Indexing techniques for improving structured, semi-structured and unstructured database performance

Brenda Taboga

Haaga-Helia University of Applied Sciences
Business Information Technology
Research
2023

Abstract

Author(s)
Brenda Taboga

Degree
Bachelor of Business Information Technology

Report/Thesis Title
Indexing techniques for improving structured, semi-structured and unstructured database performance

Number of pages and appendix pages
53 + 1

The main purpose of this thesis is to study the difference between structured, semi-structured and unstructured data. To set up the right database, it is important to know the difference between relational databases, which store structured data, and NoSQL databases, which store semi-structured and unstructured data. Once the various concepts have been assimilated, it is time to set up the appropriate database.

Once the database has been chosen and implemented, it is vital to maintain good performance. Since data is a vital commodity these days, it is important to be able to extract and analyse it efficiently. There are many tools available for this purpose, including indexes. Indexes make it possible to extract data efficiently. To do so, it is important to know which indexes to use, depending on the data and queries being executed.

There are many different indexes for indexing data in the most appropriate way. In addition, some databases index data automatically, so that it is important to keep track of all the indexes in the database. Moreover, it is important to create only useful indexes, because creating indexes, even though it improves performance, requires storage space. So it is crucial to create them only where they are needed.

Through the tests, it is possible to get a partial view of the performance improvement when creating indexes. Last but not least, it is also possible to compare the time improvement between different indexes on the same database column or field, to illustrate the theoretical part.

This thesis proposes some tools to give an initial idea of how to set up and analyse indexes in databases based on diverse data models.

Key words
Structured, semi-structured, unstructured, data, indexes, performance, NoSQL, relational, database.

Table of contents

1 Introduction

First of all, I would like to explain the reasons why I choose this particular subject. During my first two years of Bachelor at the HES-SO, in Switzerland, we mainly focused on databases, especially Oracle databases. I was immediately interested in all aspects of databases, from design to optimization. Afterwards, during my exchange at Haaga-Helia, I had the chance to have a course called "Database Developer" in which we focused on data indexing. Since then, data indexing has become a subject of great interest to me.

To begin with, it is necessary to talk about the importance of indexes. Nowadays, one of the most important values of companies is data. This means that the efficiency with which data is accessed is vital for the company's business. Indexes are one of the main solutions to improve this efficiency. Without an index providing a map to find the data, the database will have to search the entire table. Indexes are therefore like a dictionary to access data efficiently (Strate 2019, chapter 1).

As mentioned above, indexes are one of the main solutions to improve performance. However, it is necessary to set up the right indexes according to the data we have. Indeed, a certain type of index will have no effect on some columns but will greatly improve the performance of another. This represents the great challenge when choosing the right indexes.

A database without an index can lead to many problems. As an index works like a dictionary, it is easy to imagine how time-consuming and tedious it would be to search and find a word in an unordered dictionary. In a database, this translates into much longer and more resource-intensive queries. Now we could imagine that several users are using the same database, and that they all access it at the same time. This will take much more resources and time from the database. However, with the creation of appropriate indexes, it is possible to significantly reduce the resources and time needed to access the data. This will save time and energy. In addition, different users can have access to data much faster, which will increase their efficiency.

In order to set up different databases and analyse the performance using these different tools, I chose to compare the indexing of structured and unstructured data. To set up an adequate environment, I used Docker. In order to do so, I set up a database instance through Docker and analysed the performance to find out what kind of index type is appropriate. This research has allowed me to deepen my knowledge in terms of data indexing and to compare different types of indexing. Last but not least, it has given me the tools to choose the appropriate type of indexing while being aware of the possible compromises that it can bring.

1.1 Research question and objectives

The research question will allow the writer to make a connection between current and future knowledge. Through appropriate methods and advanced literature review, the writer will understand the core subject of the research. One of the most important aspects is the research question. Indeed, even though developing relevant research questions and objectives is hard, it is necessary to realise a successful research project (University of Newcastle Library 2023).

1.1.1 Research questions

- What is the most suitable index type according to the structure of the data?
- What are the differences in performance improvement between the different indexes?
- How to set up and compare indexes?

1.1.2 Research objectives

This research will make it possible to immediately understand the stakes of a good data indexation. It will also allow the reader to know the right method to use for his database. Several analyses of different databases will have to be set up (using the appropriate database depending on the structuring of the data). Beforehand, an analysis of the different structuring of the data as well as an analysis of different indexes will have to be carried out.

1.2 Research method

For this research, I will use quantitative research. This method will allow me to gather all my numerical data and to rank, measure and categorize them through analysis (University of Newcastle Library 2023).

Regarding the techniques and tools, I will use “Observation”. This technique includes counting the number of times a phenomenon occurs or translate observations into numbers. This method will be useful when implementing and analysing the indexes of my various databases.

The second technique I plan to use is “Experiments”. This method will help me to test my hypothesis and to analyse the effects through experimentation. In my case, the experiences will be the different implementations of my databases, as well as the different indexing techniques implemented.

1.3 Utilization of the results

The results of this research can be used in the creation of different databases. Through this thesis, the reader will have tools to decide what kind of indexing exists and how to set them up and compare them. Furthermore, through the different analyses, the reader will also know what to expect according to the different types of indexing used. The results of this research may be useful for mainly database administrators concerned about the performance of their database.

1.4 Scope

Regarding the scope, this thesis will focus on indexes implementable by the DBMS used. Those DBMS mainly include Microsoft SQL Server and MongoDB. Moreover, the theoretical part of this research will concern optimization through indexing only. This means that only indexes for structured, semi-structured and unstructured data will be mentioned. Methods such as query optimization are not part of the research, although queries are also an extremely important element for optimisation.

Queries are important to analyse before setting up an index. It is necessary to analyse which queries are executed the most to know where the indexes should be set up. As some types of indexes (e.g., clustered indexes) can only be present once in each table, knowing which columns are frequently used is vital. Moreover, it is not possible to talk about performance without mentioning queries. The way queries are structured has a great influence on response time and performance (Strate 2019, chapter 12). As mentioned above, queries have a direct link to indexes and performance, although these will not be analysed in this thesis. Another aspect that will not be mentioned in this thesis are the relationships between tables, which also have a great influence on the performance of the database. Knowing which primary and foreign keys to set up and optimise will result in much better performance and time efficiency.

2 Theoretical framework

To support my thesis, I will rely on the knowledge I acquired during the courses of Database Developer, as well as on several different books. Indeed, by combining the two, I will be able to build a knowledge base allowing me to analyse the different performances of my indexes on my implemented databases. To begin this theoretical framework, it is essential to discuss and cite the different types of data. Through various works and sources, I will start by laying the groundwork for a better understanding of the empirical part.

2.1 Structured Data

Structured data forms the bulk of the data we usually use. It is data that can be categorized and quantified. This data is predefined and formatted before being stored in a relational database. Usually, structured data is stored in tables, with clear relationships between them. Relational databases organize data in rows and tables (Patel 2020, chapter 5). Examples of structured data include dates, social security numbers, customer names, addresses, email prices, and so on. To extract, manipulate and analyse this data from a relational database, we generally use a programming language, named structured query language (SQL) (Patel 2020, chapter 5).

As mentioned above, structured data is data you can organise in tables with rows and columns. Structured data is quantitative data which means that data contains numeric values, so you can measure it and use it to make statistical analysis. In brief, that this data is easily understandable for analysis tools, such as Business Intelligence, Machine Learning, and many others (Ryan 2021, chapter 1.4).

There are many other advantages. We can point out, for example, that they are easily accessible by a lot of tools and easily understandable for users. Machine Learning and Artificial Intelligence tools can also use them. However, there are also the following disadvantages, especially for deep learning (Ryan 2021, chapter 1.5):

- Limited usage: Sometimes, structured datasets are too small to feed deep learning. This disadvantage depends on the domain. There are many different domains in which the structured dataset is large enough.
- Limited storage options: To store these data, we must use databases based on rigid schemas.

2.1.1 Relational databases

Relational databases form the bulk of databases used today. These databases are developed for transaction processing (Harrington 2016, Chapter 1). Relational databases are composed of tables that are the centre of the relational database universe. This means that for a quality database, it is necessary to look at the design of the tables. The tables will mirror the business of the company. A table, in the world of relational databases, is a combination of columns and rows. Each column should have a name, unique for a specific table. The columns, on the other hand, must have a certain data type. Depending on the DBMS, the types may vary (Ardeleanu 2016, Chapter 1).

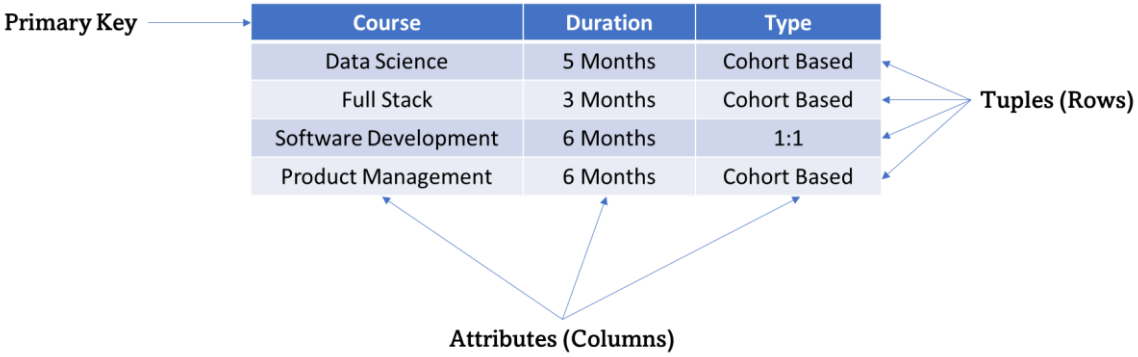


Figure 1. Relational Model in DBMS (Adapted from Board Infinity 2022)

Transactions are also often associated with relational databases. They are commands that include one or more SQL queries executed on a dataset. At the end of the transaction, you can choose to execute a COMMIT or a ROLLBACK on it. This will execute the changes or roll them back respectively (Silpiö 2010). Transactions must have the following well-defined properties:

- **Atomicity**: it implies that all changes are made or none.
- **Consistency**: the data must always be in a consistent state.
- **Isolation**: each transaction performed must be isolated from the others and therefore not impact them.
- **Durability**: once the transaction is completed, even if the database encounters a problem, the data must persist.

Secondly, databases may have constraints. There are column constraints and table constraints. For example, there are **NOT NULL** constraints, which guarantee that the column will not be empty, **Primary Key** constraints, which will identify the row, **UNIQUE** constraints, which will force the entry of unique data for each different row, **CHECK** constraints, which will check that the column follows a rule defined by the creator of the constraint, and **Foreign Key** constraints, which will allow you to point to a row in another table (Ardeleanu 2016, Chapter 1).

Finally, the isolation level will also influence performance. In the case of the various tests, the default isolation level is used. However, depending on the needs and users of the database, other levels of isolation will need to be put in place. As mentioned, there are different isolation levels, these are the levels commonly used in SQL Server (Silpiö 2010):

- **Read uncommitted**: No locks used, which means that this level does not prevent any problems.
- **Read committed**: Locks cover all selected rows and are released after reading the rows. It prevents *Dirty Read* (when the transaction manages to read a row that another transaction has written but not committed).
- **Repeatable read**: Locks cover all selected rows and are held until the end of the transaction. It prevents *Dirty Read* and *Non-Repeatable Read* (when another transaction manages to modify a row that our transaction already read).
- **Serializable**: Locks cover the search predicate and are held until the end of the transaction. It prevents every problem such as *Dirty Read*, *Non-Repeatable Read* and *Phantom* (when another transaction manages to add a new row, but our transaction already made an operation, which means that there will be more rows if we re-execute our transaction).

As can be imagined, the higher isolation level is (and therefore prevents more problems), the more performance will be affected, and more resources will be required from the database. As mentioned above, for my various tests, the default isolation level was used (Read committed).

2.1.2 Importance of modelling: database tables reflect the target system

To start with, it is mandatory to create a model before creating the tables in the database. This relational data model has characteristics for the different rows and columns. In addition, the model will also make it possible to define and represent the relationships between the tables (Harrington 2016, Chapter 5).

Now, we will concentrate on the effects of a bad database design. We could imagine an *Order* table containing the following columns: *customerName*, *customerAddress*, *nbItems*, *dateOfOrder*. You will quickly notice that when a customer has several orders, there will be redundancies in the customer's information, because for each order, the customer's data must be added. Moreover, if the customer's address changes, the address of each order will have to be changed, which is not optimised at all and even dangerous according to the constraints of the table. This is only an example of one of the problems that can occur. Many different types of problem can arise as a result of poor database design, such as data consistency problems, problems when inserting and deleting data (Harrington 2016, Chapter 3).

It is important to note that the schema creation process has different stages. The first step is to create entity–relationship (ER) model. This type of model will allow to create a conceptual model. It has two levels of definition, one rather simple and one more complex. The creation of this model will require, at first, communication with the different future users of the database to define their needs. The simple model will contain only entities and relations to visualize the schema. This schema is therefore mainly intended for the end users, so that they can approve it. The most complex level of this model has concepts that go beyond the simple. This model will define constraints, such as which columns can be null, which data types are required for each column, etc. This model is therefore intended for people who will implement the database (Teorey, Lightstone, Nadeau & Jagadish 2011, chapter 2).

The second step is to transform the ER model into a logical model. More often than not, this transformation will be done by the software on which the first model was created. This transformation follows several strict and precise rules. Among other things, it will make it possible to transform the simple relationship between two entities into a primary and foreign key pair or even, depending on the cardinality of the relationship, to create an intermediate table (Teorey, Lightstone, Nadeau & Jagadish 2011, chapter 2).

My tests were not about the importance of schema creation but about optimization through indexes, so I simply created a logical model without going through the ER model. In order to do that, I chose a basic model which corresponded to the needs of my tests.

2.2 Unstructured Data

Unstructured data generally form a large mass of data that cannot be processed in the same way as structured data. These data are generally qualified as qualitative, meaning that they cannot be analysed with conventional data tools, such as Power BI or Excel for example. Since these data have no real defined schemas, it is necessary to store them in a so-called non-relational database (NoSQL). Example of unstructured data include audio and video files, images, texts, reviews, and tagged formats such as XML, HTML and JSON (Ryan 2021, chapter 1.4).

Qualitative data is data that can be defined as more descriptive, related to language. To be able to analyse it, it is necessary to categorize it by creating meaningful themes with the use of AI tools or by creating collections. There are many advantages to unstructured data, such as the variety, the fact that these data do not have a structured model, its speed of collection because it does not need to be formatted before being stored and the huge amount of data that can be stored. But obviously, there are also some disadvantages. Among these, we can mention the irregularities and ambiguities that are problematic to traditional analytics techniques and the obligation to use specialized tools for the analysis of unstructured data (Isson 2018, chapter 2).

2.2.1 Non-Relational Databases

A non-relational database is a database that does not use tables, columns, and rows unlike relational databases. These are sometimes called NoSQL which literally means “Not Only SQL”. There are different types of NoSQL databases (MongoDB):

- **Document databases**: this type of database stores data, usually in JSON, in a document. This allows a wide variety of different data types to be stored (string, int, float, objects, arrays, etc.). The advantage of this type of database is that there is, by default, no schema but it is still possible to use a schema for validation. Not having a schema allows great flexibility but it considerably affects the data processing(MongoDB).
- **Key-value databases**: these are relatively basic databases that store data in two parts: a key and a value. Through the key, which is unique, it is possible to retrieve the stored value. However, this simplicity does not allow to store complex data (MongoDB).
- **Graph databases**: these databases are very specialized; they have a structure made of nodes and edges. This allows you to create many relationships and you can easily add more nodes. However, since the relationships are not clearly defined, if at all, it is very difficult to search for data. Moreover, there is no specific language, which complicates the search even more (MongoDB).
- **Wide-column databases**: these databases are more like relational databases. They also use tables, columns, and rows for data storage. However, the main difference with

a relational database is that the row type does not have to match each row, which allows to store data across different servers (MongoDB).

To sum up, it is interesting to use a non-relational database when the stored data need flexibility and therefore cannot be simply stored in a relational database. In terms of performance, these databases offer many advantages depending on the data stored (MongoDB). The advantages of a NoSQL database, especially a document database, are the followings:

- **Ease of use**: Because this is not a relational database, the scaling out is way easier. Moreover, it is easier to represent complex relationships with only one document. It is also way easier to experiment (Bradshaw, Brazil & Chodorow 2019, chapter 1).
- **Scaling**: Document databases are also created to scale out. It means that this kind of model makes it easier to split data across multiple servers (Bradshaw, Brazil & Chodorow 2019, chapter 1).
- **Features and performance**: With NoSQL databases, there is a lot of features available (indexing, aggregation, file storage, and so on). In addition to this, the implementation of these features has very little effect on performance. For example, in the case of MongoDB, its maintenance of this streamlined design allows great performance (Bradshaw, Brazil & Chodorow 2019, chapter 1).

To analyse the performance of indexing techniques for unstructured data, the non-relational database MongoDB will be used.

2.3 Semi-structured Data

Semi-structured data is data that does not have a strict structure unlike structured data, but it does have some structural properties unlike unstructured data. This means that they cannot be stored in relational databases. This type of data includes for example emails, because it is possible to categorize them to a minimum, but they are not structured. We can also mention CSV, XML and JSON as examples (Wolff 16 November 2020).

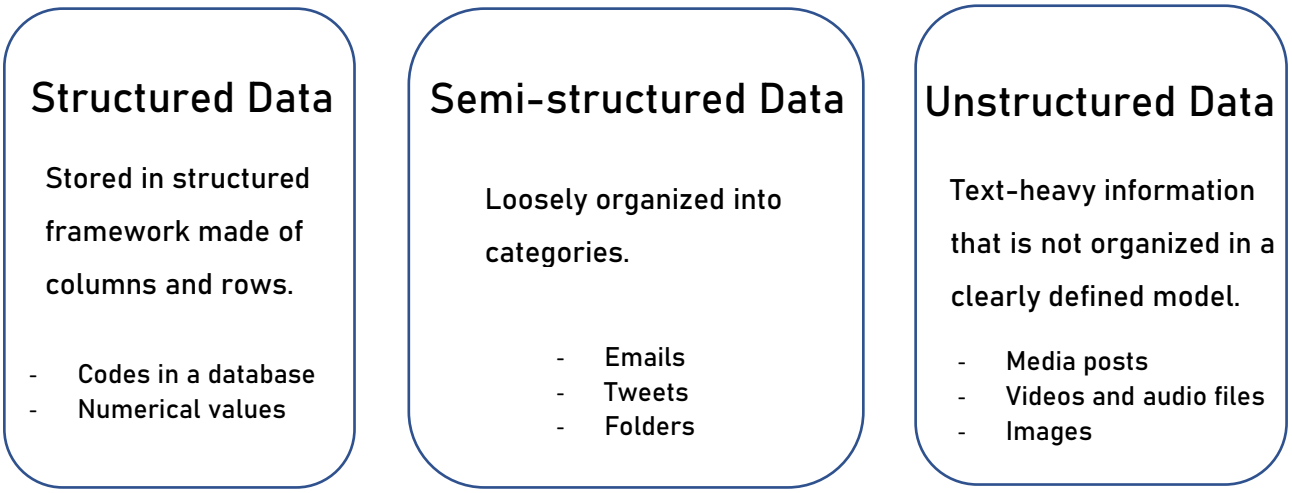


Figure 2. Unstructured vs Structured Data (MonkeyLearn)

With this illustration, it is easy to understand that it is a combination of structured and unstructured data. We can also realize that in this category of data, we are dealing with data that clearly cannot be totally structured, but we can still categorize them in several ways. Because of their lack of structure, it is necessary to store semi-structured data in non-relational databases (NoSQL). The flexibility of these databases allows the management of unstructured and semi-structured data.

The main advantage that we can mention is its better ability to be analysed and processed by Machine Learning compared to unstructured data. However, it takes time to format them, unlike structured data which does not need it. In addition, semi-structured data are way more storable and portable than unstructured data (Ryan 2021, chapter 1.4).

2.4 Indexes for structured data

A database index is a data structure that organizes data records on disk in a way that facilitates efficient retrieval operations. This means that, if queries use the indexes correctly, indexes allow to quickly locate the different rows in a database by using keys (Petrov 2019, chapter 1). Indexes can be created on one or more columns of the database to ensure quick access to the data. However, this will require storage space to store them. In relational databases, indexes are always sorted, which makes it easy to find the data. There are one-level indexes and multi-level indexes.

To begin with, it is important to specify that the data in a database is not stored in a sorted way. This means that when a special request is made, it is necessary to access all the data. As far as searching is concerned, as mentioned above, it is necessary to execute a table scan when the database needs to be searched in its entirety. This means that the minimum of blocks / pages

accessed will be 1 (if we are lucky). Otherwise, the maximum number of blocks / pages accessed will be *n* (the total number of data items in the database) (Silpiö 2015).

Secondly, when indexes are created, data can be accessed more quickly. Indeed, during an index-based search, the minimum number of blocks / pages accessed will be 0 (if, for example, the data is already in memory). Otherwise, the maximum number of blocks / pages accessed will be 1. This is because, thanks to the index, when the right data is found, the pointer goes directly to the page / block where the data is located (Silpiö 2015).

2.4.1 Dense and sparse indexes

One-level indexes point directly to the data in the database. Those indexes can be separated into two distinct types of indexes (Silpiö 2020):

1. **Dense index**: This type of index has one index entry for each search key value. It means that records can be in any order in the database.

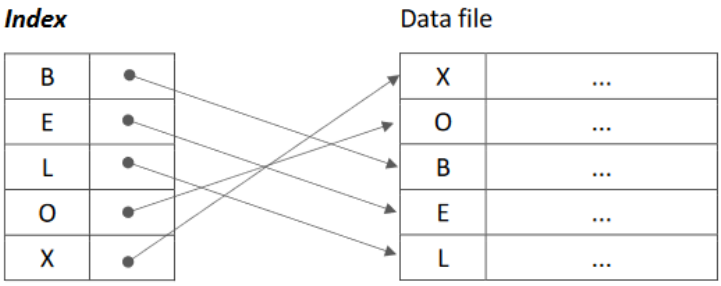


Figure 3. Dense index (Silpiö, K. 2020)

2. **Sparse index**: This type of index has one index entry for each block. The records must be clustered (usually in ascending order) according to the search key. The index entry in the sparse index points to the lowest search key in each block in the database.

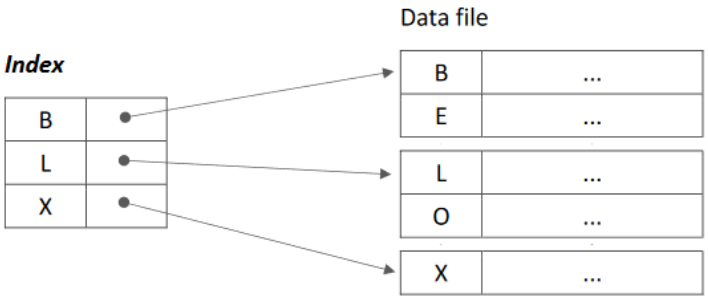


Figure 4. Sparse index (Silpiö, K. 2020)

2.4.2 Non-clustered and clustered indexes

Multi-level indexes have multiple levels of indexes. It means that the root index level point to another index level, and so on. There can be two or more levels of indexes. Multi-level indexes can be separated into two categories, non-clustered and clustered (Silpiö 2020):

1. **Non-clustered multi-level index**: This type of index has multiple levels of indexes. The root level point to the leaf level, which is the last index level. The root level is a sparse index, and the last index level is a dense index that points to the actual data files (in the database). If there are levels in-between, these levels will be sparse. There can be several non-clustered multi-level indexes in a table because the leaf level is not the actual data.

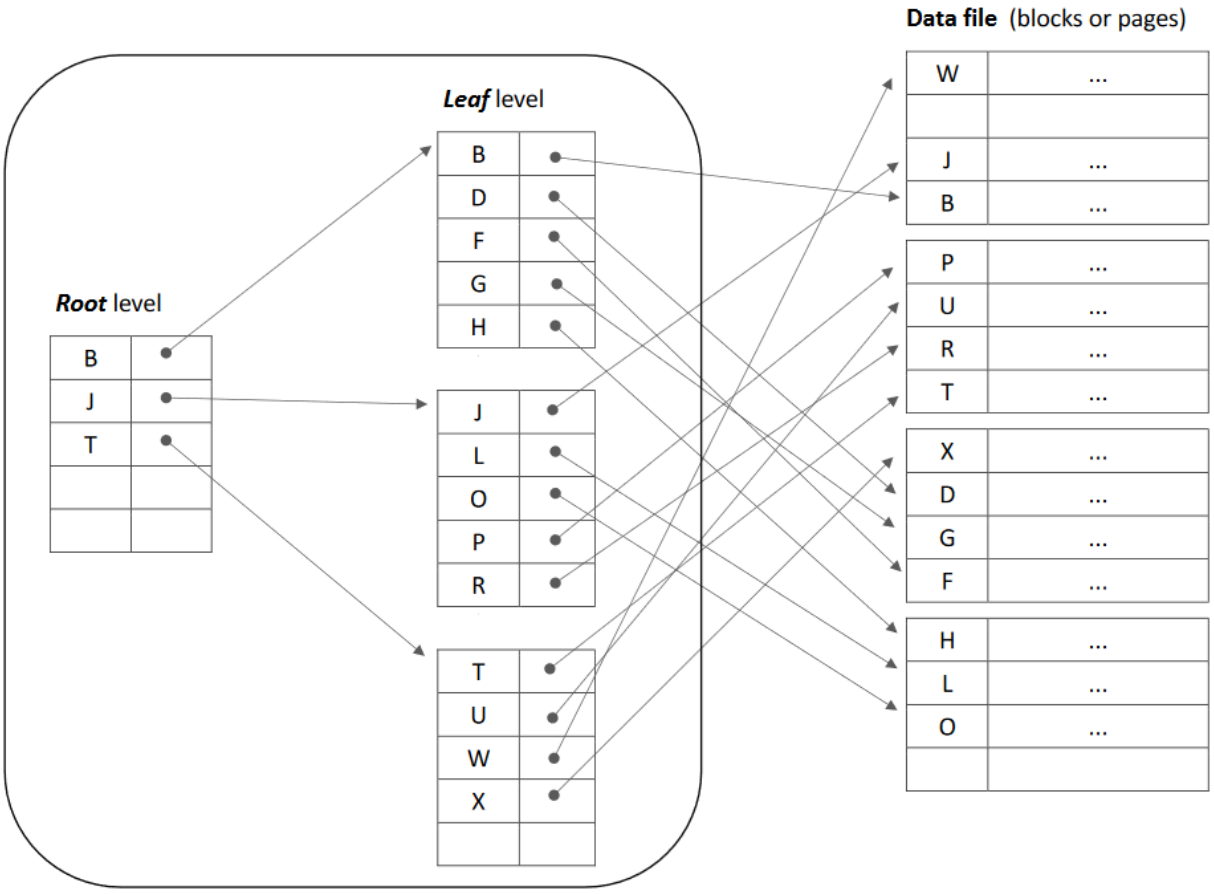


Figure 5. Non-clustered multi-level index (Silpiö, K. 2020)

2. **Clustered multi-level index**: This type of index also has multiple levels of indexes. The root level is also a sparse index that points to other levels of sparse indexes. But the main difference between clustered and non-clustered indexes is that the leaf level of the index is the actual data. It means that the actual data must be sorted. Because of this, there can be only one clustered multi-level index in a table (usually the primary key).

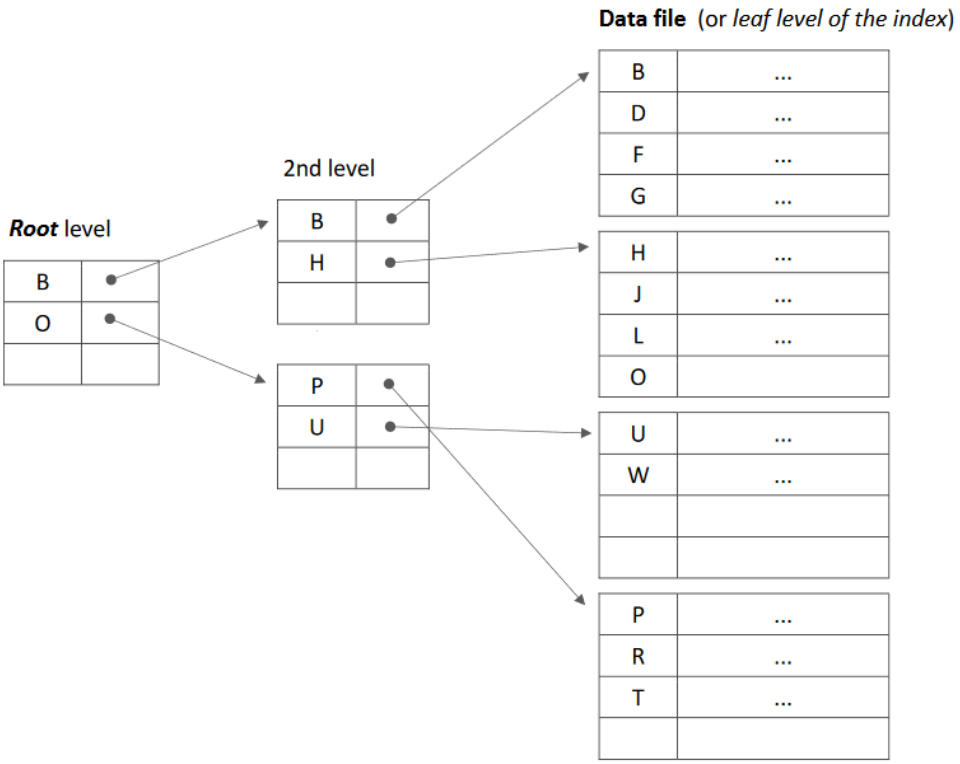


Figure 6. Clustered multi-level index (Silpiö, K.)

2.4.3 Heap tables

A heap table, in SQL Server, is a table without any clustered index. In a heap table, there is only data pages, and they are not saved in particular order.

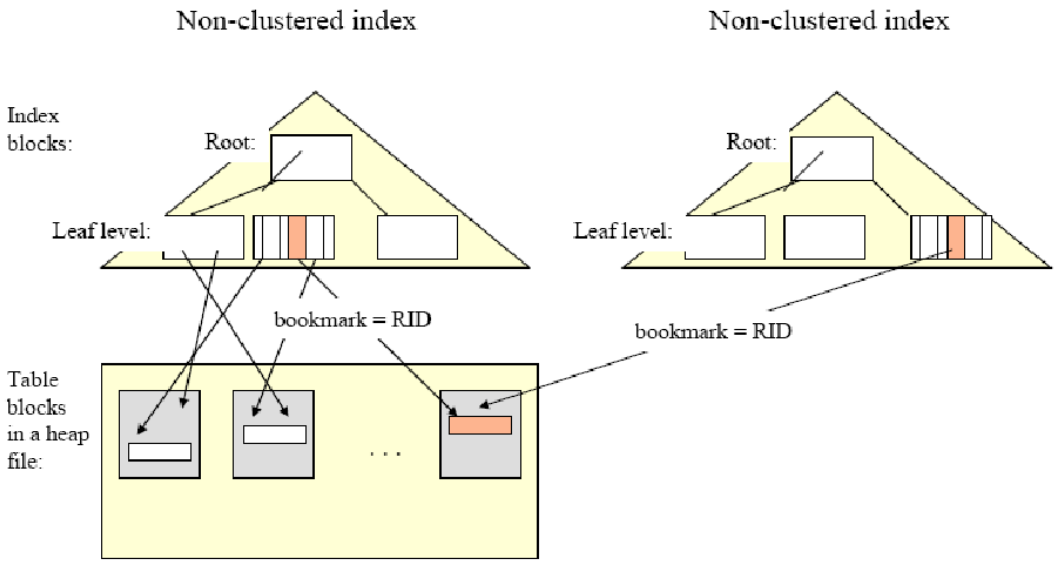


Figure 7. Heap table (Laiho, M. 2016)

On the picture above, we can see that there can be multiple non-clustered indexes in a table. This image also illustrates the possibility of multiple indexes to point to the same row. Here, the data is not sorted.

2.4.4 Table with a unique clustered index

A clustered table in SQL Server is a table with a clustered index. Even though we mention clustered index, this is not only an index, it contains the actual data. Here, unique means that there cannot be two same data with the same identifier. By default, SQL Server creates a unique clustered index on the primary key. In a clustered table, the data are saved in ascending order (Silpiö & Laiho 2016).

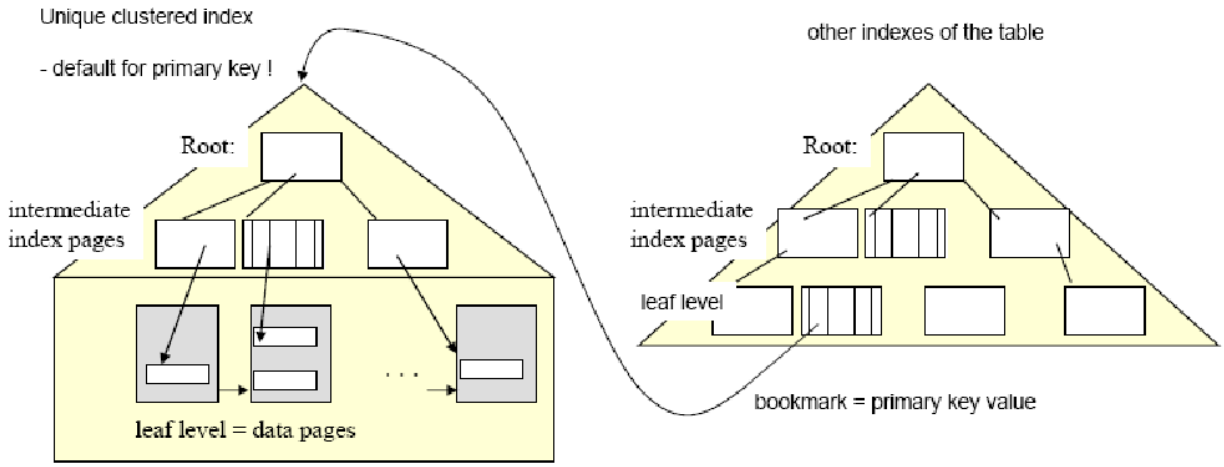


Figure 8. Unique clustered table (Laiho, M. 2016)

On this picture, we can notice that when there is a clustered index, the other non-clustered indexes will use it. It means that the leaf level of the non-clustered index will go to the root level of the clustered index and use it to find the right data.

2.4.5 Table with a non-unique clustered index

A non-unique clustered table in SQL Server is also a table with a clustered index, but here, non-unique means that the clustered index is non-unique. Because of this, SQL Server has to add a uniqueifier column to the table. This will make each row identifiable by both the clustered key and the uniqueifier.

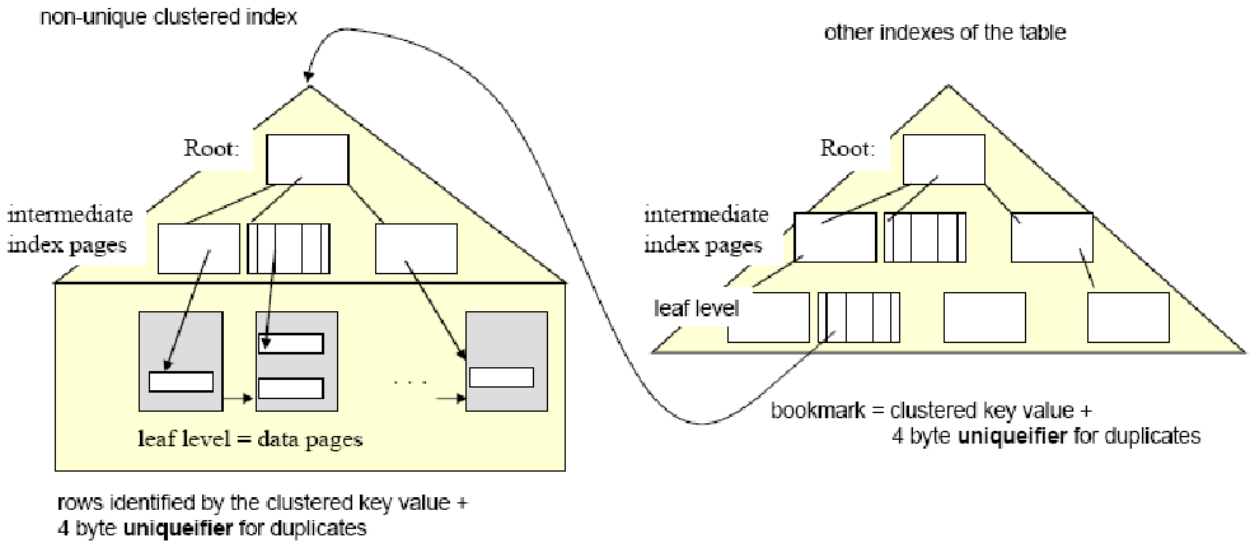


Figure 9. Non-unique clustered table (Laiho, M. 2016)

The main difference in this picture is that the bookmark in non-unique clustered table is the clustered key value and the 4 bytes uniqueifier. To illustrate, we can suppose a table with two indexes, a **non-unique clustered** index on the “lastName” column and a **unique non-clustered** index on the “empno” column (which is the unique number identifier of employees). The index entry in at the leaf level of the non-unique clustered index includes “empno” (which is the key) and the bookmark (which includes “lastName” and the uniqueifier).

For example, the employee whose “empno” is 36 can be found as follows (Silpiö & Laiho 2016):

1. First, by finding the bookmark for “empno” 36 in the non-clustered index. The bookmark will be ‘Smith’ + the uniqueifier value of 1.
2. Then, by finding the data row in the unique clustered index by using the bookmark (‘Smith’ and the uniqueifier 1).

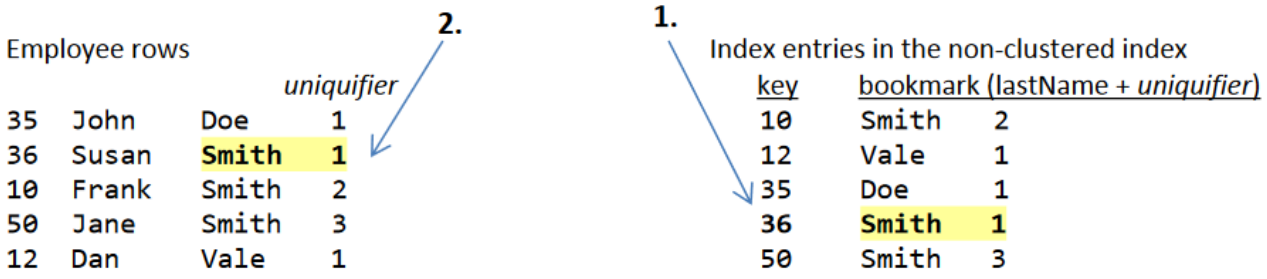


Figure 10. Example of a search (Laiho, M. 2016)

2.4.6 Candidate columns for non-clustered index

The best candidate columns for non-clustered index are the ones that are frequently grouped with the GROUP BY clause, or the ones involved in JOIN and WHERE conditions, because it will return the exact match values rather than returning a large set of data (Yaseen 7 May 2018). Columns that have large numbers of different values are also good candidates, because it will allow the database to quickly locate the rows that match without having to scan the entire table, even if the rows are not sorted. In contrast, columns that have few numbers of different values such as gender columns are **not** great candidates, because it will be more efficient to scan the entire table directly, especially if it is sorted.

2.4.7 Candidate columns for clustered index

The primary key is a good candidate for clustered index, which is why SQL Server creates a unique clustered index on the primary key column by default, unless we already have created a clustered index on the same table (Yaseen 3 May 2018). Among the other candidates that can be mentioned are the columns that are frequently ordered or grouped, but because a table can only have one clustered index, choosing the right column is very important.

2.5 Indexes for unstructured and semi-structured data

There are also indexing techniques for unstructured and semi-structured data. As I will only use the MongoDB database for my various analyses, I will therefore quote the main methods used by this database management system.

2.5.1 B-Tree Indexes

Although B-tree indexes are indexing methods supported by relational databases such as SQL Server and Oracle (Lightstone, Teorey & Nadeau 2010, chapter 2.1), I decided to mention this method and compare it only when implementing the NoSQL database. Indeed, for the purposes of this thesis, I thought it more relevant to compare other indexes for the relational database part.

The B-tree index, also called “balanced tree” is MongoDB’s default index. This index, resembling a tree, will use pointers to go further and further down and finally find the appropriate branch (Harrison 2021, chapter 5).

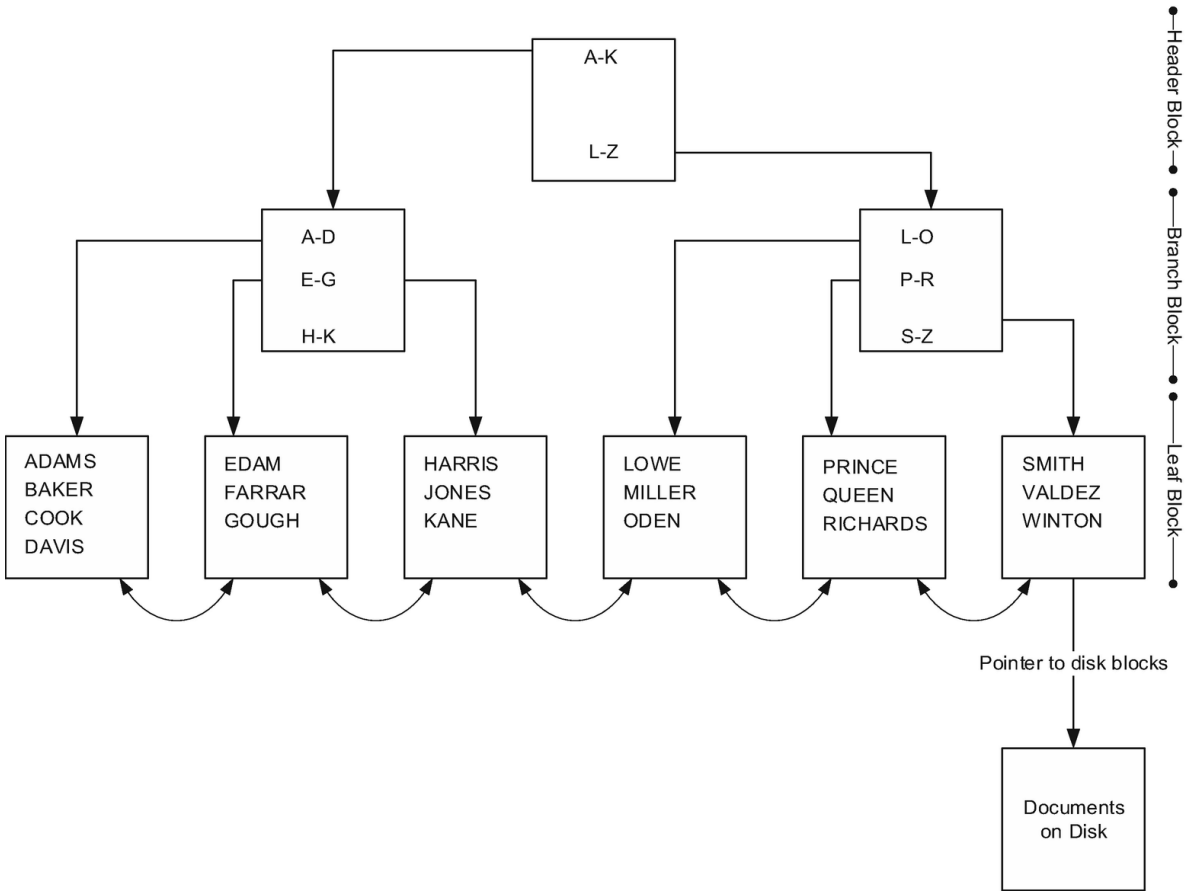


Figure 11. B-tree index structure (Harrison, G. 2021)

We can see on the image above, that the tree has a root, named Header Block. From this root, with the help of pointers, we will be able to go down to the different branches of the index, called Branch Block. From there, we can reach the leaves, named Leaf Block, which contain a list of the actual values. Then, we can find the desired value and its associated disk location. Leaf blocks are also connected to each other, each leaf is connected to the previous one, but also to the next one.

There are several advantages to B-tree indexes (Harrison 2021, chapter 5):

- Leaf nodes are in the same depth, which means that performance is quite predictable. Do data will be mor than three or four blocks away.
- Because B-tree have at most four blocks (a header, two branches and a leaf block), the performance is excellent because it will not take long to access the data. More than that, because usually the database keeps the header and branch blocks in memory, there will normally be no more than two accesses to the disk.
- Because of the links between leaf blocks, B-tree indexes support lookups and range queries.

There is also a big disadvantage to B-tree indexes: the maintenance of this type of index can be very expensive. This happens when there is no more space in a block where we want to add a new entry. In this case, there will be an index split and a new block will be allocated and half of the entries in the existing block will be transferred to the new one. A new entry will be added in the old block that will point to the new one (Harrison 2021, chapter 5).

2.5.2 Candidate columns for B-tree index

In order to understand and analyse which are the good candidates for this index, it is necessary to introduce the index selectivity and unique indexes. The selectivity is a measure of the number of documents associated with an index key value. This means that an index is selective when it has a large number of unique values and only a few duplicate ones. For example, lastName column is very selective, because there are a lot of different family names, without too many duplicates. Unlike family names, a gender column that only have a few different values and a lot of duplicates is called non-selective (Harrison 2021, chapter 5).

Unique indexes will prevent any duplicate values. Therefore, if you try to create a unique index on data that have duplicates, there will be an error. Because unique indexes do not have any duplicates, they are very efficient. By default, MongoDB create a unique index on the id attribute (Harrison 2021, chapter 5).

A good candidate for a B-tree index will be a selective column. By default, MongoDB will try to use the most selective index it finds.

2.5.3 Compound Indexes

As above, there are also composite indexes in relational databases. However, they are not included in the relational database part of this thesis, even though they are useful and widely used indexes. A single index can be defined on more than one table column. It means that the index key can be composed of more than only one value. Composite indexes in relational databases can be useful to efficiently search a database table when we have a complex SQL search criterion (for example having a lot of “AND” in a “WHERE” condition).

Unlike single field indexes, a compound index, in a NoSQL database, is an index which includes two or more attributes. The main advantage is selectivity. Indeed, as there will be several attributes affected, this index will be more selective than a single field index, such as a B-tree index. Simply combining attributes will point to a smaller amount of data. In general, an improvement in performance can be noticed when adding an attribute to a compound index.

To use these compound indexes properly, it is necessary to understand the compound key order. In the case where we want to create an index on 3 attributes (let's take for example in order, the *lastName*, the *firstName* and the *dateOfBirth* columns), it is necessary to understand that this index can optimise the following queries:

- Queries on *lastName*, *firstName* and *dateOfBirth*.
- Queries on *lastName* and *firstName*.
- Queries on *lastName* and *dateOfBirth*.

However, it will not be effective with queries on *firstName* alone or on *dateOfBirth* alone, **because of the order** (Harrison 2021, chapter 5).

2.5.4 Candidate columns for compound index

If you often search for certain data based on several attributes, it is very advantageous to create a compound index on those attributes. Let's take the example of a company's customers. If the search for customers is based on their first and last names, it is preferable to create a compound index on the first and last name attributes (Harrison 2021, chapter 5).

2.5.5 Partial and Sparse Indexes

Performance is best when all data is kept in memory. However, in many cases it is necessary to manipulate a very large amount of data, so it is not possible to keep all the data in memory. Moreover, with this huge amount of data, MongoDB will not be able to keep all the indexes in memory. In this scenario, it might be useful to create partial or sparse indexes.
Partial indexes only index a subset of a document. This subset has to meet a specific filter expression. Because of this, they have lower storage requirements and performance costs are reduced (MongoDB).
Sparse indexes only contain entries for documents that have the indexed field. It means that those indexes do not include all documents of a collection, that is why they are called sparse (MongoDB).

Although those two indexes are interesting to analyse, they will not be included in this thesis. Indeed, to enable an accurate comparison between structured and unstructured databases, the data in the two different databases are the same. This means that the documents in the different MongoDB collections will all have the same fields, so it makes no sense to test the sparse index in that case. In addition, as far as the data is concerned, it would not make sense to create a partial index either. In fact, this index is rather useful when the data is unstructured, and therefore sometimes requires indexing only a small part of the documents in the collection, which is not the case here.

2.6 Conclusion

To conclude, it is necessary to know whether the data we are going to process is structured, semi-structured or unstructured. This will enable us to know which database will best satisfy us. Once the database has been chosen, it is necessary to analyse and test our data to know which index will be the most beneficial, as having good indexes will allow for much better performance and more efficient searches.

As far as indexes are concerned, there are several that allow data to be indexed in an efficient manner. Moreover, it is also possible to combine and customise the different indexes depending on the data we have.

It is important to conclude by mentioning that it is indeed possible and recommended to index structured, semi-structured and unstructured data (Nesavich & Inmon 2007, chapter 7). In addition, it is important to be aware of the data present in our database and it is also necessary to take many factors into account when indexing, such as the different queries that will be executed. It is also crucial to be aware of the different users of the database, so that we can create the most appropriate levels of isolation to maintain adequate database performance.

3 Empirical part

To illustrate the theoretical part, several tests will be performed using two different databases: Microsoft SQL Server for structured data and MongoDB for semi-structured and unstructured data.

3.1 Technologies used

For my relational database implementation, I will use Microsoft SQL Server Management Studio (SSMS). It is an integrated environment for managing any SQL infrastructure. Microsoft SQL Server Management Studio will give me the tools to create different indexes and analyse them (Microsoft 2023). For my NoSQL database, I will use MongoDB. MongoDB is a document-oriented database management system that can be distributed to any number of computers and does not require a predefined data schema. To implement this database and its environment, I will use Docker. Through Docker, I will be able to create different containers by using an image containing the MongoDB database.

3.2 Implementation of the Microsoft SQL Server database

Regarding the implementation of the Microsoft SQL Server database, I started by installing the latest version of Microsoft SQL Server Management Studio, which is the version 19.0.2. This version was released on March 13, 2023. To begin with, one of the special features of Microsoft SQL Server Management Studio is that you can create several different databases, using the command:

```
CREATE DATABASE IndexTestDatabase;
```

Then, when the database is created, you can access its properties:

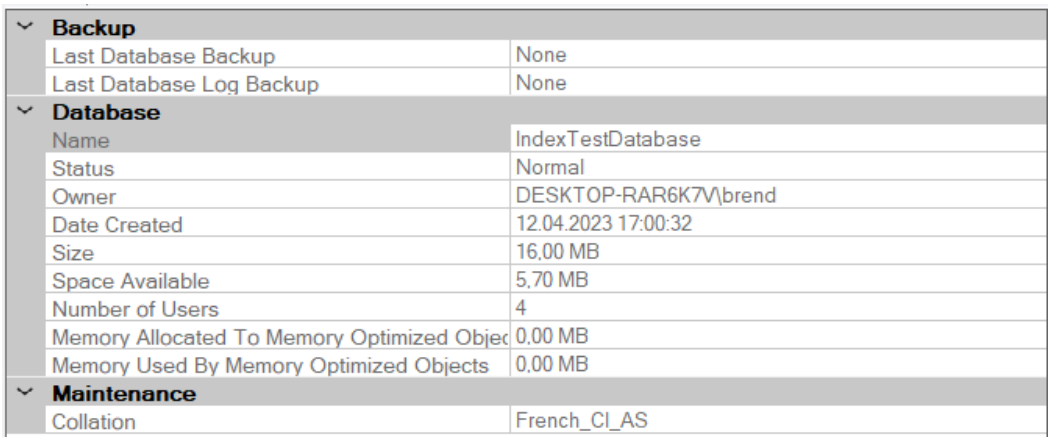


Figure 12. Database Properties

Here, we can see some information, such as the database's status, owner, size, space available and so on. Once the database is created, we can create different tables, populate them, and create different indexes for our tests.

3.2.1 Creation of the tables

For the tests on my relational database, I used DbSchema to create a schema. DbSchema is a software which allows to create schemas, depending on the database. For my various schemas, I used the free version which offered me the necessary tools. For my tests, I chose to create a scheme representing orders that customers can have.

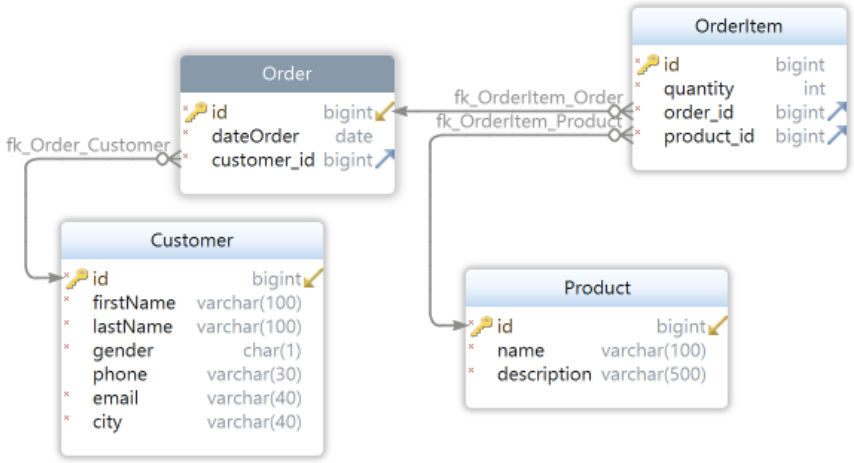


Figure 13. Customer schema

From this diagram, we can see that *Customer* has zero or more orders. The *Order*, in turn, has an *OrderItem* which in turn have one and only one associated *Product*.

To have a better understanding of the above diagram, here are the legends:

- **Table**: This is the table legend.

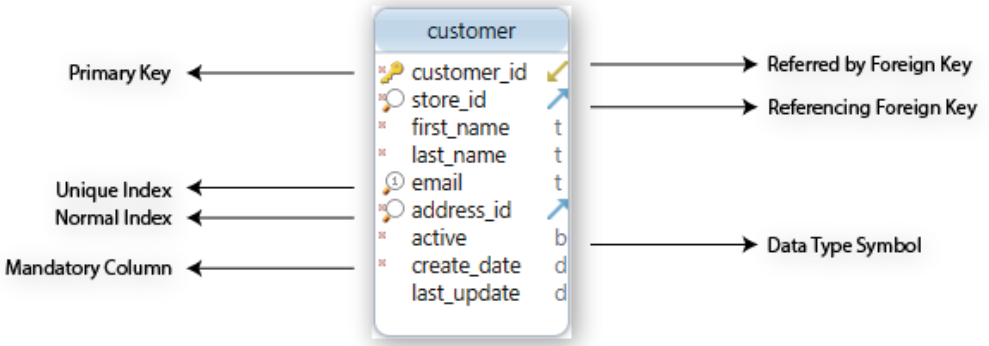


Figure 14. Table layout (DbSchema)

- **One to zero or many**: In the child table, the referencing column is not mandatory, nor unique. In this case, the parent table (1) can refer to zero or many entries in the child table (2).

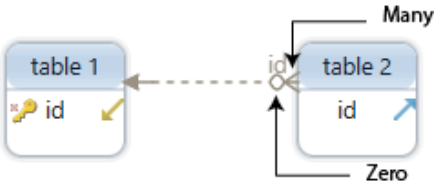


Figure 15. One to zero or many (DbSchema)

- **The full arrow**: The solid arrow means that the minimum cardinality is one. This means that, for example, an *Order* must have one and only one *Customer*.

3.2.2 Creating tables in SQL

The second step is to create the tables in our database using SQL code. DbSchema offers the possibility to generate the SQL code for each table. It is therefore a matter of creating the tables in the right order according to the relations and keys.

1. ***Customer* table**: Here is the code for the *Customer* table:

```
CREATE TABLE Customer (
    id                  bigint    IDENTITY  NOT NULL,
    firstName           varchar(100)      NOT NULL,
    lastName            varchar(100)      NOT NULL,
    gender              char(1)       NOT NULL,
    phone               varchar(30)       NULL,
    email               varchar(40)       NOT NULL,
    city                varchar(40)       NOT NULL,
    CONSTRAINT pk_Customer PRIMARY KEY  ( id ) ,
    CONSTRAINT CHECK_Customer_gender CHECK ( gender in ('M','F')
));
```

2. ***Order* table**: Here is the code for the *Order* table:

```
CREATE TABLE [Order] (
    id                  bigint    IDENTITY  NOT NULL,
    dateOrder           date  DEFAULT getdate()    NOT NULL,
    customer_id         bigint      NOT NULL,
    CONSTRAINT pk_Tbl PRIMARY KEY  ( id )
);

ALTER TABLE [Order]
    ADD CONSTRAINT fk_Order_Customer FOREIGN KEY ( customer_id )
        REFERENCES Customer( id )
        ON DELETE CASCADE
        ON UPDATE CASCADE;
```

3. ***Product* table**: Here is the code for the *Product* table:

```
CREATE TABLE Product (
    id                  bigint    IDENTITY  NOT NULL,
    name                varchar(100)      NOT NULL,
    description         varchar(500)      NOT NULL,
    CONSTRAINT pk_Product PRIMARY KEY  ( id )
);
```

4. ***OrderItem* table**: Here is the code for the *OrderItem* table:

```
CREATE TABLE OrderItem (
    id                  bigint    IDENTITY  NOT NULL,
    quantity            int  DEFAULT 1    NOT NULL,
    order_id            bigint      NOT NULL,
    product_id          bigint      NOT NULL,
    CONSTRAINT pk_OrderItem PRIMARY KEY  ( id )
);

ALTER TABLE OrderItem ADD CONSTRAINT fk_OrderItem_Order
    FOREIGN KEY ( order_id )
    REFERENCES [Order]( id )
    ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE OrderItem ADD CONSTRAINT fk_OrderItem_Product
    FOREIGN KEY ( product_id )
    REFERENCES Product( id )
    ON DELETE CASCADE ON UPDATE CASCADE;
```

3.2.3 Populate the database

There is a tool in DbSchema called Data Generator which fills database tables with random data. This random generator is using patterns for setting how the generated data should look like. For the *Customer* and *Product* tables, it was easy to use this generator, I just had to change some patterns. Then, DbSchema just generate INSERT queries. I chose to generate 10'000 products and 100'000 customers.

After generating the products and customers, because of the Primary Keys were generated by the database, I had to modify the pattern directly in the DbSchema generator. To do that, I simply ordered the result of my queries (`SELECT id FROM customer ORDER BY id ASC;`) and (`SELECT id FROM product ORDER BY id ASC;`) to find out the minimum and maximum *id* of the *Customer* and *Product* table. Then, I simply had to create a pattern to randomly generate the id ranges. I chose to generate 60'000 orders and 80'000 orderItems.

3.2.4 Observe performance – Settings

Now that our database is populated, we can start to measure performance. To do this, it is necessary to activate statistics on elapsed time by executing `SET STATISTICS IO, TIME ON` on each query. This will allow the display of various performance measures that will be useful to us.

Next, we will check which indexes SQL Server created when creating our different tables. This is simply a matter of running the command: `sp_helpindex table_name;` After running this

command, you can see that SQL Server automatically creates a clustered, unique index on the Primary Key of every tables. Here is an example of the *Customer* table:

	index_name	index_description	index_keys
1	pk_Customer	clustered, unique, primary key located on PRIMARY	id

Figure 16. Index on Customer Primary Key

The same applies to the other three tables. Now, as a first step, I will test by creating non-clustered indexes (because there can be only one clustered index on a table) on the different columns (excluding the Primary Key) while keeping the clustered index on the primary key. Then, as a second step, I will test by removing the clustered index on the Primary Key and create clustered indexes on some columns.

3.2.5 Buffer Management and Cache

Before starting the tests, it is necessary to talk about the Buffer and how it affects performance. Most databases are built using a two-level memory hierarchy: slower persistent storage (disk) and faster main memory (RAM). As disk access takes longer than RAM access, the pages (elements that contain data) are *cached* in RAM, which will consequently reduce disk access. This means that when the page is requested by the database, it is the one in the cache that is returned. For a page in the cache to be returned, there must have been no change to the data it contains on disk (Petrov 2019, Chapter 5).

Regarding the Page Buffer, the page remains in it until the DBMS needs to remove it, for example due to lack of space when accessing other data. Furthermore, with this system, changes are not made directly in the disk, but in the copy in the Page Buffer, this is called *Lazy Writing*. There may be multiple updates and changes within the Buffer Page. The main purpose of this *Lazy Writing* is to reduce the input and output on the disk.

However, it is mandatory, at some point, to write these dirty pages (pages in the Buffer) to disk. Furthermore, if the database accidentally shuts down, everything in the Buffer is lost. This is why the database will do it when:

- When the Buffer has pages that have not been used for a while.
- During a **checkpoint** that the database has set up.
- During a **checkpoint** that a user ran or set up.

In order to have the most accurate view of the efficiency of the different indexes used in the tests, between each request, a cache cleaning will be carried out. To clean the cache, there is the following command:

```
DBCC DROPCLEANBUFFERS;
```

3.2.6 How to create indexes

To get started, here is the syntax for creating indexes in Microsoft SQL Server:

```
CREATE [UNIQUE][CLUSTERED|NONCLUSTERED]
    INDEX index_name
    ON table_Name ( column_name [ { , column_name } ... ] )
    [WITH FILLFACTOR = fillfactor [ , PADINDEX = ON ]]
```

Here are some rules for creating indexes and understanding the above syntax (Kari Silpiö 2018):

- What is in brackets is facultative.
- What is underlined is the default value.
- Indexes are by default non-unique.
- There can be many non-clustered indexes on the table.
- There can be only one clustered index on the table.
- Primary Key constraint: by default, SQL Server creates a clustered, unique index.
- Unique constraint: by default, SQL Server creates a non-clustered unique index.
- Typically, we create a non-unique, non-clustered index on a foreign key.
- For special purposes, we can also create a filtered index using the WHERE clause.

Then, there is the fill factor. This *fillfactor* (0 - 100) specifies a percentage that indicates how full the leaf level of each index page should be made during index creation or rebuild. The default value (completely full) can be changed on the DBMS instance level. For the different tests, we will use the default *fillfactor* value and only compare which are the best indexes according to the columns of the table. For the different tests, I will first execute and analyse the query 5 times without any index on the column, and then 5 times after creating the index on the desired column.

3.2.7 Creating a non-clustered index on the *firstName* column (*Customer* table)

I chose the *firstName* column because it is a non-unique column, but with many different data. Furthermore, it is a column that is often used and can also be sorted or manipulated in different cases.

For this analysis, here are the different commands I will execute 5 times each:

1. `SELECT firstname, lastname FROM Customer ORDER BY firstname ASC;`
2. `SELECT firstname, lastname FROM Customer WHERE firstname = 'Jeremy'; -- This name does exist`
3. `SELECT firstname, lastname FROM Customer WHERE firstname = 'Svetlana'; -- This name does not exist`
4. `SELECT firstname, lastname FROM Customer WHERE firstname LIKE '%es%'; -- This does exist`
5. `SELECT firstname, lastname FROM Customer WHERE firstname LIKE '%xp%'; -- This does not exist`

Then, after testing all the queries without any index, I created the index using this command:

```
CREATE NONCLUSTERED INDEX ix_firstname ON Customer (firstname);
```

Here we can see that I have two indexes now:

	index_name	index_description	index_keys
1	ix_firstname	nonclustered located on PRIMARY	firstName
2	pk_Customer	clustered, unique, primary key located on PRIMARY	id

Figure 17. Two Indexes on Customer table

Results:

1. Here are the 5 different outputs and the average for the first query, CPU time is the quantity of processor time taken by the process and **elapsed time** is the total duration of the task.
   a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
CPU time (ms)	218	188	327	171	235	227,8
Elapsed time (ms)	506	474	481	464	477	480,4

   b. **With an index**:

With an index	1	2	3	4	5	**Average**
CPU time (ms)	251	171	173	220	203	203,6
Elapsed time (ms)	448	494	488	538	500	493,6

2. This query displays all the ‘Jeremy’ from the database (There are some ‘Jeremys’ in the database).
    a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
CPU time (ms)	15	0	0	0	16	6,2
Elapsed time (ms)	17	16	18	19	17	17,4

    b. **With an index**:

With an index	1	2	3	4	5	**Average**
CPU time (ms)	0	0	0	0	0	0
Elapsed time (ms)	2	2	3	3	3	2,6

3. This query displays the ‘Svetlana’ (There is no ‘Svetlana’ in the database).
    a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
CPU time (ms)	16	16	0	0	0	6,4
Elapsed time (ms)	17	19	16	18	17	17,4

    b. **With an index**:

With an index	1	2	3	4	5	**Average**
CPU time (ms)	0	0	0	0	0	0
Elapsed time (ms)	0	0	0	0	0	0

4. This query displays names that contains the letters ‘es’ (There are some in the database).
    a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
CPU time (ms)	47	63	62	47	31	50
Elapsed time (ms)	104	115	102	102	105	105,6

    b. **With an index**:

With an index	1	2	3	4	5	**Average**
CPU time (ms)	31	63	47	31	31	40,6
Elapsed time (ms)	110	101	104	117	101	106,6

5. This query displays names that contains the letters ‘xp’ (There is no such names in the database).
    a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
CPU time (ms)	47	31	15	63	47	40,6
Elapsed time (ms)	60	59	59	57	62	59,4

    b. **With an index**:

With an index	1	2	3	4	5	**Average**
CPU time (ms)	47	63	47	47	32	47,2
Elapsed time (ms)	55	62	56	58	56	57,4

Graphs and conclusion:

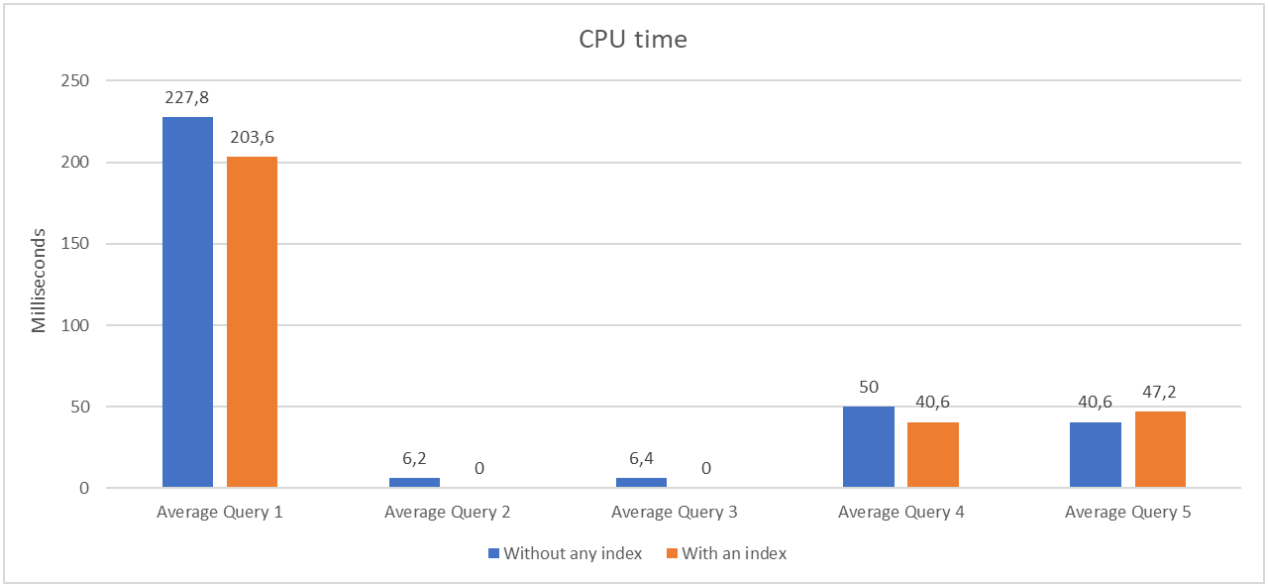


Figure 18. CPU time comparisons

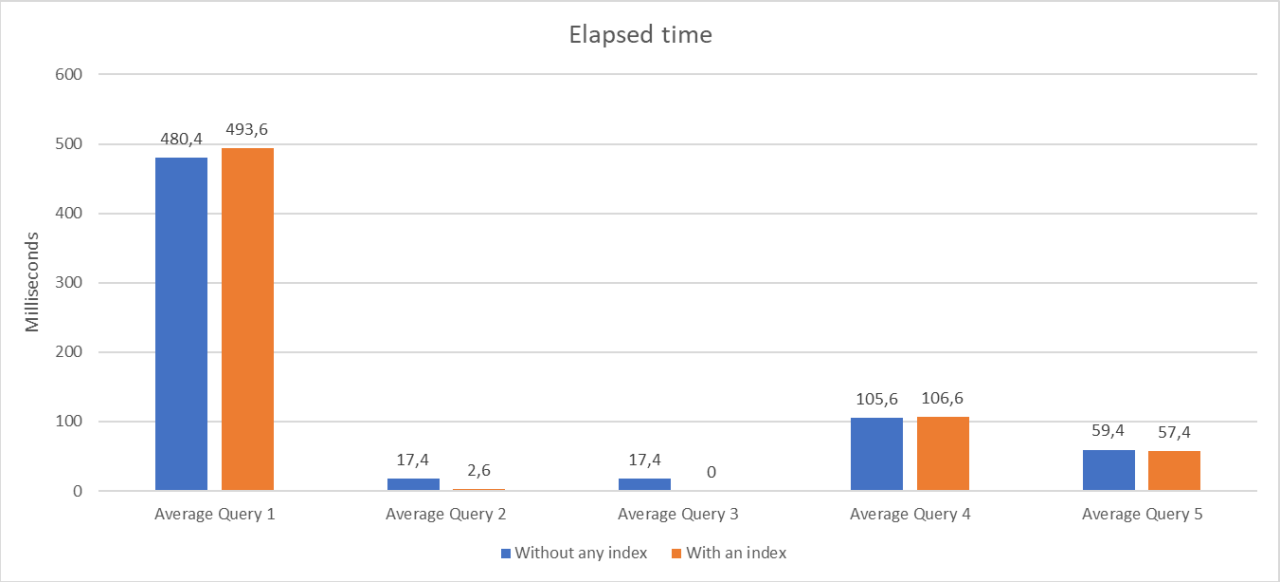


Figure 19. Elapsed time comparisons

1. In this first query, we can see that the average elapsed time is not lower even after creating the non-clustered index on the column. This can be explained by the fact that since we are making a query that asks for all the first names in the database, we will have to scan the entire database anyway.
2. For this second query, we can see that the index has improved the query enormously. Indeed, the average time went from 17,4 milliseconds to 2,6 milliseconds. This can be explained by the fact that this is a query where we are looking for a particular first name that exists in the database. The database, thanks to the index, will be able to find it much more quickly.
3. This is the same process as the request above. As the first name searched for does not exist, thanks to the index, the database will not need to search any longer.
4. For this fourth query, the index will not be of much use. Indeed, the index is useful to search for specific first names (a bit like a dictionary), but not to search for first names containing specific letters. The database will therefore have to scan the entire table anyway.
5. For this last request, as for the one above, the index will not be useful. However, one can see that there is a clear difference in time between query number 4 and 5 (the fifth query takes about half the elapsed time). This could be explained by the fact that the letters chosen are not very common.

3.2.8 Creating a non-clustered index on the *gender* column (*Customer* table)

I chose this column because it has only two different values, which means that it has many duplicates.

For this analysis, here are the different commands I will execute 5 times each:

1. `SELECT firstname, lastname, gender FROM Customer ORDER BY gender ASC;`
2. `SELECT firstname, lastname, gender FROM Customer WHERE gender = 'F';`

Then, after testing without any index, I created this index by using this command:

```
CREATE NONCLUSTERED INDEX ix_gender ON Customer (gender);
```

Results:

1. This query will display all the customers ordered by gender (it means that it will display first all the female customers and then all the male customers).
   a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
CPU time (ms)	233	109	141	249	218	190
Elapsed time (ms)	706	588	561	552	546	590,6

   b. **Without any index**:

With an index	1	2	3	4	5	**Average**
CPU time (ms)	108	112	93	111	157	116,2
Elapsed time (ms)	541	537	543	584	573	555,6

2. This query will display all the female customers.
   a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
CPU time (ms)	0	0	0	0	0	0
Elapsed time (ms)	284	282	288	287	297	287,6

   b. **With an index**:

With an index	1	2	3	4	5	**Average**
CPU time (ms)	0	0	11	31	0	8,4
Elapsed time (ms)	272	284	288	301	350	299

Graphs and conclusion:

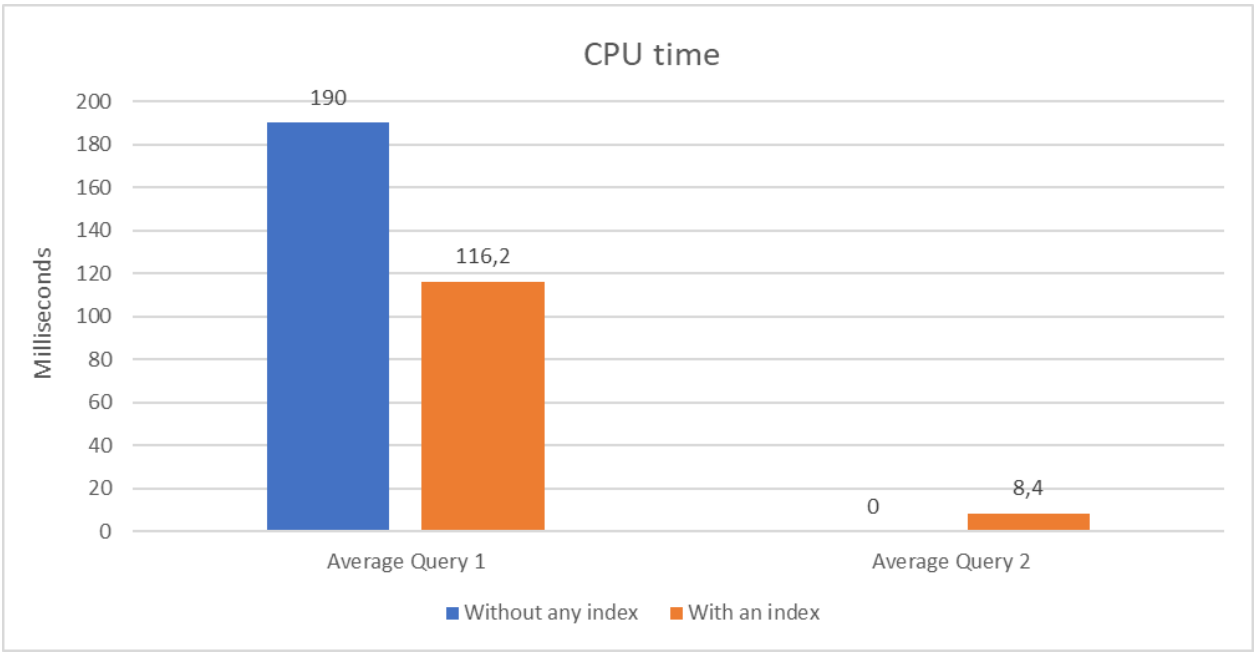


Figure 20. CPU time comparisons

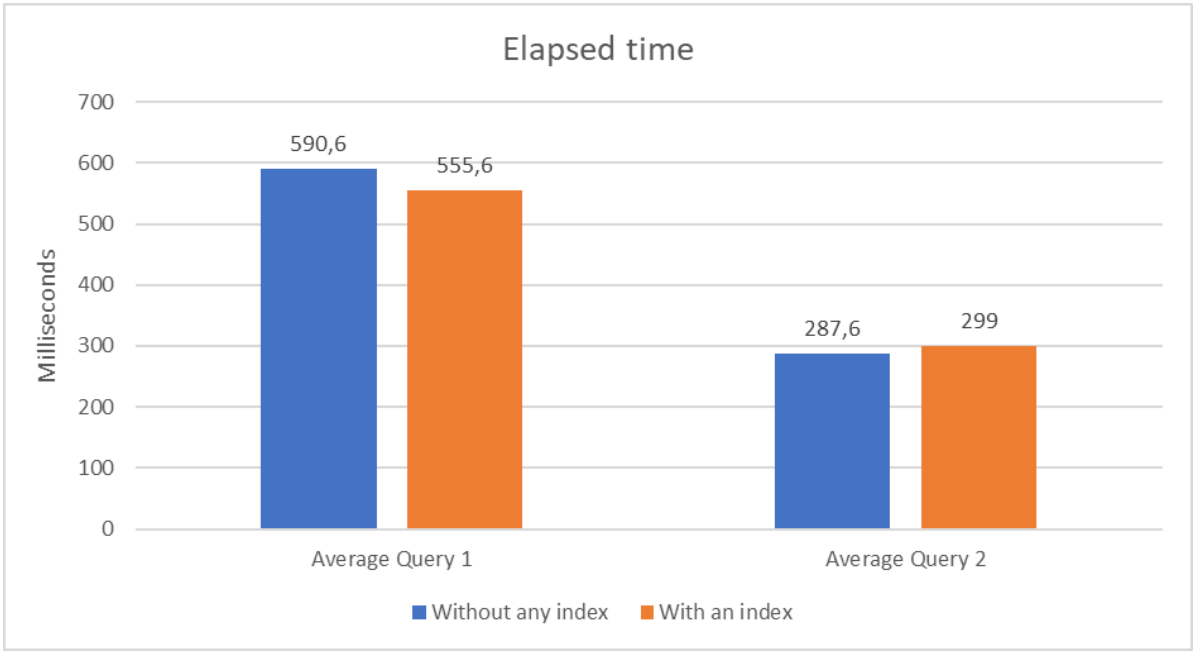


Figure 21. Elapsed time comparisons

1. In this query, we can again observe that when we query the whole table and there will be a scan table, the index will not be that useful.
2. Here we can see that the non-clustered index does not improve the query at all. This is because it is a column containing only two values, therefore the non-clustered index will not be of any use.

3.2.9 Creating a non-clustered index on the *customer_id* column (*Order* table)

I chose this column because it is a foreign key. In many cases, it is advisable to create an index on foreign keys when they are often manipulated or used in queries.

For this analysis, this is the command I will execute 5 times:

1.
```
SELECT o.id, o.dateorder, c.firstname, c.lastname FROM [Order] o
    LEFT OUTER JOIN Customer c ON c.id = o.customer_id
    ORDER BY c.id;
```

Then, after testing without any index, I created the index using this command:

```
CREATE NONCLUSTERED INDEX ix_customer_id ON [Order] (customer_id);
```

Results:

1. This query will display all the orders and their related customer.
    a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
CPU time (ms)	78	171	93	93	77	102,4
Elapsed time (ms)	407	407	420	425	478	427,4

    b. **With an index**:

With an index	1	2	3	4	5	**Average**
CPU time (ms)	127	138	80	94	124	112,6
Elapsed time (ms)	401	456	388	410	391	409,2

Graphs and conclusion:

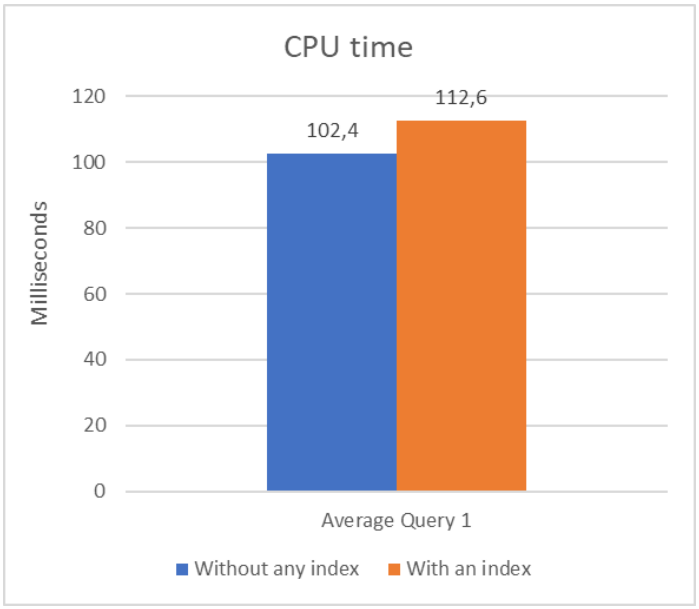


Figure 22. CPU time comparisons

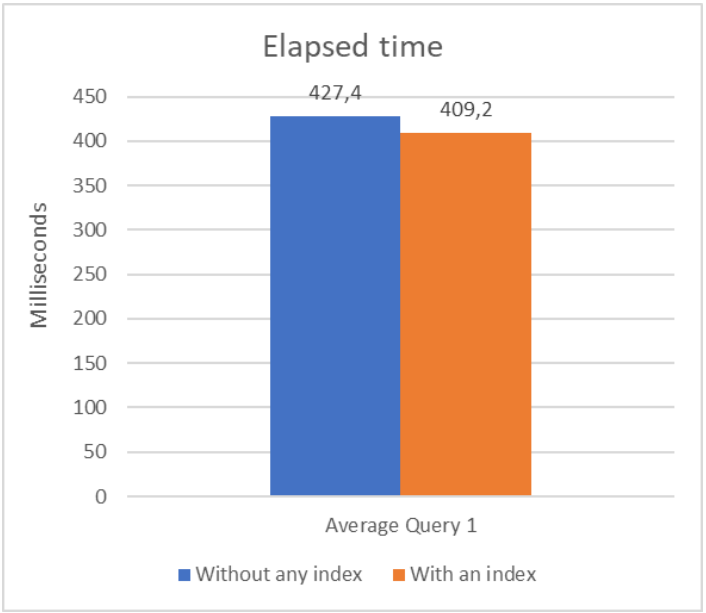


Figure 23. Elapsed time comparisons

1. Here we can notice a very slight decrease in elapsed time, however it is not obvious. To have better test results and to know if an index can be good on this foreign key, it could be beneficial to test on an even larger amount of data, especially on a larger number of orders.

3.2.10 Creating a clustered index on the *gender* column (*Customer* table)

Now it is time to test a clustered index for the *gender* column. In the case of the *gender* column, it is much more beneficial to create a clustered index rather than a non-clustered index.

For the tests on this column without index, please refer to the beginning of the chapter 3.2.7 (Results without any index). For this new test, I will also execute the two queries also located in this same chapter.

To begin, I will drop the non-clustered index on the *gender* column and the Primary Key constraint. Before doing so, I will drop the foreign key in the *Order* table, and then re-create it once the manipulations are complete:

```
DROP INDEX ix_gender ON Customer;
ALTER TABLE [Order] DROP CONSTRAINT fk_Order_Customer;
```

Then, I will remove the primary key constraint on the Customer table. It will delete the clustered index at the same time (to be able to create the clustered index on the *gender* column):

```
ALTER TABLE Customer DROP CONSTRAINT pk_Customer;
```

Then, I will create a new Primary Key with a non-clustered index this time:

```
ALTER TABLE Customer ADD CONSTRAINT ix_nonclustered_pk
    PRIMARY KEY NONCLUSTERED (id);
```

Then, it will simply be a matter of recreating a foreign key on the *Order* table that will point to the *id* column of the *Customer* table:

```
ALTER TABLE [Order] ADD CONSTRAINT fk_Order_Customer
    FOREIGN KEY (customer_id)
    REFERENCES Customer (id) ON DELETE CASCADE ON UPDATE CASCADE;
```

Then, for the last step, I will create a clustered index on the *gender* column:

```
CREATE CLUSTERED INDEX ix_clustered_gender ON Customer (gender);
```

Here are the indexes we now have on the *Customer* table:

	index_name	index_description	index_keys
1	ix_clustered_gender	clustered located on PRIMARY	gender
2	ix_firstname	nonclustered located on PRIMARY	firstName
3	ix_nonclustered_pk	nonclustered, unique, primary key located on PRI...	id

Figure 24. Indexes on *Customer* table

Results with a clustered index:

1.

With an index	1	2	3	4	5	**Average**
CPU time (ms)	0	0	0	0	0	0
Elapsed time (ms)	494	496	516	547	515	513,6

2.

With an index	1	2	3	4	5	**Average**
CPU time (ms)	16	0	0	15	0	6,2
Elapsed time (ms)	278	266	276	284	279	276,6

Graphs and conclusion:

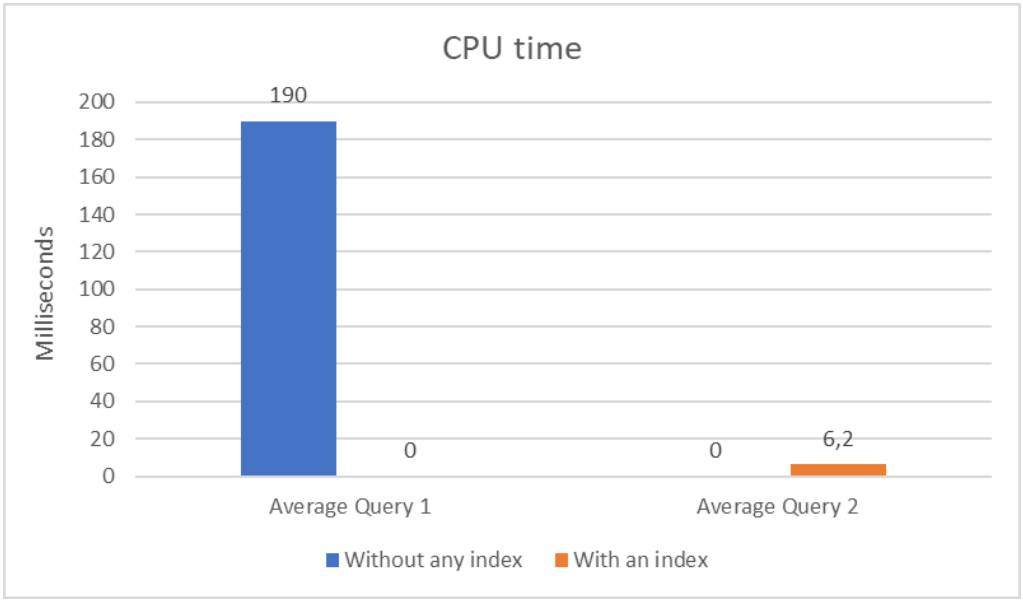


Figure 25. CPU time comparisons

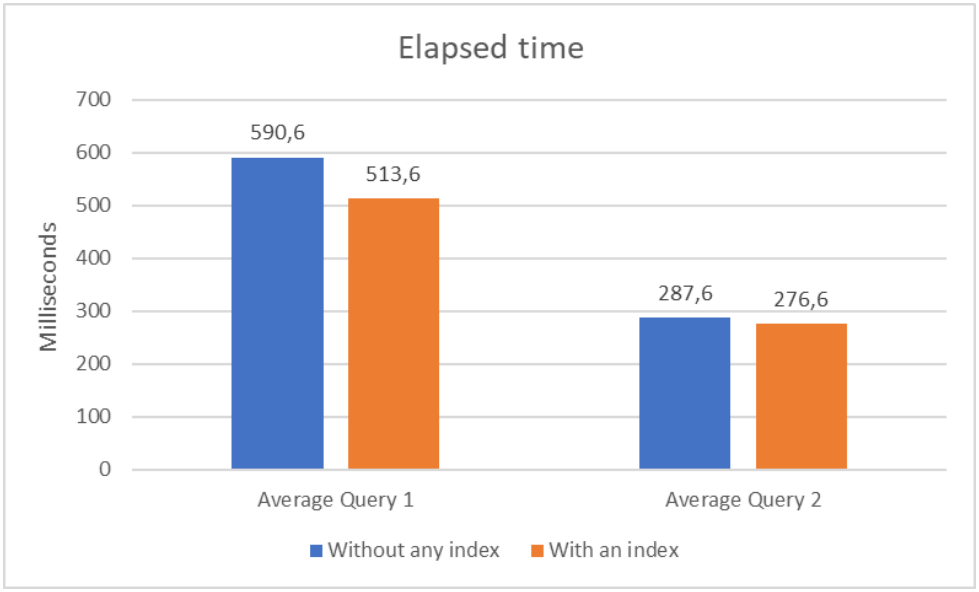


Figure 26. Elapsed time comparisons

1. When the all the data is displayed, there is a slight decrease in elapsed time with the clustered index, this could be due to the data already being sorted (because of the clustered index). However, there is a big decrease in CPU time.
2. However, for the second request, we can see that there is only a slight decrease in CPU time and elapsed time. This could be due to the lack of data for my tests. Indeed, it might be possible to notice a much better improvement on a database with hundreds of thousands of data.

3.2.11 Conclusion of the tests

To conclude these tests, we can see that the creation of indexes, even in the case of small amounts of data, can already reduce performance by almost half. Moreover, what we must remember from these tests is that even if in some cases the performance did not seem to be much improved (in the case of the creation of the clustered index on the *gender* column), we must keep in mind that the database tables are constantly filled and updated. It is therefore necessary to set up the right indexes that will improve the performance of the database to the maximum, even when the database contains hundreds of thousands of data.

3.3 Implementation of the MongoDB database

To start with, I chose to use Docker to implement my MongoDB database. Indeed, Docker allows me to create an implementation (container) for my database that already has all the required dependencies. Thanks to Docker, I do not have to worry about these dependencies, and I can benefit from a database configured in a simple way.

3.3.1 Docker basics and MongoDB Compass

Regarding the basics of Docker, it is important to know how it works. First, after installing Docker on my Windows machine, it is necessary to find a Docker image on the Docker hub that contains the various MongoDB settings. A Docker image is a template that contains a set of instructions that can be run on Docker. You can create them yourself or find them on the Docker hub. For my MongoDB database, I chose an image known and approved by Docker (JFrog 17 March 2021).

Then, from the Docker image, I could simply run a new container from the Docker user interface. This created an instance of my database. It is possible to create as many containers as you want from a Docker image. For my various tests, I will run a single container. In a second step, I installed Compass. Compass is the GUI (graphical user interface) of MongoDB that will allow me to have a visual help to implement my different documents in my database. To connect my database to Compass, I simply had to give the port used.

3.3.2 Creation of the database

NoSQL databases are different from relational databases. They do not use tables and relationships to store data. MongoDB databases are containers of collections. A collection is a grouping of MongoDB documents. Those documents, within a collection, can have different fields. A collection is the equivalent of a table in a relational database. As with tables, a collection is only part of a single database. Finally, the documents are individual records in a MongoDB collection. They can be compared to a row in a relational database (MongoDB).

Concerning the database that I will create for the different tests, I will base myself on the schema used during the creation of the relational database. Indeed, reusing the same schema will allow a better comparison of the different databases as well as a more accurate comparison of the different indexes. To start with, with the help of the MongoDB Compass user interface, I created a database for my tests, which I called *IndexTests*. At the time of its creation, I was asked to create a first collection. So, I created my first collection, which I named *Customer*. In a second step, I generated my different customers. To do this, it is possible to simply open a CSV or JSON file in MongoDB Compass. As this database is a NoSQL database, it is not necessary to create a structure or a schema. Each document within a collection can have its own structure. For the generation of my JSON file, I simply used an online tool (https://json-generator.com/) which allowed me to generate random data in JSON format. As for the relational database, I chose to generate 100’000 *Customers*, 60’000 *Orders*, 80’000 *OrderItems*, and 10’000 *Products*.

Concerning the generation of my data, as the relationships are different from a relational database, I first created my *Customers* and *Products*. In a second step, I documented on MongoDB how to

manage the various relations (One-to-One, One-to-Many). All I had to do was add an attribute to the *Orders* documents that contain the *id* of the desired product. I then had to do the same for *OrderItems* with *Orders* and *Products*. However, unlike primary and foreign keys, there is no check on the existence of the document when referencing it. The absence of a check on the existence of the referenced document endangers the integrity of the database. When a document is deleted, there will be no warning or prohibition, even if the document in question is referenced.

3.3.3 How to create indexes

To create indexes in my MongoDB database, I will use MongoDB Compass. From this interface, it is possible to create and manage indexes. To do so, you just have to select the desired collection, then go to the "Indexes" tab. From this tab, it is possible to create indexes on any field, and to select any type of index. Moreover, in the additional options, it is possible to specify other options (unique or not, sparse, etc.).

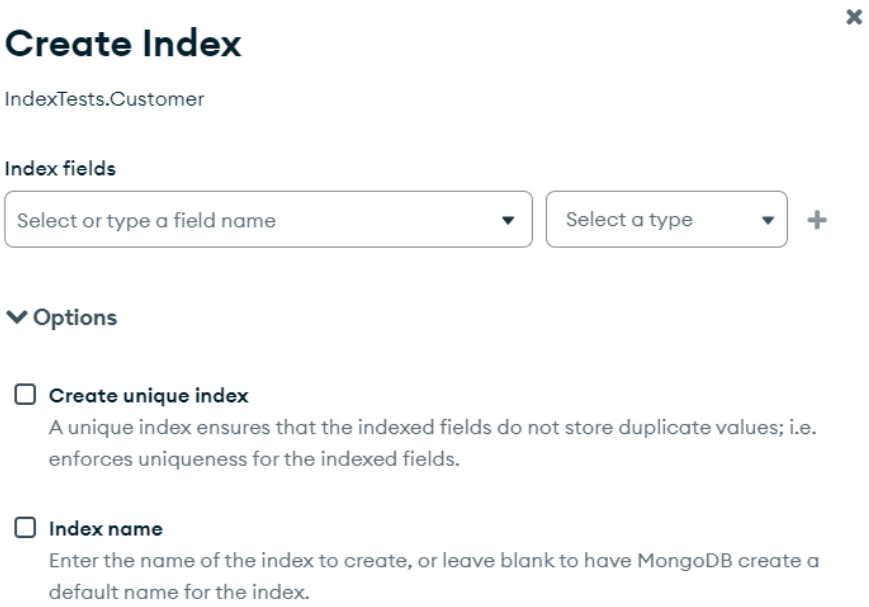
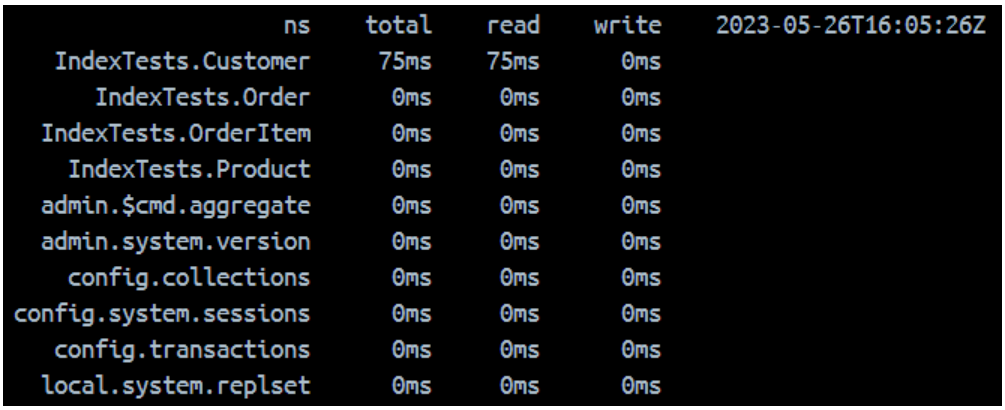


Figure 27. Indexes MongoDB Compass

3.3.4 Observe performance

To analyse performance, I will use the terminal of my MongoDB container, in Docker Desktop. Indeed, through this terminal, I will be able to execute the command *mongotop*, that will allow me to track and report the current read and write activity of my MongoDB instance and also report the statistics of the different collections. Here is an example of the statistics:



ns	total	read	write	2023-05-26T16:05:26Z
IndexTests.Customer	75ms	75ms	0ms	
IndexTests.Order	0ms	0ms	0ms	
IndexTests.OrderItem	0ms	0ms	0ms	
IndexTests.Product	0ms	0ms	0ms	
admin.$cmd.aggregate	0ms	0ms	0ms	
admin.system.version	0ms	0ms	0ms	
config.collections	0ms	0ms	0ms	
config.system.sessions	0ms	0ms	0ms	
config.transactions	0ms	0ms	0ms	
local.system.replset	0ms	0ms	0ms	

Figure 28. MongoDB performance analysis

3.3.5 Querying in MongoDB

It is perfectly possible to execute queries in MongoDB. In fact, it is possible query for ranges, set inclusion and inequalities. It is also possible to use *$* to add conditionals. Queries return a database cursor, that returns batches of documents. This is also possible to add some operations to perform on a cursor, such as skipping results, limiting the number of documents returned and sorting the results (Bradshaw, Brazil & Chodorow 2019, chapter 4).

The *find* method

The find method is used to perform queries. Those queries return a subset of documents in a collection, from no documents at all to the entire documents of the collection. To use this *find* method, we need to start by adding key/value pairs to the query. Here is an example of a simple query:

```
> db.users.find({"age" : 27})
```

Figure 29. Simple query (Bradshaw S., Brazil E. & Chodorow K. 2019)

It is also possible to add another key/value pair to the query. It will be interpreted as an *AND* between the conditions. Here is an example of a two key/value pairs query:

```
> db.users.find({"username" : "joe", "age" : 27})
```

Figure 30. Two key/value pairs query (Bradshaw S., Brazil E. & Chodorow K. 2019)

It is possible to specify which keys the query returns, instead of all of them. For example, if we have a *User* collection, and we are only interested by the username and email, it is possible to specify this in the query as follows:

```
> db.users.find({}, {"username" : 1, "email" : 1})
{
    "_id" : ObjectId("4ba0f0dfd22aa494fd523620"),
    "username" : "joe",
    "email" : "joe@example.com"
}
```

Figure 31. Specify keys returned (Bradshaw S., Brazil E. & Chodorow K. 2019)

It is also possible to exclude a key/value pairs from the results of the query, as follows:

```
> db.users.find({}, {"username" : 1, "_id" : 0})
{
    "username" : "joe",
}
```

Figure 32. Specify keys to exclude (Bradshaw S., Brazil E. & Chodorow K. 2019)

Conditionals

There are also comparison operators corresponding to <, <=, > and >=, respectively *$lt*, *$lte*, *$gt* and *$gte*. For example, if we want to look for users who are between the ages of 18 and 30, we can do as follows:

```
> db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
```

Figure 33. Users between 18 and 30 (Bradshaw S., Brazil E. & Chodorow K. 2019)

There are, of course, many other parameters that could be added to further specify the queries, but these parameters are not necessary for the index tests I am going to perform. Indeed, the parameters mentioned above will be sufficient to perform the various index tests on my MongoDB database.

3.3.6 Creating a unique index on the *id* key (*Customer* collection)

I chose to create a unique index on this key because those identifiers will be unique. I also chose to do it on the *Customer* collection because this is the collection with the most documents, so there will be better results. To create the different queries, as I am using the MongoDB Compass GUI, I do not need to write the whole search function, I can just use the filter field. This works in the same way as the *find()* function, but I only need to type the search condition.

To test this index, I created three different filters (I will also execute them 5 times):

1. **{**id**:** 666**}** *// This one does exist*
2. **{**id: 100000**}** *// This one does not exist*
3. **{**id: **{**$in: **[**9876, 2234**]}}** *// Both do exist*

Then, after testing without index, I created the unique index by using the tool for creating indexes directly in MongoDB Compass.

Results:

1. This filter will display the document with the *id* 666.
   a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
Time (ms)	56	51	57	53	51	53,6

   b. **With an index**:

With an index	1	2	3	4	5	**Average**
Time (ms)	14	0	11	7	5	7,4

2. This filter will not display any document (because the *id* 100000 does not exist).
   a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
Time (ms)	45	56	48	75	54	55,6

   b. **With an index**:

With an index	1	2	3	4	5	**Average**
Time (ms)	1	2	0	3	1	1,4

3. This filter will display two documents; one with the *id* 9876 and one with the *id* 2234.
   a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
Time (ms)	54	54	57	56	65	57,2

   b. **With an index**:

With an index	1	2	3	4	5	**Average**
Time (ms)	0	3	6	4	0	2,6

Graphs and conclusion:

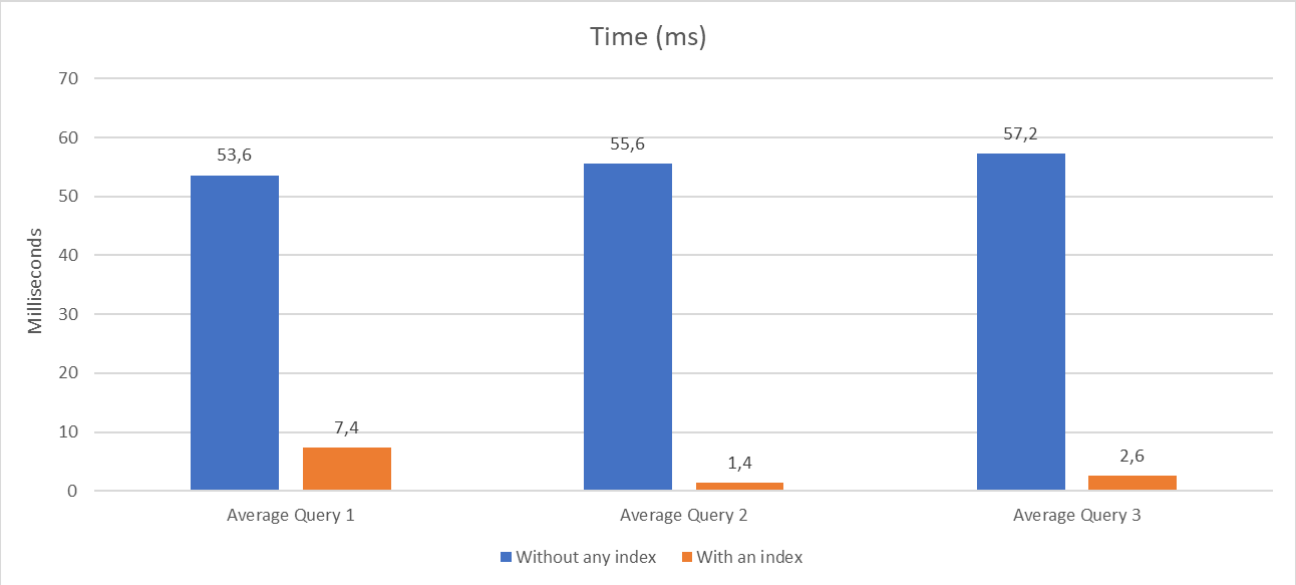


Figure 34. Time comparisons

1. For this first query, we can see a clear difference in performance when creating a unique index. When this index is created using MongoDB Compass, it is possible to specify the sort order of the index. As my identifiers were unique, it was possible to sort them through an index, which resulted in a significant improvement in time.
2. Regarding this second query, we can also see that time has been drastically reduced even when the identifier we are looking for does not exist. This can also be explained by the fact that the index is ordered.
3. Regarding this last query, we can see a big improvement in time. This shows that this index improves performance even when the query returns more than one document.

3.3.7 Creating a compound index on the *firstName* and *lastName* keys (*Customer* collection)

I chose to create a compound index on those two keys because they are often linked in a search. In addition, MongoDB uses this example to highlight the advantages of compound indexes.

To test this index, I created three different filters (I will also execute them 5 times):

1. **{**firstName: "Mccoy", lastName: "Wiley"**}** // This one does exist
2. **{**firstName: "Mccoy", lastName: "Jeanrenaud"**}** // This one does not exist
3. **{**firstName: **{***$in*: **[**"Eaton", "Nadia"**]}**, lastName: **{***$in*: **[**"Daugherty", "Hebert"**]}**
   // Both do exist

Then, after testing without index, I created the compound index by using the tool for creating indexes directly in MongoDB Compass.

Results:

1. This filter will display the documents with the *firstName* Mccoy and the *lastName* Wiley.
   a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
Time (ms)	56	93	54	67	64	66,8

   b. **With an index**:

With an index	1	2	3	4	5	**Average**
Time (ms)	4	2	8	0	2	3,2

2. This filter will not display any document (because the *lastName* Jeanrenaud does not exist).
   a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
Time (ms)	174	137	102	125	90	125,6

   b. **With an index**:

With an index	1	2	3	4	5	**Average**
Time (ms)	0	0	2	3	1	1,2

3. This filter will display two documents; one with the *firstName* Eaton and the *lastName* Daugherty and one with the *firstName* Nadia and the *lastName* Hebert.
   a. **Without any index**:

Without any index	1	2	3	4	5	**Average**
Time (ms)	53	80	102	115	93	88,6

   b. **With an index**:

With an index	1	2	3	4	5	**Average**
Time (ms)	0	1	2	0	2	1

Graphs and conclusion:

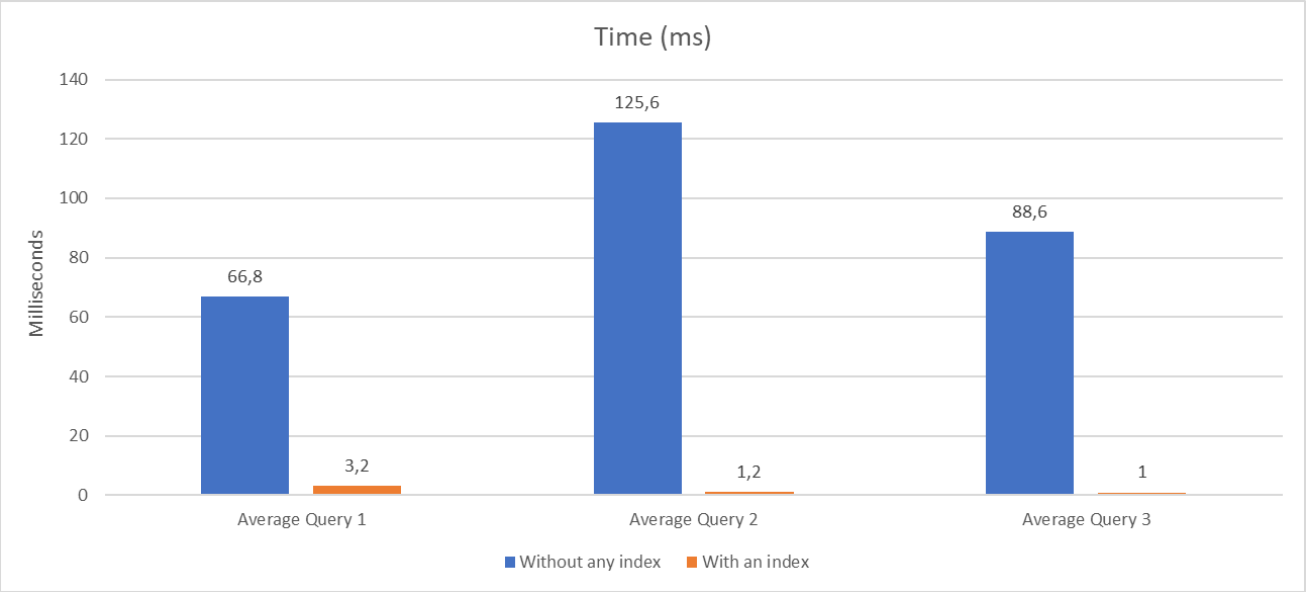


Figure 35. Time comparisons

Whether it is the first, second or third query, we can see that creating a compound index greatly improves performance. Indeed, as shown in the graph above, we can see that this index improves time, whether the person exists or not. Moreover, even when the query returns two documents, this index is still useful.

3.3.8 Semi-structured data in MongoDB

MongoDB is also the database for storing, analysing, and processing semi-structured data. Semi-structured data cannot be stored in a relational database, but it can be stored in a NoSQL database. Because the data is similar to unstructured data and some of MondoDB's indexes have been analysed, there will be no tests on a database containing semi-structured data. As the aim of this thesis is to compare indexes between the different databases implemented, the choice was made to keep the same data structure. However, it should be noted that MongoDB offers a number of tools for managing unstructured data that are not mentioned in this thesis. In addition, there are other indexes not covered in this thesis that may be interesting to implement for a semi-structured database.

3.3.9 Conclusion of the tests

To conclude those tests, we can see that, even if the quantity of data is small, indexes improve database performance enormously. This can be seen clearly in the two tests above, where the creation of an index has drastically reduced time. Although this thesis only tests two indexes, there are many others in MongoDB, which are particularly useful for semi-structured and unstructured data. In conclusion, we can see that the indexes available in NoSQL databases can very well improve performance, whether for structured, semi-structured or unstructured data. However, these databases are not recommended for structured data, due to their lack of constraints, particularly when it comes to relationships between documents. When it comes to semi-structured data, these NoSQL databases offer schemas that can handle a small structural part of the data. In addition, partial and sparse indexes are perfectly suited to unstructured data.

4 Conclusion

To conclude this thesis, I would like to highlight that it is essential to choose the right database for the data to be stored. As the amount of data continues to increase, it is essential to set up the right database from the outset. Then, in the case of a relational database, it is important to analyse which data will be accessed the most, so as to know which columns to place the various indexes on, and which indexes to set up. In addition, it is also important to know which queries will be executed the most, in order to analyse performance and, if necessary, set up other indexes. These steps are crucial, because maintaining good database performance is vital. As data is an extremely important asset, often used to manage an entire company, it is essential to be able to access it efficiently. Extracting data is therefore a major challenge, as it will be used on a regular basis in Business Intelligence tools.

When it comes to semi-structured and unstructured data, it is becoming increasingly important to know how to manage it effectively. Indeed, these data are often extremely abundant in Big Data, and are becoming more and more present. Unstructured data can be found in many forms, including images, audio, and media. The advantage of this data is that it can be easily stored in data lakes, for example. However, they need to be processed before they can be used in Business Intelligence tools. It is therefore necessary to be able to extract them efficiently, thanks in particular to indexes. MongoDB's indexes are well suited to semi-structured and unstructured data. As in the case of relational databases, the stakes are the same: to maintain good performance, you need to be able to manage the company using this data as effectively as possible.

To conclude, this thesis describes and tests a few indexes, but there are many others that may be more suitable depending on the data. The main aim of this thesis is above all to introduce the creation of indexes and to highlight the importance of index creation in maintaining and even increasing database performance. In addition, this thesis also introduces some performance tests, through time analysis for example. This provides a partial view of database performance.

Sources

Ardeleanu, S. 2016. Relational Database Programming: A Set-Oriented Approach. Apress. Bucharest, Romania. E-book. Accessed: 25 April 2023.

Board Infinity. Relational Model in DBMS. URL: https://www.boardinfinity.com/blog/content/images/2022/12/Your-paragraph-text--90-.jpg. Accessed: 22 March 2023.

Bradshaw, S., Brazil, E. & Chodorow, K. 2019. MongoDB: The Definitive Guide, 3rd Edition. Third Edition. O'Reilly Media, Inc. United States of America. E-book. Accessed: 22 May 2023.

DbSchema. One to zero or many. URL: https://dbschema.com/documentation/img/layouts/one-to-0-many.png. Accessed: 16 April 2023.

DbSchema. Table layout. URL: https://dbschema.com/documentation/img/layouts/layout-table.png. Accessed: 16 April 2023.

Harrington, J. L. 2016. Relational Database Design and Implementation. 4th Edition. Morgan Kaufmann. Amsterdam. E-book. Accessed: 25 April 2023.

Harrison, G. 2021. MongoDB Performance Tuning: Optimizing MongoDB Databases and their Applications. Apress. Kingsville, VIC, Australia. E-book. Accessed: 10 March 2023.

Isson, J. P. 2018. Unstructured Data Analytics. Wiley. Hoboken, New Jersey. E-book. Accessed: 22 May 2023.

JFrog 17 March 2021. A Beginner’s Guide to Understanding and Building Docker Images. JFrog. URL: https://jfrog.com/devops-tools/article/understanding-and-building-docker-images/. Accessed: 17 May 2023.

Laiho, M. (images), Kari Silpiö 2016. SQL Server Indexes. Haaga-Helia. URL: http://myy.haaga-helia.fi/~bit/swd8tf040/Slides/DBD_SQL_Server_Indexes.pdf. Accessed: 29 March 2023.

Lightstone, S. S., Teorey, T. J. & Nadeau, T. 2010. Physical Database Design. Morgan Kaufmann. Amsterdam. E-book. Accessed: 26 May 2023.

Microsoft 2023. What is SQL Server Management Studio (SSMS)? URL: https://learn.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver16. Accessed: 12 April 2023.

MongoDB. Model One-to-Many Relationships with Document References. URL: https://www.mongodb.com/docs/manual/tutorial/model-referenced-one-to-many-relationships-between-documents/. Accessed: 19 May 2023.

MongoDB. MongoDB Compass: Databases. URL: https://www.mongodb.com/docs/compass/current/databases/. Accessed: 17 May 2023.

MongoDB. Partial Indexes. URL: https://www.mongodb.com/docs/v6.0/core/index-partial/. Accessed: 11 April 2023.

MongoDB. Relational vs. Non-Relational Databases. URL: https://www.mongodb.com/compare/relational-vs-non-relational-databases. Accessed: 22 March 2023.

MonkeyLearn. Unstructured vs Structured Data. URL: https://monkeylearn.com/static/306f5989991713a9d77314a514e161c2/899e8/image5.webp. Accessed: 28 March 2023.

Nesavich, A. & Inmon, W. H. 2007. Tapping into Unstructured Data: Integrating Unstructured Data and Textual Analytics into Business Intelligence. Pearson. Upper Saddle River, NJ. E-book. Accessed: 26 May 2023.

Patel, J. M. 2020. Getting Structured Data from the Internet: Running Web Crawlers/Scrapers on a Big Data Production Scale. Apress. Specrom Analytics, Ahmedabad, India. E-book. Accessed: 22 May 2023.

Petrov, A. 2019. Database Internals. O'Reilly Media, Inc. Sebastopol, Canada. E-book. Accessed: 25 April 2023.

Ryan, M. 2021. Deep Learning with Structured Data. Manning Publications. Shelter Island. E-book. Accessed: 22 May 2023.

Silpiö, K. 2010. Basics of DB Transactions. Haaga-Helia. URL: http://myy.haaga-helia.fi/~bit/swd8tf040/Slides/DBD_TransactionBasics.pdf. Accessed: 22 May 2023.

Silpiö, K. 2015. File organization and Indexes. Haaga-Helia. URL: http://myy.haaga-helia.fi/~bit/swd8tf040/Slides/DBD_File_Organization_and_Indexes.pdf. Accessed: 26 May 2023.

Silpiö, K. 2018. Quick Reference for the Index Lab. Haaga-Helia. URL: http://myy.haaga-helia.fi/~bit/swd8tf040/Slides/DBD_Index_Lab_Quick_Reference.pdf. Accessed: 12 April 2023.

Silpiö, K. 2018. SQL Server databases under the hood. Haaga-Helia. URL: http://myy.haaga-helia.fi/~bit/swd8tf040/Slides/DBD_Databases_Under_the_Hood.pdf. Accessed: 25 April 2023.

Silpiö, K. 2020. Basics of indexes. Haaga-Helia. URL: http://myy.haaga-helia.fi/~bit/swd8tf040/Slides/Drawing_indexes.pdf. Accessed: 29 March 2023.

Strate, J. 2019. Expert Performance Indexing in SQL Server 2019: Toward Faster Results and Lower Maintenance. Third Edition. Apress. Hugo, MN, USA. E-book. Accessed: 10 March 2023.

Teorey, T. J., Lightstone, S. S., Nadeau, T. & Jagadish, H. V. 2011. Database Modeling and Design. 5th Edition. Morgan Kaufmann. Amsterdam. E-book. Accessed: 25 April 2023.

University of Newcastle Library 2023. Research Methods
URL: https://libguides.newcastle.edu.au/researchmethods. Accessed: 11 March 2023.

Wolff, R. 16 November 2020. What Is Semi-Structured Data? MonkeyLearn Blog. URL: https://monkeylearn.com/blog/semi-structured-data/. Accessed: 24 March 2023.

Yaseen, A. 3 May 2018. Designing effective SQL Server clustered indexes. SQLShack. URL: https://www.sqlshack.com/designing-effective-sql-server-clustered-indexes/. Accessed: 6 April 2023.

Yaseen, A. 7 May 2018. Designing effective SQL Server non-clustered indexes. SQLShack. URL: https://www.sqlshack.com/designing-effective-sql-server-non-clustered-indexes/. Accessed: 4 April 2023.

Appendices

Work Plan

Tasks and sub-tasks	Resources required	Estimated beginning	Estimated end	Actual start	Actual end	Notes
Writing of the introduction (1/3)	-	**14.03.2023**	**16.03.2023**	**11.03.2023**	**14.03.2023**	**Listen and analyse seminar presentations in two seminars**
Theoretical framework (1/3)	**Efficient Indexing for Structured and Unstructured Data (Patil, Manish Madhukar)**	**17.03.2023**	**14.04.2023**	**20.03.2023**	-	**Listen and analyse seminar presentations in two seminars**
Learn and write about structured data (1/3)	Internet research, Database Developer Materials	17.03.2023	22.03.2023	20.03.2023	21.03.2023	
Learn and write about semi-structured and unstructured data (1/3)	Unstructured Data Analysis (Matthew Windham), Internet research	23.03.2023	27.03.2023	21.03.2023	28.03.2023	
Learn and write about some indexing methods for structured data (1/3)	Internet research, Database Developer Materials, Expert Performance Indexing in SQL Server 2019 (Jason Strate)	28.03.2023	04.04.2023	29.03.2023	06.04.2023	
Learn and write about some indexing methods for semi-structured and unstructured data (1/3)	MongoDB Performance Tuning: Optimizing MongoDB Databases and their Applications (Guy Harrison, Michael Harrison), Internet research	05.04.2023	14.04.2023	06.04.2023	12.04.2023	
Carrying out the study (2/3)	-	**15.04.2023**	**30.04.2023**	**12.04.2023**	-	**Enrol to seminar presentation**
Tests on structured databases (Microsoft SQL Server) (2/3)	Internet research, Database Developer Materials, Expert Performance Indexing in SQL Server 2019 (Jason Strate)	15.04.2023	21.04.2023	12.04.2023	21.04.2023	Corrections based on Outi's comments --> 25.04.2023
Tests on semi-structured and unstructured databases (MongoDB, Docker) (2/3)	MongoDB Performance Tuning: Optimizing MongoDB Databases and their Applications (Guy Harrison, Michael Harrison), Internet research	22.04.2023	30.04.2023	08.05.2023	15.06.2023	
Results (3/3)	-	**01.05.2023**	**07.05.2023**	-	-	**Enrol to seminar presentation**
Discussion, Sources and Appendices (3/3)	-	**08.05.2023**	**12.05.2023**	-	-	
Finalization, Maturity test, Pair review (3/3)	-	**13.05.2023**	**22.05.2023**	-	-	