MASTER'S THESIS 2022

Evaluating ClickHouse as a Big Data Processing Solution for IoT-Telemetry

Adrian Göransson, Oskar Wändesjö

ISSN 1650-2884
LU-CS-EX: 2022-14

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-14

Evaluating ClickHouse as a Big Data Processing Solution for IoT-Telemetry

Utvärdering av ClickHouse för bearbetning av Big Data från IoT-telemetri

Adrian Göransson, Oskar Wändesjö

Evaluating ClickHouse as a Big Data Processing Solution for IoT-Telemetry

Adrian Göransson
adrian.goransson.8223@student.lth.se

Oskar Wändesjö
oskar.wandesjo.7461@student.lth.se

April 2, 2022

Master's thesis work carried out at Axis Communications AB.

Supervisors: Anton Friberg, anton.friberg@axis.com
Markus Borg, markus.borg@cs.lth.se

Examiner: Alma Orucevic-Alagic, alma.orucevic-alagic@cs.lth.se

Abstract

With data analysis migrating into the realm of big data, storage and analytics tools that have traditionally worked well in the past have begun to show their limits. To address the problems with greater volume, velocity, and variety of data, ClickHouse, among other new technologies, has emerged. ClickHouse is a column-oriented OLAP DBMS developed at Yandex, open-sourced in 2016, and in September 2021 was spun out, creating ClickHouse, Inc., reaching a valuation of two billion US dollars the following month.

This thesis evaluated ClickHouse on telemetry data analysis through experimental benchmarks based on real use-cases at Axis Communications with real-life metrics from IoT devices. ClickHouse was compared to the current implementation comprising Elasticsearch and MinIO as an on-premise solution.

The results established ClickHouse as a suitable candidate to handle the challenges of big data processing while still being cost-effective and highly approachable for new adopters.

Keywords: ClickHouse, Big data processing, Data warehousing, Column-oriented database, OLAP DBMS, Experimental benchmark

Acknowledgements

This thesis was carried out in collaboration with Axis Communications AB and the Department of Computer Science at the Faculty of Engineering at Lund University. Both parties have provided us with a great deal of support and assistance, for which we are very grateful.

We want to thank Anton Friberg, our supervisor at Axis, for his tireless support in tackling the many practical issues this master's thesis has faced. We thank him for sharing his valuable insights and opinions, helping us see the big picture, and carry on when we got stuck.

We would also like to thank Markus Borg, our supervisor at LTH, for the continuous feedback and valuable guidance provided throughout the course of this thesis project. His encouraging and insightful suggestions always nudged us in the right direction and have brought this thesis to a higher level.

We also want to express our gratitude to the department of Diagnostics & Data Management at Axis for the opportunity to conduct this research. Thanks in particular to the data engineers for participating in our interviews and providing us with important insights.

Special thanks to the fellow master's thesis students at DDM, for making this autumn an extra fun one and for providing input, assistance, and great company. We are grateful for the much-needed breaks through countless coffee runs.

Last but not least, to our close friends and significant others, thank you for tolerating us during this semester.

Contents

1 Introduction **7**
1.1 Problem statement 8
1.2 Motivation 8
1.3 Related work 9

2 Background **11**
2.1 Database and data storage concepts 11
2.1.1 Relational databases 11
2.1.2 NoSQL 13
2.1.3 OLAP 15
2.1.4 Object storage 16
2.1.5 Data compression 17
2.2 Data storage products 18
2.2.1 Elasticsearch 18
2.2.2 MinIO 19
2.2.3 ClickHouse 19
2.3 Data analysis context at DDM 19
2.3.1 Current setup 20
2.3.2 Use cases 22

3 Method **27**
3.1 Approach 28
3.2 Experimental design 29
3.2.1 Hardware specifications 29
3.2.2 Test data 30
3.2.3 Experimental subjects setup 30
3.3 Experiments 31
3.3.1 Exp A: Ingest 31
3.3.2 Exp B: Storage 33
3.3.3 Exp C: Extraction rate 33

4 Results **37**
4.1 Exp A: Ingest 37
4.2 Exp B: Storage 37
4.3 Exp C: Extraction rate 38

5 Discussion **43**
5.1 Benchmarking experiments 43
5.1.1 Ingest performance 43
5.1.2 Storage 44
5.1.3 Extraction rate 45
5.2 Usability and workflow 46
5.3 Threats to validity 49
5.3.1 Internal validity 49
5.3.2 External validity 49
5.4 Future work 50

6 Conclusion **51**

References **53**

Appendix A Dedicated server setup **63**

Appendix B Initial setup **65**

Appendix C Data extraction queries and scripts **67**

Appendix D Raw results **75**

Chapter 1

Introduction

Business intelligence, the practice of analyzing data to gain insights that inform business decisions, is an essential tool in the industry and has been for over three decades. During this time, the amount of data available and collected has grown substantially. As a consequence, the limitations of traditionally used data management tools began to make themselves apparent, giving name to the term *Big data* [94]. Big data is used for data characterized by the three Vs: Volume (requiring vast storage capacity), Velocity (being created and processed at high speeds), and Variety (containing complex information) [72, 59, 94]. Madden elegantly describes it as data that is "*too big*, *too fast*, or *too hard* for existing tools to process" [70]. The three Vs were originally coined in 2001 and have over time grown into the five Vs with Veracity and Volatility, referring to the quality and durability of the data [59].

To address the problems of big data processing, new technologies are emerging and evolving at a rapid pace. Over the last decade, several database systems categorized as *NoSQL* have entered the market and has seen a drastic increase in popularity as an alternative to the traditional relational databases [79]. NoSQL is a paradigm that aims to mitigate the limitations (such as horizontal scalability, i.e., scaling by adding more servers) of conventional relational databases, by rethinking the requirements of a database system [10]. Relational databases and NoSQL are covered in further detail in sections 2.1.1 and 2.1.2. One particular NoSQL database that has appeared recently and that this thesis will focus on is ClickHouse.

ClickHouse is a column-oriented database management system for online analytical processing (OLAP) [13] initially developed at Yandex for their web analytics service *Yandex.Metrica* in 2009 [12]. In 2016, ClickHouse was open-sourced and in September 2021 ClickHouse, Inc. was created as a spin-out from Yandex [74], reaching a valuation of two billion US dollars the following month [16]. ClickHouse boasts high-performance analytics for large amounts of data in real-time. This thesis aims to evaluate these claims in a real-world setting.

1.1 Problem statement

The department of Diagnostics and Data Management (henceforth referred to as DDM) at Axis face challenges in scaling up their device analytics. The current solution relies primarily on Elasticsearch which can handle large amounts of data, with flexible schemas (i.e., the logical structure of data entities can vary). The downside with Elasticsearch is that it is very costly with regard to both storage and memory, as well as being slow at ingesting data (i.e., importing and processing data). As a consequence, only a few months of data is indexed and processed at DDM. This constraint imposes great limitations on the analysis workflow in certain teams, to the extent that it virtually eliminates any valuable time-series analysis. Hardware designers, for example, may need temperature statistics over several years to make sustainable decisions for future devices regarding assembly and materials. To work around these limitations, MinIO, an object storage solution was introduced that was able to store longer periods (several years) of highly compressed data. However, this alone does not solve the shortcomings of Elasticsearch. To enable insights from the entire catalog of source data, the storage solution was complemented with in-house developed processing scripts.

The internal workflow is not without issues either, requiring a significant amount of time and effort to use and maintain. Such problems are common when trying to expand analysis tools to match the data size [70]. Another significant aspect is the memory and time requirements required to execute the scripts, where out of memory errors and eight hours or more of processing time is common.

It is clear that in order to handle the five—or even original three—Vs of big data, a more suitable database solution must be implemented.

1.2 Motivation

A reasonable concern when adopting a new technology is its maturity and production readiness. ClickHouse is still a relative newcomer on the stage of NoSQL databases. In 2016, ClickHouse was open-sourced [28] and had its documentation translated from Russian to English, Chinese, and Japanese, increasing its international reach greatly. The spin-out from Yandex to a separate international company further emphasizes the focus put on bringing ClickHouse to an international crowd, and their two billion valuation signals great interest in the technology, yet there is a lack of academic research on ClickHouse. This thesis aims to give better insights into the advantages and drawbacks of using ClickHouse for storage and analytics in the realm of big data. The mix of quantitative and qualitative findings presented in this thesis will be utilized by DDM in their evaluation process of ClickHouse for big data processing. Companies in similar situations are encouraged to make use of the results as a foundation for further research in their pursuit to better understand the data they collect.

We aim to answer the following three questions:

RQ1 What is the state of DDM's big data solution in terms of performance and usability?

RQ2 How can ClickHouse help mitigate issues identified in **RQ1** in DDM's environment?

RQ3 How are existing workflows for data consumers affected by migrating to ClickHouse?

1.3 Related work

As previously stated, with ClickHouse being a fairly recent addition in the big data processing field, the amount of academic research concerning ClickHouse has been fairly limited.

Wickramasekara, Liyanage, and Kumarasinghe [91] compared ClickHouse with MySQL in a low-performance environment. Benchmarking experiments were performed with 1.5 million records on a virtual Linux environment with 1 GB RAM and a 1.8 Ghz single-core processor. The results show that ClickHouse was better at utilizing the given resources and achieved a lower execution time as well as a higher disk write speed. Although it provided some insights into how resource-efficient ClickHouse can be, given the small data set and the low performance environment, it is not really comparable to a real-life industrial setting.

Imasheva et al. [65] studied the replacement of Oracle, a relational database management system, with ClickHouse. The study defines the term big data according to the three Vs previously mentioned, and compares the theoretical differences between the two databases. Finally, benchmarking experiments were conducted on the two databases using a 1.5 terabyte data set. The report shows promising numbers in favor of ClickHouse, citing a notable 2,290 time speedup on reads and 12 time speedup for inserts. While impressive, the authors do not disclose how the research was conducted, and neither test data nor benchmarks are presented. As such, it is difficult to verify the validity of the study. There are nonetheless confounding factors to consider. The experiments were performed on separate machines, with different operating systems (Windows and Linux) and clear differences in hardware specifications, as ClickHouse had significantly more memory and CPU cores to work with than Oracle.

Struckov et al. [89] evaluated ClickHouse and compared it to the time-series databases InfluxDB and OpenTSDB as well as the PostgreSQL extension TimescaleDB. Generated data was used with similar characteristics to real data found in three different use cases with variations in frequency, origin, and time span. The study found that ClickHouse performed well overall and excelled at ingesting large amounts of data. It performed less well at compressing the ingested data and filtering it during extraction. The research provides promising insights into the performance characteristics of ClickHouse using small and synthetic data sets. This study, on the other hand, is focused on ClickHouse's performance for real and large-scale data, to get a better understanding of how capable it would be in an industrial deployment setting. In addition, there have been several performance improvements in ClickHouse [17] as it has matured since 2019 when the article was published.

Another study comparing ClickHouse with InfluxDB was performed at CERN by Vasile, Avolio, and Soloviev [90]. The study sought to find a suitable database candidate that could handle large amounts of operational monitoring data at high frequency. The work tested the ingest speed of both databases and found that in five out of six tests, ClickHouse outperformed InfluxDB. While ingest speed is an important dimension to consider when evaluating a database for big data, the read performance of the two databases was not tested.

Alongside the research focused on ClickHouse specifically, research in the field of DBMSs have also looked at implementing other emerging NoSQL DBMSs for OLAP workloads and big data aggregations. One example of such being a study by Correia et al. [34] on Apache Druid, a column-oriented distributed data store. The study evaluated Druid's performance

for OLAP workloads using a synthetic benchmark suite called Star Schema Benchmark, and looked deeper into query granularity and partitioning to achieve better processing times. In a follow-up paper [33], many of the same authors evaluated if Druid would hold up as an alternative to the SQL-on-Hadoop technologies, Hive and Presto. In all processing benchmarks, Druid achieved significantly better performance and positioned Druid as a top contender in the realm of big data processing. Although, the study remarks that other qualities than data processing need to be considered when looking to adopt a technology.

Another emerging contender for big data processing is Apache Pinot. Like Apache Druid, it is also a column-oriented distributed data store. Fu and Soman [54] conducted an experimental evaluation including Pinot, Elasticsearch, and Druid in an effort to find an OLAP solution for the big data processing pipeline at Uber. The results of the study show that Pinot consumed one quarter of the memory and one eight of the disk usage compared to Elasticsearch, as well as lower query latency. Druid, with a similar architecture to Pinot, was not far off in performance but Pinot was given the advantage for its implementation of optimized indices to achieve faster query execution and lower query latency. As the evaluation of the OLAP layer was only a smaller part of the paper, not a lot of information was given about how these experiments were conducted and the exact results were not published.

Elasticsearch, a search database, has also been studied for its big data processing capabilities. In a report by Zamfir et al. [93], the authors investigated Elasticsearch as a monitoring solution for large amounts of log data. The purpose was to create a framework for DevOps monitoring that could process logs and events in real-time. To do so, an experiment suite was created using different search and aggregation queries to rate how suited Elasticsearch was for big data processing. The report concluded that Elasticsearch was an appropriate implementation for the use case presented. However, with only 25 GB of disk storage, it is questionable whether the data tested could actually be considered *big data*, and it is difficult to generalize the findings to apply for larger data sets. Another report by Seda et al. [84], compared Elasticsearch against MySQL, a relational database management system. In the tests, which focused on key-value data, Elasticsearch performed approximately five times better than the worst-case for MySQL, and three times better against MySQL where the key was indexed. It should be noted, and is mentioned in the report as well, that there are databases better suited specifically for key-value workloads.

In conclusion, previous studies, albeit few, have been conducted on both ClickHouse and Elasticsearch. ClickHouse has been noted for its resource efficiency and being competitive to both relational DBMSs and time-series databases. The ingest performance in particular is found to be excellent. Studies more closely focused on the OLAP aspects, however, tend to focus more on the modeling and structuring of the data, instead of ClickHouse's performance operating on the data. And in most work, the benchmarks are synthetic with pre-generated data. Studies on Elasticsearch seem even more limited, and are largely focused on its characteristics as a search engine for log data, and does not test it for the purpose of analysis that resembles the situation at DDM.

We found no previous work comparing ClickHouse and Elasticsearch, in any capacity.

Chapter 2

Background

To get the most out of this thesis, some key concepts need to be understood as well as some specific knowledge about our experimental setup that will affect our implementation and evaluation of ClickHouse, Elasticsearch, and MinIO. Therefore, this chapter describes the current setup at DDM and the concepts that will be used to better understand how ClickHouse functions as well as what to consider when evaluating our implementation.

2.1 Database and data storage concepts

We begin by reviewing some essential knowledge about database design and types to understand what makes ClickHouse different from previous database management systems. This is followed by some concepts that help us evaluate ClickHouse and compare it to the solutions currently implemented at DDM.

2.1.1 Relational databases

A relational database is a database that is based on the relational model first proposed by Codd in 1970 [29] and it has been the dominant database type ever since [68]. Since the late 2000s, there has been a growing awareness for other database paradigms such as NoSQL [88]. These will be discussed with greater detail in section 2.1.2.

When the first databases started to emerge in the 1960s [8], there was no widely accepted way of storing data. Every application used its own unique data structure that required getting familiar with before being able to utilize the stored information [8]. To make matters worse, these implementations were often inefficient, hard to maintain, and hard to optimize [80].

With the adoption of the relational model, developers were provided a standard way of representing data, freeing them from having to reinvent the wheel every time they wanted to store data. The model organizes the stored data in a table structure, as shown in table 2.1, where rows are records and the columns are attributes of the records [80]. This structure allows the data in the tables to be identified and accessed in relation to each other, i.e., all the data regarding a specified record (row) can be accessed in its entirety or every record satisfying a chosen condition for a specified column. The relational model provides a flexible way to store and access data which made it possible to be used by a large variety of applications, while also being intuitive and efficient compared to prior solutions [80].

Table 2.1: A conventional database table structure. Records are represented by rows, and their attributes by columns.

model_name	serial_number	…	max_temp	min_temp
ABC	123	…	43.47	12.36
…	…	…	39.42	10.06
ABC	456	…	43.56	12.55
…	…	…	39.98	11.63

Today, the standard way to manage a database is to use a database management system (DBMS). A DBMS is a software that utilizes a database for storing data and also provides a façade for defining, constructing, manipulating, and sharing databases among various users and applications [52]. The abstraction created by the use of a DBMS eliminates the need to know the details of how the data is stored in the database and focus can instead be directed at the application using the database. A DBMS that is based on a relational model is also referred to as a relational database management system (RDBMS) [52].

To communicate with an RDBMS, the standard way is to use Structured Query Language (SQL) [64]. SQL is a language that enables, in an easy and fairly non-technical way, the accessing and insertion of data into a database with no need to know how to get the data and where on the disk the relevant table is stored. Benefits of SQL include the ability to join tables to create meaningful information by combining data from separate tables. Another feature that SQL supports is basic data aggregations such as: count, add, and grouping queries as well as basic math functions. The results can be ordered by any of the columns. Since SQL is standardized, the same query should work on any compliant RDBMS.

Although a relational database has several benefits, there are a few constraints that need to be considered; one such consideration is the actual data being stored. A relational database works best when the data has a known structure (as opposed to unstructured data, such as images or audio) and can be placed inside a table [68]. To place every data point in its correct spot a schema needs to be predefined with fixed names and types for the columns [68]. Another constraint that has grown into a considerable problem with the arrival of big data is how the database scales when handling terabytes or even petabytes of data [94].

ACID

A key feature of most relational databases is their ability to perform work in transactions. Transactions enable concurrent access to the database by treating one or more operations as

a single unit of computation. This unit can then be rolled back or committed to the database depending on the success of each one of its operations. Transactions are possible and safe to use given four properties: atomicity, consistency, isolation, and durability or ACID [81].

- Atomicity ensures that a transaction is one single unit of computation. All operations within the transaction must succeed if the transaction is to succeed. Otherwise, the transaction fails and all operations that have succeeded up to that point are discarded.
- Consistency ensures that a transaction is not committed unless all constraints for maintaining the data in a correct state are satisfied. Examples of incorrect state includes having a null value in a non-nullable column or a foreign key that does not exist or is in other ways invalid. The database is always left in a valid and consistent state.
- Isolation ensures that transactions do not affect each other until either has been committed.
- Durability makes certain that a committed transaction is permanently stored and will be preserved if the system was to crash.

An ACID-compliant DBMS provides users with strong consistency guarantees for their data. These guarantees, however, make it difficult to scale up by distributing the workload over several machines [10, 88]. To address the shortcomings of ACID, NoSQL databases have emerged.

2.1.2 NoSQL

There is no consensus on the formal definition of NoSQL, but it has been widely accepted to mean "Not Only SQL" [10]. NoSQL is an umbrella term comprising databases that oppose the relational model proposed by Codd. NoSQL gained popularity around 2010 [88] because of its performance and ease to scale horizontally (as in, distributing the workload over several machines) [10]. One of the main reasons why NoSQL databases have the ability to scale is that they often do not provide the guarantees of ACID by default or at all, sacrificing correctness to leverage greater performance [10]. MongoDB, one of the most popular NoSQL databases [36, 87] estimate that 80–90% of applications using their database do not need to make use of ACID transactions [78].

NoSQL databases are often put in one of four groups: key-value stores, graph databases, document-based stores, and column-oriented databases [68]. Key-value stores indexes keys pointing to data and is often suitable for caching purposes. Graph databases use graphs instead of tables as their underlying structure where queries are performed on nodes and edges. Document-based stores are an unstructured collection of documents. Documents are usually stored in a structured form but do not have to follow a specific structure or schema [10]. Column-oriented databases transpose the conventional row layout found in relational databases. In the row layout, each individual row and the contents of its columns are stored adjacently in memory and persistent storage [21] as illustrated in table 2.1, making the complete entry fast to retrieve. In contrast, the column-oriented layout visualized in table 2.2 stores all the values of each column next to each other for greater data locality. Data locality is highly beneficial, or crucial even, for high performance as it decreases the amount of time needed to seek the data on disk [37]. Data locality makes the column-oriented ap-

proach more efficient than the row-layout when it is necessary to make computations on a specific field or aggregate data. A simplified view of data locality is illustrated in fig. 2.1.

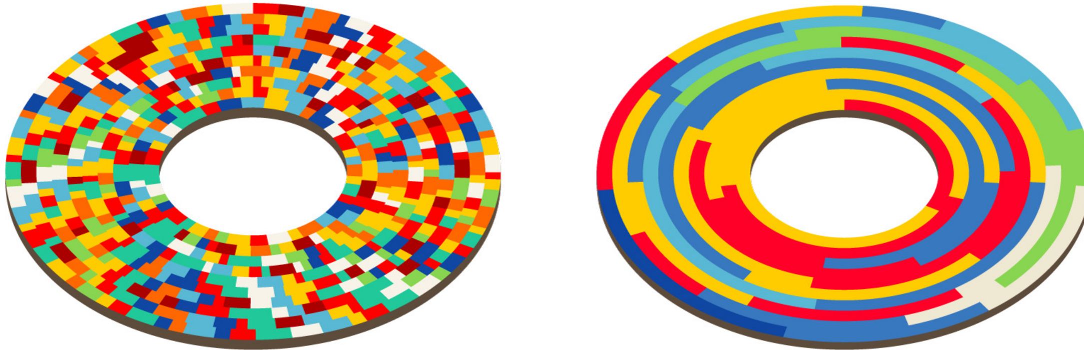


Figure 2.1: An illustration of data locality [21]. The colors symbolize homogeneous data, i.e., contents from the same column.

Consider the case where you are trying to get the average max temperature reported in a device fleet with model name *ABC*, using tables 2.1 and 2.2. A row-oriented lookup would have to fetch every row, checking to make sure that the `model_name` field matches, before deciding whether to include `max_temp` in the computation. The column-oriented database on the other hand, can traverse the `model_name` column immediately. If `model_name[index]` is a match, then `max_temp[index]` should be included in the computation.

Table 2.2: A column-oriented representation of the data in table 2.1. Each record is instead represented as a column, and attributes are stored adjacently in rows.

`model_name`	ABC	…	ABC	…
`serial_number`	123	…	456	…
…	…	…	…	…
`max_temp`	43.47	39.42	43.56	39.98
`min_temp`	12.36	10.06	12.55	11.63

CAP

While NoSQL databases benefit from not being ACID-compliant in terms of performance, they must still adhere to the CAP theorem [10] as a big part of the performance is usually achieved by adopting a distributed system. A distributed system in this case means using the hardware components of multiple servers (also referred to as nodes) to distribute the load between the total available processing, memory, and storage capacity.

The CAP theorem states that any distributed system, in case of a partition, i.e., a communication breakdown between nodes, can only provide two out of the following three guarantees: consistency, availability, and partition-tolerance [55].

- Consistency reflects the consistency of data between nodes. If a system is consistent all nodes will present the same data at the same time for all receiving clients [62].

- Availability means that any request to the system will be answered with a valid response without exception [62].
- Partition-tolerance means the capability to continue functioning despite the introduction of any number of partitions within the system [62].

Most NoSQL DBMSs choose to sacrifice consistency [10] as they usually opt for *eventual consistency* instead. This means that when a node is updated the system is initially inconsistent before the data has propagated through the whole system and all nodes have become consistent. During this process, the nodes can present different results to users of the system [55].

A system that opts for *strong consistency*, where all nodes always present the same data regardless of time and place, sacrifices availability as the system needs to block all access to the nodes until the propagation has completed [55].

Sacrificing partition-tolerance is a special case as it is practically only possible for single-node systems where there can be no partitions. Furthermore, sacrificing it for a distributed system is still a debated area as this would require a fault-free network between the nodes, and this is not a feasible assumption in 2021 [9].

The author of the CAP theorem, Brewer published a follow-up article in 2012 [9] where he emphasizes that the CAP theorem is not to be seen as binary where one of the properties is chosen completely on the behalf of the other. Instead, it is to be seen as a gradual range and modern databases should strive to maximize the combinations of consistency and availability to make sense for its specific use-case.

2.1.3 OLAP

Online analytical processing (OLAP) is a type of software system used to perform large-scale analysis at high speeds [85]. The term was first coined by Codd, Codd, and Salley in 1993 [30] as a counterpart to the conventional online transaction processing (OLTP) systems due to new requirements for data analysis being established. As OLTP systems are typically used to handle the day-to-day operations of a company they are optimized for handling smaller operations numbering in the thousands to millions from equally many users such as ATM deposits or updating the inventory of products [2]. Consequently, the underlying database in OLTP systems is most often a relational or ACID-compliant database where a mix of inserts, updates, and deletes are happening within transactions.

OLAP on the other hand focuses on data extraction for business intelligence purposes where data is rarely modified once it has been inserted [85]. Thus, they are primarily optimized for processing massive amounts of data needed for analysis with query intensive workloads, i.e., consisting mostly of heavy read operations, from much fewer users numbering in the tens and hundreds contrary to OLTP systems [2]. This is achieved by sacrificing the ability to alter the handled data, sometimes even restricting it to be read-only or append-only [2].

Codd, Codd, and Salley also popularized the use of the multidimensional conceptual view for OLAP systems [30]. In a multidimensional model, the objects of analysis are called *measures*. These are most often numeric to facilitate different kinds of aggregations such as the sum, average, max, or min functions [11]. Examples of this could be the number of sales of a product

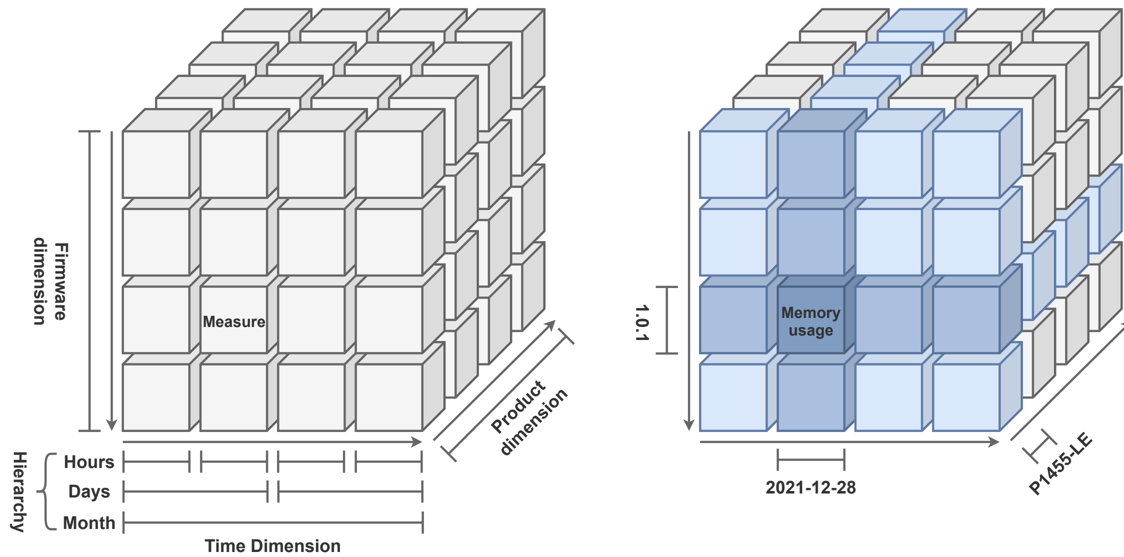


Figure 2.2: Demonstration of a data cube (Left) with a concrete example with the average memory usage for P1455-LE cameras using firmware version 1.0.1 on the 28th December 2021 (right).

or how much memory a camera is using. The measures are in turn dependent on so-called *dimensions*, these provide context for the measures. Examples include time and firmware, which enables the aggregation of sales for a specific product during a specific time, or how much memory a camera with a specific firmware is using [11]. There can be hierarchies within dimensions, often in the form of a one-to-many relationship. A good example of this is the time dimension where a month consists of days and days consist of hours. The standard way to describe the relationship between dimensions and measures is by the use of a *data cube* [11], as demonstrated in fig. 2.2. The data cube is also a suitable medium to represent some popular *OLAP operations* [11], namely the slice, dice, roll-up, and drill-down operations.

The slice and dice operations are used to select a subset of the data in the cube, where slicing discards a dimension (thus reducing dimensionality) while dicing applies filters to one or more dimensions, reducing their size [11].

Roll-up and drill-down are opposites, where roll-up means going up in the concept hierarchy and often looking at the bigger picture [2]. This could be a result of aggregating the measures in another dimension or grouping them in a certain way [2]. An example of a roll-up operation would be to group measures in the time dimension by months or years instead of days. A drill-down would go the opposite way in the hierarchy and look at data measured in smaller time increments, like minutes or seconds [2]. All OLAP operations are demonstrated in fig. 2.3.

2.1.4 Object storage

Object storage is a high-level abstraction for cross-platform data sharing [73]. Objects eliminate complexities and scalability issues found in block-based file systems. Each object is a self-contained unit comprising data, metadata, and identification [63, 73]. Object storage has become increasingly popular with the advent of cloud providers offering cheap and massively

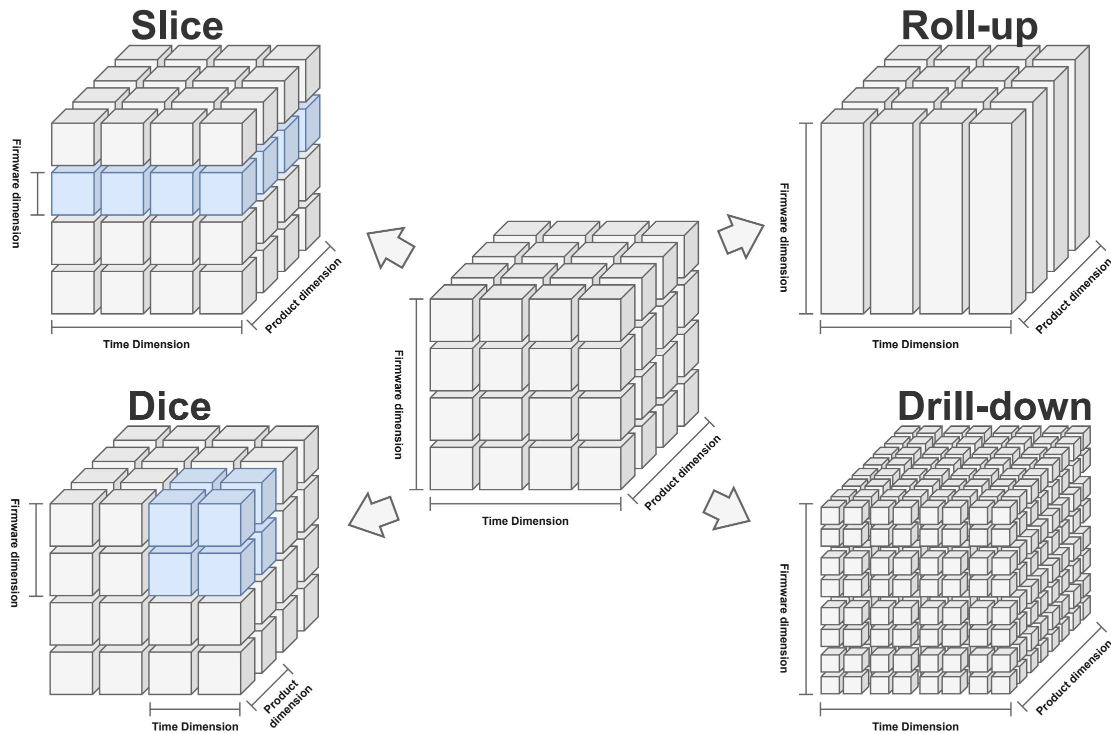


Figure 2.3: Demonstration of the OLAP operations on the data cube.

scalable [63] storage for images, videos, binary data and other unstructured data. AWS S3, launched in 2006, stored one trillion objects in 2012 [6] and 100 trillion objects by 2021 [7].

2.1.5 Data compression

Compression is a process used to reduce the size of input data by removing redundancy [83]. While it does play an important role in DBMSs simply by minimizing the disk space required for a given data set, there is a lot more to it than just storage space. Using compression, you increase the information density of your data, meaning that less data needs to be transferred to and from disk as well as between the client and server [58, 82], trading CPU time for higher effective bandwidth of both disk and network. Higher bandwidth will in turn improve the speed of query processing [58, 82]. Compression can also be used for indexes in a database, which helps speeding up query processing by more efficiently evaluating boolean expressions [67] and improving the memory utilization of indexes [57].

The importance of compression in DBMSs increases with growing amounts of data; even a small amount of compression can make a difference of hundreds of gigabytes if the input data is large enough. With larger data sets the amount of redundant data is likely to be higher, meaning that the compression can be more effective [83]. The importance of compression is also reflected in the query processing performance on that data, specifically the seek time, i.e., the time it takes to find the information asked for, grows with the amount of data needed to traverse before arriving at the correct place on the physical disk [58].

Another point of interest for compression is the data locality discussed in section 2.1.2 and visualized in fig. 2.1. A lower data cardinality (number of distinct values), and lower relative

entropy (difference between adjacent data points) leads to more efficient compression [1, 57]. Column-oriented DBMSs are especially suited to take advantage of this fact by storing the data column-wise, lowering the relative entropy [1].

Algorithms used for compression are typically classified as either lossy or lossless. A lossy compression algorithm will discard data that by some heuristic is considered less important, resulting in an approximate representation of the original data [83]. Lossless compression on the other hand preserves all the original data. When comparing compression algorithms, one is usually interested in the compression ratio (size of the output divided by the input size) [83] and the speed at which the data may be encoded or decoded. The freedom to discard data generally means that lossy algorithms can achieve a higher compression ratio than lossless algorithms [83]; for the purpose of this thesis, where data is archived and analyzed, lossless compression is the preferred choice.

In this thesis, we will focus on lossless algorithms only, specifically Snappy [86] and LZ4 [69] as those are already in use or will be used in the experiments presented later in this thesis. Common for these algorithms is that they aim for speed first and compression ratio second, making them viable for real-time compression.

2.2 Data storage products

It is clear that there are several distinct paradigms to follow on how to persistently store data, and with each paradigm, there are plenty of implementations available to choose from. In this section we present the products that are part of DDM's current setup and finally the contender: ClickHouse. Table 2.3 showcases the main differences between these products.

2.2.1 Elasticsearch

Elasticsearch by Elastic is a distributed search and analytics engine and NoSQL DBMS [41], based on Apache Lucene, a Java library providing "powerful indexing and search features, as well as spellchecking, hit highlighting and advanced analysis/tokenization capabilities" [3]. Elasticsearch is a document-based store designed to be fast for schema-flexible data (documents), expressed in JSON. Incoming data is indexed by default to provide responsive and easy-to-use search functionality [39] and Elasticsearch is built to scale by distributing the workload over several machines (i.e., horizontal scaling) in clusters [48].

Elasticsearch offers an HTTP API where the database can be queried using a query language based on JSON [46]. Optionally, one may use Elastic's scripting language *Painless* to perform computations on the query results as well as update the database. Painless is out of scope for this thesis and will not be explored further. Elasticsearch is part of the Elastic Stack and is commonly paired together with Kibana [45], a visualization product tailor-made for Elasticsearch, to create dashboards and visualizations to gather insights from business data.

2.2.2 MinIO

MinIO is an open-source object storage server implementation, designed to be compatible with S3, the object storage product offered by Amazon Web Services (AWS) [77]. It features a distributed mode, allowing you to pool together multiple drives and machines in one object storage server [75]. MinIO offers an HTTP API for transferring files and managing the server.

2.2.3 ClickHouse

ClickHouse is an open-source column-oriented OLAP DBMS designed for real-time analytics. It was initially developed at Yandex for their web analytics service *Yandex.Metrica* [12], but was later open-sourced [28] and, in 2021, spun out to a separate company [74]. ClickHouse is called a "true" column-oriented DBMS by its developers, because it stores no metadata along with the values [20]. This is claimed to be one of its biggest advantages for the OLAP scenario, as it minimizes the so-called unnecessary data and maximizes the processing throughput [20].

Similar to Elasticsearch, ClickHouse can be deployed as part of clusters. When deployed in a cluster, ClickHouse utilizes parallel and distributed query processing, and it is designed to scale linearly with added instances, meaning that there is a proportional increase in database throughput and/or storage when adding instances to the cluster [14]. ClickHouse clusters are coordinated by Apache ZooKeeper, an open-source server for coordination of distributed applications [15]. At the time of writing, the ClickHouse developers are working on their own coordination software to replace ZooKeeper, called ClickHouse Keeper [15].

The query language used in ClickHouse is a dialect of SQL that is largely compatible with ANSI standardized SQL, with additional functionality for analysis [20]. ClickHouse offers APIs to communicate with the DBMS over HTTP, native TCP, and gRPC [22].

ClickHouse is optimized for an OLAP workflow and has some limitations when compared to an RDBMS: no support for transactions, only partial (and nonstandard) implementation of UPDATE/DELETE queries, and it being inefficient at retrieving single rows of data [20].

Table 2.3: Comparison of data storage products.

	MinIO	**Elasticsearch**	**ClickHouse**
DBMS	No	Yes	Yes
Storage model	Object storage	Document store	Columnar
Query language	RESTful API	JSON DSL	SQL
Data compression	No	Yes (LZ4)	Yes (LZ4, Zstd)
Interfaces	HTTP	HTTP	HTTP, TCP, gRPC

2.3 Data analysis context at DDM

In 2019, Axis decided to put a larger focus on data collection and analysis, and created the department for Diagnostics and Data Management, or DDM for short. This means that a growing section of their device catalog is offered with cloud integration and opt-in device diagnostics, commonly referred to as telemetry.

This section will give a more detailed depiction of how the handled data is structured, how the current data handling is set up, and finally some examples of how the data is then used.

2.3.1 Current setup

As DDM is still in its infancy, the current solution that we will explain has been developed from what was originally conceived as a proof of concept (PoC). The original goal of the PoC was to optimize storage use (the proprietary storage back-end in use at DDM is comparatively expensive) while still being able to retain and analyze as much historical data as possible. As such, compromises were made that limits the amount of useful analysis that can be performed.

Data structure

The metrics data comprises hundreds of device diagnostics measures but the most important include memory usage and this is what our data set is centered around. The data contains four dimensions: time, product, executable, and firmware. The product dimension is a hierarchy of attributes, in order of increasing granularity: product type, serial number, and boot ID.

Data pipeline

As visualized in fig. 2.4, each unit in Axis' device fleet emit metrics at semiregular intervals that, given that the owner has opted in, are shared with DDM. Initially, metrics were directly inserted into an on-premise Elasticsearch cluster to enable analysis of the incoming data. Elasticsearch was chosen for its ease of use, speed, and familiarity among the data scientists at DDM. Using tools like Kibana, the Elasticsearch cluster can be queried efficiently from a web interface with the results presented as easily digestible visualizations and dashboards.

Soon, however, as the amount of data grew it became evident that Elasticsearch was not the ideal choice for meeting DDM's needs of both analysis and long term storage. With memory and storage usage increasing, so did also the cost for the backing hardware, and it was later decided too expensive to continue using Elasticsearch for all collected data. Instead, MinIO was introduced to act as long-term storage where all current and historical metrics could be stored. Elasticsearch is then loaded with the last three months of metrics stored in MinIO.

Before being inserted into MinIO, metrics are processed in a pipeline as seen in fig. 2.4. The pipeline processes the semi-structured metric data according to a schema and partitions it by the hour that they were emitted. Finally, the data is stored in an Apache Parquet file, a columnar storage format [4], and compressed using Snappy (yielding a compression ratio ranging from 2:1–20:1 depending on the type of metric). This process requires on average ten minutes per hour of incoming metrics.

To perform analysis reaching beyond the three most recent months, DDM developed their own processing scripts to access long-term metrics directly from MinIO. These scripts are developed in Python, using the popular data analysis library Pandas [32] to perform computations. We are not at liberty to disclose the exact workings of these processing scripts, but the pseudocode in listings 1 and 2 should be sufficient to understand the general idea. The scripts are divided into two parts as seen in fig. 2.5: extraction and analysis.

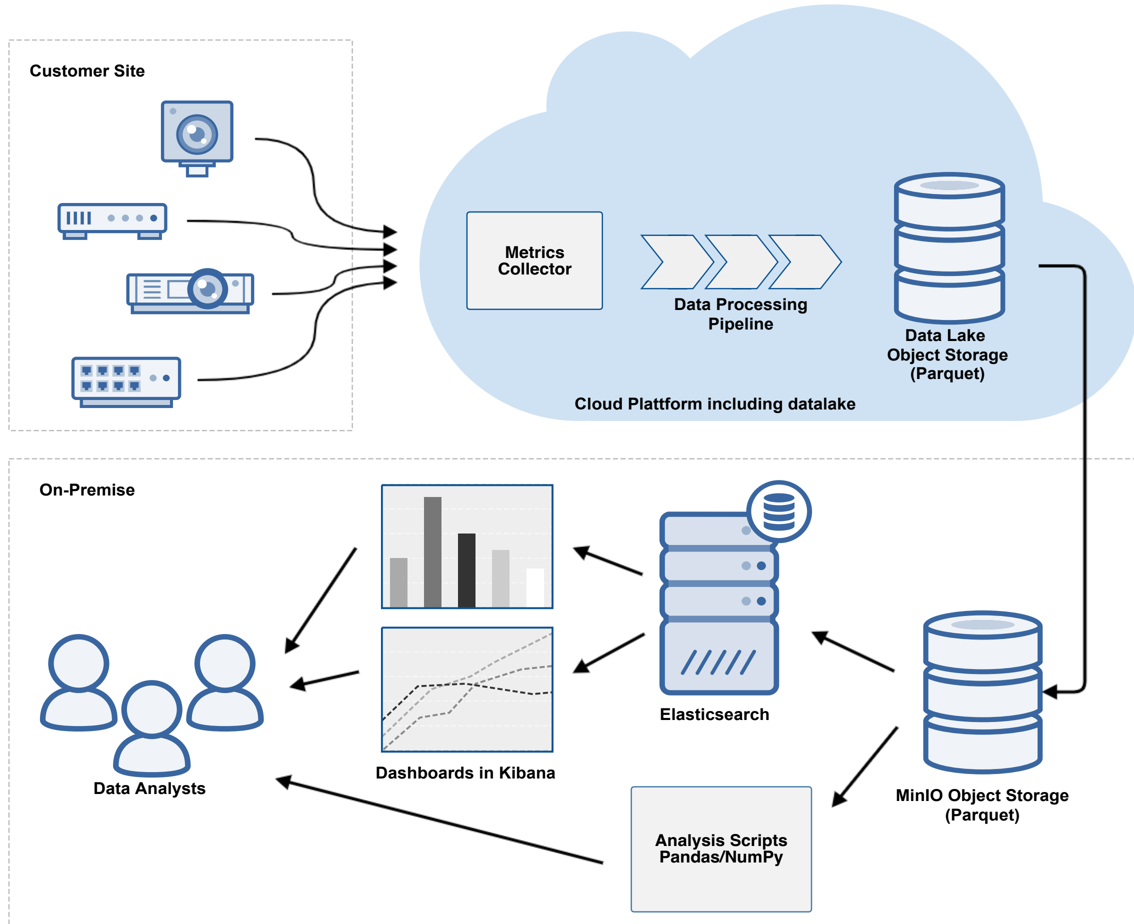


Figure 2.4: Overview of the big data processing pipeline at DDM.

```
for partition in storage:
    take valuable columns where EXPRESSION
    encode category, compression, metadata...
    write preprocessed partition to disk using new partition schema
```

Listing 1: The extraction part of the processing scripts.

In the extractor for the on-premise object storage, raw data, partitioned per hour, is downloaded and filtered to only contain the data needed for further analysis in order to maximize the possible amount of valuable data to be stored in memory. These preprocessed files are then stowed away in a temporary workspace where they can be used for analysis.

```
for partition in preprocessed partitions:
    load data frame (into memory) from partition on disk
    perform analysis on the data frame
```

Listing 2: The analysis part of the processing scripts.

The preprocessed results are then loaded into data frames, using Pandas, and executed. The entire pipeline process is visualized in fig. 2.4.

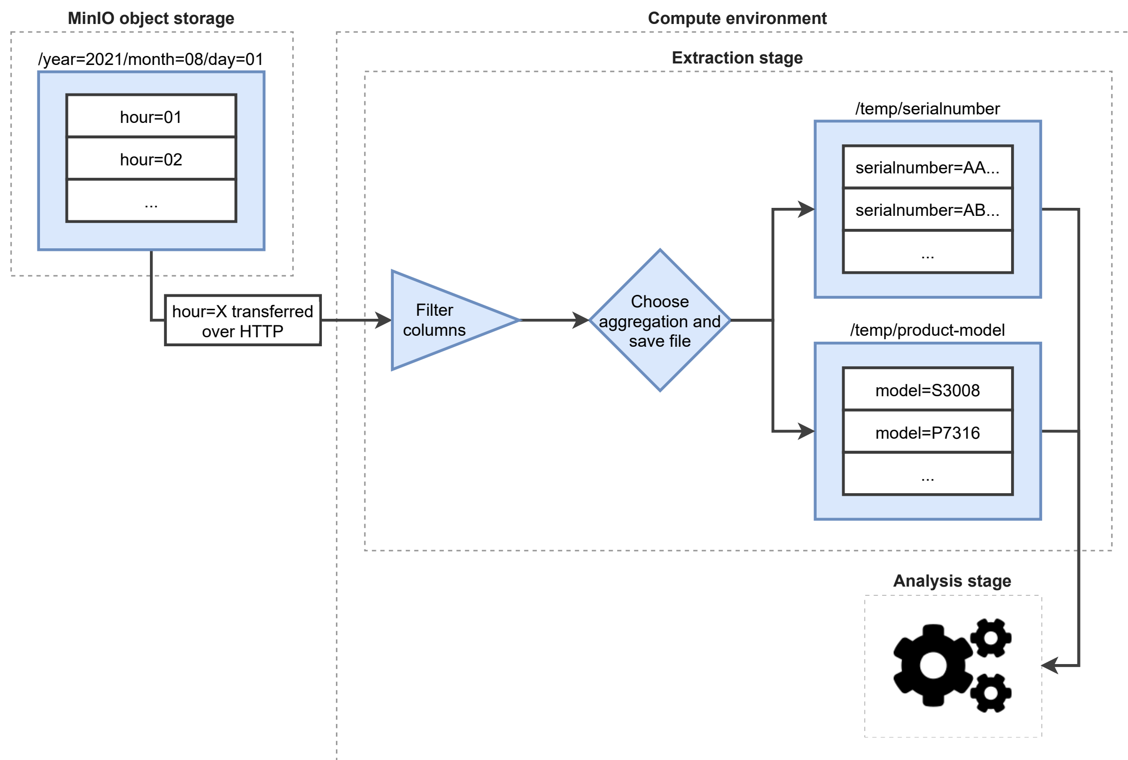


Figure 2.5: Compute pipeline from MinIO storage.

2.3.2 Use cases

The primary purpose of DDM is to provide other departments (the data consumers) with the necessary insights to manage their objectives, goals, risks, and problems. The term "necessary insights" is vague and means different things for most data consumers, as there are certain types of analyzes that are must-haves while some would only be considered nice-to-have. One of the more useful capabilities is to be able to draw correlations between different sets of data. Due to the inherent limitations of Elasticsearch at performing joins in the relational sense, drawing correlations between different sets of data requires much effort and consideration. It also requires duplicating some of the data, in other words its not something to be done carelessly when trying to minimize storage use. Along with the join limitations, there is also the need for data analysis over a longer period that simply uses too much memory in Elasticsearch forcing DDM to resort to their time-consuming analysis scripts. Percentiles, residuals, and techniques such as linear regression are difficult—or even impossible— to use in an online real-time fashion today.

Below is a list of five data analysis use cases for DDM. All data extractions are handled using Elasticsearch and Kibana dashboards today, and only the last use case makes use of MinIO. As our test data (further explained in section 3.2.2) contains memory usage metrics, we will focus on such use cases too. The use cases were chosen to give a good diversity of different data aggregations and use of the OLAP operations present in DDM data analysis.

These use cases will be explored further in our experiments as part of chapter 3.

UC1: Memory usage per executable and firmware

Firmware is generally developed in low-level programming languages on embedded devices with constrained resources. The combination of these factors makes memory leaks a severe risk. Consequently, metrics on memory usage within devices are of great interest to both firmware and plugin developers. A heat map displaying this sort of information can be seen in fig. 2.6 where executables are lined up on the vertical axis with the firmware version on the horizontal axis, creating a grid. The cell of each grid is gradually saturated as the memory usage for that combination of executable and firmware increases.

To put this in the context of OLAP this use case would, with memory usage as the relevant measure, utilize the roll-up operation as far as possible in two dimensions, i.e., the time and product dimensions. This reduces the dimensionality to two dimensions saving the executable and firmware dimensions. The aggregate function used in the roll-up operations can be changed according to the goal of the analysis, e.g., the *AVG* function would give a good overview of the memory usage for each combination while the *MAX* function could be used to find outliers. In fig. 2.6 the *AVG* function is used.

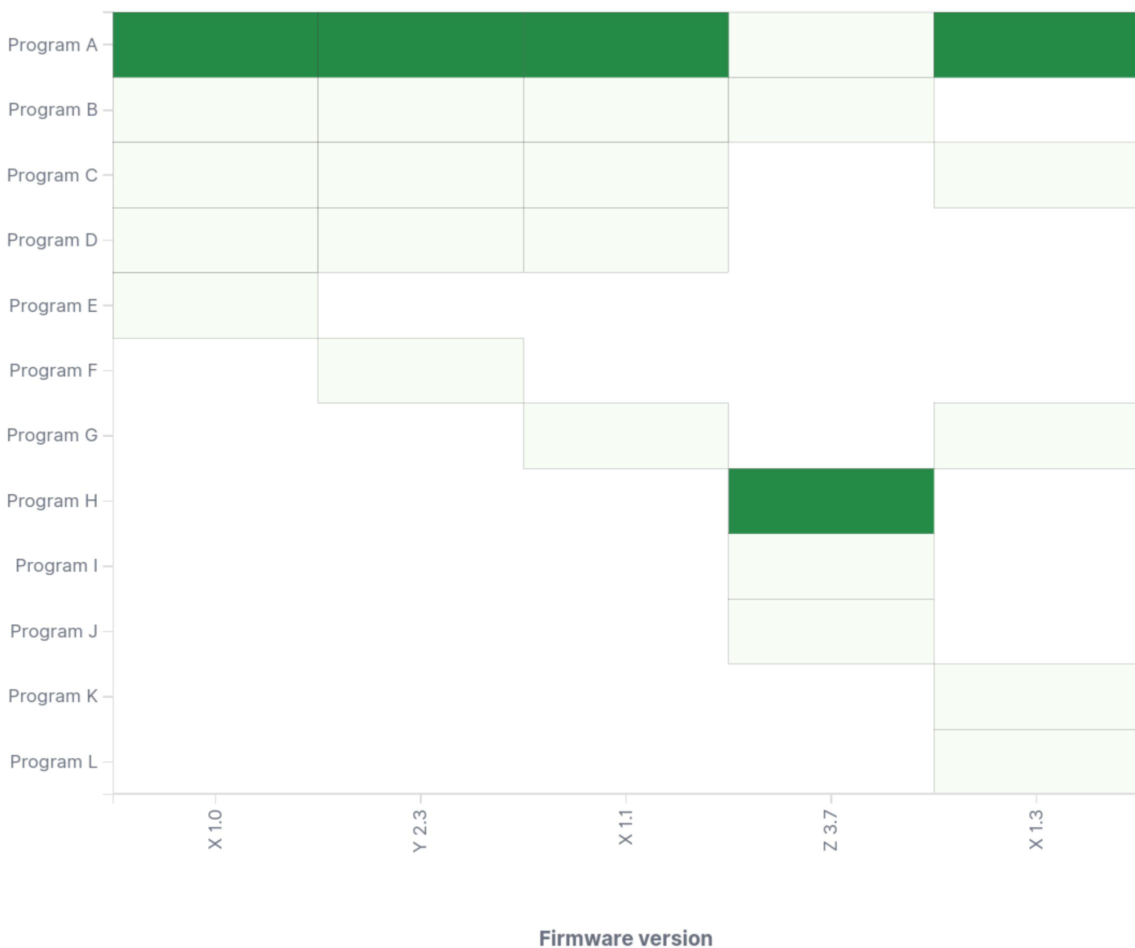


Figure 2.6: Heat map showing memory usage per executable and firmware. The color legend has been redacted.

UC2: Memory usage for one device over time

In fig. 2.7, the average memory usage on the entire device is shown over time, with one week passing between each tick on the horizontal axis. In a two-week interval, a slight reduction can be seen followed by an even bigger dip before the graph is restored to its earlier appearance. Investigating further into these dips, one may find that the device was rebooted due to power outages, memory leaks, or that the device was going through a firmware upgrade.

This use case first slices the data in the product dimension to filter out an individual camera. Then, it is rolled-up with the *AVG* function in the time dimension to group the data by weeks.

Figure 2.7: Average memory consumption for one device over time. An upward trend may indicate memory leaks, and may be identified by drill-down. Labels on the Y axis have been redacted.

UC3: Number of devices transmitting metrics

To verify that the metric collection works, and to just get a sense of the device distribution over time, one may consult the dashboard in fig. 2.8. From the dashboard one can easily see the number of unique devices, grouped by product model, that transmits metrics to DDM.

Here, the data is simply rolled up in the product dimension using a *COUNT UNIQUE* function to group all product types, and show the number of transmitting devices per hour.

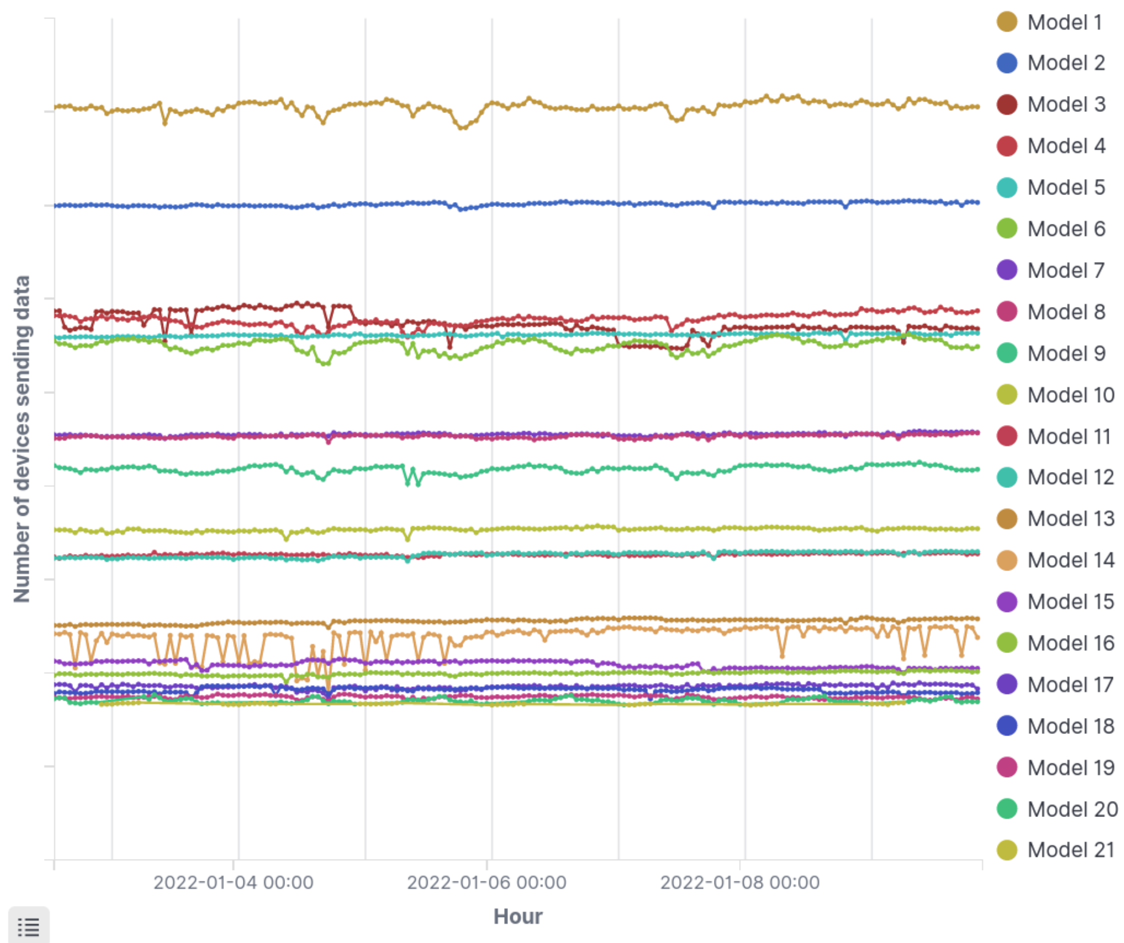


Figure 2.8: Interactive line graph displaying the number of unique devices transmitting metrics over time, grouped by product model. Labels on the Y axis have been redacted.

UC4: Memory usage for specific executable per product model

Axis has a wide offering of products, each with its own composition of hardware and software. To make sure that the firmware is functioning, it can be useful to track memory usage outliers over different models. In fig. 2.9, a visualization is seen displaying the average memory usage for one specific executable on the vertical axis and different product models on the horizontal axis. From fig. 2.9, we can see that there might be reason to investigate “Model 1”.

In the OLAP context, a slice is used to filter out a specific executable and the resulting data is rolled-up in the product dimension to individual product types with the *AVG* function.

UC5: Raw data extraction

A problem with databases, as reported by Madden [70], is that while they might be able to scale well for large amounts of data, they cannot offer as sophisticated analysis as client-side computation tools such as R [53] or Matlab [71]. For DDM to retain their advanced analysis scripts, they must be able to extract data that has already been ingested.

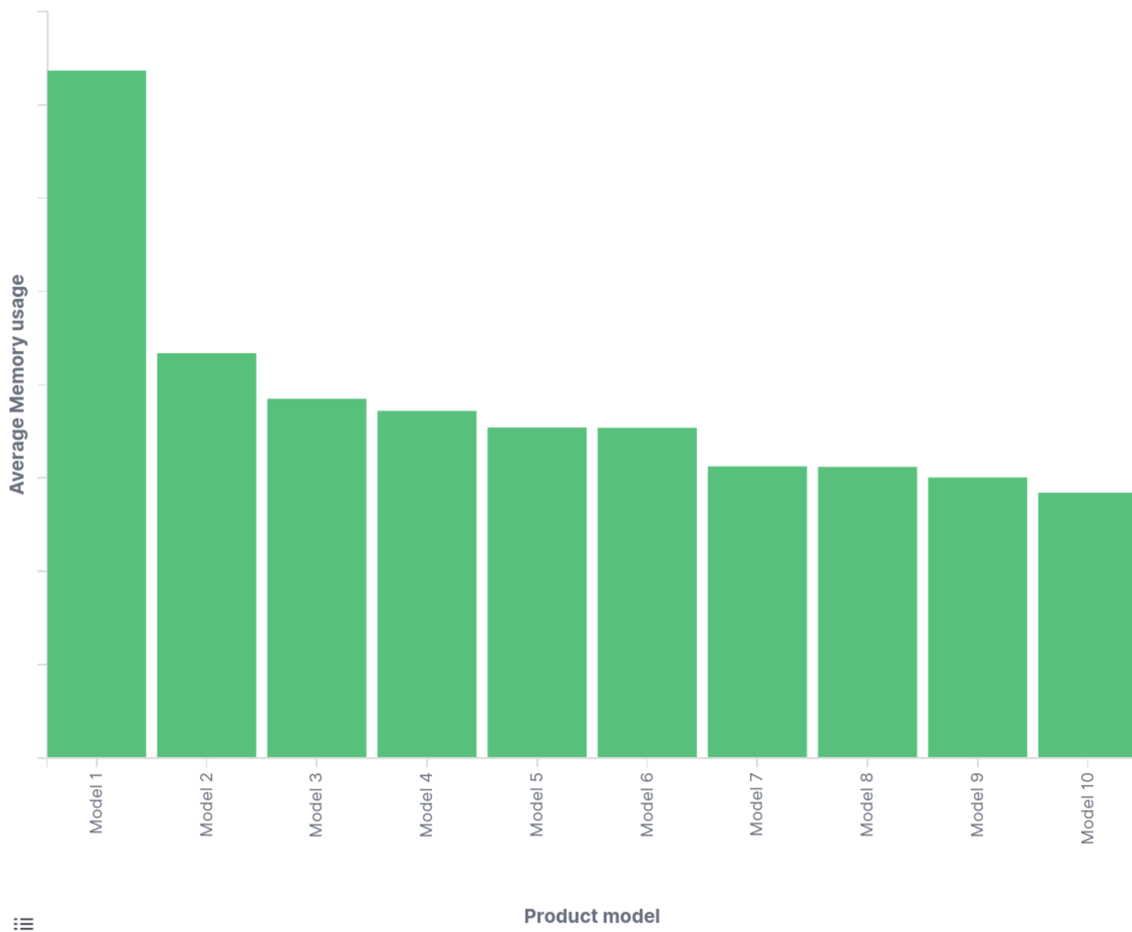


Figure 2.9: The average memory usage for one executable over different product models. Labels on the Y axis have been redacted.

Chapter 3

Method

The research methodology chosen for our research was *Experimental Research*. This methodology will aid us in answering our research questions in a formal and scientific way, giving reliability and validity to our results. This chapter gives some theoretical background on experimental research, an overview of our approach, and defines our experiment design. Lastly, we describe each of our experiments more in detail.

Experimental research is characterized by having a fixed design with three key elements [61]. It typically includes formulating a hypothesis based on the goals of the research, choosing independent variable(s) that can affect the result, and dependent variables that can be measured and compared. If a hypothesis is formulated, it needs to be testable so that the research results either support or reject it [61]. While the results primarily consist of quantitative data gathered by measuring the dependent variables, there can be room for different assessments regarding the interpretation of the data. To provide reproducible and legitimate results it is important that the experiments are carried out in a controlled environment where everything except the independent variables stays the same [61]. The methodology is suitable for studying the causes of different phenomena as well as comparing multiple technical solutions to each other in specific use cases [61]. The latter applies to this project, i.e., we design a suite of benchmarking experiments to compare different data storage solutions. Note that we do not express any formal hypotheses followed by inferential statistics in our work. Instead, we follow an empirical standard proposal by Hasselbring [60]. In practice, we used Hasselbring's proposed standard as a checklist to ensure that the essential attributes of benchmarking studies have been reported. Unfortunately, we will not be able to provide a replication package, proposed as one of the attributes, since many of the components used in the experiments are proprietary in some way.

Table 3.1: Overview of our experiments, mapped to relevant experimental variables involved in each experiment.

<table>
<tr><th colspan="2">Experiments</th><th>Dependent variable</th><th>Independent variable</th><th>Fixed variables</th></tr>
<tr><td colspan="2">Exp A: Ingest</td><td>Ingest Time</td><td rowspan="7">Experiment Subject</td><td rowspan="7">Hardware & Data set</td></tr>
<tr><td colspan="2">Exp B: Storage</td><td>Storage Space</td></tr>
<tr><td rowspan="4">Exp C: Extraction</td><td>Exp C.1</td><td rowspan="4">Capability & Extraction Time</td></tr>
<tr><td>Exp C.2</td></tr>
<tr><td>⋮</td></tr>
<tr><td>Exp C.9</td></tr>
</table>

3.1 Approach

To answer the performance part of **RQ1**, we analyzed the current implementation, i.e., Elasticsearch and MinIO at DDM in order to find any bottlenecks that could potentially be solved or at least mitigated by introducing another DBMS. To then answer how ClickHouse could mitigate the issues found in **RQ1** for **RQ2** we needed to know how well ClickHouse would perform in the same use-cases and context as the current implementations. Therefore, following our needs to answer **RQ1** and **RQ2** we chose to conduct the following three benchmarking experiments:

- Exp A: Ingestion time.
- Exp B: Disk space usage.
- Exp C: Data extraction (capability to perform the extraction and if so, the time it takes) divided into sub-experiments: C.1–C.9.

In table 3.1 we present an overview of the experiments as well as determining the experimental variables involved in each experiment. For brevity and ease of comprehension we will from here on out refer to our three experimental subjects: Elasticsearch, MinIO, and ClickHouse as **ELASTIC**, **MINIO**, and **CLICKH**, respectively.

The results from these three experiments will provide us with sufficient data to compare their performance within the DDM data processing pipeline. We will then conduct open interviews with data scientists at DDM based on the results in to find out how existing workflows will be affected by migrating to **CLICKH**, answering **RQ3**. The interviews will be used alongside our subjective experiences of conducting the experiments as a basis to answer **RQ1** in terms of usability. ISO 25010 defines usability as the "Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use." [66]. The results of our interviews as well as our interpretation will be presented in section 5.2 in the discussion chapter.

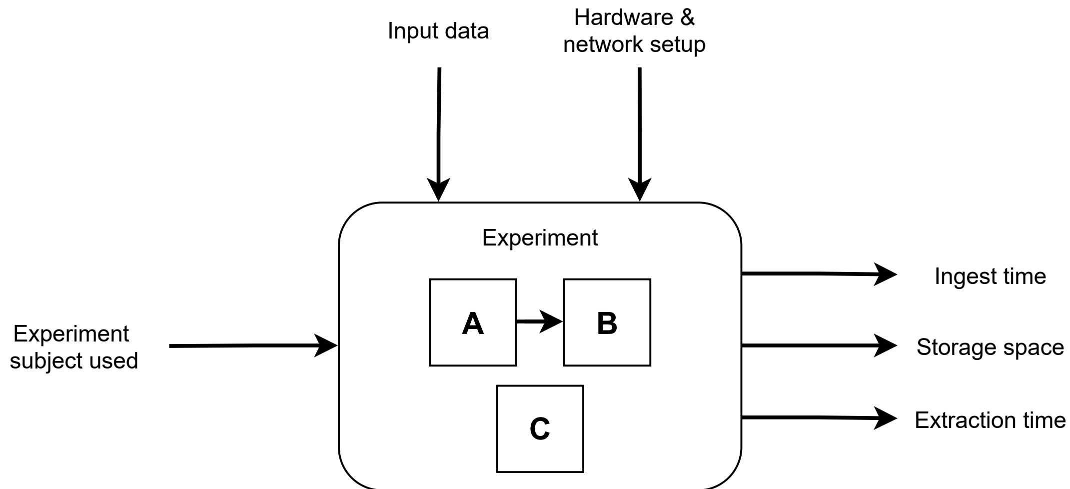


Figure 3.1: Overview of controlled experiment. The arrow shows that Exp B depends on Exp A succeeding.

3.2 Experimental design

In fig. 3.1 we give an overview of our experimental setup. Our experimental subjects, **MINIO**, **ELASTIC**, and **CLICKH**, are represented by the input arrow from the left depicting our independent variable for each experiment. The independent variable will be subjected to experiments A, B, and C. Time-consuming experiments A and B were repeated three times for each subject. Experiment C, however, is divided into nine smaller sub-experiments: C.1–C.9 and repeated ten times for each sub-experiment and subject. The results will be based on our 11 experiments and sub-experiments, yielding 33 experimental runs and 288 data points.

For all experimental runs, we used the same hardware and data sets, referred to as our fixed variables in table 3.1 and depicted as the vertical arrows in fig. 3.1. In the same figure, the dependent variables are displayed as arrows exiting the experiment.

The following subsections define our fixed variables: hardware specifications and data set, as well as our setup for each experimental subject. We also specify our benchmark design for each of the main experiments: A, B, and C.

3.2.1 Hardware specifications

To be able to fairly evaluate the aforementioned experimental subjects, we configured a dedicated server (henceforth **SERVER**) and a virtual machine (**CLIENT**) in an isolated and managed environment. Both machines were running Ubuntu 20.04 LTS and equipped with hardware according to table 3.2. Detailed hardware specifications can be found in appendix A.

Furthermore, Elastic recommends disabling swapping when running Elasticsearch [40]. Swapping is a mechanism used to free up memory by temporarily moving data to disk. Swapping can significantly impact performance [40] and as such, was disabled on **SERVER** for all tests.

Table 3.2: Technical specifications for **SERVER** (left) and **CLIENT** (right), both running Ubuntu 20.04 LTS.

	SERVER		CLIENT
CPU	Intel Xeon-G 5220R	CPU	12 vCPUs (Intel Xeon-G 6230)
RAM	96 GB	RAM	62 GB
Storage	13.5 TB SATA SSD (RAID0)	Storage	500 GB SATA SSD
Network	10 Gbit/s	Network	1 Gbit/s

3.2.2 Test data

The test data that will be used for all experiments consists of continuously sampled time-series metrics from Axis' devices detailing their memory usage. It is the largest set of metrics collected by DDM, with documents comprising 35 different attributes, collected at high frequency. Roughly one third of the attributes are strings and the rest are signed 64-bit integers. Only two string attributes are non-nullable: the device serial number and boot identifier. A timestamp attribute stores the date and time when the document was collected. The test data covers two full months of real metrics and amounts to approximately one terabyte of data over 11,880,623,162 documents from 110,471 unique devices. As described in section 2.3.1, the metrics are stored in Apache Parquet files that are compressed using Snappy with a compression ratio of approximately 2:1.

3.2.3 Experimental subjects setup

This section describes how we configured all experimental subjects before running any experiments, and how we run them in order to perform said experiments. For the exact setup commands used, see appendix B.

We used Docker to run all test subjects. Docker enables us to run applications in containerized form, where instead of installing and configuring the software and all of its dependencies, we download an official image that is ready to go with everything bundled together [38]. The image is launched in a sandboxed container, and in order to communicate with the outside world, we map ports on the host machine to forward traffic to ports in the container over a virtualized network interface. To utilize persistent storage, we mount volumes inside the container, thus creating a file system bridge between the host machine and the container.

MINIO

To replicate the object storage solution that is in use at DDM today, we used the official Docker image published by MinIO Inc. on Docker Hub. We used the latest release version at the time of testing, tagged `RELEASE.2021-11-03T03-36-36Z` [76].

There was no configuration to the MinIO server, all options were set to their default values.

ELASTIC

To get Elasticsearch operational, we followed the official documentation using Docker for a single-node deployment [43]. We used the latest version at the time of testing: 7.15.2.

In order to begin using Elasticsearch in production, some configuration is recommended for optimal performance. We followed the official recommendations for running Elasticsearch in production [44] and performed the following changes:

- Increase the limits on virtual memory mappings, to 262,144 memory map areas to avoid out of memory exceptions in Elasticsearch.
- Increase the limits on open files inside the docker container, to a maximum of 65,536 open file descriptors.

As already mentioned in section 3.2.1, swapping was disabled as well.

Elasticsearch indexes data to achieve high search performance. However, for search performance and error recovery, Elastic recommends limiting the size of said indexes [42]. In addition, for time-series data that is append-only, they recommend using so-called Data streams [47]. Data streams allow for automatic rollover when the index has reached a certain threshold in either size, document count, or age [47]. As a result of only performing the tests on a single-node deployment, we did not need to worry about the implications of replicas as we would in a cluster. We configured our deployment to use data streams with automatic rollover after 50 GB of data and even though no replicas were created since we only deployed a singe-node, we still choose to specify the number of replicas to 0 and the refresh interval to 60s per recommendations from Elastic as this might improve indexing speed [51].

CLICKH

We started a single-node ClickHouse server inside a Docker container using the official image published on Docker Hub [18]. We used the latest version at the time of testing: 21.8.11.4.

Before performing the experiment we needed to create a table in which we could ingest data into. For the exact table created, see appendix B. We defined a simple table structure, without optimizations, where all numerical attributes were defined as 64-bit integers, and all other attributes as strings. The only exception being the timestamp attribute, which we stored as a `DateTime`, and used to partition the table on a per-month basis, as is recommended by the ClickHouse developers [19]. The timestamp was also used to order the data as this felt intuitive for a time-series data set. The table engine chosen was MergeTree since this is considered the default engine for a singe-node deployment [24].

3.3 Experiments

All benchmarks were executed on **CLIENT** and test the performance of our experimental subjects, running on **SERVER**. For each benchmark, only the actual test subject were running.

3.3.1 Exp A: Ingest

To keep up with the amount of incoming data every hour, a database must be able to ingest data at a reasonable speed. To test the ingest performance of each experimental subject,we measured the number of documents or rows the server can receive per second.

The tests were conducted by transferring data from the file system on **CLIENT** to each experiment subject running on **SERVER**. To make the workload sustainable for all tested solutions, we limited the input data to a subset of the original data. We conducted preliminary testing to find a subset that would require at least 20 minutes to ingest in order to saturate the link bandwidth and reduce the impact of potential spikes in the measurement. To simplify testing, the subset should not take longer than ten hours for any subject to ingest. With these requirements in mind, we chose 10 days of data which amounts to 152 GB or 1,734,529,007 documents stored in 696 Snappy compressed Parquet files.

To test ingestion speed, we set up each implementation as described and performed three ingest tests for each subject. All data was cleared between runs and the DBMS was restarted.

MINIO

To perform the ingest benchmark, the tool Rclone was used. Rclone was chosen because it supports a wide array of storage providers, including MinIO, and is easily ran in parallel [35].

```
$ rclone copy --transfers 12 --checkers 12 \
      /path/to/local/data \
      minio:bucket/path
```

The options `transfers` and `checkers` sets the number of threads to use for data transfers and checksums. To maximize the transfer speed, all twelve cores on **CLIENT** should be running in parallel.

The argument `minio` is the destination *remote*, followed by a colon and a path argument. Remotes are set up by running `$ rclone config`.

ELASTIC

To ingest data into the Elasticsearch instance, we had to make use of the existing transfer system, an extract-transform-load (ETL) pipeline, based on the official Elasticsearch Python library. The pipeline extracts data from the file system, parses it into Pandas data frames that are then transformed into Elasticsearch-compatible JSON. The JSON is then inserted into the Elasticsearch server. To maximize performance, the ETL pipeline transfers data in chunks, performs all work in parallel and utilizes all available CPU cores.

CLICKH

ClickHouse has native support for ingesting Parquet files, greatly simplifying this benchmark. Ingesting is just a matter of passing the data to the client's standard input.

```
$ clickhouse-client \
    --query="INSERT INTO thesis FORMAT Parquet" < data.parquet
```

To insert multiple files, `cat` may be used to concatenate the input, which is then piped to the ClickHouse client for ingestion.

```
$ cat *.parquet | clickhouse-client \
    --query="INSERT INTO thesis FORMAT Parquet"
```

However, passing too many files to the client program too quickly will overload it and cause it to crash. Another thing to note is that the client is not running its queries in parallel[1]. To achieve parallel execution, we used GNU `xargs` to spawn multiple clients, each inserting one parquet file. Due to memory constraints of **CLIENT**, we could not run one ingest job per processor core and still guarantee full ingestion. In addition, to successfully insert all files, the memory limit per query had to be increased from the default of 10 GB. The setting `max_memory_usage` can be set in a configuration file to be persistent or per session as an argument to the client program. A value of 0 disables the limit, which is what we used as a session parameter while ingesting data. The finalized insert command can be seen below.

```
$ ls *.parquet | xargs \
  -P8   \ # 8 processes at a time.
  -I{}  \ # Use `{}` as the placeholder for arguments.
  sh -c \
  'cat {} | clickhouse-client --max_memory_usage=0' # --query=[...]
```

3.3.2 Exp B: Storage

The aim of implementing a MinIO storage server was primarily to minimize storage cost. This will still be of considerable importance when considering a new DBMS and, because of this, we will compare the disk usage of our different implementations. We will measure this using the `du` command, a GNU coreutils program that reports how much of the file system space is used by the specified set of files [56].

Storage tests were conducted by initially verifying that the mounted data path was relatively empty before ingesting data. Relatively because each database will persist data to disk immediately upon startup, however no more than around 100 MB. Further storage tests were conducted after ingest tests and were measured immediately after ingest as well as two hours after the ingest was done. The two-hour wait is to take into account the effects of compression performed by both ClickHouse and Elasticsearch.

3.3.3 Exp C: Extraction rate

The data extraction rate is an important metric to measure how long it takes to perform various analytical tasks using each experimental subject. To enable measurements of extraction rate, we cleared each implementation, removing the data that had been stored for experiments A and B, and instead loaded them with our full test data set described in section 3.2.2.

To ensure measurements that are representative for DDM's use cases, we created database queries that we believe offer a diverse set of results using the same source data. They should give a suitable representation for each of the different OLAP operations mentioned in section 2.1.3. This section describes the experiments. Full queries can be found in appendix C.

For each experimental run, only the tested experimental subject was running on **SERVER**, and the relevant benchmark suite on **CLIENT**. Spot-checks were performed with the process viewer `htop` [31] to verify that the benchmarks were running and utilizing CPU as expected.

[1] Parallel reading is supported with ClickHouse 21.12 [26].

Exp C.1–C.8

Experiments C.1–C.4 map to use cases UC1–UC4 (section 2.3.2) respectively with one week of metrics. This gives us a good mix of data analysis present in DDM's data analysis pipeline. Exp C.5–C.8 are variations of C.1–C.4, but instead query the full test data set of two months.

We tested **ELASTIC** and **CLICKH** by writing queries in their respective query languages and measured the time it took to get the complete response. To communicate with the HTTP API exposed by **ELASTIC**, we used the command line tool `curl`.

```
$ curl \
    -d @query.json \ # Load request data from the file `query.json`.
    -H 'Content-Type: application/json' \ # HTTP header.
    "$ELASTIC_URL/index_name/_search"
```

For **CLICKH**, we used the official ClickHouse client.

```
$ clickhouse-client \
    -h "$CLICKHOUSE_HOST" \
    --query "$SQL_QUERY"
```

For Exp C.4 and C.7 (UC3 with different amount of metrics), we had to disable the query memory limit by setting `--max_memory_usage=0` like we did for **CLICKH** in Exp A.

In this series of experiments, **MINIO** was immediately disqualified. While they are possible to test, the analytics scripts are not designed for real-time online analysis, but rather historical trend analysis for source data spanning several years. Any of our test cases would consume at least one hour using **MINIO** as data storage back-end, and they would need custom scripting.

Prior to each test, caches were cleared. To make sure, we waited two minutes before testing.

```
$ clear-subject-cache \
    && ssh "$SERVER" sh -c \ # Clear SERVER's system cache.
    'sync && echo 3 > /proc/sys/vm/drop_caches' \
    && sleep 120
```

To clear the **ELASTIC** cache, we sent a POST request to the API endpoint `/_cache/clear`. And for **CLICKH**, we used the ClickHouse client to execute the three following commands: (1) `SYSTEM DROP MARK CACHE`, (2) `SYSTEM DROP UNCOMPRESSED CACHE`, and (3) `SYSTEM DROP COMPILED EXPRESSION CACHE`.

Exp C.9: Raw extraction

Experiment C.9 maps to use case UC5. To perform this experiment, we decided to fetch ten hours worth of data from our two-month range. This should contain 85,939,941 documents over 7.7 GB when fetched as Parquet files.

For **MINIO**, we again used Rclone to test the raw transfer speed. Because metrics in MinIO are partitioned by the hour that they were processed, the timestamp for each individual entry (corresponding to when the metric was emitted) may not match. Metrics may leak into other partitions if the processing is lagging behind. To ensure that we extract all metrics for the

first ten hours, we fetch the eleventh hour as well. It is the responsibility of analysis tools that operate on the metrics to filter out unwanted timestamps.

ELASTIC does not support explicitly dumping records. Instead, we used the official Python library to scan through the database and store the resulting JSON files.

With **CLICKH**, we again decided to use the official ClickHouse client to select all data (in Parquet format) and redirect the output to a local file.

Chapter 4

Results

This chapter presents the collected results from our experiments defined in chapter 3. These results will help answer our research questions: **RQ1** to **RQ3**. The raw data for each experiment run that we base these results on is found in appendix D.

In chapter 5 the results are further investigated and discussed to evaluate our findings.

4.1 Exp A: Ingest

In this experiment, we measured the time required to transfer 152 GB or 1,734,529,007 documents from **CLIENT** to **SERVER** and ingest into each experimental subject. The transfer rate is presented in table 4.1 and fig. 4.1. The measurement stopped when the transfer program exited on **CLIENT** and does not necessarily measure the time required for all data to be available for querying on **SERVER**.

Table 4.1: Ingest speed in Exp A. Higher is better.

Experimental subject	Documents/s	Mbit/s	Relative
MINIO	1,249,660	834.9	100%
CLICKH	861,663	575.9	69%
ELASTIC	38,606	27.06	3%

4.2 Exp B: Storage

When testing the ingest rate, using a subset of the data, we also measured the resulting disk usage. The averaged storage results are presented in table 4.2 and fig. 4.2.

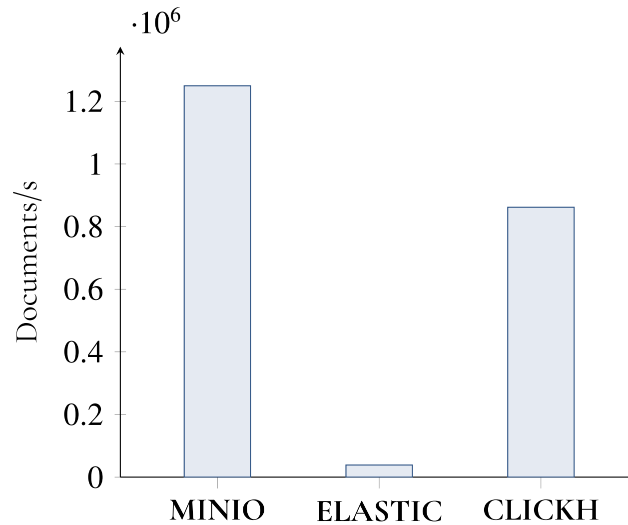


Figure 4.1: Results from Exp A. Higher is better.

To provide enough time for **CLICKH** and **ELASTIC** to compress the data and clean up intermediate files, we measured both immediately after ingest and after two hours. Measurements immediately after ingest are noted with parentheses in table 4.2 and are not shown in fig. 4.2.

We were interested in if the disk usage would scale accordingly with an increased amount of data; After having finished ingesting data for Exp C, we measured the disk usage again and found the relative disk usage between the experimental subjects to be almost the same. N.B., this is purely an observation based on one ingest run, and not part of any defined experiment.

Table 4.2: Disk usage in Exp B. Lower is better.

Experimental subject	Disk usage	% of source data
MINIO	152 GB (n/a)	100%
CLICKH	186 GB (336 GB)	122% (221%)
ELASTIC	621 GB (624 GB)	408% (410%)

4.3 Exp C: Extraction rate

Our results for experiment C and its sub-experiments, previously presented in section 3.3.3.

- Exp C.1: Memory usage per executable and firmware.
- Exp C.2: Memory usage for one device over time.
- Exp C.3: Number of devices transmitting metrics.
- Exp C.4: Memory usage per executable and product model.
- Exp C.5–C.8 (Exp C.1–C.4 respectively with two months of data).
- Exp C.9: Raw data extraction.

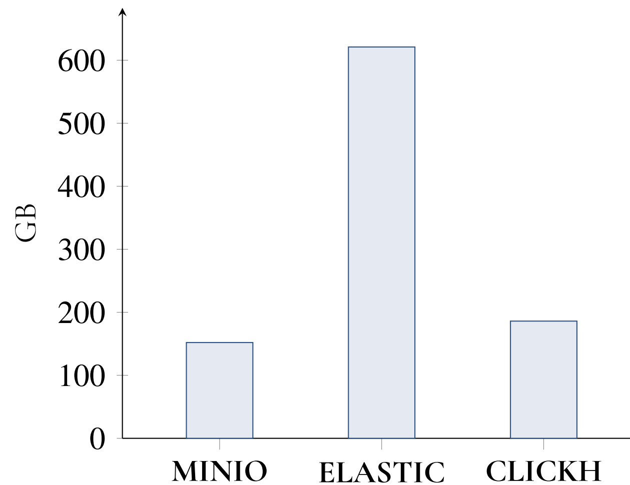


Figure 4.2: Results from Exp B. Lower is better.

Results for Exp C.1–C.8 are presented in table 4.3 and fig. 4.3 with results for Exp C.9 in table 4.4 and fig. 4.4. The figures and tables show the averaged execution time over ten runs. Additionally, the standard deviation is presented in table 4.3. As mentioned in section 3.3.3, **MINIO** was not tested in Exp C.1–C.8 and as such, is not shown in table 4.3 and fig. 4.3.

Despite our efforts to clear Elasticsearch caches before each benchmark run, there were large differences between the initial and following runs in experiments C.2 and C.4 for **ELASTIC**. Given the performance of **ELASTIC** in other tests (in particular the standard deviation), we suspect that only the first sample is representative, and the rest are incorrectly cached. However, the average of what we suspect are cached results in C.2 are still approximately 17 times faster than the results in C.4 that we also suspect are cached. This might be because the query in C.4 is more difficult to parse or cache, or it might be because these results are actually correct, and the initial result is flawed, though we can not explain why. In table 4.3, the results for C.2 and C.4 are presented without the outlier, which is noted in parentheses instead. We refer readers to the raw results in appendix D.

In experiment C.7, **CLICKH** failed because the server ran out of usable memory, returning an error instead of a valid result. This error was consistent over ten runs, so it seems that the query is not possible to execute in ClickHouse, at least given the table structure we have implemented. We discuss possible optimizations to the table structure in chapter 5.

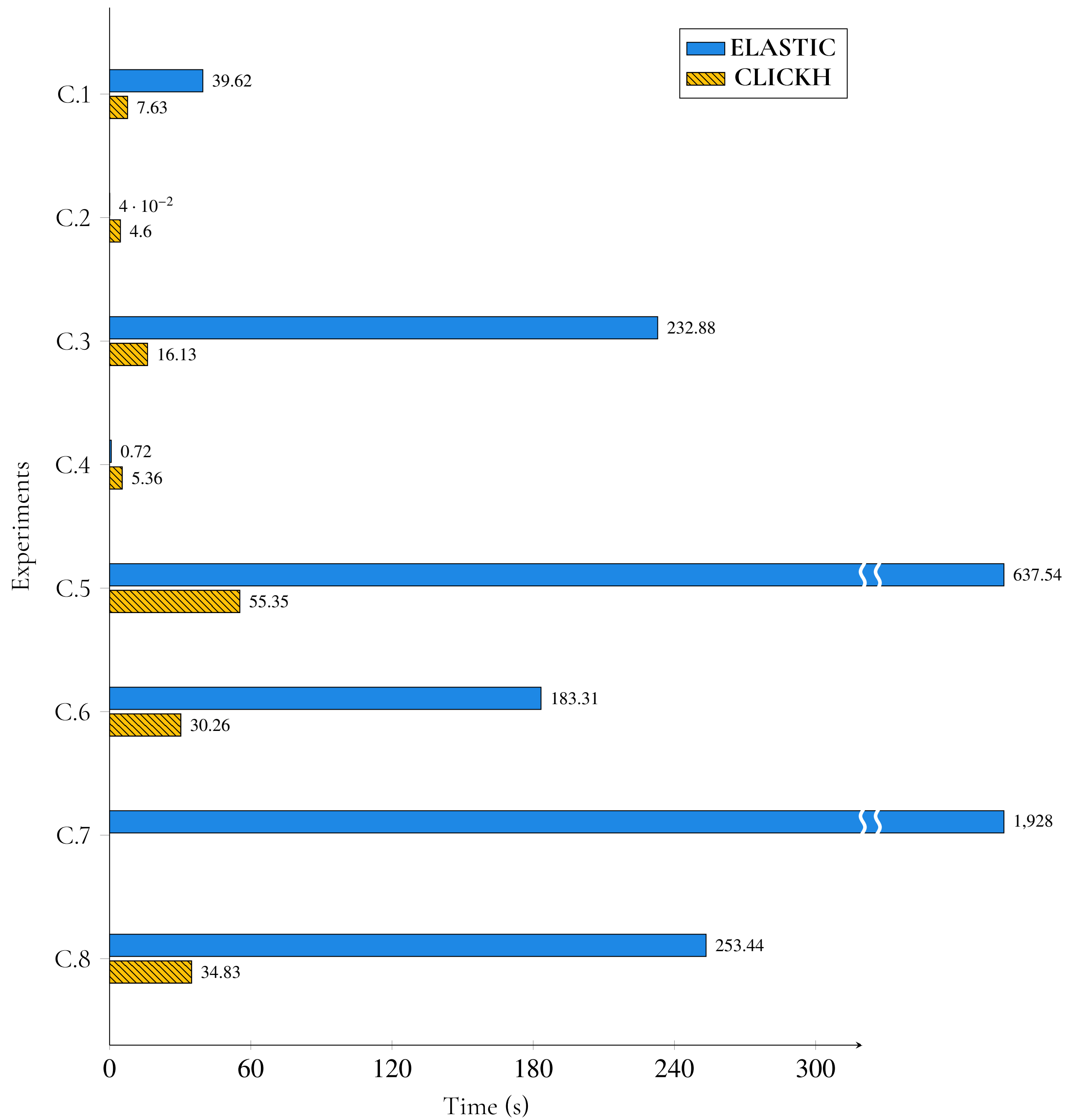


Figure 4.3: Results from experiments C.1–C.8. Lower is better.

Table 4.3: Experiments C.1–C.8. Results are measured in seconds and are averaged over ten runs. Lower is better. Values in parentheses are outliers.

	ELASTIC	(std)	**CLICKH**	(std)
Exp C.1	39.62	0.26	7.63	0.39
Exp C.2	0.04 (3.70)	0.003	4.60	0.12
Exp C.3	232.88	2.63	16.13	0.20
Exp C.4	0.72 (5.76)	0.01	5.36	0.26
Exp C.5	637.54	2.87	55.35	0.34
Exp C.6	183.31	10.05	30.26	0.05
Exp C.7	1,928.60	32.17	Failed	Failed
Exp C.8	253.44	15.85	34.83	0.07

Table 4.4: Exp C.9: Raw data extraction speed. Higher is better.

Experimental subject	Documents/s	Mbit/s	Relative
MINIO	1,207,870	865.78	100%
CLICKH	255,774	183.33	21%
ELASTIC	154	0.11	0.001%

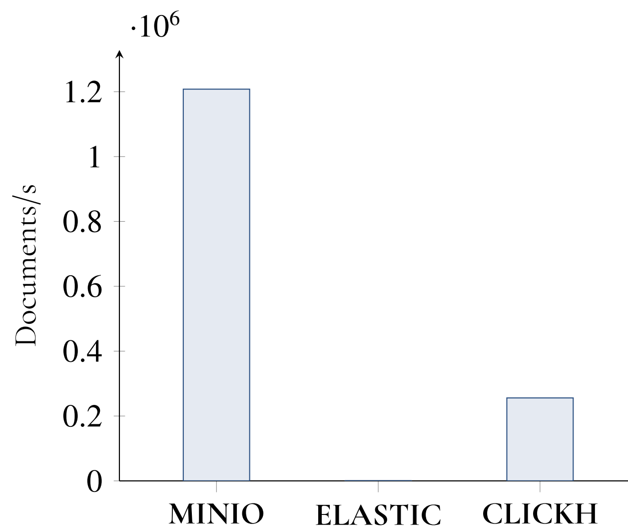


Figure 4.4: Results from Exp C.9. Higher is better. Note that the bar representing **ELASTIC** (with a value of 154) is barely visible.

Chapter 5

Discussion

In this chapter, we evaluate the results of our experiments to answer the research questions we formulated in chapter 1 and propose areas for future work.

5.1 Benchmarking experiments

The benchmarking experiments defined in section 3.1 were conducted to give an indicator on the current state of DDM's big data solution (Elasticsearch and MinIO) and how ClickHouse would perform in its place. Together, the results answer the performance part of **RQ1** as well as **RQ2**. In the following section, we dive deeper into the results gathered from each experiment and try to give thoughts on why the results looks like they do and what impact this has on a big data processing pipeline.

5.1.1 Ingest performance

The first performance indicator likely to be encountered when evaluating different data storage back-ends is how well it handles ingesting large amounts of data. Recall two of the Vs of big data: volume and velocity, the latter of the two being tested here. From the results in chapter 4, we saw that ClickHouse fared well at handling large-scale data fast, confirming the findings of our related work in section 1.3, especially that of Vasile, Avolio, and Soloviev [90].

ClickHouse could ingest documents at almost 70% of the speed compared to MinIO. It should be noted that MinIO is basically a file system transfer over HTTP, and while the protocol will induce a slight overhead, the results should be very close to the capacity of the 1 Gb/s link. When compared to Elasticsearch—which only achieved 3% of MinIO's speed—we saw an impressive 21x speedup.

Exactly why Elasticsearch performed so poorly is unknown to us. The ETL pipeline described in section 3.3.1 is highly parallelized, and we have verified that the transform step from Parquet files to JSON documents introduces marginal overhead compared to the work spent loading/ingesting the data (in bulk, as recommended by Elastic [51]). One caveat, however, with Elasticsearch's ingest is that "When a document is stored, it is indexed and fully searchable in near real-time" [39], though in our case, this "refresh interval" was configured to be 60 seconds. While it is certainly a feature to have all data be near-instantly searchable, it might be detrimental to performance when ingesting large amounts, as in our experiment. We did not verify how fast results became available in ClickHouse, though we did execute a count query following each ingest to verify that all documents were ingested, which they were. Still, we cannot say whether the input data was actually queryable at that time.

During setup of the ingest experiment, we found that the limiting factor for ClickHouse when ingesting in parallel was the client's memory usage. When passing too many files at the same time to the ClickHouse client, it would crash. Consequently, to guarantee full ingestion, we had to limit the client to use only eight parallel processes instead of the available twelve. After performing the experiments, a new version of the ClickHouse client was released, with built-in support for parallel inserts. This could result in greater ingest performance as this is a purpose built tool from the developers at ClickHouse. In addition, it would have greatly simplified setting up the conducted benchmarking experiment.

When ingesting the full test data set for experiment C into Elasticsearch, the ETL-pipeline that we had successfully used during our ingest experiments crashed after having processed approximately ten percent of the data. We are not certain about why it crashed, but given our time constraints and Elasticsearch's poor ingest performance we decided it was best to restart the ingestion process from the position where the crash was encountered. This resulted in duplicated data that, as a consequence of how Parquet files are laid out (timestamps of documents in a file do not necessarily match the timestamp of the file), were spread out but mostly concentrated to one hour. To minimize their impact on our results, we chose to delete all metrics for this particular hour in all three experimental subjects. Duplicates were still present after having deleted this hour of metrics, again, due to the Parquet archiving process. After deletion, there was a 0.00066% difference in the number of documents between MinIO and ClickHouse, where MinIO contained the most documents and ClickHouse the fewest.

5.1.2 Storage

Our results show that ClickHouse, again, performs very similar to MinIO, with only a 22% increase in storage use compared to the source Parquet files. The same cannot be said for Elasticsearch, measuring more than 4x the size of files stored in MinIO. When comparing the results of ClickHouse and MinIO (using Parquet) with the results for Elasticsearch it confirms the impressive compression efficiency of column-oriented storage covered in section 2.1.5.

We did see quite a large difference between the disk usage for ClickHouse immediately after ingest and after two hours had passed. It is evident that ClickHouse prioritizes ingesting the data fully before doing much optimization, this means that it can probably do a better job when it is time to optimize. Recall from section 2.1.5 that the efficiency of data compression generally increases with the amount of raw data. Elasticsearch, on the other hand, did not

improve the disk usage much over time, which could mean that it tries to optimize immediately when ingesting. This could also be part of the reason why it was remarkably slower than ClickHouse at ingesting data as already discussed in section 5.1.1. MinIO is not applicable to this comparison as it does not try to optimize the data neither during nor after ingest.

The numbers for ClickHouse are very impressive, given that we have not paid any attention to optimizing the structure of our table and its attributes. Even so, ClickHouse manages to analyze the data that it is given and optimizes it well. This is promising, as it allows users to start off with a naive configuration and optimize it further as they go along. The single most important choice we made regarding storage use for our ClickHouse deployment was which key we used to partition the data by as well as by which keys we ordered the data. Our choice of partitioning by the time dimension on a per-month basis was by far the most intuitive as the data set consists of time-series data. This is also recommended by ClickHouse as to not partition with greater granularity than needed, hindering compression [19]. As covered in section 2.1.5, data locality influences the compression rate and by partitioning the data by a less appropriate key would have a large negative impact on our results.

We believe there are several optimization techniques that could be applied to our ClickHouse setup for greater results, both in terms of storage and execution time. Firstly, choosing adequate data types for our attributes would allow ClickHouse to compress the data further. Having attributes being both nullable and 64-bit signed integers may be too permissive, given the domain of the data. Furthermore, there are columns that have very low cardinality (that is, the set of distinct values is small), such as the executable name for a process. ClickHouse has built-in support to encode such information, whereby it will use more efficient compression. With low-entropy ("predictable") values, it is possible to use specialized compression codecs. For time-series data, such as the data used in this thesis, ClickHouse recommends using the `DoubleDelta` codec on the timestamp column for optimal compression rate [25].

With Elasticsearch, few optimizations have been implemented at DDM, and carried over to our experiments, to manage increasing storage needs. The most meaningful is using data streams where append-only indices roll over after reaching a size of 50 GB, and are automatically removed after 90 days. Additionally, the segments within indices are forcefully merged together and shrunk to remove redundant data and increase search performance. These optimizations are recommended by Elastic in general [50] and have been recommended specifically for DDM by Elastic consultants. Additional improvements could be made for Elasticsearch in terms of storage use, such as enabling the DEFLATE codec for better compression instead of the default (LZ4), and disabling full-text indexing for strings (used for search). These improvements have been tested at DDM, but were considered to be too large compromises of the usability of the Elasticsearch workflow, and have not been implemented.

5.1.3 Extraction rate

Similar to our previous experiments, we saw that ClickHouse performed very well. In subexperiments where both succeeded with reproducible results, ClickHouse was 5–14x faster than Elasticsearch. When comparing raw data extraction, ClickHouse was barely five times slower than copying the data directly from MinIO, while yielding comparably small, Parquet formatted, output files. The files we stored from Elasticsearch were unmodified JSON

responses from the search query API, that were saved to disk in batches. The main benefit with both ClickHouse and Elasticsearch compared to MinIO is of course the ability to filter data before extracting it. With Parquet files stored in MinIO, you can only filter on partitioned columns, whereas with ClickHouse and Elasticsearch you can leverage the full power of the query engine and filter, group, or aggregate data before downloading it. This not only reduces network and disk usage on the client, it also eliminates the processing time required to transform the extracted data on the client side, enabling it to proceed immediately to analysis of the data.

The largest point of interest that deserves closer inspection are the results from Exp C.2 and C.4. As mentioned in section 4.3, we suspect that we did not succeed in clearing the cache from Elasticsearch between each run for both experiments. It would make logical sense, as in both experiments it was the initial run that was an outlier and performed more in line with the rest of the sub-experiments, while the following nine runs reached results far outperforming the first run which makes us suspect that they are using the cached results from the first run instead of actually performing the query on the data. Again, we refer readers to appendix D for the exact results. We did wait for two minutes to give Elasticsearch ample time to clear the cache, and evidently, that was enough for the rest of the experiments where this phenomenon is not present, but not for C.2 and C.4.

An observation regarding both Exp C.2 and C.4 is that they are also the most filtered queries and should result in the least documents needed to be returned. This could be the reason for the uncleared cache but if the initial result is any indication to go by, the results are way more competitive, and even better for Exp C.2, to those of ClickHouse, a clear differentiation from the rest of the experiments. Different results depending on the type of query was of course to be expected and one of the reasons for diversifying the queries in the sub-experiments to begin with. Although, we are not certain, we do believe the more filtered types of queries better reflects the strengths of Elasticsearch, i.e., Lucene, through its blazingly fast lookups for specific documents in large amount of data. For aggregated queries, however, it is not as fast as ClickHouse, which can utilize vectorized queries and SIMD instructions [27].

Lastly, in our extraction experiments, we were limited to two months of data because of hardware and time restrictions, mostly because the time it took to ingest data into Elasticsearch was a slow process when compared to ClickHouse. As a result, we were restricted from benchmarking data extraction queries with a time span longer than two months. We therefore do not know how any of the subjects would scale when aggregating data spanning several years which is a desirable use-case at DDM. This is especially interesting as ClickHouse crashed due to reaching the memory usage limit during sub-experiment C.7. We do, however, know that by optimizing either the data types in the table as mentioned in section 5.1.2 or the query we could have gotten a valid result, but again, this was not tested due to time restrictions.

5.2 Usability and workflow

To answer **RQ3**, we conducted open interviews with three data scientists at Axis, asking them how their workflow would be affected by faster queries and a larger span of metrics. We use these answers in conjunction with our own experiences of conducting the experiments to get a grasp of the implications on usability and workflow by migrating to ClickHouse.

The main benefit of a faster querying capacity, as pointed out by all of our interviewees, was the ability to work in a more agile manner. With a constant feedback loop, there would be fewer context switches and a more efficient workflow. It would enable data scientists to test their way forward with greater ease and help debugging. In our experiments we noted that ClickHouse was not only quite fast, it was also very stable, with low variation in execution time. Elasticsearch on the other hand managed well for queries on small subsets of metrics (Exp C.1–C.4), but with comparably high variation except for the results we believe were cached. The most difficult experiment for Elasticsearch was Exp C.9, where we tested raw data extraction, to see how it and ClickHouse performed compared to MinIO, which was initially implemented at DDM to enable large extraction of time partitioned data. As we have mentioned previously, such kind of extraction might be necessary when the tools provided by Elasticsearch or ClickHouse are insufficient to provide the analysis needed, such as machine learning scripts or analysis requiring complex joins of data sets. ClickHouse too was outperformed clearly by MinIO, but on the other hand produced comparable output files, which was not the case for Elasticsearch, where we just dumped the server-sent JSON to disk in batches, making the processing down the line more complicated as it has to piece together the files. In contrast to MinIO, both support filtering the data before downloading it, which should make them much faster when extracting dimensions other than time.

When asked about a larger span of metrics, our respondents replied that it is useful, especially for hardware designers that want to assess hardware sustainability over longer periods of time, often several years. In such cases, a larger span of metrics would yield more reliable results. It could also enable the teams to observe slow moving or seasonal trends, and perform predictive maintenance. In our experiments, ClickHouse performed much better than Elasticsearch at ingesting and storing large amounts of data, and it scaled considerably better in all but one extraction test. The exception being Exp C.3, where it failed when scaling up the amount of metrics (Exp C.7). We suspect this might be caused by our lack of optimizations, as previously discussed. However, given the rest of our results, and ClickHouse's efficient disk usage, we believe it may be a suitable candidate for even larger sets of metrics.

Another factor to take into consideration under usability is the feature set provided by the experimental subjects. A particular problem with MinIO is the opacity of Parquet files, meaning that it is very difficult to verify the content of the data without downloading the entire file or data set. This leads to larger limitations, for instance, it is very difficult to update or delete data because the data set requires lots of time and work to go through. The long lead times make it very inconvenient to fix schema changes, duplicated data, or resolving GDPR-related privacy issues. Updates and deletions are supported directly in Elasticsearch, though we have noted from the team at DDM that updates in particular are very slow, and do not work well with data streams and automatic rollover, as old indices are automatically set to read-only for performance reasons. ClickHouse supports asynchronous updates and deletes, though they are not compliant with standard SQL.

The ability to combine data from different data sets into one is very powerful in analytics. In ClickHouse this is supported using the SQL-compliant join operation, albeit without query optimization [23]. Unfortunately, we did not have the time to perform any such experiments, but we do know that such operations are not supported in Elasticsearch, which is why for some data sets, DDM has been forced to denormalize the data before ingesting it as a workaround to get the desired analytics. With MinIO, it would be necessary to download the

necessary Parquet files from all data sets and then build a new, denormalized data set locally before proceeding with processing, increasing the already substantial extraction time.

Lastly, we will discuss the ease of use for each of the respective subjects. As we have explained earlier in this thesis, the interfaces to interact with the individual subjects are quite different. MinIO supports fetching files from its object storage over HTTP. This operation could be considered easy, or even beginner friendly, given only fundamental knowledge in command line tools or scripting. MinIO even offers a web-based object browser, enabling users to navigate and download objects/files using their web browser. It is significantly more difficult, however, to perform filtering and analysis on the downloaded files. At DDM, data scientists use Python and the popular data analysis library Pandas, offering a powerful suite of tools to gather insights from data. This requires knowledge on how to parse Parquet files, transforming them to Pandas data frames and then using the Pandas API to get the desired output. Given that the end user can write and understand basic Python code, there is still a significant learning curve that needs to be considered.

Elasticsearch offers a more integrated solution, using a query engine to filter data before serving it to the user. To do so, the user can use a JSON based query language (see appendix C to get a sense of how one would write such a query) or by using an extension enabling "SQL-like" queries [49]. JSON, like most serialization formats, is easy to read and parse for computers, but less ergonomic and intuitive for humans to read and write. Our experience writing these queries was that they are very verbose, and it is not always obvious how a query should be written to be efficient. For instance, in most of our queries to Elasticsearch, we specify the `size` attribute on the root level of the query object to be zero. This is because Elasticsearch may return both the aggregation you have asked for, and the most matching search results for your query, the latter of which is rather expensive. If the size attribute is omitted, the default of ten is chosen, which makes the query execution significantly slower. That said, Elasticsearch is oftentimes paired with Kibana, a visualization software developed for Elasticsearch by Elastic. This web-based tool offers a no-code toolkit to create visualizations and dashboards, abstracting away the Elasticsearch queries from users. It should, in our opinion, be considered a necessity alongside an Elasticsearch deployment for effective analysis.

ClickHouse on the other hand, is powered by a dialect of SQL, a standardized language that has been used to query databases for decades, and is prominently used both in academia and the industry. The syntax of SQL is short and declarative, making it easy to read for humans and potentially minimizing bugs. It is still powerful, and most databases make use of a query engine that tries to optimize the query before traversing the stored data. The query language is composable, and enables queries within queries (sub-queries) as well as joins to combine multiple tables into one result. It is suitable for querying structured data, and its widespread use makes it more likely that data scientists have had experience with it before, making it possible for them to get started and feeling comfortable with the system faster. We would claim that ClickHouse with SQL is the most accessible of the experimental subjects covered in this thesis, apart from visualization software like Kibana. An alternative to Elasticsearch and Kibana is Apache Superset, a "modern data exploration and visualization platform" [5], that should offer a Kibana-like experience powered by SQL databases. We have not examined it in this thesis, but it is something we would recommend exploring further in comparison with Elasticsearch-backed Kibana.

5.3 Threats to validity

There are a number of threats to the validity of this thesis that need to be mentioned. These threats can be categorized as threats to internal validity and external validity [92].

5.3.1 Internal validity

Internal validity refers to the confidence that the independent variable has not been affected by confounding factors [92]. For example, we had issues with Elasticsearch in certain extraction tests where we could not be sure whether the results returned were cached or not, and if we could have configured our setup differently to avoid the situation.

In terms of configuration, it is important to once again point out that we set up our subjects with very little configuration, and relying mostly on the default settings. With Elasticsearch, we replicated the few relevant configurations from DDM's production setup, such as disable swapping and increase the refresh interval to improve indexing speed. Furthermore, the ETL-processing scripts, developed at DDM and responsible for ingesting Parquet files into Elasticsearch, have been optimized over time to maximize performance for ingest by tuning the size of the bulk requests and the number of threads sending data to Elasticsearch.

Because of the requirement to use JSON, we had to write our own script to perform Exp C.9 as well. Like the ETL scripts, this was written in Python using the official Elasticsearch library by Elastic. To verify that no scripts were responsible for slowing down our tests, we performed spot-checks using `htop` on both systems. Doing so, we observed only short bursts of CPU usage on the client while preparing data for ingest and when saving data to disk during extraction. When observing the dedicated server, we noted the opposite effect. Thus, we feel confident that there was very little performance impact imposed by the scripts.

Finally, the quality of the network link between our two systems should be considered. The experiments were performed with on-premise machines at Axis, albeit in different server halls. The systems reached each other over a shared connection, over which we had no control, and so the amount of traffic may have varied greatly over the course of the day.

5.3.2 External validity

External validity refers to the degree of confidence that the results of our experiments can be generalized to industrial practice [92]. In that case, the experiments we have performed, while using real used-in-production metrics, they are also proprietary making the tests difficult to reproduce or generalize beyond the table structure that we have disclosed. In addition, the experiment suite is synthetic, and all tests are run in isolation. This is preferable for accurate results, but it is not representative for how the different subjects perform during everyday workloads, where data may be consumed and ingested simultaneously.

The selection of data scientists at DDM that we interviewed are both consumers and maintainers of the current implementation, and have certain experience with the current system and its methods for querying and analyzing data, which is different from ClickHouse SQL. Future studies are needed to ensure generalizability to data scientists without this bias.

5.4 Future work

All the experimental subjects studied in this thesis have been configured as as single-node deployment. However, they all support clustering of multiple servers in some way or form. Implementing all three subjects in a clustered setup would be interesting to investigate how this would affect our results and if any of the subjects would benefit more by this than the others. Therefore, testing our subjects in a clustered setup is recommended as future work.

An important limitation in this thesis is that our approach to configuring ClickHouse is rather naive and there is still a lot of optimization that can be implemented and tested for an improved result. An experimental test suite with fully optimized subjects would show their upper limits and could be compared to the results presented here. We would recommend future work with ClickHouse, mainly taking into consideration the effects of optimizing the table structure as already outlined in section 5.1.2.

Our research has been centered around the current setup at Axis, and consequently, compared ClickHouse first and foremost to Elasticsearch. This is however not a perfect apples-to-apples comparison in an OLAP workflow and we would encourage further research to explore other, either column-oriented DBMSs such as Druid and Cassandra, or a more traditional approach used in big data workflows such as Apache Spark, using experiments similar to ours.

An initial goal was to explore long-reaching trend analysis using ClickHouse, to see if it could replace not only Elasticsearch, but also the use of MinIO at DDM. Unfortunately, we found ourselves constrained for both time and storage space on the dedicated server to extend the time span of our experiments much longer. Such tests could also make use of more advanced analysis features in ClickHouse, beyond the simple aggregations presented in this thesis.

In addition, we initially wanted to test the real-time analysis use case for the experimental subjects discussed, but found no reliable or reproducible way to carry out experiments in this manner. By real-time, we mean the case where the database is actively serving other users, like ingesting data at the same time that it is being queried. To accurately benchmark this scenario, we believe it would be necessary to create an experimental test suite that would take into account the difference in ingest performance to verify the results of the extracted data. It is also fair to point out that in a real scenario, ClickHouse, and probably even more so Elasticsearch, would benefit greatly if they were permitted to cache common requests.

As mentioned in section 5.2, both Elasticsearch and ClickHouse (because it uses SQL), can be integrated with visualization software such as Kibana and Apache Superset respectively to assist users in gathering insights, using beginner-friendly no-code toolkits. We did not explore these tools in this thesis, but feel that a comparison in accessibility would be suitable to test how viable ClickHouse would be as an Elasticsearch replacement in environments where the intended audience may have less technical proficiency.

Chapter 6

Conclusion

Our thesis work aimed to investigate whether ClickHouse was a suitable candidate for big data processing. Specifically for DDM, we were interested to find if it could match, or even replace their current processing pipeline that is mainly dependent on Elasticsearch for insights on recent data and MinIO for long-term storage. With several terabytes of metrics being produced per month, it is fair to classify the workload as *big data*, which requires carefully chosen tools to manage and process.

The existing implementation has managed the incoming data with decent results since it was introduced, but as we saw when we delved deeper into the current setup, aiming to answer **RQ1**, there are lots of compromises involved. Compromises that would be highly beneficial to avoid, mainly having a 90-day limit on metrics being processed by Elasticsearch, but also the significant added complexity, not only by maintaining two different data stores, but also having to extract and transform compressed metrics in order to process them on the client.

The findings of **RQ1** showed us that the challenge of big data processing is composed of three important problems. We identified these problems as ingesting data, extracting data, and storing the data efficiently. Having identified these problems, we set out to create a suite of experimental benchmarks that would saturate the capability of each experimental subject for the given problem, thus helping us answer **RQ2**.

Using our benchmark suite, we saw that ClickHouse was, in spite of our naive configuration with very little optimizations, quite adept at handling the challenges we set it up for, with remarkable performance shown specifically in our benchmarks of ingest speed and storage use. Extraction performance was impressive too, outperforming Elasticsearch in most query types. For the rest, our unoptimized configuration was likely the limiting factor, yielding an experiment that ClickHouse could not successfully finish, and two in which its performance was comparable, though slightly worse, to Elasticsearch.

Our last goal was to investigate the usability impact a migration to ClickHouse would imply for the users, and we answered **RQ3** by presenting our findings from **RQ2** to data scientists at Axis, together with our subjective experiences during the course of the thesis work. We found that having faster queries and a larger archive of metrics to query, could yield significant improvements in efficiency by enabling a more agile workflow, with fewer context switches, easier schema migrations and faster debugging. In addition, we found that Elasticsearch is neither intuitive nor concise to query using the provided query language, and using MinIO for analysis requires knowledge of programming as well as a data analysis toolkit. Meanwhile, ClickHouse, using a standardized query language in SQL, will naturally be more familiar and approachable to experienced users and beginners respectively.

To summarize, we found that ClickHouse is a suitable candidate for big data processing. As indicated by our results, it is well suited for handling incoming data at large scale, and stores it efficiently too, which allows it to scale very well. It has fast extraction rate for different queries, and by being sufficiently fast at extracting raw data, offers a convenient escape hatch when client-side computing is needed.

We feel confident in recommending the use of ClickHouse for big data processing at DDM.

References

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. "Integrating compression and execution in column-oriented database systems". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (2006).

[2] Alberto Abelló and Oscar Romero. "On-Line Analytical Processing". In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. New York, NY: Springer New York, 2016, pp. 1–7. ISBN: 978-1-4899-7993-3.

[3] The Apache Software Foundation. *Apache Lucene*. URL: `https://lucene.apache.org/` (visited on Jan. 11, 2022).

[4] The Apache Software Foundation. *Apache Parquet*. URL: `https://parquet.apache.org/` (visited on Oct. 8, 2021).

[5] The Apache Software Foundation. *Superset*. URL: `https://superset.apache.org/` (visited on Jan. 14, 2022).

[6] Jeff Barr. *Amazon S3 – The First Trillion Objects*. 2012. URL: `https://aws.amazon.com/blogs/aws/amazon-s3-the-first-trillion-objects/` (visited on Dec. 14, 2021).

[7] Jeff Barr. *Celebrate 15 Years of Amazon S3 with 'Pi Week' Livestream Events*. 2021. URL: `https://aws.amazon.com/blogs/aws/amazon-s3s-15th-birthday-it-is-still-day-1-after-5475-days-100-trillion-objects/` (visited on Dec. 14, 2021).

[8] Kristi Berg, Dr. Tom Seymour, and Richa Goel. "History Of Databases". In: *International Journal of Management & Information Systems (IJMIS)* 17 (Dec. 2012), p. 29. DOI: `10.19030/ijmis.v17i1.7587`.

[9] Eric Brewer. "CAP twelve years later: How the "rules" have changed". In: *Computer* 45.2 (2012), pp. 23–29.

[10] Rick Cattell. "Scalable SQL and NoSQL Data Stores". In: *SIGMOD Rec.* 39.4 (May 2011), pp. 12–27. ISSN: 0163-5808.

[11] Surajit Chaudhuri and Umeshwar Dayal. “An Overview of Data Warehousing and OLAP Technology”. In: *SIGMOD Rec.* 26.1 (Mar. 1997), pp. 65–74. ISSN: 0163-5808.

[12] ClickHouse Inc. *ClickHouse History*. URL: `https://clickhouse.com/docs/en/introduction/history/` (visited on Dec. 22, 2021).

[13] ClickHouse Inc. *ClickHouse Inc.* URL: `https://clickhouse.com/docs/en/` (visited on Mar. 8, 2022).

[14] ClickHouse Inc. *ClickHouse Inc.* URL: `https://clickhouse.com/` (visited on Dec. 22, 2021).

[15] ClickHouse Inc. *ClickHouse Keeper*. URL: `https://clickhouse.com/docs/en/operations/clickhouse-keeper/` (visited on Dec. 22, 2021).

[16] ClickHouse Inc. *ClickHouse raises a $250M Series B at a $2B valuation...and we are hiring*. URL: `https://clickhouse.com/blog/en/2021/clickhouse-raises-250m-series-b/` (visited on Jan. 11, 2022).

[17] ClickHouse Inc. *ClickHouse release v21.8, 2021-08-12*. 2021. URL: `https://clickhouse.com/docs/en/whats-new/changelog/#clickhouse-release-v21-8-2021-08-12` (visited on Dec. 23, 2021).

[18] ClickHouse Inc. *ClickHouse Server Docker Image*. URL: `https://hub.docker.com/r/clickhouse/clickhouse-server/` (visited on Dec. 8, 2021).

[19] ClickHouse Inc. *Custom Partitioning Key*. URL: `https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/custom-partitioning-key/` (visited on Jan. 11, 2022).

[20] ClickHouse Inc. *Distinctive Features of ClickHouse*. URL: `https://clickhouse.com/docs/en/introduction/distinctive-features/` (visited on Dec. 22, 2021).

[21] ClickHouse Inc. *Evolution of Data Structures in Yandex.Metrica*. 2016. URL: `https://clickhouse.com/blog/en/2016/evolution-of-data-structures-in-yandex-metrica/` (visited on Oct. 8, 2021).

[22] ClickHouse Inc. *Interfaces*. URL: `https://clickhouse.com/docs/en/interfaces/` (visited on Dec. 22, 2021).

[23] ClickHouse Inc. *JOIN*. URL: `https://clickhouse.com/docs/en/sql-reference/statements/select/join/#performance` (visited on Jan. 14, 2022).

[24] ClickHouse Inc. *MergeTree Engine Family*. URL: `https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/` (visited on Jan. 11, 2022).

[25] ClickHouse Inc. *TABLE*. URL: `https://clickhouse.com/docs/en/sql-reference/statements/create/table/#create-query-general-purpose-codecs` (visited on Jan. 12, 2022).

[26] ClickHouse Inc. *What's New in ClickHouse 21.12*. 2021. URL: `https://clickhouse.com/blog/en/2021/clickhouse-v21.12-released/#from-infile-in-clickhouse-client-now-supports-glob-patterns-and-parallel-reading` (visited on Jan. 11, 2022).

[27] ClickHouse Inc. *Why ClickHouse Is So Fast?* URL: `https://clickhouse.com/docs/en/faq/general/why-clickhouse-is-so-fast/` (visited on Jan. 16, 2022).

[28] ClickHouse Inc. *Yandex Opensources ClickHouse*. 2016. URL: `https://clickhouse.tech/blog/en/2016/yandex-opensources-clickhouse/` (visited on Sept. 20, 2021).

[29] Edgar F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782.

[30] Edgar. F. Codd, S. B. Codd, and Columbus Salley. "Providing OLAP to User-Analysts: An IT Mandate". In: 1993.

[31] Htop contributors. *htop—an interactive process viewer*. URL: `https://htop.dev/` (visited on Feb. 27, 2022).

[32] Pandas contributors. *Pandas - Python Data Analysis Library*. URL: `https://pandas.pydata.org/` (visited on Feb. 27, 2022).

[33] José Correia, Carlos Costa, and Maribel Yasmina Santos. "Challenging SQL-on-Hadoop Performance with Apache Druid". In: *Business Information Systems*. Ed. by Witold Abramowicz and Rafael Corchuelo. Cham: Springer International Publishing, 2019, pp. 149–161. ISBN: 978-3-030-20485-3.

[34] José Correia, Maribel Yasmina Santos, Carlos Costa, and Carina Andrade. "Fast Online Analytical Processing for Big Data Warehousing". In: *2018 International Conference on Intelligent Systems (IS)*. 2018, pp. 435–442. DOI: `10.1109/IS.2018.8710583`.

[35] Nick Craig-Wood and Rclone contributors. *Documentation - Rclone*. URL: `https://rclone.org/docs/` (visited on Nov. 17, 2021).

[36] DB-Engines. *DB-Engines Ranking - Trend Popularity*. 2021. URL: `https://db-engines.com/en/ranking_trend` (visited on Dec. 26, 2021).

[37] Peter J. Denning. "The Locality Principle". In: *Commun. ACM* 48.7 (July 2005), pp. 19–24. ISSN: 0001-0782.

[38] Docker. *What is a container?* URL: `https://www.docker.com/resources/what-container` (visited on Dec. 13, 2021).

[39] Elastic. *Data in: documents and indices*. URL: `https://www.elastic.co/guide/en/elasticsearch/reference/7.15/documents-indices.html` (visited on Dec. 13, 2021).

[40] Elastic. *Disable swapping*. URL: `https://www.elastic.co/guide/en/elasticsearch/reference/7.15/setup-configuration-memory.html` (visited on Dec. 6, 2021).

[41] Elastic. *Elasticsearch*. URL: `https://www.elastic.co/elasticsearch/` (visited on Dec. 13, 2021).

[42] Elastic. *How many shards should I have in my Elasticsearch cluster?* URL: `https://www.elastic.co/blog/how-many-shards-should-i-have-in-my-elasticsearch-cluster` (visited on Dec. 30, 2021).

[43] Elastic. *Install Elasticsearch with Docker*. URL: `https://www.elastic.co/guide/en/elasticsearch/reference/7.15/docker.html` (visited on Dec. 8, 2021).

[44] Elastic. *Install Elasticsearch with Docker - Using the Docker images in production*. URL: `https://www.elastic.co/guide/en/elasticsearch/reference/7.15/docker.html#docker-prod-prerequisites` (visited on Dec. 8, 2021).

[45] Elastic. *Kibana: Explore, Visualize, Discover Data*. URL: https://www.elastic.co/kibana/ (visited on Feb. 27, 2022).

[46] Elastic. *Query DSL*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/7.15/query-dsl.html (visited on Dec. 14, 2021).

[47] Elastic. *Rollover*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/7.15/index-rollover.html (visited on Nov. 24, 2021).

[48] Elastic. *Scalability and resilience: clusters, nodes, and shards*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/7.15/scalability.html (visited on Dec. 13, 2021).

[49] Elastic. *SQL Overview*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/7.15/sql-overview.html (visited on Jan. 14, 2022).

[50] Elastic. *Tune for disk usage*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/7.15/tune-for-disk-usage.html (visited on Jan. 5, 2021).

[51] Elastic. *Tune for indexing speed*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/7.15/tune-for-indexing-speed.html (visited on Jan. 5, 2021).

[52] Ramez Elmasri and Sham Navathe. *Fundamentals of database systems*. Pearson/Addison-Wesley, 2004. ISBN: 0321204484.

[53] The R Foundation. *R: The R Project for Statistical Computing*. URL: https://www.r-project.org/ (visited on Feb. 27, 2022).

[54] Yupeng Fu and Chinmay Soman. "Real-Time Data Infrastructure at Uber". In: *Proceedings of the 2021 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 2503–2516. ISBN: 9781450383431. URL: https://doi.org/10.1145/3448016.3457552.

[55] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700.

[56] GNU. *du: Estimate file space usage*. URL: https://www.gnu.org/software/coreutils/manual/html_node/du-invocation.html#du-invocation (visited on Dec. 13, 2021).

[57] J. Goldstein, R. Ramakrishnan, and U. Shaft. "Compressing relations and indexes". In: *Proceedings 14th International Conference on Data Engineering*. 1998, pp. 370–379.

[58] G. Graefe and L.D. Shapiro. "Data compression and database performance". In: *[Proceedings] 1991 Symposium on Applied Computing*. 1991, pp. 22–27.

[59] Cornelia Hammer, Diane Kostroch, and Gabriel Quiros. *Big Data: Potential, Challenges and Statistical Implications*. Staff Discussion Notes. International Monetary Fund, 2017. ISBN: 9781484310908.

[60] Wilhelm Hasselbring. "Benchmarking as Empirical Standard in Software Engineering Research". In: *Evaluation and Assessment in Software Engineering*. EASE 2021. Trondheim, Norway: Association for Computing Machinery, 2021, pp. 365–372. ISBN: 9781450390538.

[61] Martin Höst, Björn Regnell, and Per Runesson. *Att genomföra examensarbete*. Studentlitteratur AB, 2006. ISBN: 978-91-44-00521-8.

[62] IBM Cloud Education. *CAP Theorem*. 2019. URL: `https://www.ibm.com/cloud/learn/cap-theorem` (visited on Dec. 12, 2021).

[63] IBM Cloud Education. *Object Storage*. 2019. URL: `https://www.ibm.com/cloud/learn/object-storage` (visited on Dec. 23, 2021).

[64] IBM Cloud Education. *Relational Databases*. 2019. URL: `https://www.ibm.com/cloud/learn/relational-databases` (visited on Sept. 24, 2021).

[65] Baktagul Imasheva, Nakispekov Azamat, Andrey Sidelkovskiy, and Ainur Sidelkovskaya. "The Practice of Moving to Big Data on the Case of the NoSQL Database, Clickhouse". In: *Optimization of Complex Systems: Theory, Models, Algorithms and Applications*. Ed. by Hoai An Le Thi, Hoai Minh Le, and Tao Pham Dinh. Cham: Springer International Publishing, 2020, pp. 820–828. ISBN: 978-3-030-21803-4.

[66] iso25000.com. *Usability*. URL: `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/61-usability` (visited on Jan. 14, 2022).

[67] Theodore Johnson. "Performance Measurements of Compressed Bitmap Indices". In: *VLDB*. 1999.

[68] Neal Leavitt. "Will NoSQL Databases Live Up to Their Promise?" In: *Computer* 43.2 (2010), pp. 12–14.

[69] LZ4 contributors. *LZ4*. URL: `https://lz4.github.io/lz4/` (visited on Sept. 21, 2021).

[70] Samuel Madden. "From Databases to Big Data". In: *IEEE Internet Computing* 16.03 (2012), pp. 4–6. ISSN: 1941-0131.

[71] Mathworks. *MATLAB*. URL: `https://mathworks.com/products/matlab.html` (visited on Feb. 27, 2022).

[72] Monerah Al-Mekhlal and Amir Ali Khwaja. "A Synthesis of Big Data Definition and Characteristics". In: *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. 2019, pp. 314–322. DOI: `10.1109/CSE/EUC.2019.00067`.

[73] M. Mesnier, G.R. Ganger, and E. Riedel. "Object-based storage". In: *IEEE Communications Magazine* 41.8 (2003), pp. 84–90.

[74] Alexey Milovidov. *Introducing ClickHouse, Inc.* 2021. URL: `https://clickhouse.com/blog/en/2021/clickhouse-inc/` (visited on Sept. 20, 2021).

[75] MinIO Inc. *Distributed MinIO Quickstart Guide*. URL: `https://docs.min.io/docs/distributed-minio-quickstart-guide.html` (visited on Jan. 11, 2022).

[76] MinIO Inc. *minio/minio:RELEASE.2021-11-03T03-36-36Z*. URL: `https://hub.docker.com/layers/minio/minio/RELEASE.2021-11-03T03-36-36Z/images/sha256-65d1540c0ee7f34036a0013ed8e6740c6260da98822657492bed8eb4b27a491b?context=explore` (visited on Dec. 13, 2021).

[77] MinIO Inc. *The MinIO quickstart guide*. URL: `https://docs.min.io/` (visited on Dec. 2, 2021).

[78] MongoDB. *What are ACID transactions?* URL: https://www.mongodb.com/basics/acid-transactions (visited on Dec. 26, 2021).

[79] A B M Moniruzzaman and Syed Hossain. "NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison". In: *Int J Database Theor Appl* 6 (June 2013).

[80] Oracle. *What is a relational database?* URL: https://www.oracle.com/database/what-is-a-relational-database/ (visited on Sept. 24, 2021).

[81] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer International Publishing, 2020. ISBN: 9783030262525.

[82] Gautam Ray, Jayant R. Haritsa, and Sridhar Seshadri. "Database Compression: A Performance Enhancement Tool". In: *COMAD*. 1995.

[83] David Salomon and Giovanni Motta. *Handbook of data compression.* Springer, 2010. ISBN: 9781282835993.

[84] Pavel Seda, Jiri Hosek, Pavel Masek, and Jiri Pokorny. "Performance testing of NoSQL and RDBMS for storing big data in e-applications". In: *2018 3rd International Conference on Intelligent Green Building and Smart Grid (IGBSG)*. 2018, pp. 1–4. DOI: 10.1109/IGBSG.2018.8393559.

[85] Tanmay Sinha. *OLAP vs. OLTP: What's the Difference?* 2021. URL: https://www.ibm.com/cloud/blog/olap-vs-oltp (visited on Oct. 18, 2021).

[86] Snappy contributors. *README*. URL: https://github.com/google/snappy/blob/master/docs/README.md (visited on Sept. 21, 2021).

[87] Stack Overflow. *Databases - Stack Overflow Developer Survey 2021*. 2021. URL: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-databases (visited on Dec. 26, 2021).

[88] Michael Stonebraker. "SQL Databases v. NoSQL Databases". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 10–11. ISSN: 0001-0782.

[89] Alexey Struckov, Semen Yufa, Alexander A. Visheratin, and Denis Nasonov. "Evaluation of modern tools and techniques for storing time-series data". In: *Procedia Computer Science* 156 (2019). 8th International Young Scientists Conference on Computational Science, YSC2019, 24-28 June 2019, Heraklion, Greece, pp. 19–28. ISSN: 1877-0509.

[90] Matei-Eugen Vasile, Giuseppe Avolio, and Igor Soloviev. "Evaluating InfluxDB and ClickHouse database technologies for improvements of the ATLAS operational monitoring data archiving". In: *Journal of Physics: Conference Series* 1525 (Apr. 2020), p. 012027.

[91] Akila Wickramasekara, M.P.P. Liyanage, and Udayagee Kumarasinghe. "A comparative study between the capabilities of MySQL and ClickHouse in low-performance Linux environment". In: *2020 20th International Conference on Advances in ICT for Emerging Regions (ICTer)*. 2020, pp. 276–277. DOI: 10.1109/ICTer51097.2020.9325483.

[92] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012. ISBN: 9783642290435.

[93] Vlad-Andrei Zamfir, Mihai Carabas, Costin Carabas, and Nicolae Tapus. "Systems Monitoring and Big Data Analysis Using the Elasticsearch System". In: *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*. 2019, pp. 188–193. DOI: 10.1109/CSCS.2019.00039.

[94] Paul Zikopoulos and Chris Eaton. *Understanding big data: Analytics for enterprise class Hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

Appendices

Appendix A

Dedicated server setup

Table A.1: Hardware configuration for **SERVER**.

Purpose	**Description**	**Quantity**
CPU	Intel Xeon-G 5220R FIO Kit for DL380 G10	1
RAM Module	HPE 16 GB 1Rx4 PC4-2933Y-R Smart Kit	6
OS Disk Drive	HPE 240 GB SATA RI SFF SC MV SSD	1
Data Disk Drive	HPE 1.92 TB SATA RI SFF SC MV SSD	7
RAID Card	HPE Smart Array P408i-a SR Gen10 Ctrlr	1
10 Gb Network Adapter	HPE Ethernet 10Gb 2P 530T Adptr	1

Appendix B

Initial setup

MinIO

```
$ docker run \
 -p 9000:9000 \ # Expose the server port 9000.
 -v /srv/minio:/data \ # Mount /srv/minio to /data in the container.
 minio/minio:RELEASE.2021-11-03T03-36-36Z \
 server /data # Run server and point to the mounted data path.
```

Elasticsearch

```
$ docker run -d \
  --name elasticsearch \
  -p 9200:9200 \
  -e discovery.type=single-node \ # Form a single-node cluster.
  -e ELASTIC_USER -e ELASTIC_PASSWORD \ # Pass variables to container.
  --ulimit nofile=65536:65536 \
  -v /srv/elasticsearch:/usr/share/elasticsearch/data \
  docker.elastic.co/elasticsearch/elasticsearch:7.15.2
```

Clickhouse

```
$ docker run -d \
    --name clickhouse \
```

```
    -p 9000:9000 \
    --ulimit nofile=262144:262144 \
    -v /srv/clickhouse:/var/lib/clickhouse \
    clickhouse/clickhouse-server:21.8.11.4
```

```
CREATE TABLE thesis.memory
(
    `Executable` Nullable(String),
    `dirty` Nullable(Int64),
    `lib` Nullable(Int64),
    `name` Nullable(String),
    `pid` Nullable(Int64),
    `Pid` Nullable(Int64),
    `Private_Clean` Nullable(Int64),
    `Private_Dirty` Nullable(Int64),
    `progres` Nullable(Int64),
    `Pss` Nullable(Int64),
    `Pss_Dirty` Nullable(Int64),
    `Rss` Nullable(Int64),
    `shared` Nullable(Int64),
    `Shared_Clean` Nullable(Int64),
    `Shared_Dirty` Nullable(Int64),
    `starttime` Nullable(Int64),
    `swap` Nullable(Int64),
    `Swap` Nullable(Int64),
    `SwapPss` Nullable(Int64),
    `text` Nullable(Int64),
    `totres` Nullable(Int64),
    `proc_uptime` Nullable(Int64),
    `sum_memory` Nullable(Int64),
    `boot_id` String,
    `Env` Nullable(String),
    `idd_Acapversion` Nullable(String),
    `origdocumentpath` Nullable(String),
    `Product` Nullable(String),
    `Firmware_Version` Nullable(String),
    `SerialNumber` String,
    `timestamp` DateTime,
    `unused_field_1` Nullable(String),
    `platform_id` Nullable(String)
    `raw_document_source` Nullable(String),
    `uptime` Nullable(Int64),
    `unused_field_2` Nullable(Int64),
)
ENGINE = MergeTree
PARTITION BY toYYYYMM(toDate(timestamp))
ORDER BY timestamp
```

Appendix C

Data extraction queries and scripts

Exp C.1.

MinIO

Not tested.

Elasticsearch

```
{
  "aggs": {
    "2": {
      "terms": {
        "field": "Firmware_Version.keyword",
        "order": {
          "1": "desc"
        },
        "size": 5
      },
      "aggs": {
        "1": {
          "avg": {
            "field": "memory.Private_Dirty"
          }
        },
        "3": {
```

```
          "terms": {
            "field": "memory.Executable.keyword",
            "order": {
              "1": "desc"
            },
            "size": 5
          },
          "aggs": {
            "1": {
              "avg": {
                "field": "memory.Private_Dirty"
              }
            }
          }
        }
      }
    }
  },
  "size": 0,
  "query": {
    "bool": {
      "filter": [
        {
          "range": {
            "@timestamp": {
              "gte": "2021-08-01T00:00:00.000Z",
              "lte": "2021-11-01T00:00:00.000Z",
              "format": "strict_date_optional_time"
            }
          }
        }
      ]
    }
  }
}
```

ClickHouse

```
SELECT
    Firmware_Version,
    Executable,
    avg(Private_Dirty)
FROM thesis.test
WHERE Firmware_Version IN (
    SELECT Firmware_Version
    FROM thesis.test
```

```
    GROUP BY Firmware_Version
    ORDER BY avg(Private_Dirty) DESC
    LIMIT 5
)
GROUP BY
    Executable,
    Firmware_Version
ORDER BY avg(Private_Dirty) DESC
LIMIT 5 BY Firmware_Version
```

Exp C.2.

MinIO

Not tested.

Elasticsearch

```
{
  "aggs": {
    "2": {
      "date_histogram": {
        "field": "@timestamp",
        "calendar_interval": "1d",
        "time_zone": "Europe/Stockholm",
        "min_doc_count": 1
      },
      "aggs": {
        "1": {
          "avg": {
            "field": "memory.Pss_Dirty"
          }
        }
      }
    }
  },
  "size": 0,
  "query": {
    "bool": {
      "filter": [
        {
          "match_phrase": {
            "target.SerialNumber": "A1A1A1A1A1A1"
          }
        },
```

```
        {
          "range": {
            "@timestamp": {
              "gte": "2021-07-01T00:00:00.000Z",
              "lte": "2021-09-01T02:00:00.000Z",
              "format": "strict_date_optional_time"
            }
          }
        }
      ]
    }
  }
}
```

ClickHouse

```
SELECT
    toDate(timestamp, 'Europe/Stockholm') AS date,
    avg(Pss_Dirty)
FROM thesis.test
WHERE SerialNumber = 'A1A1A1A1A1A1'
GROUP BY date
ORDER BY date
```

Exp C.3.

MinIO

Not tested.

Elasticsearch

```
{
  "aggs": {
    "2": {
      "date_histogram": {
        "field": "@timestamp",
        "calendar_interval": "1h",
        "time_zone": "Europe/Stockholm",
        "min_doc_count": 1
      },
      "aggs": {
        "3": {
          "terms": {
            "field": "target.Product.keyword",
```

```
            "order": {
              "1": "desc"
            },
            "size": 15
          },
          "aggs": {
            "1": {
              "cardinality": {
                "field": "target.SerialNumber.keyword"
              }
            }
          }
        }
      }
    }
  },
  "size": 0,
  "query": {
    "bool": {
      "filter": [
        {
          "range": {
            "@timestamp": {
              "gte": "2021-07-01T00:00:00.000Z",
              "lte": "2021-08-01T02:00:00.000Z",
              "format": "strict_date_optional_time"
            }
          }
        }
      ]
    }
  }
}
```

ClickHouse

```
SELECT
    dateTrunc(
      'hour',
      timestamp,
      'Europe/Stockholm'
    ) AS date,
    countDistinct(SerialNumber),
    Product,
    count(*)
FROM thesis.test
```

```
WHERE (date >= '2021-08-01') AND (date <= '2021-08-07')
GROUP BY
    date,
    Product
ORDER BY date ASC
```

Exp C.4.

MinIO

Not tested.

Elasticsearch

```
{
  "aggs": {
    "2": {
      "terms": {
        "field": "target.Product.keyword",
        "order": {
          "1": "desc"
        },
        "size": 10
      },
      "aggs": {
        "1": {
          "avg": {
            "field": "memory.Pss_Dirty"
          }
        }
      }
    }
  },
  "size": 0,
  "query": {
    "bool": {
      "filter": [
        {
          "match_phrase": {
            "memory.Executable.keyword": "program1"
          }
        },
        {
          "range": {
            "@timestamp": {
```

```
                "gte": "2021-07-01T00:00:00.000Z",
                "lte": "2021-11-01T02:00:00.000Z",
                "format": "strict_date_optional_time"
              }
            }
          }
        ]
      }
    }
}
```

ClickHouse

```sql
SELECT
    Product,
    avg(Pss_Dirty)
FROM thesis.test
WHERE Executable = 'program1'
GROUP BY Product
ORDER BY avg(Pss_Dirty) DESC
LIMIT 10
```

Exp C.9.

To test raw data extraction, we decided to fetch ten hours worth of data from our two month range. For simplicity, we chose the first ten hours chronologically.

MinIO

```
$ rclone copy \
      --include 'hour=[0-9]/*.parquet' \
      --include 'hour=10/*.parquet' \
      --transfers 12 --checkers 12 \
      minio:bucket/memory/year=2021/month=8/day=1 \
      /path/to/local/data
```

ClickHouse

```
$ clickhouse-client --query \
    "SELECT *
    FROM thesis.test
    WHERE timestamp < '2021-08-01 10:00:00'
    FORMAT Parquet" \
  > data.parquet
```

Elasticsearch

Since there is no built in support for raw extraction, we wrote a Python script to extract all data using the official Elasticsearch library. The scan method used is an abstraction over the scroll API, and fetches 10,000 results (the max size) per request and stores them in memory. When 5,000,000 results have been fetched, the JSON documents are dumped to file.

```
import json
from elasticsearch import Elasticsearch, helpers

def write_to_file(results, index):
    # json.dumps(list) is faster than json.dump(list, file)
    # if we have the memory available for it.
    serialized = json.dumps(results)
    with open(f"es-dump-{index}.json", "w") as out:
        out.write(serialized)

query = {
    "query": {
        "bool": {
            "must": {
                "match_all": {}
            },
            "filter": [
                {
                    "range": {
                        "@timestamp": {
                            "lt": "2021-08-01T10:00:00.000Z",
                            "format": "strict_date_optional_time"
                        }
                    }
                }
            ]
        }
    }
}

es, index, results = Elasticsearch(ES_ENDPOINT_URL), 0, []
for hit in helpers.scan(es, query, index=ES_INDEX_NAME, size=10000):
    results.append(hit)
    if len(results) >= 5000000:
        write_to_file(results, index)
        results = []
        index += 1

if len(results) > 0:
    write_to_file(results, index)
```

Appendix D

Raw results

Exp A: Ingest

Table D.1: Exp A: Ingest. Transfer time in seconds.

	MINIO	ELASTIC	CLICKH
Run 1	1,389	45,354	1,977
Run 2	1,390	45,707	1,986
Run 3	1,387	45,707	2,002

Exp B: Storage

Table D.2: Exp B: Storage use in GB. Measurements immediately after ingest are noted in parentheses.

	MINIO	ELASTIC	CLICKH
Run 1	152	620 (622)	186 (280)
Run 2	152	620 (624)	186 (357)
Run 3	152	621 (626)	186 (370)

Exp C: Extraction rate

Table D.3: Exp C.1. Execution time in seconds.

	ELASTIC	CLICKH
Run 1	39.468872037	7.636695946
Run 2	39.505596954	7.454333315
Run 3	39.850363293	8.712967669
Run 4	39.424436749	7.441369848
Run 5	39.178884600	7.432631532
Run 6	39.471937193	7.474382540
Run 7	39.766114013	7.528771817
Run 8	40.032892012	7.547876603
Run 9	39.693789813	7.600013086
Run 10	39.857188351	7.471453874

Table D.4: Exp C.2. Execution time in seconds.

	ELASTIC	CLICKH
Run 1	3.697920456	4.581286327
Run 2	0.043449516	4.918519457
Run 3	0.038520861	4.594558699
Run 4	0.047978713	4.488945893
Run 5	0.044014322	4.547258952
Run 6	0.041242109	4.556078396
Run 7	0.047741518	4.527704714
Run 8	0.041057660	4.577401036
Run 9	0.041675885	4.561963036
Run 10	0.041389254	4.674475131

Table D.5: Exp C.3. Execution time in seconds.

	ELASTIC	CLICKH
Run 1	226.086654555	15.750639980
Run 2	233.504078026	16.158404735
Run 3	231.442777261	15.960129853
Run 4	234.547322732	16.347730252
Run 5	233.602792717	15.945108092
Run 6	232.415022596	16.280487336
Run 7	233.577138135	16.323002607
Run 8	233.659388246	16.268199078
Run 9	234.726147362	16.255170123
Run 10	235.292303323	16.003353958

Table D.6: Exp C.4. Execution time in seconds.

	ELASTIC	CLICKH
Run 1	5.755578149	5.985482376
Run 2	0.700347428	5.226242395
Run 3	0.742086227	5.242065391
Run 4	0.724358475	5.198093403
Run 5	0.739347438	5.659624115
Run 6	0.722640143	5.303758468
Run 7	0.730048692	5.257330676
Run 8	0.719913420	5.365259020
Run 9	0.726058374	5.277200862
Run 10	0.716483587	5.129347874

Table D.7: Exp C.5. Execution time in seconds.

	ELASTIC	CLICKH
Run 1	630.004861692	55.528402794
Run 2	637.778929173	55.572750402
Run 3	637.397820415	54.998032722
Run 4	640.105377256	55.050438133
Run 5	639.957177130	55.243229576
Run 6	637.910285159	55.129427660
Run 7	638.830093734	55.128641317
Run 8	639.067251362	55.631400112
Run 9	636.823579986	55.113933356
Run 10	637.546283469	56.069277844

Table D.8: Exp C.6. Execution time in seconds.

	ELASTIC	CLICKH
Run 1	175.590842876	30.310682879
Run 2	170.384822259	30.256843198
Run 3	170.384735087	30.221557260
Run 4	170.659342415	30.239793664
Run 5	190.727321452	30.189139850
Run 6	190.851733102	30.256216953
Run 7	191.054955104	30.252177672
Run 8	191.055224827	30.354546402
Run 9	191.190239079	30.236794986
Run 10	191.186323135	30.293447664

Table D.9: Exp C.7. Execution time in seconds.

	ELASTIC	CLICKH
Run 1	1,895.244063637	Failed (ran out of memory) after 47.373851170
Run 2	1,894.536010459	Failed (ran out of memory) after 45.131942059
Run 3	1,890.798993381	Failed (ran out of memory) after 47.834385013
Run 4	1,888.702138128	Failed (ran out of memory) after 48.599351006
Run 5	1,932.319168396	Failed (ran out of memory) after 47.305670480
Run 6	1,957.877363136	Failed (ran out of memory) after 48.190600892
Run 7	1,954.396968247	Failed (ran out of memory) after 43.843315295
Run 8	1,957.680313553	Failed (ran out of memory) after 48.640711598
Run 9	1,956.925801334	Failed (ran out of memory) after 45.747029421
Run 10	1,957.513382523	Failed (ran out of memory) after 46.900910119

Table D.10: Exp C.8. Execution time in seconds.

	ELASTIC	CLICKH
Run 1	295.167762173	34.951629579
Run 2	265.789655374	34.784797738
Run 3	245.815007772	34.863814788
Run 4	246.440271029	34.818451851
Run 5	246.567408145	34.917577236
Run 6	247.065249239	34.868013454
Run 7	246.340014285	34.862011670
Run 8	247.088031550	34.776979904
Run 9	246.485076440	34.764945120
Run 10	247.633355723	34.741739745

Table D.11: Exp C.9. Transfer time in seconds.

	MINIO	ELASTIC	CLICKH
Run 1	71.015762598	5,528.452644635	367.570235226
Run 2	71.106565528	5,616.393959781	372.213909726
Run 3	71.391599332	5,448.884645142	365.013800481
Run 4	71.114836013	5,513.881490003	372.415507810
Run 5	71.472710631	5,454.653877125	366.843512232
Run 6	71.279200327	5,610.239802356	371.603499119
Run 7	71.034284177	5,618.170198987	366.140639713
Run 8	71.036471405	5,505.952754833	360.973810885
Run 9	71.017811756	5,786.973985060	361.321421024
Run 10	71.018076605	5,664.184136904	361.943817915

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-02-07

EXAMENSARBETE Evaluating ClickHouse as a Big Data Processing Solution for IoT-Telemetry
STUDENTER Adrian Göransson, Oskar Wändesjö
HANDLEDARE Markus Borg (LTH), Anton Friberg (Axis)
EXAMINATOR Alma Orucevic-Alagic (LTH)

ClickHouse för skalbar analys och lagring av massiva mängder data

POPULÄRVETENSKAPLIG SAMMANFATTNING **Adrian Göransson, Oskar Wändesjö**

Insamling och analys av stora mängder data för Business Intelligence är idag inget ovanligt. Insamlingen av data har ökat i aggressiv takt, och i samband med detta har utmaningar att skala upp sin analys- och lagringskapacitet vuxit. Detta arbete har undersökt hur ClickHouse, en kolumnbaserad databas, kan ta sig an dessa utmaningar.

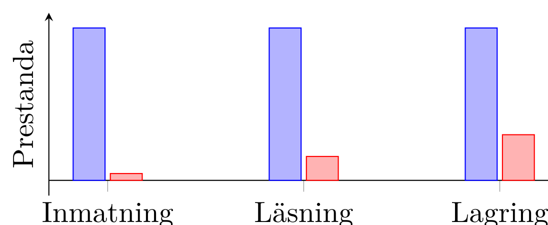
För att få insyn och kunna basera beslut på stora mängder data krävs det att man har en välplanerad infrastruktur som låter en lagra och hämta data på ett effektivt sätt. Relationsdatabasen är en väl beprövad lagringsmodell som funnits sedan 70-talet, där data sparas som rader i olika tabeller. Modellen erbjuder strikta garantier för integriteten av inmatad data och är väl lämpad för många och små transaktioner där en blandning av operationer som läser, skapar eller uppdaterar data görs. I större system, där datan som matas in inte kommer att förändras, utan bara läsas, exempelvis metrikdata för analys, kan dock dessa strikta garantier istället leda till prestandaproblem.

De senaste 15 åren har flertalet databaser som gått från den radbaserade tabellmodellen utvecklats i olika former, till exempel dokumentdatabaser och grafdatabaser. De erbjuder oftast inte samma strikta garantier som relationsdatabasen, och kan därför prestera bra även vid uppskalning.

ClickHouse är en kolumnbaserad databas som ursprungligen utvecklats av Yandex, Rysslands största internetbolag, men som hösten 2021 avknoppades till ett amerikanskt bolag i ClickHouse Inc. ClickHouse är specialanpassat för höga skriv och läshastigheter och utlovar mycket effektiv lagring optimerad för många typer av data.

I detta arbete testade vi ClickHouse hos Axis, ett företag som samlar in stora mängder metrikdata från en bred flotta med IoT-enheter (Internet of Things). Analysavdelningen har stött på utmaningarna att skala upp sin analys, som primärt beror på deras val av datalagringslösning.

Vi utformade en experimentell testsvit för att undersöka främst hur ClickHouse stod sig mot dagens lösning i form av sökdatabasen Elasticsearch. Vi testade inmatning och läsning av metrikdata, samt hur effektivt de olika subjekten lagrade data.



Figur 1: Relativa prestandaresultat i olika tester för ClickHouse (blå), och Elasticsearch (röd).

Våra resultat visade att ClickHouse i snitt var 22 och 6,4 gånger snabbare än Elasticsearch i inmatning respektive läsning samt 3,3 gånger mer effektiv på att lagra datan.