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

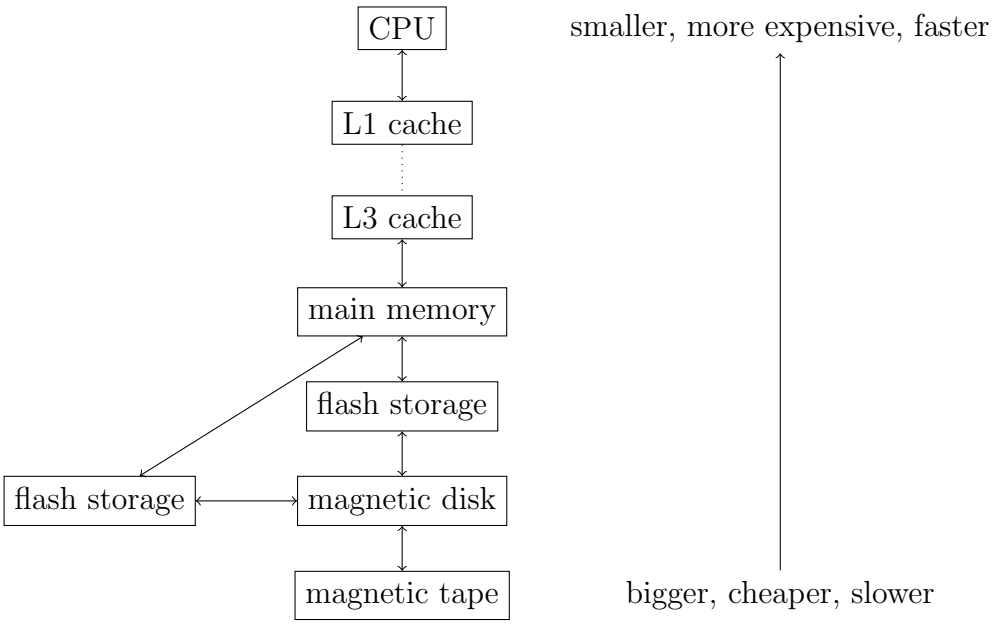
Today, disks are still the go-to storage medium for large amounts of files, and have become fairly affordable (not as affordable as tape for archival storage, but much cheaper than other media, such as SSD or main memory). Unlike other media, disks have mechanical parts leading to differences access 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

data is read from the disk onto main memory (buffered), and changes are written back to the disk from the buffer.

High-end databases are in the petabyte range, therefore using main memory would be extremely expensive. In addition to that, main memory is volatile - meaning the data is lost if power goes out. However, main-memory databases do exist for smaller sized, performance optimised systems, in which volatility is tolerable.

Flash storage isn't commonly used in DBMS as main storage (due to the cost), but can also be used as an accelerator or enabler, in the form of a specialised cache, etc.

The storage hierarchy is as follows. Note that flash storage can either be used to the side of magnetic disk (for a specific portion of the files), or as a buffer on top (before main memory). *Jim Gray* has an analogy for how far away the data is (registers being knowledge in your head, etc), illustrating the scale of access time.



Disks are still the main secondary storage device of choice, with the main advantage over tapes being the ability to perform random access, rather than purely sequential. Data is stored in units in **disk blocks** or **pages** (may be used interchangeably), typically in the order of kilobytes and occasionally megabytes. Unlike RAM, the time to retrieve / write a disk page varies depending on the location, with a tremendous impact on time to retrieve, hence relative placement of pages on a disk has a major impact on performance. The lecture then goes over the anatomy of a disk;

- spindle and platters spin between 5,000 - 15,000 RPM (with physical limitations)
- arm assembly has multiple disk head, one for each platter - the tracks under the heads form a cylinder
- only one head reads or writes at once

The time to access a disk block has three elements;

- **seek time** time to move the arm to the correct position (in and out, towards the spindle)
- **rotational delay** arm assembly in a fixed position, block needs to be rotated under head
- **transfer time** actual time to read data from disk into main memory

The time for a smaller number of cylinders traversed is dominated by the seek time (head positioning). In modern disks, short seeks are dominated by the **settle time**, which gets larger with increased disk track density.

Generally, the seek time and rotational delay dominates, with the former being in the range of 1 to 20ms, and the latter being in the range of 0 to 10ms. On the other hand, the transfer rate is less than

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 $1,000\times$), and sequential I/O is faster than random (roughly $10\times$).

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).

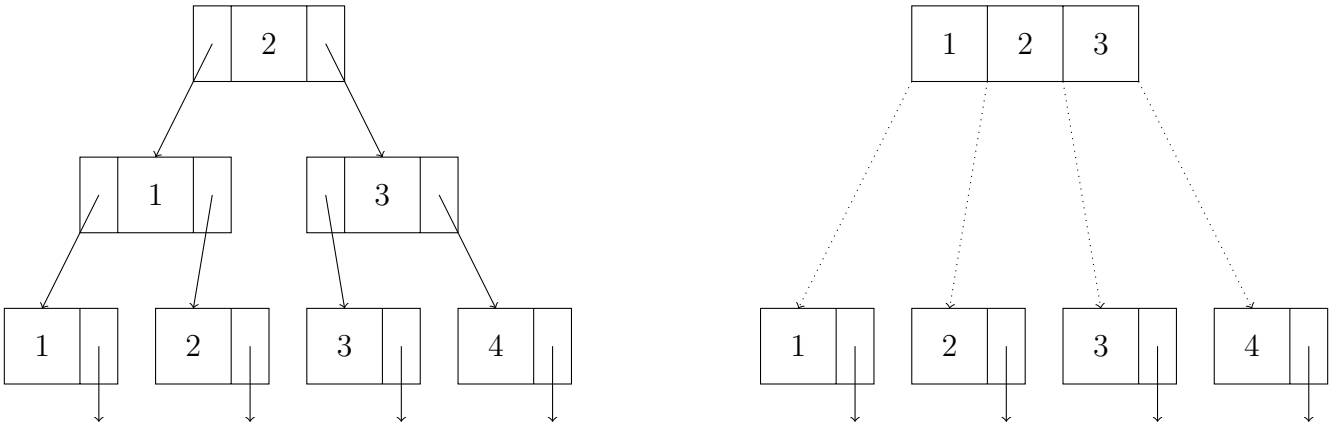
Trees

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.

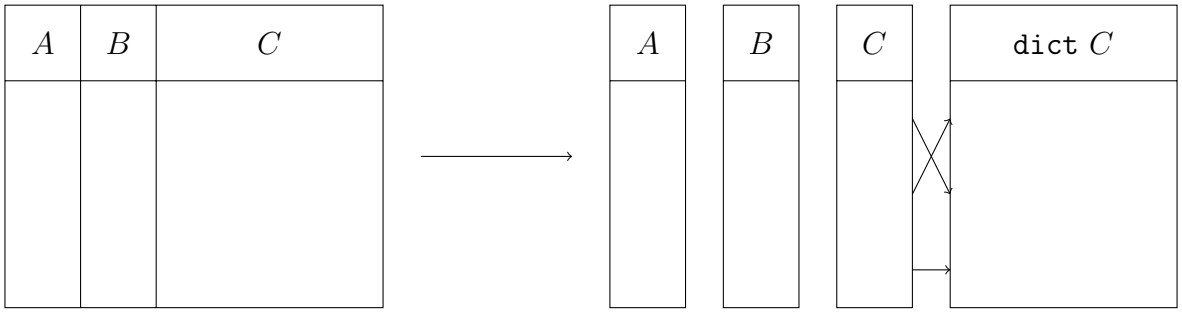
A **CSB+** tree addresses this. Children of the same node are stored in an array / node group and the parent only has a single pointer to the child array. This has a similar search performance to the CSS tree, and has good update performance if no split occurs. Splits are still required as we still have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

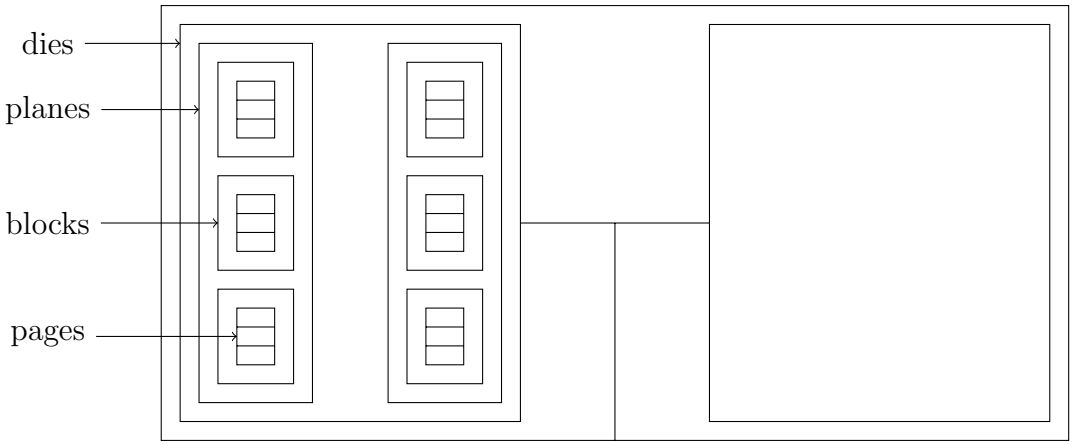
An adjacent block doesn't necessarily mean a short / small seek distance. The abstract concept of an adjacent block could also be called a temporally close block; could be short seek distance or based on rotation. Note that the platter is rotating constantly, and by the time the head has finished moving, 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 stores the data in and builds 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.

2.1 - Solid State Storage

Flash disks can be used as secondary storage or as a caching layer, with the main advantage over traditional disks being that random reads are as fast as sequential reads. Disadvantages are slower random writes and a limited number of write cycles. Data is organised in pages, which is organised in **flash blocks**. Similar to RAM, the time to retrieve a page isn't related to the location. The internals of a flash package are as follows;



There are many of these within an SSD, which in turn is structured as an interface (either SATA or PCI) connected to the internal CPU which is connected to internal memory and the flash controller, which is connected to many flash packages. There are no moving parts, hence no mechanical limitations. The flash controller helps to ensure the entire drive degrades at roughly the same rate (wear levelling), preventing the capacity from shrinking. The access time depends on the bandwidth of the flash packages, device organisation as well as software efficiency. The flash translation layer (FTL) is the firmware that manages the hardware.

A write involves copying the page we are writing to a new page and then deleting the old page, leading to slower random writes. On the other hand, if we do a serial write, we can keep it in memory until we have a full flash page. Deletion takes time (and is typically on the block level, rather than page) as the pages are densely packed and a high voltage is required to erase a page, seeping into other pages. Blocks are isolated from each other, but pages are not (would waste space).

As the latency gap between CPU and RAM grew, caches were introduced to bridge the gap. Similarly, the gap between RAM and HDD can be bridged with SSMs (solid state memory) as a cache or buffer.

Today, only flash and PCM (phase change memory) are pursued commercially. PCM could be placed below flash in the memory hierarchy as the density is still too low.

DBMS is traditionally designed around a HDD model (buffer management, etc). Transactions are only really there (ACID properties) as we need to run things in parallel (since we're waiting on disk and don't want to block). Similarly, query plans are also HDD optimisations in the sense they prefer sequential over random access.

Flash can either be used to replace HDDs, act as an intermediate layer between HDD and RAM, or be used with HDDs on the same level. The correct use depends on the workload, such as the amount of data or how the data is accessed, as well as future trends in terms of density.

Some uses of SSDs in databases;

1. **flash-only OLTP**

   OLTP is dominated by random reads and writes, which is very bad for disks, but much faster on flash. With random writes, when used naively in place of a HDD, the throughput drops substantially over time, with significant variance.

   The append / pack algorithm; for an append-only operation, we write sequentially as much as we can. No updates are done in place, but rather the old page is invalidated (after reading the content)

and a new page is written. Once we run out of space, we reclaim the space and reorganise the storage and continue to write the pages in append-only. Essentially, random writes are transformed into sequential writes and a garbage collection step takes place (with some overhead). This adapts the properties of the algorithm to the underlying hardware.

2. **flash-aided business intelligence (OLAP)**

   These are typically read-only queries, with very few scattered updates and complex queries. There are typically two choices - for freshness, updates are performed in-place (with the queries), whereas for performance, updates are done in batches (separate from queries). However, there may be stale data in the latter case.

   Flash can be used as a write cache for analytics. Incoming data is written to the SSD, which is then merged with data from the disks to answer a query. We are only using SSDs where it's beneficial; for random accesses. This essentially buffers updates on flash rather than memory, which is larger, cheaper, and also persistent. Of course, this comes with limitations including avoiding random writes (which we saw how to overcome earlier), as well as a limited number of writes due to endurance issues.

   The concept behind **Materialised Sort-Merge (MaSM)** is that we have a system for large scans of tables (like OLAP) which keeps the data on disk when it does it well, but puts updates (where the performance is poor) on SSDs. The data is then merged when queried.

3. **logging on flash + HDD**

   Transaction logging is a major bottleneck. Logs are typically what transactions are to be executed (so it can be redone on a failure), which are small sequential writes. The access pattern for this involves small sequential writes which causes a HDD to incur rotational delays.

In general, they can be used as a helper on the memory level (not changing the DBMS), adapt the I/O pattern with small DBMS changes, or to fully change storage management.

2.2 - In-Memory Databases

OLAP deals with large amounts of data (data warehousing / data mining) where we look at the output of operational databases, turn them into historical databases and run large, complex, aggregation queries. The queries are long running and deal with a large number of tables, and are mostly read-only (only analysing the business). On the other hand, OLTP are primarily transactions, hence many small updates. There are very few tables touched, with generated queries (not typically human-written).

OLAP databases are in the order of petabytes, which need to be stored on disks or similar. On the other hand, transactional databases are quite small and not growing much; it's possible to buy a TB of main memory.

On the *Shore DBMS* prototype, only 4% of the CPU cycles are spent on actual useful work, whereas the remaining 96% are spent (equally) on latching, recovery (dealing with the log file), locking, and the buffer pool (reading and writing between main memory and disk). To improve the performance, improving overhead in processing (better data structures etc., only improves 4% of the actual performance). However, deploying in main memory gets rid of the buffer pool entirely, leading to a solid improvement in performance.

We have three choices of solutions;

- **OldSQL** legacy RDBMS

  Traditional SQL doesn't scale well in the distributed sense.

- **NoSQL** give up ACID properties to accelerate performance

  ACID properties and transactions are difficult to scale out. Also gives up SQL; the SQL is translated at compile time to a sequence of low level operations (difficult to do). High level queries are also good for abstraction. Giving up data consistency can also be a big issue with NoSQL.

  This is appropriate for non-transactional systems, without shared state, and single record transactions that are commutative.

- **NewSQL** preserves SQL and ACID

  This has both SQL and ACID, and performance and scalability is provided through modern software architecture. It requires solutions for the following, all of which are large sources of overhead;

  - **traditional record level locking** timestamp order
  - **buffer pool overhead** removed when moved to main memory
  - **latching for shared data structures** single-threading for small operations
  - **write-ahead logging** how to address failure

The example we're going to use for NewSQL is *VoltDB*. It's all in main-memory (no buffer pool), small single threaded transaction (no concurrency, hence no locks or latches required), and has durability and availability through copies (redundancy) - therefore no traditional log is required. Now the 95% of the cycles are for useful work (with the remainder still being used for some locking). *VoltDB* currently runs a subset of SQL, on clusters with LAN and WAN replication. It can scale to 384 cores. The only locking involved is for multi-partition operations.

For each of the solutions, only NewSQL is suitable for new OLTP;

- OldSQL is too slow and doesn't scale
- NoSQL lacks consistency guarantees and has no language to express queries on a high level (difficult to use)
- NewSQL is fast, scalable, consistent, and supports a high level language

There is a partition per physical CPU core, hence each physical server typically has multiple *VoltDB* partitions. Tables can either be **partitioned** or **replicated**. The former has a single column acting as the partitioning key, allowing rows to be spread across all partitions by the partition column. This is better for data with a high modification frequency (transactional data) as it reduces the amount of data being touched. On the other hand, the latter has all rows existing in **all** partitions and is better for mostly static data.

Similarly, there are also two types of work, which are both ACID. Single-partition work happens within a single partition (and requires no locking as it's in a single thread) and is more suitable for the majority of transactional work. On the other hand, multi-partition work where we have queries that touch multiple partitions.

Consider the following "schema", where we have three partitions (customer IDs 1 and 4 are in partition 1, 2 and 5 in partition 2, and 3 and 6 in partition 3);

```
table orders (partitioned):
    customer_id (partition key)
    order_id
    product_id

table products (replicated):
    product_id
    product_name
```

Examples of queries are;

- `select count(*) from orders where customer_id = 5` single (p2)
- `select count(*) from orders where product_id = 3` multi

  multi-partition as we don't partition on `product_id`, however it's a read operation and no locking is required
- `insert into orders (customer_id, order_id, product_id) values (3,302,2)` single (p3)

  single-partition if all the values fit in the same partition
- `update products set product_name = 'spork' where product_id = 3` multi

  updating replicated data requires some locking

In each partition there is a work queue where the queries are coming in. There is data for the table and the indices, and a single execution engine which executes requests sequentially - once again, no locking is required as there is no parallelism.

The database is constructed from the schema (DDL), the work (Java stored procedures), and the project containing users, groups, and partitioning. The *VoltCompiler* creates an application catalogue distributed to all machines. All access is via Java stored procedures, a single invocation is a transaction that is committed on success, limiting round trips between the DB and the application. Communication with the client is asynchronous.

This scales by adding more machines (or more RAM for more main memory). There is high availability from K-safety for redundancy, and snapshots are available, either scheduled, continuous, or on demand (which can be spooled to a data warehouse).

Due to the asynchronous calls, invocations are sent and responses are pulled from the server, allowing a single client to generate more than 100K transactions per second (TPS) - the client will behave synchronously if required.

```
traditional:
  salary = get_salary(employee_id);
VoltDB:
  callProcedure(asyncCallback, "get_salary", employee_id);
```

However, it doesn't support client-side transaction control (the client cannot perform a rollback) - a stored procedure will commit if it's successful and rollback otherwise. The procedure can can call for rollback.

The lack of concurrency can be beneficial, but it's also important to ensure that any single query is short (transactions should execute in microseconds). 'Inventory' type applications benefit from this. Since locking doesn't have to be considered, dead-locks aren't a concern either. As other transactions wait for the running transaction to complete, nothing intensive should be done in a stored procedure (requesting web pages, etc.) - this is useful for OLTP but not OLAP. This is optimised for throughput, not latency.

A subset of SQL is supported; `SELECT`, `INSERT` (with values), `UPDATE`, and `DELETE`. Aggregations such as `AVG`, `COUNT`, `MAX`, `MIN`, and `SUM` are also supported. Functionality will be added over time, but can be done in Java already. The execution plan is created at compile time (can be analysed), but may lead to performance issues if there's a drastic change to the cardinality of data (recompile and redeploy in this case)

October 18 - Live Lecture

With SSDs, each page has a limited number of write cycles (around 3,000 per page). However, we cannot write to a specific page directly anyways, as this abstracted away by the firmware. One important thing to keep in mind is that due to this limitation on SSDs, there may be issues when replacing a HDD for

a write-heavy workload. Wear-levelling attempts to spread the writes evenly (as much as possible). To minimise writes, we should pack writes to be in the sizes of pages. When we scan (read sequentially), we are typically limited by the bandwidth of the connection rather than the devices.

The need for transactions comes from the use of concurrency, which is in place to keep the CPU busy (in traditional databases). However, this is not a concern in *VoltDB* as operations are done in serial. Locks are to separate user transactions (and protects database contents during entire transactions) and latches are used to separate threads (and protects in-memory data structures during critical sections).

A concrete example for a fine-grained index versus a coarse-grained one is as follows. The former would index every element of an array whereas the latter could take a bunch of entries and index on that; once you're in the 'bunch', you still have to perform a scan within it. This reduces the complexity by reducing the number of items indexed.

While *VoltDB* could be constrained by the network bandwidth, however it would likely not reach a bottleneck as the operations are very small (OLTP). Additionally, we can't increase transactions indefinitely as it still takes time to process (CPU would be overwhelmed first).

Append-pack requires more space on the SSD as there is redundant data. There is garbage collection to run over the data and pack it together, similar to how defragmentation works. There will be some extra work to do (to check whether the page is valid), however overall it's fast enough.

3 - Graph Databases

The example in this lecture will use *Neo4j*, other examples include *OrientDB*, *InfiniteGraph*, and *AllegroGraph*. A graph database is a fundamentally different model from traditional relational databases, where we have nodes and relationships between the nodes.

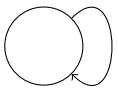
Graph databases are fairly powerful (and as general as RDBMS, with a fairly direct correspondence between them for translation). Works very well if they fit in the main memory of the machine. Graph queries check which nodes are connected together. Additionally, it's easy to query and scales well on one machine. However, scaling out (to multiple machines) is quite difficult - we have graph queries that query relationships between nodes, however this may lead to querying different machines.

A graph is an abstract representation of a set of objects, with pairs of these objects (also known as vertices or nodes) connected by links (also referred to as edges, arcs, or relationships). The different types of graphs are as follows;

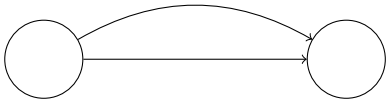
- **undirected** — no indication of direction

- **directed** — indication of direction, obviously

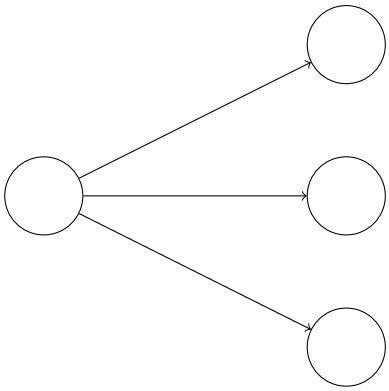
- **pseudo** — only one relationship, pointing to itself



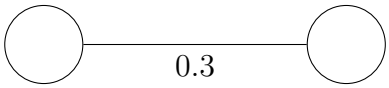
- **multi** — multiple relationships between nodes



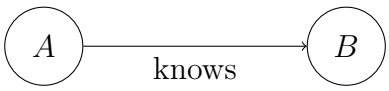
- **hyper** multiple outgoing relationships



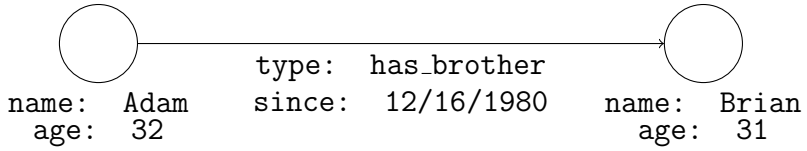
- **weighted** associated weight with the relationship



- **labelled** labels for nodes and relationships



- **property**



  This is the closest to what *Neo4j* uses. In this property graphs, nodes and relationships can have arbitrary properties, typically key-value pairs.

Consider a many-to-many relationship in a typical RDBMS, in this example we have `people`, `departments`, and `department_member` tables (where the latter joins the former two). In a graph model, there could simply be relationships from each entry in `people` to each of the entries in `department`, with a `belongs_to` (or similar) relationship. This eliminates the requirement for joins, which can be complicated if many joins are involved. A graph data model can be easily constructed by writing down the entities involved as nodes and then creating relationships between them. From here, properties (key-value pairs) can be added in to the nodes and relationships. It's generally quite similar to ER mapping, but we don't need to consider cardinality.

A graph databases has an explicit graph structure, with each node knowing its adjacent nodes (has links to) - which ideally lives in the same machine. Each node has indices for its neighbouring nodes to allow for fast access; the cost for a local step remains the same even if the number of nodes increase, however it can become slower if the hops are to different machines (network latency).

To translate to *Neo4j*;

- each entity table is a label on nodes

- each row in an entity table is a node, with the columns being node properties
- add unique constraints for business primary keys and indices for attributes which are frequently looked up

A node can have relationships and properties. A relationship has a start and end node, has a relationship type which is uniquely identified by a name, and can have properties. Property keys are strings, whereas property values can either be a primitive type or an array of a primitive type. A path in *Neo4j* is one or more nodes with connection relationships that can be retrieved as a query or the result of a traversal. Querying this in a traditional relational database could be very inefficient as you may end up querying tables multiple times.

Cypher is very similar to SQL, but for graphs. It is a declarative (no need to tell it how to retrieve things), pattern-matching language. Some examples are as follows;

- retrieves all pairs of nodes that are connected with a relationship; doesn't specify **what** relationship

```
START a=node(*) // now optional
MATCH (a)-->(b) // a and b are just placeholders (connection is important)
RETURN a, b;
```

- find all nodes with an outgoing relationship and get a specific property (`name` in this case) - no set property schema (similar to document databases), just ensure no name collisions

```
START a=node(*)
MATCH (a)-->()
RETURN a.name;
```

- find all actors which have acted in a movie

```
START a=node(*)
MATCH (a)-[:ACTED_IN]->(m)
RETURN a, m;
```

- with paths, note that `(a)-->(b)-->(c)` is different from `(a)-->(b)<--(c)`
- get actor and director names for movie titles;

```
START a=node(*)
MATCH (a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d)
// equivalent to the following (since m is named);
// MATCH (a)-[:ACTED_IN]->(m), (m)<-[:DIRECTED]-(d)
// MATCH (a)-[:ACTED_IN]->(m), (d)-[:DIRECTED]->(m)
RETURN a.name, m.title, d.name;
```

- sorting and limit can also be done (same as SQL) (count movies actors and directors have worked together in);

```
START a=node(*)
MATCH (a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d)
RETURN a.name, d.name, count(*) AS count
ORDER BY(count) DESC
LIMIT 5;
```

- aggregations Aggregations can also be done in a similar way;
  - `count(x)` add number of occurrences
  - `min(x)` get lowest value
  - `max(x)` get highest value

  - `avg(x)` — average of numeric values
  - `collect(x)` — collects into an array ('transposes' a column)

  An example of this is as follows, where we get the actors and directors who have worked together (and an array of movie titles);

```
START a=node(*)
MATCH (a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d)
RETURN a.name, d.name, collect(m.title);
```

- find a specific node (query all nodes); in this example we search for the name `John Smith`

```
START n=node(*)
WHERE has(n.name) AND n.name = "John Smith"
RETURN n;
```

  Note that the `has(n.name)` part ensures that the `name` property exists. This is required as we don't know that every node has this property.

- create a new `KNOWS` relationships between actors and directors who have worked together

```
START a=node(*)
MATCH (a)-[:ACTED_IN|DIRECTED]->()<-[:ACTED_IN|DIRECTED]-(b)
CREATE UNIQUE (a)-[:KNOWS]->(b);
```

- paths can have variable lengths - the following example matches the following paths; `(a)-->(b)`, `(a)-->()-->(b)`, or `(a)-->()-->()-->(b)`

```
START a=node(*)
MATCH (a)-[*1..3]->(b)
RETURN a, b;
```

  This is quite difficult to represent in a traditional relational database. An application of this is as follows, where we try to find the friends-of-friends of `John Smith`;

```
MATCH (john)-[:KNOWS*2]->(fof)
WHERE has(john.name) AND john.name = "John Smith"
RETURN DISTINCT fof.name;
```

- this searches for movies in which `Keanu Reeves` played `Neo` - note that it also shows that properties can be sets

```
MATCH (actor)-[r:ACTED_IN]->(movie)
WHERE "Neo" IN r.roles AND actor.name = "Keanu Reeves"
RETURN DISTINCT movie.title;
```

- actors who worked with `Gene Hackman` and are also directors (constraints can also be patterns)

```
MATCH (gene)-[:ACTED_IN]->(movie)<-[:ACTED_IN]-(n)
WHERE (n)-[:DIRECTED]->() AND gene.name = "Gene Hackman"
RETURN DISTINCT n.name;
```

- creating a node and adding properties

```
CREATE ({title:"Mystic River", released:1993}); // create new node

// similar to SQL, adding a tagline
START movie=node(*)
MATCH (movie)
WHERE movie.title = "Mystic River"
SET movie.tagline = "Lorem Ipsum"
```

- creating a relationship

```
CREATE UNIQUE (kevin)-[:ACTED_IN {roles:["Sean"]}]->(movie)
WHERE movie.title = "Mystic River" AND kevin.name = "Kevin Bacon";
```

- deleting relationships (remove all relationships for `John Smith`)

```
MATCH (a)-[r]->()
WHERE a.name = "John Smith"
DELETE r;
```

  Similarly, if we wanted to delete `John Smith` and relationships (regardless of whether any exist);

```
MATCH (a)-[r?]->()
WHERE a.name = "John Smith"
DELETE r, a;
```

October 21 - Live Lecture

If there is any meaningful amount of data that doesn't fit into main memory, graph databases can be less efficient than traditional SQL databases. The latter utilises the buffer pool to adapt to different types of workloads. Generally, there isn't much of a point in using graph databases unless the structure of the data can take advantage of it. Most of the cost is in memory accesses (rather than computation). Graph databases also build indices to optimise queries, however this is done in the background (rather than explicitly specified).

4.1 - XML Shredding

XML (extensible markup language) is designed to exchange data between systems with completely different data representations. It contains both a (self-sufficient) description of the data as well as what the data actually is. The rules for XML are as follows;

- first tag represents the root (single root for entire tree)
- other matching tag pairs are nodes; if there are a pair of tags inside another pair, the **contained pair** is the child of the **containing pair**,children have a defined order
- text is a child of the node corresponding to the tag enclosing the text - always a leaf node (text cannot have children)
- single tags are allowed and always become leaves with a box
- tags are case sensitive and must be properly nested

XML documents have a DTD (document type definition) which tells us whether an XML document is well formed or not. This is typically contained as a URL in the top of XML file. DTD is often used to generate a parser for the XML document. An example DTD is as follows (as well as an example XML file);

```
<?XML version="1.0"?>
<!DOCTYPE note[
<!Element note(to,from,heading,
    notebody)>
<!Element to(#PCDATA)>
<!Element from(#PCDATA)>
<!Element heading(#PCDATA)>
<!Element notebody(#PCDATA)>
]>
```

```
<note>
    <to>alice</to>
    <from>bob</from>
    <heading>message</heading>
    <notebody>lorem ipsum</notebody>
</note>
```

Note that line 3 defines the `note` element to have four elements within. Similarly, line 4 defines the `to` element to be of type `#PCDATA` (parsed character data). DTD can also declare the number of occurrences;

- `<!Element note(message+)>` one or more (at least 1) messages
- `<!Element note(message*)>` zero or more messages
- `<!Element note(message?)>` zero or one message

XML is an open standard, is human readable, and is easy to process. However, it tends to inflate the data (increase size) as it includes a lot of information that could be easily compressed away.

XML was primarily introduced as a means to exchange information, which would eventually be stored. This could be stored in a number of ways, including specialised systems, object-oriented databases, in a regular file system (which may be inefficient to search), or in a relational database. The primary ways to store XML in databases are as follows;

- **Structure-Mapping approach** - derives the database schema from the DTD (which describes the structure of the document)
- **Model-Mapping approach** - a fixed schema is used to store XML documents without any DTD information (some documents may not follow the DTD precisely), this looks at what the structure of the actual XML document is like
  - capable of storing any XML application (can be either static or dynamic, the latter being when the DTD varies over time)
  - can support XML that doesn't follow DTD as long as it's well-formed
  - doesn't require extending database models to support XML

  Some of the key terms that will be used throughout;

  - **Ordinal** the order of the element among its siblings (share same parent)
  - **LabelPath** dot separated sequence of edge labels (e.g. `DBGroup.Member.Name`)
  - **DataPath** dot separated sequence of element nodes (e.g. `&1.&2.&7`)

  There are the following approaches to perform model mapping (the last one is a node oriented approach, with the others being edge oriented). Each of these have advantages and disadvantages based on the queries we'd want to perform

  - **Edge** edges stored in a single table

    This can be represented as `Edge(Source, Ordinal, Target, Label, Flag, Value)`

    * `Source` represents source node in data graph
    * `Order` ordinal of the edge among siblings
    * `Target` target node which the source is pointing to
    * `Label` name in XML document
    * `Flag` data type being represented (reference or value)
    * `Value` represents data in the XML document
  - **Monet** partitions the edge table according to all possible label paths

    Stores XML in multiple tables, as queries are specific to label-paths. The number of tables is equal to the number of distinct label paths. Tables are classified as the follows (contrasted to the Edge approach, this gets rid of the flag attribute);

    * `ElementNode(Source, Target, Ordinal)` represents a unique edge
    * `TextNode(ID, Value)` value type is implicit in the table name
  - **XParent** structures on LabelPath, DataPath, Element, and Data

    Similar to XRel, this has four tables;

    * `LabelPath(ID, Len, Path)` also stores length of path
    * `DataPath(Pid, Cid)` stores pairs of parent ID and child ID
    * `Element(pathID, Ordinal, Did)` has path ID from `LabelPath` and data ID
    * `Data(PathID, Did, Ordinal, Value)`
  - **XRel** stores based on Path, Element, Text, and Attribute

    This stores data in four tables;

    * `Path(PathID, PathExp)` maintains a path expression identifier and path expression
    * `Element(PathID, Start, End, Ordinal)`
      contains the start and end positions of regions for a given `PathID`; region denotes the start and end positions of the node in the XML document
    * `Text(PathID, Start, End, Value)`
    * `Attribute(PathID, Start, End, Value)`

  All of these approaches allow us to reconstruct the document but looks at the data from different perspectives depending on the most frequently asked queries.

To select the names of all members who are older than 20;

- **Edge** 6 selections, 3 equi-joins

```
SELECT name.Value
FROM Edge dbgroup, Edge member, Edge age, Edge name
WHERE
  dbgroup.Label = 'DBGroup' AND member.Label = 'Member' AND
  age.Label = 'Age' AND name.Label = 'Name' AND
  dbgroup.Source = 0 AND dbgroup.Target = member.Source AND
  member.Target = age.Source AND member.Target = name.Source AND
  CAST(age.Value AS INT) > 20
```

- **Monet** 1 selection, 4 equi-joins

```
SELECT cn.Value
FROM
  DBGroup.Member.Name n,
  DBGroup.Member.Age a,
  DBGroup.Member m,
  DBGroup.Member.Name.CDATA cn,
  DBGroup.Member.Age.CDATA ca
WHERE
  m.Target = n.Source AND m.Target = a.Source AND
  a.Target = ca.Id AND n.Target = cn.Id AND
  CAST(age.Value AS INT) > 20
```

- **XParent** 3 selections and 5 equi-joins

```
SELECT d2.Value
FROM
  Data d1, Data d2, Element e1,
  LabelPath lp1, LabelPath lp2, DataPath p1, DataPath p2
WHERE
  lp1.Path = './DBGroup./Member./Age' AND
  lp2.Path = './DBGroup./Member./Name' AND
  CAST(d1.Value AS INT) > 20 AND
  d1.PathID = lp1.Id AND d2.PathID = lp2.Id AND
  d1.Did = p1.Cid AND d2.Did = p2.Cid AND p1.Pid = p2.Pid
```

- **XRel** 4 selections and 7 joins (not equi-joins)

```
SELECT v2.Value
FROM Element e1, Path p1, Path p2, Path p3, Text v1, Text v2
WHERE
  p1.PathExp = '#DBGroup#/Member#' AND
  p2.PathExp = '#DBGroup#/Member#/Age' AND
  p3.PathExp = '#DBGroup#/Member#/Name' AND
  e1.PathID = p1.PathID AND
  v1.PathID = p2.PathID AND
  v2.PathID = p3.PathID AND
  e1.Start < v1.Start AND e1.End > v1.End AND
  e1.Start < v2.Start AND e1.End > v2.End AND
  CAST(v1.Value AS INT) > 20
```

Typically XRel and XParent outperform Edge in complex queries, however Edge performs better with simple queries (fewer joins).

4.2 - Document DB

The examples used in this lecture will revolve around *MongoDB*. SQL has issues with a rigid schema, difficulty scaling (built around transactions), and also requires joins which aren't intuitive (complex and difficult to write). On the other hand, *MongoDB* can interface easily with many common languages and keeps essential features from RDBMS, but also takes features from NoSQL key-value stores. The data model is as follows;

- document based, maximum size of 16MB (can be seen as similar to a SQL tuple)
- BSON format documents (field-value pairs) - binary representation of JSON

  JSON is JavaScript Object Notation - easy for humans to read / write (and also easy for computers to parse / generate). Objects can be nested, similar to XML, and is built on name / value pairs and ordered list of values. Binary JSON, a binary-encoded serialisation of JSON-like documents, also allows for referencing (reduces need for joins, similar to embedded structure).

  Note that there is a `_id` field (inserted by *MongoDB*). This acts as a primary key for the collection - it's unique, immutable, and can be any type (other than arrays). By default, the type is `ObjectId` - sorting on this is roughly equivalent to sorting by creation time.
- document stored in collection
  - documents can have completely different structures in a collection
  - have a common index set
  - similar to tables in a relational database

Using the shell, we have the following commands;

- `db` check which database is in use
- `show dbs` show all databases
- `use <name>` change database or make a new one
- `show collections` display collections
- `db.<collection>.insert({<field>:<value>})` insert a document
- `db.<collection>.find()` returns a cursor to display first 20
  `.limit(<number>)` to limit results

- `db.<collection>.findOne()` get one document
- `db.<collection>.find({<field1>:<value1>,<field2>:<value2>})`

  equivalent to `WHERE <field1>=<value1> AND <field2>=<value2>`
- `db.<collection>.find({$or: [<field1>:<value1>,<field2>:<value2>]})`

  equivalent to `WHERE <field1>=<value1> OR <field2>=<value2>`
- `db.<collection>.find({<field1>: {$in [<value1>,<value2>]}})` match collection
- `db.<collection>.find({<field1>:<value1>}, {<field2>:0})`

  equivalent to `SELECT <field1> FROM <table> WHERE <field1>=<value1>`
- `db.<collection>.find({<field>: {$exists: true}})` find with or without field
- updating multiple documents (if `upsert` is true, it creates a new document when nothing matches the search criteria);

```
db.<collection>.update(
    {<field1>:<value1>},
    {$: {<field2>:<value2>}},
    {multi: true}
)
```

- `db.<collection>.update({<field>:<value>}, {$unset: {<field>:1}})` remove a field
- `db.<collection>.remove({<field>:<value>}, true)`

  remove first record (if `true` is omitted, remove all)
- `db.<collection>.ensureIndex({<field>: 1})` creation index

All writes are atomic on the level of a single document - all writes can be interleaved with other operations. The flexibility of the schema gives many advantages such as easy modifications, however it moves the complexity of checking which fields we have into the application code.

MongoDB has two patterns to avoid the use of joining;

- **embedding** (like pre-joining) - if we have a one-to-one relationship and the objects are likely to be queried together, it's easier to store it as one document rather than two
- **linking** - if we have a one-to-many relationship embedding could lead to significant replication, as such we have linking which pushes the join into the application code

A significant amount of modern data is already structured in a JSON-like schema, and as such there's no point in translating it into a relational schema and then translating it back. `find()` is also more semantically clear for programming. Data denormalisation provides data locality, which in turn provides speed (however it can lead to more data being retrieved).

If we wanted to find all users with a score below 30, for example, we would have to scan every document (given we have no index), which is inefficient as it processes a large volume of data. Indexes are special structures that help to find data quicker (small portions of a collection's data set in an easy to traverse form, that points to the document). The three types are single field (index points into collection for a specific field), compound field indexes (similar to a secondary index), and multi-key indexes (multiple keys for a specific field).

Aggregations are operations that process data and return computed results. By running aggregation on the `monogod` (primary daemon) instance simplifies application code. This is implemented with pipelines with filters that operate like queries and document transformations which modify the output form.

October 25 - Live Lecture

Ordinals are required as XML documents have an order of elements. When we query, we want to be able to raise queries like the first node of a specific kind, etc.

There is no concept of updating XML in graph databases. XML is human-readable, but is primarily used as an exchange medium between machines (commonly in service-oriented architectures).

The purpose of a region in XRel can help find the entire subtree of a node. The start and end values are the first and last characters.

Denormalisation (embedding) removes the need for joining, but may make it less efficient. Additionally, updating an embedded field in a one-to-many relationship would require touching every document that contains the field.

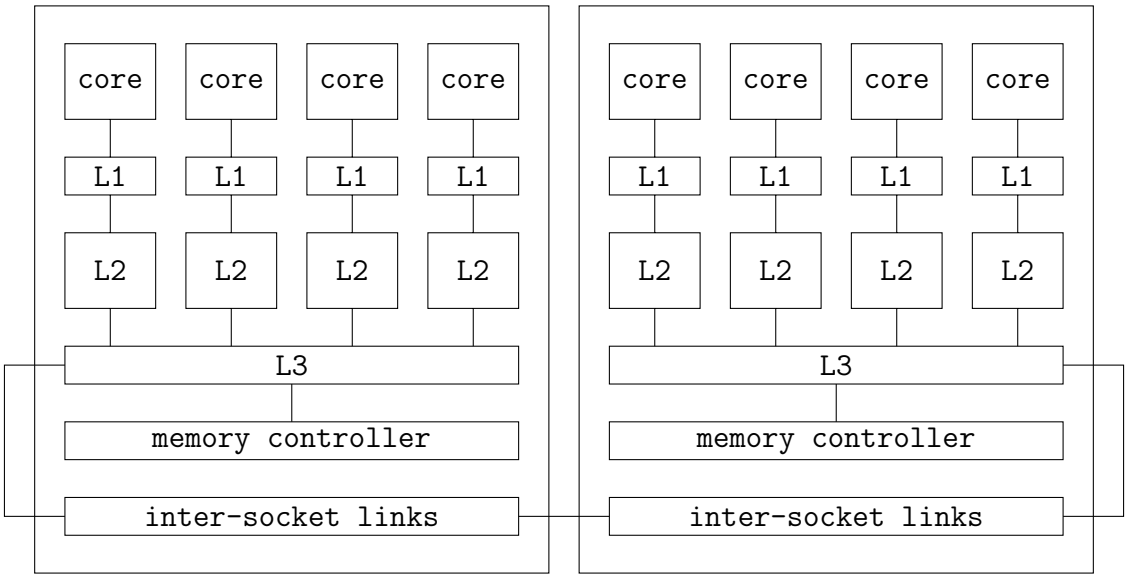
Under the hood, it's the same as a relational database, just a different data representation. However, queries are slightly different; with a relational database we ideally have fixed length fields, whereas with a document database we can make fewer assumptions about the data.

The structure mapping approach is used when you have the DTD, and you know it's static. The model mapping approaches captures everything, including changes to the format and structure of the XML (dynamic).

5 - Multicores

Power, clock speed, and performance per clock starts to stall despite the number of transistors still following Moore's Law (note that this graph was from 2010). The trends of processors started from pipelining and multithreading. Once that hit a limit, multicores (CMP) starting being introduced as there was a limit on the amount of power in a single core. Recently, multiple sockets each with multiple cores have been used. The overall goal is scalability. Each core has its own L1 data and instruction caches and L2 caches, but share an L3 cache and also access main memory together.

In the horizontal dimension, the goal is to exploit abundant parallelism. Note that accessing data from different socket will be slower;



With OLAP, as the number of threads increases, the throughput increases until a certain point where it begins to plateau. However, with OLTP, it initially goes up but then goes back down and plateaus at a lower level. This is due to the access latency and locking of shared data structures. On the other hand, the bottleneck with OLAP is due to the memory bandwidth.

In practice, accessing the core to L1 only costs around 4 cycles and can be considered to be local. However, any of the lower layers can lead to stalls, which costs power and resources. Even in cloud workloads, over 50% of the time (in most cases) goes to stalls, leading to approximately one instruction

per cycle. In database workloads, for the *Shore-MT* example, the stall cycles are dominated by L1I and L3D. For data intensive applications, between 50% to 80% of the cycles are stalls; due to instruction fetch and long-latency data misses (unavoidable; if we load unseen data, we are going to have a cache miss). To optimise this, we need to look at instruction cache locality as well as cache line utilisation for data.

Some approaches for minimising memory stalls are as follows;

- **prefetching**

  Simple prefetching approaches include next line, where if we miss A, we fetch $A+1$, or stream, where if we miss $A, A+1$, then we fetch $A+2, A+3$. This favours sequential access as well as spatial locality, however doesn't help with branches / function calls (where we need branch prediction), or pointer chasing (which can be done with the stride approach, where if we miss $A, A+20$, we fetch $A+40, A+60$, in the case where we have a predictable structure). The former is a case for instructions whereas the latter is a case for data. These strategies are preferred on hardware as they are simple. The penalty for a miss is a few cycles; if we exceed this to prefetch, we lose the benefit.

  Temporal streaming is based on the fact that programs tend to follow the same overall flow (in the way data and instructions are accessed) repeatedly. The idea is to exploit recurring control flow; this is more accurate, however requires a higher space cost.

  An example of software-guided prefetching is when we visit a node, we may want to prefetch the children nodes.

- **being cache conscious**

  An example of code optimisation is to simplify the code; in-memory databases have a smaller instruction footprint. If the code is written to minimise jumps, it can better exploit the next line prefetcher. The code can also be profiled for static optimisations, or just-in-time for dynamic. Queries can also be compiled directly into machine code (rather than being interpreted on-the-fly).

  Data layouts can also be cache conscious. Assume we have two 16-byte columns and a cache line of 64 bytes. If our queries are `SELECT *`, then a row store (good for OLTP) is beneficial, otherwise if our queries are in the form of `SELECT <col>`, then a column store (good for OLAP) is beneficial. The goal is to maximise the utilisation of the cache line as well as exploit the next line prefetcher.

  In the example of an index tree, if we are likely to perform a lookup-heavy workload, it should be stored in the visitation order of DFS, whereas if we have a scan-heavy workload (where we look at each level in order), it should be stored in memory in the BFS order. The goal remains the same.

  The volcano iterator model passes a single tuple between operators. A trivial improvement is to perform vectorised instructions which passes multiple tuples at a time between operators. This provides good data and instruction cache locality, as well as allows for exploiting SIMD.

- **exploiting common instructions**

  On executing the *TPC-C* benchmark on *Shore-MT*, the amount of data that is hot (reused often) is quite low. However, the instruction reuse for certain workloads is quite high. Consider a thread `T1` that has the instructions `A,B,C,D` and another thread `T2` that has the instructions `A,B,C`, and 4 total cores. In the first cycle, `T1` executes on core 1, leaving `A` in the cache. In the next cycle `T2` executes on core 1, which already has `A` cached, and `T1` executes on core 2, leaving `B` in the cache. Then `T2` executes on core 2, and `T1` executes on core 3, and so on. The thread is executed on the core where the instruction is cached (if possible). This approach is called SLICC.

In general, the problem with the L1I misses is due to capacity and can be solved by minimising the footprint as well as maximising reuse to create the illusion of a larger cache. LLC (L3) data misses can't be avoided, but we can maximise cache line utilisation with better algorithms and data layout.

If multiple cores are accessing (modifying) the same data, locks need to be used to ensure safety. The data access pattern across threads is highly unpredictable. For each transaction, latching and locking accounts for a high number of critical sections. There are different types of critical sections (goal is to go from unbounded to fixed or cooperative);

- **unbounded** all threads have access to centralised synchronisation point
  many threads could want access at the same time; can be a major scalability bottleneck
- **fixed** fixed number of threads entering no matter how many threads in the system
  not a bottleneck
- **cooperative** threads can combine requests as they're entering

To scale up OLTP, we need to consider the following;

- **unscalable components**

  Hot shared locks cause contention. Consider multiple threads which all acquire different cold locks, but the same hot locks. This leads to the same locks being released and requested repeatedly. A solution is to perform speculative lock inheritance. In this process, the first transaction commits **without releasing** hot locks and instead passes them to the next transaction which can either release them or use them, and so on. This reduces lock contention.

  We want to move from centralised locking to thread-local locking, allowing each thread to use the locks as it pleases. Predictable data access can be achieved by having a thread-to-data pattern; where each thread accesses the same subset of data. This is beneficial as locks don't need to be passed between threads, since the data is on the same thread. The same concept is used in in-memory databases.

  To reduce latching, we can use physiological partitioning (PLP). An example of this is a multi-rooted B-tree. Consider two workers, each working on half of the tree. If we have something that access the first partition, the work is passed to the first worker, and similarly for the second partition.

  In a serial log performing WAL (write-ahead logging), locks are used to ensure that the threads write serially. A number of methods can be used;

  - **early lock release**
  - **flush pipelining** log writer on each thread and flush to the writer
  - **consolidation array** minimises log contention

- **synchronisation**
- **non-uniform communication**

  Recall the diagram for the multiple sockets. Accessing the L1 cache costs less than 10 cycles, accessing the L3 cache takes around 50 cycles, and accessing between sockets costs around 500 cycles. As this is expensive, each socket should be considered an 'island'. OLTP on islands are as follows (similar ideas are in cluster computing);

  - **shared-everything** everything is shared between sockets and cores
    sub-optimal due to the cost of accessing between sockets.
  - **island shared-nothing** cores on the same socket share
    good middle ground, avoids expensive inter-socket communication

– **shared-nothing** nothing is shared, even between cores
fast, however it's sensitive to the workload

The challenge lies in configuring the workload for the hardware, as repartitioning is expensive. Adaptive Transaction Processing use the system state to determine how we want to run tasks. It makes shared-everything system scalable on islands and adaptive to any changes to the hardware / workload. PLP is used between islands, and uses different threads to access data on different subsets of the data, located on different islands.

Scaling up OLTP involves identifying bottlenecks in existing systems (and eliminating them). We also cannot assume uniformity in communication (different orders of magnitude of time to access different data). The correct synchronisation mechanisms should also be chosen.

October 28 - Live Lecture

Predictions cannot take longer than a stall, so very simplistic models need to be used.

We need to ensure we isolated updates (either on the level of a table or row). Some care might need to be taken for reads as well (to ensure data isn't being read during an update).

In speculative lock inheritance, we pass hot locks between transactions, after committing.

Shared-everything, in distributed computing, works well when there is low latency between islands. However in the case of sockets, there is (relatively) high overhead. Shared-nothing requires the work to be partitioned where everything is local.

In serial logging, the transaction that will be executed is written into the serial log (persistent). However, not every transaction can write at the same time, hence access to it needs to be serialised. The idea for flush pipelining is to start writing, but start working on other tasks when waiting; not improving latency but improving throughput.

6.1 - Cold Storage

Data can be classified by how frequently it's accessed (as well as size). The amount of data is growing rapidly, however the majority of it is infrequently accessed (cold) - compared to frequently accessed data (hot). The goal is to find a good tradeoff between price and performance; it doesn't make sense to store all cold data in SSD. Archival data is typically written once and quite likely to never be read, and is stored on tape (we don't care about latency). Cold storage devices (CSD) fit between the hot / warm tiers and the archive tier. The cold tier has low hardware costs and a lower latency than tape.

A key idea is adequate provisioning. We want to provide just enough cold storage resources just for the cold data workload. Disks could be archival and SMR, rather than standard commodity drives. We also want to limit power, cooling, and bandwidth - there's no need to provision for peak power and have all the disks spinning all the time; just enough disks spun up to saturate bandwidth. The advantage of removing unnecessary resources include low hardware costs and operating costs, as well as a high density of storage.

An example for this is *Microsoft Pelican*, where only $\frac{1}{K}$ of the disks are active ($\approx 8\%$) - one disk group spun up. Spinning up a new group of disks (disk switch latency) is between 10 and 30 seconds. The workload that CSDs fall under is known as WORO; write once read occasionally.

The *Pelican* rack has a very high storage density (52U with 1152 archival 3.5" SATA disks, approximately 5 PB of storage); achieved by removing unnecessary infrastructure / hardware - for example fixed cooling channels. It's designed to store blobs which are partitioned into multiple chunks. Management is made simpler by grouping the disks into domains (limits on the following);

- cooling only 2 disks can be active at a time in a group
- power only 1 disk active at a time in a group

- bandwidth

Disks use resources from a set of resource domains (which are provisioned to supply resources to a subset of disks). Disks are either domain-conflicting (same resource domain) or domain-disjoint (share no common resource domains). The slides contain a representation of this.

This is generally more of an object store than a file system. Each blob is stored over a set of disks (split into a sequence of 128kB fragments - for each k fragment, additional r fragments are generated for error correction, the $k + r$ fragments form a stripe; if $k = 15$ then $r = 3$). *Pelican* statically partitions disks into groups (disks can be concurrently active); conflicts are concentrated over a few sets of disks. The lecture then goes over the placement of data. Two groups can be fully colliding / conflicting, but if none of the disks in the same group is colliding then it can all be accessed at the same time.

Traditional schedulers reorder IOs to minimise seek latency, whereas *Pelican* reorders the requests in the queue (read / write) to minimise the impact of spin up latency. *Pelican* also has a reordering operation (similar to disk defragmentation); the scheduler is more likely to write to an active (spinning) disk than to spin up a different disk.

The implementation needs to ensure that disks are powered up in standby, to ensure it doesn't spin up without *Pelican* managing the constraints. Group abstraction is exploited to parallelise the boot process - the most important aspect of the groups is to simplify scheduling (if we don't have groups, we need to consider which other disks it conflicts with; there are 3 domains and 1152 disks to consider).

The next section discusses *Pelican* compared against a system that has full provisioning for power and cooling; same organisation (physical internal disk topology), but the disks are never spun down (obviously better for latency). As expected, the throughput for FP goes up as the number of requests go up (until it hits the bandwidth limit), whereas *Pelican* does something similar however hits a limit earlier due to the group switching (and one group not entirely saturating the bandwidth). The time it takes to read the first byte for *Pelican* starts around the 14.2 second point (the average time it takes to activate a group), but as requests goes up it becomes the same value as the FP. However, for power consumption with all disks spun down (just powering system), the draw is at 1.8kW, whereas for the *Pelican* peak, it's at 3.7kW, and for all disks active, it's at 10.8kW.

The benefits of this are reduced cost and power consumption (right-provisioned), has accuracy from error codes, and abstraction simplifies IO scheduling. On the other hand, it's not flexible to changes and is sensitive to hardware changes.

Data Processing

In the traditional setting, we typically had a database accessing a disk, with uniform access, a controlled layout, and static (pull-based) execution, all done via blocks. On the other hand, we're likely to have multiple databases reading objects over the network for the cold storage tier. There is no longer uniform access or controlled layout, and pull-based execution will lead to group switches.

We cannot just simply replace HDD with CSD in the background, as it will have an impact on the software. Therefore we require hardware-software co-design; access needs to be hardware-driven to minimise group switches. Execution engine has to process the data from storage in an out-of-order / unpredictable manner. Data round-trips should also be reduced with smart caching.

Common batch processes on CSD include;

- massive-scale group-by / join
- duplicate detection
- data localisation
- in-place map-reduce

We typically don't do operations on cold data, but when we do it should still be done efficiently. Data needs to be partitioned into K groups to be distributed between K disk groups. Data from each disk group is read into memory (buffer), partitioned, and then flushed to the different disk groups - allowing the a CSD group to have partitions 1 to m and then the next group having partitions $m+1$ to n, and so on.

However, when the buffer is full, we need to flush it. The naive approach is to flush a partition once it's full, and then fill up just that partition. However, this leads to many switches. The Buff-Pack approach tries to avoid switching by flushing to the current disk group if possible, otherwise it switches to the disk group with the largest buffer. The intuition for the Off-Pack approach is that we flush the entire buffer (into the current disk group) even if it's in the wrong group. After the flush, all of the offload buffer is moved.

The estimated total time is as follows, note that the switching time dominates this;

$$T_{\text{total}} = T_{\text{switch}} + T_{\text{seek}} + T_{\text{read}} + T_{\text{write}}$$

In general, Off-Pack performs better as it has fewer switches and more read / writes. It's better for a smaller buffer, high number of disk groups, and high throughput. For a low number of disk groups and smaller switch latency, Buff-Pack can outperform Off-Pack. Off-Pack works well with smaller buffer sizes, however as the size gets larger, Buff-Pack performs better.

6.2 - DNA Storage

An issue with tape storage is that 60% of archival data is stored for longer than 20 years, however tape has a lifetime of 10-20 years. This means that the data needs to be migrated every couple of years. A similar issue is present with microfilm, called vinegar syndrome, causing the film to disintegrate.

DNA has been synthesised for medical purposes. It offers a higher volumetric information density (gigabits per cubic millimetre) than other mediums, with a potential 10^7 improvement over existing storage media. Additionally, it's also very durable. The demand for silicon is also starting to exceed the global supply. Finally, biological operations on DNA are highly parallel (millions, if not billions of operations in parallel).

DNA is a double, long chain of molecules called nucleotides (A, T, C, and G). The complementary nucleotides (A / T, and C / G) provide stability.

To store information, the binary information is encoded into nucleotides, synthesised into DNA, sequenced back into the nucleotides, and then finally decoded back into binary. The goal is to minimise the number of nucleotides per bit, as it's still quite expensive. A simple encoding would be $\texttt{00} \to \texttt{A}, \texttt{01} \to \texttt{T}, \texttt{10} \to \texttt{C}, \texttt{11} \to \texttt{G}$, which would theoretically have have half a nucleotide per bit. However, there are biological constraints; for example we can't have too many of the same nucleotide in a row as it leads to breakages. Additionally, synthesis and sequencing are error-prone. In reality, we have the following, with an identifier / offset, error correction codes to account for errors in the aforementioned steps, and the actual value;

AGGCTCAGATAGATCTAATT

This leads to around 1 nucleotide per bit. There are also biological constraints on the length of the sequence, due to the technology to synthesise but also the likeliness of breakage in longer sequences. As such, data may need to be broken down into an array of sequences, and can be addressed by the identifier.

If we have two structures which are perfectly complementary, they are highly likely to bind together. This has been historically used to solve combinatorial problems, and can also be used for database joins. The idea is to exploit chemical processes;

- annealing of complementary nucleotides

- polymerase chain reaction (PCR) to replicate / amplify DNA sequences

  Content retrieval through amplification. We need to know the beginning of the sequence and the end of the sequence. If we have a mix of sequences, we can apply PCR with the knowledge to create millions of copies, leading to it dominating the mix.

- loop-mediated isothermal amplification

  content detection

A simple approach is to encode a binary dump of the database. This wastes space on the identifier ($\log_4(|\text{segments}|)$), doesn't support point queries, and near-molecule data processing can't be performed. Reassembling this would require clustering and reassembly, which is a necessary but time-consuming step before decoding.

A structured data layout approach is to use NSM on DNA, with one row per oligo (short single strands of synthetic DNA). The unique primary key is used to avoid extra indexing. DSM on DNA is a columnset partitioning for large rows, which keeps the primary key and a **single** field. With structure-preserving encoding, the DNA read restoration can be mapped to a data cleaning operation.

To perform point queries, we can use PCR with the table and column ID as the start sequence, and the value as the end sequence.

Join

The goal is to use the complementarity (matching base pairs) to anneal complementary single stranded oligos. Matching records / attributes have complementary encodings of the values.

Each attribute is encoded as before, but with an additional reversed oligo with the value complemented (`TAAAGATCTACTCAGATAGG`). If this reversed oligo matches a non-reversed oligo, the values will be perfectly complementary (exactly anneal). See the example below - note the three nucleotides in parentheses on either line are the table and column IDs, and the square brackets are the complementary values.

```
(AGG)CTCAGATAGATCTA[ATT]
                   [TAA]ATCGAGGGATTACA(TTT)
```

However, this could lead to wrongly annealed pairs. PCR is then used to obtain only the chains which have annealed and are the IDs we are looking for. If they anneal, they are twice the weight of a single sequence.

OligoArchive

The idea is to implement all the layers of the storage stack alongside automation on all layers;

- **application layer** encoding structured (database) and unstructured (imaging) data
- **OS layer** file system abstraction
- **controller layer** data processing capabilities
- **media layer** synthesis and sequencing

Nanopore sequencing pulls DNA through a membrane which unzips the helix into two strands. Each base blocks the flow to a different degree, altering the current which can then be used to identify them electronically.

The current roadblock is the synthesis cost. For medical purposes, the synthesis has to be precise, but for storage purposes there is some tolerance for errors. Other approaches include longer sequences, longer alphabets (more nucleotides), and synthetic molecules.

Current research is looking at encoding, synthesis costs, synthetic molecules, automation, and packaging / storage (wrapping long sequences around synthetic histones).