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.