneously running many programs. As there is a cost to perform a context switch, it must be managed efficiently to be of overall benefit to the system.

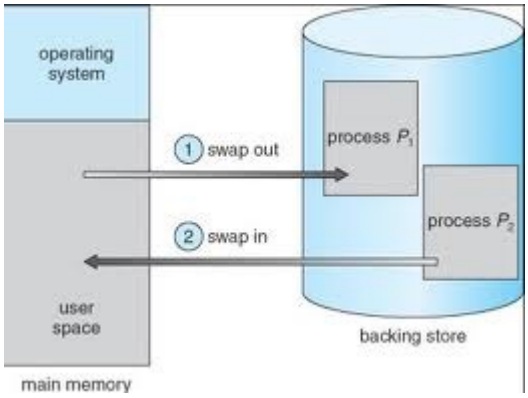


http://www.6502.org/users/andre/icapos/mp.html

http://en.wikipedia.org/wiki/Context_switch

In computing, multitasking is a method where multiple tasks, also known as processes, are performed during the same period of time. The tasks share common processing resources, such as a CPU and main memory. In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved. Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs.

As multitasking greatly improved the throughput of computers, programmers started to implement applications as sets of cooperating processes (e. g., one process gathering input data, one process processing input data, one process writing out results on disk). This, however, required some tools to allow processes to efficiently exchange data.



Swapping context in/out to disk would be slow to the point of useless. Threads manage Main Memory.

Threads were born from the idea that the most efficient way for cooperating processes to exchange data would be to share their entire memory space. Thus, threads are basically processes that run in the same *(main process)* memory context. Threads are described as lightweight because switching between threads does not involve changing the *(full)* memory context. (my italics: BT)

Note: different tasks are running different program instructions (and registry values etc), so to swap the CPU still requires some context change to move from one task to another.

WEDNESDAY, JUNE 23, 2010 HTTP://TROELSARVIN.BLOGSPOT.IE/

Separation or co-location of database and application

In a classical three-tier architecture (database, application-server, client), a choice will always have to be made: Should the database and the application-server reside on separate servers, or co-located on a shared server?

Often, I see recommendations for separation, with vague claims about performance improvements. But I assert that the main argument for separation is bureaucratic (license-related), and that separation may well hurt performance. Sure, if you separate the application and the database on separate servers, you gain an easy scale-out effect, but you also end up with a less efficient communication path.

If a database and an application is running on the same operating system instance, you may use very efficient communication channels. With DB2, for example, you may use shared-memory based inter-process communication (IPC) when the application and the database are co-located. If they are on separate servers, TCP must be used. TCP is a nice protocol, offering reliable delivery, congestion control, etc, but it also entails several protocol layers, each contributing overhead.

But let's try to quantify the difference, focusing as closely on the query round-trips as possible.

I wrote a little Java program, *LatencyTester*, which I used to measure differences between a number of database<>application setups. Java was chosen because it's a compiled, statically typed language; that way, the test-program should have as little internal execution overhead as possible. This can be important: I have sometimes written database benchmark programs in Python (which is a much nicer programming experience), but as Python can be rather inefficient, I ended up benchmarking Python, instead of the database.

The program connects to a DB2 database in one of two ways:

- If you specify a servername and a username, it will communicate using "DRDA" over TCP
- If you leave out servername and username it will use a local, shared memory based channel.

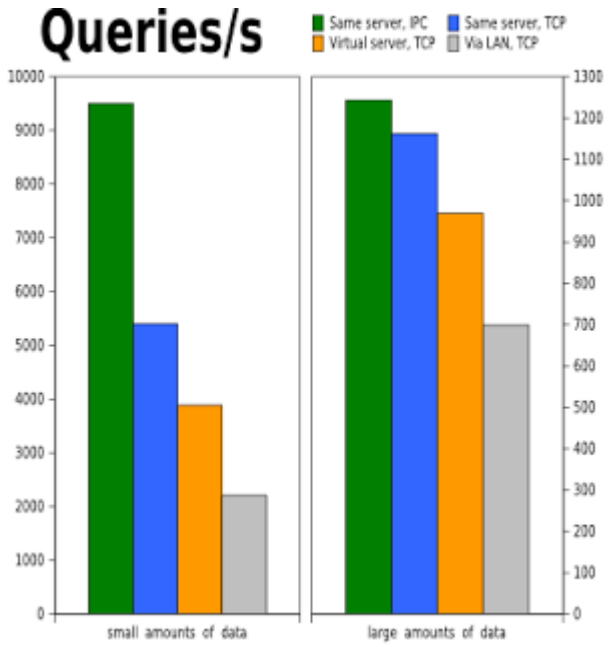
After connecting to the database, the program issues 10000 queries which shouldn't result in I/Os, because no data in the database system is referenced. The timer starts after connection setup, just before the first query; it stops immediately after the last query.

The application issues statements like `VALUES(...)` where `...` is a value set by the application. Note the lack of a `SELECT` and a `FROM` in the statement. When invoking the program, you must choose between *short* or *long* statements. If you choose *short*, statements like this will be issued:
`VALUES(? + 1)`
where `?` is a randomly chosen host variable.
If you choose *long*, statements like
`VALUES('lksjflkvjw...pvoiwepwvk')`
will be issued, where *lksjflkvjw...pvoiwepwvk* is a 2000-character pseudo-randomly composed string.
In other words: The program may run in a mode where very short or very long units are sent back and forth between the application and the database. The short queries are effectively measuring latency, while the long queries may be viewed as measuring throughput.

I used the program to benchmark four different setups:

- Application and database on the same server, using a local connection
- Application and database on the same server, using a TCP connection
- Application on a virtual server hosted by the same server as the database, using TCP
- Application and database on different physical servers, but on the same local area network (LAN), using TCP

The results are displayed in the following graph:



Click on figure to enlarge; opens in new window

Clearly, the the highest number of queries per second is seen when the database and the application are co-located. This is especially true for the short queries: Here, more than four times as many queries may be executed per second when co-locating on the same server, compared to separating over a LAN. When using TCP on the same server, short-query round-trips run at around half the speed.

The results need to be put in perspective, though: While there are clear differences, the absolute numbers may not matter much in the Real World. The average query-time for short local queries were 0.1ms, compared to 0.8ms for short queries over the LAN. Let's assume that we are dealing with a web application where each page-view results in ten short queries. In this case, the round-trip overhead for a location connection is 10x0.1ms=1ms, whereas round-trip overhead for the LAN-connected setup is 10x0.8ms=8ms. Other factors (like query disk-I/O, and browser-to-server roundtrips) will most likely dominate, and the user will hardly notice a difference.

Even though queries over a LAN will normally not be noticeably slower, LANs may sometimes exhibit congestion. And all else being equal, the more servers and the more equipment being involved, the more things can go wrong.

Having established that application/database server-separation will not improve query performance (neither latency, nor throughput), what other factors are involved in the decision between co-location and separation?

- Software licensing terms may make it cheaper to put the database on its own, dedicated hardware: The less CPUs beneath on the database system, the less licensing costs. The same goes for the application server: If it is priced per CPU, it may be very expensive to pay for CPUs which are primarily used for other parts of the solution.
- Organizational aspects may dictate that the the DBA and the application server administrator each have their "own boxes". Or conversely, the organization may put an effort into operating as few servers as possible to keep administration work down.
- The optimal operating system for the database may not be the optimal operating system for the application server.
- The database server may need to run on hardware with special, high-performance storage system attachments. - While the application server (which probably doesn't perform much disk I/O) may be better off running in a virtual server, taking advantage of the flexible administration advantages of virtualization.
- Buying one very powerful piece of server hardware is sometimes more expensive than buying two servers which add up to the same horsepower. But it may also be the other way around, especially if cooling, electricity, service agreements, and rack space is taken into account.

- Handling authentication and group memberships may be easier when there is only one server. E.g., DB2 and PostgreSQL allows the operating system to handle authentication if an application connects locally, meaning that no authentication configuration needs to be set up in the application. (Don't you just hate it when passwords sneak into the version control system?)
- A mis-behaving (e.g. memory leaking) application may disturb the database if the two are running on the same system. Operating systems generally provide mechanisms for resource-constraining processes, but that can often be tricky to setup.

Summarized:

Pro separation	**Con** separation
Misbehaving application will not be able to disturb the database as much.	Slightly higher latency.
May provide for more tailored installations (the database gets its perfect environment, and so does the application).	Less predictable connections on shared networks.
If the application server is split into two servers in combination with a load balancing system, each application server may be patched individually without affecting the database server, and with little of no visible down-time for the users.	More hardware/software/systems to maintain, monitor, backup, and document.
May save large amounts of money if the database and/or the application is CPU-licensed.	Potentially more base software licensing fees (operating systems, supporting software).
Potentially cheaper to buy two modest servers than to buy one top-of-the-market server.	Potentially more expensive to buy two servers if cooling and electricity is taken into account.
Allows for advanced scale-out/application-clustering scenarios.	Prevents easy packaging of a complete database+application product, such as a turn-key solution in a single VMWare or Amazon EC2 image.

Indsendt af Troels Arvin kl. 02:53 2 comments: Links til dette indlæg
Etiketter: database, performance

http://www.monash.com/uploads/explosion-database-choice.pdf
http://pages.cs.wisc.edu/~jignesh/publ/turboSSD.pdf
http://www.dbms2.com/2013/02/21/one-database-to-rule-them-all/#more-7750