Column-based storage for analysis of high-frequency stock trading data

ABDALLAH HASSAN
Column-based storage for analysis of high-frequency stock trading data

ABDALLAH HASSAN

Master in Computer Science
Email: abdhas@kth.se
Date: July 1, 2019
Supervisor: Håkan Lane
Examiner: Elena Troubitsyna
School of Electrical Engineering and Computer Science
Host company: Nordic Growth Market (NGM)
Swedish title: Kolumnbaserad lagring för analys av högfrekvent börshandelsdata
Abstract

This study investigated the efficiency of the available open-source column-based storage formats with support for semi-flexible data in combination with query engines that support querying these formats. Two different formats were identified, Parquet and ORC, and both were tested in two different modes, un-compressed and compressed with the compression algorithm Snappy. They were tested by running two queries on the host company’s data converted to the appropriate formats, one simple averaging query and one more complicated with counts and filtering. The queries were run with two different query engines, Spark and Drill. They were also run on two dataset with different sizes to test scalability. The query execution time was recorded for each tested alternative. The results show that Snappy compressed formats always outperformed their non-compressed counterparts, and that Parquet was always faster than ORC. Drill performed faster on the simple query while Spark performed faster on the complex query. Drill also had the least increase in query execution time when the size of the dataset increased on both queries. The conclusion is that Parquet with Snappy is the storage format which gives the fastest execution times. However, both Spark and Drill have their own advantages as query engines.
Sammanfattning

# Contents

1 Introduction ................................................. 1  
   1.1 Motivation ............................................... 1  
   1.2 Research Question ..................................... 3  
   1.3 Contribution ............................................ 3  
   1.4 Delimitation ............................................ 3  

2 Background ................................................. 5  
   2.1 Data Warehouses ....................................... 5  
   2.2 Column-oriented Relational Databases ..................... 5  
      2.2.1 Disk Reads ....................................... 7  
      2.2.2 Disk Writes ..................................... 9  
      2.2.3 Compression And Encoding ......................... 9  
      2.2.4 Materialisation .................................. 11  
      2.2.5 Hardware Trends .................................. 12  
   2.3 Non-relational Databases ................................ 13  
   2.4 Nested Columnar Storage Formats ....................... 14  
      2.4.1 Dremel ........................................ 14  
      2.4.2 Parquet ......................................... 15  
      2.4.3 ORC ............................................... 16  
      2.4.4 Drill ........................................... 18  
      2.4.5 Spark ............................................ 18  
   2.5 Related research ...................................... 18  
      2.5.1 Dremel ........................................ 18  
      2.5.2 Parquet vs. PSV ................................... 19  

3 Methods .................................................... 20  
   3.1 Data .................................................. 20  
   3.2 System specification ................................... 21  
   3.3 Technologies .......................................... 21
3.4 Queries ............................................. 21
3.5 Measurements ..................................... 22

4 Results ........................................... 23
  4.1 Table sizes .................................. 23
  4.2 Query execution time ......................... 23

5 Discussion ....................................... 30
  5.1 Data size analysis ............................ 30
  5.2 Execution time analysis ....................... 30
  5.3 Sources of error .............................. 31
  5.4 Sustainability ................................. 32

6 Conclusions ..................................... 34
  6.1 Summary ....................................... 34
  6.2 Conclusions ................................... 35
  6.3 Future work .................................. 35

Bibliography ...................................... 37

A Query execution time results .................. 39
Chapter 1

Introduction

1.1 Motivation

Over the last few decades, the amount of data generated and stored in various organizations and corporations all over the world has increased considerably. Partially thanks to the increased use of electronic devices but also cheaper secondary storage, which has enabled the collection of vast amount of data, a trend which shows no signs of slowing down over the next years. It has been estimated that the total amount of data generated all over the world was over 1 Zettabytes (the equivalence of $10^9$ Terabytes) in 2010 and is projected to grow to 35 Zettabytes in 2020 [1].

Today, many websites log a large amount of data generated by user activity and the problem of how to store and analyze this data efficiently has gathered more attention. Traditionally, the main problem for most organizations was how to enable more and faster transactions to their database. Today and in the future the main challenge will be finding methods to analyze the logged data and discover how it can benefit the organizations and what value it can add to the business. At the same time, the growth of data is expected to outperform Moore’s law or any improvements in hardware performance over the next years, expanding the need for designing smarter infrastructure for storing and pipelining this data [1]. Here, the choice of technology for storing data in memory, such as the file format or how a query engine is optimized to retrieve data from secondary memory among other things, might prove essential to achieving this task. This thesis aims to investigate this angle of the problem.

A special case of the above-mentioned phenomenon can be observed in fi-
nancial markets, where the volume of generated trading data each day has increased significantly, not least thanks to the arrival of high-frequency trading (HFT). As of 2012, HFT was estimated to account for approximately 55% of US equity trading volume, 40% of the European markets and is growing in the Asian markets and other types of markets such as commodity trading and foreign exchange [2]. This has increased the need to efficiently store and analyze large amounts of trading data, but also to provide new opportunities to traders, who obtain these quantities of data at their disposal to uncover trends and patterns in it. A more specific example is the host company of this thesis, which run a trading system that generates around 5GB worth of data each day stored as text files. This translates to over one Terabyte per year, and parsing all this data to perform any computations takes several weeks, which can be considered too slow for most useful applications.

Today there are plenty of systems and frameworks which are intended for data analytics on large data sets, both commercial and open source. They all differ from each other significantly in how they were designed and implemented, each one consisting of different modules and combining a number of ideas and technologies into one fully working system. One part that has received a significant deal of attention is the storage system and the file format, basically how it organizes the data in memory and how it interacts with the query engine or the file reader. One particular set of technologies that has gained traction over the few last years is the so-called column-oriented databases, sometimes also called column-stores. These are characterized by their way of partitioning the table of a database by their columns, and storing each column separately instead of the usual method where the table is partitioned by rows and each row being stored as one atomic unit. The idea is to avoid unnecessary disk reads when we would like to fetch only one or few of the columns into main memory. It should, in theory, improve query execution time. However, column-stores introduce some overhead in performance, especially when the tuples are reconstructed from their divided parts or when new tuples are inserted into the database [3, 4]. Also, column-orientation naturally lends itself to the relational model of databases, where data is stored as two-dimensional tables. However, this model does not fit the need for all organizations, many of which prefer to use the semi-structured model in which the form of the data evolves and changes over time. New research has attempted to address these issues which has laid the groundwork for modern column-stores. One notable example is Google’s Dremel [5], which is the query engine used in their BigQuery service.
Dremel is notable for combining column-based storage with a flexible semi-structured data model. It has inspired a number of open-source projects such as Drill and Parquet which implemented a storage format similar to the one described in Dremel. All these projects have used slightly different approaches and differ from each other beyond the fact that they are all column-oriented and support semi-structured data. Hence we should expect them to perform differently from each other. This project aims to explore these differences and how they might affect the performance of the system, specifically when performing complex data analysis on trading data.

1.2 Research Question

The research question is as follows: Out of the current implementations of column-oriented storage formats with support for semi-structured data, which one gives the best performance in terms of query execution time and scalability in terms of larger data volumes when used to perform data analysis on trading data? Moreover, how can we explain these differences with respect to the implementation details of the systems?

1.3 Contribution

The objective of this thesis is to explore which of the existing column-stores is the better alternative for storing complex nested trading data for the purpose of data analysis. The purpose is to aid organizations who wish to migrate their data to a more efficient format but are unsure which of the alternatives might work best for them. Another purpose is to explore why some implementation may have performed more efficiently than the other, i.e. which are the main factors that affect real performance in a modern system. This could be used as a basis for future improvements in the design and implementation of future storage formats intended for storing trading data or other similar time-series data.

1.4 Delimitation

The main limitation of this work is that it investigates column-stores which allow for flexible semi-structured data and which could evolve over time. Another limitation is that the performance is considered with respect to complex
analysis of trading data where the queries scan large parts of the data to find patterns inside them. What is being investigated here could generalize to any kind of time-series data. Also, all tests will be performed on a single computer, so the results may not necessarily apply to cases in which large clusters of computers are used. When it comes to testing scalability, the tests look at how the systems scale with larger datasets rather than more nodes in a cluster.
Chapter 2

Background

2.1 Data Warehouses

In the theory of databases, it is common to distinguish between two kinds of query workloads, namely Online Transaction Processing (OLTP) and Online Analytic Processing (OLAP). OLTP operations are usually small and only need a small portion of the database, but they often modify the database and there is usually many of them running at the same time. Most online operations such as bank transactions are of this form. OLAP operations on the other hand tend to be large and complex, scanning a large section of the database and performing many aggregations, but without modifying the underlying data. OLAP queries are of analytical nature and are used by organizations to find useful patterns in the data. OLTP and OLAP operations are often performed on two separate copies of the database, because long-running OLAP queries may block OLTP queries thus delaying crucial transactions. Moreover, it makes it possible to optimize each database for their respective workload. The copy in which OLAP queries are performed on is called a data warehouse [6, pp. 456–458].

2.2 Column-oriented Relational Databases

A relational database is usually conceptualized as a two-dimensional table. The table consists of a number of rows, sometimes called records or tuples, which consist of the values of the columns, sometimes called attributes, of the relation. Each column has a unique attribute name and belongs to a simple type. However, the memory of a computer system whether it is main memory or secondary storage is managed as a one-dimensional array of bytes each with
a unique memory address. This means that when a relation is serialized in order to be stored in memory, it needs to be partitioned first either row-by-row or column-by-column, see figure 2.1 for example. In row-based partitioning, the values of each tuple will be stored consecutively in memory while in column-based partitioning, or vertical storage as it is sometimes called [7], the tuples are split so that each column is stored separately on disk as a consecutive unit [4, 3].

Figure 2.1: An example relation and the two different ways it can be partitioned
From a high-level point of view, row-based and column-based systems are no different from each other. It should be possible to execute the same query on both systems and retrieve the same result. However, performance-wise they might differ from each other and so the choice between the two is important in the design and implementation of storage formats. While Column-oriented systems have existed already in the 70s, row-oriented databases have been the standard up until recently. This is due to the fact that there exists many performance trade-offs between the two choices which were not always in favor of column-stores [4, 3]. These trade-offs are discussed below.

2.2.1 Disk Reads

Virtually all operating systems manage virtual memory in terms of pages, typically 4 to 64 Kilobytes. The virtual memory pages correspond to disk blocks on the hard disk. When transferring data between disk and main memory, it is done in units of pages. A key observation is that the speed of moving data between disk and main memory is slow compared to the CPU. If we need to access data from the disk to perform computations then the processor will be stalled waiting for the data to arrive. It is therefore beneficial to make as few disk accesses as possible when executing a query. A similar observation can be made about the time it takes to load data into the CPU cache from main memory, which is slower than when the CPU loads data directly from cache. This is sometimes referred to as the I/O model of computation. Generally speaking, it is beneficial to avoid moving data between the layers of the memory hierarchy, and keep data which is accessed frequently as close to the CPU as possible. However, most databases in use are much larger than the available main memory, making numerous and repeated disk reads unavoidable [6, pp. 548–557].

The time it takes to read or write a certain block on disk can be divided into three parts, each corresponding to an action taken by the disk controller, which is a processor that controls and plans the actions of a disk. The first one is the **seek time**, which is the time it takes for the head assembly to arrive at the track containing the block. Second is the **rotational latency**, which is the time it takes for the platters to rotate before the correct block arrives under the head. Lastly there is **transfer time**, which is the time it takes for the controller to read or write the data to the block. Note that both seek time and rotational latency depend on the position of the head relatively to the block; if the head is already positioned at the right position we eliminate both of them. If we access disk at random blocks then performance will naturally suffer due to repeated disk
seeks. A smarter way would be to position the data that we want to access in adjacent locations on disk and schedule the disk to read many contiguous blocks consecutively thus eliminating the need for many disk seeks. This pattern is called sequential access, as opposed to random access. Another technique used to speed up access is **prefetching**. The blocks that are needed soon but not immediately are fetched and buffered in main memory. The advantage is to schedule the disk better by reading as many adjacent blocks as possible [6, pp. 552–561]. This highlights the importance of organizing data in memory so that the segments which are usually accessed close in time should be stored adjacent in space.

DBMSs have attempted to solve this issue by introducing *indexes*. The idea is that if we only want to fetch a few particular tuples, instead of loading the whole database into main memory and searching for the tuples there, it would have been better to know beforehand which disk blocks contain these tuples and only load these blocks. A specific example is if we have a database of all citizens in a country, and we want to get the records of those who were born on a specific date, then we could create a small table associating each date with the disk blocks in the database that contains records with this date. This way, we could use this table to find out exactly which disk pages are needed to get the desired result. This is an example of an index. Indexes from an abstract point of view can be seen as a mapping from search values to pointers to memory locations which contain records matching the search values. Exactly how indexes are implemented varies widely across the different implementations but they all introduce some overhead by maintaining some data structure which needs to be stored and searched through when executing a query, so there is always some trade-off when using indexes. Also, to get an actual improvement in performance the data must be structured in such a way that the specific records we’re searching for are stored near each other. In our previous example, if records of people who are born on the same day are spread out over the whole database then we will not get an improvement since we would need to scan the whole database anyway. But if the records are sorted by birthdate beforehand then we would only need to scan a small portion of the database. This is why DBMSs which use indexes sometimes sort the relations by their primary key creating a so called **sequential file** [6, ch. 14].

The previous technique does not solve the case in which we need a few columns from a large number of tuples. If we want to compute the average age of all citizens, for example, we would need to scan through the whole database
even though we only need a small section of it, the birthdate column. Vertical
storage provides a simple solution to this problem. If the relation is partitioned
column-wise, then the disk only needs to read the pages which cover these
columns, and skip over all the other ones. Naturally, there are many factors
which could affect how much performance gain this technique gives, if there
is any at all. Some of these factors are related to the characteristics of the data,
most importantly the number of rows, number of columns and size of the rows
and attributes in bytes. Other factors are usually related to the characteristics
of the query such as how many columns it needs to read or the proportion of
tuples that are selected in the predicates, so called selectivity [7]. Naturally,
the fewer columns we select and the wider the tuple is in bytes, the better
performance we should expect from a column-store.

2.2.2 Disk Writes
In principle, disk writes are similar to disk reads in that we want to minimize
the number of disk writes performed. Disk writes are performed whenever
tuples are deleted, inserted or updated. Here, column-stores should perform
worse, since in typical OLTP applications we’re usually interested in updating
one or a few tuples in the relation. Rarely does the need to update one col-
umn for all tuples arise, and in a column-store we would need to perform a
disk seek for each delete or for each modified value. Furthermore, column-
stores usually compress the columns before storing them on disk. This makes
inserts, deletes and updates even slower since the data must potentially be de-
compressed before any changes are performed, and then compressed again
before storing it on disk. This means that column-stores, at least in theory, are
a better fit for read-intensive analytics-focused data warehouses but a worse fit
for update-heavy OLTP workloads [7, 4, 3].

2.2.3 Compression And Encoding
One technique that’s used to reduce disk accesses is compressing the data on
disk. The more we can compress the data, the more we can fit into one page
and the less pages we need to read into memory. If compression is used, we
need to decompress the data to operate on it and compress it again before stor-
ing it on disk, which takes additional time but it compensates for it by the
decrease in disk I/O. There are several compression schemes, some of which
give a higher compression-ratio at the cost of decompression time, while oth-
ers trade a faster decompression for lower compression ratio [8].
Previous research has shown that column-stores have an advantage over row-stores in the use of compression. Regardless of the choice of compression algorithm, it is possible to get a better compression on data with low information entropy. When data is stored column-wise then data of the same type and with similar values can be stored in close proximity. For example if there’s a column for storing the ages of people, since many people have the same age, there will be many repetitions in this column which the compression algorithm can take better advantage of. This is in comparison to if it tried to find similarities in different values in one tuple, such as age and name which are bound to be different and thus have high information entropy [8, 3].

This also means that the characteristics of the data in a column will have a great impact on how well it compresses. Certain instances of a column will achieve a better compression-ratio than others. Previous research has shown that sorted columns with many repeated values are the easiest to compress. They also have the advantage that it is possible to perform certain operations on them without decompressing them. For example if we wish to compute the sum of a column of sorted integers with many repeated values, a compression algorithm might encode the column as a list of ordered pairs, each pair containing the value and the number of repeats, forming a so-called run-length encoding, see figure 2.2 for an example of RLE. Then it is trivial to compute the sum without decompressing the data [8]. Another thing that affects the effectiveness of compression is the so called column cardinality, or the number of unique values within a column. If we have a column of type Byte (8 bits) that currently only contains $16 = 2^4$ unique values, then it is sufficient to represent each value with 4 bits thus encoding two values per Byte instead of just one. This technique is called bit-packing [7].
2.2.4 Materialisation

In practice, many queries will access more than one attribute of a table and will need the different values for each attribute of some tuple at a certain time. This means that at one time during the query execution we will need all the selected values of a tuple. In a row-store this is not a problem since the values will be stored together, but in a column-store the values of a tuple are spread across many locations on disk, making it necessary to join these split values into a tuple in order for the query to execute. This operation, which is common in column-oriented systems, is referred to as materialisation, tuple construction or tuple reconstruction. The naive way to achieve this is by reading all the selected columns, construct the tuples and then perform the usual database operations on them. This strategy is called early materialisation. But research has shown that it is possible to get better performance if we delay the construction of tuples and instead try to perform the operations on the columns directly and only construct the tuples at the end [4, 3]. Take an example in which a query selects some tuples based on the values of attributes $A$ and $B$. It would be smarter to perform the selection on the $A$ column first and record which have been selected, then fetch the corresponding values in the $B$ column and perform the second selection and again record which positions have been selected, and lastly construct only those tuples with the selected positions. By doing so we have avoided constructing any tuples which would have been filtered out, saving potentially both I/O and CPU time. This strategy is commonly referred to as late materialisation. Note however that this approach requires that we store some extra information, namely the list of selected positions. Still in many cases it gives improvements in performance [8, 4, 3].

Advantages of using late materialisation, other than avoiding constructing unnecessary tuples, include avoiding decompressing columns early, which is needed for early materialisation. It makes it impossible to operate directly on compressed columns. It also improves cache performance. Since many operations operate on one column only, it is advantageous to fetch data only from the relevant column into CPU cache to make the most use of it. This is relevant especially in modern hardware systems, where CPU cache has become fast compared to main memory making cache misses an increasingly significant performance bottleneck [3].
2.2.5 Hardware Trends

The performance of the underlying hardware and its individual subcomponents has a great impact on the performance of a DBMS. Over the past 40 years computer hardware, especially processor technology, has undergone a major transformation in speed and capacity. At the same time the main design principles of database systems have stayed largely the same. However over the last 20 years, column-based storage has gained popularity mainly in data warehouse applications due to the trends in hardware performance [3]. The main improvement is in processing speed and storage capacity. From 1980 to 2010 a typical computer has become about 1000-2000 times faster in terms of CPU speed, and larger in terms of RAM and cache sizes. Disk storage capacities improved at even greater rate achieving 10000 times larger storage. At the same time, disk transfer rates increased only about 65 times and the average seek times only improved by a factor of 10. One way to look at the disparity between storage capacity and transfer rates is to look at the transfer bandwidth per (available) byte, which in 1980 was about 0.015 and in 2010 about 0.0001. Hence one perspective is that hard drives has gotten 150 times slower relatively to how much they can store. Another interesting metric is the ratio of sequential to random transfer rates, which increased from 5 : 1 to 33 : 1 in that time period, making it more important than ever to avoid random disk I/O operations and perform sequential reads/writes as much as possible [3, 9].

But it is not only disk transfer rates that has lagged behind in the performance increase. Memory access times have become the new bottleneck in computer systems. In 1980 it took about 6 CPU cycles to access main memory, while 30 years later it took 200 cycles per memory access mainly thanks to the increase in processing speed. There is also a growing disparity between the cache levels; a 2010 CPU had about 2-8 Megabytes L2 cache with an access speed of 20 cycles per access, and about 64 Kilobytes L1 cache with 2 cycles per access. In 1980 there was only one level of 8 Kilobytes cache. With these trends in mind, it becomes apparent that avoiding data cache stalls while executing a query, which is influenced by how the data in the table is organized, has become more important. If a query performs some operation on one column, then it makes sense to load only data of that column into cache after a cache miss. A column-based partitioning of the data allows us to do that. Another aspect of this is that when processing speed is fast and memory access is slow, it makes more sense to compress the data. Indeed, while compressing and
decompressing takes time, we can still make gains in performance by trading CPU cycles for less memory access [9], and for reasons mentioned earlier, it is generally possible to get a better compression ratio in a column-store.

2.3 Non-relational Databases

The previous section focused on databases which follow the relational model, which consist of collections of tuples with fixed attributes. Relational DBMSs were the standard in database technology until about 2005, in which the growing mismatch between the needs of emerging web applications and the relational model spurred the creation of completely new DBMSs with radically different models. The problem with the relational model is that it is too inflexible. Since the tuples consist of atomic values of fixed attributes, there is no way to store nested data consisting of objects and variable length arrays. There were several attempts to solve this problem. The most prominent one was the so-called Object-Relational Mapping (ORM) in which an object is mapped to relational tables. However, since the fields of an object could be complex and contain other objects, most objects needed to be mapped to several interlinked tables. It was both complicated and possibly leading to performance issues. Another inflexibility in the relational model is the use of a fixed schema that could not be changed. Many applications have the structure of their data changed over time. The need for a DBMS that can handle changes in the schema only reinforced the need for development of new database technologies. With the rise of the internet a new data format which belongs to the semi-structured model became widely used, namely JavaScript Object Notation (JSON). JSON became, and is still today, the unofficial standard for serializing objects to send over the internet or storing on disk. This led to the emergence of DBMSs which could operate on JSON files, so called document databases, although the term can refer to XML databases too. One of the earlier and most popular examples is MongoDB [10], but today there are many DBMSs that can be integrated with JSON or similar formats, and even some RDBMSs have added support for a JSON datatype. Document databases belong to a larger category often called NoSQL, a catch-all term for databases that deviate from the relational model [11, ch. 1].
2.4 Nested Columnar Storage Formats

As mentioned earlier, non-relational databases have gained popularity over the past years, and today many organizations store large amounts of complex nested data. In order to perform large-scale analytics on this data, many frameworks and storage formats have been developed to meet the challenges of storing and querying nested data efficiently. Traditionally, column-oriented formats have only been used for relational databases. However, recently there has been some development into creating column-oriented formats for nested data and query engines built around these formats. The earliest one is Google’s Dremel [5], which has inspired other similar projects after its reveal.

2.4.1 Dremel

Dremel is a query system developed by Google for analyzing large sets of nested data. It has been used internally in Google since 2006, and revealed first in a paper in 2010 [5]. It is the engine used in the Google BigQuery service, which can briefly be described as an API providing access to Dremel [12]. It uses a data model called protocol buffers, which is a strongly-typed model that in some sense is similar to JSON objects. It allows nested data and arrays of values (called repeated fields) but has a fixed schema defining the records of the data. A record corresponds to one object as defined by the schema and is the Dremel analogue of a tuple from relational databases. Apart from allowing nested data, the fields in a record can be required in which case they must have a value or optional in which they do not [5], see figure 2.3 for an example.

```plaintext
message city {
  required string name;
  repeated group district {
    required string district_name;
    optional int64 population;
    repeated int64 postcode;
  }
}
```

Figure 2.3: An example schema and a corresponding example record

A data file consists of many records, and the goal of Dremel was to split up the records into columns, so that the values of each field from the schema are stored together. In the example above the records would be split up into four
columns name, district.district_name, district.population and district.postcode. The difficulty of this approach lies in reconstructing the records faithfully back to their original form, which is problematic because it is impossible to know in which place a value from a repeated field should be. For example if we have only the column containing the three values of district.postcode there is no way to know which record or district within a record each value belongs to. A similar problem is with optional fields. We do not know which were defined and which weren’t. To solve that, Dremel attaches extra information to each value of repeated and optional fields which they call repetition and definition levels, basically specifying which values are defined and which are null and where new lists start. Using this information, Dremel can calculate from which place a certain value comes and then reconstruct the records to their original form. This introduces an additional overhead in the system, both in storage and CPU time for record materialization, but it is the trade-off made to gain flexibility.

2.4.2 Parquet

Apache Parquet is an open-source columnar storage format initially released by Twitter and Cloudera in 2013. It supports complex nested data which is implemented in the same way as Dremel including repetition and definition levels. It employs a number of encoding techniques such as bit-packing and run-length encoding [13, 14, 15]. Parquet divides the data into row groups, which is a logical partitioning of the rows. Each column inside a row group is then divided into column chunks, which are stored contiguously in a file. Column chunks are further divided into pages, which are treated as indivisible units in terms of encoding and compression. See figure 2.4 for an overview of the format.

A Parquet file contains both file metadata and column metadata, which store the positions of the individual column chunks. This metadata is intended to be used by a reader to locate the relevant column chunks and read only these chunks sequentially. Parquet supports a minimal set of primitive data types consisting of BOOLEAN, INT32, INT64, INT96, FLOAT, DOUBLE and BYTE_ARRAY. These primitive types can be extended to more complex logical types, for example a string which can be implemented using the BYTE_ARRAY primitive type. The idea is to reduce the complexity of the format and make it possible to use efficient encodings for all types [15].
2.4.3 ORC

Apache ORC (Optimized Row Columnar) is another columnar storage format with the support for nested data released in 2013 by Hortonworks and Facebook [16]. ORC divides the data into multiple large chunks called stripes, which are usually about 200MB and are intended to be self-contained units which can be processed in parallel. Each stripe has its own indexes and its own metadata in the stripe footers, see figure 2.5 for an illustration of the format.

Each stripe is further divided into several so called streams which can have different types and purposes, but each holding the data for a particular column. The DATA stream holds the actual values of the column, the PRESENT stream is a boolean stream that tells whether the values in the DATA stream are set or not and the LENGTH stream that records the lengths of the values for data of variable length, are among the important types of streams implemented in
ORC [17]. In this regard, ORC is a little different from Parquet and Dremel in that it uses the \texttt{PRESENT} and \texttt{LENGTH} streams instead of repetition and definition levels. Besides the values themselves, ORC maintains column statistics of various kinds such as the count, minimum value, maximum value, sum or the count of true/false values in the case of binary columns. These values are used to support faster aggregate function evaluations. ORC also makes use of index structures to speed up the access to the data, including minimum and maximum values for each attribute of the whole data and sections of 10,000 records [17].
2.4.4 Drill

Apache Drill is a query engine implemented in Java that is capable of running queries on a variety of data formats in a SQL-like language, including Parquet. It is optimized for columnar storage and uses a columnar in-memory data model, and its implementation is inspired by Google’s Dremel. The architecture of Drill enables it to achieve a high level of parallelisation, which is attained by dividing up the work to several processes called drillbits. When a query is submitted it is sent to one drillbit which parses and optimizes the query before dividing it into several fragments which can be run in parallel and sent to several other drillbits for execution [18].

2.4.5 Spark

Apache Spark is a cluster computing system implemented in Scala. It can read a wide variety of data, including in Parquet and ORC format, distributed across several nodes in a cluster and perform computations mainly intended for data analytics. It represents data internally as resilient distributed datasets (RDDs) which is a collection of data split among the nodes and which can be processed in parallel. Spark supports two types of operations: transformations, which take a dataset and create a new one from it and actions, which return a value to the main program based on the values of a dataset. Transformations are evaluated lazily, as in they’re only evaluated when an action requires the result from the transformation [11, 19].

2.5 Related research

With the development of storage formats such as Parquet, ORC and query engines like Drill and Spark, research aiming to evaluate the performance of these technologies started to appear. Although up until now, most research papers in this topic tested the performance of one particular technology against formats such as plain text. No systematic test of all combinations of the previously mentioned technologies appears to exist. Still, the most interesting results from previous research will be presented here.

2.5.1 Dremel

The original Dremel paper [5] tested its performance when reading 1 GB fragment of 87 TB table and found that record construction is an expensive opera-
tion, and that retrieval time grows linearly with the number of fields. The performance of Dremel was also compared against MapReduce (MR) for counting the average of one field from the previous table on 3000 nodes. It was found that Dremel performed about two orders of magnitude (100) faster than MR when MR read the whole data, and about one order of magnitude faster when MR only read the required column. To test scalability a query scanning about 4.2 TB data from a 105 TB table was run in a range from 1000 to 4000 nodes and found that execution time decreases almost linearly with the number of nodes. Another experiment found that in a query most data was processed in a short time, while a small portion of the data may stall the query for a considerable amount of time. This means that it is possible to trade time for a little of accuracy by ending a query after it processed most of the data.

2.5.2 Parquet vs. PSV

In a 2016 paper [20] three researchers tested the performance of Drill on heterogeneous schema-less healthcare data. The experiments compared Parquet against pipe-separated values (PSV), both plain text and compressed with gzip. The volume of the data was 5.8 GB in plain uncompressed text and 1.4 GB in Parquet format. The tests were run on a single machine with 8 GB of RAM. Four different queries of varying characteristics were run while measuring the time it took to complete each query. In all cases Parquet was the fastest alternative while compressed delimited text was the slowest one. The biggest difference was in row count queries, where Parquet ran 173 times faster than plain PSV and 450 times faster than compressed PSV, while queries performing joins on several tables resulted in the smallest difference as Parquet was only about twice as fast as PSV. The experiments showed that Parquet is definitely a better choice than plain text even in the worst cases, and that joining is an expensive operation for Parquet compared to other operations.
Chapter 3

Methods

3.1 Data

The data used in the experiments is stock-trading data provided by the company NGM. The data consists of logged messages sorted in time, each message with a timestamp in millisecond precision and a specific message type corresponding to an event in the system, such as a stock order or a system event. The data was converted to the tested file formats, and each message type was mapped to its own table/file with a specific schema. Furthermore, the tests were performed on two datasets of different sizes to test how the systems scale with increased data sizes. The small dataset consists of 9 months of logged data making up a total of 701GB gzipped text files, while the large dataset consists of 17 months of data including the 9 from the small set, and has a total size of 1158GB, making it almost twice as large as the small set.

Each message consists of key-value pairs, each value consisting of a specific type such as a number, string or array. An array can itself be nested, containing subarrays and objects. Also, there is no guarantee that the values of any field are set, and the schema of each message evolves over time with new fields added or disappearing. For these reasons, all the fields were set to nullable in the schema, or in the language of protocol buffers, they were set to optional.

Non-integer numeric fields were all stored as a `Decimal(19, 3)` instead of `Double` or `Float`, since accuracy was a crucial requirement in representing numeric values. In the context of databases, `Decimal(19, 3)` refers to a decimal with a precision of 19 and scale of 3. Precision is the total number of digits in the decimal, while scale denotes the number of digits after the decimal.
3.2 System specification

The benchmarking was performed on a single computer with a 64-core 2.1 GHz CPU. The cache sizes are 32KB L1i, 32KB L1d, 256KB L2 and 40MB L3 caches. The total amount of installed RAM is 503GB, but not all of it is available for the queries, and in all cases the amount of available RAM will be smaller than the dataset under test. The software used in the benchmarking were run in linux under nice and ionice, giving the processes a lower priority than normal. During the benchmarking all the tested alternatives were given the same resources as described here in order to make an unbiased evaluation.

3.3 Technologies

The frameworks that were tested are Spark version 2.4.0 and Drill version 1.15.0. Both were the latest stable releases of each project, and both were run with the default configurations that come with their installations. They were tested with the file formats Parquet and ORC, but only Spark was used with ORC since Drill doesn’t support querying ORC files yet. The data was converted to the tested formats using Spark. Both file formats were tested in two different modes: non-compressed and compressed with Snappy. Snappy is a compression format developed by Google that is optimized for fast compression and decompression at the expense of lower compression ratio [21]. Snappy is supported by both Spark and Drill in order to enable light-weight compression without trading too much performance. The purpose of this test is to exactly measure the impact on performance incurred by using such compression formats.

Due to hardware constraints, only compressed formats will be tested with the large data set.

3.4 Queries

For the benchmarking, two queries in increasing complexity were conceived. These queries were determined by which operations they need to use and how complex they are, rather than how meaningful the output is, although they were made to be similar in style to how a real analytic query could look like. One
thing that they all have in common is that they require a scan of a section of the
data from the whole time period, so if the whole dataset spans a timeframe of
one month, then the query will cover the whole month. Another thing is that
they all use only reads, no inserts or deletes are tested below. The reason being
that this thesis is focused on data analytics in data warehouses, updates which
are frequent in transaction-intensive applications are of no interest here. The
queries were run on one table, henceforth named T1, which was used since it
is the biggest one in term of size. T1 consists of 38 fields out of which 10 are
numbers (either Long or Decimal(19,3)) and one is an array.

Since the queries were run on different frameworks, there was a variation in
the exact implementation for each of them. For example, Drill uses a dialect
of SQL and Spark uses Scala instead. However, all tested alternatives support
standard operations such as joining, grouping, aggregation etc. So the differ-
ences were not major. Below, the two queries are described in plain words.

Q1: Calculate the average value of 2 decimal fields from T1.

Q2: From T1, find the most occurring value of a string field, then select all
the records with this value and calculate their average for 5 decimal fields.

3.5 Measurements

The measurand in the benchmarking is query execution time, which is mea-
sured by recording a timestamp once before a query is submitted and again af-
fter a successful query execution. Each query is executed twice for each tested
format and dataset, to see if there is a deviation in the resulting time between
the two runs. The time it takes to start a program at the beginning, for example
starting a Spark or Drill session, was not included in the measurement. For
each file format and dataset, the total size in bytes will be presented.
Chapter 4

Results

4.1 Table sizes

Below in table 4.1 are the table sizes for each file format and dataset. We see that ORC with Snappy is always the smallest format while uncompressed Parquet is the largest one. On the small dataset, we also see that Snappy gives a compression ratio of about 2.1 on Parquet and 1.5 on ORC.

<table>
<thead>
<tr>
<th></th>
<th>Uncompressed</th>
<th>Snappy</th>
<th>Uncompressed</th>
<th>Snappy</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>1.5T</td>
<td>699G</td>
<td>821G</td>
<td>540G</td>
</tr>
<tr>
<td>big</td>
<td>-</td>
<td>1.2T</td>
<td>-</td>
<td>880G</td>
</tr>
</tbody>
</table>

Table 4.1: The size of table T1 for each file format and dataset in Bytes

4.2 Query execution time

Below are the query execution times for the tested alternatives, each graph representing a particular query and dataset size. The tested alternatives are abbreviated by the query engine first (Drill or Spark) followed by the file format (’p’ = Parquet, ’o’ = ORC) and the compression (’s’ = Snappy, ’n’ = No compression). So for example, ’drillps’ represents Parquet compressed with Snappy an run with Drill as the query engine. Since each query was run twice, both execution times will be presented as trial number 1 and 2.
Figure 4.1: The execution times for Q1 on the small dataset

In 4.1 we see that for trial number 1 drillps gives the best result at 4000s (1h6m) while sparkon gives the worst result at almost 12000s (3h13m). In all cases Parquet gives lower execution times than ORC, Snappy gives lower time that no compression and Drill is faster than Spark. In many cases the second run is faster than the first one.
In 4.2 we see more or less the same result as in the previous figure, with drillps being the fastest at around 1h14m while sparkos is the slowest at 4h37m for trial number 1. Interestingly, drillps becomes slower at the second run.
Figure 4.3: The execution times for Q2 on the small dataset

In 4.3 Spark with ORC continues to be the slowest alternative at around 4-5 hours for both Snappy and no compression. The change from before is that Spark becomes the faster alternative, with sparkps taking the lead at around 41m. One trend that continues from before is that Snappy generally gives better results than no compression.
Figure 4.4: The execution times for Q2 on the big dataset

In 4.4 we see that sparkps is again the fastest alternative at 3h13m in the first run. Again, the trend continues with Snappy being faster than no compression and the second run being generally faster than the first one.

Below, the trend in execution time from the small dataset to the big one are shown, each graph representing one of the two queries. Only the data from trial number 1 was used below.
In 4.5 we see the trend in performance as the data size increases. As we saw before, drillps is the fastest while sparkos is the slowest. Furthermore, we see that drillps scales the best by having the smallest increase in execution time, while sparkps increases the most.
Figure 4.6: A comparison of the execution times for Q2 between the small and big dataset

In 4.6 sparkps is the fastest alternative while sparkos just as we saw before. However, drillps does scale slightly better than sparkps, as the difference in performance between the two decreases as we move from the small to the big dataset.
Chapter 5

Discussion

5.1 Data size analysis

The sizes for each file format show unsurprisingly that Snappy compressed files are smaller than their non-compressed counterparts. More surprisingly, there was a big difference between non-compressed Parquet and ORC, with Parquet being almost twice as large. This disparity decreases when looking at the Snappy compressed files, where Parquet is only about 30%-40% larger than ORC. This is also shown by the different compression ratios which Snappy achieves on the two file formats, 2.1 for Parquet and 1.5 for ORC. The obvious explanation for this is that ORC uses encodings which achieve better compression ratios than Parquet, which means that Parquet files have lower information entropy which compression algorithms like Snappy can use better. It is generally harder to further-compress data which is already well-compressed.

5.2 Execution time analysis

Looking at the execution time results, a few general trends become obvious. First of all, the second trial is sometimes faster than the first one, except for one case (drillps on Q1 - big dataset) which could be a random fluctuation, most probably due to caching effects (see sources of error for more discussion). Due to this fact, the first trial will be seen as the most representative one, although in almost all cases the two trials agree on which alternative is the best and worst one.

Secondly, we see that Snappy compressed files always give a faster result than their non-compressed counterpart. This is interesting because a compressed
file must be decompressed before it can be used in a query, which should increase CPU time. But as we discussed in section 2.2.3, a better compressed file will require less disk IO, and when CPU performance outweighs IO performance it makes sense, in theory, to trade less disk IO for more CPU cycles spent on decompression. The results in the benchmarking show that the theory is correct, but another way to look at it is than neither Parquet nor ORC are using sufficient built-in encodings.

Furthermore, in all the queries ORC performed slower than Parquet, sometimes as much as 3-4 times slower than the best alternative. The reasons for this are hard to explain, but we know that Parquet uses Dremel’s repetition and definition levels to encode nested messages while ORC uses its own streams encoding. This is one candidate for the reason in the performance difference, another one could be that Spark and Drill have a better implementation for reading Parquet files. It should be noted that ORC has a better compression than Parquet and yet performs worse, which means that disk IO is far from the only thing that affects performance.

Comparing Spark and Drill gives a more ambiguous result. Here, Drill performed faster in Q1 while Spark performed faster in Q2. The second query is in some sense more complicated than the first one, since it performs a row count, then a filtering, then averaging a few values while Q1 only performs a simple averaging on the whole data. This could indicate that while Spark is slower at reading data, it could be better at query planning which gives it an advantage at more complex queries. Testing more complex and varied queries could confirm this hypothesis.

Lastly, looking at the last two graphs about the trend in performance, it becomes clear that Drill scales the best with increasing the volume of the data. For Q1 the increase in time was marginal for Drill, while for sparkps it more than doubled. For Q2, even though sparkps had the fastest performance, it was drillps which had the smallest increase, decreasing the gap between itself and sparkps. If the trend continues, Drill may even outperform Spark for larger datasets, but more tests need to be performed to confirm this idea.

### 5.3 Sources of error

There are a few sources of error that could be associated with the experiments. First of all, there could be random changes in the execution time from different
runs. To account for that, all queries were run twice, and the general trend is that the second trial was faster than the first one. This is most probably due to the effect of caching. In order to avoid this, the two trials were never run one after another. So instead of testing one alternative two times and then moving to the next one, all alternatives were tested one time after another, and then a second time. This way the second trial couldn’t reuse the data loaded into the RAM from the first trial. But still, frameworks like Drill and Spark or the operating system itself may cache some data and it is generally not easy to turn it off. Despite that, the trends that present themselves in the results are consistent and there was no case where it was unclear which of the alternatives was the fastest one.

There are other system configurations which could have affected the outcome. Both Drill and Spark can be configured in different ways, but the configuration parameters are too numerous and most of them may not affect the performance at all. For example, the number of allowed Spark connections at the same time is not relevant here since the queries above were run one at a time rather than in parallel. In the benchmarking, which was done in this thesis, the default configuration coming with the installation was used for both Spark and Drill.

5.4 Sustainability

This thesis is not directly concerned with sustainability per se but, it is linked to using hardware more efficiently, which obviously improves both economical and environmental sustainability. If a certain system is twice as fast as the other, then the same query can be run with half the resources, for example half the CPUs or electricity spent on it. Likewise, a file format that can compress the data to half the size compared to another format will need only half the disk storage amount. The actual impact depends on the scale of the system itself, for small organisations with little data this may not mean so much, but for large organisations which deal with very large volumes of data, even a small difference in efficiency makes a big impact in terms of resource use.

This work doesn’t, however, relate to social sustainability or ethics in any obvious way, as it doesn’t involve issues such as privacy, human rights, equality etc. One could argue that big data analytics has made it possible for organisations to monitor and profile people more easily, which gives rise to concerns about privacy, and that this thesis investigates technologies that make privacy violations easier. Of course, it is important to be aware that this technology may
be used for such purposes, but there are many applications where this isn’t the case. The purpose of this thesis, for example, is to examine these technologies for the use of analysing trading data, which doesn’t involve privacy or ethical issues as long as it is used in a legal way.
Chapter 6

Conclusions

6.1 Summary

The results can be summarised as follows. ORC has a built-in compression with a higher compression ratio than Parquet, giving it smaller data sizes, but the difference between the two diminishes when using the Snappy compression algorithm. In all cases, the Snappy compressed formats performed faster than non-compressed formats even though they need to be decompressed during query execution, giving support to the theory that it is important to reduce disk IO even at the cost for extra CPU cycles. However, disk IO is evidently not everything, as the ORC format performed consistently slower than Parquet in the benchmarking. The reasons for this are not clear but more research that delves deeper into the differences between the two formats could clear this up. The benchmarking also showed that Drill and Spark performed differently for the two queries, with Drill performing faster at a simple averaging query while Spark performed faster at a more complex query involving a row count and filtering. This could indicate that Drill is faster at scanning data but Spark is better at query planning giving it more advantage at more complicated queries, but more testing need to be performed to confirm this hypothesis. The benchmarking also showed that Drill scaled better than Spark even at the complex query, having the smallest increase in execution time between the two datasets of different sizes. This could potentially mean that Drill would’ve performed better with even larger datasets.
6.2 Conclusions

The main conclusion of this thesis is, Snappy compressed Parquet is clearly the best file format out of the tested alternatives in terms of giving the fastest query execution times. It consistently performs the fastest for all combinations of queries and query engines tested. Compressing Parquet files have the additional advantage that they require less disk space. The choice between Drill and Spark is not as clear as they both seem to have their own advantages. Spark performed faster for a complex query, and analytical queries tend to be more complex involving operations such as filtering etc. However, Drill scaled much better than Spark which is important as data volumes get bigger, which is a situation than many companies face today. Overall, switching between Spark and Drill or any other similar software framework that could be developed in the future is relatively easy, and both can even be used at the same time. The choice of the file format is a more important task as data migration is a lengthy and involved process, meaning it is not easy to switch from one format to another if it becomes evident that the other format is better. This thesis clearly reveals the best format, so it fulfilled its main objective.

Explaining the reason behind the difference in performance turned out to be a much harder task than expected. The main problem is that most of the tested technologies are poorly documented. There’s no overview of how Spark or Drill works internally. Another reason is that there are many factors which affect performance. A file format like ORC and Parquet combine many ideas and encodings into one single format, and any combination of these ideas could affect performance in different ways. A more thorough investigation that looks at each individual component of the system and tests each parameter could be necessary to understand the differences better.

6.3 Future work

As discussed above, one possible future work is to investigate the technologies more deeply by looking into the individual components and parameters more to gain a better understanding of the difference in performance. Another possible work is to run the benchmarking for even bigger datasets or more complicated queries to see if Drill or Spark still perform better. Yet another possible idea is to compare the alternatives not just against each other, but also against a different technology like MySQL or column-based relational
databases. It would be interesting to see how formats like Parquet and ORC compare to more traditional RMBDSs.


Appendix A

Query execution time results

<table>
<thead>
<tr>
<th></th>
<th>drillpn</th>
<th>drillps</th>
<th>sparkpn</th>
<th>sparkps</th>
<th>sparkon</th>
<th>sparkos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 small</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trial 1</td>
<td>5253.903</td>
<td>3984.34</td>
<td>6979.059</td>
<td>5911.087</td>
<td>11570.314</td>
<td>10786.276</td>
</tr>
<tr>
<td>trial 2</td>
<td>2182.973</td>
<td>2203.966</td>
<td>5889.875</td>
<td>6008.044</td>
<td>11526.137</td>
<td>10856.737</td>
</tr>
<tr>
<td>Q2 small</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trial 1</td>
<td>10413.904</td>
<td>6977.83</td>
<td>4476.069</td>
<td>2489.68</td>
<td>15747.48</td>
<td>17391.517</td>
</tr>
<tr>
<td>trial 2</td>
<td>12819.021</td>
<td>5356.928</td>
<td>2369.525</td>
<td>2393.643</td>
<td>16125.629</td>
<td>14472.26</td>
</tr>
<tr>
<td>Q1 big</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trial 1</td>
<td>-</td>
<td>4476.407</td>
<td>-</td>
<td>15908.722</td>
<td>-</td>
<td>16641.874</td>
</tr>
<tr>
<td>trial 2</td>
<td>-</td>
<td>8575.362</td>
<td>-</td>
<td>11359.931</td>
<td>-</td>
<td>8297.913</td>
</tr>
<tr>
<td>Q2 big</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trial 1</td>
<td>-</td>
<td>13473.053</td>
<td>-</td>
<td>11588.593</td>
<td>-</td>
<td>24010.431</td>
</tr>
<tr>
<td>trial 2</td>
<td>-</td>
<td>13690.335</td>
<td>-</td>
<td>6104.987</td>
<td>-</td>
<td>11014.063</td>
</tr>
</tbody>
</table>

Table A.1: The execution times in seconds