UCLouvain

LOUVAIN
School of Management

Louvain School of Management

Indexing Methods Performances on Usual Finance Datasets

Auteur·e(s) : LECHIEN Grégoire
Promoteur·rice(s) : KOLP Manuel
Année académique 2022-2023
Travail de fin d'études (TFE) en vue d'obtenir le titre de
Master (60) en Sciences de Gestion
Horaire de jour / Horaire décalé

Abstract

This work examines the impact of primary and secondary indexes, both in hashtable and B-tree forms, on query optimization. It also offers insights into the scenarios where secondary indexes can significantly enhance query times. This study provides practical guidance for effectively utilizing secondary indexes to improve database query performance, catering to both theoretical understanding and practical implementation.

Contents

Chapter 1

Introduction

In today's data-centric world, databases serve as the backbone of modern businesses, driving critical operations and facilitating data-driven decision-making. As data volumes continue to grow exponentially [19], the performance and efficiency of databases become paramount for organizations to remain competitive and agile [42] [20]. Among the various mechanisms that impact database performance, indexes play a central role in accelerating data retrieval and query processing [16].

The efficient use of indexes in a database environment can significantly improve query response times and enhance overall system performance. Indexes enable faster data access by creating organized data structures that expedite the search and retrieval of information. However, the effectiveness of indexes is intricately linked to their precise configuration and alignment with the unique needs of a database[35].

This project aims at exploring the different indexes and their specificities aswell as the critical role of indexes in optimizing database performance, with a specific focus on their impact on query time. Through comprehensive experimentation and analysis, we seek to investigate the scenarios in which the different indexes yield substantial performance improvements and those in which their effects are less pronounced or even counterproductive.

The objective of this study is to gain a deeper understanding of the intricate relationship between indexes and query execution times and how the work. We will explore various index types, including primary indexes, secondary indexes, and multi-level indexes. And we will test their effectiveness under different query conditions and database sizes. By conducting a series of carefully designed experiments, we will observe how index configurations influence query performance and identify the most efficient indexing strategies for specific use cases.

Additionally, we will explore real-world database scenarios : datasets related to finance and direct marketing campaigns, to showcase the practical implications of index optimization in business. Understanding the performance implications of indexes in these scenarios will provide a concrete example of how mastering indexes can bring valuable insights for businessmen to improve their databases uses for data analysis.

This study may serve as a practical guide for novices in databases, offering valuable insights and step-by-step explanations to optimize database indexing. It will help aspiring entrepreneurs with the essential knowledge about the power of indexes for their business, maximizing

query performance and data efficiency.

In conclusion, this TFE embarks on a journey to explore the vital role of indexes in database performance. By investigating the connection between index utilization and query execution times, we aim to empower businesses and organizations with the knowledge to make informed decisions on index design and configuration. Ultimately, this study seeks to unlock the full potential of databases, paving the way for improved efficiency, competitive advantage, and data-driven success in the ever-evolving digital landscape.

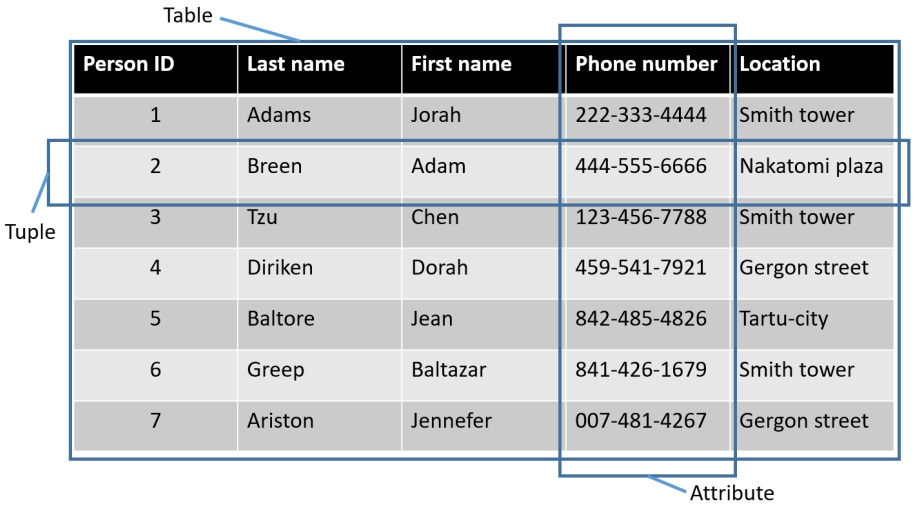
Chapter 2

Theoretical Part

2.1 Relational Databases

2.1.0.1 General Organization

Relational databases are a type of database that organizes and stores data in tables [5][36]. Each table corresponds to a specific 'type of object' or entity. For instance, if we want to manage information about people, we would have a table named 'People.' Each row in this table, referred to as a 'tuple,' represents an individual person, while each column represents a specific attribute associated with the person. An illustrative example of the 'People' table can be seen in Fig.2.1.



Person ID	Last name	First name	Phone number	Location
1	Adams	Jorah	222-333-4444	Smith tower
2	Breen	Adam	444-555-6666	Nakatomi plaza
3	Tzu	Chen	123-456-7788	Smith tower
4	Diriken	Dorah	459-541-7921	Gergon street
5	Baltore	Jean	842-485-4826	Tartu-city
6	Greep	Baltazar	841-426-1679	Smith tower
7	Ariston	Jennefer	007-481-4267	Gergon street

Figure 2.1: Table for People

However, databases usually are made off multiple and interconnecting tables : for example with a table representing the loving relation between the people or the parenting relations as on Fig.2.2. These tables are linked by the ID of the people represented, the tuple representing the person from the 'People' table is addressed by its ID in the other tables. For example the tuple (1,4) of the Lovers table means that the person with the ID 1 loves the person with the ID 4 and we have to look in the People table to get more informations on who these

ID's represent and what are their other attributes.

People Table

Person ID	Last name	First name	Phone number	Location
1	Adams	Jorah	222-333-4444	Smith tower
2	Breen	Adam	444-555-6666	Nakatomi plaza
3	Tzu	Chen	123-456-7788	Smith tower
4	Diriken	Dorah	459-541-7921	Gergon street
5	Baltore	Jean	842-485-4826	Tartu-city
6	Greep	Baltazar	841-426-1679	Smith tower
7	Ariston	Jennefer	007-481-4267	Gergon street

Lovers Table

ID lover	ID loved
1	4
4	1
2	7
7	2
3	7

Parenting Table

ID children	ID Parent1	ID Parent2
5	1	4
6	2	7

Figure 2.2: Linked People, Lovers and parenting tables

In the continuation of this work, we will specifically focus on the SQL (Structured Query Language) Databases. SQL is the most common language [30] used to manipulate relational databases, being a powerful query language either on traditional relational and on highly distributed and scalable systems that process Big Data, (datasets with high volume, velocity and variety) [40]

2.1.0.2 Queries

The most used operation on databases, and the one we will be focusing in this work, is the query. It is the operation used to read tuples from a database. Queries in SQL typically look like this [22] :

```
SELECT * FROM people WHERE location = 'Gergon street';
```

The upper letter words are part of the syntax of the language, they are useful to tidy up the query and separate it in three different parts or clauses [13] :

- The first part is the SELECT clause : it allows you to tell the attributes of the tuples you want in the response, if you want the whole tuple you can select all by putting an asterisk '*', if you want multiple attributes you can writes them with the pattern 'attribute1, attribute2, ...'.
- The second part is the FROM clause : it's used to indicate the tables you want to take informations from. There can be multiple table, separated by a coma, if you need informations from multiple tables. The tables can also be fused in different ways [26], the most known being using a 'JOIN', but this will not be discussed in this work.

- The third part is the condition clause : it contains the conditions that your tuples need to respect in order to be part of the response. Any tuples that don't respect one condition or more will be pruned out of the results [33]. The different conditions are separated by the 'AND' syntactic word.

These explanations provide a foundational understanding of the few concepts that will be relevant for our work. It is essential to acknowledge that the SQL is considerably more intricate than what we've covered here. Feel free to investigate the cited links if it finds your interest. However, our current level of comprehension of the queries is sufficient for the tasks at hand, delving deeper into the intricacies of SQL might not be necessary for the scope of our project and so we won't.

2.1.0.3 Keys

One last thing it is important to understand is that in relational databases, the tables have some keys : the keys of a table are the sets of attributes for which each tuple contained in the database is **unique**[28].
Usually the key is decided at the creation of the table and then database is filled with the tuples, this means that keys represent the attributes for which the input tuples must be different. Still when using a database that already exist, it might be important to find what attributes might be keys, and some methods exist to find such keys [27].

The most common case for tables is to have an 'ID' attribute used as a key, (it is even common to have it automatically added by the DataBase Management Systems or DBMS, the software used to manage you databases)[6]. Indeed, it ensures that your tuples have at least one simple key by adding an incremental ID to each tuple, making sure that even with namesake the 'People' table will not contain twice the same tuple to avoid redundancy.

For example when looking at the People table from Fig.2.2, the attributes [Person ID], [Last name], [First name] and [Phone number] each are a key for the table as, for these attributes, there isn't twice the same value in any tuples. [Location] however is not a key by itself as the values "Smith tower" and "Gergon street" appear multiple times. Still, the [location] attribute might have been part of a key if the table contained another column for which, joined with the [location] column, every tuple would be unique [23] (for example with column for the number of the house in the street or something else).

To show such composed keys, take a look at the Fig.2.3. This is a modified version of the previous Lovers Table in which we added the last tuple ('7','3'), forming a love triangle between 2,3 and 7. The Lovers table now has no simple key because the value '7' appears twice in [ID loved] and in [ID lover]. In this case, the key would be a composed key : the set of both attributes : [ID lover, ID loved]. Indeed, when selecting both attributes we don't have the same couple of values repeating twice. Making sure that no loving situation is repeated in the table to avoid redudancy[8].

Lovers Table

ID lover	ID loved
1	4
4	1
2	7
7	2
3	7
7	3

Figure 2.3: Lovers table with a composed key

Note that keys are considered as such only if they contain no subset that also is a key.

Finally, to make sure that each table has at least one key, one of the keys must be chosen as the Primary key by the author. The other keys will be denoted as candidates or secondary keys [17].
Primary keys are mandatory in each table as they are essential for the indexing that will be addressed in the following sections. This is the main reason that most DMBS's will automatically add an usually hidden attribute containing the ID of the tuple when no primary key is indicated while creating a table. This allows to use the newly created attribute as the missing primary key.

2.2 The Relational Database Storing

To understand better the different indexing methods we must first come back to the bases of computer memory and on how relational databases are physically stored in hard disk[15].

Hard disks can be seen as a pile of platters which can each be seen as some sort of disks or CD's as shown on Fig. 2.4[3][2].

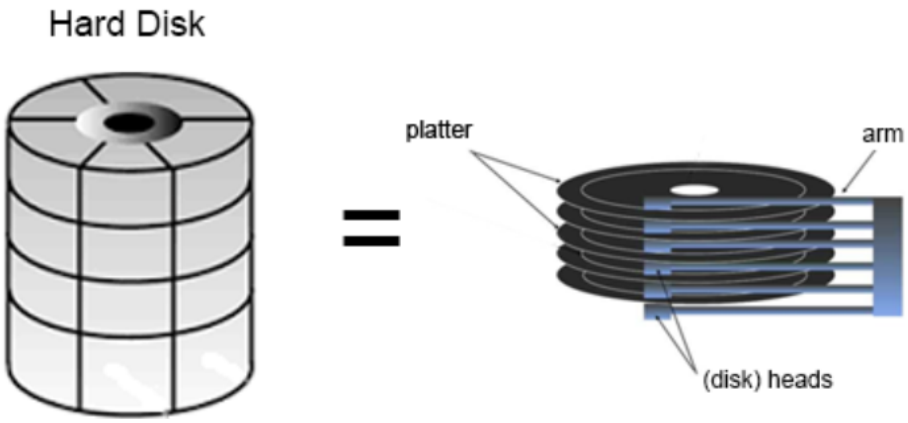


Figure 2.4: Hard Disk subdivision in platters

These platters are themselves made of sectors[1][21] (which may be called blocks when addressed in the software universe), each made off the same number of bytes, usually 512 bytes (see Fig2.5). Those bytes can only be read and written on altogether, which mean that to read or write one byte of the block, one has to read or rewrite them all.

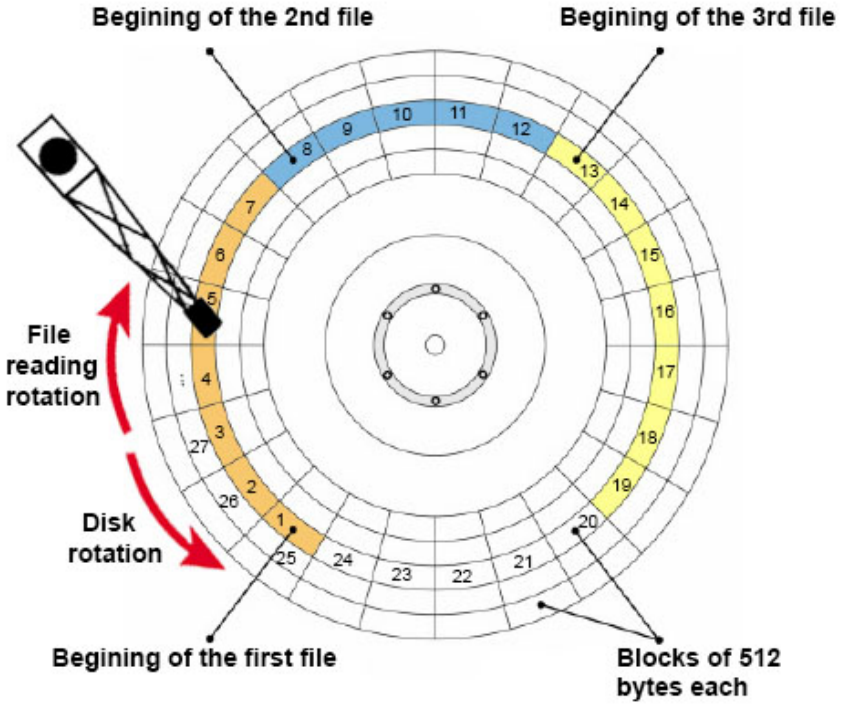


Figure 2.5: Disk subdivision in blocks

You can also observe that the blocks are ordered in such a way that makes it faster to read one file going from block 1 to block 7 than if the files were filling blocks in a random order.

Of course the contemporary hard disks are more complex devices that what is presented here [12] but the main idea remains the same, same for SSD's : they also work with such systems and blocks [14].

This organisation in blocks is important to keep in mind. Indeed, the indexing methods we will be discussing about are closely related to the blocks organisation [25] as their purpose is to be some sort of a map of the blocks based on the values they contain.

2.3 The Indexing Methods

A database index is a data structure that improves the efficiency of queries in a database. It is created on one or more columns of a table. I can be seen as a sort of lookup table, allowing the database engine to locate data more efficiently by limiting the number of block reading. It speeds up query execution by creating a separate structure containing key values that will be used as a map to guide the engine through the huge amount of blocks of the database. Indexes are very useful to optimize database performance, mainly for querying but also when writing tuples.

These indexes being map with key values to facilitate the navigation between the data, There may be multiple ways to select what strategic data to keep and how. Thus, multiple indexes (or indexing methods) exists, Each having some pros and some cons. The different indexes are only related to one table at the time. Meaning that Even if the databases are multi-table databases we will address it as if it was a one-table database. The other tables can be treated as if the were whole distinct databases. [38][37]

2.3.1 Primary Index

As explained above, we have two entities, two parts of memory that will be important for indexing methods :

- The index table : the part of the memory where the indexes are stored (the map).
- And the database table : the part of the memory where the tuples will be stored. Such table contain the whole database which can be really huge in terms of memory and fills up multiple blocks. It will then be represented as a sequence of memory blocks.

The first indexes we will address are the primary indexes. This indexing method requires to sort the tuples of the database table. The tuples are sorted by their value for the primary key attribute, each table having one and only one primary key.

The index table will store pairs of value : the primary value of the first tuple of every block, coupled with a pointer of the block it refers. In the case of a query, keeping the first value of each block allows the engine to directly read the right block. Indeed, the engine will perform a bisection on the values stored in the index table, returning the pointer of the block where the queried value is stored.

As an example, on Fig.2.6 is one simplified example database and the index table corresponding. On this example we consider that only four tuples would fit in a block (it is referred as the blocking factor) and that they fit the block size perfectly with no space left at the end of each block. In practice it may be more complex and multiple adaptations are possible but they will not be discussed here.

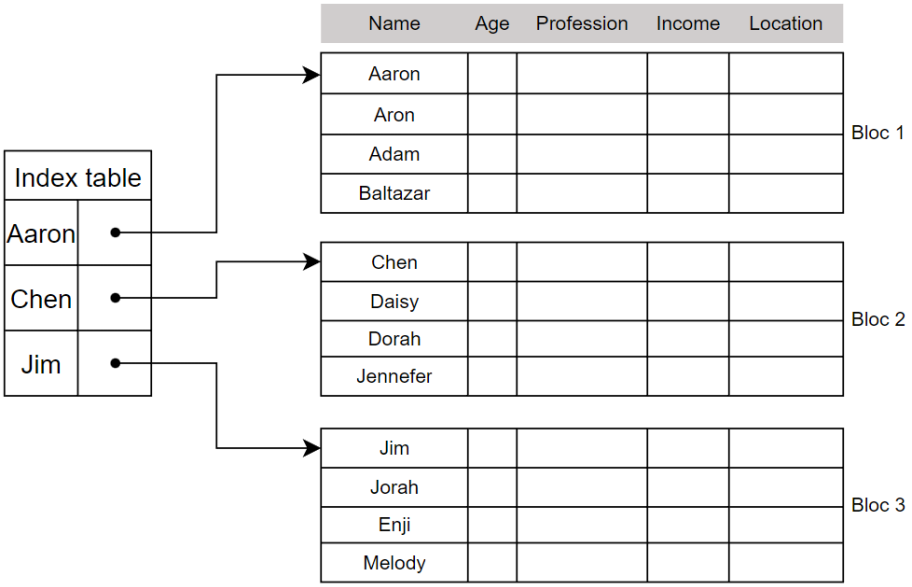


Figure 2.6: Primary Index

We need to keep in mind that real world databases are way bigger than the one represented on the Fig.2.6. And the time gains are proportional to the size of the database, meaning they really are important compared to what we may think observing the example.

The problem with this method is that, as for any cluster indexing method [29], we have to order the data by one and only one attribute. This means that we can't have such index on any other attribute at the same time. It is a problem as it will not improve the queries on any other attribute than the one its sorted by. To settle this, we need another type of indexes.

2.3.2 Secondary Indexes

Secondary indexes are indexes on tables that have already a primary index, meaning that they are already sorted on another key attribute. This means that in addition to the primary index over a key attribute, we can have additional indexes that can handle any attribute, would it be a key attribute or not.

The secondary indexes are indexes that will work on datasets already sorted on another attribute. This means that we can't use the order in which the tuples are to spare some entries. The secondary indexes will be dense indexes.[41]

The secondary indexes are dense indexes. This means that when a secondary index is created on an attribute, every value of the attribute will have an entry in the index table.

As the secondary indexes can be about non-key attributes, multiple occurrences of the

same values may appear. In This case the pointer will not be a simple pointer but a list of pointers, one pointer for each occurrence of the value(see Fig.2.7 with the value'22' appearing multiple times).

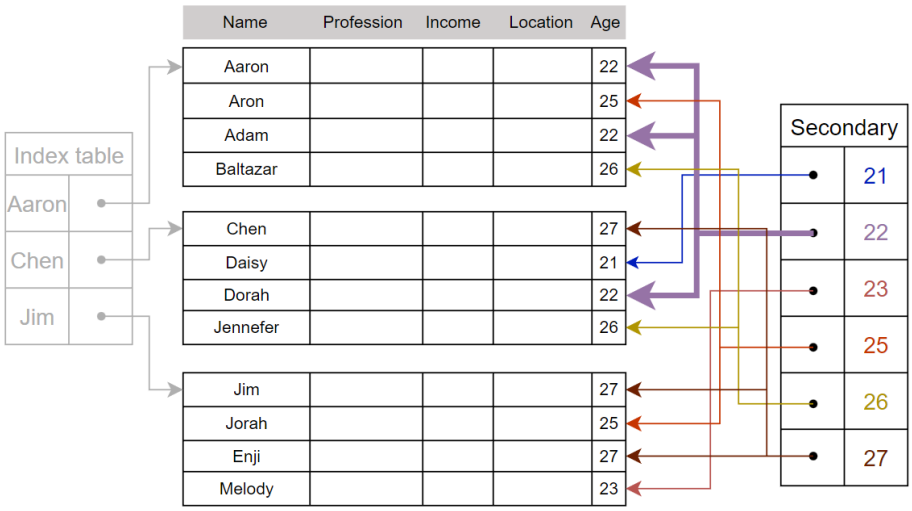


Figure 2.7: Secondary Index

2.3.3 B-tree Indexes

B-trees have different implementations but we will discuss here the one used by the engine InnoDB as it is the one that will be tested in the section 3 Experiments [11] [7].

B-tree indexes are multi-level indexes. This means that the index table is no more stored in the form of a simple ordonned list but that the indexes are stored in a more complex (multi-dimansional) structure, here in the form of a tree.

This structure is made such that the pointer to the data tuples is directly found following a deterministic path in the tree. Sparing the need to perform a full bisection to find the the pointer you are looking for.

Let's show this in an example : Looking at the Fig.2.8, you want to read from the memory the tuple about Daisy in the fastest way possible. You start on the root node. The first nood contains the values 'Chen', 'Jim' and 'Enji' aswell as 5 pointers, the instructions in this case are to follow :

- The first pointer is for values that are lower or equal[1] than Chen.
- The second pointer must be followed for values between Chen and Jennefer.
- The third pointer must be followed for values between Jennefer and Jorah.
- The fourth pointer must be followed for values greater than Jorah

[1]lower in strings means that it comes before in alphabetical order

- The last pointer is not used in the root node but for leafs and internal nodes, it points to its right neighbour, if there is one. This pointer is used only when deleting or adding new values to the dataset [2].

With 'Daisy' as a value, we will thus follow the second pointer (Chen ¡ Daisy ¡ Jennefer) And find the Daisy Value In the leaf node in second position, which means the pointer leading to the Daisy tuple will be the second one.

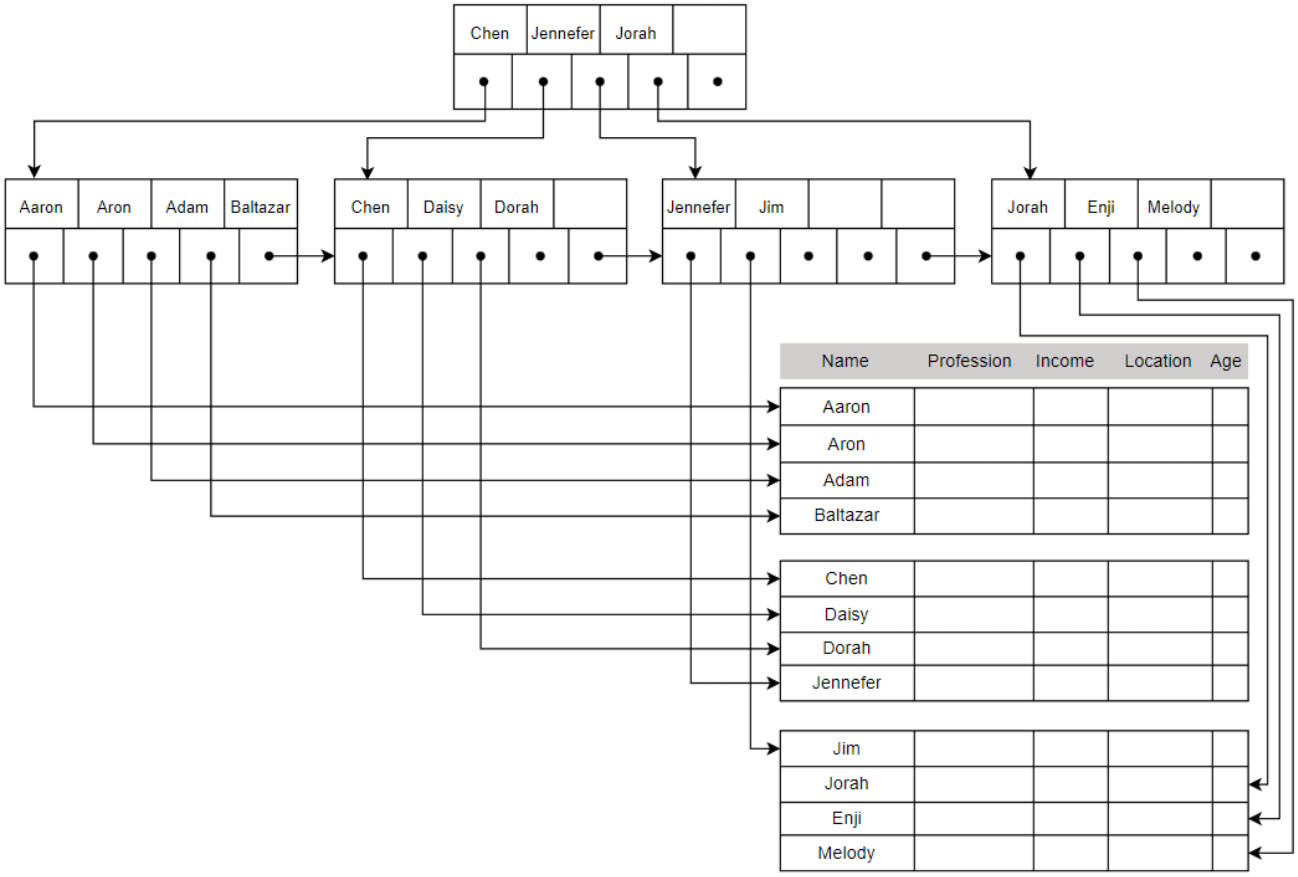


Figure 2.8: B-tree Index

A tree for the same database may have a different look, for example we could have had four values in the first 3 leaf nodes and only the value Melody in the last leaf node. The tree look depends on the order the values were added or deleted but the procedure explained above will always bring you to the right pointer, whatever the look of the tree is [9].

The B-tree index of the Fig.2.8 is a really specific version of a B-tree because, as an example database, it stores very few values compared to usual databases. If you were to add at least four values to the the structure, we would have no place left for the fourth value with this exact structure. To overcome this, The tree will add an auxiliary row of nodes : the internal nodes.

Each internal node of the newly created row will work the same way as the root node in the Fig2.8 but their last pointer will now points to their right neighbour and they will be

[2]Add and delete features of the dataset will not be discussed here as we are interested in the queries time and not in the modification of the dataset

topped by another root node. For example, on Fig2.9 is one index B-tree on the alphabet letters with internal nodes.

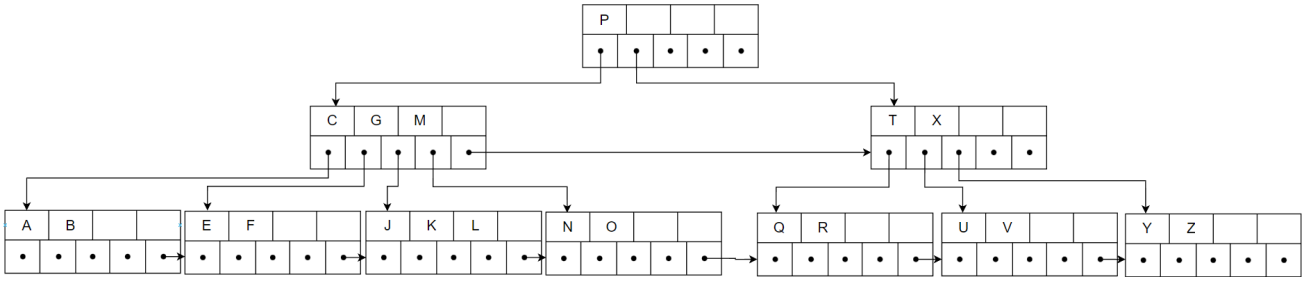


Figure 2.9: B-tree Index with internal nodes

2.3.4 Combined Indexes

In the case of a composed primary key, the attribute used to sort the data and stored in the the index is no more a single attribute but is replaced by a pair of attribute or even a tuple of attributes, one for each attribute of the primary key [39] [18]. The data is then ordered by the first attribute and then its equities are ordered by second attribute then third, etc. until all the data is sorted.

This system also works for any type of indexes, not only the primary indexes. The difference will be that the data won't be sorted in the data table but the values in the index table will be.

When using a primary index, the fact data will be sorted in the same order as the attribute placed in the SQL script when creating the index implies that this specific order is very important because it will determine how all the data is sorted. You can see an example on Fig.2.10.

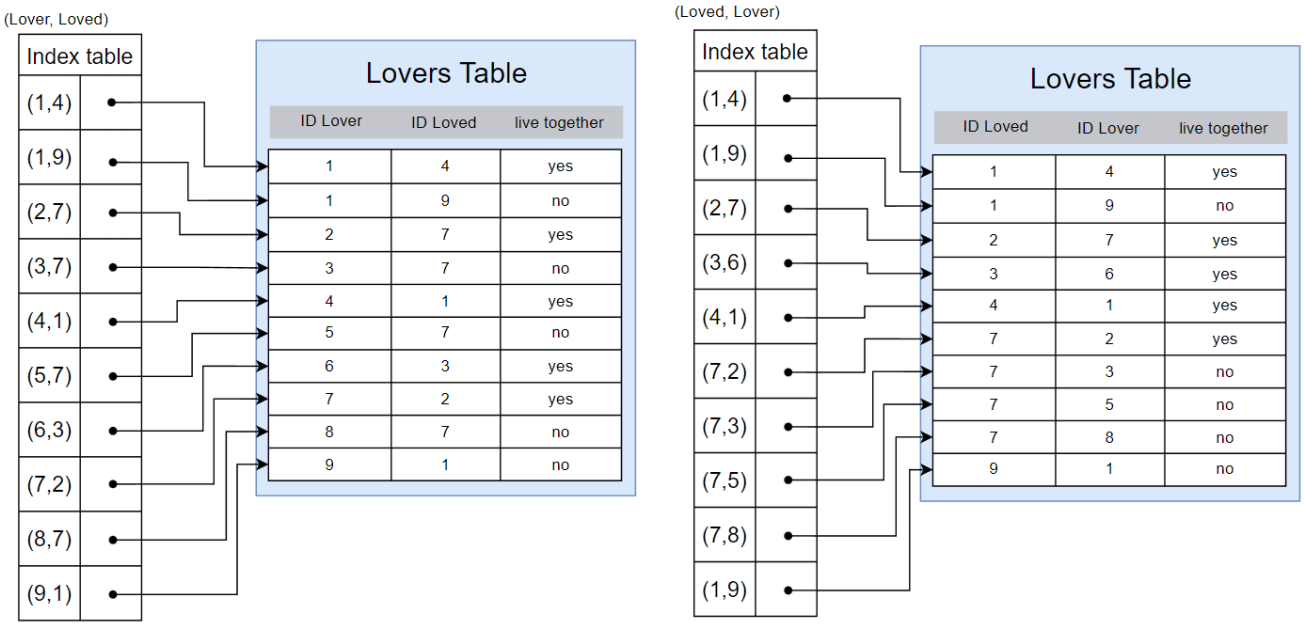


Figure 2.10: Primary Index on (ID Loved, ID Lover) and on (Id Lover, ID Loved)

In this example we take the same table as in Fig.2.3 but in a specific case where, let's say the person corresponding to the ID 7 is a celebrity and a lot of people are loving him. In this case, the index (ID Loved, ID Lover) brings all the tuples where 7 is loved together, bringing without surprise improvements for queries searching for people loving 7. In another case in which we would have someone with attachment disorder and loving five different people, the (ID Lover, ID Loved) index would bring the most sense. This means the author of the database should keep in mind what his database will be used for and what general aspect the database will have. To help the author have a better understanding in which index use, we will address the occurrences in which an index will bring improvements or not in the next section.

2.3.5 Occurrences Of Improvements From Indexes

Indexes are useful to improve the query time in certain situations but not in every situation. Indeed there is only some restricted cases in which a given index will enhance the performances. This means that the choice of which index choose is important and a real subject to analyze[24].

As seen above, the indexes are implemented on a sequence of attributes that will determine the ordering of the tuples. This means that, if you have a multi-attribute index, and if you condition clause addresses only one attribute that is not the first of the index attributes, then the index will bring far lower improvement that if the condition attribute was the first of the index attributes.

This gives an idea of how some index may improve better the query than some other. In practice, there is the main three key factors that will be conditions to have an impacting index on a single table database [32] [10] :

- When equality conditions involve attributes forming a prefix of the index attributes.
- When an inequality condition involves the index's first attribute.
- When the SELECT clause features an aggregate function (SUM, AVG, MAX, and MIN) on the first index attribute.

It is also important when selecting the index attributes order to take into account the sparsity of the values for each attribute [4]. Indeed, when indexing the attributes ('a', 'b'), the indexing will bring way more improvements if the attribute 'a' is very sparsely distributed with a few possible values and the attribute 'b' is very distributed with a lot of possible values. If the opposite occures, you will prefer indexing on ('b', 'a') instead.

Chapter 3

Experiments

3.1 Datasets

3.1.1 Default Of Credit Card Clients[44]

This dataset was found on the `UCI Machine Learning Repositery`[1] site. It was used as a one-table dataset to predict the probability of default for customers payments in Taiwan. It compares the accuracy of six data mining methods and introduces the 'Sorting Smoothing Method' to estimate the real probability of default. The results show that the artificial neural network model outperforms the other techniques, accurately estimating the real probability of default. This finding highlights the importance of using predictive accuracy in risk management rather than simple binary classification.

This dataset was chosen because even if it was a one-table dataset, some columns could be turned into other tables : the dataset was initially composed of the following columns :

- the first five columns that contain information avout the clients, respectively the amount of the credit they were given, their gender, their level of education, their marital status and their age.
- the 6th to 11th columns contain their repayment status in the last six month, first column being the current month, second column the last month third the M-2 month etc.
- the 12th to 17th contain the amount of their bill statement for the last six month, same order as the preceding one.
- the 18th to 23th contain the amount of their payment statement for the last six month, again with the same storing method.
- finally the last column is a binary column about the default payment situation of the client.

This is interesting as we are looking for multi-table database to test our indexes on. Indeed, this dataset is easy to cut into four tables : the first one with the informations on the

[1] `https://archive.ics.uci.edu/ml/index.php`

clients with the first five columns and the last one, the second table about the repayment status of the clients with the informations of the 6th to 11th columns, the third table about the bills of the clients with the informations of the 12th to 17th columns and finally the table about the payments statements of the clients with the informations of the 18th to 23th columns.

The relational schema is represented on Fig.3.1

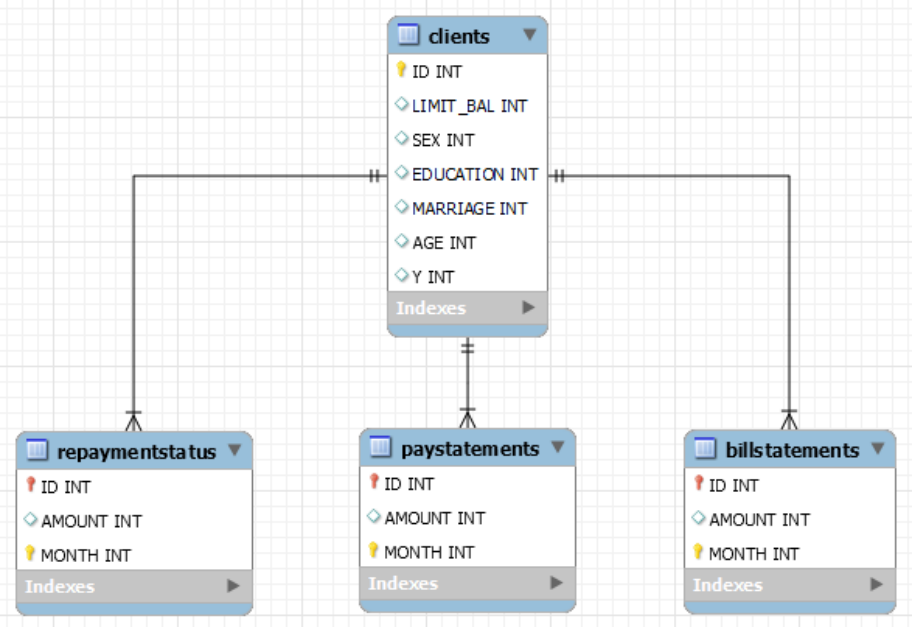


Figure 3.1: Relational schema of the Default Of Credit Card Clients database

As you can see, for the 3 other tables than clients, the informations have been regrouped in only three columns. The ID of the client the tuples is referring to, the month it refers and the amount (or the value that was initially in the excel). This means we have six times more tuples in those tables than in clients but each tuple is lighter.

Finally, one information it is worth pointing out is that every value in our tables is stocked as an integer and thus, the informations that will be analyzed below may be accurate only for such data type (even if it shouldn't be).

3.1.2 Bank Marketing Data Set[31]

This dataset is also from the `UCI Machine Learning Repositery`[2] site. This dataset is also in the form of a one-table dataset but this one doesn't have an architecture such that we could divide it into multiple tables. The queries will then only be applied on the one table database, allowing us to compare their behavior on one-table or multiple table databases.

The dataset itself is a dataset of 45211 features with 20 attributes. It is originally from a study for a direct marketing campaigns of a Portuguese banking institution. The marketing campaigns involved phone calls, and multiple contacts were made to the same client to determine if they would subscribe to a bank term deposit ('yes') or not ('no').

[2] `https://archive.ics.uci.edu/ml/index.php`

The dataset includes the following input variables:

- Age: The age of the bank client (numeric).
- Job: The type of job the client has, categorized as various job titles (categorical).
- Marital Status: The marital status of the client, including categories such as divorced, married, single, or unknown (categorical).
- Education: The educational background of the client, including categories such as basic education levels, university degrees, or unknown (categorical).
- Default: Indicates if the client has any credit in default, with categories of no, yes, or unknown (categorical).
- Housing: Indicates if the client has a housing loan, with categories of no, yes, or unknown (categorical).
- Loan: Indicates if the client has a personal loan, with categories of no, yes, or unknown (categorical).
- Contact: The communication type used for the last contact, categorized as cellular or telephone (categorical).
- Month: The month of the last contact, represented by abbreviations such as jan, feb, mar, and so on (categorical).
- Day of Week: The day of the week of the last contact, categorized as mon, tue, wed, thu, or fri (categorical).
- Duration: The duration of the last contact in seconds (numeric). Note that this attribute strongly influences the output target, but its value is only known after the call is made.
- Campaign: The number of contacts performed during the campaign for a particular client, including the last contact (numeric).
- Pdays: The number of days that passed since the client was last contacted in a previous campaign (numeric). A value of 999 indicates that the client was not previously contacted.
- Previous: The number of contacts performed before the current campaign for a particular client (numeric).
- Poutcome: The outcome of the previous marketing campaign, with categories of failure, nonexistent, or success (categorical).
- Emp.var.rate: The employment variation rate, representing a quarterly indicator (numeric).
- Cons.price.idx: The consumer price index, representing a monthly indicator (numeric).

3.2 Methodology

All of the experiments and query time computing discussed below were tested on the databases stored in mySQL in a local instance. The tables were created in the `MySQL Workbench` program . The data from the csv files were filled and the querying times were computed via a python script that you can find in the Annexe A.

3.3 Results

3.3.1 Default in credit cards

The following subsections of the section 3.3.1 contain the querying time for the SQL queries above (in seconds) :

- `SELECT c.*, rs.AMOUNT FROM clients AS c INNER JOIN repaymentstatus AS rs ON c.ID = rs.ID AND rs.MONTH = 0`
- `SELECT c.ID, SUM(bs.AMOUNT) AS total_bill_amount FROM clients AS c LEFT JOIN billstatements AS bs ON c.ID = bs.ID GROUP BY c.ID`
- `SELECT c.* FROM clients AS c WHERE c.MARRIAGE = 1 AND c.EDUCATION = 1`
- `SELECT ps.MONTH, AVG(ps.AMOUNT) AS average_payment_amount FROM paystatements AS ps GROUP BY ps.MONTH`
- `SELECT c.*, bs.AMOUNT AS bill_amount, ps.AMOUNT AS payment_amount FROM clients AS c LEFT JOIN billstatements AS bs ON c.ID = bs.ID LEFT JOIN paystatements AS ps ON c.ID = ps.ID AND bs.MONTH = ps.MONTH WHERE bs.MONTH = 2`
- `SELECT c.* FROM clients AS c WHERE c.SEX = '1' AND c.LIMIT_BAL > 500000`
- `SELECT c.*, bs.AMOUNT AS bill_amount FROM clients AS c INNER JOIN billstatements AS bs ON c.ID = bs.ID WHERE bs.AMOUNT > (SELECT AVG(AMOUNT) FROM billstatements)`
- `SELECT c.EDUCATION, COUNT(*) AS client_count FROM clients AS c GROUP BY c.EDUCATION`
- `SELECT c.* FROM clients AS c INNER JOIN paystatements AS ps ON c.ID = ps.ID AND ps.MONTH = 4 WHERE c.AGE BETWEEN 25 AND 35`
- `SELECT c.*, bs.AMOUNT AS bill_amount, rs.AMOUNT AS repayment_amount FROM clients AS c LEFT JOIN billstatements AS bs ON c.ID = bs.ID AND bs.MONTH = 5 LEFT JOIN repaymentstatus AS rs ON c.ID = rs.ID AND rs.MONTH = 5 ORDER BY bs.AMOUNT DESC`
- `SELECT c.ID, SUM(ps.AMOUNT) AS total_payment_amount FROM clients AS c INNER JOIN paystatements AS ps ON c.ID = ps.ID GROUP BY c.ID`

The results are printed such that the each SQL query matches with its corresponding column. The first ten rows are each an occurrence of the query run from the python script. The last row is the most interesting one as it contains the means of the ten occurrences.

3.3.1.1 Without Any Index

The first step of our analyze is to compute the querying time on the database with no index at all. It will serve to analyze the positive impact indexes may bring afterward (removing indexes required to remove foregin keys aswell because the software would not let a foreign key refer to a non indexed attribute).

0.172	0.483	0.023	0.168	0.375	0.023	0.346	0.024	0.132	0.518	0.296
0.174	0.473	0.024	0.141	0.409	0.022	0.343	0.023	0.139	0.53	0.31
0.168	0.449	0.021	0.149	0.383	0.021	0.356	0.023	0.136	0.495	0.284
0.166	0.441	0.022	0.143	0.369	0.022	0.33	0.023	0.135	0.503	0.282
0.163	0.443	0.023	0.143	0.362	0.02	0.337	0.022	0.13	0.503	0.286
0.156	0.447	0.022	0.144	0.361	0.021	0.332	0.023	0.135	0.517	0.299
0.172	0.448	0.022	0.167	0.392	0.023	0.335	0.026	0.143	0.5	0.275
0.163	0.457	0.023	0.168	0.4	0.022	0.351	0.025	0.142	0.499	0.323
0.17	0.473	0.022	0.146	0.362	0.02	0.327	0.022	0.136	0.489	0.279
0.164	0.465	0.021	0.151	0.396	0.021	0.334	0.021	0.133	0.481	0.274
0.167	**0.458**	**0.023**	**0.152**	**0.381**	**0.022**	**0.339**	**0.023**	**0.136**	**0.503**	**0.291**

Table 3.1: Query times without any index [sec]

3.3.1.2 Hashtable Primary Index

The analyze of the performances for our SQL queries is not really relevant for such hashtable indexes as they would not use the same engine in MySQL (see section 4.1.1 for more informations).

0.234	0.594	0.094	0.562	0.359	0.078	1.067	0.078	0.109	0.391	0.662
0.219	0.562	0.078	0.547	0.344	0.094	1.079	0.078	0.109	0.414	0.71
0.234	0.6	0.084	0.58	0.364	0.08	1.113	0.078	0.114	0.501	0.702
0.233	0.602	0.089	0.62	0.357	0.082	1.149	0.091	0.107	0.418	0.666
0.219	0.574	0.093	0.536	0.344	0.079	1.062	0.095	0.102	0.383	0.656
0.203	0.578	0.078	0.531	0.344	0.094	1.082	0.094	0.109	0.422	0.661
0.219	0.578	0.078	0.531	0.344	0.078	1.047	0.078	0.109	0.391	0.64
0.219	0.562	0.094	0.533	0.344	0.078	1.051	0.094	0.094	0.391	0.64
0.219	0.567	0.078	0.531	0.344	0.078	1.055	0.094	0.094	0.391	0.64
0.219	0.562	0.094	0.531	0.344	0.078	1.047	0.094	0.094	0.392	0.651
0.222	**0.578**	**0.086**	**0.55**	**0.349**	**0.082**	**1.075**	**0.087**	**0.104**	**0.409**	**0.663**

Table 3.2: Query times with primary index (hashtable) [sec]

The fact that we cannot use the InnoDB engine for such hashtable indexes brings generally clearly worse results. However we can see that, for the queries 5, 9 and 10, the query time is reduces and the index will still bring better results than without the index but with a faster engine. It is because all three queries have a JOIN clause on the ID attributes, which are the ones concerned by the primary indexes.

Still it is not really a correct to comparison, if we want it to make sens, we should compare these values to the results without index but with the Isam engine(issam without index would give similar values than InnoDB without index but slightly inflated). It is not done here as what has our interest is comparing our hashtable indexes values with the B-tree indexes, made impossible as both engine only supports one of the index method.

3.3.1.3 B-Tree Primary Index

The B-tree primary index brings such results :

0.104	0.152	0.014	0.093	0.195	0.013	0.199	0.015	0.03	0.136	0.154
0.056	0.15	0.014	0.09	0.085	0.012	0.195	0.014	0.031	0.135	0.162
0.059	0.159	0.015	0.094	0.091	0.014	0.204	0.015	0.03	0.13	0.154
0.055	0.156	0.014	0.092	0.091	0.013	0.203	0.016	0.03	0.137	0.15
0.054	0.161	0.015	0.085	0.087	0.013	0.198	0.014	0.029	0.137	0.154
0.052	0.148	0.013	0.09	0.088	0.013	0.2	0.015	0.029	0.133	0.147
0.054	0.156	0.014	0.09	0.092	0.013	0.194	0.014	0.029	0.136	0.154
0.055	0.156	0.014	0.093	0.091	0.013	0.205	0.014	0.029	0.141	0.15
0.054	0.153	0.014	0.09	0.093	0.012	0.206	0.016	0.03	0.139	0.158
0.053	0.155	0.014	0.087	0.089	0.013	0.207	0.015	0.031	0.139	0.15
0.059	**0.155**	**0.014**	**0.09**	**0.1**	**0.013**	**0.201**	**0.014**	**0.03**	**0.136**	**0.153**

Table 3.3: Query times on primary index under the form of B-tree [sec]

Which brings better results than using no index or hashtable index in literally every cases. Again, the comparison with hashtable index is not really correct, which prevents us from having hasty conclusions. We will have to restrain on comparing the InnoDB results with each other.

3.3.1.4 Secondary Index on Education

This time, let us compare the results of using primary indexes on our database on the ID's) and adding a secondary index on the attribute 'Education' on the 'clients' table :

0.062	0.625	0.016	0.094	0.271	0.011	1.09	0.009	0.042	0.244	0.179
0.048	0.591	0.017	0.083	0.267	0.0	1.077	0.004	0.047	0.234	0.149
0.047	0.602	0.0	0.086	0.277	0.016	1.052	0.008	0.029	0.237	0.147
0.033	0.601	0.016	0.089	0.26	0.016	1.072	0.002	0.047	0.251	0.151
0.048	0.582	0.016	0.085	0.263	0.016	1.045	0.0	0.047	0.228	0.15
0.031	0.609	0.015	0.083	0.269	0.012	1.068	0.001	0.048	0.242	0.127
0.065	0.601	0.015	0.084	0.265	0.016	1.038	0.016	0.041	0.26	0.151
0.047	0.603	0.001	0.08	0.28	0.0	1.06	0.0	0.047	0.244	0.133
0.053	0.601	0.016	0.084	0.266	0.016	1.064	0.0	0.05	0.228	0.144
0.05	0.589	0.016	0.078	0.262	0.027	1.056	0.008	0.043	0.262	0.141
0.048	**0.6**	**0.013**	**0.085**	**0.268**	**0.013**	**1.062**	**0.005**	**0.044**	**0.243**	**0.147**

Table 3.4: Query times on B-tree primary and second index on 'Education' [sec]

The attribute 'EDUCATION' only appears in the third and eighth queries. The third query time is still relatively even with and without the secondary index. the eighth however, got reduced by nearly half. The main differences is that the third query only uses the 'Education' attribute in a really specific equality when the eighth uses it in a GROUP BY clause. More details about cases of improvement due to indexes will be given in the section 3.3.2.

Another thing to note is that the secondary index also impacted other queries, and it tends to increase the query time (the cases of improvement are relatively close but the cases of worsening are way more consequent). It shows that one should be cautious when adding secondary index because it can cost more than it brings to add indexes on attribute you will not use that much.

3.3.1.5 Secondary Indexes on Age

Let us now compare when adding to our primary index a secondary index on the 'Age' attribute instead of the 'Education' attribute.

0.047	0.594	0.016	0.078	0.26	0.016	1.124	0.017	0.05	0.24	0.141
0.047	0.617	0.012	0.077	0.279	0.011	1.096	0.013	0.043	0.238	0.141
0.031	0.61	0.002	0.083	0.285	0.001	1.099	0.016	0.034	0.25	0.132
0.047	0.601	0.016	0.086	0.265	0.016	1.073	0.016	0.033	0.249	0.15
0.05	0.618	0.017	0.083	0.266	0.017	1.067	0.017	0.048	0.234	0.147
0.047	0.605	0.016	0.083	0.259	0.016	1.087	0.0	0.047	0.245	0.134
0.056	0.59	0.016	0.096	0.255	0.016	1.081	0.016	0.04	0.233	0.15
0.049	0.617	0.018	0.082	0.306	0.012	1.074	0.016	0.031	0.234	0.157
0.035	0.62	0.016	0.089	0.265	0.012	1.082	0.013	0.04	0.247	0.143
0.04	0.616	0.013	0.083	0.271	0.002	1.091	0.014	0.027	0.235	0.141
0.045	**0.609**	**0.014**	**0.084**	**0.271**	**0.012**	**1.088**	**0.014**	**0.039**	**0.241**	**0.143**

Table 3.5: Query times on B-tree primary +second index on 'AGE' [sec]

The attribute 'AGE' is only used in the tenth query as a really specific condition again. The results are pretty similar to the ones obtained when the secondary index was about the

'Education' attribute, except for the eighth query where the 'Education' attribute was used in a GROUP BY clause. This shows that even though the secondary attribute may have a good impact when used properly, such index may turn out to be pretty useless when poorly used (as in these really specific condition clauses).

3.3.2 Bank

With this dataset, we will try to determine more precisely the conditions in which the secondary indexes will bring improvements. The indexes from now on will be secondary B-tree indexes as we will use InnoDB as engine.

3.3.2.1 Order Of The Index Attributes In The Query

In the theory section 2, we addressed the fact that an index helps as long as the condition attribute are a prefix of the index attributes. In this section we will analyse if all the indexes respecting this condition help equally.

Let's first measure the querying times when using multi-attribute index and trying to swap the order of the attributes either in the query condition clause, either in the index.

We input these two queries :

- SELECT * FROM bank WHERE job = 'technician' AND marital = 'married';
- SELECT * FROM bank WHERE marital = 'married' AND job = 'technician';

which are basically the same query but both conditions have been swapped. We obtain the queries times without index displayed in Table 3.6 the first sub-table contains the results for the first query and the second sub-table contains the results for the second query, both being displayed as twenty iterations of the query and their mean on the right of the double line.

0.062	0.062	0.062	0.062	0.062	0.062	0.047	0.047	0.062	0.062	
0.062	0.062	0.062	0.047	0.047	0.047	0.047	0.062	0.047	0.047	0.056
0.047	0.047	0.047	0.047	0.047	0.062	0.062	0.047	0.047	0.047	
0.047	0.047	0.047	0.062	0.062	0.062	0.047	0.047	0.062	0.062	0.052

Table 3.6: With no index [sec]

Both queries either have a query time of 0.047 sec or 0.062 sec. This most probably results from the DMBS cache : either the query is planned by the DBMS to be requested next and then will be kept in the cache memory either it is not, resulting in the DMBS having to search the results from scratch. This, however, is nothing but a supposition that would need further analyze, which will not be conducted here as it is not the goal of this paper. Our query times comparisons will still be possible by comparing the high times and low times of the different instances, every instance apparently having such binary values.

In this precise case, although the mean time of the first query is higher, it lead us to the conclusion that the order of the attributes in the condition clause does not affect the query times, both queries having the same high and low query time. The mean time being fluctuating at each iteration in practice, depending of the number of cache miss only.

Let us now see if the attribute order in the index has an influence : in the tables 3.7 and 3.8 we listed the results of both queries for respectively an index ('job','marital') and a index ('marital','job').

0.031	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	
0.016	0.016	0.016	0.031	0.016	0.016	0.016	0.016	0.016	0.016	0.017
0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	
0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016

Table 3.7: Index ('job','marital') [sec]

0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	
0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016
0.016	0.031	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	
0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016

Table 3.8: Index ('marital','job') [sec]

This experiment shows no sign of a diversity between the permutation of the attributes in the index or in the condition.

3.3.2.2 Number Of Matching Attributes

Let's now interest ourselves in the comparison of different number of matching attribute : in the table 3.9, we computed the results with the index ('education',housing') and the two following queries :

- "SELECT * FROM bank WHERE education = 'university.degree' AND housing = 'no';"
- "SELECT * FROM bank WHERE marital = 'single' AND education = 'high.school';"

As you may see, the goal is to compare when both attributes of the condition ('education', 'housing') are part of the index and when only one of the two query attributes ('education', 'marital') is in the index.

0.016	0.016	0.031	0.023	0.023	0.023	0.023	0.022	0.023	0.023	
0.022	0.022	0.022	0.016	0.016	0.031	0.031	0.016	0.016	0.016	0.021
0.031	0.031	0.03	0.029	0.028	0.029	0.029	0.029	0.028	0.029	
0.028	0.036	0.025	0.031	0.031	0.016	0.031	0.031	0.031	0.031	0.029

Table 3.9: Index ('education','housing') [sec]

As expected, the first request with both matching attribute performs better. However, the question that remains is if there is an improvement when only one condition attribute is matching or if the query with no index at all performs better. To analyze it we will query only the second query without any index to compare :

- "SELECT * FROM bank WHERE marital = 'single' AND education = 'high.school';"

0.031	0.031	0.031	0.016	0.031	0.031	0.016	0.031	0.031	0.016	
0.031	0.031	0.016	0.031	0.031	0.031	0.031	0.016	0.031	0.031	0.027

Table 3.10: Index ('education') [sec]

We can see that it brings better query time to perform a query on a table with no index at all (as proves the figures of table 3.11 than on a table with an index matching only a subset of the condition attributes. Even if the matching attribute is the first of the index attributes.

0.047	0.062	0.047	0.062	0.047	0.047	0.062	0.047	0.047	0.062	
0.047	0.062	0.047	0.047	0.062	0.047	0.062	0.047	0.047	0.062	0.053

Table 3.11: Pas d'index [sec]

This time let us analyze the impact of an index that is too precise compared to what is requested. On tables 3.12 and 3.13 are the querying times for the same query as above :

- "SELECT * FROM bank WHERE marital = 'single' AND education = 'high.school';"

The first table is the query time requested on a table with an index that perfectly match the query attributes ('education', 'marital') and the second table is about the queries on a table with an index ('education', 'marital', 'duration', 'loan') that is too precise : the attributes of the query are all in the index but the index contains even more attributes than the ones required.

0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.0	
0.016	0.016	0.016	0.016	0.016	0.0	0.016	0.016	0.016	0.016	0.014

Table 3.12: Index ('education','marital') [sec]

0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.003	
0.026	0.015	0.014	0.012	0.016	0.016	0.0	0.016	0.016	0.016	0.014

Table 3.13: Index ('education','marital','duration', 'loan') [sec]

These tables show that using indexes that containing more attributes than needed has no negative impact on the querying times. Together with the results of the index only partially

matching the query attributes, these results show that one should favor the use of indexes a little bit too much attributes than being a little sparing.

This doesn't mean one should put every attributes in his index without thinking because when you add the unused attributes in front of the used attributes, you may considerably increase the duration of the queries, as shown with the table 3.14 informing of the query times with an index ('duration', 'loan', 'marital', 'education').

0.047	0.062	0.047	0.047	0.062	0.047	0.047	0.047	0.062	0.047	
0.062	0.047	0.047	0.062	0.047	0.047	0.062	0.047	0.047	0.047	0.052

Table 3.14: Index ('duration', 'loan', 'marital', 'education') [sec]

3.3.2.3 Matching Patterns

When querying, you can put conditions clauses for an attribute to match a certain pattern. It is mostly used with the LIKE clause : a clause that allows to use the joker symbol '%' to express any value of any length, just like the '*' in shell and many others. So for an example, if you want your selection to contain only monthes that start with m, you can add a condition clause of the form "WHERE month LIKE 'm%' ". Other pattern matching expression exist like 'ALIKE' or 'REGEXP' but they are mainly adaptations of the 'LIKE' expression and won't be discussed here.

Let us analyze what results the indexes bring for such special condition clauses with two requests, one using a 'LIKE' expression to match with a joker symbol at the start and one at the end of the attributes:

- "SELECT * FROM bank WHERE education LIKE '%school';"
- "SELECT * FROM bank WHERE education LIKE 'high%';"

0.078	0.063	0.062	0.062	0.078	0.062	0.062	0.078	0.062	0.078	
0.069	0.062	0.08	0.063	0.062	0.078	0.062	0.062	0.078	0.062	0.068
0.062	0.078	0.062	0.062	0.062	0.062	0.062	0.062	0.062	0.062	
0.062	0.109	0.065	0.062	0.062	0.062	0.062	0.062	0.062	0.062	0.066

Table 3.15: Index ('education') [sec]

Now let us compare when we add an index on the 'education' attribute :

0.062	0.062	0.062	0.078	0.062	0.062	0.078	0.062	0.062	0.084	
0.07	0.068	0.073	0.074	0.071	0.07	0.069	0.068	0.069	0.07	0.069
0.078	0.062	0.062	0.062	0.062	0.062	0.062	0.078	0.062	0.065	
0.065	0.068	0.07	0.07	0.065	0.065	0.065	0.064	0.065	0.066	0.066

Table 3.16: Pas d'index [sec]

We observe that the indexes won't bring any improvement either when the pattern starts the value or not. Such pattern matching queries are to be avoided if possible as they disable the use of indexes.

3.3.2.4 Inequality

Let us now try with an inequality clause :

- "SELECT * FROM bank WHERE duration > 300;"

0.078	0.062	0.062	0.062	0.062	0.062	0.062	0.062	0.062	0.062	
0.062	0.062	0.062	0.062	0.078	0.062	0.062	0.062	0.062	0.062	0.064

Table 3.17: Pas d'index [sec]

0.062	0.062	0.062	0.062	0.062	0.062	0.062	0.078	0.066	0.066	
0.066	0.066	0.066	0.067	0.066	0.068	0.061	0.062	0.062	0.062	0.065

Table 3.18: Index ('duration') ASC [sec]

We can see that there is no difference when adding the index on 'duration' even though the inequality attribute appears in the index and the theory in section 2 said it would work.

To discover why such index brought no improvement to the query, we started by looking if a descending (DESC) index instead of ascending (ASC) would change anything, maybe ascending would work for < and descending for >.

0.07	0.062	0.063	0.062	0.078	0.062	0.062	0.062	0.062	0.062	
0.062	0.062	0.078	0.062	0.062	0.062	0.062	0.062	0.062	0.062	0.064

Table 3.19: Index ('duration') DESC [sec]

This shows no changes from the ASC index.

When asked about such weird results, chatGPT replied (among other) this : "[...] However, it is important to note that the use of an index can vary depending on the cardinality of the data. If the inequality condition selects a large portion of the table, it is possible that the query optimizer may choose not to use the index and instead opt for a full table scan. [...]"

And to check this, the following query, which is the same as before but with 3000 instead of 300, selecting a really smaller proportion of the tuples (from 11204 tuples to 15) : "SELECT * FROM bank WHERE duration > 3000;"

0.031	0.047	0.047	0.047	0.047	0.047	0.031	0.047	0.047	0.047	
0.047	0.047	0.047	0.031	0.047	0.047	0.047	0.031	0.047	0.047	0.044

Table 3.20: Pas d'Index [sec]

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.016	0.0	0.001

Table 3.21: Index ('duration') ASC [sec]

Finally our index brought improvement.

This shows how volatile index usage can be.

3.3.2.5 Importance Of The Attributes Order In The Index

The following section will help highlight the importance of the order of the index attributes:

Imagine you want to query the following query :

- "SELECT * FROM bank WHERE duration > 500 AND marital = 'single' AND education = 'high.school';"

Then you might want to optimize the index by putting the inequality attribute as the first one in your index :

0.0	0.016	0.0	0.0	0.0	0.016	0.0	0.0	0.0	0.016	
0.0	0.0	0.0	0.016	0.0	0.0	0.0	0.0	0.016	0.0	0.004

Table 3.22: Index ('duration', 'marital', 'education') [sec]

but now, the same query with an index starting with 'marital' and 'education' :

0.0	0.0	0.0	0.0	0.016	0.0	0.0	0.0	0.0	0.0	
0.016	0.0	0.0	0.0	0.0	0.0	0.016	0.0	0.0	0.0	0.002

Table 3.23: Index ('marital', 'education', 'duration') [sec]

The order in which you put the index attributes is very important. The 'marital' attribute in first place improves the query time better than the 'duration'. Indeed, it will help prune a lot more tuples early in the query. But in this case, does the 'duration' attribute really belong to the index attributes ? As it is an inequality attribute and it is no longer the first attribute of the index, it should not. However, in practice, we can observe slightly longer query times when removing it from the index:

0.008	0.009	0.002	0.016	0.0	0.016	0.0	0.016	0.009	0.0	
0.016	0.0	0.016	0.0	0.016	0.016	0.0	0.016	0.0	0.016	0.008

Table 3.24: Index ('marital', 'education') [sec]

But does equality attributes work when an inequality is first?

0.016	0.016	0.0	0.016	0.016	0.016	0.016	0.0	0.016	0.016	
0.016	0.0	0.016	0.016	0.016	0.016	0.0	0.016	0.016	0.016	0.012

Table 3.25: Index ('duration') [sec]

Remember that the condition 'duration' > 300 prunes nearly nothing as it was not even enough to be used in an index in the section above about inequalities. By try trial and error, we find that around 450 we go from not using the indexes at all to using them, improving a lot the query time.
This means that the value 500 for duration will help prune enough tuples to be used as an index but, 500 being close to 450, it will help prune less tuples than putting 'marital' and 'education' as first index attributes (4928 tuples left vs 3150).

This leads us to the conclusion that both indexes ('duration', 'marital', 'education') and ('marital', 'education', 'duration') can be better, depending on the pruning they will bring. ('duration', 'marital', 'education') will be better in the cases where the condition on 'duration' prunes more than the conditions on 'marital' and 'education' and the opposite .
This shows again how easily an index can be ruined by adding to many attributes at bad places : with the condition 'duration' > 300 that would select nearly the entire database, the 'duration' attribute in the first place would ruin the index.

3.3.2.6 MAX And MIN Clauses

The last point in theory pointed out that the index work for Aggregate clauses like MIN and MAX when the maximized attribute is the first of the index attributes. let's compare the following request: one is selecting the tuples with a condition on 'marital' and 'education', the second use the same condition but selects only the maximum for 'age' (which ends up being 60 with such conditions) and the third selects the whole tuples that will correspond to the second request (4 tuples with the 'age' of 60)

- "SELECT * FROM bank WHERE marital = 'single' AND education = 'high.school';"
- "SELECT Max(age) FROM bank WHERE marital = 'single' AND education = 'high.school';"
- "SELECT * FROM bank WHERE marital = 'single' AND education = 'high.school' AND age = '60';"

0.047	0.047	0.062	0.062	0.047	0.047	0.047	0.047	0.062	0.062	
0.053	0.055	0.053	0.052	0.045	0.047	0.062	0.062	0.047	0.047	0.053
0.031	0.016	0.016	0.016	0.031	0.016	0.031	0.016	0.016	0.016	
0.019	0.02	0.02	0.02	0.016	0.031	0.016	0.016	0.016	0.016	0.02
0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.053	
0.048	0.049	0.045	0.045	0.047	0.031	0.047	0.047	0.062	0.047	0.047

Table 3.26: Without index [sec]

0.062	0.047	0.047	0.062	0.047	0.062	0.047	0.047	0.047	0.062	
0.047	0.047	0.062	0.047	0.047	0.062	0.047	0.047	0.062	0.062	0.053
0.016	0.031	0.016	0.016	0.016	0.016	0.031	0.016	0.016	0.016	
0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.017
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.016	0.0	0.0	
0.016	0.0	0.0	0.016	0.0	0.0	0.016	0.0	0.0	0.0	0.003

Table 3.27: Index ('age') [sec]

The index on age seems to improve way more on the third selection than on the maximum one. Now what if we compare them with an index focusing on the conditions first and then the maximized attribute ?

0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	
0.016	0.016	0.0	0.0	0.016	0.016	0.016	0.016	0.016	0.016	0.014
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
0.0	0.0	0.016	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 3.28: Index ('marital','education','age') [sec]

In this case the queries perform better than when the maximized attribute is put first. It appears that querying a maximum with conditions performs better with index on the conditions than on the maximized attribute. However, is it always the case ? In our case we see that the maximum age fulfilling our condition is 60 which is low compared to the maximum without conditions (910 tuples have an 'age' bigger than 60). Even with the tuples sorted by 'age', you can't directly find the ones with the maximum 'age' AND corresponding to the conditions, as they are not placed on a specific spot. The conditions must first be checked and prune the data and then the maximum value appears at a specific spot easy to check (the last places).

0.047	0.047	0.047	0.047	0.047	0.062	0.047	0.047	0.062	0.047	
0.047	0.047	0.062	0.047	0.047	0.062	0.062	0.047	0.047	0.062	0.052
0.031	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.0	0.016	
0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016
0.047	0.063	0.047	0.047	0.047	0.047	0.047	0.047	0.062	0.047	
0.062	0.047	0.047	0.047	0.047	0.031	0.047	0.047	0.047	0.047	0.048

Table 3.29: Index ('housing','age','marital','education') [sec]

With a random additional attribute in front in the index attributes, The results approximately go back to their no index values, which is kind of expected.

To find a case where the index actually improves the result for a maximization, we try with the following query, which a 'Max' selection on 'age' with no other conditions (the maximum being '90' with two tuples at that value) :

- "SELECT Max(age) FROM bank ;"

0.016	0.016	0.0	0.016	0.016	0.016	0.016	0.016	0.0	0.016	
0.016	0.016	0.016	0.0	0.016	0.016	0.016	0.0	0.016	0.016	0.012

Table 3.30: Without Index [sec]

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 3.31: Index ('age') [sec]

It shows pretty convincing results. And further tests give the conclusion that the ascending or descending index seems not to impact the results.

Chapter 4

Discussion

4.1 Limitations

4.1.1 Engine Lack Of Adaptability

The first limitation encountered is the fact that the B-tree index and the hash index can't really be compared properly as it is not possible with a given engine to modify which index method it will use. To compare such index the method used was to use either the InnoDB engine to register the query time with a B-tree index or the Isam engine for the Hash index. This means that it is not only the index method that change when comparing their time. Here are the main differences between the two engine in terms of performance :

- InnoDB is generally faster for write-intensive workloads due to its row-level locking(a mechanism where only specific rows of the table are locked during a transaction, allowing better concurrency and reducing contention among multiple transactions).
- MyISAM may be faster for read-intensive operations on small tables, but performance can degrade on larger tables due to table-level locking.
- InnoDB's use of clustered indexes can provide faster access to data for certain queries.
- InnoDB's crash recovery features may slightly impact write performance compared to MyISAM.
- InnoDB's support for transactions allows for more complex queries but can lead to some overhead compared to MyISAM for simple read-only queries.

These show how impacting the choice of engine is, and not only for the indexing methods.

4.1.2 Binary Timers

Another limitation encountered is the fact that the query times usually return in two distinct values (especially on the Bank database). With no little variations. It seems that these values correspond to a binary condition encountered or not. The first idea we had was that the low value would be when some cache memory stored the result of the query and would be faster to return it than in the cases where it is not stored in such cache, resulting in the high values.

However we happen to have some queries for which the distribution between these high values and low values is not the same as the query they are compared to, even though they have the same high and low values. This results in them having different mean computing time (one example is the comparison between the first and last query of Table 3.26, and there are other cases). And they have different distribution in a consistent way. It is not due to one random iteration, even when repeating the 10 comparisons, we obtain such distributions where one consistently have more higher value and the other has more lower values.

And in other tables, the values can even variate between more than 2 values. In certain case even up to 6 distinct values. In these cases our results are getting closer to an usual distribution. However the difference between the possible values in these case and in two-value cases still show that there is some misunderstanding in what is exactly happening. Unfortunately, we won't have time and resources for a deeper analyze here.

4.2 Future Work

The initial study focused on the bank dataset, covering various scenarios, but there remain unexplored areas. To enhance the research, conducting cross-case analysis involving combinations of equalities and inequalities, as well as exploring specific situations independently (e.g., queries with inequalities and max values, double inequalities, etc.) would yield valuable insights.

Expanding the scope of the study to include datasets with multiple tables presents an interesting opportunity. Analyzing how indexes perform when queries involve joint clauses on index attributes or not, and examining the multi-temporal use of indexes (e.g., utilizing an index for a joint clause and then for a condition clause) could provide a deeper and more comprehensive understanding of index working.

Lastly, investigating the impact of multiple indexes on both single-table and multi-table databases could be enlightening. Currently, the study assesses the contribution of a single index, but considering multiple indexes could reveal additional performance enhancements and trade-offs.

4.3 Conclusion

From this work, it becomes evident that the use of indexes plays a critical role in optimizing query performance, especially when dealing with large databases. Indexes have the potential to significantly enhance query response times, making them indispensable for efficient data retrieval.

However, it is essential to recognize that the effectiveness of indexes is closely tied to their proper utilization. Precision and knowledge are paramount when designing and implementing indexes, as even minor adjustments to their configuration can dramatically impact their efficiency. This underscores the importance of having knowledgeable individuals, such as database administrators or data architects, involved in the decision-making process of index creation and management.

Understanding the nature of queries that will be executed on the database is pivotal for selecting the appropriate indexes. Adapting the indexes to align with the specific types of queries expected to be performed can yield substantial improvements in overall performance. This emphasizes the need for a thorough understanding of both the data and the intended usage of the database to optimize index utilization effectively. This can be done by analyzing at hand the tables but for more complex database, some algorithms exist to help summarize the database [43] [34].

Furthermore, end-users of the database should also be aware of the data's characteristics, including its shape and value distributions. Having such insights empowers users to formulate more efficient queries and make informed decisions when accessing the data, contributing to better overall database performance and user experiences.

In today's data-driven landscape, where businesses and organizations heavily rely on databases to understand the insights and drive decision-making, the proper use of indexes emerges as a crucial aspect of database management. By leveraging indexes strategically and aligning them with query requirements, organizations can unlock the full potential of their databases, delivering faster and more efficient access to valuable information, ultimately enhancing productivity and competitiveness. This can be considered as a good strategical advantage for a company, which, as we have seen our courses during the year, is the first step for the businessman that seeks to make his business flourish.

Bibliography

[1] Disk space management, 17 may 2016. URL http://dbmsfortech.blogspot.com/2016/05/disk-space-management.html. Consulté le 8 aout 2023.

[2] Best practices for destroying hard disk drives: Shredding versus degaussing, 2017. URL https://www.intimus.com/resources/white-papers/best-practices-destroying-hard-disk-drives-shredding-vs-degaussing/. Consulté le 8 aout 2023.

[3] Hard disk drive basics, 2023. URL https://www.file-recovery.com/recovery-hard-disk-drive-basics.htm. Consulté le 8 aout 2023.

[4] P. Ameri. On a self-tuning index recommendation approach for databases. *2016 IEEE 32nd International Conference on Data Engineering Workshops (ICDEW)*, pages 201–205, 2016. URL https://api.semanticscholar.org/CorpusID:9458657.

[5] M. B. Douglas Blansit MPS. The basics of relational databases using mysql. *Journal of Electronic Resources in Medical Libraries*, 3(3):135–148, 2006. doi: 10.1300/J383v03n03_10. URL https://doi.org/10.1300/J383v03n03_10.

[6] A. Bahmani, M. Naghibzadeh, and B. Bahmani. Automatic database normalization and primary key generation. *2008 Canadian Conference on Electrical and Computer Engineering*, pages 000011–000016, 2008. URL https://api.semanticscholar.org/CorpusID:15961468.

[7] R. Bayer. The universal b-tree for multidimensional indexing. 1996. URL https://api.semanticscholar.org/CorpusID:59980991.

[8] A. A. Chikkamannur and S. M. Handigund. Automated methodology to reduce the redundancy in relational database. 2013. URL https://api.semanticscholar.org/CorpusID:235675892.

[9] J.-H. Chu and G. D. Knott. An analysis of b-trees and their variants. *Inf. Syst.*, 14:359–370, 1989. URL https://api.semanticscholar.org/CorpusID:5605137.

[10] R. A. Elmasri and S. B. Navathe. Fundamentals of database systems. 1989. URL https://api.semanticscholar.org/CorpusID:33048093.

[11] P. Ferragina and R. Grossi. The string b-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46:236–280, 1999. URL https://api.semanticscholar.org/CorpusID:3352480.

[12] J. Gim and Y. Won. Extract and infer quickly: Obtaining sector geometry of modern hard disk drives. *ACM Trans. Storage*, 6:6:1–6:26, 2010. URL https://api.semanticscholar.org/CorpusID:10642651.

[13] P. Guagliardo and L. Libkin. A Formal Semantics of SQL Queries, Its Validation, and Applications. *Proceedings of the VLDB Endowment*, 11:27–39. doi: 10.14778/3151113.3151116.

[14] B. Ha, H. jin Cho, and Y. I. Eom. A study on the block fragmentation problem of ssd based on nand flash memory. In *International Conference on Ubiquitous Information Management and Communication*, 2011. URL https://api.semanticscholar.org/CorpusID:14886289.

[15] J. Hao, X. Chen, Y. Qiao, Y. Zhang, and T. Zhang. On the design of smr hdd block device driver. *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1291–1299, 2020. URL https://api.semanticscholar.org/CorpusID:233434961.

[16] M. Hayati and A. Setyanto. Index effect on data manipulation toward database performance. *Journal of Physics: Conference Series*, 1140(1):012036, dec 2018. doi: 10.1088/1742-6596/1140/1/012036. URL https://dx.doi.org/10.1088/1742-6596/1140/1/012036.

[17] J. A. Hoffer. A survey of primary and secondary keys through a case study. *Inf. Manag.*, 2:99–106, 1979. URL https://api.semanticscholar.org/CorpusID:30993905.

[18] M. S. Hosain and M. A. H. Newton. Multi-key index for distributed database system. *Int. J. Softw. Eng. Knowl. Eng.*, 15:433–438, 2005. URL https://api.semanticscholar.org/CorpusID:18367325.

[19] M. Iqbal, S. H. A. Kazmi, A. Manzoor, A. R. Soomrani, S. H. Butt, and K. A. Shaikh. A study of big data for business growth in smes: Opportunities challenges. In *2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*, pages 1–7, 2018. doi: 10.1109/ICOMET.2018.8346368.

[20] M. A. Islam and M. E. Rana. Utilization of big data with cloud computing in modern business environment: A review. In *2022 16th International Conference on Ubiquitous Information Management and Communication (IMCOM)*, pages 1–7, 2022. doi: 10.1109/IMCOM53663.2022.9721778.

[21] K. Kalajdzic and A. Patel. A fast scheme for recovery of deleted files with evidential recording for digital forensics. 08 2023.

[22] W. Kim, D. S. Reiner, and D. S. Batory, editors. *Query Processing in Database Systems*. Topics in Information Systems. Springer Berlin, Heidelberg, 1 edition. doi: 10.1007/978-3-642-82375-6.

[23] H. Köhler, U. Leck, S. Link, and X. Zhou. Possible and certain keys for sql. *The VLDB Journal*, 25:571–596, 2016. URL https://api.semanticscholar.org/CorpusID:16108707.

[24] M. Kormilitsin, R. Y. Chirkova, Y. Fathi, and M. F. Stallmann. View and index selection for query-performance improvement: quality-centered algorithms and heuristics. In *International Conference on Information and Knowledge Management*, 2008. URL https://api.semanticscholar.org/CorpusID:14386313.

[25] M. Kvet. Database block management using master index. *2022 32nd Conference of Open Innovations Association (FRUCT)*, pages 135–142, 2022. URL https://api.semanticscholar.org/CorpusID:254102522.

[26] K. P. Lafler. Joining tables of data in the sql procedure. 1994. URL https://api.semanticscholar.org/CorpusID:59324374.

[27] V. B. T. Le, S. Link, and M. Memari. Schema- and data-driven discovery of sql keys. *J. Comput. Sci. Eng.*, 6:193–206, 2012. URL https://api.semanticscholar.org/CorpusID:2122064.

[28] V. B. T. Le, S. Link, and M. Memari. Discovery of keys from sql tables. In *International Conference on Database Systems for Advanced Applications*, 2012. URL https://api.semanticscholar.org/CorpusID:17039891.

[29] S. Lightstone and B. Bhattacharjee. Automated design of multidimensional clustering tables for relational databases. In *Very Large Data Bases Conference*, 2004. URL https://api.semanticscholar.org/CorpusID:9456502.

[30] H. Lu, H. C. Chan, and K. K. Wei. A Survey on Usage of SQL. *ACM SIGMOD Record*, 22 (4):60–65, December 1993. doi: 10.1145/166635.166656. URL https://doi.org/10.1145/166635.166656.

[31] S. Moro, P. Cortez, and P. Rita. A data-driven approach to predict the success of bank telemarketing. *Decision Support Systems*, 62:22–31, June 2014. doi: 10.1016/j.dss.2014.02.003.

[32] S. Nijssen. Linfo2172: Databases, 2023. Course materials, UCLouvain.

[33] C.-W. Park and C.-W. Chung. An effective query pruning technique for multiple regular path expressions. *J. Syst. Softw.*, 64:219–233, 2002. URL https://api.semanticscholar.org/CorpusID:36261983.

[34] M. Piattini, C. Calero, and M. Genero. Table oriented metrics for relational databases. *Software Quality Journal*, 9(2):79–97, June 2001. doi: 10.1023/A:1016670717863. URL https://doi.org/10.1023/A:1016670717863.

[35] Y. Qi. Researching effect of index on query cost. *Journal of South China Normal University*, 2004. URL https://api.semanticscholar.org/CorpusID:156538617.

[36] C. Ritchie. Relational database principles. 1998. URL https://api.semanticscholar.org/CorpusID:53902058.

[37] N. Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7: 258–290, 1982. URL https://api.semanticscholar.org/CorpusID:16747847.

[38] T. K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Inf. Syst.*, 13:175–185, 1988. URL https://api.semanticscholar.org/CorpusID:38650409.

[39] B. Shneiderman. Recuced combined indexes for efficient multiple attribute retrieval. *Inf. Syst.*, 2:149–154, 1977. URL https://api.semanticscholar.org/CorpusID:45919714.

[40] Y. N. Silva, I. Almeida, and M. Queiroz. SQL: From Traditional Databases to Big Data. In *SIGCSE '16: Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 413–418, February 2016. doi: 10.1145/2839509.2844560. URL https://doi.org/10.1145/2839509.2844560.

[41] M. Stonebraker. A comparison of the use of links and secondary indices in a relational data base system. In *International Conference on Software Engineering*, 1976. URL https://api.semanticscholar.org/CorpusID:2086952.

[42] M. Wahyudi, V. Meilinda, and A. Khoirunisa. The digital economy's use of big data. *International Transactions on Artificial Intelligence*, 1(1), November 2022. doi: 10.33050/italic.v1i1.167. URL https://doi.org/10.33050/italic.v1i1.167.

[43] X. Yang, C. M. Procopiuc, and D. Srivastava. Summarizing relational databases. 2(1): 634–645, aug 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687699. URL https://doi.org/10.14778/1687627.1687699.

[44] I.-C. Yeh and C.-H. Lien. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Expert Systems with Applications*, 36(2):2473–2480, 2009.

Appendix A

python script

The python script will be divided into multiple parts according to their goals. The code may have had been adapted for specific queries during the testing but is mainly the same as presented below.

A.1 Filling the databases

The first script will be the one used to connect to the database server and to set the parameters

```
# ================================================================================
# Creating the DB
# ================================================================================


table = pd.read_csv("DefaultCC.csv",delimiter = ";", header =1)
table.rename(columns = {'default payment next month':'Y'}, inplace = True)


clients = table[['ID', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', 'Y']]
history = table[['ID','PAY_0','PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6']]
billsamounts = table[['ID','BILL_AMT1', 'BILL_AMT2', 'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6']]
payamounts = table[['ID','PAY_AMT1', 'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6']]




# Replace 'your_database' with the name of your MySQL database
database_name = 'your_database'
# Replace 'your_table' with the name of the table you want to create in MySQL
table_name = 'your_table'
# Replace 'username' and 'password' with your MySQL credentials
username = 'username'
password = 'password'

# Connect to the MySQL database
cnx = mysql.connector.connect(user=username, password=password, host='localhost', database=database_name)
cursor = cnx.cursor()
```

The four next will serve to fill the database created manually in the MySQL Workbench :

```
# Insert the DataFrame into MySQL                    BILL STATEMENTS
for _, row in billsamounts.iterrows():

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']),  int(row['BILL_AMT1']), 0)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['BILL_AMT2']), 2)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['BILL_AMT3']), 3)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['BILL_AMT4']), 4)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['BILL_AMT5']), 5)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['BILL_AMT6']), 6)
    cursor.execute(insert_query, values)
```

```
# Insert clients the DataFrame into MySQL                CLIENTS
for _, row in clients.iterrows():
    insert_query = f"INSERT INTO {table_name} (ID, LIMIT_BAL, SEX, EDUCATION, MARRIAGE, AGE, Y) VALUES (%s, %s, %s, %s, %s, %s, %s)"
    values = (int(row['ID']), int(row['LIMIT_BAL']), int(row['SEX']), int(row['EDUCATION']), int(row['MARRIAGE']), int(row['AGE']), int(row['Y']))
    cursor.execute(insert_query, values)
```

```
# Insert the DataFrame into MySQL                    PAY AMOUNTS
for _, row in payamounts.iterrows():

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']),  int(row['PAY_AMT1']), 0)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['PAY_AMT2']), 2)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['PAY_AMT3']), 3)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['PAY_AMT4']), 4)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['PAY_AMT5']), 5)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['PAY_AMT6']), 6)
    cursor.execute(insert_query, values)
```

```
# Insert the DataFrame into MySQL                    PAYSTATUS
for _, row in history.iterrows():

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']),  int(row['PAY_0']), 0)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['PAY_2']), 2)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['PAY_3']), 3)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['PAY_4']), 4)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['PAY_5']), 5)
    cursor.execute(insert_query, values)

    insert_query = f"INSERT INTO {table_name} (ID, AMOUNT, MONTH) VALUES (%s, %s, %s)"
    values = (int(row['ID']), int(row['PAY_6']), 6)
    cursor.execute(insert_query, values)
```

The following will create the table with the SQL queries :

```
# ================================================================================
# Queries timer
# ================================================================================

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="bazouka2000",
  database="defaultcc"
)

mycursor = mydb.cursor()


queries = [
    "SELECT c.*, rs.AMOUNT FROM clients AS c INNER JOIN repayementstatus AS rs ON c.ID = rs.ID AND rs.MONTH = 0",
    "SELECT c.ID, SUM(bs.AMOUNT) AS total_bill_amount FROM clients AS c LEFT JOIN billstatements AS bs ON c.ID = bs.ID GROUP BY c.ID",
    "SELECT c.* FROM clients AS c WHERE c.MARRIAGE = 1 AND c.EDUCATION = 1",
    "SELECT ps.MONTH, AVG(ps.AMOUNT) AS average_payment_amount FROM paystatements AS ps GROUP BY ps.MONTH",
    "SELECT c.*, bs.AMOUNT AS bill_amount, ps.AMOUNT AS payment_amount FROM clients AS c LEFT JOIN billstatements AS bs ON c.ID = bs.ID LEFT JOIN paystatements AS ps"+
    "ON c.ID = ps.ID AND bs.MONTH = ps.MONTH WHERE bs.MONTH = 2",
    "SELECT c.* FROM clients AS c WHERE c.SEX = '1' AND c.LIMIT_BAL > 500000",
    "SELECT c.*, bs.AMOUNT AS bill_amount FROM clients AS c INNER JOIN billstatements AS bs ON c.ID = bs.ID WHERE bs.AMOUNT > (SELECT AVG(AMOUNT) FROM billstatements)",
    "SELECT c.EDUCATION, COUNT(*) AS client_count FROM clients AS c GROUP BY c.EDUCATION",
    "SELECT c.* FROM clients AS c INNER JOIN paystatements AS ps ON c.ID = ps.ID AND ps.MONTH = 4 WHERE c.AGE BETWEEN 25 AND 35",
    "SELECT c.*, bs.AMOUNT AS bill_amount, rs.AMOUNT AS repayment_amount FROM clients AS c LEFT JOIN billstatements AS bs ON c.ID = bs.ID AND bs.MONTH = 5 LEFT JOIN "+
    "repayementstatus AS rs ON c.ID = rs.ID AND rs.MONTH = 5 ORDER BY bs.AMOUNT DESC",
    "SELECT c.ID, SUM(ps.AMOUNT) AS total_payment_amount FROM clients AS c INNER JOIN paystatements AS ps ON c.ID = ps.ID GROUP BY c.ID"
]
```

The following will help compute and print the query times :

```
table = [[],[],[],[],[],[],[],[],[],[]]
for iteration in range(10) :
    timers = []
    for query in queries :
        start_time = time.time()
        mycursor.execute(query)
        myresult = mycursor.fetchall()
        end_time = time.time()
        timers.append(end_time-start_time)

    table[iteration] = timers

strtable = ""
for timers in table :
    toprint = str(round(timers[0], 3))
    for i in range(1,len(timers)) :
        toprint += " & " + str(round(timers[i], 3))
    strtable += toprint + "\\\\  \hline \n"

strtable+= "\hline \n"
means = np.mean(table, 0)

toprint = str(round(means[0], 3))
for i in range(1,len(means)) :
    toprint += " & " + str(round(means[i], 3))
strtable += toprint + "\\\\  \hline \n"

print(strtable)

mycursor.close()

# Close the database connection
mydb.close()
```

And the last one will be needed to log out of the database and let it connect to the next user or request.

```
# Commit the changes and close the cursor
cnx.commit()
cursor.close()

# Close the database connection
cnx.close()
```

FOr the other database, only the query time computing and printing will be shown as the rest is only adaptation of what is shown above :

```
x = 20  # nombre d'iterations
table = [[] for _ in range(x)]


for iteration in range(x) :
    timers = []
    for query in queriesAgeSolo :
        start_time = time.time()
        mycursor.execute(query)
        myresult = mycursor.fetchall()
        end_time = time.time()
        timers.append(end_time-start_time)

    table[iteration] = timers


table2 = np.asarray(table)
table = np.transpose(table2)
means = np.mean(table, 1)

strtable = ""
for j in range(len(table)) :
    timers = table[j]
    toprint = str(round(timers[0], 3))
    for i in range(1,len(timers)) :
        if i % 10 == 0:
            toprint += " & \\\\ \hline " + str(round(timers[i], 3))
        else:
            toprint += " & " + str(round(timers[i], 3))
    if j < len(table)-1 :
        strtable += toprint + " & " + str(round(means[j],3)) + " \\\\  \hline \hline \n"
    else:
        strtable += toprint + " & " + str(round(means[j],3)) + " \\\\  \hline \n"



print(strtable)
```

Abstract :

This work examines the impact of primary and secondary indexes, both in hashtable and B-tree forms, on query optimization. It also offers insights into the scenarios where secondary indexes can significantly enhance query times. This study provides practical guidance for effectively utilizing secondary indexes to improve database query performance, catering to both theoretical understanding and practical implementation.

Résumé :

Ce travail se concentre sur l'analyse des avantages apportés par les index primaires et secondaires dans les bases de données, qu'ils se présentent sous la forme de tables de hachage ou d'arbres B. Il examine également de manière approfondie dans quelles situations les index secondaires sont en mesure d'améliorer les temps d'exécution des requêtes. En analysant les relations entre les attributs indexés, les types de requêtes et les performances, cette étude offre un guide pratique pour identifier les cas où l'utilisation d'index secondaire est bénéfique. Ces conclusions visent à fournir une compréhension solide des mécanismes sous-jacents aux index et à offrir des recommandations utiles pour optimiser les performances des requêtes dans le contexte des bases de données."

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
Louvain School of Management
Place des Doyens, 1 bte L2.01.01, 1348 Louvain-la-Neuve
Boulevard Emile Devreux 6, 6000 Charleroi, Belgique
Chaussée de Binche 151, 7000 Mons, Belgique
www.uclouvain.be/lsm