Database Systems on Modern CPU Architectures

Ilaria Battiston *

Summer Semester 2020-2021

*All notes are collected with the aid of material provided by T. Neumann. All images have been retrieved by slides present on the TUM Course Webpage.

Contents

1 Introduction

Database systems are extremely important in the real world, in all use cases involving management of large quantities of data: of course, they need to provide some guarantees such as scalability, reliability and concurrency, which can be a challenge.

There are scenarios which can either be optimized, or generate some terrible runtime, such as a simple join: its complexity can vary from $O(n)$ (hash join) to $O(n^2)$ with a nested loop. Obviously, the latter case is ruled out since it forbids any kind of scalability.

This can get even more complex when one of the data sources fits in memory, and the other does not: in this case, an additional index structure is employed, to perform lookups in external memory.

When neither fits, sorting is an option, although already a difficult problem; another approach is partitioning, breaking data into sub-problems until they fit in memory.

In general, there are many corner cases to consider, and sometimes it is necessary to make assumptions about the data, making code complexity very high.

Historically, DMBS are designed such that I/O operations are random and expensive, and data is much larger than main memory. Conservative designs are scalable, yet in modern hardware systems main memory size is increasing, reducing I/O costs.

This has consequences for database implementation, since the old architecture becomes suboptimal. Ideally, DBMS should be adapted in order to also maintain durability and good performance.

1.1 Set-oriented processing

Set-oriented processing is a way to avoid using nested loops (accessing every element of one list and then each element of the other) and subsequent squared runtimes, as this operation is not optimal.

It is helpful to perform operations for large batches of data, handling input in sets. Notation takes advantage of relational algebra operations, since these are already set-oriented; even in the case of duplicates (bags), those can still be mapped to unique values.

Algebra needs some extensions for real-world scenarios, such as map, group by or dependent join (predicates which must be evaluated after others).

In short, processing whole batches of tuples can help sorting, hashing, re-organizing the data and creating index structures, ideally achieving worst-case logarithmic runtime.

2 Storing the data

Storing the data is a common issue within database implementation: the application data must be mapped in the file system, taking into account several design choices (such as not even using a file system but just using a physical drive).

However, the application layer should not be aware of how the data is being stored, and storage has to be able to scale up to very large sizes while keeping fast retrieval and update properties.

To simplify the implementation of all requirements and allow decoupling of structures, DBMS employ a layer architecture, in which each layer only communicates with the one below. Layers in this way can be replaced easily independently of the others.

A simplified layer architecture is composed by:

1. Query layer, performing SQL instructions and optimizing them;
2. Access layer, organizing data in pages, managing records and access paths, creating indices and exposing relations or views;
3. Storage layer, containing a database buffer in which data is kept before writing it to disk, which manages accesses and releases of pages;
4. DB, the physical medium to store data.

A more detailed architecture has more layers with specific functionalities and data structures, adding interfaces to have more granularity. Transaction isolation and recovery need to be implemented as well, despite not present in this representation.

Most DBMS nowadays, however, deviate slightly from the classical architecture, delegating hard disk access or buffer management directly to the OS.

2.1 Hard disk access

While designing architecture, hardware power must also be taken into account: storage systems are constantly getting faster and more powerful, and CPU speed had exponential growth until the last years.

Storage hierarchy can be represented with a pyramid: the upper parts are faster but obviously more expensive, and therefore smaller, while lower parts are way bigger and slower. The difference between units can go from ns to even seconds with offline archive storage.

Hard disk is still the dominant external storage, employing rotating platters and therefore leading to an imbalance between random and sequential I/O.

Moving the disk is terribly expensive, so typically a larger chunk is loaded (usually one page, 4 to 16 kilobytes). Reading a full page might lead to wasting time by handling unnecessary information, however larger pages can be used in specific use cases.

The page structure is prominent within the DBMS: often, multiple pages are loaded at once with a read-ahead technique, to avoid multiple successive requests; write-back is implemented by sorting pages before applying changes.

Some pages are accessed frequently, therefore the I/O can be reduced by buffering or caching. This must be coupled with recovery, in particular logging must be accurate.

A basic page interface works in the following way:

- A page is fixed when it is contained in the buffer, and can be manipulated but not discarded, read by multiple transactions at once but only exclusively modified, keeping a dirty state if it has been subject to changes;
- A page gets then unfixed when it is no longer accessed, and memory is released.

2.2 Buffer management

A buffer uses a hash table to manage its pages, in which each entry links to a page stored in memory, and handles collisions with multiple layers of buffer managers. To protect from concurrent access,

latches are maintained.

When memory is full, some buffer pages need to be replaced: dirty ones have to be written to disk first, while clean ones can be simply discarded.

The simplest strategy is FIFO, keeping a linked list with oldest pages and just removing the head, but this does not take locality into account and this method becomes worse as memory size increases.

LRU (Least Recently Used) is one of the mostly used strategies, keeping a double linked list in which a page is moved to the tail if it is unfixed, so that often-accessed pages always stay in it. Latching requires care, since multiple users will try to access the list at the same time.

LRU is simulated through clock or second-chance algorithms, approximating lists with one bit (representing whether the page has been used). When evicting, pages are replaced with an unset bit.

LFU (Least Frequently Used) organizes pages by reordering them according to the number of accesses, but it is quite expensive to maintain.

2Q is a method specific for database systems, based on the fact that not all pages have the same access frequency: all pages are kept in a FIFO queue, and when a page there is referenced again, it gets moved into a LRU queue. This way, hot pages are in LRU, while read-once pages in FIFO.

Since the DBMS already might know which pages will be accessed, having information about the query, it is possible to give hints to the buffer manager and eventually change placement in queue.

2.3 Segments

Segments are the traditional data structure to organize pages, allowing to handle relations, indexes, space management and such. For instance, a DROP TABLE statement implies memory must be de-allocated, and a segment indicates which pages correspond to the entity to be deleted.

A segment is conceptually similar to a file system, however segments do not necessarily have contiguous data and inherent orders (linearity is not guaranteed). Another comparison would be with virtual memory, since both involve mapping.

The interface of a segment contains low-level operations such as allocating or releasing, iterating pages, dropping the whole segment and optionally offering a linear address space (not by default).

This means, for instance, that when a relation occupies a contiguous amount of space and grows over time, it is not given that latter parts will be allocated in the same area: this leads to potential holes which could be closed whenever more space is required by the relation, even allocating more than needed.

However, tuples cannot be returned in the same order as they were inserted, since insertion order may differ from chunk order. This is not an issue since SELECT instructions do not guarantee a deterministic ordering.

There are a few data structures requiring a linear address space, i. e. the iteration order to be the same as insertion order, which can be obtained by adding numbers for pages but is not ideal.

2.3.1 Block allocation

Block allocation can be performed in several ways:

1. Static file mapping, catalog having a reserved static area which is easy to implement but hard to resize;
2. Dynamic block mapping, having pointers to some static catalogs, quite flexible but requires an additional overhead;
3. Dynamic extent mapping, easy to grow with a slight overhead, growing exponentially.

2.3.2 Segment types

Segments can be classified into types:

- Public or private;
- Permanent or temporary (the latter is obviously cheaper, while permanent storage guarantees persistency by writing all data twice);
- Automatic or manual;
- With or without recovery (recovery might not needed when storing one-time data or index structures to pay less overhead).

Most DBMSs have at least two low-level segments, one for segment inventory (keeping track of allocated pages) and one for free-space inventory. In addition to these are built high level segments, such as schema, relations and temporary ones.

Updates are performed either with steal or with force in case of running out of memory: the first one works taking an unmodified page and throwing it away, however if the page is dirty there is a lesser amount of flexibility which leads to complicated management and recovery.

Force, on the other hand, writes every dirty page on disk, no matter when they will be modified next and triggering more writes than needed. For this reason, steal is preferred.

One problem with dirty handling is the multiplicity of users in a database systems: other users should not be allowed to see changes until a commit takes place, locking data and disallowing good concurrency.

One particularly elegant solution to this is shadow paging, relying on the idea of having copies of each dirty page and noting down each modified page in a structure, such that other users can still read the previous version.

There are two downsides, namely the increase on difficulty of implementation, and the lack of locality among data: shadow pages interrupt sequential runs, causing more seek time.

A similar concept to remedy the previous problems is delta files, working in three steps:

1. On change, pages are copied to a separate file;
2. A copied page can be changed in-place;
3. On commit, the file is discarded, otherwise it gets copied back.

Delta can keep either dirty or clean pages, as long as one copy gets destroyed after a commit. Both ways have pros and cons, making expensive either the commit or the abort.

Delta files allow to preserve locality among its advantages, and make recovery easier by having no mixture of clean and dirty pages. However, it leads to more I/O and keeping track of delta pages is non-trivial.

3 Access paths

Access paths can be seen as a general concept for indexes on relations: the DBMS needs several data structures, mainly for space management and retrieval.

A common problem in database implementation consists in where to store incoming data, especially since not all tuples have the same size. A traditional solution for that is the free space bitmap, in which each nibble (an array of half a byte per page) indicates the fill status for each page.

This means that information on space utilization has to be encoded in 4 bits, approximating the status with the linear formula $data\ size / \frac{page\ size}{2^{bits}}$, which however leads to accuracy loss.

Logarithmic scale is often better ($\lceil log_2(free\ size) \rceil$), or interpolation, or a combination of methods since it is unlikely that each page only has a couple of tuples.

When inserting the data, the required FSI entry is computed and FSI gets scanned for a match, then data is inserted. Searching leads to linear runtime, which can be too expensive, especially since the size is so small. This method therefore gets slower and slower, as FSI gets longer.

3.1 Allocation

Performing allocation benefits from application knowledge: larger pieces are often inserted with short amount of time in between, or there can be one single huge item.

Allocation should be contiguous, and to achieve this the interface $allocate(min, max)$ contains size parameters to improve layout, help deciding location and reduce fragmentation. Tuples are more difficult to store, and can cause over-allocation. Unfortunately, most programming languages do not support such options.

Taking a vector and continuously increasing its size, for example, will at some point lead to some misalignment if memory hasn't been allocated before: in this case, pointers can be employed to know next location, but MALLOC cannot give bounds on the overhead.

3.2 Slotted Pages

Slotted pages are useful to allocate pages a fixed size on a segment, using slots supporting combinations of page number and slot number to retrieve information.

It can happen that a record overflows: in that case, it is moved to a new page, and a pointer gets stored in the previous slot. This forward mechanism enables to keep a structured storage while not having to perform cascading updates.

When a record which has already been moved overflows again, in fact, there is no need to create a chain: the existing link can just be replaced, only forming a chain in case of an index. This system, however, relies on the hope that most tuples will not be moved, instead they will be never updated.

Slotted pages are implemented by storing tuples in a page in which data grows from one side and slots from the other. Each page will have a header and a new slot getting added as new data items

get added, until it is full when two sides meet.

In reality, data items have variable size, and it is unknown how many slots will be there. Updates and deletions can be complicated as well, since they might change the size: in this case, some data needs to be moved with periodic compactification to optimize space consumption.

Header parameters are:

- LSN, flag to check whether there have been crashes;
- slotCount, number of used slots;
- firstFreeSlot, pointer to the first usable slot;
- dataStart, lower end of data, which should not be updated;
- freeSpace, available space after compactification, which can be performed when this number is bigger than the effective free space.

The effective free space can be calculated with the following formula:

$$EFS = pageSize - headerSize - slots \cdot slotSize - (pageSize - dataStart)$$

Scanning slots to find a free one can take quadratic runtime, therefore pointers to free slots are employed; however, performance of slotted pages is still not ideal.

Conceptually, each slot can be assigned an offset and length of the respective data item, even if empty tuples must be distinguished from deleted one through offset length.

The main issue with this method is the lack of available space in the case a transaction to shrink a page aborts and the following to add data commits, making it necessary to store a further bit (flag) in the TID which would be stored inside the slot: if the TID is valid, the entry is redirected.

3.3 Record layout

A record layout has the purpose of materialize and serialize the tuples, trying to avoid the linear worst-case runtime of accessing an attribute.

Instead of offset, the length is stored:

1. Each tuple is split in two parts;
2. The header has fixed size, while tail is variable;
3. Header contains pointers to the tail (beginning can just be computed);
4. Attributes are accessed in constant time.

For even better performance, attributes can be reordered by decreasing alignment, i. e. the number of bits of its type. Sorting works since alignment is always a power of two, and it helps identifying the data type.

Variable length-data, which modern processors easily handle, is placed at the end and given alignment 1 by default, without wasting space on padding.

NULL values can either be stored as invalid or identified with an additional bit, which allows to omit actual information to save more space. This is useful when NULL values are common.

3.4 Compression

Some DBMS store data in a compressed way, with the aim of saving space and most importantly improve performance: reducing size reduces bandwidth consumption as well, decreasing I/O costs.

General compression schemes work better as the data gets larger (for instance LZ77), but every DBMS should be able to update information without decompressing and recompressing everything. Page size is also a problem, since it can vary.

Compressing large chunks is therefore not an option, so it is performed tuple by tuple. Huffman encodings are a possibility (building decision trees to take advantage of similarities in data) except for the problem of not knowing the tree to decompress, which is usually not serialized to avoid the additional overhead.

To remedy this, adapted Huffman encoding is an algorithm in order to reconstruct trees by changing encoding to make frequent letters cheaper, so that trees will only depend on frequencies.

Performance of such technique is quite bad, and might not balance the time saved from I/O. A reasonable compromise is byte-wise encoding through VLE, a bit worse in terms of size but faster in the long run and able to handle null values.

Compressed data, however, has variable length, and sometimes depend on precedent compression. Lookup tables help precomputing and locating quickly.

Dictionary encoding is a particular method used for strings, since data tends to be redundant: it stores strings in a dictionary, and only the ID is contained in the tuple, greatly reducing the total size.

3.5 Long records

Long records consist in a long-term solution in case the previously mentioned methods cannot be used: the root of the idea are pages which do not fit into a single page (BLOB, for instance).

These could be stored with a pointer to the attribute to a whole page, or eventually a long chain of pointers, but this implementation is problematic since it implies reading a potentially unbounded amount of pages (troublesome for buffering or locking). Furthermore, updates involve multiple allocations or releases in the system.

Instead, tuples are stored separately from BLOB objects, and not parsed until a query is actually performed; sometimes IDs are assigned to BLOBs, hiding their complexity.

In most systems, it is not easily possible to perform many operations on such data types, for instance looking at the content, and costs are generally high even though processing is simple.

When objects are in the order of MBs, data is again mapped to pages, storing information in a B-tree like fashion. Each node contains a part of the BLOB and a key representing a range (offset) for searching. Relative offsets allow logarithmic worst-case runtime when updating, the only downside is the overly complicated interface.

Using an extent list is simpler, having a chain of real tuples pointing to BLOB tuples with unlikely worst-case scenarios. BLOBS can be manipulated as a whole piece, but are harder to update in pieces.

Postgres employs TOAST, a mechanism to shorten long tuples.

Other approaches encode short BLOBs in the TID, and uses the full procedure only for actually large objects. The ones smaller than the page size are stored in records.

3.6 Index structures

Index structures come in handy when a small part of all the available tuples need to be fetched (either point or range queries), or when making sure primary keys exist.

3.7 B-trees

B-trees are the most popular data structure used for external storage: every tree has a degree k and each nodes has $k \leq n \leq 2k$ entries. All leaves must have the same depth, which can be bounded with the logarithm of the size (usually 3 or 4).

Given they are particularly easier to implement and generally faster than red-black trees, B-trees are widely used in their variant of B+-trees, which are even simpler and store TIDs only in leaves. Inner nodes contain separators which may or may not occur in the data.

Each node contains:

- LSN for recovery;
- Upper node or leaf node marker, the latter being a value which cannot appear in the former;
- Number of entries;
- Key/TID/child;
- Next leaf node (only for leaves, a flag).

3.8 B+-trees

3.8.1 Insertion

Insertion is performed in the following way:

- The leaf page is looked up;
- If there is available space, the entry is inserted;

Bulk insertions can be optimized, since the tree needs to be rebuilt several times and there is a lot of random I/O. Each database system implements it in its own way (Postgres for instance loads a whole file through a COPY instruction), trying to improve locality.

Sorting is an option since the order is not random, but it triggers the worst-case scenario of space consumption: all the pages will be half-full, since the rightmost one is split.

Since a database is usually aware of bulk loading, this can be optimized further:

- Data is sorted;
- Data is spooled into leaf pages, filling them completely and storing the largest value in each;
- Separators are spooled into inner pages, again remembering inner separators;
- This is repeated until one page is left (the root).

This method provides a compact B+-tree, performing only sequential I/O with all full pages except for the rightmost and improving locality on disk.

Inserting in an existing tree requires a level of merging to be employed in the above procedure, beginning with merging all data and then starting a new chunk once a page would contain only entries from the original tree and then merging separators.

Under-full pages can therefore be accepted to avoid moving all data already contained in old pages but obviously not guaranteeing adjacent pages anymore.

3.8.2 Deletion

Deletion is the hardest operation to perform in B+-trees, since it may violate the strong criterion of having at least half the pages half full.

In case the removal of an entry violates this property, logarithmic cost is not guaranteed anymore, therefore the tree needs to be rebalanced: this happens through merging pages with their neighboring ones, updating separators.

Finding neighbors, however, can be a difficult task: the succeeding value can be located far away in the structure, hence a simpler solution is to just choose another child of the parent, achieving constant time.

Many systems do not implement a proper deletion logic, since it is so complicated, and just accept pages under-full, eventually deleting them completely. In the long run, however, if the value of newly-inserted entries is much bigger than the deleted ones, free space may be never used.

3.8.3 Range scans

Range scan is the operation reading all entries within a range. This is efficient thanks to pointers to next element, therefore only the first lookup needs to be performed.

This works particularly well when leaf nodes are stored consequently on disk, or cached (clusters).

3.9 Compound keys

Compound keys are compared lexicographically according to the following rule:

$$(a_1, a_2) < (b_1, b_2) \leftrightarrow (a_1 < b_1) \vee (a_1 = b_1 \wedge a_2 < b_2)$$

Other than this, they are quite similar to atomic keys. Search can be performed by imposing a prefix of the compound and using anything else as a range, however the other way around can be difficult to implement.

3.10 Non-unique values

Users may decide to apply an index on columns containing many duplicates, which need to be retrieved for updates and deletions.

The solution in this case is to only index unique values, appending a TID to the non-key attributes working as a tie-breaker. Space complexity grows sightly, but logarithmic access is still guaranteed.

3.11 Concurrent access

Concurrent access cannot be handled through simple page locking or latching, since this method only works for leaf nodes; inner pages locking might cause missed values.

The classical technique is lock coupling, working in the following way:

- A thread locks both the page and its parent;
- The parent is released as a new lock is acquired on the child, and so on;

This prevents conflicts and deadlocks, as latches are ordered and pages can only be split when the parent is latched.

Concurrent insertions might cause the split to propagate up, until the root, making lock coupling not an option anymore since the parent may need to be locked, causing deadlocks.

Another solution is safe inner pages: while going down, a check if the inner page has enough space is performed, and if not the page is split. This method ensures going up of never more than a step.

An alternative to this is just restarting the process, releasing all latches only if the inner node must be split, and keeping all latches up to the root in the new execution. It is not ideal since it involves less concurrency, yet happens extremely rarely.

3.12 Partitioned B-trees

Bulk insertions in trees work well if they are not too frequent, but usually require taking the index offline and impacting optimization.

Partitioned B-trees are created by adding an artificial column (partitioning key) causing all new values to be larger than the existing ones. Partitions are largely independent and can be efficiently merged, but lookups need to access all partitions and deletions are complicated to implement.

3.13 Variable length records

In reality, entries can have variable length, making it harder to check whether a page is half full. Having keys of different length (or alternating) can produce multiple tree structures and depths, depending on the separator.

This emphasizes on the importance of choosing the right separator, a non-trivial issue to achieve logarithmic height. The main idea behind algorithms is to pick the smallest value within 20% around the median whenever a page overflows.

3.14 Hashing

Hash tables are a data structure used for single, infrequent lookups, providing amortized expected constant runtime through hashing keys either with chaining or open addressing.

To map keys to their hashed values, a prime module function is used. Calculations can be sped up with the following methods:

- Making the modulo operation cheap by precomputing some magic numbers to then perform a sequence of multiplications and shifting;
- Making sure that N is a power of two;

- Converting 32 bits to 64, multiplying by the hash table size and shifting again.

DBMS benefit from this implementing hash tables as indexes, but downsides are many, such as random I/O and expensive rehashing. In practice, hash tables are mainly used for primary keys, since the number of collisions should be less due to uniqueness of values. Duplicate values can be handled by reindexing them, or just with a tree.

In real life it is hard to predict the table size beforehand, and rehashings are to be avoided, due to their expensiveness: extendible hashing is a technique in which pointers lead to buckets (pages), even the same one, obtaining variable depth.

When a bucket overflows, it is split, increasing the depth and adding a bit. Pointers must be unshared and items are redistributed. Exponential growth, however, should stop at some point, falling back to chaining.