Final Project

Design and implementation of simple storage and query engines for a column-based and access optimized relational table structure on disk

Lorena Prieto Horcajo
Ingeniería Informática Superior

To my father

These page is intentionally left in blank

Abstract

Relational database management systems currently keep large volumes of data in secondary storage. But with the increasing development of hardware features such as quantity of memory, number of CPUs and CPUs cores, computation cycles, etc. the new trend is to store data in main memory. Such storage will lead to a suited different organization of data that only is efficient if data is always available in main memory.

Taking into account this idea, SAP has developed a new relational DBMS: SAP HANA. It is an implementation of in-memory database technology designed for enterprise computing. In this database it is necessary to separate data into two differenced categories: cold data and hot data. This separation is necessary to handle the increasing, but still limited capacity of main memory efficiently while keeping all data available for reporting needs. Data in a database has its own life cycle and the sequence of events that certain data has passed through determines its state. The term cold data refers to data that has become passive and can be stored in disk if it will not be changed any longer and, thus, this data will be accessed less often. In the other hand, the term hot data is used to refer data that is frequently accessed. The disadvantage with in-memory databases is the memory consumption especially for data which are rarely accessed, i.e. cold data.

Therefore some mechanisms of streaming cold data in/out disk without using any kind of buffering, which allows consuming the minimal amount of memory, have been designed, implemented and evaluated. According to experimental results, the approach that consists of using a column store data model and applying some compression techniques show the best performance.

En la actualidad los sistemas de gestión de bases de datos relacionales mantienen grandes volúmenes de datos en almacenamiento secundario. Pero con el creciente desarrollo de las características del hardware, como la cantidad de memoria, el número de CPUs y núcleos de CPU, ciclos de computación, etc. la nueva tendencia es la de almacenar los datos en la memoria principal. Tal almacenamiento dará lugar a una organización diferente de los datos, que sólo es eficiente si los datos están siempre disponibles en memoria principal.

Teniendo en cuenta esta idea, SAP ha desarrollado un nuevo sistema gestor de base de datos relacional: SAP HANA. Este sistema es una implementación de la tecnología de base de datos en memoria, diseñado para la computación empresarial. En esta base de datos, es necesario separar los datos en dos categorías diferenciadas: *hot data* vs. *cold data*. Esta separación es necesaria para manejar el aumento, pero todavía limitada capacidad, de la memoria principal de manera eficiente, manteniendo todos los datos disponibles. Los datos en una base de datos tienen su propio ciclo de vida y la secuencia de eventos que cierto dato ha pasado, determina su estado. El término *cold data* se refiere a datos que se han vuelto pasivos y se pueden almacenar en el disco si no se cambiarán por más tiempo y, por lo tanto, estos datos son accedidos con menor frecuencia. Por otro lado, el término *hot data* se usa para referirse a datos a los que se accede con frecuencia. La desventaja con bases de datos en memoria es el consumo de memoria, especialmente para los datos a los que rara vez se tiene acceso, es decir, *cold data*.

Por lo tanto, algunos mecanismos de *streaming* de *cold/hot data* con el disco sin usar ningún tipo de *buffering*,lo que permite consumir la mínima cantidad de memoria, han sido diseñados, implementados y evaluados. De acuerdo con los resultados experimentales, el método o aproximación que consiste en usar un modelo de datos en columna y aplicar algunas técnicas de compresión es el que muestra el mejor rendimiento.

Index

Figure Index

Table Index

This page is intentionally left blank

1. Introduction

Today, specialized database management systems (DBMS) are used for storing and managing data. To these systems, the ever increasing volume of data [1] is a significant challenge that necessitates continuous adaption and innovation for existing DMBS. Thus there are two major requirements for a new database management system: data from various sources have to be combined in a single database management system and this data has to be analysed in real-time to support interactive decision taking. [1]

Companies are more data driven than ever before. For example, during manufacturing a much higher amount of data is produced, e.g. by manufacturing robots or assembly line sensors. Moreover, companies process data at a much larger scale to support management decisions.

There are four main types of data and they come from several different sources: [1]

- Transactional data, data entry. Sources: machines, transactional apps, user interaction, e.g. ERP systems.
- Event processing, stream data. Sources: machines, sensors, typically high volume systems.
- Real-time analytics, structured data. Sources: planning, simulation, reporting.
- Text analytics, unstructured data. Sources: web, logs, support systems, etc.

All this kind of data in large volumes has to be handled by current DBMS. This is particularly challenging as these systems are optimized either for daily transactional or analytical workloads. As a result, data is often stored twice and redundant in separated systems. This separation also influences the way data is stored and processed internally, e.g. the underlying persistency layers are storing this data in row or column format. [1] In database management systems, data is kept in secondary storage, i.e. disk, but if the data would be stored in main memory, which is the primary persistence for data, this will lead to a different organization of data that only works if data is always available in memory. There are already some systems that take advantage of that fact, supporting their success in an advanced memory management. For example, Oracle has a system where its physical structures are not so evolved but its buffering really is.

Hardware is also an aspect to take into account. Current DBMS must also take advantages of the improvement of hardware. According to Rock's Law [3] hardware is getting cheaper, this means access to an increasing amount of memory, number of CPUs and CPUs cores [1], computation cycles and more storage capacity at higher speeds for the same budget, enabling the design of bigger and more efficient parallel systems.

Having all this aspects above in mind, SAP has developed a new relational DBMS: SAP HANA. It is an implementation of in-memory database technology designed for enterprise computing.

A formal and technical definition is showed here: “HANA takes advantage of the low cost of main memory data processing […] to deliver better performance of analytical and transactional applications. It offers a multi-engine query processing environment […] as well as graph and text processing for semi and unstructured data management within the same system.” [2]

Since HANA is the database we are working on and it works in main memory, it is necessary to separate the data into cold data and hot data as an approach to handle the increasing, but still limited capacity of main memory efficiently while keeping all data available for reporting needs.

Data in a database has its own life cycle and it can be separated into hot and cold states. The sequence of events certain data has passed through determines its state. The term cold data refers to data that has become passive and can be stored in disk if it will not be changed any longer and, thus, this data will be accessed less often. [1]

This kind of data will still be used for example, reporting. In the other hand, the term hot data is used to refer data that is frequently accessed. Probably, more than 90% of all queries are going against hot data. [1] Hot and cold data can be treated differently as access patterns differ, e.g., read-only access in cold data versus read and writes access in hot data. Different data partitioning, other storage media and different materialization strategies can further be used dependent on the state of the data.

Another concept to take into account is storing data column-wise instead of row-wise. In column-orientation complete columns are stored in adjacent blocks. This can be contrasted with row-oriented storage where complete rows (tuples) are stored in adjacent blocks. Column-oriented storage is well suited for reading consecutive entries from a single column and this can be useful for column scans or aggregations.

We are working with a pure main memory database where data is stored column-wise. But of course, the data is stored on disk as well for durability reasons. The advantage of such a main memory approach is the query speed and the ability to guarantee response times. The disadvantage is the memory consumption especially for data which are rarely accessed, this means cold data.

For all the above, the goal of the thesis is to establish some mechanisms of streaming cold data in/out disk without using any kind of buffering and measure their query performance.

1.1. Document Structure

The document structure is detailed in this section. After the Introduction chapter is the Framework chapter where the work done by others that somehow ties in with this thesis is highlighted, it is work that the thesis is based on.

The next chapter is High-level architecture. This chapter gives an overview of the components of the prototype designed, what their functionality is and how they interact.

Afterwards, Design and Implementation chapters are detailed and explained. Design chapter addresses the question “what do the approaches do?” and explains how the solution given to the problem to solve has been found. Implementation chapter contains descriptions of the implementation of each of the different solutions designed to solve the problem. It is about to explain “how do the approaches work?”

The next chapter is Evaluation and it will present the results obtained for each implemented approach. Each of these solutions has its own evaluation section and it consists of graphics and its explanation. There will be also a comparison between approaches with their advantages and disadvantages and one of the solutions will be chosen as the best for the purpose of the project.

Then, in the Conclusion and Future Work chapter, a summary of the thesis and possible future work to improve the prototype implemented of other approaches is presented.

The penultimate chapter is the References chapter and the last chapter is the Graphics Appendix and it will present more results obtained for each implemented approach due to completeness reasons and that have not been showed in the Evaluation chapter.

2. Framework

This chapter surveys previous work in database management systems (DBMS), more particularly of in-memory data management and in-memory columnar databases.

Something that is much sought and prized for database architecture research is to find a solution that is scalable and speedy, stable and secure, small and simple and self-managing. Almost forty years of relational database technology seemingly converged into one commonly accepted system construction lore but in the meanwhile, the hardware landscape changed considerably. [4] High performance applications requirements posed by e.g., business/intelligence analytic applications; typically as part of a data warehouse (central repository for all or significant parts of the data that an enterprise's various business systems collect). [7] Similarly, data being managed changed from relational tables only through object-oriented, putting increasing complexity onto existing systems. These aspects combined, caused a major shift in how database management solutions can be created. [4]

The abundance of main memory makes it the prime choice for current database processing. However, effective use of CPU caches became crucial. Designing a DBMS from the perspective of large main memories and multiple data models called for a re-examination of the basic storage structured needed. Column-store architecture have become crucial because they reduce the amount of data manipulated within a database engine and also because columns form an ideal building block for realizing more complex structures (tables, objects, trees or graphs). [4]

Column Store is the design of a read-optimized relational DBMS that contrasts with most of the systems that existed years ago, which was write-optimized. With the row store architecture, a single disk write sufficed to push all the fields of a single record out to disk. Hence, high performance writes are achieved, so this is a write-optimized system. These are especially effective on OLTP-style applications (OLTP: OnLine Transaction Processing). In contrast, systems oriented toward ad-hoc querying of large amounts of data should be read-optimized. These are OLAP systems (OLAP: OnLine Analytical Processing). Data warehouses represent one class of read-optimized system, electronic library catalogs, etc. [5] In such environments, a column store architecture, in which the values for each single column are stored contiguously, should be more efficient. Products like Sybase IQ [6], [8] demonstrate this efficiency. Column storage was successfully used for many years in OLAP and really surged when main memory became abundant. [10]

With a column store architecture, a DBMS need only read the values of columns required for processing a given query and can avoid bringing intro memory irrelevant attributes. In warehouse environments where typical queries involve aggregates performed over large numbers of data items, a column store has a sizeable performance advantage. In past DBMS, values were stored in their native format because it was though that it was too expensive to shift data values onto byte or word boundaries in main memory for processing. However, CPUs are getting faster at much greater rate than disk bandwidth is increasing. Hence, it makes sense to trade CPU cycles, which are abundant, for disk bandwidth, which is not. This trade-off appears especially profitable in read-mostly environment. [5]

But main memory is the new bottleneck and it is required to minimize access to it. Accessing a smaller number of columns can do this on the one hand; so only required attributes are queried. On the other hand, decreasing the number of bits used for data representation can reduce both memory consumption and memory access times. Dictionary encoding builds the basis for several other compression techniques. [9] The main effect of dictionary encoding is that long values, such as texts, are represented as short integer values. This technique is relatively simple and it works column-wise. [1] It consists of two structures: dictionary and index. Dictionary is structure that contains all the distinct values of a column alphabetically sorted and index is a structure where every distinct value of the column is replaced by a distinct integer value. The benefits come to effect with values appearing more than once in a column. The more often identical values appear, the greater the benefits. Since enterprise data has low entropy [1], dictionary encoding is well suited and grants a good compression ratio.

With respect to the in-memory databases (IMDB, also known as a main memory database or MMDB) is a database whose data is stored in main memory to facilitate faster response times. Source data is loaded into system memory in a compressed format. In-memory databases streamline the work involved in processing queries. An IMDB is one type of analytic database, which is a read-only system that stores historical data on metrics for business intelligence/business analytics applications, typically as part of a data warehouse. These systems allow users to run queries and reports on the information contained, which is regularly updated to incorporate recent transaction data from an organization's operational systems. [11]

All the previous research and evolution explained before was taken into account by SAP. In order to take away the different systems for OLAP and OLTP, their mixed workloads [15] and to combine them all in one unique system, SAP started to develop a hybrid system called SAP HANA.

The overall goal of the SAP HANA database is to provide a generic but powerful system for different query scenarios, both transactional and analytical, on the same data representation within a highly scalable execution environment. [12]

SAP HANA database is a main-memory centric database system full ACID. The main features of the SAP HANA database for the scope of typical business applications are:

- The HANA database contains a multi-engine query that offers different data abstractions supporting data different degrees of structure from well-structured relational data to irregularly structured data graphs to unstructured text data. [12]
- It supports the representation of application-specific business objects directly inside the database engine. [12]
- HANA database is optimized to efficiently communicate between the data management and the application layer. [12]
- The SAP HANA database supports the efficient progressing of both transactional and analytical workloads on the same physical database leveraging a highly-optimized column-oriented data representation. [12]

The system provides life cycle management for an individual database record. The SAP HANA conceptually propagates records through different stages of a physical representation. [12] There are three stages for records within a regular table as showed in Figure 1 and the most important and related with the thesis are explained below.

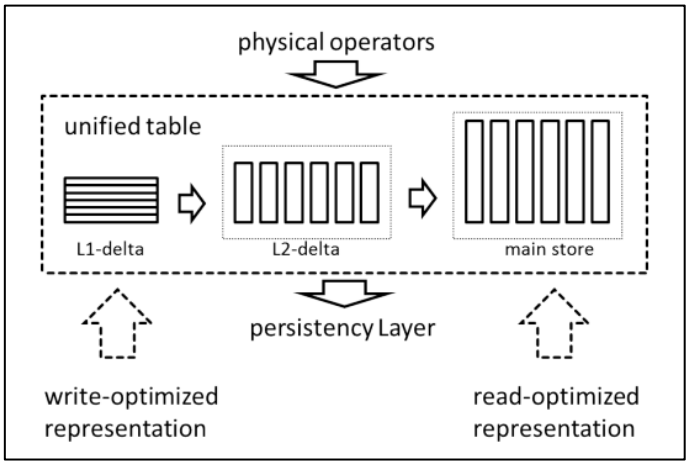

Figure 1: Overview of the unified table concept [12]

- L2-delta: The L2-delta structure represents the second stage of the record life cycle and is organized in the column store format. The L2-delta employs dictionary encoding to achieve better memory usage. However, for performance reasons, the dictionary is unsorted requiring secondary index structures to optimally support point query access patterns. The L2-delta is well suited to store up to 10 million of rows. [12]

- Main store: The main store represents the core data format with the highest compression rate exploiting a variety of different compression schemes. All values within a column are represented via the position in a sorted dictionary and stored in a bit-packed manner to have a tight packing of the individual values [13]. Combinations of different compression techniques are applied to further reduce the main memory footprint [14].

In relation with the storage mechanisms and the I/O performance and to optimize data transfer to and from mass storage, relational DBMSs have organized records in slotted disk pages using the N-ary Storage Model (NSM). This technique stores records contiguously starting from the beginning of each disk page, and uses an offset table at the end of the page to locate the beginning of each record. [21] To minimize unnecessary I/O, the Decomposition Storage Model (DSM) was proposed [23]. DSM partitions an n-attribute relation vertically into n sub-relations, each of which is accessed only when the corresponding attribute is needed. Queries that involve multiple attributes from a relation, however, must spend tremendous additional time to join the participating subrelations together.

A new data organization model appeared on 2001: Partition Attributes Across (PAX). For a given relation, PAX stores the same data on each page as NSM. Within each page, however, PAX groups all the values of a particular attribute together on a mini page. At the same time, all parts of the record are on the same page. To reconstruct a record one needs to perform a mini-join among mini pages, which incurs minimal cost because it does not have to look beyond the page. [22]

To store a relation with degree n, PAX partitions each page into n mini pages. Then it stores values of the first attribute in the first mini page, values of the second attribute in the second mini page, and so on. There is also a header structure which contains pointers to the beginning of each mini page, the number of attributes, the attributes sizes, the current number of records on the page and the total space available on the page. Each mini page consists on fixed-length (at the end of each mini page there is a vector with one entry per record that denotes null values) or variable-length attribute values (at the end of the each mini page there is a vector with pointers to the end of each value). Each newly allocated page contains a page header and as many mini pages as the degree of the relation. [22] This design is shown in Figure 2.

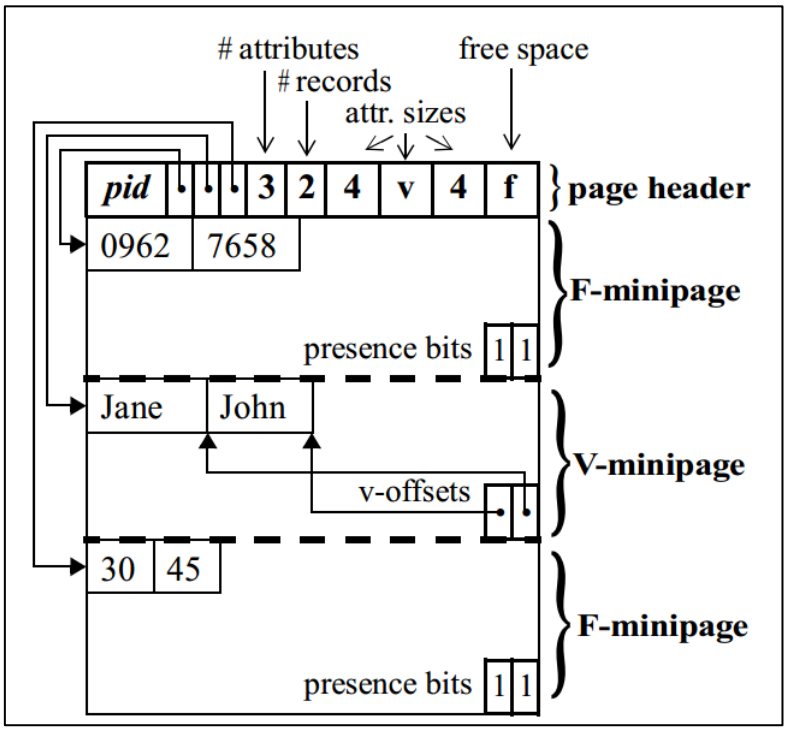

Figure 2: An example PAX page [22]

3. High-level architecture

As it has stated in the Introduction chapter, the goal of the thesis is to establish some mechanisms of streaming cold data in/out disk without using any kind of buffering and measure their query performance.

The high-level architecture gives an overview of the components of the prototype designed and implemented, what their functionality is and how they interact. Some figures are showed for a better understanding of the prototype.

Figure 3 is a general vision of the prototype.

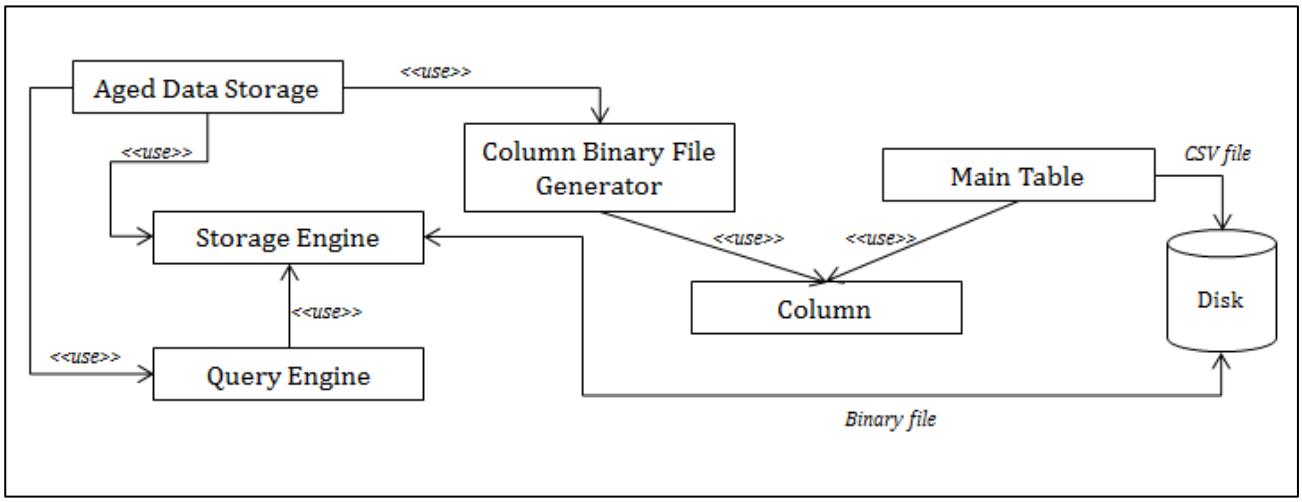

Figure 3: Prototype

- Main Table: this component of the prototype is in charge of parsing a .csv (Comma Separated Values) file from disk and loads each column of the table in main memory. It interacts with the disk, the Column component and the Column Binary File Generator component: the interaction with the disk is done during the parsing of the .csv file, the interaction with the Column component is done once the .csv file is parsed and one Column object has to be created and loaded in main memory for each column of the table (each value of a row of the .csv file is a column of a table) and the interaction with the Column Binary File Generator is performed when the table loaded in main memory has to be written to a binary file on disk.

- Column: represents a column object from a table in main memory. The columns are loaded in main memory with compression; this means that a column is represented by two concepts: index and dictionary. These two concepts are explained in the Framework chapter. Column component interacts with the Main Table component and with the Column Binary File Generator component as explained above.

- Column Binary File Generator: this component of the prototype is responsible of writing the table columns that are in main memory in a binary file. It interacts with the Store Engine component to choose the method to perform the writing of the binary file.

- Storage Engine: this part represents the engine in charge of writing the table columns in a binary file using a certain approach and it is also in charge of reading the binary file from disk using a certain approach. It interacts with the disk to read/write the binary file and it interacts with the Aged Data Storage component because it is the main part of the prototype and it is in charge of choosing the approach to use for read/write the binary file.

- Query Engine: this element of the prototype represents the engine in charge of executing the queries detailed in the Evaluation Design section of chapter 6. It interacts with the Storage Engine component because it uses this component to read the data of the binary file from disk and perform the query operations. This component is responsible of identifying the resources required by the query, retrieve the necessary information from disk invoking the Storage Engine, making the processing and creating the structure where the results will be stored.

- Aged Data Storage: the last element is the Aged Data Storage which is the main component. Its functionality consists of execute the whole process: get the .csv file from disk, parse it, load the columns in main memory, write the columns from main memory to a binary file on disk, read the binary file from disk and execute the queries to test the performance.

Figure 4 is a component diagram of the main parts of the prototype. There is a main component, Aged Data Storage, which is used to make the I/O operations with the disk and the queries. This component is dependent from the Query Engine and the Storage Engine components.

The Storage Engine component is the element that writes and reads a binary file to/from disk following certain approach or method to write or read the data to/from disk.

The Query Engine component is the element that performs the query operations to test the performance of the method used to write the binary file to disk. That is why it is dependent from the Storage Engine component, because it uses this component to read the data of the binary file from disk.

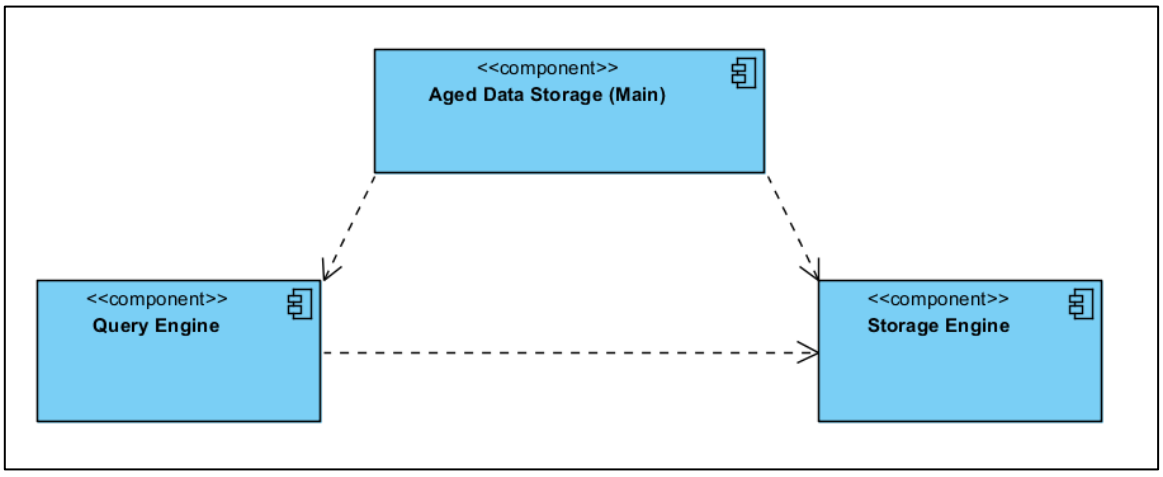

Figure 4: Component Diagram

There are four different approaches or solutions for the I/O operations; therefore each type of approach has its own component. This is shown in Figure 5. The different approaches are explained in the Design and Implementation chapters but their names are:

- Approach 1: Column Store.
- Approach 2: Column Store with compression.
- Approach 3: Row Store without compression.
- Approach 4: Partition Attributes Across (PAX).

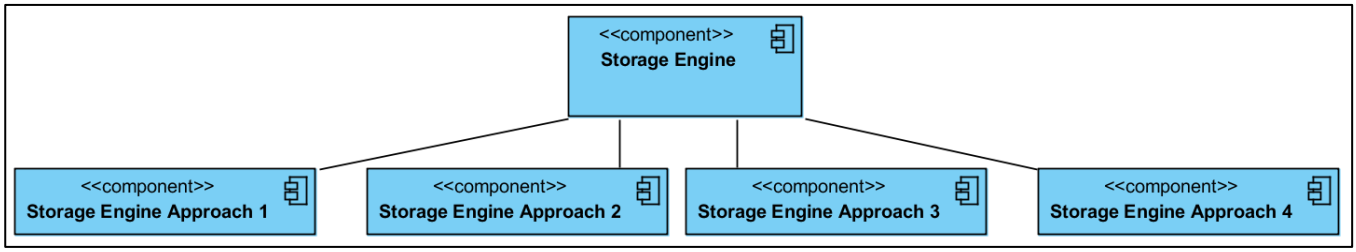

Figure 5: Storage Engine Component Diagram

Just as there are four approaches of storing and retrieving data on disk, there are four ways to test the performance of these approaches. Each of them corresponding to a different approach: Query Engine Approach 1 is the engine to test the performance of the Storage Engine Approach 1; Query Engine Approach 2 is the engine to test the performance of the Storage Engine Approach 2, etc. This Query Engine is shown in Figure 6.

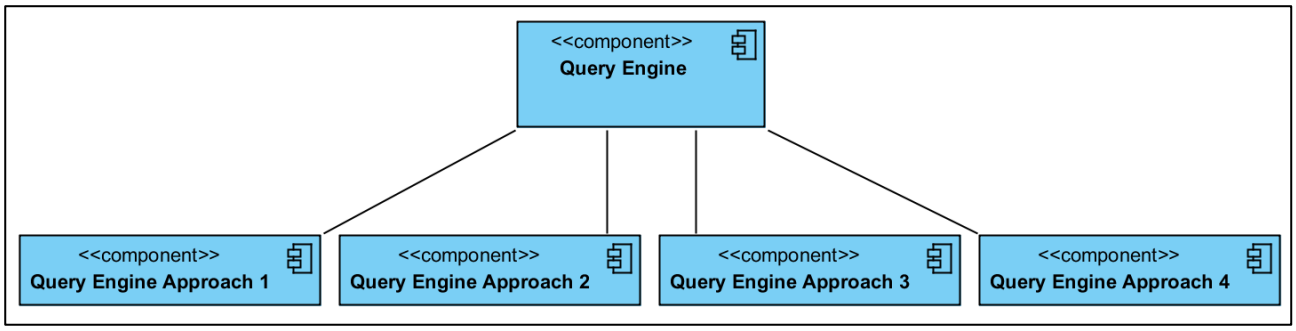

Figure 6: Query Engine Component Diagram

4. Analysis and Design

This chapter contains the descriptions of the different approaches. This chapter addresses the question "what do the approaches do?" and explains how the solution given the problem to solve has been found.

The purpose of the project is to design and implement some mechanisms of streaming data in and out of disk; hence some algorithms were designed and implemented before the final approaches.

As explained in previous chapters HANA is a database in-memory so one of the requirements to test the possible approaches was to design and implement a little prototype of one of the parts of this database instead of work directly with HANA.

The basic idea is: having a .csv file, load it in columns in main memory. To achieve this, the file is read and parsed as a table in columns and then these columns are stored in main memory. This is the Main Store. Once the columns of the table are in main memory, they have to be stored in disk.

Since main memory is the new bottleneck [1], it is required to minimize access to it. Decreasing the number of bits used for data representation can reduce both memory consumption and memory access times. Therefore, the columns are loaded in main memory with compression; this means that a column is represented by two concepts: index and dictionary.

As it was expected this technique of compression works column-wise and the benefits come to effect with values appearing more than once in a column.

Various ideas were considered and implemented before the final approaches. These attempts do not strictly belong to the project but they show how the design has evolved to the final solutions. The next section will explain some of those ideas and possible solutions.

- Two binary files per column: one dictionary file and one index file. Read the whole dictionary file and store it in memory and read the whole index file and store it in memory. Then, for the queries, work with this two structures in memory. The advantage of this solution is the possibility of working with the entire content of the files in memory without any additional operations (chunking, etc.). The disadvantage of this solution is obvious: the memory consumption could be huge with a big data file.

- Two binary files per column: one dictionary file and one index file. Both index and dictionary files are read in chunks of certain size and loaded in main memory. Therefore, the two files are loaded entirely in main memory without causing high memory consumption during reading (due to petitions of chunks of memory), but at the same time they cause high memory consumption being entirely in main memory. For the queries, work with this two structures in memory. Hence, the read operation in chunks of memory

has no effect. The disadvantage of this solution is obvious: the memory consumption as in the previous approach.

- Two binary files per column: one dictionary file and one index file. Both index and dictionary files are read in chunks of certain size and loaded in main memory. Each chunk is processed for the query after load it in main memory. In order to not to cause high memory consumption, the files are read in small chunks but this produces a lot of read operations and low performance for the queries.

- Pointers to the dictionary. In an attempt to take up less memory and read big chunks of data, the reading of the binary file for the index was implemented as pointer-based. Using a structure of pointers to the strings, there is no copy of the strings in main memory when the string is needed and the result is the same. This structure of string pointers had the same length as the index and each element pointed to its equivalent dictionary value. This approach was not valid due to the need of materialization of the result.

All these alternatives worked with some mechanisms of intermediate buffering and the real need was to find a mechanism that would make the data stream as real as possible, directly from disk. Hence, the alternatives above are not valid.

The objective is the performance I/O operations that are made on the disk: the way the disk is accessed to write/read, where there must not be any kind of structure of intermediate buffering whereas performing these operations, the read operation above all.

Keeping all the above in mind, the final approaches for the project are presented in the next pages and they are divided in two subsections: writing files and reading files. The first one contains the design of the writing algorithm to store the file on disk with and the second one contains the design of the reading algorithm to retrieve the file from disk.

4.1. Approach 1: Column Store

This first solution is based on Column Store principle. A table of cold data is to be stored on disk in a columnar way without any kind of compression. All the columns will be stored on disk as they are in the table.

The next sections explain the way data is written in a file as well as its reading from file, this means mechanisms of I/O.

4.1.1. Writing files

Write binary file: a unique binary file is created. It contains the column itself without any compression (index or/and dictionary), every string is also stored with its length and in chunks of the buffer size, in order to make the reading easier. While storing the data in chunks in the file, if there is any free space in one block due to the length of the

string to store (it does not fit in this chunk and it will be in the next chunk), the free space will be completed with the end of word character "/0".

The graphic representation is shown in Figure 7:

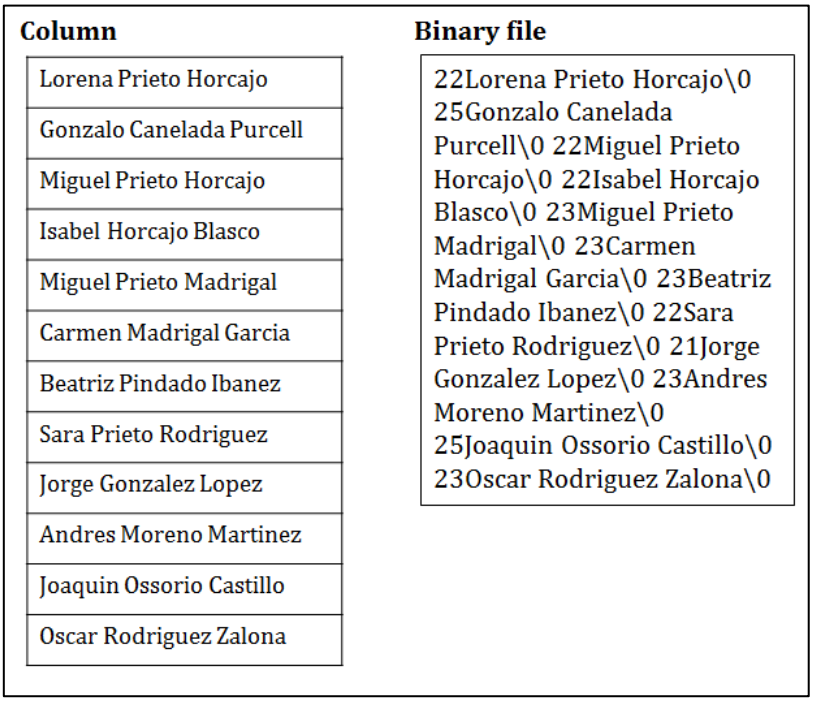

Figure 7: Write column no compression in file

4.1.2. Reading files

Read binary file: in order to read each string of the file, the length of the string is read first and then the number of bytes specified by the length; thus, the string will be read, too.

This read operation is performed differently depending on the type of the query that is executed. For example, reading the file in order to retrieve all the values of a column is done in a different way from reading the file to retrieve some values of a column (in case of a selective query). Each different case to read the column is explained below.

- Read column: with this operation a full scan of the column can be performed. With this approach there is no index, which is why the writing of the file was done in blocks, allowing an easier reading of the whole column or a part of it. To carry out this operation the column is read in chunks of size given by the buffer size and inside of each chunk the length of each string is read with the string itself.

- Read column for a projection with a selection: the first thing to perform is the selection and it consists of getting all the row indexes where the selected value is found. In this case, each element of the column is read and if it is the selected value, its row number is stored. Once the result of the selection is retrieved, all the values

of the projected column have to be read in order to get only their row index and check if they match with the selected ones in the selection process.

4.2. Approach 2: Column Store with compression

This second approach is also based in the Column Store principle. A table of cold data is to be stored on disk in a columnar way with an index and a dictionary per column. The dictionary contains all the possible unique values of the column alphabetically ordered and the index represents all the values of the column with integer values.

4.2.1. Writing files

Write dictionary: a binary file is created with all the unique values of a column, the dictionary. In order to make the later reading of the data easier, these values are stored with its own length in chunks. The size of the chunks is given by the buffer size used for the reading part. While the storing of data in chunks in the file, if the string to store does not fit in the chunk due to its length, there will be some free space at the end of the page and the string will be stored in the next page. This free space will be completed with the end of string character "/0". The graphic representation is shown in Figure 8:

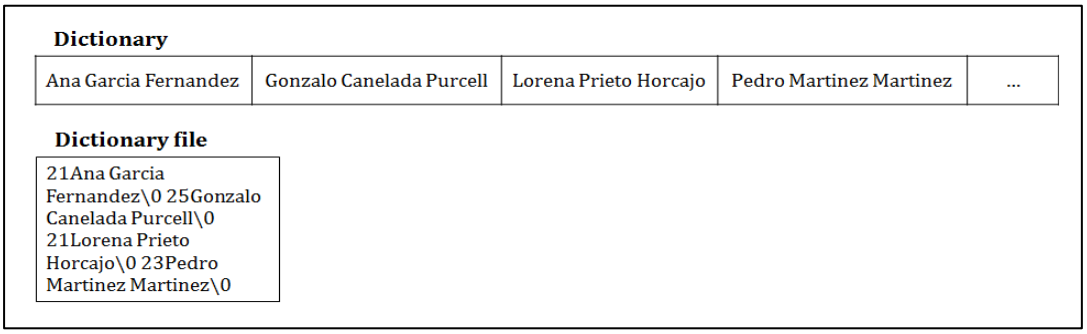

Figure 8: Write dictionary in file

Write index: given that the index of a column consists only in numbers, each one of them representing a value of the dictionary, a binary file is created with all these numbers having each of them 4 bytes long. The graphic representation is shown in Figure 9:

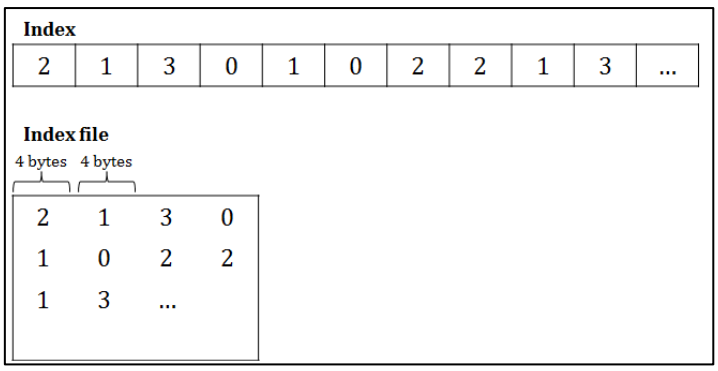

Figure 9: Write index in file

4.2.2. Reading files

Read dictionary: in order to read each string of the dictionary: the length of the string is read first and then the number of bytes specified by the length; thus, the string will be read too. The dictionary is read in chunks of a size given by a buffer size.

Read index: this operation is performed differently depending on the type of the query that is executed. For example, reading the index in order to retrieve all the values of a column is done in a different way from reading the index to retrieve some values of a column (in case of a selective query). The index is read in chunks of a size given by a buffer size. Each different case of how to read the index is explained below:

- Read the index for a projection: since the length of the element is known (4 bytes), each element of the index is read and the materialization is done there (get each dictionary value for each index element), immediately obtaining the result.

- Read the index for a projection with a selection: the first operation to perform is the selection and it consists of getting all the row indexes where the selected value is found. In this case, each element of the index is read and their value for the dictionary is materialized. If the value obtained is the selected value, its row number is stored. Once the result of the selection is retrieved, only the values of the projected column index are read and materialized.

4.3. Approach 3: Row Store without compression

In order to compare the performance of different approaches with different storage methods, the row store approach is designed and evaluated. This third approach is based on the Row Store principle. A table of cold data is to be stored on disk row wise in contrast to the previous approaches but as in the first approach, no compression is used on the data; therefore, there will only be one binary file where the cold data will be stored and read.

4.3.1. Writing files

Write binary file: a unique binary file is created. The data are stored in rows in pages of certain size. Each row is stored complete in the file; in case that the row does not fit into the page it will be stored in the next page of the binary file.

As in previous approaches, the length of each string will be stored before the string itself in order to make the reading easier. An additional integer is added at the beginning of each row in this approach, the length of the complete row.

The graphic representation is shown in Figure 10 and blank spaces are added in the graphic representation for ease of reading. The character "/0" in Figure 10 denotes the end of string.

Rows

Miguel Prieto Horcajo	30	B3-18	Hyundai	Dublin	Sales	Red	7000
Lorena Prieto Horcajo	47	D6-12	Jaguar	Walldorf	Customer Service	Grey	8000
Gonzalo Canelada Purcell	32	C4-42	Subaru	Edinburgh	Marketing	Red	7300
Miguel Prieto Madrigal	47	G6-43	Jaguar	Heidelberg	Marketing	Red	3500
Isabel Horcajo Blasco	45	B5-30	Audi	Reykjavik	IT	Brown	3700

Binary file

61 22 Miguel Prieto Horcajo\0 3 30\0 6 B3-18\0 8 Hyundai\0 7 Dublin\0 6 Sales\0 4 Red\0 5 7000\0 74 22 Lorena Prieto Horcajo\0 3 47\0 6 D6-12\0 7 Jaguar\0 9 Walldorf\0 17 Customer Service\0 5 Grey\0 5 8000\0 (....)

Figure 10: Write rows in binary file

4.3.2. Reading files

In this approach a specific read operation is designed for each specific operation of querying.

For example there is an operation to read the file for a projection query and there is a different operation to read the file for a projection with two or more selection clauses. The file is read in chunks or pages of a size given by a buffer size for all types of reading.

Each different case to read the file depending on the query is explained below.

- Read file for a full scan column / projection query: once the file is opened and all the pages of the file are not read yet: get the length of the row, get the length of the first string and read all the values of the row until the attribute of the projection is reached. (For example, if the attribute *office* is the projection and it is in the fourth position of the row, the three first values have to be read first and then the value of *office* will be read). Then the value is read and stored in a memory structure. The next point to start to read will be calculated with the length of the row. The next step is to follow the same mechanism as explained above with all the pages of the file.

- Read file for a selection query (with projection): once the file is opened and all the pages of the file are not read yet: read all the values of the row until the attribute or column of the selection is reached, and then read its value. If the value read is the value searched for in the selection, read again all the values of the row until the attribute of the projection are reached and store it in a memory structure. The next step is to follow the same mechanism as explained above with all the pages of the file.

- Read file for a multiple selection query (with projection): this operation is derived from the previous one; the only difference is the number of attributes to

check. If the first attribute of the selection is true, the next attribute is checked, otherwise not. If all the attributes of the selection are true, then the attribute of the projection is read and stored in a memory structure. The next step is to follow the same mechanism as explained above with all the pages of the file.

- Read file for a full scan table query: once the file is opened and all the pages of the file are not read yet: for each column, read each value of the row and store it in a memory structure. The next step is to follow the same mechanism as explained above with all the pages of the file.

- Read file for a full scan table with multiple selection query: this operation uses the design of the "read file for a multiple selection query (with projection)" operation in order to get which values fulfil the conditions. For each column of the table all the values that fulfil the conditions are stored in a memory structure. The next step is to follow the same mechanism as explained above with all the pages of the file.

4.4. Approach 4: Partition Attributes Across (PAX)

This fourth approach is based also on column store principle and a variation of the PAX (Partition Attributes Across) strategy explained in the Framework chapter.

A table of cold data is to be stored on disk in a columnar way as in the previous approaches, but in this case there is a special strategy for placing records of the table on a file. The structure of each mini page is: values of the same column are stored in the same mini page and at the end of it there is section of pointers to the end of each value. Obviously, all mini pages that belong to the same page have the same number of records stored. The graphical representation of this approach is shown in Figure 11:

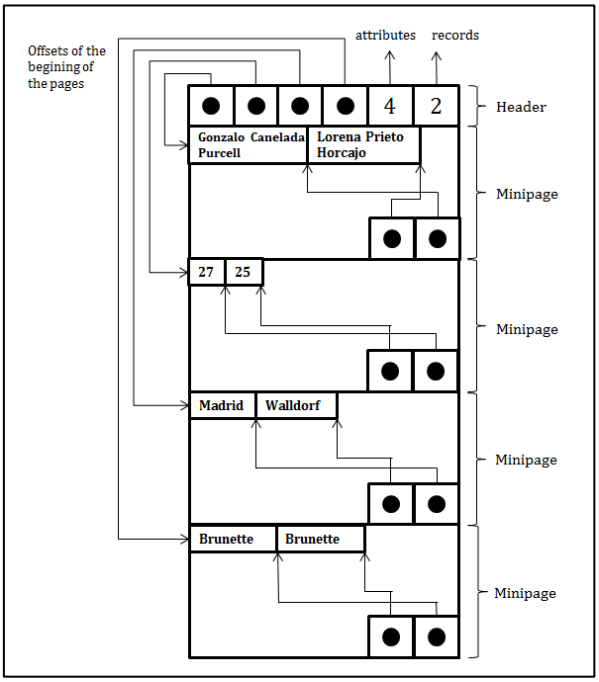

Figure 11: Approach 4 design

The number of records that fit into each mini page is the same for all mini pages of the same page and this number is calculated from the column with longer strings. That is, the column of the table that contains the longer strings will determine the number of values of the column that fit in each mini page.

The next sections explain the way data is written in a file as well as its reading from file, this means mechanisms of I/O.

4.4.1. Writing files

Write binary file: a unique binary file is created. The data are stored in pages of certain size and following the design explained in the introduction of this approach.

Due to the need of having the same number of elements for all mini pages of the same page, the number is pre-calculated and used to know how many elements fit in the mini pages of one page. This number is calculated from the column with the longest strings. If only 4 strings fit in the mini page of the column of the longest strings, 4 strings will be stored in the rest of the mini pages. This is done because the column with the longest strings delimits the rest of the columns since it consumes more space than the other columns.

4.4.2. Reading files

Read operations are associated with query operations so reading and querying are designed together. This is explained below:

- Read file for a full scan column / projection query: once the file is opened, the header of each page is read as the pointers of the end of the mini page of the attribute corresponding to the projection. Then, for each pointer of the end of the mini page, the values of the mini page are read and stored in a memory structure. The next step is to read the next page and follow the same mechanism as explained above and so on.

- Read file for a selection query (with projection): once the file is opened, the header of each page is read as the pointers of the end of the mini page of the attribute corresponding to the projection. Then, for each pointer of the end of the mini page, each value of the mini page is read and if it is equal to the value of the selection, the value read is stored in a memory structure. The next step is to read the next page and follow the same mechanism as explained above and so on.

- Read file for a multiple selection query (with projection): once the file is opened, the header of each page is read. Then, the pointers of the mini page of the projection attribute are read and for each value, pointed by these pointers, is checked whether it fulfil the multiple selection query (if the first condition is true, the next condition is checked, otherwise the next condition is not checked). If the values of the multiple selection query fulfil all the conditions its corresponding

selection value is read from its mini page (value of the same row that fulfil the conditions). The next step is to read the next page and follow the same mechanism as explained above and so on.

- Read file for a full scan table query: once the file is opened, the header of each page is read. Then for each column of the table, each set of pointers of the end of a mini page is read and for each pointer each value of the mini page is read. This read value is stored in a memory structure row-wise. The next step is to read the next page and follow the same mechanism as explained above and do it with all the pages of the file. So, in order to read one page, all the values of one mini page are read first, then all the values of the second mini page and so on.

- Read file for a full scan table with multiple selection query: this operation uses the design of the "read file for a multiple selection query (with projection)" operation in order to get which values fulfil the conditions. For each column of the table all the values that fulfil the conditions are returned.

5. Implementation

The Implementation chapter contains descriptions and aspects about the prototype developing. This part is about to explain "how does the approach work?" The programming language is C++ and development environment used is Visual Studio 2010 Professional on Windows 7. The application is developed in 32-bit but it is also test in 64-bit.

Within Visual Studio on Windows 7, there is a Windows programming function that can create binary files [16] where one of its parameters is a flag that prevents no intermediate buffering. This flag indicates that the file is being opened with no system caching for data reads and writes, so that it does not affect hard disk caching. The flag gives complete and direct control over data I/O buffering. There are data alignment requirements that must be considered:

- File access sizes must be for a number of bytes that is an integer multiple of the volume sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, 1536 or 2048 bytes, but not of 335, 981, or 7171 bytes. [19]

- File access buffer addresses for read and write operations should be physical sector-aligned, which means on addresses in memory that are integer multiples of the volume's physical sector size. Depending on the disk, this requirement may not be enforced. [19]

All the approaches use Windows functions on C++ on the reading [18] sections of the code; these functions were not used in the writing sections since the performance is measured with the reading part of the approaches. The meaning is that is not that important to use Windows functions when writing the binary files because the performance of the approach is measured with the queries and get the data directly from disk is done in the reading part.

As already mentioned above, there are certain requirements or restrictions when using Windows functions and the flag for no file buffering (this flag disables system caching of data being read or write to the file).

To meet the two requirements of file access sizes and file access buffer, the following criteria was used in all the approaches: use a variable that indicates the number of bytes to write/read that is a multiple of 512 bytes as the disk sector size is 512 bytes. This variable is called BUFFERSIZE and is defined in every class for the storage using #define pre-processor macro.

Because buffer addresses for read and write operations must be sector-aligned, the application must have direct control of how these buffers are allocated. One way to sector-align buffers is to use the VirtualAlloc function to allocate the buffers. [19] More criteria have to be considered:

- VirtualAlloc allocates memory that is aligned on addresses that are integer multiples of the system's page size. Page size is 4096 bytes on x64 systems. [19]

- Sector size is 512 bytes. [19]

For all the above, the file is read/write in chunks of a size given by a buffer size with a granularity (512*n).

In the implementation of all the approaches the function VirtualAlloc is called in the constructor of the class that implements the methods to write and read the binary files, it is a void pointer. That statement is:

```
this->indexp = VirtualAlloc(NULL, BUFFERSIZE, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
```

Where the parameters are: NULL meaning that the system determines where to allocate the region, BUFFERSIZE is the size of the region in bytes, MEM_RESERVE | MEM_COMMIT reserves and commit pages in one step (MEM_COMMIT allocates memory charges from the overall size of memory and the paging files on disk for the specified reserved memory pages and MEM_RESERVE reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk).

This function can be used to reserve a region of memory within the virtual address space of a specified process. This region of memory can then be used to map physical pages from/to virtual memory as required by the application.

When this region of allocated memory is no longer needed, the VirtualFree function of Windows is called from the destructor of the class:

```
VirtualFree(this->indexp, BUFFERSIZE, MEM_RELEASE);
```

Where the parameters are: the pointer (indexp) to the base address of the region of pages to be freed, BUFFERSIZE is the region of memory to be freed in bytes and MEM_RELEASE releases the specified region of pages.

To make the measurement of performance, boost library has been used for timing. [20]

Another important thing to take into account is the way the results are returned to the user once a query is executed: for the materialization of the query, some memory structure has to be returned to the user, so it would make sense to use the return statement in every query to send back the result. But there is an inconvenience using this statement and is its performance. When using the return by value statement, the compiler is forced to do an unnecessary copy-in-return when the constructor is invoked and this copy operation has a negative effect in the performance, consuming a lot of time.

An advantage of return by value is that variables that involve local variables declared within the function can be returned. There are no problems when the variable goes out of scope at the end of the function due to the copy made before.

A high performance is the most important feature to achieve in the project, therefore the return statement cannot be used due to the copy that its constructor performs. When a projection

of a column is executed, a large number of values must be placed in memory and then sent back, so they take up a large space on memory.

If for example, there is a table with one million rows and a query as SELECT column FROM table is executed, a million values have to be returned in some kind of memory structure. In our case, a vector<string> is used, so the vector<string> would be in memory with a million of elements; if in addition each element is a very long string, the space taken in memory is much higher. A return statement will generate a copy of this structure, having more taking memory, so this is something to be avoided.

The possible solution is to passing an argument by reference and this argument will be the result. This is a better solution when the function needs to modify the values of an array or a vector (as in our case). In this case is more efficient and clearer to have the function modify the actual vector passed to it, rather than trying to return something back to the caller. When the function is called, the parameter passed by reference will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument. But the most important thing is that a copy of the argument is not made, it is faster, even when used with large structures. []

Running some tests, only the return statement took 50 seconds of the total time of the query, which was 54 seconds. Using the parameter by reference, it only takes ~4 seconds. Hence, this method of passing the arguments by reference has been used in all the approaches for all the queries.

Keeping all the above in mind, the implementation of the final approaches for the project are presented in the next pages and they are divided in two subsections: writing files and reading files. The first one contains the implementation of the writing algorithm to store the file on disk with and the second one contains the implementation of the reading algorithm to retrieve the file from disk.

5.1. Approach 1: Column Store

In this section the implementation of the first approach is described by some text and pseudo-code algorithms.

5.1.1. Writing files

The following section explains the algorithm used to write one file per column.

Write column in binary file without compression: as explained in previous chapters, the Main Store of the prototype implemented contains in memory the columns of a table coding in index and dictionary. Therefore, the first thing to do is to undo the index and dictionary and load the column in memory without any compression. The next thing is to create the binary file with the name of the column and open it.

Then, a "for loop" is needed to write all the elements of the column in the file: get each element of the column, get its length and if the element fits in the chunk (determined by BUFFERSIZE), write the length, write de element. Otherwise, write many "\0" characters (padding) as free space remains in the chunk. Once the loop is done, the last step is to close the file.

------- Pseudo-code to write column without compression -------

1. Create a name for the binary file depending on the name of the column
2. Create and open binary file
3. **For** (iterate over all the elements of the column)
4. **Get i- nth string** from the vector<string> that contains the column
5. Get its length
6. Bytes to be written = string length + size of int + 1 (1 due to \0)
7. **If** the string does not fit in the chunk
8. Write many "\0" as free space remains in the chunk
9. Bytes written will be 0
10. **End if**
11. **Write the length** of the string
12. **Write the string** itself
13. Update bytes written
14. **End for loop**
15. Close the binary file

5.1.2. Reading files

The following section explains the algorithms used to read the files depending on the query. Since there is no compression on the file, the values can be directly read and returned.

Read column for a projection: create the handle of the file with the flag of NO BUFFERING (explained in the introduction of this chapter). While there are bytes to read in the file, read them in chunks determined by BUFFERSIZE and while reading all the bytes of the chunk is not finished, get the length of the element. If the length of the element is 0, it is the end of the page. Otherwise, read the element, push it in a memory structure and update the bytes read until this moment. The last step is to close the handle of the binary file.

------- Pseudo-code to read column for a projection -------

1. Create file handle with flag of NO BUFFERING active
2. **Do-while** (bytes read == BUFFERSIZE)
3. Read next BUFFERSIZE bytes of file
4. **While** (bytes read until this moment < bytes read)
5. Get the length of the string
6. **If** the length of the string is 0 (end of page)
7. Bytes read until this moment = bytes of the string read
8. End if

9. **Read the string**
10. Push the string in a vector<string>
11. Update the bytes read until this moment
12. **End while loop**
13. **End do-while loop**
14. Close handle of the binary file

Read column for a selection: the code implemented for this operation is the same as the previous one but with one difference: once the element is read, it is checked if the element is equal to the element of the selection. If this is true, the row index of the element is stored in a memory structure.

------- Pseudo-code to read column for a selection -------

1. Create file handle with flag of NO BUFFERING active
2. **Do-while** (bytes read == BUFFERSIZE)
3. Read next BUFFERSIZE bytes of file
4. **While** (bytes read until this moment < bytes read)
5. Get the length of the string
6. **If** the length of the string is 0 (end of page)
7. Bytes read until this moment = bytes of the string read
8. End if
9. **Read the string**
10. **If** the string read == string of selection
11. **Store row index in vector<string>**
12. **End if**
13. Update the bytes read until this moment
14. **End while loop**
15. **End do-while loop**
16. Close handle of the binary file

Read column for a selection with a projection: create the handle of the file with the flag of NO BUFFERING (explained in the introduction of this chapter). Then, iterate over the result of the previous operation (Read column for a selection). While there are bytes to read in the file corresponding to the projection column, read them in chunks determined by BUFFERSIZE and while reading all the bytes of the chunk is not finished, get the length of the element. If the length of the element is 0, it is the end of the page. Otherwise, read the element and if it is the element searched, push it in a memory structure and update the bytes read until this moment. The last step is to close the handle of the binary file.

------- Pseudo-code to read column for a selection with a projection -------

1. Create file handle with flag of NO BUFFERING active
2. **Iterator** over the positions where the value is found (iterator it)
3. **Do-while** (bytes read == BUFFERSIZE)
4. Read next BUFFERSIZE bytes of index file

5. **While** (bytes read until this moment < bytes read and the iteration is not finished)
6. **Read the length** of the string
7. **If** the length of the string == 0 (end of page)
8. Bytes read until this moment = bytes read
9. **End if**
10. **Read the string**
11. **If** the string read is the string wanted (string_read == element pointed by the iterator)
12. **Push** in the vector<string> result the **string**
13. Update iterator index (++it)
14. **End if**
15. Update the bytes read until this moment
16. **End while loop**
17. **End do-while loop**
18. Close handle of the binary file

5.2. Approach 2: Column Store with compression

In this section the implementation of the second approach is described by some text and pseudo-code algorithms.

5.2.1. Writing files

The following text and pseudo-code represents the algorithm to write the index and the dictionary in binary files on disk.

Write index in binary file: having the index and the dictionary of a column in main memory, the only thing to do to write the index in a binary file is: create the name for the file depending on the name of the column, create and open the binary file, write the whole index (each element has the size of an unsigned integer) and close the binary file. Since this implementation is very simple there is no need of pseudo-code to explain this algorithm.

Write dictionary in binary file: write the dictionary is a little bit harder than write the index in a binary file since the elements to write are strings and not integers but this process is done as in the first approach to write the column without compression:

Create the binary file with the name of the column and open it. Then, a "for loop" is needed to write all the elements of the column in the file: get each element of the column, get its length and if the element fits in the chunk (determined by BUFFERSIZE), write the length, write de element. Otherwise, write many "\0" characters (padding) as free space remains in the chunk. Once the loop is done, the last step is to close the file. The pseudo-code of this algorithm is detailed below:

------- Pseudo-code to write dictionary -------

1. Create a name for the binary file depending on the name of the column
2. Create and open binary file
3. **For** (iterate over all the elements of the column)
4. **Get i- nth string** from the vector<string> that contains the column
5. Get its length
6. Bytes to be written = string length + size of int + 1 (1 due to \0)
7. **If** the string does not fit in the chunk
8. Write many "\0" as free space remains in the chunk
9. Bytes written will be 0
10. **End if**
11. **Write the length** of the string
12. **Write the string** itself
13. Update bytes written
14. **End for loop**
15. Close the binary file

5.2.2. Reading files

The following section explains the algorithms used to read the files depending on the query.

Read dictionary: the algorithm and pseudo-code of this operation is the same as in section 7.1.2 (Read column for a projection).

Read index for a projection: this operation undo the index replacing each integer element by its corresponding value in the dictionary. The first step is to create the handle of the file with the flag of NO BUFFERING (explained in the introduction of this chapter). While the file has pages left, read the file in chunks determined by BUFFERSIZE and then a "for loop" is needed to read all the integer elements of the chunk read: store in a memory structure the result of the dictionary corresponding with the index value. Then update the bytes read until this moment. The last step is to close the handle of the binary file.

------- Read index for a projection -------

1. Create file handle with flag of NO BUFFERING active
2. **While** (file has pages left)
3. Read next BUFFERSIZE bytes of index file
4. **For** (read all the elements (int) in page)
5. **Store** in the corresponding position of the vector **the result** of the int that is read from the file page, the string of the dictionary that correspond with the index value:
   result[(current_page * (BUFFERSIZE/4)) + i] = dictionary[offset[i]];
6. **End for loop**
7. Update total read size and current page

8. **End while loop**
9. Close handle of the binary file

Read index for a selection: the code implemented for this operation is the same as the previous one but with one difference: once the element is read, it is checked if the element is equal to the element of the selection. If this is true, the row index of the element is stored in a memory structure. There is no need of pseudo-code of this operation.

Read index for a selection with a projection: the easiest way to explain this algorithm is with the pseudo-code.

------- Pseudo-code to read index for a selection with a projection -------

1. Create file handle with flag of NO BUFFERING active
2. **While** (file has pages left)
3. Read next BUFFERSIZE bytes of index file
4. Get the page number that we are reading
5. Get first element of the page
6. **Do-while** (page_number == current_page and value indexes from the selection (values_in_where) is not process completely)
7. **Get** the **dictionary value** that corresponds to the row number that is stored in values_in_where
8. **Store** the **dictionary value** in a vector<string>
9. Update the offset of values_in_where
10. **If** values_in_where is not process completely
11. Update page number
12. **End if**
13. Update total read size and current page
14. **End while loop**
15. Close handle of the binary file

5.3. Approach 3: Row Store without compression

In this section the implementation of the third approach is described by some text and pseudo-code algorithms.

5.3.1. Writing files

The following section explains the algorithm used to write the table in one file with a row-wise organization.

Write table in binary file (row store): as explained in previous chapters, the Main Store of the prototype implemented contains in memory the columns of a table coding in index and dictionary. Therefore, the first thing to do is to undo the index and

dictionary and load the table in memory without any compression. The next thing is to create the binary file and open it.

Two nested loops are necessary to write all the rows of the table in the file: the first loop iterates over the rows and the second loop iterates over the columns calculating the length of the row (adding the length of each string of each column of the same row). Once the length of the row is calculated, if the row fits in the available chunk, the length of the row is written and another loop begins to write each length and each element of the row. Otherwise, padding is added at the end of the chunk. The last two steps are to update the bytes written in one loop and once all the iterations are done, close the file.

------- Pseudo-code to write binary file row store -------

1. Create a name for the binary file
2. Create and open binary file
3. **For** (number of rows: row_index)
4. **For** (number of columns: column_index)
5. Calculate the length of the row
6. **End for loop**
7. **If** there is no more space in the page
8. Write "\0" character from the last character written until the end of the page
9. **End if**
10. **Write the length of the row**
11. **For** (number of columns)
12. Value = column.at(column_index).at(row_index)
13. **Write the length of the value**
14. **Write the value**
15. **End for loop**
16. Update bytes written
17. **End for loop**
18. Close the binary file

5.3.2. Reading files

The following section explains the algorithms used to read the files depending on the query.

Taking into account that a row in the binary looks like:

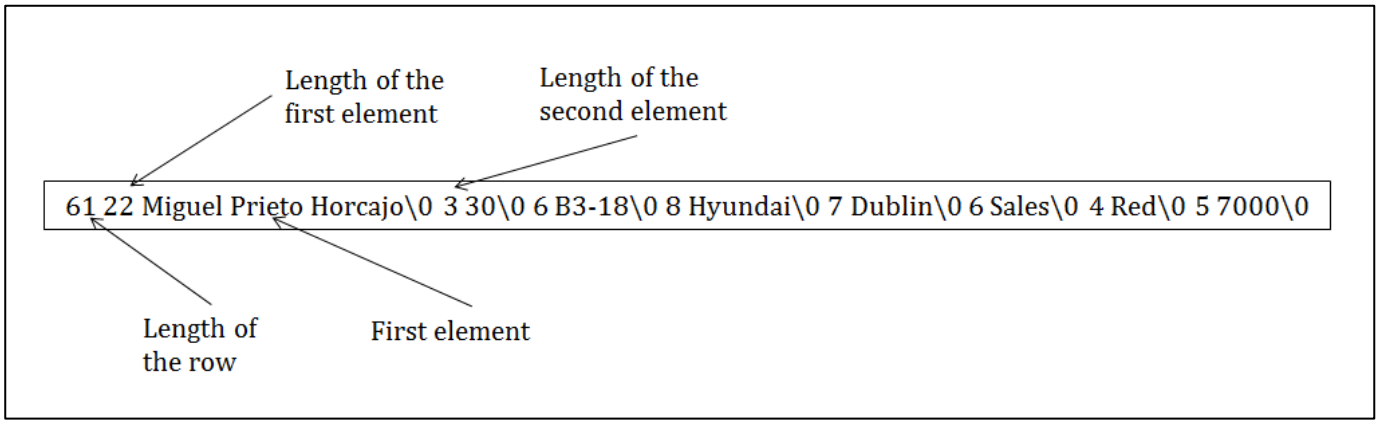

Figure 12: Row represented in the binary file

Read file for a full scan column / projection query: create the handle of the file with the flag of NO BUFFERING (explained in the introduction of this chapter). While there are pages of the file to read, read them in chunks determined by BUFFERSIZE.

A variable called "offset" is created and it is an unsigned pointer to the data that has been read. "Do-while" loop: get the length of the word (second unsigned value of the data read: &offset[1]) and get the word ((char*)&offset[2]). Inside this loop another loop is necessary ("for loop") to read each element of the row until the value of the projection column is reached; hence, the length and the word itself have to be calculated in each iteration of the loop.

When the element of the projection column is read in the row, the element is stored in a memory structure and some variables are updated. The last step is to close the handle of the binary file.

------- Pseudo-code to read file for a full scan column / projection query -------

1. Create file handle with flag of NO BUFFERING active
2. **While** (pages read <= pages to read)
3. Read next BUFFERSIZE bytes of the binary file
4. Offset = (unsigned*) data read. Unsigned pointer to the chunk read.
5. **Do-while** (bytes read < BUFFERSIZE and length of row != 0)
6. Word length = (unsigned *)&offset[1]
7. Word = (char *)&offset[2]
8. **For** (number of the column of the projection)
9. Get all the previous values of the columns of the row until the value of the projection column is reached (calculate in the loop the word length and the word again)
10. **End for loop**
11. **Store the value** in a memory structure
12. Update the position of the memory structure where the value is stored
13. Update bytes read
14. Update the offset to read the next row
15. **End do-while loop**

16. Update pages read
17. **End while loop**
18. Close handle of the binary file

Read file for a selection query (with projection): this algorithm works as the previous one but it has a difference: for each element of the projection column of the row, its value is compared with the value searched for. If these two values are equal, the value of the selection column of the row is read (another "for loop" is necessary to read all the previous values until the value of the selection column of the row is reached); otherwise, the next row has to be read.

Read file for full scan table query: create the handle of the file with the flag of NO BUFFERING (explained in the introduction of this chapter). While there are pages of the file to read, read them in chunks determined by BUFFERSIZE.

A variable called "offset" is created and it is an unsigned pointer to the data that has been read. "Do-while" loop: read each element of the row and store it in a memory structure while there are still bytes to read and the length of the row is different from zero; hence, the length of the word and the word itself have to be calculated in each iteration of the loop. The last step is to update de number of pages read and at the end, close the handle of the binary file.

------- Pseudo-code to read file for a full scan table query -------

1. Create file handle with flag of NO BUFFERING active
2. **While** (pages read <= pages to read)
3. Read next BUFFERSIZE bytes of the binary file
4. Offset = (unsigned*) data read. Unsigned pointer to the chunk read.
5. **Do-while** (bytes read < BUFFERSIZE and length of row != 0)
6. **For** (number of columns)
7. **Get each value of the row and store it** in a memory structure
8. **End for loop**
9. Update bytes read
10. Update the offset to read the next row
11. **End do-while loop**
12. Update pages read
13. **End while loop**
14. Close handle of the binary file

Read file for a full scan table with multiple selection query: having in mind all the algorithms explained before, the easiest way to explain this algorithm is with the pseudo-code.

--- Pseudo-code to read file for a full scan table with multiple selection query --

1. Create file handle with flag of NO BUFFERING active
2. **While** (pages read <= pages to read)

3. Read next BUFFERSIZE bytes of the binary file
4. Offset = (unsigned*) data read. Unsigned pointer to the chunk read.
5. **Do-while** (bytes read < BUFFERSIZE and length of row != 0)
6. **If** the row fulfil all conditions
7. **For** (number of columns)
8. Get the name of the attribute of the column
9. **Get the value** of the attribute
10. **Push the value** in a memory structure
11. **End for loop**
12. Push the previous memory structure in other memory structure
13. **End if**
14. Update bytes read
15. Update the offset to read the next row
16. **End do-while loop**
17. Update pages read
18. **End while loop**
19. Close handle of the binary file

5.4. Approach 4: Partition Attributes Across (PAX)

In this section the implementation of the fourth approach is described by some text and pseudo-code algorithms.

5.4.1. Writing files

The following text and pseudo-code represents the algorithm to write the table of cold data following the design of this approach. Since to write the data with this design is something tricky some methods are implemented and explained below.

Write a binary file with the PAX modified design is divided in methods:

Method 1: PAX Storage. This is the main method to write one file for one table with the design of the fourth approach. The pseudo-code is easy:

------- Pseudo-code PAX Storage -------

1. **While** (number of elements written < total number of elements to write)
2. Calculate number of elements that fit in one mini page
3. **For** (number of columns)
4. **PAXStorageColumn**
5. **End for loop**
6. **End while loop**

Method 2: PAX Storage Column. This is the method to write n elements of a column in a mini page. The method also stored a memory structure of the pointers to the end of each value of each mini page.

The first step is to create the name for the binary file and create it. If the column to write in the file is the first of the table, the header of the page has to be written first:

Then, the elements of the column can be written. For this step a "for loop" is needed (the loop is repeated as many times as the number of elements indicates): get the element of the column, calculate the number of bytes to be written, write the element, update the bytes written, update the pointer to the end of the element (byte number where the element ends) and add this last value to a memory structure. The last step is to update the total number of elements written and to update the end of the mini page.

------- Pseudo-code PAX Storage Column -------

1. Create a name for the binary file
2. Create and open binary file
3. **If** the column is the first of the table
4. **WriteHeader**
5. **End if**
6. **For** (number of elements per minipage)
7. **Get the item** of the column
8. Calculate the bytes to be written (item size + 1 + size of int)
9. **Write the item**
10. Update bytes written (bytes written += bytes to be written)
11. Update the pointer of the item written (pointer_offset += item size + 1)
12. Add this pointer_offset to a memory structure
13. Update number of elements written
14. **End for loop**
15. **EndOfMinipageUpdate**
16. Close file

Method 3: Write Header. This is the method to write the header of each page. The first step is to create a memory structure to store all the values of the header. Then store in the memory structure the following information: header size, values of the pointers to the beginning of each mini page, number of columns, and number of elements per mini page. As a last step, write this memory structure in the file.

Method 4: Update the end of mini page. This is the method to write the set of pointers at the end of each mini page. Before writing the set of pointers, some padding has to be added from the end of the last element to the first position of the set of pointers that is going to be written. The next step is to reverse the vector of pointers that has the values on bytes of the ending of each value of the column in the binary file, write this vector in the file and update the bytes written.

5.4.2. Reading files

The following text and pseudo-code represents the algorithms to read the file depending on the query operation.

For understanding some of the reading algorithms, some descriptive figures have been included.

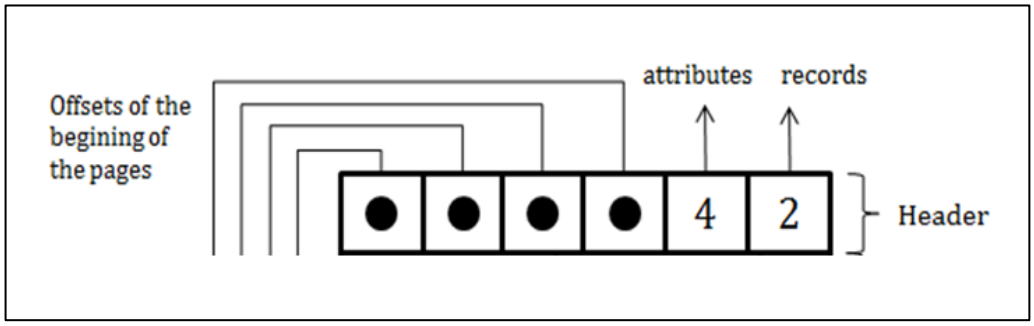

Figure 13: Page header example of Approach 4

Read file for a full scan column / projection query: this algorithm reads each mini page of the same column in the binary file. Once the file handle with the flag of NO BUFFERING active is created, a "while loop" is necessary to read all the pages of the file: while there are pages of the file to read, read them in chunks determined by BUFFERSIZE. Since the first thing of each page is the header, an unsigned pointer to the bytes read is created and initialized. With this pointer, the info of the header is accessible.

The next step is to get the mini page number; since the table contains 8 columns, each page will have 8 mini pages (one per column) so the mini pages are numbered as follows: first column is mini page number 0, second column is mini page number 1, third column is mini page number 2, etc. This is done to get the position and the value of the pointer of the header that point to the beginning of the mini page.

The following step is a "for loop" from 0 to the number of elements allow per mini page. The following figure is showed for a better understanding:

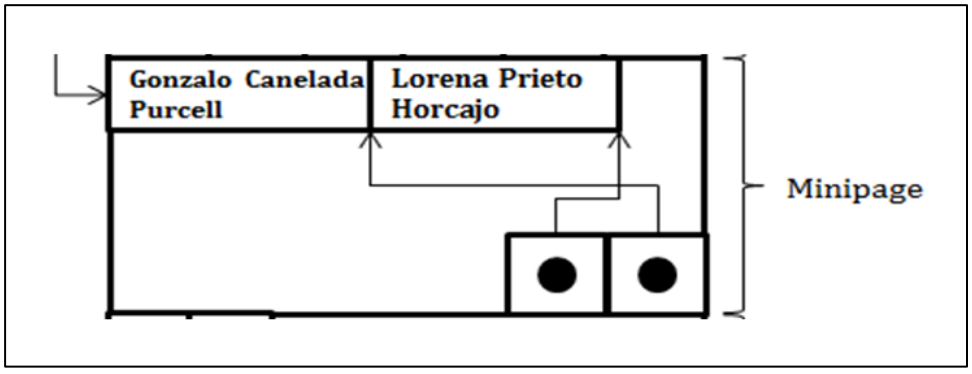

Figure 14: Mini page example of Approach 4

The length of each element of the column is calculated in two different cases:

- If it is the first iteration: it is the first element of the column to be read. The length of the element is to be read and it is calculated with the last pointer of the pointers at the end of the mini page minus the header size.

- If it is not the first iteration: the length of the element is to be read and it is calculated with its corresponding pointer of the pointers at the end of the mini page minus the length of the previous element. Taking Figure 14 as an example the length of the element "*Lorena Prieto Horcajo*" is calculated subtracting the byte number where the element ends minus the byte number where the previous element ("*Gonzalo Canelada Purcell*") ends. Hence, if "*Gonzalo Canelada Purcell*" ends on the byte number 50 (this value is contained on the set of pointers) and "*Lorena Prieto Horcajo*" ends on the byte number 72 (this value is contained in the set of pointers), the length of "*Lorena Prieto Horcajo*" will be 22 (72 - 50).

Once the length of the element is calculate, the element itself can be retrieved creating the string and store it in a memory structure. When the "for loop" is finished, one mini page has been read.

All the algorithm explained above will be repeated in the "while loop" for all the pages of the file.

------- Pseudo-code to read file for a full scan column / projection query -------

1. Create file handle with flag of NO BUFFERING active
2. **While** (pages read <= pages to read)
3. Read next BUFFERSIZE bytes of the binary file
4. int* header = (int*) data read
5. Get the **mini page number** of the projection column
6. Get the **value of the pointer** to the beginning of the mini page of the projection column
7. **For** (number of elements per mini page (k))
8. **If** k = 0
9. bytes per word = (pointers end mini page[header[9]-k-1] – header size)
10. **End if**
11. **Else**
12. bytes per word = (pointers end mini page[header[9]-k-1] – pointers end mini page[header[9]-k])
13. **End else**
14. **Create the string** s = string(&data read[beginning of the word], bytes per word)
15. **Store the string** in a memory structure
16. **End for loop**
17. Update pages read
18. **End while loop**
19. Close handle of the binary file

Read file for a selection query (with projection): the next algorithm is similar to the previous one but the difference comes with the selection. Once the element of the projection query has been read, the element is compared to the value searched in the selection. If it is the element searched for, the length and the value of the projection column corresponding to the element, is retrieved and stored in a memory structure.

Read file for a full scan table query: this algorithm is used to read all the mini pages of all pages of the file and retrieved row-wise. Once the file handle with the flag of NO BUFFERING active is created, a "while loop" is necessary to read all the pages of the file: while there are pages of the file to read, read them in chunks determined by BUFFERSIZE. Since the first thing of each page is the header, an unsigned pointer to the bytes read is created and initialized. With this pointer, the info of the header is accessible.

The next step is a "for loop" iterating over the number of columns, so all the mini pages of the page read are going to be read. For each mini page, its set of pointers at the end of the mini page is retrieved and every element of the mini page is read and stored in a memory structure (the length of each element is calculated following the steps explained in the first algorithm of this section). Therefore, the next thing is to process the next mini page until all the mini pages of the same page are processed. Then, read the next page and follow the algorithm and so on.

6. Evaluation

The next chapter will introduce the queries that will be used to evaluate the approaches and the workloads used. It also presents the results obtained and their evaluation for each implemented approach.

6.1. Evaluation design

The table of data used to test the performance of the approaches designed is a table of 8 columns and 1 million rows. Table 1 specifies the types and values of each column.

The first column contains the names of each column of the table, the second column contains the number of possible distinct values of the column, third column has the average number of characters of each column and the last column contains the type of length of the column, i.e. variable or fixed. All the values of the table are strings of characters, although column age and column salary are numbers but they are treated also as strings of characters.

Table 1: Description of table used to test the approaches' performance

Column	Distinct values	Average number of characters	Element length in column
Name	26	22	Variable
Age	15	2	Fixed
Office	15	6	Fixed
Car	19	6	Variable
City	7	8	Variable
Department	10	11	Variable
Hair	5	5	Variable
Salary	15	4	Fixed

This table has been chosen to test the performance of the approaches due to the following aspects:

- All data is generated by randomly, selecting elements from a set of values fixed for each column.
- Each column contains elements of variable or fixed length (not both in the same column).
- Variable length values are both short and long to test how these lengths affect the performance.
- The number of rows is large enough to produce large files on disk.
- The number of distinct values is high enough to test the performance of some algorithms used in the approaches, e.g. binary search.

Table 2 shows the queries chosen to test the performance of the approaches with the workload explained above.

Table 2: Queries used to test the approaches' performance

QUERY 1	SELECT name FROM table
QUERY 2	SELECT age FROM table
QUERY 3	SELECT name WHERE age = "16"
QUERY 4	SELECT name WHERE department = "Human Resources"
QUERY 5	SELECT department WHERE city = "Walldorf"
QUERY 6	SELECT age WHERE department = "Human Resources"
QUERY 7	SELECT salary WHERE car = "BMW"
QUERY 8	SELECT name WHERE age="16" AND city="Walldorf"
QUERY 9	SELECT * WHERE age="16" AND city="Walldorf"
QUERY 10	SELECT * FROM table

Since the work is done in an in-memory database with columnar organization and the mechanisms to be implemented and evaluated are only I/O operations of read and write, the queries above are chosen as the best ones to be considered to test the performance of the approaches. And since the kind of data is cold data, it has been considered that no deletes or updates operations would be performed.

Queries 1 to 8 are full scans of one column and these types of queries are well suited for the column-oriented storage (consecutive entries in main memory). The projection queries are also simple, but they are used to take basis measurements of reading and how the data model chosen behaves and they serve as baseline to compare the rest of queries.

Both queries 1 and 2 are projections or full scans of one column but they are chosen since the difference between them is that the first one is a full scan over the attribute with the largest length while the second one is a full scan over the attribute with the shortest length.

Queries 4 to 9 have also the selection operator of relational algebra represented by the "where" clause in the queries. This operator was also chosen for the queries to test how these additional clauses affects the performance of the approaches in terms read operations needed, time to execute the query, etc.

Queries 3 and 4 have the same projection column but Query 4 has a restrictive clause which is a selection of another attribute. These queries are chosen to show the performance difference between a single projection and a projection with a selection.

Query 6 is also selected to show the performance difference with Query 4. Both queries have the same selection clause but the projection attribute is different: selection attribute in Query 4 is the one with the largest length of the table while selection attribute in Query 6 is the one with the shortest length of the table.

The "and" operator of Query 8 and 9 was chosen to make the results of the first selection clause affect the following results of the second selection clause to take measure of how the read operation affects: "does the read operation cause more effect on the final result or does the processing of the data cause more effect on the final result?".

Queries 3 and 8 have the same projection attribute but Query 8 has an additional selection clause, which causes a more restrictive result. Query 8 was selected to compare its performance with the performance of Query 3, to show the difference performance with more selection clauses in the query.

Queries 9 and 10 are full scans of the table which, in contrast to the rest of the queries, are operations well suited for row-oriented storage and being cold data not all the data table will be always retrieved from disk, but these two queries have also been chosen to test the performance of the designed and implemented approaches. The difference between them is that Query 10 has to return all the columns for all the rows of the table while Query 9 has two restrictive clauses and it has to return all the columns but not for all the rows and this is a more expensive operation.

The queries are simple to implement to reduce the computation required to perform them and to give more importance to the readings and the elements to read instead of the processing thereof. They increase in complexity (the first ones are the simplest) and in number of necessary read operations so the expected result is to have a longer time from one query to the next.

6.2. Evaluation implementation

A computer with a specific hardware and software setup has been used to perform this evaluation. The hardware setup is an Intel Xeon CPU X650 @ 2.67 GHz (2 processors) and 24GB of RAM and a hard disk of 1TB, 7200 rpm. The software setup is a 64-bit Operating System Windows 7 Enterprise. The programming language is C++ and development environment used is Visual Studio 2010 Professional.

The test suite for every approach consists on the average times of execution of the queries (showed in the previous section) with different buffer sizes; each of them executed one 1000 times. The buffer size is increased by the power of 2, starting with 512 Bytes and 1048576 Bytes (1 MB) as the maximum size.

As already said, each query is executed 1000 times for each approach. The performance of each query is given by the average of the results of these 1000 tests. This number was chosen to have a large enough sample and to avoid that the possible variations in the results alter the final outcome. In relation to the buffer sizes, 512 bytes is the minimum buffer size due to the restriction explained in the introduction of the Implementation chapter. And in order not to consume memory capacity in an application that is permanently in main memory (SAP HANA), the fact of reserving some of it for cold data functions impacts the performance as it cannot take advantage of all the memory available, so the less space required by the functions, the better. Hence, the use of different buffer configurations to identify which is the optimal point between performance and this memory consumption with the buffer sizes.

The system followed to evaluate each of these solutions consists of an advantages and disadvantages section that will be confirmed in another section with the tests results and charts, measuring the performance of each solution. The charts that will be included will be representative cases to evaluate the approach, as explained in chapter 6 in section 6.1: Evaluation

Design. These charts are comparison between Query 1 and Query 2, comparison between Queries 1, 3 and 8 and comparison between Query 9 and Query 10.

The first chart will show the comparison between the first two queries, Query 1 and Query 2. Both queries are projections of the column name and the column age, respectively. This means to return all the values of the column name and the column age.

Queries 3, 8 and 1 will be the second chat. They have the same projection attribute but Query 8 has an additional selection clause, which causes a more restrictive result. Query 8 was selected to compare its performance with the performance of Query 3, to show the difference performance with more selection clauses in the query. Query 1 is also included in the chart to observe the difference of time of a query without the selection clause but the same projection clause.

In contrast to the rest of the queries, Queries 9 and 10 are full scans of the table, but these two queries have also been chosen to test the performance. The difference between them is that Query 10 has to return all the columns for all the rows of the table while Query 9 has two restrictive clauses and it has to return all the columns but not for all the rows and this is a more expensive operation.

Unexpected results will be presented too. Further on, there will be a comparison between approaches and one of them will be chosen as the best one for the purpose of the project.

The expected result for every query is a curve where the maximum time value will be in the 512 bytes buffer size, while the minimum time value will be in the 1048576 bytes buffer size.

NOTE: Not all the queries that have been tested are graphically shown in this chapter. The rest of the queries are shown in the Graphics Appendix with its own analysis an explanation for completeness reasons.

The next sections present the results of each approach and its own analysis. The last section of this chapter presents de discussion and contrast or comparison of all the approaches.

6.3. Approach 1: Column Store

This section consists of the supposed advantages and disadvantages of the Column Store approach and its test and evaluation. In this last sub-section the pros and cons of the approach will be confirmed along with the unexpected results.

6.3.1. Advantages and disadvantages

One of the obvious advantages of this approach is that only one binary file needs to be created per column and since there is no compression, there is only need of reading the column itself, no decode is required, only to read the values of the column.

With the design of the algorithm the response times of the queries are expected to be faster since there is no compression. In order to retrieve a column, it only has to be read and no decoding process is necessary. But since the CPU is designed to perform operations on numbers not on characters this could be also a disadvantage. The hardware and the CPU in particular, are more tuned to work with numbers than to work with characters.

When dealing with a data model that does not use compression in the file representation, there are two main disadvantages: files are larger on disk because all elements must be present with the same representation, in the same quantity and order and as a side effect, dealing with the raw representation of data, strings of characters, becomes cumbersome compared with the usage of integers as mentioned before about the hardware and CPUs design.

If a search has to be performed on the column to find a concrete value or the positions where a value is stored, another disadvantage arises because there is not a sorted dictionary of the different values of the column, where a search algorithm, as the binary search method, can be applied to. When retrieving a value of a column, the average time for the lookup process is *O(n)*, which means a full scan through the column and this adversely affects the performance.

6.3.2. Results and Evaluation

The following table, Table 3, shows the obtained results of the first approach running the test suite detailed on the beginning of this chapter with the use cases showed.

The first column of the table represents the buffer size used in each test and the first row represents the queries. The last row of the table is the average time for each query with all the buffer sizes. Each cell of time is the average from the 1000 tests executed for each buffer size. For ease of reading the legend of the queries is included again below the table.

Table 3: Test results in seconds of Approach 1

BUFFER-SIZE	QUERY 1	QUERY 2	QUERY 3	QUERY 4	QUERY 5	QUERY 6	QUERY 7	QUERY 8	QUERY 9	QUERY 10
512	15,88411	2,070758	18,29083	19,01149	14,15534	8,890834	8,600166	20,85332	53,23201	48,14598
1024	6,314296	0,947796	6,92179	6,287686	3,25403	2,68004	3,33858	8,354936	19,67373	15,24396
2048	2,458011	0,52664	2,525592	2,909549	1,943017	1,514178	1,501533	3,278412	8,516658	8,327812
4096	1,524178	0,379651	2,721006	2,167867	1,533224	1,162605	1,595058	3,381247	6,383748	6,112135
8192	1,228236	0,281473	2,258852	1,874396	1,201054	0,9056	1,402479	3,058193	4,624783	5,435573
16384	1,05476	0,290194	1,479477	1,675923	1,07949	0,85424	0,95571	2,716522	4,664581	4,577379
32768	1,001261	0,241654	1,361605	1,575289	0,883566	0,712553	0,611009	1,707528	3,972928	3,825141
65536	0,625099	0,166471	0,753698	0,926592	0,625043	0,503016	0,441067	1,003077	2,661765	2,637225
131072	0,469847	0,119585	0,548246	0,67185	0,446036	0,35346	0,279252	0,735778	1,921701	1,920491
262144	0,356084	0,083878	0,375535	0,471778	0,329274	0,255602	0,201629	0,517685	1,439237	1,534103
524288	0,318697	0,09022	0,330827	0,388924	0,258891	0,233721	0,182124	0,434861	1,071454	1,269056
1048576	0,308109	0,089687	0,327138	0,364709	0,244077	0,229286	0,170154	0,427777	1,054032	1,268848
AVERAGE	2,725224	0,440667	3,241216	3,193882	2,162753	1,524443	1,6311563	4,005778	8,934719	8,358142

Table 4: Query Legend

SELECT name FROM table	**QUERY 1**
SELECT age FROM table	**QUERY 2**
SELECT name WHERE age = "16"	**QUERY 3**
SELECT name WHERE department = "Human Resources"	**QUERY 4**
SELECT department WHERE city = "Walldorf"	**QUERY 5**
SELECT age WHERE department = "Human Resources"	**QUERY 6**
SELECT salary WHERE car = "BMW"	**QUERY 7**
SELECT name WHERE age="16" AND city="Walldorf"	**QUERY 8**
SELECT * WHERE age="16" AND city="Walldorf"	**QUERY 9**
SELECT * FROM table	**QUERY 10**

The first chart is shown in Figure 15. It makes sense that if the column is not compressed in any way, the application only has to read the column from the binary file in chunks and there is no need to decompress or decode the data as in the second approach (Column Store with index and dictionary). The chunk of file read has to be processed only to get the strings.

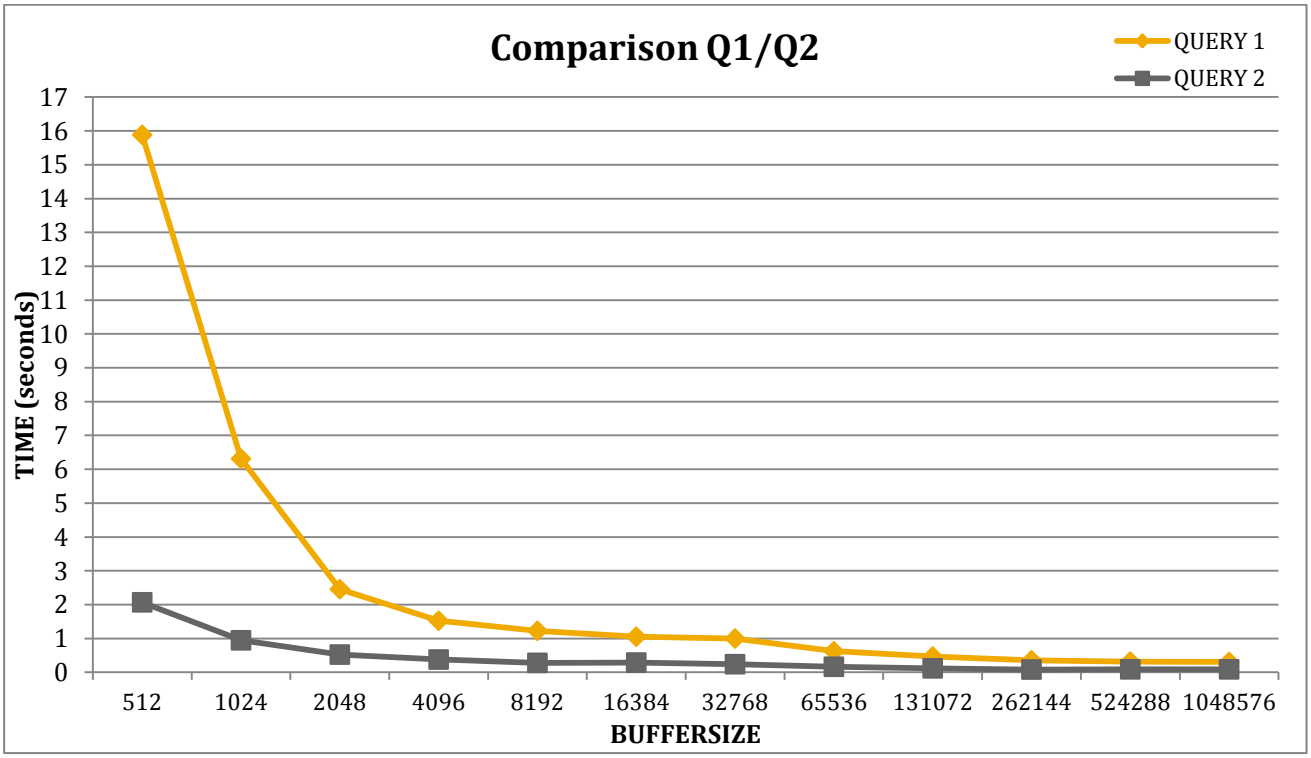

Figure 15: Comparison between Query 1 and Query 2

Query 1 has a sharp descent line from 512 Bytes to 1024 Bytes, meaning that the number of necessary readings to read the entire file is drastically reduced since the double amount of buffer allows the reading of strings in a single read operation, not so in Query 2 with a buffer size of 1024 Bytes.

Query 1 takes much longer than Query 2 and between them there is a high difference of time with the smallest buffer sizes. More strings of 2 bytes length (column *age*) than strings of 22 bytes length (column *name*) can be read with a single read operation; therefore, the time of the query is reduced.

In summary, for small buffer sizes, more read operations are needed in order to retrieve the full column from the file on disk. Moreover if the strings are long as it occurs with the column name.

One of the advantages of the design is proved here: the simplicity of the query operations, since there is no compression, there is only need of reading the column itself. And also a disadvantage is proved: difficulty to manage the strings since there is no compression and the time that this entails causing a bad performance time.

NOTE: An important thing to take into account in this last chart in Figure 15 is the scale. Both curves have the same shape, but due to the scale it cannot be appreciated.

It is also interesting to appreciate the difference between Query 3, Query 8 and Query 1, shown in Figure 16.

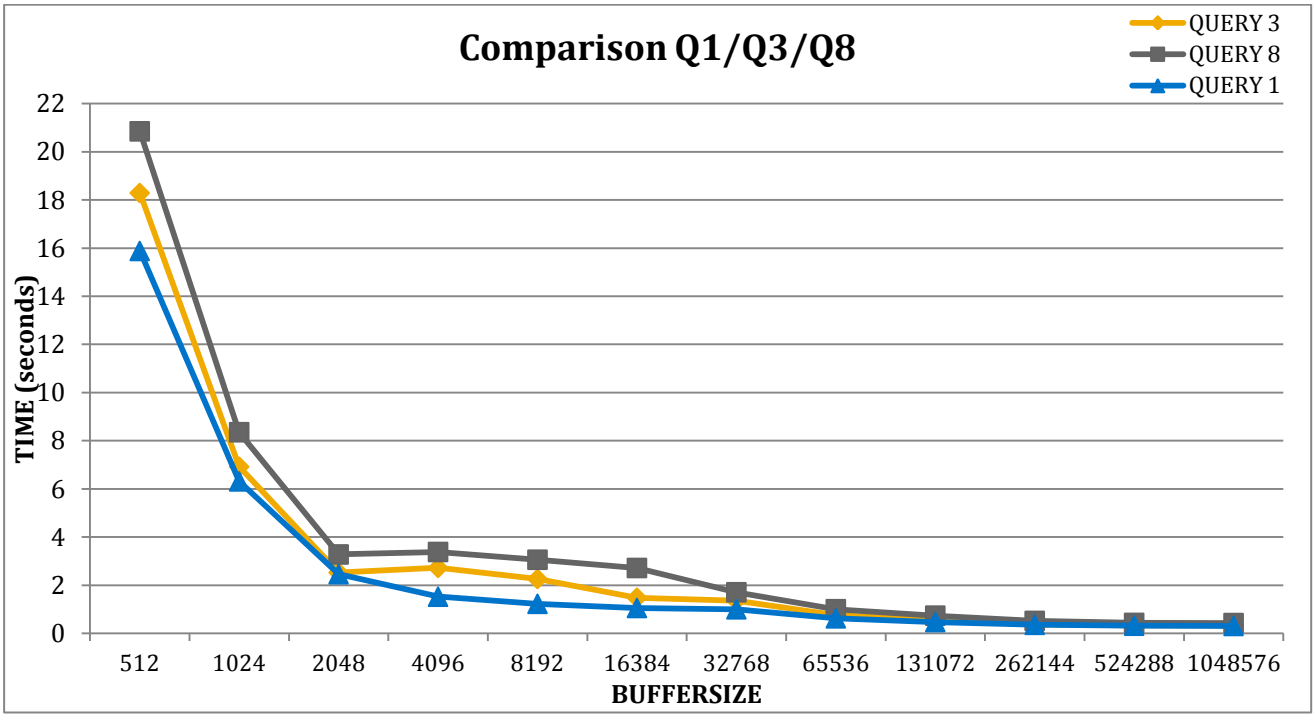

Figure 16: Comparison between Query 3 and Query 8

Query 3 is a projection with a selection, where all the values of the name column that fulfil the condition of having an *age* of *16*, must be returned. This entire operation take more time than a simple projection since the age column has to be read completely in order to find the row indexes where the value *16* is contained and then read the complete name column to get the values that fulfil the condition described by the projection.

The performance result has a fast decrease with little buffer sizes due to the smaller number of necessary input operations to read the entire file from disk and return their values. The higher decrease occurs with a buffer size of 1024 bytes where the time decreases from about 19 seconds to 7 seconds.

As the graphic shows, there is not so much difference between Query 3 and Query 8, although one of them has one selection clause and the other has two selection clauses. Both queries take very similar time in all configurations of buffer size, where Query 8 takes, in average, a little more time.

This extra time is consistent with the increased computation and reading needed to process the second conditions in the selection clause. The difference on the time measurement for each buffer size is the result of the execution of two added tasks, reading and processing the column for the extra selection clause, where the reading task takes up for the majority of the added computation time.

As the buffer size increases the time needed to read the file decreases with the number of reads needed. The computation time for the extra condition is constant through all the runs but its impact on the overall time is small as limited by the row selection done when processing all the previous conditions in the selection clause, in this case, just one.

The next chart in Figure 17 shows the comparison between Query 9 and Query 10. Both queries are a full scan of the table but Query 9 has two selection clauses.

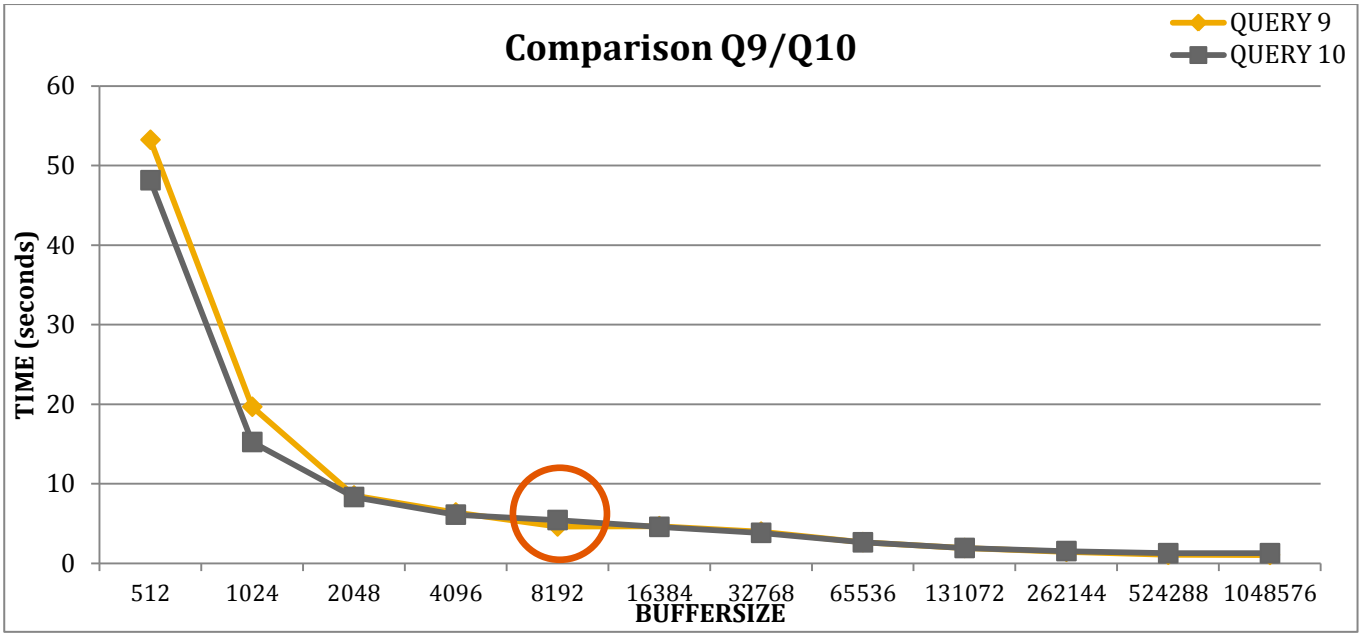

Figure 17: Comparison between Query 9 and Query 10

Query 9 shows a longer time comparing to Query 10 for little buffer sizes, this is due to the selection clauses of the query. It takes more time to make the selection for the first "where" clause which means to read the whole column string by string to lookup for the first value (*age = 16*) and with the resultant values, acting as a previous filter, lookup for the second value (*city = Walldorf*) and then return all the values for the projection column that fulfil all the two conditions than just return all the values for all columns.

An interesting thing to mention is marked in red circle in Figure 17. With an 8192 bytes buffer size is the only configuration where Query 10 takes more time than Query 9 for all the tests. These tests were executed two times to ensure that the result was correct. The measurements are done with the average of many tests to have into account all the possible variations of them and do not affect the actual outcome too, but in this precise case should not occur. At this point, the results go against the assumptions but, we believe that it is not important since it could happen because of the machine load, etc.

6.4. Approach 2: Column Store with compression

6.4.1. Advantages and disadvantages

One of the obvious advantages of this approach is the compression obtained using indexes and dictionaries instead of using any other kind of compression; the binary files will take up less space in disk due to the index structure. The main effect of this kind of encoding or compression is that long values, such as texts or long strings, are represented as short integer values. The benefits come to effect with values appearing more than once in a column, the more often identical values appear, the greater the benefits.

With this data model, when searching a value in the column it has to be searched in a sorted dictionary where the lookup process speeds up from *O(n)*, which means a full scan through the dictionary, take on average *O(log(n))*, because values in the dictionary can be found using binary search. Hence, using dictionaries is also an advantage.

The optimization of sorted dictionaries comes at a cost: every time a new value is added to the dictionary which does not belong at the end of the sorted sequence of the existing values, the dictionary has to be re-sorted. Even the insertion of only one value somewhere except the end of the dictionary causes a re-sorting, since the position of already present values behind the inserted value has to be moved on position up. While sorting the dictionary is not that costly, updating the corresponding index is. But the advantage remains in this design because there are no new insertions in the table because is cold data, neither the dictionary, nor the index needs resorting, which is a very expensive operation.

One of the key advantages of using this compression scheme, apart from reducing the memory footprint, is that many of the repetitive operations are now done using indexes, this means numbers. The hardware, and the CPU in particular, is more tuned to work with numbers than to work with characters, by indexing the contents and referring to the different values using an integer placeholder an implicit speedup is achieved on every operation.

The disadvantage of this approach could be the materialization of the encoded values of the indexes but the impact is rather small. Generally, the result set is small compared to the total table size, so the lookup of all other selected columns to materialize the query result is not that expensive.

With large dictionaries that contain many different values, binary search can negatively impact the overall performance looking up some value. This is a disadvantage of this design. In addition, the design to read the dictionary is inefficient because the dictionary has to be completely in memory in order to work with it. In case of a large dictionary, it will take up a lot of memory.

One additional more disadvantage of this method is the requirement to have two binary files per column. If the table has a high number of columns, there will be a high number of files to manage.

6.4.2. Results and Evaluation

The following table shows the obtained results of the first approach running the test suite detailed on the beginning of this chapter with the use cases showed in the first section of this chapter, section 6.1.

The first column of the table represents the buffer size used in each test and the first row represents the queries. The last row of the table is the average time for each query with all the buffer sizes. Each cell of time is the average from the 1000 tests executed for each buffer size. For ease of reading the legend of the queries is included again below the table.

Table 5: Test results in seconds of Approach 2

BUFFER-SIZE	QUERY 1	QUERY 2	QUERY 3	QUERY 4	QUERY 5	QUERY 6	QUERY 7	QUERY 8	QUERY 9	QUERY 10
512	1,071061	0,783271	1,72485	1,850089	1,750618	1,698466	1,589399	2,441837	8,050246	7,540087
1024	0,935173	0,446217	1,245572	0,951614	0,89328	0,864064	0,932733	1,641914	5,067397	6,461451
2048	0,518768	0,318007	0,54417	0,515546	0,497534	0,57424	0,4319	0,704631	3,418749	4,823176
4096	0,409264	0,217999	0,448162	0,374434	0,385321	0,337661	0,367374	0,560978	2,69625	3,509498
8192	0,275989	0,168048	0,307435	0,292218	0,298216	0,276833	0,258735	0,40267	1,203444	1,794673
16384	0,243702	0,15116	0,287577	0,295286	0,285859	0,26223	0,243372	0,380178	1,130282	1,65248
32768	0,19978	0,116584	0,231369	0,265227	0,230255	0,212257	0,195188	0,30951	0,903614	1,305517
65536	0,164631	0,083938	0,162828	0,167308	0,16434	0,150071	0,127556	0,21148	0,691004	0,983414
131072	0,14646	0,061082	0,118296	0,127387	0,123026	0,108814	0,123191	0,160551	0,44922	0,797713
262144	0,135921	0,046207	0,0854	0,103	0,093	0,0834	0,1	0,125	0,3223	0,6911
524288	0,12932	0,045061	0,07519	0,095179	0,092145	0,070155	0,090652	0,115546	0,30951	0,64498
1048576	0,124167	0,043775	0,0621965	0,083153	0,090309	0,075409	0,066097	0,073309	0,127846	0,271343
AVERAGE	0,362853	0,221668	0,441087	0,426703	0,408659	0,3928	0,377791	0,593993	2,030822	2,539619

Table 6: Query Legend

SELECT name FROM table	**QUERY 1**
SELECT age FROM table	**QUERY 2**
SELECT name WHERE age = "16"	**QUERY 3**
SELECT name WHERE department = "Human Resources"	**QUERY 4**
SELECT department WHERE city = "Walldorf"	**QUERY 5**
SELECT age WHERE department = "Human Resources"	**QUERY 6**
SELECT salary WHERE car = "BMW"	**QUERY 7**
SELECT name WHERE age="16" AND city="Walldorf"	**QUERY 8**
SELECT * WHERE age="16" AND city="Walldorf"	**QUERY 9**
SELECT * FROM table	**QUERY 10**

As the query legend shows, the queries tested with the second approximation are the same as the queries tested with the first approximation. This is obviously done in order to compare the results obtained with all the approximations.

The first chart, Figure 18, shows the comparison between the first two queries, Query 1 and Query 2.

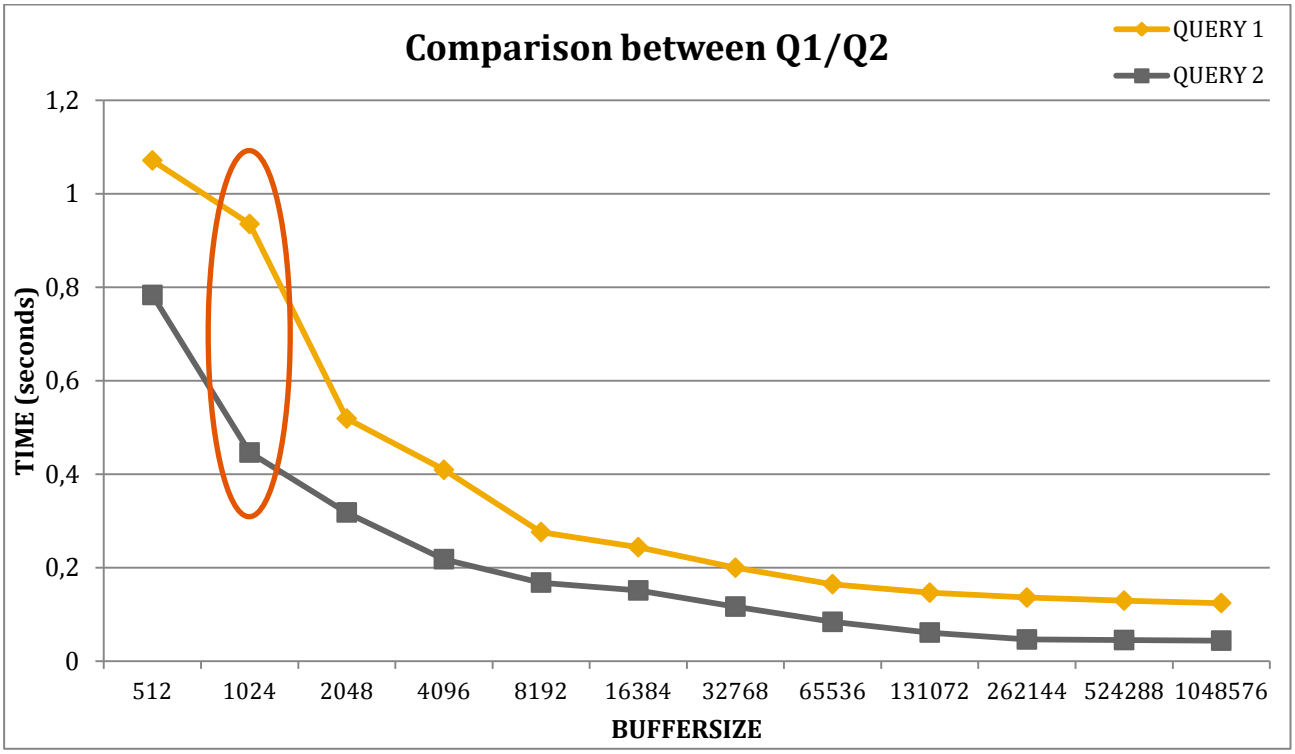

Figure 18: Comparison between Query 1 and Query 2

It takes a lot of time to read the column with little buffer size: many read operations are needed in order to read the whole index. Once the size of the buffer is 8192 bytes the number of necessary readings cannot decrease no more, hence the improvements obtained increasing the buffer sizes are only reflected in a little portion of the total time.

Times of Query 1 are higher since it has to read all the *names* and build the materialization having strings with an average of 22 characters (22 bytes); however Query 2 reads all the *ages* having only strings of 2 characters (2 bytes) and a smaller dictionary. Hence, the query operation is the same but the length of the data produces the timing difference.

Certainly the two curves almost have the same shape, as expected, having the same interval of time difference within the curve; but there is a big difference between the times with 1024 Bytes buffer (marked with a red circle in the graphic).

Query 2 has a sharp descent line from 512 Bytes to 1024 Bytes, meaning that the number of necessary readings to read the entire index file is drastically reduced since the double amount of buffer allows the reading of more index elements in a single read operation, not so in Query 1 with a buffer size of 1024 Bytes.

The previous explanation is valid once Query 1 has a buffer size of 2048 bytes, which is when its time is reduced. Since the number of index elements read is the same for all buffer sizes, this should occur with the same buffer size as Query 2 (1024 bytes) but it does not. The difference could be in the materialization of the result: it takes more time to materialize longer strings (column *name* from Query 1) than shorter strings (column *age* from Query 2).

These tests were executed two times to ensure that the result was correct. The measurements are done with the average of many tests to have into account all the possible variations of them and do not affect the actual outcome too, but in this precise case it does not occur.

The time of the materialization is also a factor to take into account. It is more expensive to read and materialize strings with an average of 22 bytes than the materialization of strings of 2 bytes.

As a next performance comparison, two queries with the same projection clause but different number of conditions in the selection clause are compared. Query 1 is also included in the graph to observe the difference of time of a query without a selection clause.

The expected result could be two curves (Query 3 and Query 8) that have their point of inflexion with a buffer of 2048 bytes of buffer size since the projection query over the column *name* has its point of inflexion with this buffer size. Also, another expected result is that the query which has more conditions to fulfil would be the most longer in time.

Query 3 is a projection over a selective query; the column *age* is retrieved and materialized in order to know which of its values has *age = 16*. Once is known which indexes of the table contain the value *16*, only the values of the column *name* that match the given indexes have to be materialized.

Query 8 is a projection over a multiple selective query; the column *age* is retrieved and materialized in order to know which of its values has *age = 16* and *city = Walldorf*. Once is known which indexes of the table contain the value *16*, only the same indexes of the column *city* will be tested to know which *city = Walldorf* . Then, only the values of the column *name* that match the given index (result of the checking of the two conditions) have to be materialized.

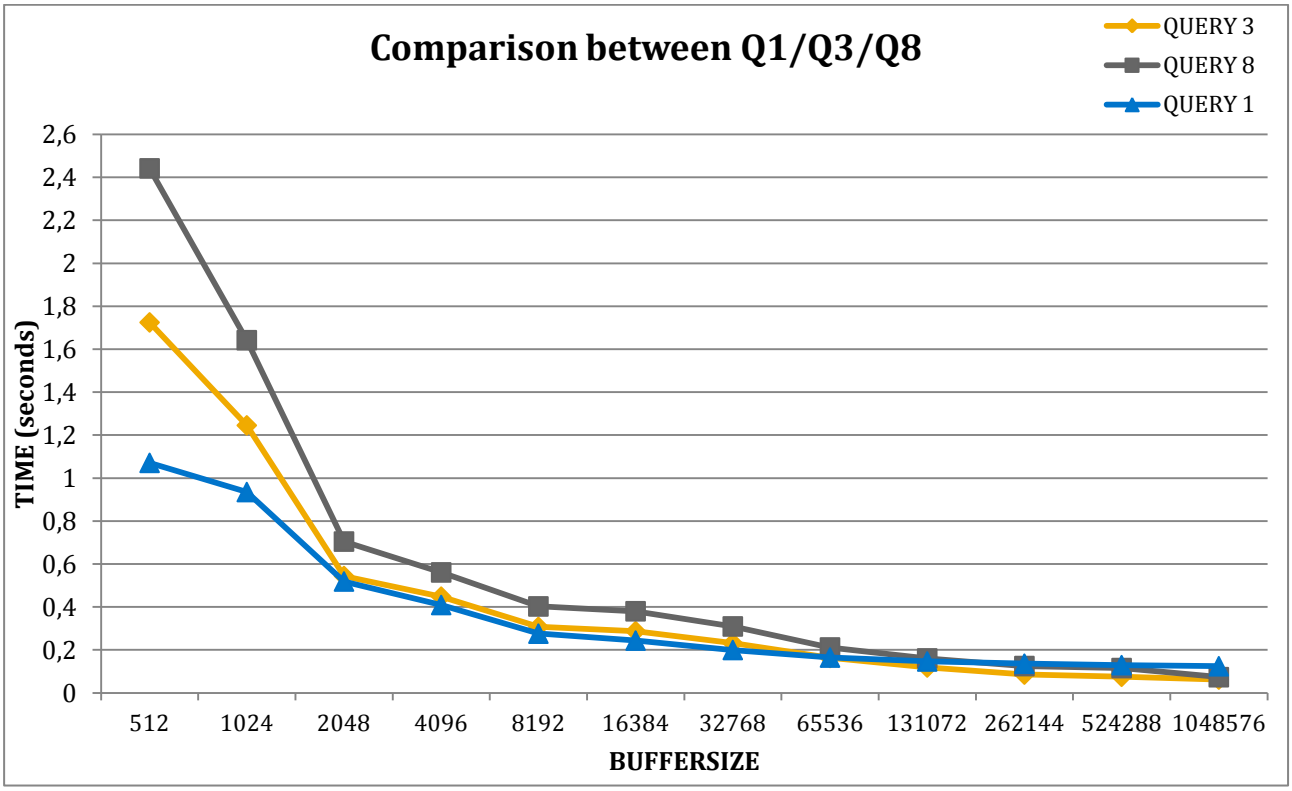

Figure 19: Comparison between Query 3 and Query 8

It can be observed that Query 8 is more restrictive than Query 3. Both curves show almost the same shape but the difference rests in the number of conditions in the selection clause: Query 8 has to make more read operations than Query 3, since it has two indexes to read while Query 3 only has to read one index (besides the rest of operations).

And as it was expected, the point of inflexion is on 2048 bytes. With this buffer size the number of needed readings cannot decrease no more, henceforth the improvement of having a bigger size of buffer to put the data and read it, is not doing much effect in the overall time.

The next chart shows the comparison between Query 9 and Query 10; both of them retrieve and materialize all the columns but the first query uses the projection of the rows that comply with the selection clause.

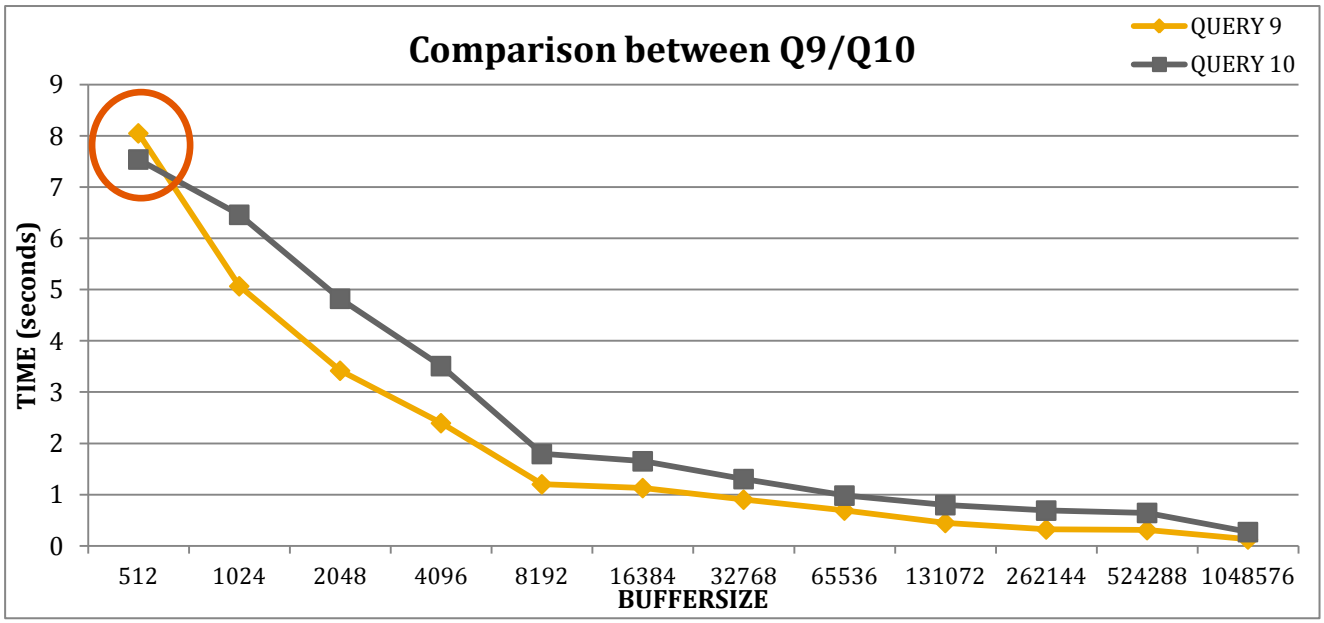

Figure 20: Comparison between Query 9 and Query 10

The second query takes more time than first query due to the significant increase in memory petitions (one for each element in each column that is part of the final result). Each memory allocation takes time and is the huge number of them that counts for the computing time difference.

An unexpected result is marked with a red circle in the graph. Query 9 has a longer first time than Query 10. These tests were executed two times to ensure that the result was correct. The measurements are done with the average of many tests to have into account all the possible variations of them and do not affect the actual outcome too, but in this precise case it does not occur.

6.5. Approach 3: Row Store without compression

6.5.1. Advantages and disadvantages

One of the main advantages of this approach is that the data model only needs one binary file needs to be created for the whole table. Another evident advantage is that the performance for queries where all the row has to be retrieved, as for example full scans, is better because the full row is already been read.

The data model designed and implemented allows skipping rows by having the length of the complete row written preceding each row in the file. This is an advantage because rows that are not necessary to process for certain operation can be avoided at the cost of increasing the size of the file that leads to the needed of more read operations.

When dealing with a data model that does not use compression in the file representation, there are two main disadvantages: files are larger on disk because all elements must be present with the same representation, in the same quantity and order and as a side effect, dealing with the raw representation of data, strings of characters, becomes cumbersome compared with the usage of integers as mentioned in 6.3.1 about the hardware and CPUs design.

Row-wise design is not optimal for full scans of one column or projections since many read operations have to be performed and the full table has to be read, the read operation cannot be limited to the data the query needs.

6.5.2. Results and Evaluation

The following table shows the obtained results of the third approach running the test suite detailed in the beginning of this chapter.

The first column of the table represents the buffer size used in each test and the first row represents the queries. Each cell of time is represented in seconds and it is the average from the 1000 tests executed for each query. For ease of reading a legend of the queries is included below the table. For this approach, the queries were only tested with one buffer size, the biggest buffer size which is 1048576 bytes (1 MB) due to time reasons.

Table 7: Test results in seconds of Approach 3

BUFFER-SIZE	QUERY 1	QUERY 2	QUERY 3	QUERY 4	QUERY 5	QUERY 6	QUERY 7	QUERY 8	QUERY 9	QUERY 10
1048576	0,502046	0,44223	0,444136	0,467686	0,464154	0,44283	0,443668	1,283264	2,139836	4,341184

Table 8: Query Legend

SELECT name FROM table	**QUERY 1**
SELECT age FROM table	**QUERY 2**
SELECT name WHERE age = "16"	**QUERY 3**
SELECT name WHERE department = "Human Resources"	**QUERY 4**
SELECT department WHERE city = "Walldorf"	**QUERY 5**
SELECT age WHERE department = "Human Resources"	**QUERY 6**
SELECT salary WHERE car = "BMW"	**QUERY 7**
SELECT name WHERE age="16" AND city="Walldorf"	**QUERY 8**
SELECT * WHERE age="16" AND city="Walldorf"	**QUERY 9**
SELECT * FROM table	**QUERY 10**

No graphics are showed in this section since there are not sufficient test results to show a curve of the performance. So the evaluation of this approach is made with the following graphic where the total time is represented for each query with the bigger buffer size, which is 1048576 bytes (1 MB).

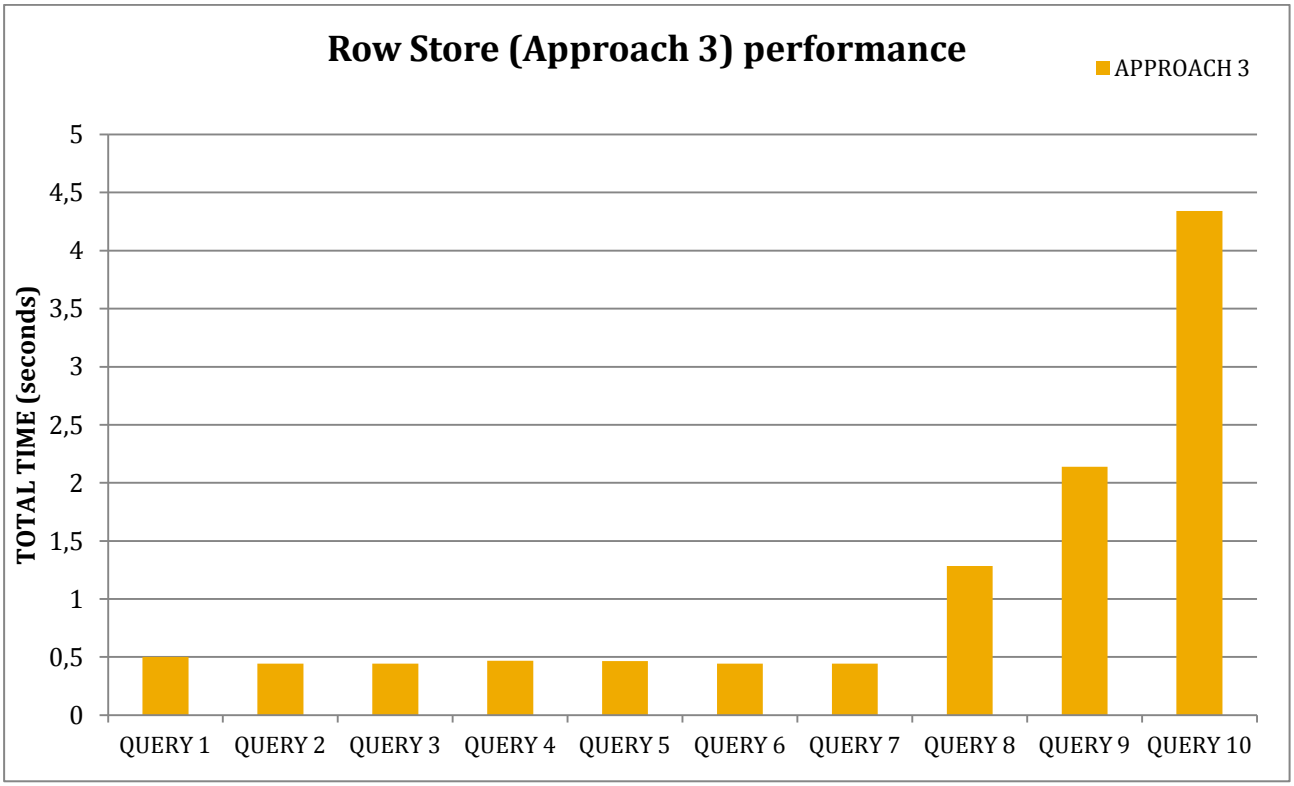

Figure 21: Approach 3 performance

6.6. Approach 4: Partition Attributes Across (PAX)

This section consists of the advantages and disadvantages of the Partition Attributes Across (PAX) approach and its test and evaluation. In this last sub-section the pros and cons of the approach will be confirmed along with the unexpected results.

6.6.1. Advantages and disadvantages

The major advantage of this design is the kind of storage of the data. Each page is composed of mini pages, each of which stores a certain number of values of a column. Therefore, the complete row can be accessed with one read operation. Having this design with mini pages the access to specific values is constant and fairly fast.

Another advantage is the set of pointers at the end of each mini page that point to the end of each value and make easier the reading in case of one specific value is needed. The same thing occurs with the pointers to the beginning of each mini page that are contained in the header of each page; these pointers make easy the access to one specific mini page without having to read the previous mini pages in case of needing a specific one.

The major disadvantage of this design is the waste of space in the file. A lot of space is wasted since the number of records that fit in one mini page is calculated from the column with the longer strings. For example, if the column that has the longer strings is the column *name* and only 4 strings fit in the mini page, the rest of the mini pages will only contain 4 values of minor length, wasting space of these mini pages.

Another disadvantage is the process to store the data with this design, it could be difficult and tedious and it also depends on the table (number of columns, column with the longest strings, etc.) So this design and implementation is strongly linked with the data contained on the table.

6.6.2. Results and Evaluation

The following table shows the obtained results of the fourth approach running the test suite detailed in the beginning of this chapter. As in the previous approaches, the first column of the table represents the buffer size used in each test and the first row represents the queries. For ease of reading a legend of the queries is included below the table.

Table 9: Results test Approach 4

BUFFER-SIZE	QUERY 1	QUERY 2	QUERY 3	QUERY 4	QUERY 5	QUERY 6	QUERY 7	QUERY 8	QUERY 9	QUERY 10
524288	27,43313	27,01344	26,895939	27,02318	26,86866	26,99713	26,8466	55,13187	208,49220	309,12881
1048576	0,98027	0,876418	0,877144	0,893608	0,897592	0,88717	0,878904	2,370254	14,105144	16,067754

Table 10: Query Legend

SELECT name FROM table	**QUERY 1**
SELECT age FROM table	**QUERY 2**
SELECT name WHERE age = "16"	**QUERY 3**
SELECT name WHERE department = "Human Resources"	**QUERY 4**
SELECT department WHERE city = "Walldorf"	**QUERY 5**
SELECT age WHERE department = "Human Resources"	**QUERY 6**
SELECT salary WHERE car = "BMW"	**QUERY 7**
SELECT name WHERE age="16" AND city="Walldorf"	**QUERY 8**
SELECT * WHERE age="16" AND city="Walldorf"	**QUERY 9**
SELECT * FROM table	**QUERY 10**

But in this approach only two buffer sizes were tested due to time reasons, as it can be observed in Table 9.

No graphics are showed in this section since there are not sufficient test results to show a curve of the performance. So the evaluation of this approach is made with the following chart where the total time is represented for each query with the bigger buffer size, which is 1048576 bytes (1 MB).

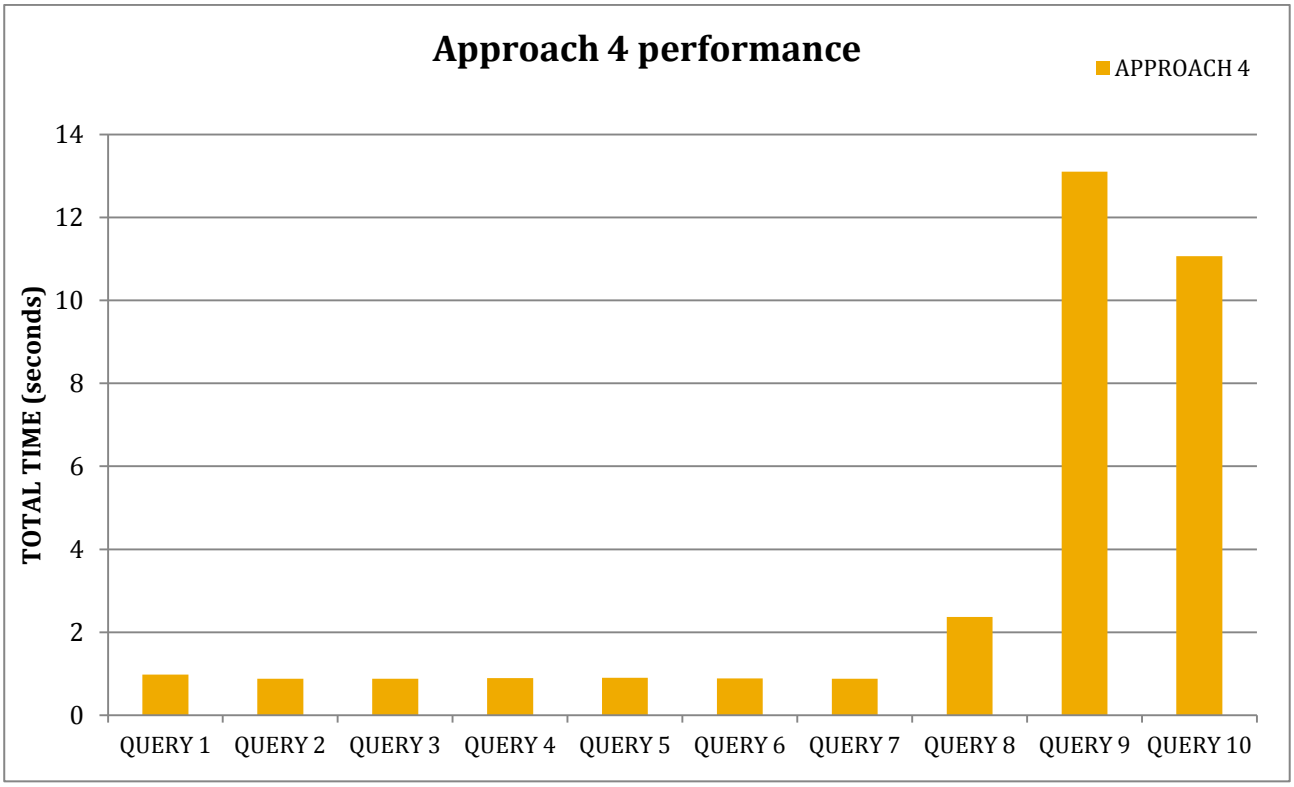

Figure 22: Approach 4 performance

Almost all the queries have a performance of 1 second but the last three queries have a higher performance time.

The high time on Query 8 is due to the two conditions that the program has to check. This is done as explained in the design and implementation chapters: the first condition is checked and if it is true, the second condition is checked and so on. Since in this approach the data is divided in pages and mini pages it has the advantage that with one read operation, one page is read, so all the columns for n records are available in memory. The columns of the projection can be checked with only one read operation but check two or more conditions has a penalty time on the overall time, therefore, the higher time of Query 8.

The long time on Query 9 and Query 10 is due to the operation itself among other factors. Query 10 is a full scan of the table so all the rows for all the columns have to be returned (row-wise). If Query 1 is taken as an example where all the rows must be returned for one column (projection query) and it takes 1 second, to return all the rows for all the columns should take (1 second * number of columns); that is 8 seconds. Some seconds have to be added to these 8 seconds taking into account that some columns have longer strings than others and reading them takes much longer. And finally it takes time to read each page, then all the values of each mini page and place each read string in its corresponding row and column in the result. So at the end the result time of this query can be 11 seconds as showed in the graph of results.

Query 9 has to return all the columns for the rows that fulfil two conditions and this operation takes more time than Query 10 since it has to check the conditions and then return the values row-wise.

6.7. Discussion

This section consists on the comparison between the different approaches designed and implemented. Through this analysis, it can be prove which the best approach for the workload chosen is, how the behaviour of each approach is and how the performance is in general, etc.

The first graph shows the comparison between Approach 1: Column Store and Approach 2: Column Store with compression for all the queries quoted in the Evaluation Design section and used to test the performance.

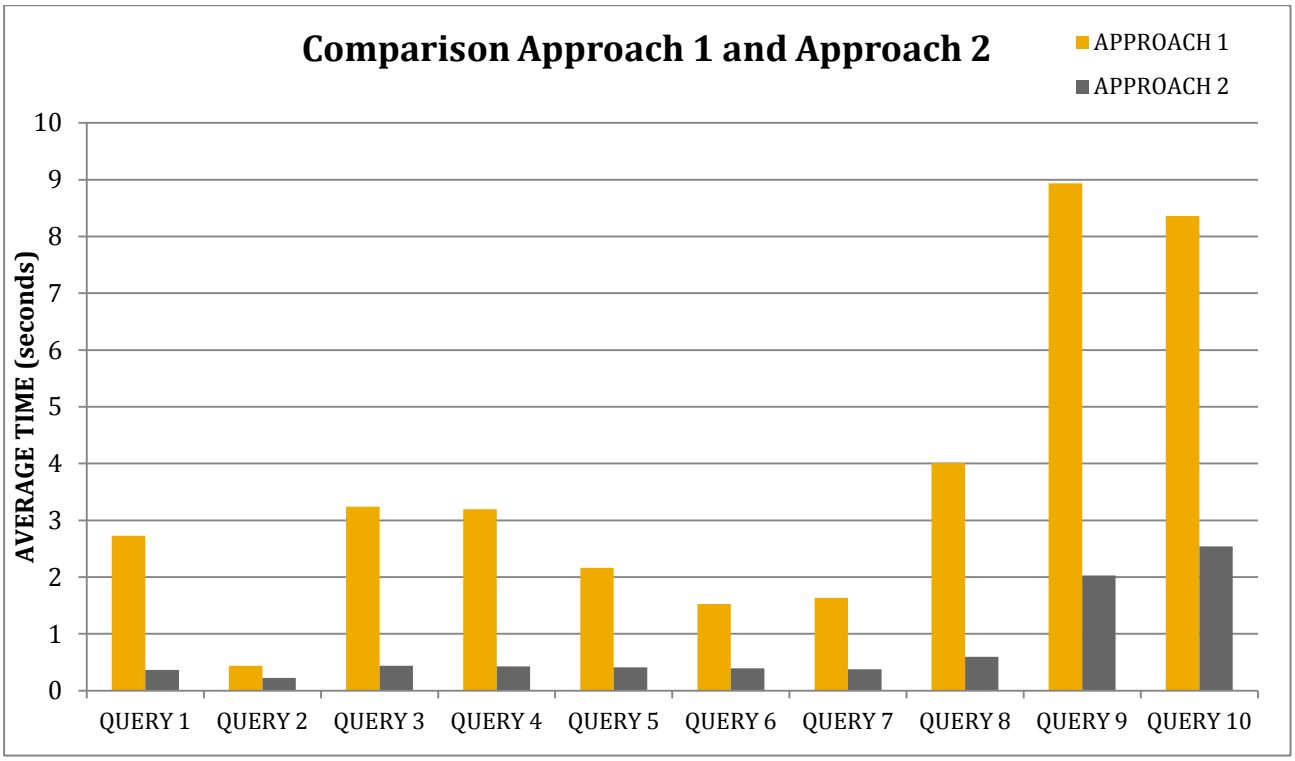

Figure 23: Comparison between Approach 1 and Approach 2, average time

As it can be clearly seen, the second approach shows better average time for all queries and therefore a better performance. Dictionary and index encoding of the second approach shows a better performance: having an index in a file allows reading a greater amount of data since the elements are integers and not characters, the time to translate these integers to strings of characters has a minimum influence in the overall time.

The compression or encoding technique also produces a smaller number of read operations, i.e. the number of read operations required to read the entire file is much smaller if the file to read is a file with an index (compression) than a file with strings of characters (no compression).

The sorted dictionary encoding is also a benefit: retrieving a value from a sorted dictionary speeds up the lookup process from *O(n)*, which means a full scan through the dictionary, to *O(log(n))*, because values in the dictionary can be found using the binary search method; while not having a dictionary requires a full scan operation through the complete column or file to find the value searched for.

The next graph shows the comparison between Approach 3: Row Store and Approach 4: PAX for all the queries quoted in the Evaluation Design section and used to test the performance. The times showed in this graph are the resultants of using the maximum buffer size, i.e. 1MB (1048576 bytes).

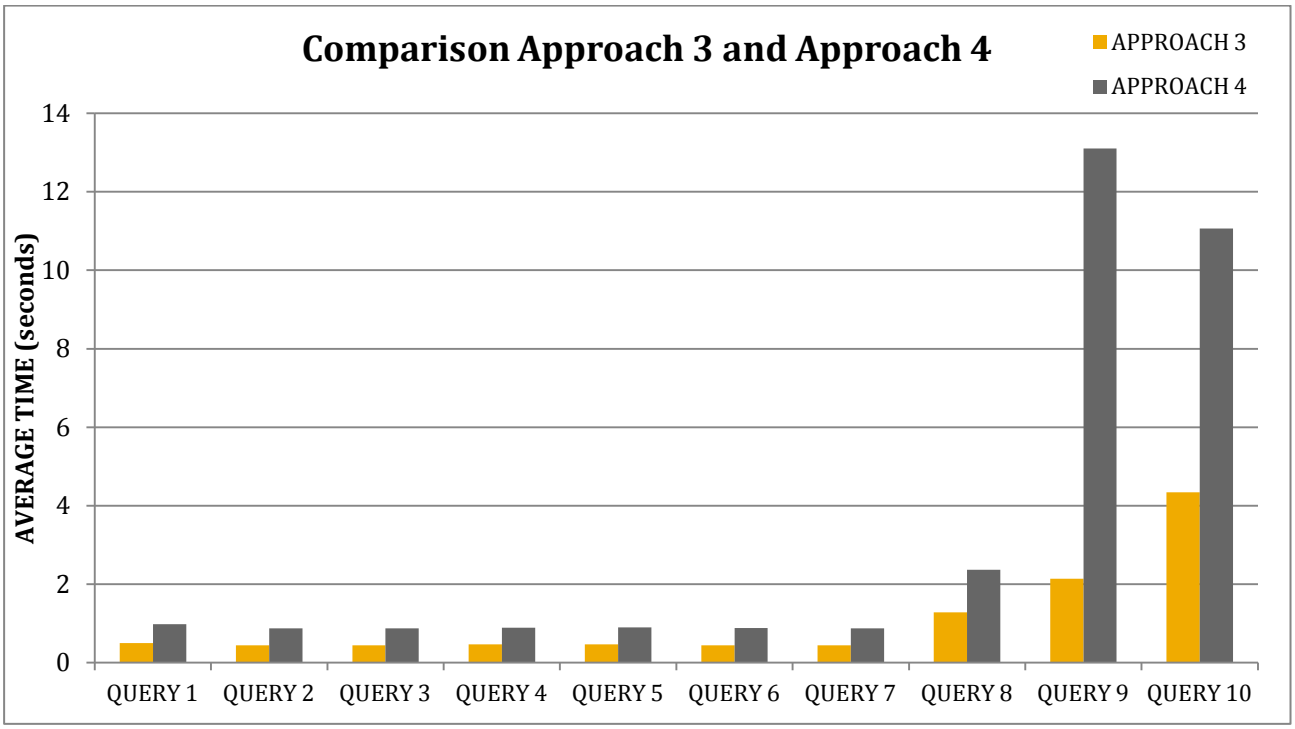

Figure 24: Comparison between Approach 3 and Approach 4, 1 MB buffer size

In this case Approach 3: Row Store has a better performance than Approach 4: PAX, but there is not a big time difference in the performance between the two approaches as in the previous comparison (Approach 1 vs. Approach 2). For all the queries except the last two, times in Approach 4 are multiplied by a factor of two compared to the times in Approach 3. The main reason for the better performance on Approach 3 is that the algorithm performs a smaller number of read operations since the file where the data is stored does not waste space.

There is also a big difference between the performance time in Query 9 and Query 10. These queries perform full scans of the table so it is logical to have a better performance if the data is stored in rows than if the data is stored in columns. PAX store the data in pages and data from a same column in mini pages and the number of items per mini page is given by the column that has the longest strings, so a lot of space is wasted in the rest of mini pages of the

same page. Read operations to retrieve all rows from the table, where the data is stored in this way, are very expensive.

The last graph is a comparison between all the approaches: Column Store, Column Store with compression, Row Store and PAX.

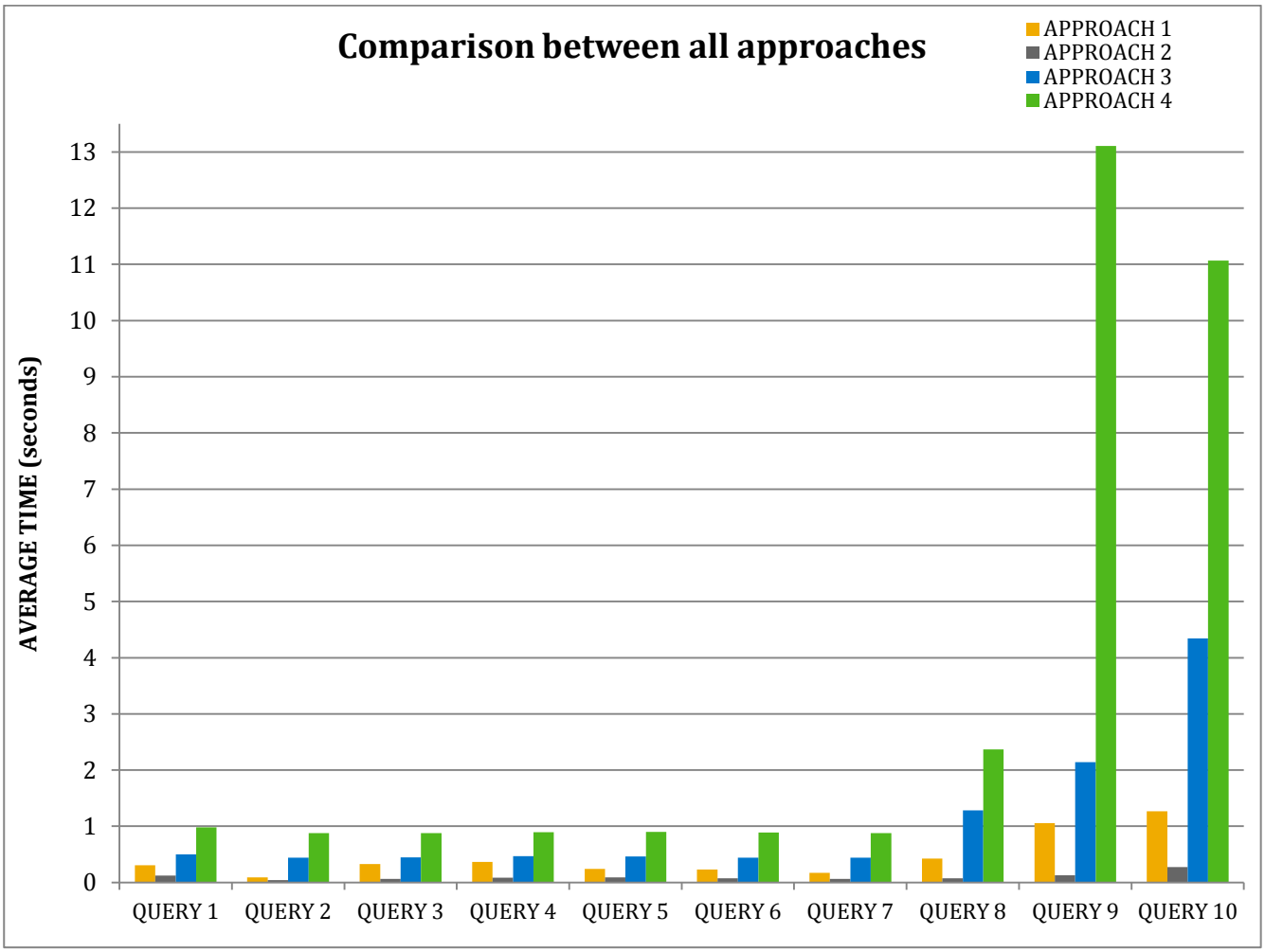

Figure 25: Comparison between all approaches, 1 MB buffer size

With a first look at the chart, the best approach is Approach 2: Column Store with compression. Its performance is superior in all queries, showing a difference of time with the rest of solutions. The evaluation of this chart is made in two parts: full scan queries over one column and full scan queries over the table:

- Full scan queries over one column: these are the first eight queries of the use cases. These queries are projections of one column and some of them have additional clauses that are restrictive and therefore, take more time to execute as it has been showed in this Evaluation chapter. If the queries are projections of one column, an approach based on the column store principle will show a better performance than an approach based on the row store approach. Hence, Approach 3: Row Store is not well suited for the first

eight queries. Approach 1 is neither well suited for these kind of queries since Approach 2 has a better performance with the same data model organization.

- Full scan queries over the table: these are covered and test in the queries of the last two use cases. These queries have to retrieve all the values in the columns for each row of the table and a data model organization based on the row store approach will performance better than other type of organization. But results contradict this hypothesis: Column Store with compression approach shows better performance than Row Store approach. An important thing to take into account is that the final result of these two queries is returned in a different way: in the Column Store with compression approach the table is returned with a column format, i.e. the table is a set of columns; while in the Row Store approach the table is returned with in row format. The result is the same but if the table has to be returned in rows, the second approach would need an additional step to transform or transpose the columns into rows. And this takes time but the estimation is that although this transposition algorithm has an impact in the performance it would not be large enough to meet the time resulting in Approach 3: Row Store provided that the workload is the same as used in this thesis. If the workload would be different, as for example, a major number of columns the impact would be larger and the performance could be the same for both approaches or even Approach 3 could be better than Approach 2. Approach 4: PAX shows large times for this kind of queries and, as it has been explained before, is due to the waste of space in the file and the high number of read operations to be performed.

For all the explained above, using the workload and queries of the Evaluation Design section, the best approach is Approach 2: Column Store with compression.

7. Conclusions and future work

Data in large volumes has to be handled by today's DBMS and it is kept in secondary storage, i.e. disk, but if the data would be stored in main memory, which is the primary persistence for data, this will lead to a different organization of data that only works if data is always available in memory. [1]

With this objective of a database that works in the primary persistence, SAP has developed a new relational DBMS: SAP HANA. It is an implementation of in-memory database technology designed for enterprise computing. And it is necessary to separate the data into cold data and hot data as an approach to handle the increasing, but still limited capacity of main memory efficiently while keeping all data available for reporting needs. Data in a database has its own life cycle and it can be separated into hot and cold states. The term cold data refers to data that has become passive and can be stored in disk. This data will not be changed any longer and will be accessed less often. [1]

We are working with a pure main memory database where data is stored column-wise. But of course, the data is stored on disk as well for durability reasons. Thus, this thesis is about to develop some mechanisms of streaming cold data in/out disk without using any kind of intermediate buffering that make the data stream as real as possible, directly from disk. Performance of these mechanisms is also tested and evaluated.

Four different mechanisms/approaches were designed, implemented and evaluated to store a table of cold data on disk:

Approach 1: Column Store. Data organization is done in a columnar way without any kind of compression. All the columns will be stored on disk as they are in the table. One binary file per column is created.

Approach 2: Column Store with compression. Data organization is done in a columnar way using some compression mechanisms: index and dictionary. Two binary files per column are created, one file is the index and the other is the dictionary.

Approach 3: Row Store. Row-wise organization. All the rows will be stored on disk as they are in the table. One binary file for the whole table is created.

Approach 4: PAX (Partition Attributes Across) version modified. Values for the same column are grouped together in mini pages; one page consists on many mini pages as many columns the table has.

Evaluating the performance and comparing of all these techniques, it can be conclude that Row Store approach is not well suited for operations where all the values or some values of the column have to be retrieved since its data model is row-wise organized and many read operations are necessary. Modified version of PAX also produces a poor performance with these types of operations because a lot of space is wasted in the binary file when there is a big difference in the length of the elements stored (~22 characters). Column Store approach generates good performance because the whole column can be read with less read operations than the previous approaches; however when the column has larger length values, as mentioned above, many read operations are necessary. Column Store with compression has the best performance comparing with the rest of approaches. Using the compression technique of index and dictionary makes the number of read operations, required to read a column, much lower, since it must read numbers of fixed length and not strings of characters of variable length. The process of "translation" to obtain the dictionary values that are equivalent to the index numbers is not expensive and does not affect the overall performance.

Compression techniques produce good results; hence another kind of indexes could be used as future work to increase the performance: inverted indexes. These indexes optimize the speed of the query: querying in a forward index requires sequential iteration through all the elements to verify a matching value; nevertheless, querying in an inverted index the query can be resolved by jumping to the value id (via random access). [23]

Future work must build upon the results gathered during the evaluation of the approaches, to complete the border cases and to improve the performance should be two very desirable goals.

When dealing with cold data, where typical operations have less impact or barely exist, such as delete and when the necessity is to limit the solution field to this kind of data, there are some real world examples than can be overlooked, like deleting or updating records. But some operations get a more important role. An objective for future work is to include a bigger subset of conditions for retrieval of data such as other operators (or, <, >), etc.

The results obtained are very dependent on the implementation of the algorithms and the data used to test those algorithms, hence one the first steps to fulfil these goals would be to create new data tables. These tables would have a higher number of columns, with a different distribution of data, longer values, empty columns, etc. The best possible scenario would be to use real data from a real application. Developed algorithms also allow multiple solutions for the same problem and the implementation used for the evaluation of the objectives and the inclusion of data models in each approach fulfilled a two part role: performance and simplicity. These features were accomplished primarily by using the native implementation of the read/write libraries that Windows provides within their operative system and the algorithms were written trying to maintain the readability; by sacrificing this aspect, a more efficient code could be developed, using different data structures to represent the data model that use code tuned to the needs of the application, use dynamic templates, etc. There is also a lot of improvement that can be done when tuning the algorithms for specific data such use compiler optimizations to gain prized CPU cycles.

The evaluation was done on a single machine with a fixed hardware and software setup. Although this setup was powerful enough, current hardware trends dictate that is possible that this type of labour, storing data and processing it from permanent storage, will be accomplished with systems that use a faster, more expensive and scarcer technology as SSD technology. It would be very beneficial to include this setup in the evaluation of future work.

There was also one limitation to the project, due to time constraints. This was to use a single thread execution to perform the algorithms; by including the segmentation to parallel processes, a higher performance could be achieved than by limiting it to one unique process. This also impacts on the available resources for this computation.

8. References

[1] Hasso Plattner, Alexander Zeier. In-Memory Data Management: Technology and Applications. Springer, 2012.

[2] Wikipedia. SAP HANA (online): http://en.wikipedia.org/wiki/SAP_HANA

[3]Wikipedia. Rock's law (online): http://en.wikipedia.org/wiki/Rock%27s_law

[4] Peter A. Boncz, Stefan Manegold and Martin L. Kersten. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. In *PVLDB*, 2009.

[5] M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran and S. B. Zdonik. C-Store: A Column-Oriented DBMS. *VLDB*, 2005 (online): http://people.csail.mit.edu/tdanford/6830papers/stonebraker-cstore.pdf

[6] Sybase. Product Sybase IQ (online): http://www.sybase.com/products/datawarehousing/sybaseiq

[7] W. H. Inmon, "What is a Data Warehouse?" Prism Tech Topic, Vol. 1, No. 1, 1995.

[8] Clark D. French. One Size Fits All Database Architectures. Do Not Work for DSS. *SIGMOD*, 1995 (online): http://lambda.csail.mit.edu/~chet/papers/others/f/french/french95sigmod.pdf

[9] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. *SIGMOD*, 2006 (online): http://db.lcs.mit.edu/projects/cstore/abadisigmod06.pdf

[10] Hasso Plattner. A Common Database Approach for OLTP and OLAP using an In-Memory Column Database. *SIGMOD*, 2009 (online): http://www.sigmod09.org/images/sigmod1ktp-plattner.pdf

[11] "Definition: in-memory database." Whatls.com, August 2012 (online): http://whatis.techtarget.com/definition/in-memory-database

[12] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database – The End of a Column Store Myth. *SIGMOD*, 2012.

[13] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, Jan Schaffner. SIMD-Scan: Ultra-Fast in-Memory Table Scan using on-Chip Vector Processing Units. *VLDB*, 2009 (online): http://www.vldb.org/pvldb/2/vldb09-327.pdf

[14] Marcus Paradies, Christian Lemke, Hasso Plattner, Wolfgang Lehner, Kai-Uwe Sattler, Alexander Zeier, Jens Krueger. How to Juggle Columns: An Entropy-Based Approach for Table Compression. *IDEAS10*, 2010.

[15] Anja Bog, Kai Sachs, Alexander Zeier, Hasso Plattner. Normallization in a Mixed OLTP and OLAP Workload Scenario (online): http://www.dvs.tu-darmstadt.de/publications/pdf/TPCTCpaper.pdf

[16] C++ Windows function CreateFile (online): http://msdn.microsoft.com/en-us/library/aa914735.aspx

[17] Returning values by value, reference and address. Learncpp.com (online): http://www.learncpp.com/cpp-tutorial/74a-returning-values-by-value-reference-and-address/

[18] C++ Windows function ReadFile (online): http://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx

[19] File Buffering in Windows (online): http://msdn.microsoft.com/en-us/library/windows/desktop/cc644950(v=vs.85).aspx

[20] Boost library for timers (online): http://www.boost.org/doc/libs/1_51_0/libs/timer/doc/index.html

[21] R. Ramakrishnan and J. Gehrke. Database Management Systems. McGraw-Hill, 2000.

[22] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, Marios Skounakis. Weaving Relations for Cache Performance. *VLDB*, 2001 (online): http://www.vldb.org/conf/2001/P169.pdf

[23] G. P. Copeland and S. F. Khoshafian. A Decomposition Storage Model. *SIGMOD*, 1985.

[24] Wikipedia. Inverted indexes (online): http://en.wikipedia.org/wiki/Inverted_index

9. Graphics Appendix

This will present more results obtained for each implemented approach due to completeness reasons and that have not been showed in the Evaluation chapter.

9.1.Approach 1: Column Store graphs

This section contains evaluation graphics for the Column Store approach.

The first two charts show the first two queries individually and that were explained in the Evaluation chapter making a comparison between them.

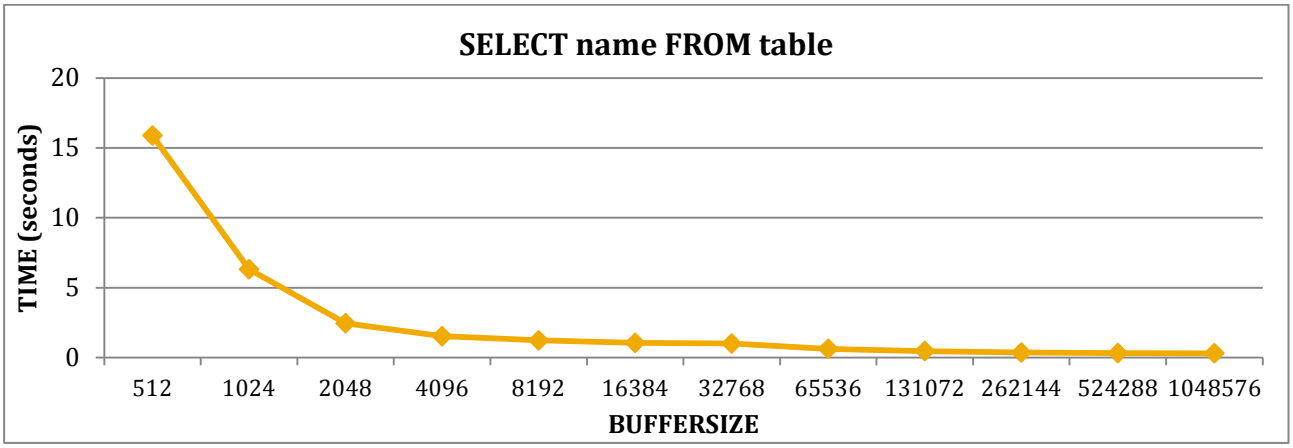

Figure 26: Query 1 performance

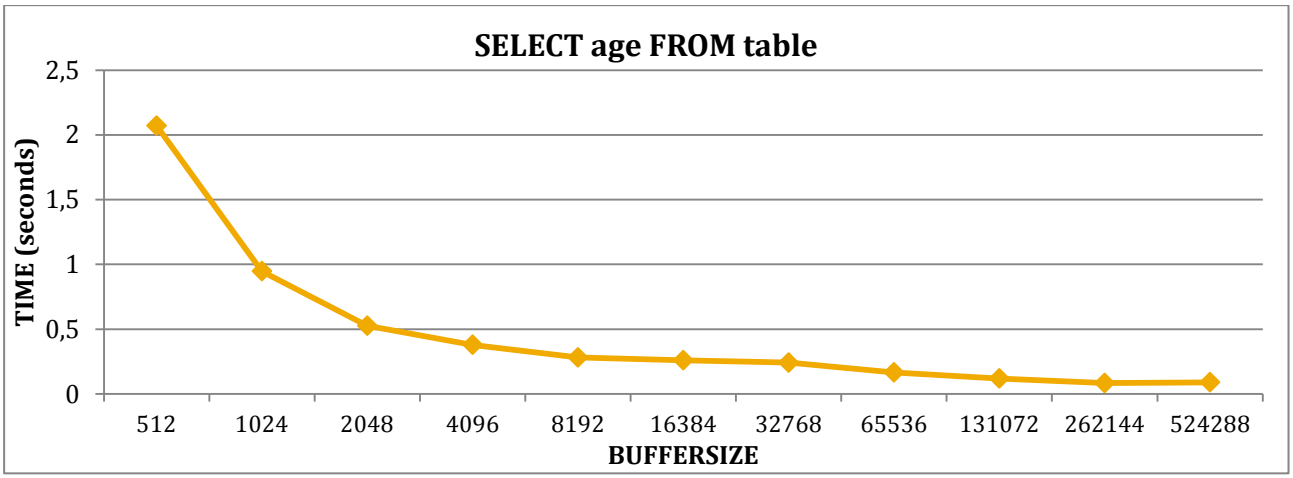

Figure 27: Query 2 performance

The next graph shows the performance of Query 3. This query is a projection with a selection, where all the values of the column *name* that fulfil the condition of having an *age* of *16*, must be returned. This entire operation take more time than a simple projection since the age column has to be read completely in order to find the row indexes where the value *16* is contained and then read the complete name column to get the values that fulfil the condition described by the projection.

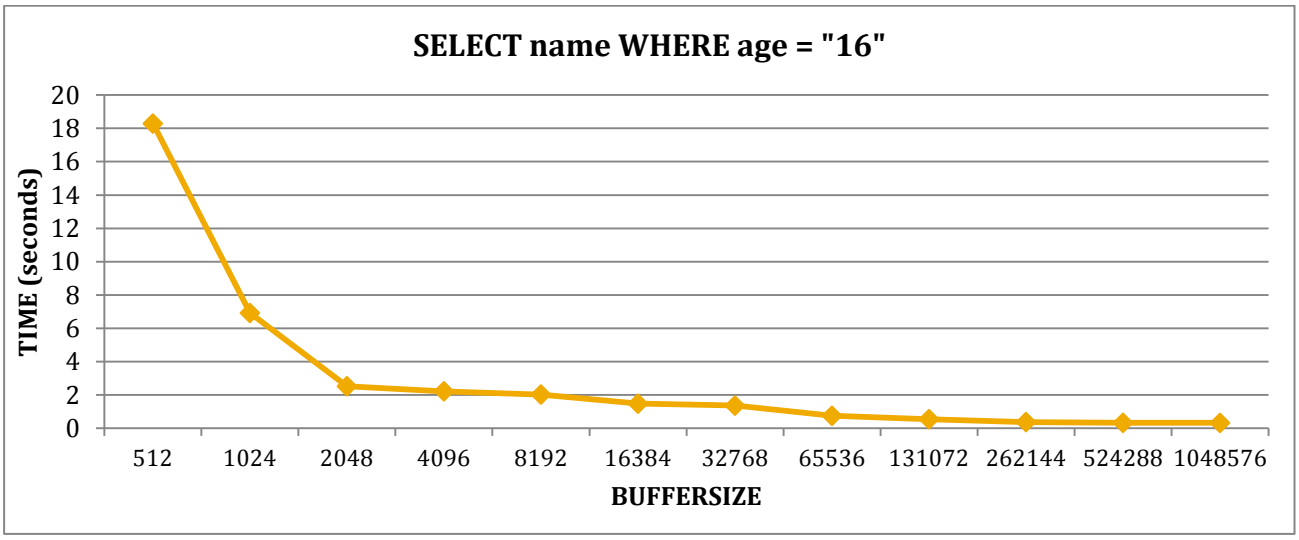

Figure 28: Query 3 performance

Next query is also a projection over a selective query; the column *name* is retrieved and materialized in order to know which of its values has *department = Human Resources*.

An obvious result of this query compared to the last one, it could be a longer time with every buffer size given that it is more expensive to materialize the strings of the *department* column than the strings of the *age* column. Following, the graphic for the fourth query is showed.

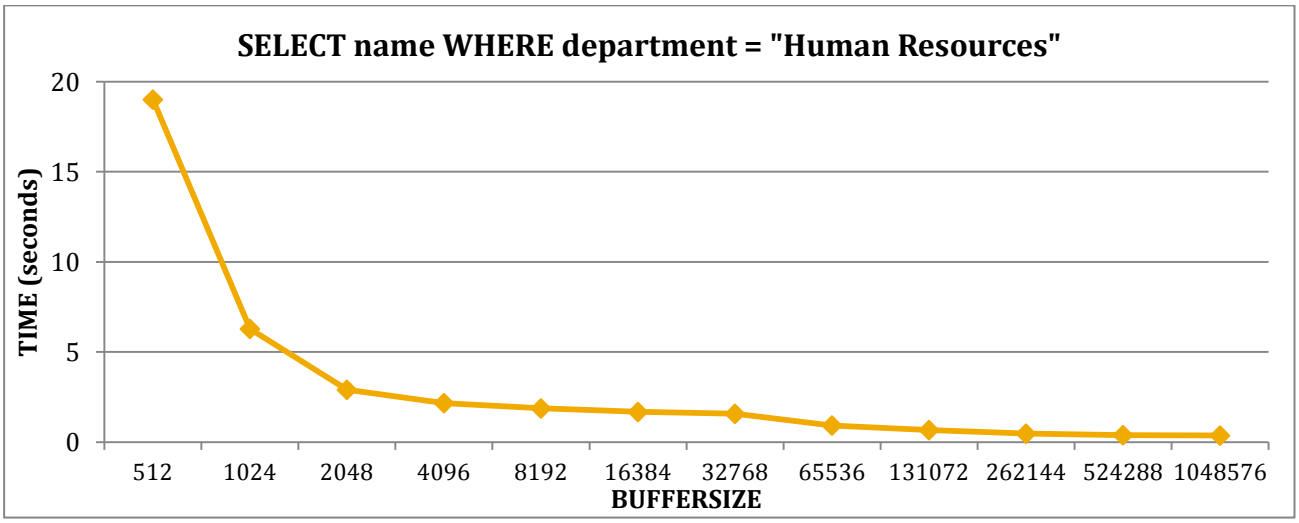

Figure 29: Query 4 performance

It can be perceived that the general trend is that the time it takes to execute the queries is greatly reduced once the read buffer size is 1024 bytes. That is, the time it takes to read and process the data chunks decreases significantly once the buffer size is 1024 bytes. The number of necessary readings cannot decrease anymore; hence the improvements obtained increasing the buffer sizes are only reflected in a little portion of the total time. This trend appears in all the next queries.

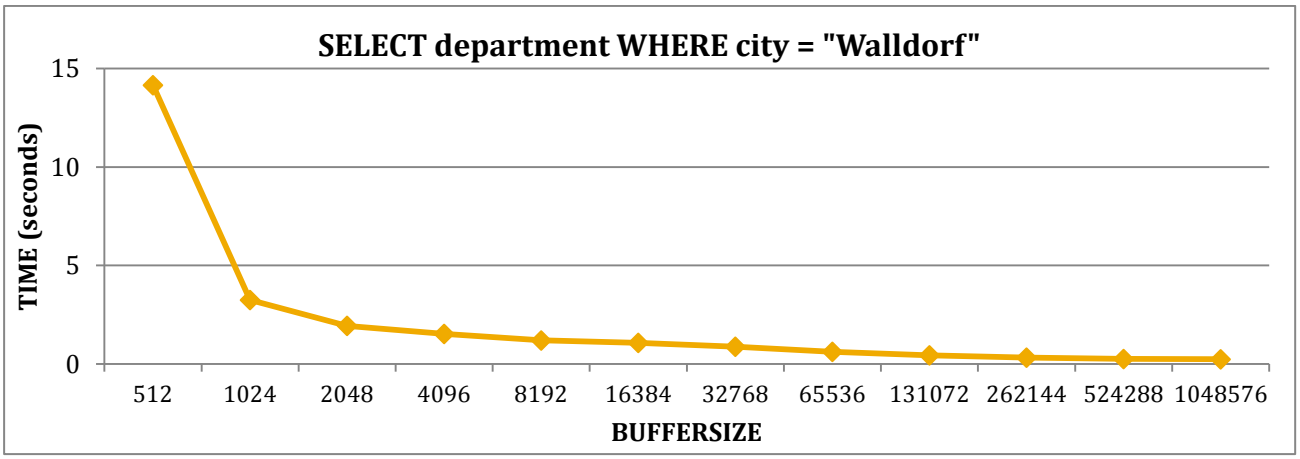

Figure 30: Query 5 performance

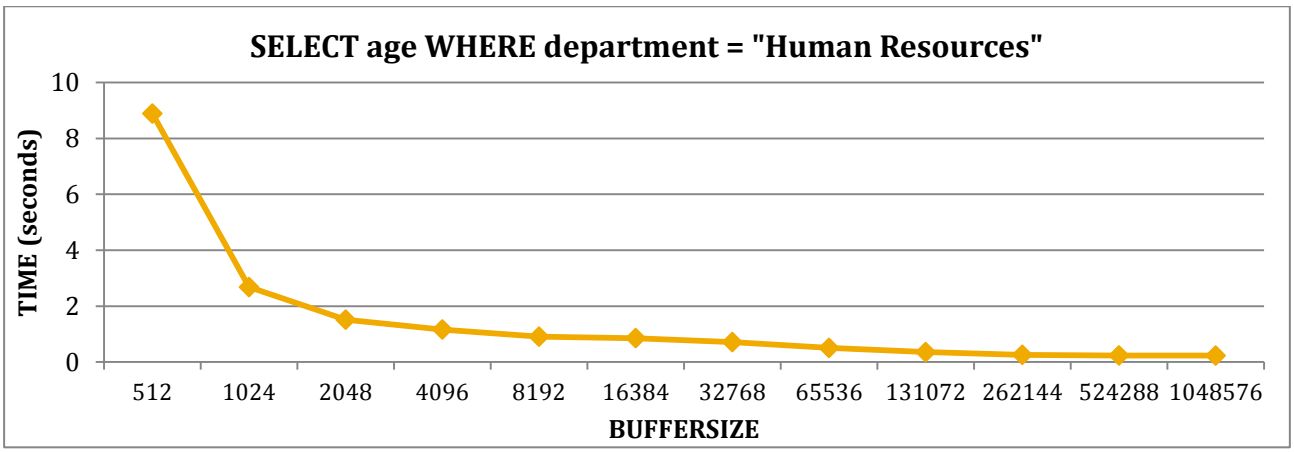

Figure 31: Query 6 performance

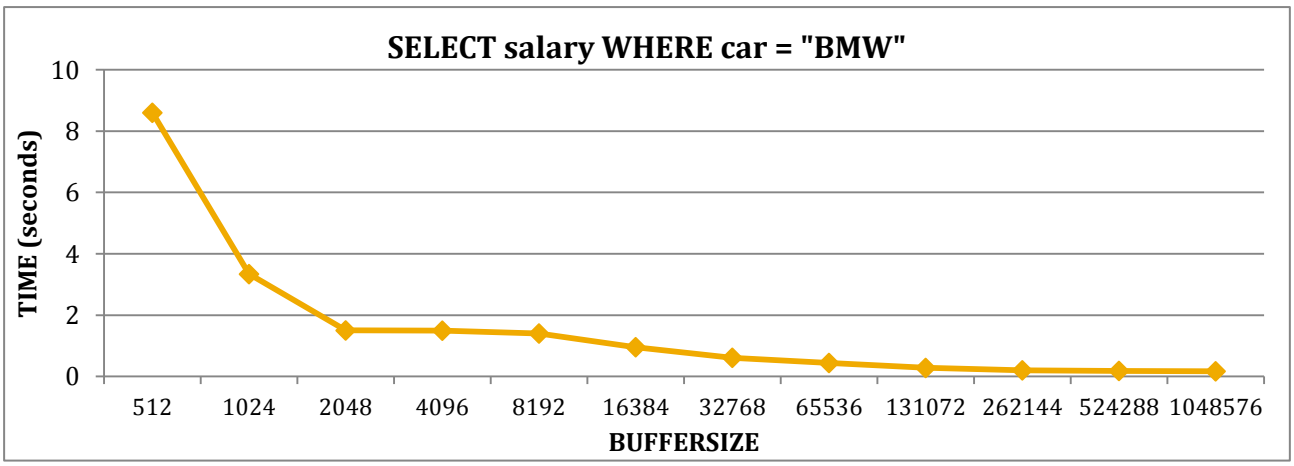

Figure 32: Query 7 performance

Another query graphic shows here one more projection with a selection clause but this time, the selection clause is composed by two different selections. As the other queries where a selection and a projection were performed, its time decreases so fast with the first buffer sizes while the rest of the buffer sizes almost do not make any difference in the final time result.

Having two conditions in the "where" clause would imply more time and worse performance since both conditions has to be tested and then produce a common result where the column *name* fulfil both conditions.

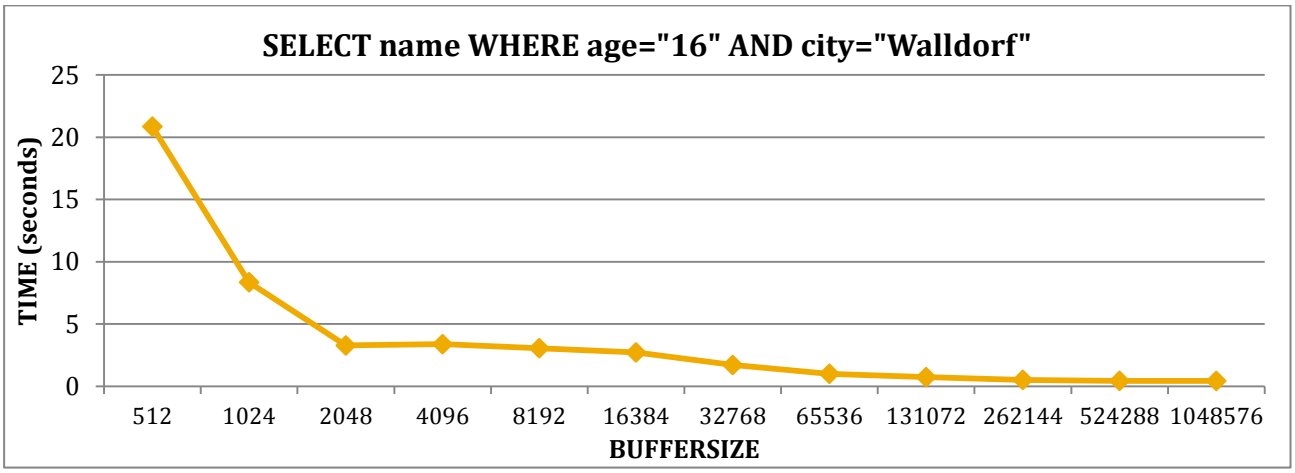

Figure 33: Query 8 performance

The next graphic shows a full scan of the table with a double selection clause. This means that all the columns of the table have to be retrieved and materialized provided that their values fulfil both conditions.

Evidently, having two conditions the execution of the query will take more time than a regular full scan of the table. If the previous results of selections are observed, the expected result for this query could be a slope curve from 512 bytes to 2048 or 4096 bytes with a minor decrease for the next buffer sizes.

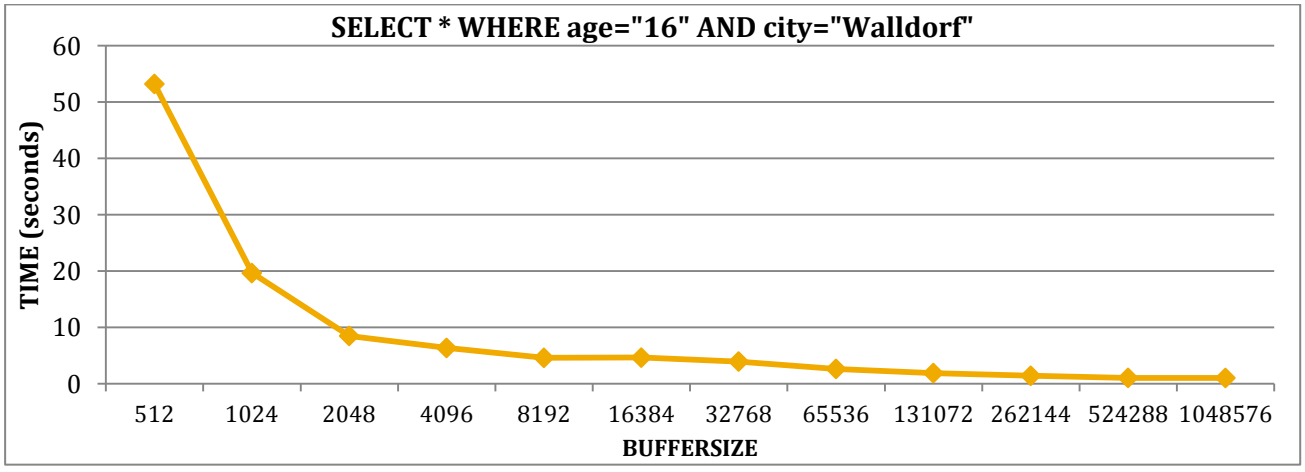

Figure 34: Query 9 performance

The last query tested is the table full scan where all the columns are retrieved and materialized.

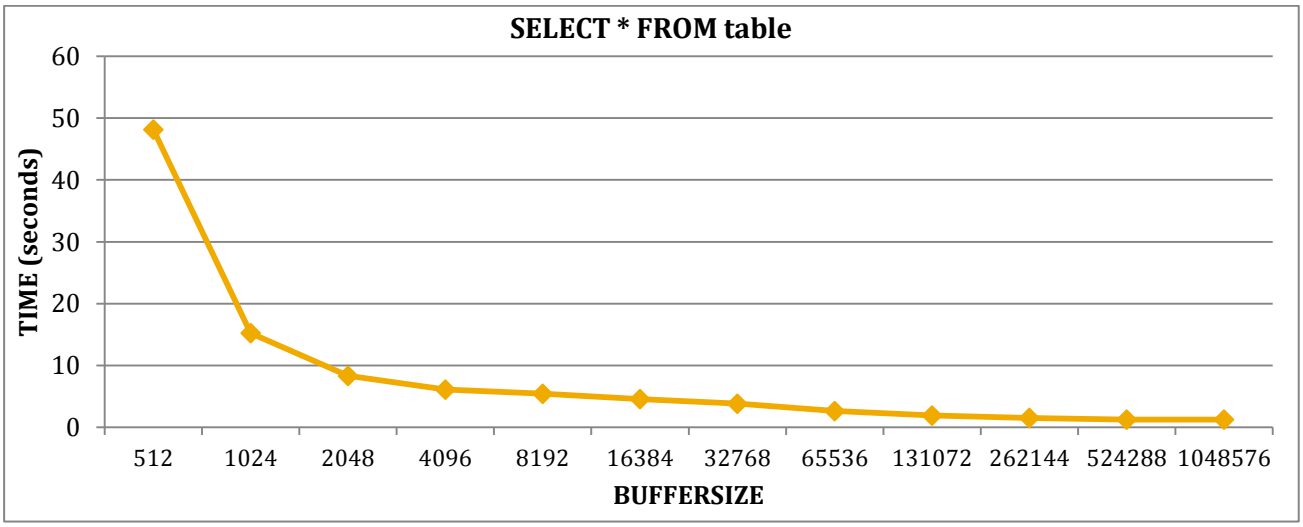

Figure 35: Query 10 performance

9.2.Approach 2: Column Store with compression graphs

This section contains evaluation graphics for the Column Store with compression approach.

The first two graphs show the first two queries individually and that were explained in the Evaluation chapter making a comparison between them.

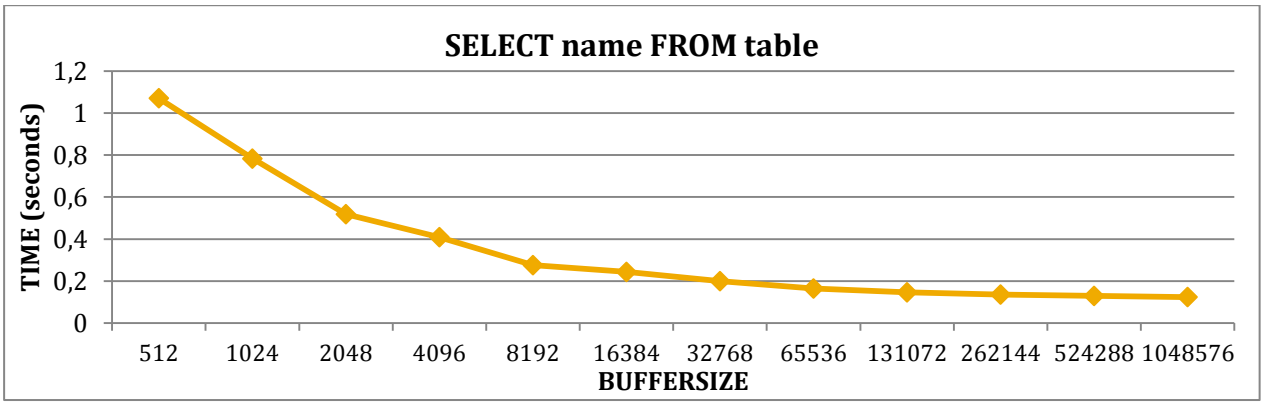

Figure 36: Query 1 performance

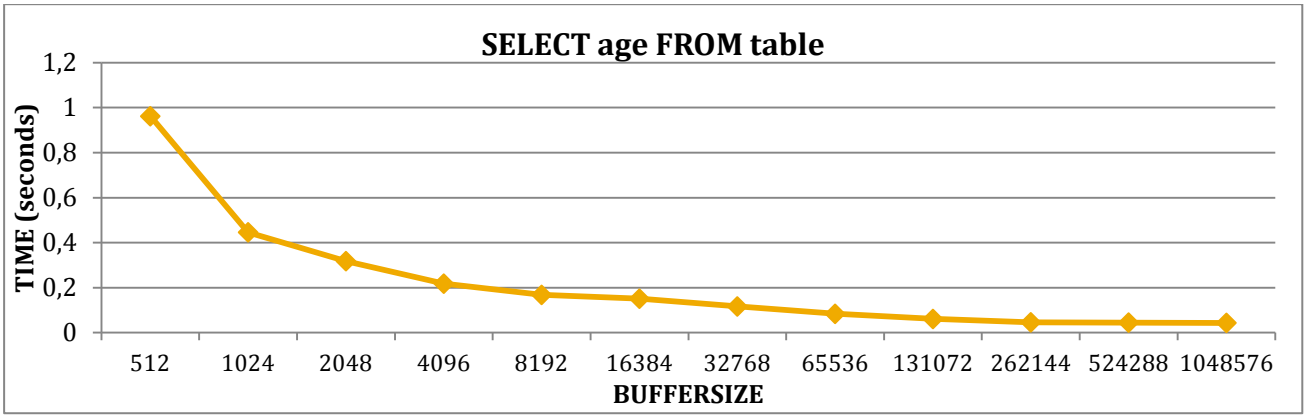

Figure 37: Query 2 performance

The next graph shows the performance of Query 3. This query is a projection with a selection, where all the values of the column *name* that fulfil the condition of having an *age* of *16*, must be returned. This entire operation take more time than a simple projection since the age column has to be read completely in order to find the row indexes where the value *16* is contained and then read the complete name column to get the values that fulfil the condition described by the projection.

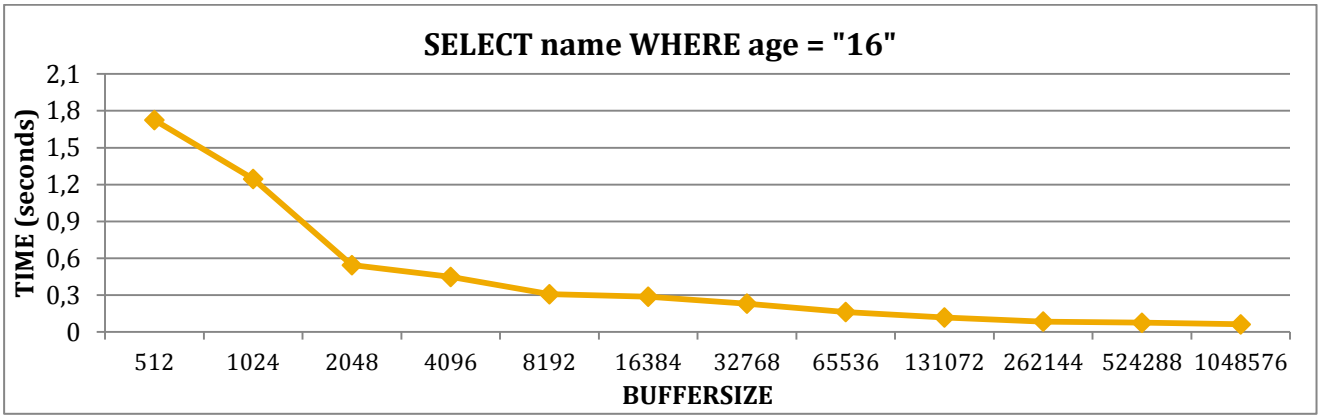

Figure 38: Query 3 performance

Next query is also a projection over a selective query; the column *name* is retrieved and materialized in order to know which of its values has *department = Human Resources*.

An obvious result of this query compared to the last one, it could be a longer time with every buffer size given that it is more expensive to materialize the strings of the *department* column than the strings of the *age* column. Following, the graphic for the fourth query is showed.

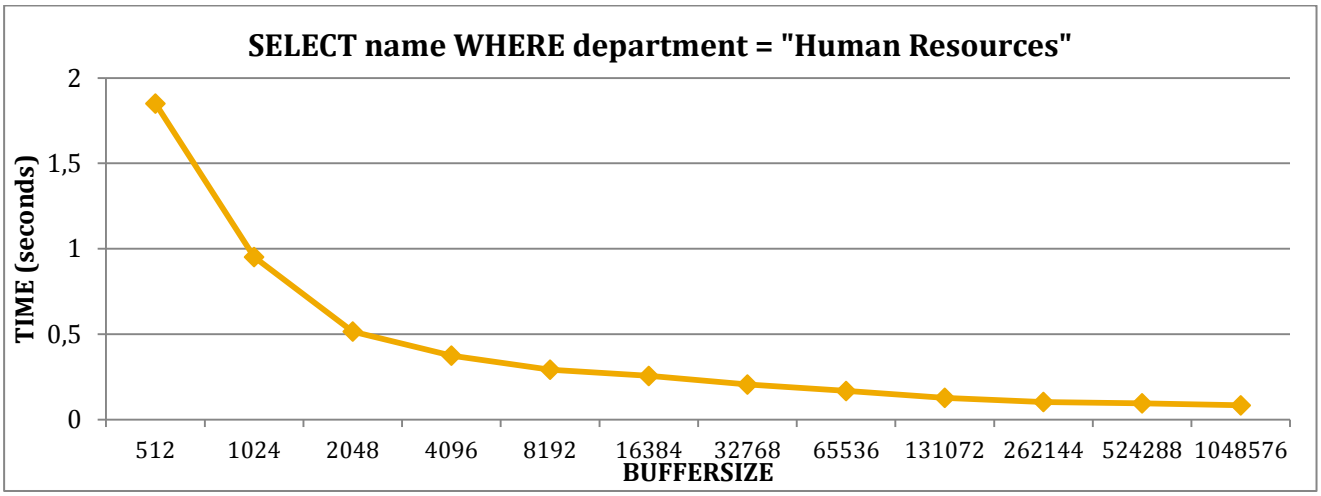

Figure 39: Query 4 performance

As expected, Query 4 has a little higher result than Query 3 since it is more expensive to materialize large strings.

If these Query 4 and Query 6 are compared, the expected result would be two curves of the same shape being the curve of Query 6 under the first curve since it would have better performance times due to the speedup in the materialization of the data: it is faster to decode 2 bytes long strings corresponding to the column *age* than strings of 22 bytes long in average of the column *name*. The comparison between these two queries is showed here:

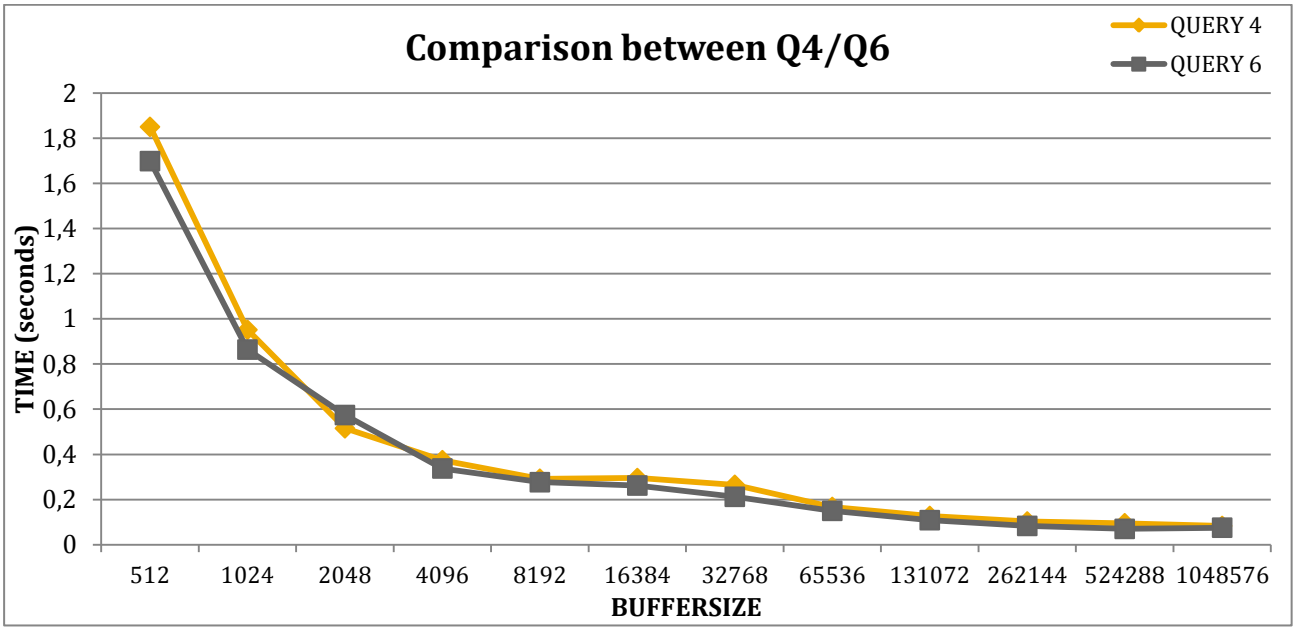

Figure 40: Comparison between Query 4 and Query 6

This graph shows the comparison between two queries with the same selection clause but a different projection. Both curves shows almost the same shape; the reason is that although the time of reading the dictionaries and index of each selected column are meaningful and decreases as the buffer size increases, the time is determined by the processing time of the "where" clause (reading index and dictionary and find the positions where the value appears).

The fact here is that the queries have to make readings for the selection or "where" clause and for the projection clause and the bigger size of the strings of the column *name* over the column *age*, does not have much impact in the overall time when the result of the query has less values than a full scan or is not selecting all the values.

Here it is the graphic for Query 5, where the column *department* is retrieved and materialized. Just like the previous two queries, this query has the same curve and similar times.

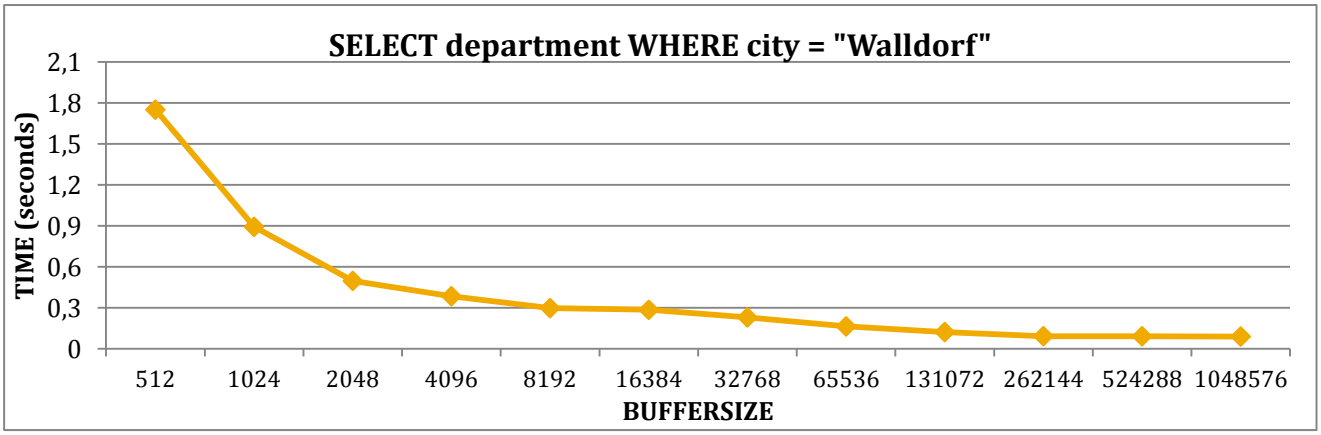

Figure 41: Query 5 performance

Query 6 is also a projection over a selective query. Its times are comparable to the last queries. The largest drop of time occurs between 512 and 1024 bytes, where the time is halved and keeps dropping until the end but in a milder form.

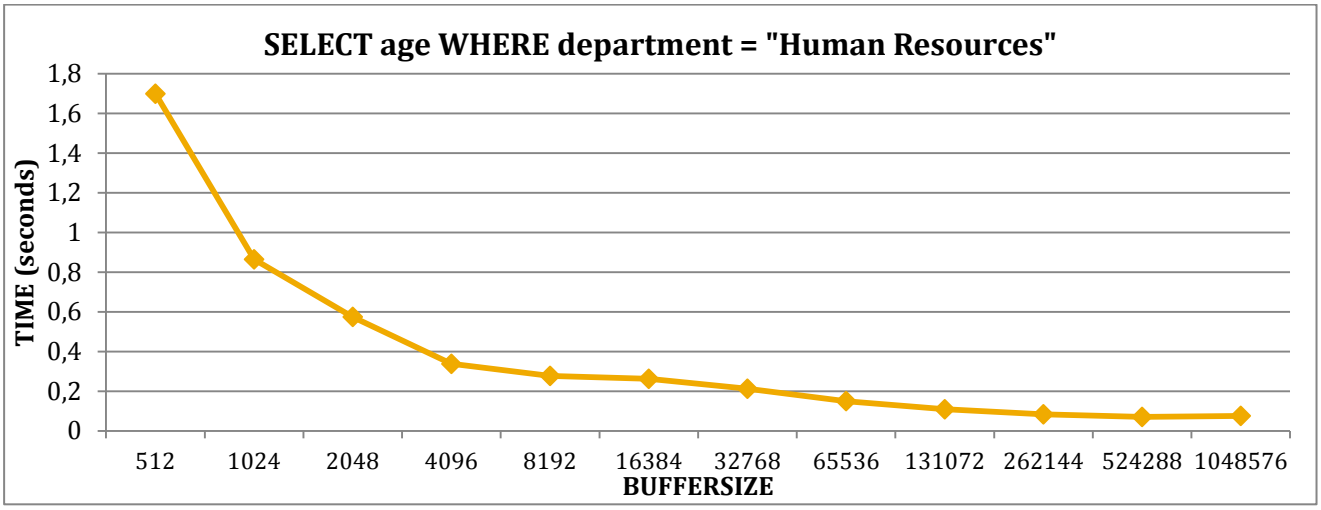

Figure 42: Query 6 performance

The next graphic shows another projection with a selection clause. This query has a fast decreasing in the first points due to the increasing buffer size; meanwhile with the other buffer sizes it has a more mild decrease in the times.

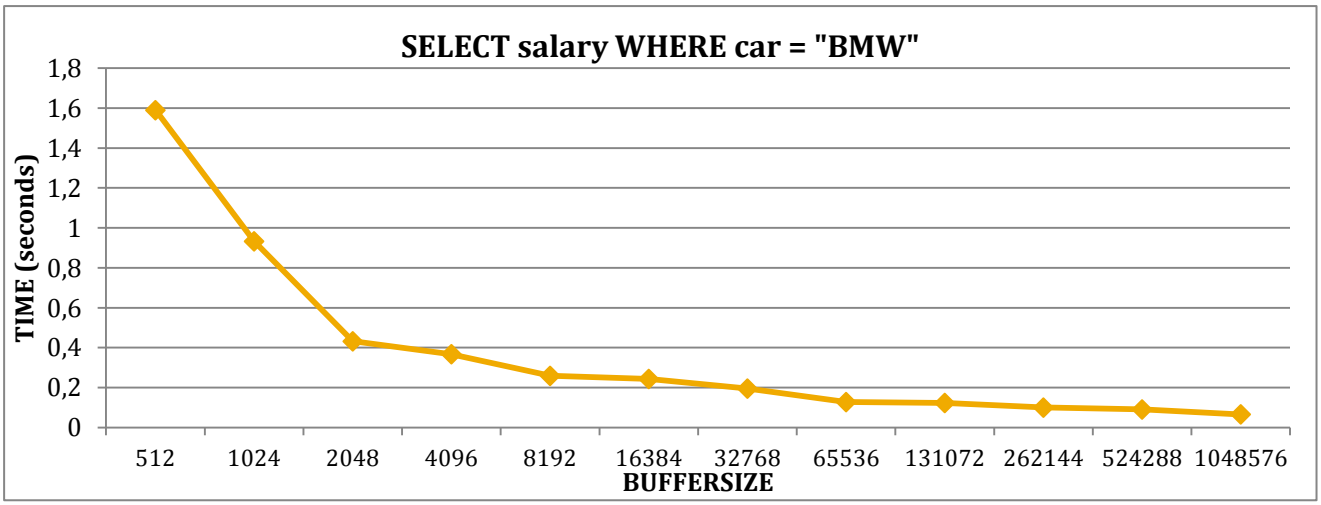

Figure 43: Query 7 performance

Another query graphic shows here another projection with a selection clause but this time, the selection clause is composed by two different selections. As the other queries where a selection and a projection were performed, its time decreases so fast with the first buffer sizes while the rest of the buffer sizes almost do not make any difference in the final time result.

Having two conditions in the "where" clause would imply more time since both conditions has to be tested and then produce a common result where the column *name* fulfil both conditions.

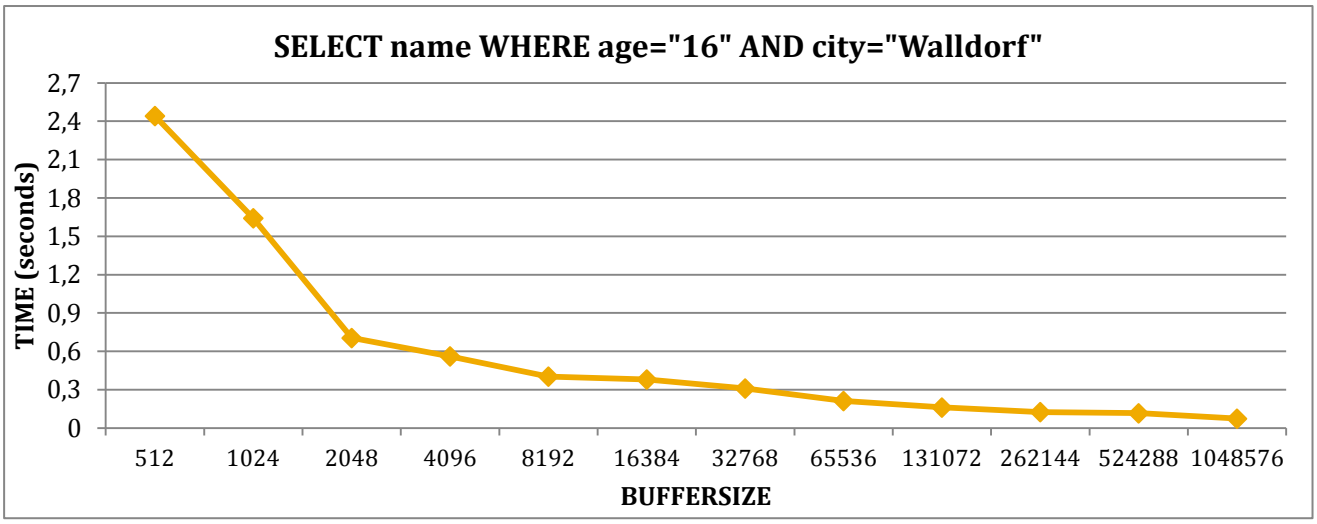

Figure 44: Query 8 performance

Actually the result is similar than the previous ones with this type of query, the only difference comes with the overall times. Whereas a query with a single condition takes 1.5 seconds in the worst case, a query with two conditions takes almost 2.5 seconds.

The next graphic shows a full scan of the table with a double selection clause. This means that all the columns of the table have to be retrieved and materialized provided that their values fulfil both conditions.

Evidently, having two conditions the execution of the query will take more time than a regular full scan of the table. If the previous results of selections are observed, the expected result for this query could be a slope curve from 512 bytes to 4096 bytes with a minor decrease for the next buffer sizes.

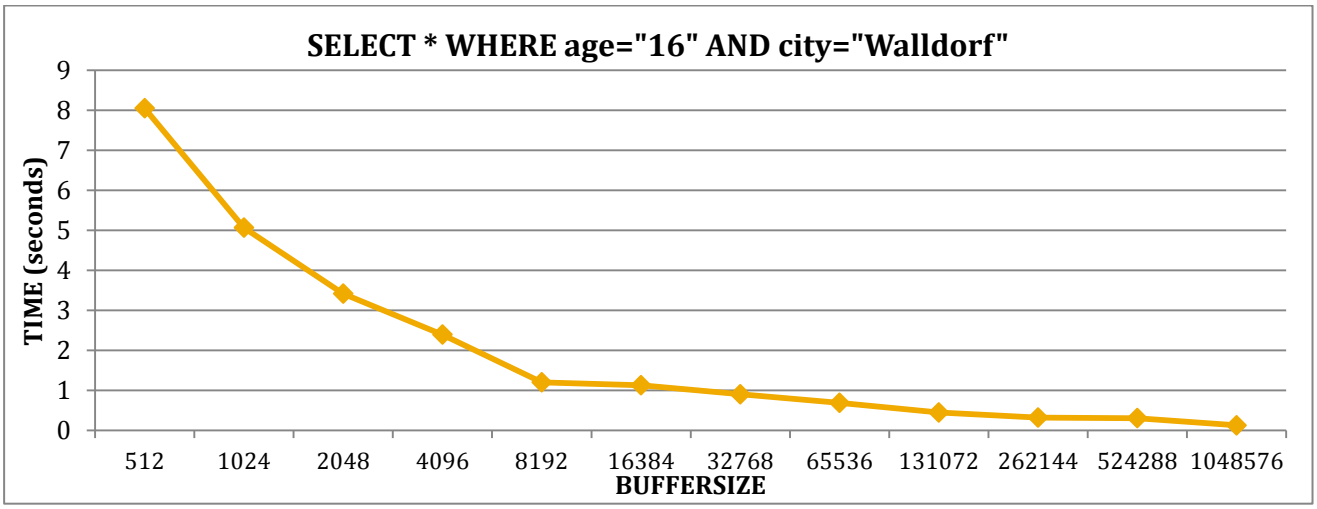

Figure 45: Query 9 performance

Again it takes a lot of time to read the indexes and dictionaries of columns age and city and obtain all the row indexes where those values appear (*age = 16* and *city = Walldorf*) and as the last step of the query get all the values that accomplish the two conditions for all the columns. It is the query that takes longer from the set of tested queries (~8.1 seconds) in the worst case (512 bytes of buffer).

Many read operations are necessary in order to read both indexes for the double selection clause along with the indexes of all columns for the projection clause; once the size of the buffer is 8192 bytes, the number of needed readings cannot decrease no more, hence the improvement obtaining increasing the buffer sizes are only reflected in a little portion of the total time.

The last query tested is the table full scan where all the columns are retrieved and materialized.

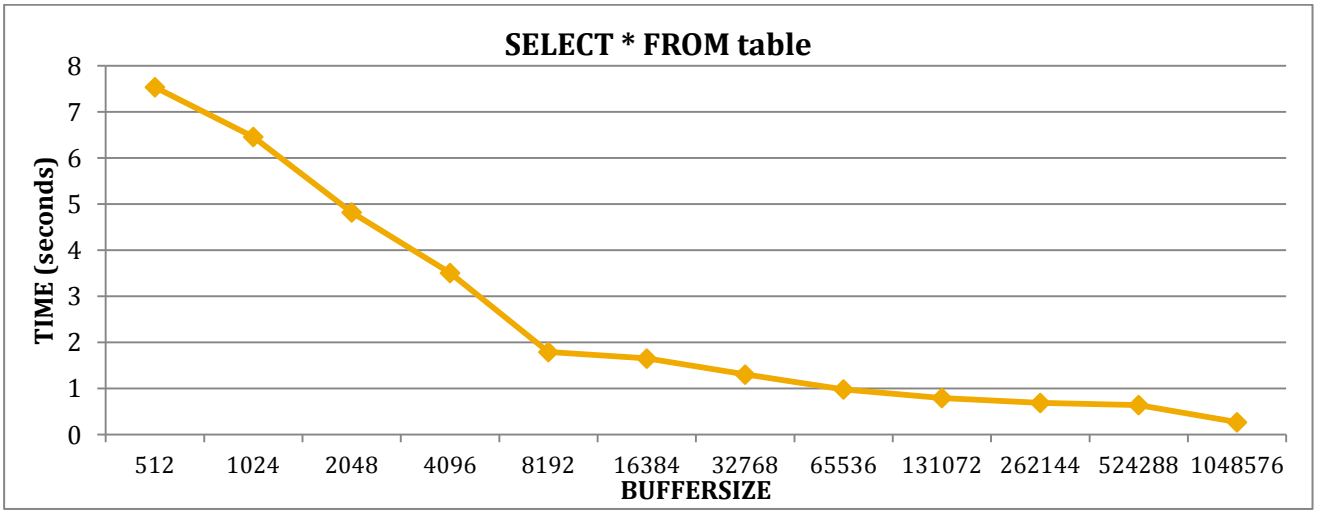

Figure 46: Query 10 performance

Next graph shows the comparison between Query 1 and Query 10, where the column name and all the columns are retrieved and materialized respectively.

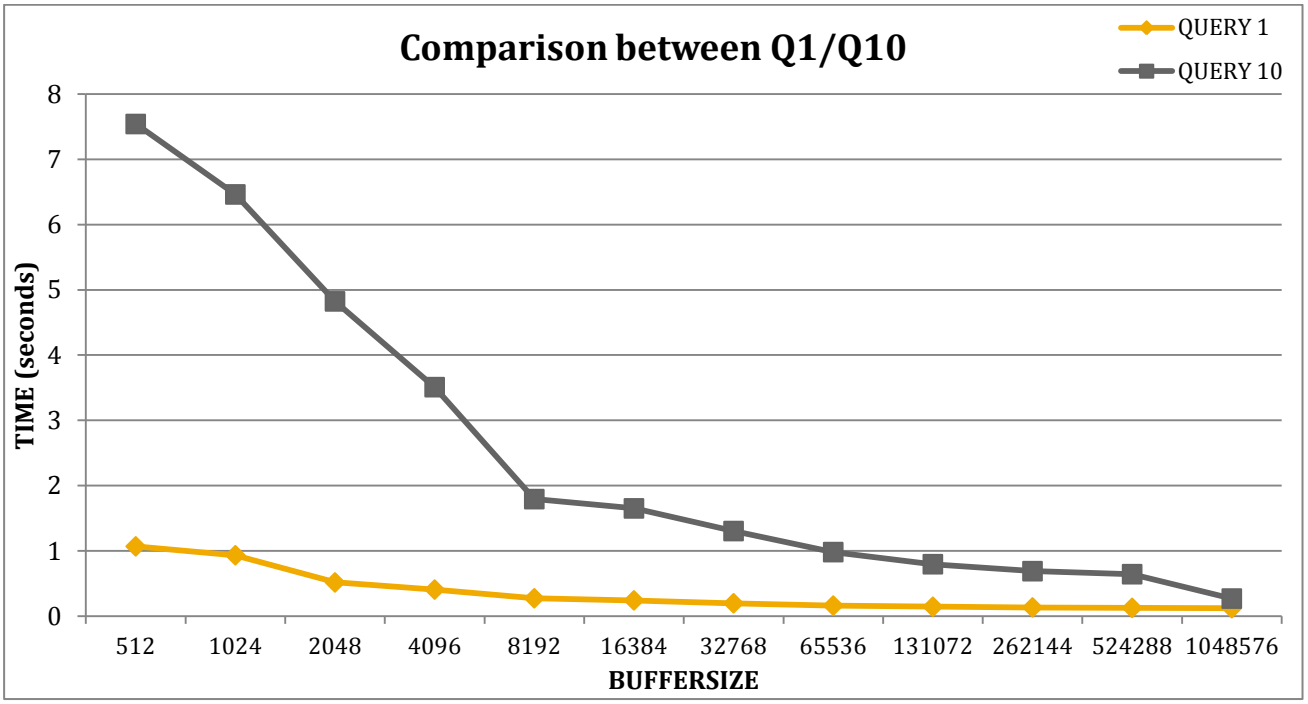

Figure 47: Comparison between Query 1 and Query 10

Logically, Query 1 takes less time than Query 10 because the first query only has to retrieve and materialize one column while the other query needs to retrieved and materializes all the columns.

The shape of Query 1 curve is the same as the curve of Query 10 but due to the scale it cannot be properly seen.

It makes sense that if retrieve and materialize the column with the longest strings takes more than a second in the worst case (512 bytes), retrieve and materialize 8 columns would take 8 seconds or less time and that is exactly the time of the query that performs the full scan in the table (7.54 seconds). But in the best case (1 MB = 1048576 bytes) the time for Query 10 is not more than eight times higher than in Query 1, but the double: Query 1 is 0.12 seconds while Query 10 is 0.27 seconds.