Data Management Systems

1 Intro

1.1 Physical and logical independence

User does not care about how the data is stored, and what format it's stored in memory - is it on RAM, is it NTFS, whatever.

Logical independence means the data is the data, and You can have different views of it.

I guess physical independence is nuts and bolts of actual storage, logical is how the data is presented to the user.

1.2 Query Optimization

Equivalent results may be achieved by different operators, and some are more efficient than others e.g. join and select v.s. select and join.

1.3 Data Integrity

Enforcement of legal values, so the database is intact/coherent.

1.4 Access Control

1.5 Concurrency Control

1.6 Recovery

1.7 Data vs Query shipping

1.8 Cons/Pros of Shared-Nothing

Pros: easy to maintain and to scale, sure.

Ideal when data can be sharded - a shard is a horizontal partition. Yeah I suppose in a horizontal case there won't be any need to do data shipping.

Basically works well as long as each node can function as it's own little island - this is broken if the data cannot be horizontally partitioned for some reason or if there are a lot of updates to the entire table, because operations that involve the entire table will probably do data shipping and that's no bueno.

1.9 Shared Memory

Not actually shared memory, but this is an abstraction that allows the node to pretend as if though all the nodes are sharing memory - so I guess the advantage here is that if a table is in some node's memory, it might as well be in Your memory. Simplifies design, and I guess if hardware allows for this it ought to be fast.

1.10 Shared Disk

Is more cloud oriented - the idea is that each node has access to some sort of giga-sized storage that contains everything, and local storage becomes more of a cache.

1.11 Exericses - Process Models

1.11.1 Thread v.s. Process

Process is a task in an OS - can basically think of it as an application. It's got it's own virtual memory, privileges, access to stuff, all that.

A process can have threads, these are I guess subtasks for the process. The main point here is that the threads share resources of the process - files, memory, whatever. So communicating between threads is fast, communicating between processes is slow and hard lol.

A DBMS client is, well, a client - it's a piece of software belonging conceptually to the user of the DBMS, and the client interacts with the DBMS through some established API.

The thing that the client interacts with is a worker - each client gets a worker. Picture a loop listening for connections on a socket. You can think of the DBMS server as the whole DBMS from the client's perspective - a client sends a request to a server, and within the server a worker is created to handle the client's requests.

1.12 Process per Worker

So the OS allocates a process for each conceptual worker.

Advantages are that it's easy to implement - every time You need something new, throw a process at it. OS handles parallelism, for better or for worse. Debugging processes is easier, since there's lots of support for that sort of thing from OS perspective.

Downsides are that DMBS workers need to communicate - locks for example. Not ideal for processes. Plus a process has it's own virtual memory, security stuff, basically overhead, so if You have a lot of workers it won't be super efficient.

1.13 Thread per worker

So You get a process for the DBMS I suppose, and within that process each worker gets a thread.

Advantage is that this is more lightweight, sharing is easier.

Disadvantage is that it can become a real mess, and You get to maybe do Your own management of resources instead of it being handled by the OS.

2 Storage Management

2.1 An Evaluation of Buffer Management Strategies for Relational Database Systems

a

3 Storage Systems

a