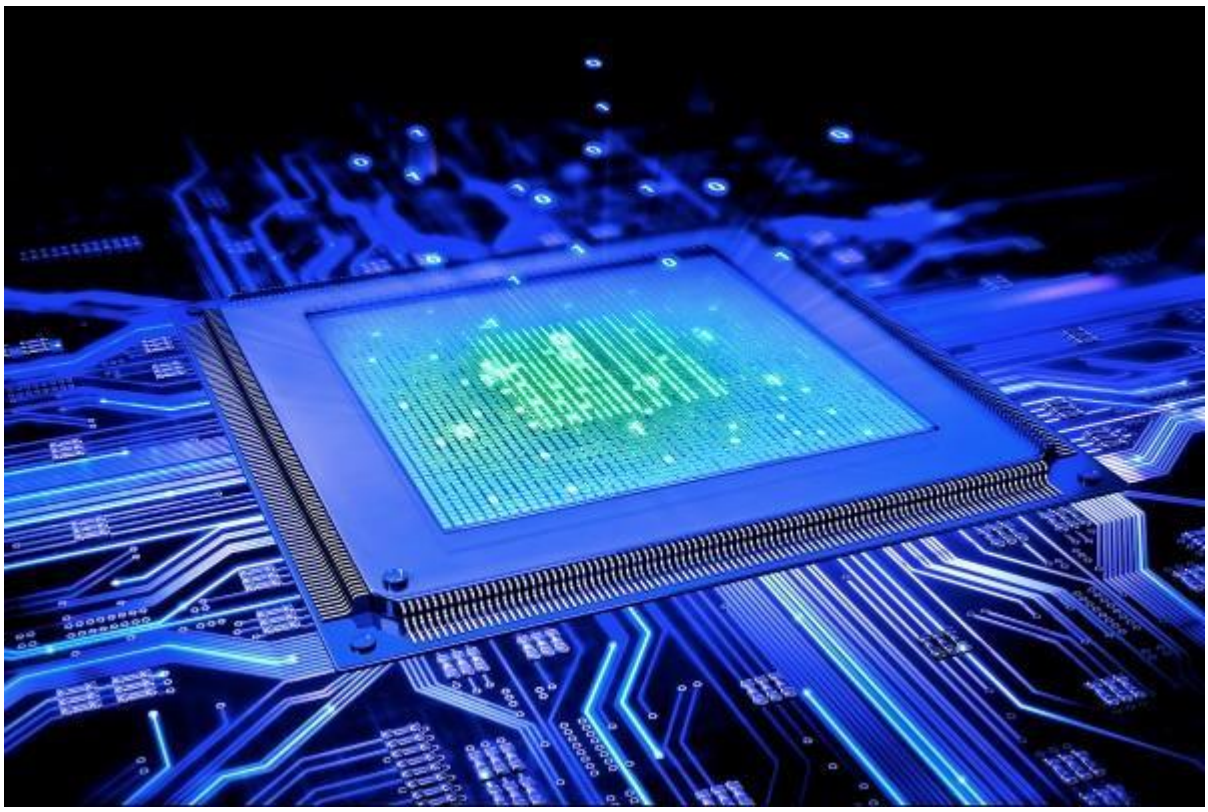
Database implementation



Team :

- Mehdy BENNANI, mehdybe@protonmail.ch
- Matthieu DENIS, matthieudenis12@gmail.com
- Alexandre GOMMEZ, alexandre.gommez@agts.fr
- Hubert de LESQUEN, Hubert.de-Lesquen-du-Plessis-Casso@polytechnique.edu

TASKS.

The purpose of this project was to create a relational database following the TPC-H schema and keep it in the main memory. Among the different tasks, we had to create:

- Database storage on the disk,
- Projection (with duplicate elimination),
- Selection,
- Join,
- Group by,
- One aggregation function

The implementation of the algorithms and data structures must be implemented in several steps:

- Single-threaded execution,
- Multi-threaded execution,
- Distributed execution (Spark).

INTRODUCTION.

The first step of the project was the programming language selection, among the different propositions. Based on the understanding of the restrictions of programming functions and on the different skills of the team, we first chose to develop the project in C++. After a while and more explanations concerning programming languages restriction, we finally changed to Python and restarted the project.

Literature review.

Data structure.

The two main data structures are row stores, and column stores. Their main characteristics are summarized in the table below:

Criterion	Row store	Column store
Data loading	Slower	Rapid
Query efficiency	Not much optimized	Optimized
Frequent transactions	Very effective	Not much support
Compression	Not efficient	Very much effective
Analytic performance	Not suitable	Very effective

Reference

Considering the use of our database, we preferred to focus on the Column store structure, very efficient for database consultation and queries.

Selection - σ.

The selection operator consists in extract a subset of rows from a table, according to a criterion.

The most known algorithms are :

Algorithm	Complexity
Linear search	$O(n)$
Binary search	$O(log\ n\ )$

Projection - π.

The projection operator consists in keeping only selected columns of a table, which means delete unwanted columns of this table. The most important difficulty of this operator is then to remove duplicate rows from the table.

Natural join - ⋈.

The natural join operator consists in merging two tables, sharing at least one attribute. The attributes of the second table are concatenated to the first table when the values of the junction attribute are matching.

The more important algorithms are:

Algorithm	Complexity
Nested loops join	$O(n \times m)$
Block nested loop join	$O(n + m)$
Hash Join	$O(n + m)$

Algorithms choices.

Selection - σ.

We chose the linear search

Projection - π.

By the architecture we have, we can project any column by the command: *my_table[col_name]* which is not an optimized pandas data access method such as *iloc* for instance.
Then we use a support table to store the data and check in the support table if the row is already there if it is the case we pass to the next line.

Join - ⋈.

We chose the hash join because it's the fastest method.

Group by.

We chose to implement our own algorithm, written by hand.

Experimental results.

You will find a small documentation in the python files to help understand what is implemented and how to use it.
We start by loading our data in the main memory, then for each implemented algorithm, we execute it on a certain example while timing it. The results are as follow:

(in seconds)	NOT THREADED	THREADED
Join	15.8	16.4
Selection	2.14	2.11
Selection by attributes	15.2	13.9
Projection	0.006	NOT IMPLEMENTED
Group By	0.02	0.05

As we can see, on the examples runned on, threading our queries doesn't significantly improve performance, and with group by, it even more than doubles the execution time.

Why is that? By "threading", we split the tables into several chunks, then we feed them at the same time to our algorithms. If the tables are too small, threading is useless (it is the case for most tables that we have).

And in group by, we didn't totally thread the algorithm, only the underlying select in it, because our handmade algorithm can't support by construction multi-threading.

Which algorithms/data structures do existing dbms use?

The most frequently used data structures for one-dimensional database indexes are dynamic tree-structured indexes such as B/B+-Trees and hash-based indexes using extendible and linear hashing. In general, hash-based indexes are especially good for equality searches
For multidimensional database, the following data structures are the most important:

- Grid file
- Multiple-key index
- R-tree
- quad tree
- bitmap index

The most used algorithm for sorting files is the merge-join, because sort-merge join is frequently used in databases.
To make our discuss and choose which algorithm to implement we used this document rich in insight : https://www.cise.ufl.edu/~mschneid/Research/papers/HS05BoCh.pdf

Our experience

At first sight, this project is perfect to use our skills in C++, as the need for a low-level language to have good performance is here. After facing lots of issues and dozens of hours of work on memory leaks and polymorphic arrays, we had to change. However, let us explain to you how this first version (you will be able to find source code in our delivery).
We first wanted to do a polymorphic vector of pointers on an abstract class and make a column per type (int, float, string, and date). But the biggest issue is that we cannot use strings because of the static length of strings. We could not use templates in this case!
Also, with the evolution of our understanding of the constraints, the ease of starting over the project in python compared to the increasing number of issues in C++ was more and more tempting. The last issue about a memory leak in a destructor made us change from C++ to python.
We all agree that the project has way more value in C++ or Java thanks to the ability to manage memory space and play around with it, however, the time was too short and collided with other deadlines. Some of us might redevelop it in C++ in our free time.
We found a real interest in this project, searching for an algorithm that is the most optimal and showed us the history of the algorithm. Trying to implement it was also a real challenge for us!
The freedom around the project has to side, one very interesting freedom, we can nearly do whatever we want! But this freedom has a cost, being lost in the wild can be sometimes confusing, and finding our way to our goal is way more complex than a defined path.
This project helped us to discover and know each other (as we all come from different backgrounds and different schools). It also took us some time to properly understand how everyone works.
We also faced some issues on the side of docker, even if some of us have a bit of experience in this field, the manipulation was very complex due to network issues (as we did not write the docker file, the comprehension is complex).
From the python side, the coding is way more pleasant, first thanks to the absence of memory management and type. And, thanks to the various documentation on small tools available (as pandas), the fact that threads do not increase the speed is frustrating! Docker are not working properly as the tables are huge and the space, I can allocate to my docker are small!
Our feelings about the language are mixed, happy to do it in python and frustrated because we could have done more in C++ with more time and more practice!

Conclusion.

We managed to nearly do everything in the project except the spark implementation and the row store implementation. However, we are proud of this project and hope it fit the expectations. We learned a lot about the system for big data and how a good design and good comprehension can save a lot of time later! We also understood that what is currently in use is really optimized and it is difficult to go further at least at our current level.