

Scalable Systems and Data

(70022)

1.1 - Database Storage Layer

This lecture covers the DBMS layers, storage hierarchy as well as the role disks takes in the DBMS. The DBMS layers are as follows, with the query going into the first layer, and storage being the final layer. Note that the last two layers (buffer management and disk space management are typically done by the OS).

1. **query optimization and execution** tries to reorganise the query to execute efficiently
2. **relational operators**
3. **file and access methods** understand which files need to be accessed and indexes we can use
4. **buffer management** handles reading disk pages and buffering in main memory for fast access
5. **disk space management**

An example of this could be a search engine, which is much simpler than a general DBMS - albeit having similar layers;

1. search string modifier
2. ranking engine
3. query execution
4. buffer management
5. disk space management

This is simpler than a DBMS as it can simply use the OS for the bottom two layers, there is typically no concurrency (nor any need for transactions, being mostly read-only), and typically has hard-wired queries. The ranking engine and query execution is a simple DBMS.

DBMS versus Using OS

An important question is why we don't simply use the OS. The layers of abstractions can be useful, but we have a lot of knowledge on how to access the data. In addition, the OS can often get in the way of the DBMS - it has some idea on what to query and what files to touch, and knows more about the OS regarding future access, which can be exploited. A DBMS needs to do things its own way, for example specialised pre-fetching with the knowledge of future access. Additionally, if we control the buffer management, we can also control the replacement policy, likely with something better than the OS. With more control over the thread / process scheduling, the DBMS can achieve a more optimal execution of the workflow as the DB locks aren't going to conflict with the OS locks (high contention). There's also control over flushing data to the disk, including writing log file (important for recovery), and shouldn't be left to the OS.

Disks and Files

Today, disks are still the go-to storage medium for large amounts of files, and have become fairly affordable (not as affordable as tape for archival storage, but much cheaper than other media, such as SSD or main memory). Unlike other media, disks have mechanical parts leading to differences access patterns or behaviour - the time to access a piece of data is affected by **where** the data is on the disk. A lot of databases today are still on disks, as it's cheap with a reasonable access time - typically the

1 millisecond for a 4KB page. As such, the key to lower I/O cost is to reduce the delays caused by seek and rotation. Additionally, in shared disks, most of the time is spent waiting for access to the arm.

The concept of the next block is as follows;

- blocks on the same track, followed by
- blocks on the same cylinder
- blocks on adjacent cylinder

We can't control where we write on the disk - that's controlled by the device driver. Typically, if data is written together, it will also be read together. Defragmentation is disk optimization, as data can be spread out all over a disk for a given file, leading to slower read times - this is done by putting all the pieces of a file closer together.

Note that an adjacent block doesn't necessarily mean physically adjacent. Data can be physically spread out over a disk but in a certain pattern that can lead to near sequential access. The adjacent blocks can be the blocks under the disk head after rotating during settle time.

In general, memory access is much faster than disk I/O (roughly 1,000×), and sequential I/O is faster than random (roughly 10×).

The lowest layer of DBMS manages the space on the disk and higher levels can call this layer to allocate / de-allocate a page, or read / write a page.

Summary

In general, the key for storing data on a disk is to store data together if it's queried together. Random access should be avoided, preferably use sequential access. Data structures should be aligned for page size - for example, if a data structure were to have a few bytes in the next page, an additional page would have to be retrieved, despite mostly being irrelevant.

1.2 - Main Memory Indexing

The general trend is that we have more main memory as time goes on. The hardware trends show that CPU speed and main memory capacity doubles every 18 months, however memory speed only grows by 10% per year.

This means that many databases, typically OLTP, can fit into main memory; OLAP is still in the order of petabytes, if not more. Memory access has become the new bottleneck for main memory databases. There is no longer a uniform random access model (NUMA), meaning that we can no longer assume that accessing each piece of data in memory takes the same amount of time. Cache performance has become crucial.

The memory hierarchy is as follows. Note that a cache **line** is the smallest unit that can be retrieved from the cache, and data structures should be aligned to a line in a similar way to pages.

- CPU (registers)
- L1 cache, takes 1 cycle, 8-64 KB, 32 bytes per line
- L2 cache, takes 2 - 10 cycles, 64 - 128 bytes per line
- TLB, takes 10 - 100 cycles, 64 entries / pages

The cache performance is crucial, similar to the disk cache / buffer pool, however the DBMS doesn't have direct control of this.

Improving Cache Performance

The primary factors are the cache capacity and data locality, the former is a given and can't be changed, whereas we can do something about the locality. An example with non-random access, such as a scan or index traversal is by clustering data structures to a multiple of a cache line, as well as squeezing more useful data into a cache line. On the other hand, with random access, such as in a hash join, the data should be partitioned to fit in cache / TLB. CPU is often also traded for memory access, such as compression (requires more CPU processing, but reduces storage usage).

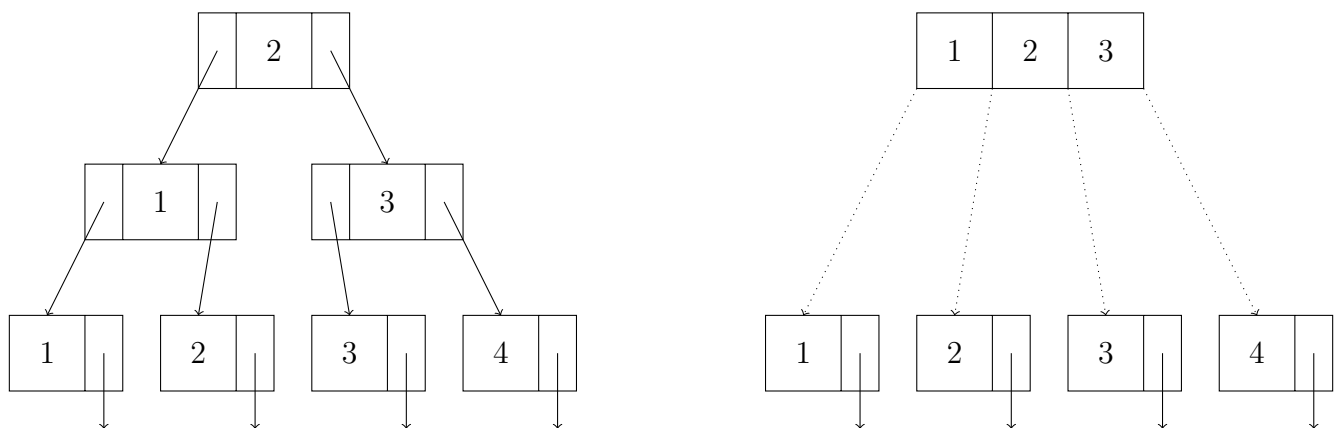
Trees

The lecture then goes through an example of a tree index, where each node holds 2 entries. Each node can point to three other nodes, where the values are either below, in between the two values, or higher than both.

The B+ tree is quite similar, but all leaf nodes are connected. This allows for faster execution of range queries, as the nodes are connected without traversing up and down the tree. The **order** is the minimum number of keys or pointers in a non-leaf node and the **fanout** of a node is the number of pointers out of the node. These have the following properties;

- balanced - leaves are at the same level, leading to predictable performance when traversing down the tree (same for every node)
- every node, other than the root, must be at least half full meaning it becomes balanced
- searching is $\log_d(n)$, where d is the order and n is the number of entries
- insertion involves finding the leaf to insert to, splitting if the node is full and adjusting index accordingly; has a similar cost to searching, have to find all the way down, and may have to split all the way back up
- deletion is similar, find leaf node, delete, and merge neighbouring nodes if required (not half-full)

Cache sensitive search trees can be thought of as a B+ tree that has been optimised specifically for main memory. The basic approach for this is to improve locality, with each node of the tree fitting into an L2 cache line, as the penalty of an L2 miss is significantly higher than that of an L1 miss, and can fit more nodes in L2 than L1. Keys are fixed length (from variable length) by the use of dictionary compression, letting us know the size of a child node. Child pointers are also eliminated; previously we had multiple pointers to point to each of the child nodes, however we now only need one pointer to a single child node as we know the size.



In the example above, we assume a cache line size of 24 bytes, a key size (and pointer size) of 4 bytes. The B+ tree on the left is 2-way, with 3 misses, and the CSS tree is 4-way, with only 2 misses.

CSS has the best search / space balance; second best search to hash, which has poor space, and also second best space to binary search, which has poor search. The space taken is roughly half that of a

B+ tree. However, this cannot support dynamic updates as the fan-out and array size must both be fixed.

A **CSB+** tree addresses this. Children of the same node are stored in an array / node group and the parent only has a single pointer to the child array. This has a similar search performance to the CSS tree, and has good update performance if no split occurs. Splits are still required as we still have a maximum capacity of an array, which requires allocating new memory.

A variant of this is a CSB+ tree with segments; the child array is divided into segments, typically 2, with one child pointer per segment. This improves split performance, but worsens search performance. Another variant is a full CSB+ tree, which is a CSB+ tree with a pre-allocated children array, obviously requiring more space but is good for search and insertion (no more memory needs to be allocated, as it's all allocated). It's important to note that none of these are as **flexible** as B+ trees, but perfectly fine for certain workloads.

The general performance is as follows, for search, CSS is fastest, followed by full CSB+ (joined with CSB+), then followed by CSB+ with segments, and finally B+. On the other hand, with insertion, B+ has the best, roughly equal to full CSB+, which is followed by CSB+ with segments, then CSB+, and finally CSS. Generally, full CSB+ is ideal if space isn't a concern, CSB+ (and with segments) is ideal if there are more reads than insertions, and finally CSS is best when read-only.

Cache Conscious Join Method

Typically, in vertical decomposed storage, what we want to do when we join in main memory is to partition a base table into m arrays, where m is the number of attributes. Variable length fields should also be converted to fixed length fields via the use of dictionary compression. For example;



Each array contains a pair of OID (uniquely identifies an entry) and value for the i^{th} attribute. Reconstruction is a simple array access in this case. Joins are much more efficient, as we no longer have to read all the data.

The existing equal-join methods;

- sort-merge
 - one of the relations will not fit in cache, likely meaning we have to read into cache multiple times (take smaller of two relations in main memory)
- hash join
 - bad if the inner relation doesn't fit into cache
- clustered hash join
 - one pass to generate cache-sized partitions, take each partition and join it with the other relation, and so on, best of the three solutions, but can be bad if the number of partitions exceed the number of cache lines / TLB entries - can lead to cache thrashing

This can be addressed with **radix join**, which first partitions one of the relations and ensure that we partition it into partitions such that we have matching partitions based on B bits of the attribute. Matching partitions are joined, nested-loop can be done for small (≤ 8 tuples) partitions or a hash join for larger partitions, which is \leq L1 cache, L2 cache, or TLB size (with L1 being ideal). This avoids thrashing, compared to clustered hash join. Saving these cache misses outweighs the cost of performing extra partitions.

Conclusion

The cache performance is important and will become more important as main memory grows. The key to this is to cluster data (into a cache line); data that is read together should be written and stored together. Irrelevant data should be omitted, including pointers and the use of vertical decomposition. Partitions should be done to avoid thrashing.

Spatial Data

Spatial data is data with any three dimensions; such as points. It has queries such as range queries, nearest neighbours, etc. Objects near each other should be stored on the same disk page / cache line, however, there isn't an ordering on data in three dimensions (two objects that are close in 1 dimension could be far apart in another).

This then goes over an example with reducing computation. In the example, checking a range on non-uniform (spatial) partitioning is expensive, as there are many irrelevant points. The computation is reduced by using several grids of varying detail; if the range covers the majority of a cell it's checked on that level of detail, if not, it's checked on a lower level of detail.

October 14 - Live Lecture

Learned indexing combines machine learning with indexing. The same concept of an index applies, but machine learning models aid to accelerate this lookup, for example by learning distributions. These models are typically small and simple, such as interpolation - however, they understand distributions better than existing structures such as B trees. The primary challenge is that these models need to fit into main memory, or even the cache to be faster.

The fundamental problem of a join is to join two relations on a shared attribute. If one of the iterations is small, you can iterate over the smaller relation and lookup the related element in the larger relation. There are typically two phases in a join, with a build (scanning through the smaller relation and building a hash table), and a probe phase where the second input relation is scanned, and the hash table of the first relation is probed for the matching tuple. The problem with this is when the hash table is large, leading to a large number of cache misses. One solution is to ensure the hash table can be broken into chunks that fit into cache. However, if there are too many partitions, the build phase involves many writes into different partitions, leading to many writes into different locations (thus leading to many TLB misses). The radix join addresses this by iteratively partitioning into cache.

The data in a B+ tree exists in the leaves. Any other numbers in internal nodes only exist to help guide the search.

An adjacent block doesn't necessarily mean a short / small seek distance. The abstract concept of an adjacent block could also be called a temporally close block; could be short seek distance or based on rotation. Note that the platter is rotating constantly, and by the time the head has finished moving, a different block would be under the head, not necessarily one that is physically nearby. If data is written together, it will be stored on adjacent (the abstract concept) blocks, which may still be spread over the disk physically.

When there's indexing in main memory, the performance is so much faster, the index needs much fewer instructions, otherwise it may be faster to just scan. On the other hand, on disks, the performance is relatively slower, allowing for more complex indices. This applies even for high-throughput devices.

DBMS doesn't typically bypass the OS, at least not without a custom OS or specific hardware. However, it organises its files or data separately, not leaving it to the OS, as well as handling its own cache and database pages (managed by the database system itself). Typically, database systems store the data in and build on the OS filesystem. You can't force the disk to read / write in a specific physical location, but you can help it write in one go.