

Halvor Fladsrud Bø
Anders Hallem Iversen
Sveinung Øverland

Achieving scalability in analytical databases used in big data pipelines

Bachelor's project in Computer Engineering
Supervisor: Nils Tesdal
May 2020

Halvor Fladsrud Bø
Anders Hallem Iversen
Sveinung Øverland

Achieving scalability in analytical databases used in big data pipelines

Bachelor's project in Computer Engineering
Supervisor: Nils Tesdal
May 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science



Preface

This paper will describe a project and research related to *scalability in big data pipelines and analytical databases*. The purpose of this paper is to give insight in scalability in the world of big data and the development of a big data related project. Furthermore, the paper will include relevant criteria that is set for the project, how the product have been developed, and the team's execution for researching in regards of searching for an answer to the problem in question.

The team involved for writing this paper has in the time of writing been working for a company that was in a need of a new analytical solution. From there, the team was quite motivated to make that project into a bachelor thesis, with the following reasons behind the motivation:

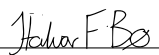
- It is a quite exciting topic, with a possibility to gain a lot of experience and knowledge in analytics, big data, and streaming pipelines.
- It would benefit the company the team was originally working for, and possibly increase the company's value.


The development and the research is based on the methodology of *design science research*, which focuses on developing and monitoring the performance of designed artifacts, and stretching to improve the functional performance of the produced artifact. The team involved has been practicing different techniques for agile software development, especially in context of *The Agile Manifesto*. Highered AS is the product owner of the developed product, and have helped with making this project become possible.

For making this project become possible, one must thank:

- Jon Bäcklund og Karstein Brynstad as representatives from Highered AS, giving the team involved technical resources and guidelines. They have also helped with narrowing down the product's criteria and given a greater understanding of the entire purpose of the project.
- Nils Tesdal, lecturer from NTNU, as the team's supervisor for the project.

The team and the authors of this project and paper:


Halvor Fladsrud Bø


Anders Hallem Iversen


Sveinug Øverland

Task

The team is set on the task of building a new analytical solution for a company named Highered. The overall goal is to create a new solution that fixes the problems and issues of Highered's current solution, where the new focus is on scalability. The new solution needs to handle Highered's current and new demands, requiring flexibility. The project is about creating a new scalable streaming pipeline that is capable of processing and storing events done by the users on the company's platforms. The purpose of the pipeline is to be capable of giving both Highered and its customers' analytical insights about what is happening on Highered's platforms and at the same time be able to handle sudden increases in data volumes and incoming requests.

While developing this task, the team will research how scalability is achieved in big data pipelines and databases, and will look at underlying architectures and implementations of existing solutions to conduct the research.

Summary

Today, many organizations depend on software and applications to keep their businesses alive, and therefore, there is an increasing need of monitoring their systems and end-users to ensure they make the next right business decision. By this, one enters the world of *analytics*, the discovery and interpretation of significant patterns in data. Organizations use analytics to find answers to business questions, find out how their users use their services and extract other meaningful information from their systems, like user behavior and system metrics. To achieve this, one may for example find interests in tracking user-activity data, like login attempts, web clicks, error occurrences, and other types of behavior. For an organization to reap the benefits of analytics, one must first decide on what data to store, how to store it, and at the same time, figure out how to make that data into something useful, which is the first challenge.

From a technical perspective, the demands of analytics have really changed over the last years. Today, there are many systems that handle several thousands of concurrent users simultaneously, tracking and storing their behavior and activity on multiple applications, resulting in gigabytes of data stored every single day. By this, the world of tech meets many challenges in relation to the processing and managing enormous amounts of event data, resulting in developing the term *big data*. Over the last 10 years, the world of tech has developed a lot of different types of technologies and methodologies for making it easier to deal with the problems of big data, resulting in increased interest for both big and small organizations to get into the world of analytics. Analytics has even become a business-idea on its own and is now a norm in almost every modern organization. With this increasing interest in analytics, there is also a higher demand for developers to have knowledge and experience in making analytics come to life, making it even more difficult to be a developer these days.

Analytics is about gathering different types of data, often from different types of sources, to the same place for interpretation. To turn the data into something useful it is normal to process the data before it gets stored, for example converting the data into a correct format or appending additional data to the raw data, resulting in a pipeline of steps before reaching storage. This is the traditional way of thinking when handling the ingestion of analytical events, and each and every organization needs to find out what these steps should be, based on their situation, demands, and needs. For example, for some organizations high performance is important, for some organizations real-time insights are important, and for some organizations being capable of dealing with petabytes of data is important, creating different demands for the different data pipelines and the analytical solution as a whole. This paper will go into detail and describe how analytical databases scale when demand increases, will go deep into the architecture of such a system, and describe an implementation of a big data pipeline to create a greater understanding of this topic.

Contents

List of Figures	10
1 Introduction	12
1.1 Background	12
1.2 The research question	12
1.3 Structure	14
1.4 Acronyms	15
2 Theory	16
2.1 Databases	16
2.1.1 Classes of databases	16
2.1.1.1 Structure	16
2.1.1.2 Scaling	17
2.1.1.3 Consistency	19
2.1.2 Implementations	20
2.1.2.1 Apache Druid	20
2.1.2.2 BigQuery	23
2.1.2.3 BigTable	24
2.1.2.4 PostgreSQL	25
2.2 Streaming pipelines	27
2.2.1 Architecture	27
2.2.1.1 Enrichment	27
2.2.2 Queues	27
2.2.3 Stream and batch processing	28
2.2.3.1 Dataflow	28
2.2.4 Aggregations & windowing	28
3 Method	29
3.1 Extracting knowledge	29
3.2 Implementing a MVP pipeline	29
3.3 The execution	30
3.3.1 Database research & experiment	30
3.3.2 Integration and dashboard	31
3.4 Choice of technologies	31
3.4.1 Google Cloud Platform	31
3.4.2 Dataflow	31
3.4.3 Cloud PubSub	32
3.4.4 Cloud Function	32
3.4.5 Memorystore (Redis)	32
3.4.6 Apache Druid	32
3.4.7 BigQuery	32
3.4.8 BigTable	33
3.4.9 PostgreSQL	33
3.4.10 Dash	33
3.5 Process	33
3.5.1 Role distribution	34

4	Results	36
4.1	Scientific results	36
4.1.1	Overview of the experiment	36
4.1.2	Recorded metrics	36
4.1.3	The results	37
4.1.3.1	Postgres	37
4.1.3.2	BigTable	42
4.1.3.3	Druid	52
4.1.3.4	BigQuery	60
4.2	Engineering professional results	64
4.2.1	Final result	64
4.2.2	Functional requirements	66
4.2.3	Non-functional requirements	69
4.3	Administrative results	71
4.3.1	Work schedule plan	71
4.3.2	The development process	71
5	Discussion	72
5.1	Scientific results	72
5.1.1	Overview of the experiment	72
5.1.2	Recorded metrics	72
5.1.3	Postgres	73
5.1.4	BigTable	73
5.1.5	Druid	74
5.1.6	BigQuery	74
5.1.7	Summary	75
5.1.8	Choosing a database	76
5.2	Engineering professional results	77
5.2.1	Final result	77
5.2.1.1	Strengths	77
5.2.1.2	Weaknesses	77
5.2.2	Functional requirements	78
5.2.3	Non-functional requirements	80
5.3	Administrative results	81
5.3.1	Work schedule plan	81
5.3.2	The development process	81
6	Conclusion and further work	83
6.1	Conclusion	83
6.2	Further work	84
7	Attachments	85
	Bibliography	86

List of Figures

1	A simple visualization of roll-ups from the official website [1]	21
2	An visualization of Druid's architecture [2]	22
3	Visual explanation of columnar storage	23
4	An visualization of BigQuery's tree-architecture	24
5	A simplified version of Postgres's underlying architecture	26
6	Illustration of the planned streaming pipeline	29
7	A screenshot of the Zenhub task-board during a sprint	34
8	A linear growth of a query execution time	38
9	A linear decline of the ingestion time	38
10	Total of 10 GB inserted	39
11	The number of elements inserted per second	39
12	Query times increases when storage utilization increases	39
13	Postgres query execution times being unstable.	40
14	A decline in ingestion times	40
15	Total of 15 GB inserted	41
16	The number of elements inserted per second	41
17	Query times increases when storage utilization increases	41
18	The pipeline for this experiment with BigTable	43
19	A linear growth of a query execution time	43
20	A high ingestion throughput for BigTable	44
21	BigTable - wave 01 - rows written per minute	44
22	BigTable - Wave 01 - an incredible high Dataflow throughput	45
23	This metric was not updated frequently by GCP's metric service	45
24	A linear increase of stored data up to 100GB	46
25	As storage utilization increases, there is no acknowledging change in query times	46
26	Dataflow's and BigTable's throughput being roughly equal	47
27	A linear growth of a query execution time	48
28	BigTable - Wave 02 - A high ingestion throughput	48
29	BigTable - Wave 02 - A high Dataflow throughput	49
30	This metric was not updated frequently by GCP's metric service	50
31	A linear increase of stored data up to 100GB	50
32	As storage utilization increases, there is no acknowledging change in query times	51
33	Dataflow's and BigTable's throughput being roughly equal	51
34	The pipeline for this experiment with Druid	52
35	A sudden increase in query times, peaking at over 2.5 minutes	53
36	The consumer lag from each partition in the Kafka queue	54
37	Druid - Wave 01 - An unstable Dataflow throughput	54
38	Druid - Wave 01 - Visualization of Druid's unscalability	55
39	A sudden increase in query times, peaking at over 2.5 minutes	55
40	The consumer lag from each partition in the Kafka queue	56
41	Druid - Wave 02 - Dataflow's throughput suddenly going down to 0.	56
42	Druid - Wave 02 - An overall increase in ingestion and query times	57
43	A sudden increase in query times, peaking at over 2.5 minutes	57
44	The consumer lag from each partition in the Kafka queue	58
45	Druid - Wave 03 - A gradually decrease in Dataflow throughput	58
46	Druid - Wave 03 - An overall increase in consumer lag and query times	59
47	The pipeline for this experiment with BigQuery	60
48	No noticeable change in query times, except for a unexpected spike.	61
49	The number of rows written to BigQuery per second.	62
50	A gradual increase in throughput	62
51	BigQuery - the number of ingested bytes utilized	63
52	BigQuery - unaffected query times when workload increases	63

53	A diagram of the final pipeline result	64
54	The mean enrichment execution times in milliseconds for an event, from all running tests . .	65
55	A MVP dashboard, visualizing fake events from the stored BigQuery data	66
56	A graph over the average throughput for the enrichment stage under the latest stress test . .	67
57	The time it takes in millisecond for an event to be inserted into BigQuery after it's occurrence	68
58	Query times (s) from stress test	70

1 Introduction

1.1 Background

Highered is a new company, founded in 2015, that has a goal of building a *Global Talent Ecosystem*, a way for recruiters to find students and alumni from over 600 top universities all over the world. The goal is to make it easy for students to find new job offers and recruiters to find talented students from different universities on a global scale. The company is collaborating with *European Foundation of Management Development* (EFMD) [3] network, as the foundation had a wish for a global talent network, connecting students and companies from all over the world. Highered is working towards building that network and does so by maintaining a university-branded job advertisement platform for over 600 universities. Companies can post job offers to the platform and have them exposed to thousands of students, all over the world. With this, Highered offers analytical insights for companies and universities, reporting student-based activity on job and university level.

With Highered's current analytics solution, they power dashboards and reports for every company and university they have an official deal with. Their job advertisement platform has several thousands of visiting students every single day, tracking their activities on the platform, like viewed jobs and companies, and jobs applied for. This way, universities can see how many students are engaging in different university-branded platforms, and companies can see how many students from all over the world are interested in their jobs and their company. This way, analytical insight creates value for Highered's customers, and profit for Highered, making their analytics solution quite important for their business.

Highered is still not quite satisfied with their current analytics solution. They have found their solution slow, making them and their customers wait up to tens of seconds for the dashboard to completely load, resulting in inefficient productivity for both Highered and their customers. Besides, the dashboards are not visualizing the data well enough, resulting in some customers to not quite understand what the data means, giving them no value. From a technical perspective, things are not perfect either. The solution is based on a combination of *Google Analytics*[4], *BigQuery*[5], and *PowerBI* [6]. Google Analytics is an analytics solution itself, provided by Google. All student behavior from the job advertisement platforms is ingested into Google Analytics, giving Highered decent student insights of the platform. The problem is that there is a need for a custom and separate dashboard outside of Google Analytics to offer Highered's customers. Therefore, the event data is extracted from Google Analytics to BigQuery, a data warehouse for storage and querying. PowerBI is a data visualization tool by Microsoft, which builds and hosts the dashboards for Highered's customers, where the data is powered by BigQuery.

This technical solution works, but is not quite optimal and is restricted to some use-cases, like real-time functionality. Overall, the current solution is slow, is costly due to PowerBI, and is not scalable due to a memory problem regarding PowerBI. This means when Highered has more users, the dashboard will eventually fail and crash. Therefore, Highered wants to improve its analytics solution, which is what this project is about.

The goal of this project is to create a more robust and functional analytics solution, in the hope of increasing the productivity of Highered's sales and marketing team, and more importantly, Highered's customers. It is also a way for Highered's technical team to have a more maintainable system, reducing time on maintaining the analytics solution, resulting in an increase in productivity for other types of work. Another goal is to reduce the overall cost of the analytics solution, increasing Highered's liquidity. Therefore, the execution of this project may be to great help for the entire company.

1.2 The research question

As the number of users increases, Highered is currently tracking more and more data about their users, but also wants to be able to track additional user events, compared to what they do now. This means Highered is a need for a new analytics solution, which can handle all the new and existing demands. There-

fore, the objective is to build a new solution, which comprehends with the following criteria stated in attachment A, *The vision document*:

- **Scalability.** The new solution must be able to handle drastic increases of data without having too much effect on performance and cost.
- **Cost.** The current solution is too costly, mainly because of PowerBI. The new solution should preferably not extend the current cost, and should strive to be cheaper if possible.
- **Performance.** The current dashboard often uses way over 10 seconds to load, which is a result of a massive amount of SQL-joins and complex queries. Such load times is not acceptable, and is affecting the overall quality of the system Highered offers to their customers. Therefore, the new solution needs to be able to power dashboards with high performance, and preferably serve the data within 5 seconds.
- **Realtime.** The current solution does not support realtime-analytics, for example like reporting how many users is currently active on their platform and doing a specific action. The company believes adding support for real-time will be quite an attractive functionality for their customers, and therefore, want the new solution to be open for having such a feature.

With collaboration with Highered's Chief Technical Officer, Jon Bäcklund, the team concluded that the new analytics project will consist of developing a new scalable streaming pipeline, to fulfill the new demands. The importance of this pipeline is the ability to scale when the amounts of data ingestion increase, with performance in mind. From a technical perspective, the following criteria were added:

- The pipeline having a throughput of at least 1000 processed elements per second.
- Should be possible to query for the same types of aggregations as in the existing solution.
- The solution should be robust, scalable, and highly available, to prevent loss of events.

With this objective in mind, the team needs to find out what makes an analytical solution scalable. What aspects are needed for such a system to scale when the demand increases? Are there different components that are more important than others, when it comes to scaling the solution as a whole? How does one implement such a big data pipeline, and how does the architecture look like? This is what the team is going to find an answer to, performing a *design science research*. Due to constraints in both time and cost, the research will mainly focus on scalability in databases, as they are the most important component in an analytics solution. Therefore, the keyword of this paper is **scalability**, which forms the research question:

How is scalability achieved in analytical databases used in big data pipelines?

1.3 Structure

This paper is divided into the following eight chapters:

- **Introduction** - an introduction to the project, its background, the research question, and the goals of the project.
- **Theory** - a section containing brief explanations of relevant theory and background information one needs to have for understanding the rest of the paper.
- **Method** - a description of a plan, explaining the execution of the project. Includes a description on how the team decided to approach the problem, and an explanation of why certain technologies are involved to solve it.
- **Results** - a presentation of the produced results, created by performing the plan explained in method.
- **Discussion** - a discussion and reflection of why the results came to be as they are, and how these results can be used to answer the research question.
- **Conclusion** - an attempt of trying to answer the research question, based on the results and discussions stated in this paper.
- **References** - the bibliography utilized by this paper.
- **Attachments** - relevant attachment related to the project and the research question.

1.4 Acronyms

- **GCP** - *Google Cloud Platform*
- **GCS** - *Google Cloud Storage*
- **SQL** - *Structured Query Language*
- **HDFS** - *Hadoop Distributed Filesystem* - collection of software with the capability to utilize a network of computers to process enormous amounts of data.[\[7\]](#)
- **RDBMS** - *Relational Database Management System*

2 Theory

2.1 Databases

In the world of analytics where all different kinds of data are needed for deeper insight and analyzing, such as user behavior and actions, there is a general need for storing and managing the data. Databases have been used since the early stages of the computer and are a system for structuring a collection of data and making it available when needed. Today there is also a focus on performance, functionality, and efficient extracting of information, and because of this there are today plenty of different databases designed for different scenarios and different goals in mind.

2.1.1 Classes of databases

2.1.1.1 Structure

Most databases do not fit or are suited for certain types of usages and are mostly good at the one aspect it was designed for. For example some databases are good at structuring and creating relations, but perhaps perform badly in terms of extracting information of big collections of data. Therefore different databases have been designed around to handle different kinds of usages and have a focus to be good at a certain aspect.

When it comes to structuring data, the most traditional way is to structure the data into tables. *Relational databases* (RDBMS) structures the data into such tables, where each table consists of a specified set of columns, called the table's schema, and rows, representing an entity in the table. In most cases there are multiple tables in the same database with a different set of schema and data, which makes it possible to separate different kinds of data from each other, in other words, normalizing the data. One specialty of *RDBMS*' is the aspect of creating relations between these tables, such that an entity in one table can be connected to one or multiple entries in a different table. This way the RDBMS can avoid data redundancy completely and is able to structure the data in a highly normalized way. Users of a RDBMS can query across tables with the usage of the *Structured Query Language* (SQL), which is a language for querying in such table-structured data. The right term for querying across the relations of tables is called a *join*, where a *join* comes with a level of performance cost, which makes it inefficient to query between a large set of tables. Therefore, even though RDBMS are good at normalizing and structuring data, it comes with some disadvantages and does not suit well for all use cases.

NoSQL is a different type of database that is a greatly popular alternative to RDBMS. NoSQL databases differ from RDBMS' because it allows storage and manipulation of unstructured and semi-structured data. RDBMS' is restricted to its column-schema, but NoSQL opens for storing data as it is, without *extracting, transforming, and loading* (ETL) when storing and sending data. All of this allows for better performance and better handling with big amounts of data. There are different categories of NoSQL data models, which are designed for different kinds of use cases. The most common ones are *key-value stores*, *document databases* and *graph databases*. [8]

Key-value stores has a quite simple data model, where there are value, the data, and a unique key representing the value. From a simple technical perspective, key-value stores are simply just a distributed and persistent map-structure that allows for very fast lookups by key. Key-value stores are the most efficient of the NoSQL versions, but are highly restricted in use cases due to lack of querying, data manipulation, and other essential functionality. With this said, key-value stores are normally used as caches for other applications that need to access data quickly. [8]

Document-oriented databases is a different type of database and a quite popular alternative to RDBMS', Instead of putting data in rows inside tables, the document-oriented database puts data in documents, and documents in collections. A document is a flexible and unstructured data-model that normally uses the JSON-format to describe the data. Documents can also contain nested data-objects, which add an order of a hierarchical nature and ownership in the data. In addition, documents do not enforce a schema, which

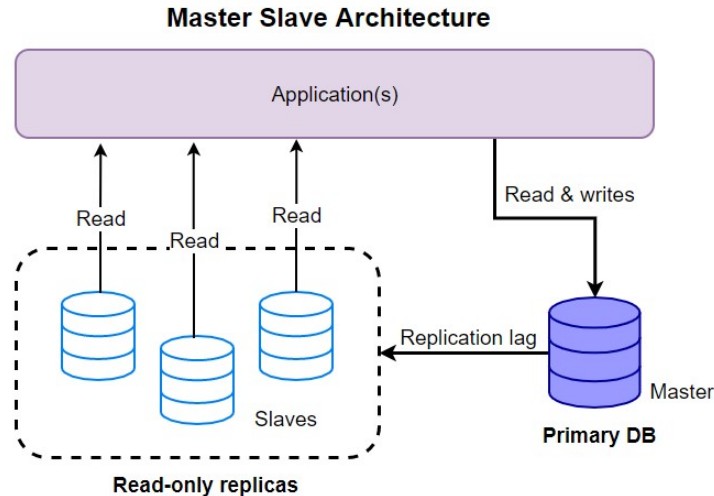
makes it possible to have different fields of data in different documents in the same collection. Document-oriented databases also allow querying and data manipulation across documents and collections, similar to relational databases. Due to the flexibility and the hierarchical nature of documents, accessing data is quite simple, high performance, and quite scalable, and with this said, document-oriented databases have a high variance of use cases, especially in big data applications and analytic-system. A disadvantage with a document-oriented database is the lack of *ACID transactions*, which is the aspect of guaranteeing validity and consistency in the emergence of errors, like power failures, and other types of errors. Instead, developers have to implement custom logic to support ACID transactions.

2.1.1.2 Scaling

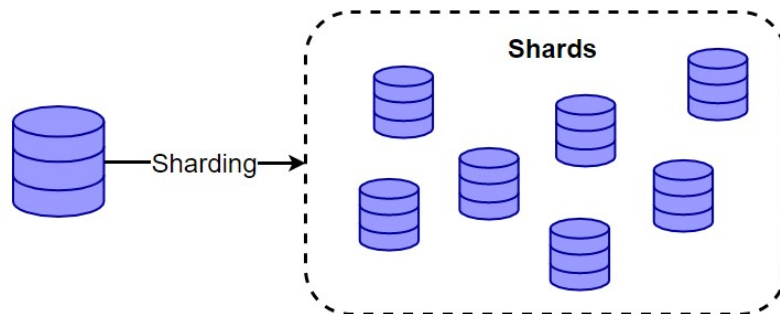
Scalability, in terms of databases, can be defined as how well the database handles the increase of data and incoming requests, in terms of performance. By this a scalable database should be able to handle an increase in workload and throughput when the demand for resources increases, or can be configured to do so. There are different ways to achieve such scalability, with its own pros and cons. Scalability of databases can be crucial because database latency can affect an application's performance, cause frustrated users, and a decrease in sales, which is not good for businesses. Such problems create a demand for being able to somehow scale a system's database, and the solution comes in a set of approaches.

One way to scale a database is through *vertical scaling*. When there is a spike in incoming requests and there is an increase in database-latency, a simple solution can be to increase the hardware resources of the machine the database is running on. This can be an increase in memory or a CPU upgrade, which will give an immediate increase in performance. A downside with vertical scaling is the requirement of reconfiguration and an eventual downtime, which can be critical in some contexts. In addition, there is also a limitation to hardware-upgrades and the hardware itself, which makes it difficult to keep on scaling vertically. By this there will be a certain point where the capabilities of a single database instance will no longer work, and different approaches need to be used. [9]

Another approach to achieve scalability is through *horizontal scaling*, or *scaling out*, which is the process of adding additional machines to the existing database system. There are several different techniques for doing so, and one of them is to use *read-only replicas*. A *read-only replica* is a copy of the original database restricted to only read-operations. Usually in most applications the most common activity is handling read-operations, and by offloading the original database's read-requests to the replicas will intensify the performance of the system as a whole. This also means the original database will alone handle all the write requests and do the job of sending updates to the replicas. This kind of system is also called a *master-slave architecture*. The downsides with this architecture are that it only implies scalability for reads, and all the write-requests will still go to the master. In addition, there will also be a time where the master and the slaves are not in sync, because it takes a bit of time to update the replicas when the writes come in. Therefore, during consideration of using such an architecture replication lag is something that one needs to be aware of. [9]



To solve the problem of increase in writes a *multi-master architecture*, or *sharding*, is possible, which is the process of *scaling out* both reads and writes into something called a *shard*. A shard is a separate instance that handles both read- and write transactions and contains only a subset of the data. In other words, the architecture consists of splitting the database into many small instances with their own part of the data. This way all the load is split between all shards, resulting in an improvement in performance and an ability to scale. However, the database system needs to include extra logic at the application level to know which shards to read or write to. For example if one wants to read a certain row in a sharded relational database by key, the system needs to include logic that connects a distinct key to a specific shard. There are also many ways to solve this problem, but a robust and traditional way is to use a *range hashing strategy*, which hashes the key with a modulo-operation to find the correct shard. An additional problem with sharding is the occurrence of complex querying involving multiple shards. Here the system needs to define extra logic to handle such queries, which also increases the complexity of the architecture.[9]



When a database consists of several read- and write-transactional servers, one starts entering the world of *distributed databases*. A distributed database is a database designed to run a cluster of networked computers. Such a database is not only good for scalability, but it also allows for establishing servers on different locations and regions on behalf of reducing the overall distance between a user and a server, basically reducing request latency. In a distributed database clients can read and write to different servers in the system simultaneously, creating the problem of unsynchronized database transactions. Here there is a need for *consensus*, a general agreement to a synchronized state, which means the different servers need to communicate and synchronize. For example in a distributed DBMS with two servers, also called a *node*, where an incoming transaction occurs for one of the nodes, and another transaction occurs after on the other node, the nodes need to sync before the second node can complete the received transaction. Due to the possibility of network failure and nodes going down it is nearly impossible to guarantee 100 percent con-

sensus in a distributed database, and therefore such systems need to be able to handle mentioned failures to achieve consistency. Here, consensus algorithms are used, like Raft and Paxos, which are algorithms for achieving consensus. They implement protocols for established leaders in the system, for taking control of global operations like schema-changes and deciding the state. The algorithms are based on having one log file for every operation that occurs on every server, and then append entries, commit changes, and synchronizing the log to achieve the same state. The main goal is to achieve consensus in a fault-tolerant distributed system. For synchronizing the logs *vector clocks* are used to create partial ordering of the entries in the log files and generate a common end-state. [10]

2.1.1.3 Consistency

When it comes to *database consistency* it states that only valid data can be written to the database and any valid transaction must change the data in allowed ways. The database will always behave in a consistent state. If a transaction tries to break the rules, for example by inserting invalid data into a field with constraints, the database will rollback to its previous consistent state and revert the transaction's previous changes. This way, the database ensures that the data always goes from one consistent state to another. This does not mean that a successful transaction is always correct, but that it did not break the rules, such as field constraints.

As mentioned before, *ACID* is a set of properties in a database transaction that guarantees validity and consistency in the emergence of errors, like power failures. It makes sure that the database is able to process the transactions with reliability. *ACID* is an acronym that stands for *atomicity, consistent, isolation* and *durability*.

- Atomicity means that the database transactions behave according to a "all or nothing" rule. If a part of a transaction fails, then the entire transaction will be considered invalid and fail.
- Consistency is as stated before, only valid data will be written to the database, according to the predefined rules. In an occurrence of consistency-violation the entire transaction will fail and the database will rollback to a previous consistent state.
- Isolation defines the aspect of making sure simultaneously running transactions do not impact each other. If for example two transactions try to read and modify the same value in the database, it might lead to incorrect results due to the side effects of reading intermediate data values. Therefore, isolation prevents these problems by making sure the transactions are separated and run sequentially.
- Durability makes sure that committed transactions to the database will not be lost, involving processes like transaction-logging and database backups. This way, restoration after failures in hardware or software is made available.

The *ACID*-model is often implemented in relational databases, but is not always implemented into databases with unstructured data, like *NoSQL*, due to violations of consistency in horizontal scaling. [11]

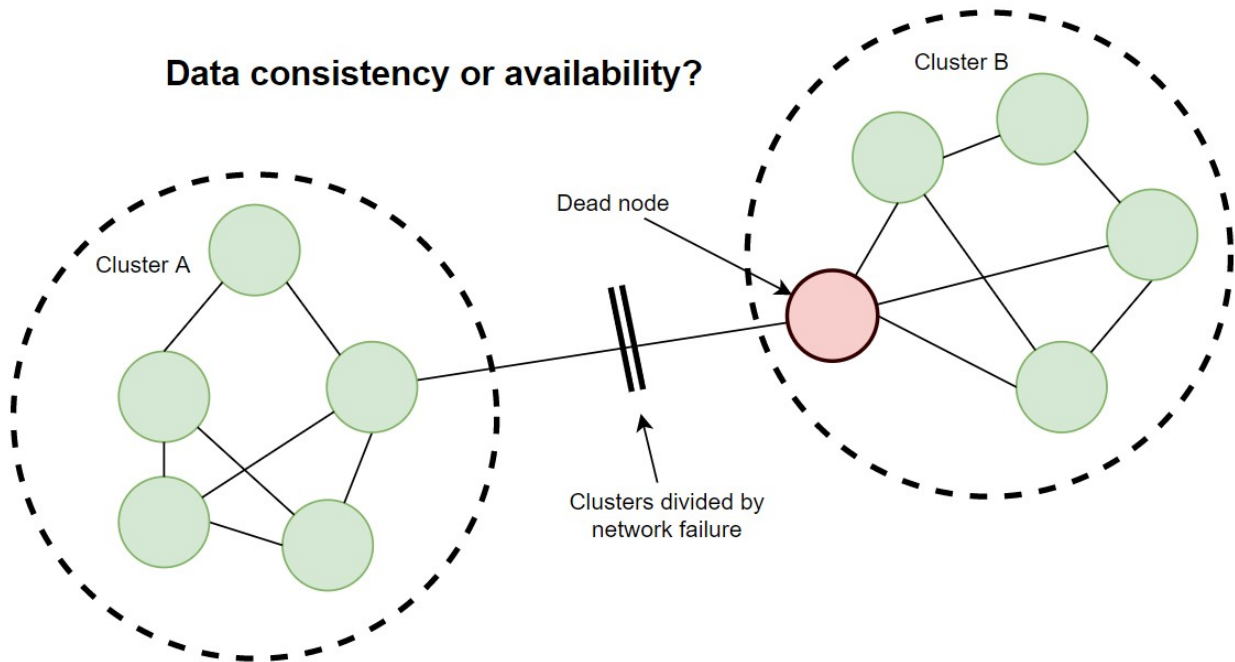
Consistency is also an important term for distributed databases. There is a known theorem in theoretical computer science that states that a distributed database can't guarantee all three of the following properties: *Consistency, availability* and *partition tolerance*. The theorem is called *The CAP theorem*, and is quite important in the world of big data.

- Consistency states that all nodes in the distributed database sees the same data all the time, leading to performing a *read* returning the data of the most recent *write*. Note that this kind of consistency is not the same kind of consistency as stated for *ACID*.
- Availability states that every request should always end up with a non-error response, requiring a functional and operational system all the time.

- Partition tolerance states that the system continues to run despite an arbitrarily number of network errors and crashing nodes.

The reason for the theorem’s statement is the presence of network partition failure, if for example a node goes down and becomes faulty it is then impossible to preserve all three properties. If such a problem occurs one would have to choose between the following options:

1. One can start cancelling incoming requests, decreasing the system’s availability, but preserving consistency (CP).
2. Or one can proceed with handling requests, which ensures availability, but sadly introduces inconsistency (AP).



Therefore, when designing distributed database systems, which has many partitions, one has to decide to choose between either availability or consistency. In real-world applications one would probably favor consistency over availability, due to a need of always responding with the correct data, which is why many databases like MongoDB is a CP database. In the world of big data and analytics, it is not always that important to always respond with the correct data, but instead be available for *reads* and *writes*. Therefore many databases designed for big data turns out to be an AP database, like Cassandra, and is okay with eventually becoming consistent over time, also called *eventual consistency*. [12, 13]

2.1.2 Implementations

2.1.2.1 Apache Druid

Apache Druid is a column-oriented, opensource, and distributed database and is described as, according to the official website, “a high-performance real-time analytics database” [1]. Druid is designed around ingesting massive quantities of event data and offer high-performance queries for analytics purposes. The database uses design techniques from data warehouses, timeseries databases, and search systems to create one single unified system for handling real-time analytics, and can handle data at petabyte scale. Therefore, Druid is good for use cases for example like user analytics, digital marketing, application performance management, and device metrics. [1]

Data inside Druid is split into smaller parts called *segments*. Segments are an immutable collection of data that is defined for a given time period, and Druid itself can be configured to create segments as one wishes. For example, one can create a segment per hour, per day or month, containing all the ingested data for that time period, resulting in time-partitioned storage. By this, when executing a query by a given time interval, Druid only searches the segments that match the given time interval, resulting in fewer data to go through during execution, which reduces query times. When the segments are ready to be set as immutable, they are marked as a *completed segment* and are pushed to a configurable deep-storage, typically a distributed object store, like S3 or a HDFS. Inside the segments, the data is stored in a column-oriented structure, meaning all the values in a column is stored together in the same storage volume, separated from the other columns. This means the data is not stored together by row, but by column, giving a speed boost to queries that select a few amounts of columns.

Before they are marked as completed, Druid can perform raw data summarization on the segments, called *roll-ups*, where it can generate configurable, first-level aggregations on data during ingestion time, resulting in a reduced amount of stored data. By using roll-ups one can for example make Druid automatically create aggregations for a pre-known query, making it easier to perform those queries on the data, which is good for analytical use cases. Data in the segments are structured into columns, containing one *timestamp*-column, a set of *dimensions*, which is filterable and aggregatable columns, and configurable *metrics*, which is a set of counters/aggregations Druid can generate. Druid can also build *inverted indexes*, which are structures designed for rapid-fast search and filtering on string-columned data, which is inspired by existing search systems. [1]

timestamp	domain	gender	clicked
2011-01-01T00:01:35Z	bieber.com	Female	1
2011-01-01T00:03:03Z	bieber.com	Female	0
2011-01-01T00:04:51Z	ultra.com	Male	1
2011-01-01T00:05:33Z	ultra.com	Male	1
2011-01-01T00:05:53Z	ultra.com	Female	0
2011-01-01T00:06:17Z	ultra.com	Female	1
2011-01-01T00:23:15Z	bieber.com	Female	0
2011-01-01T00:38:51Z	ultra.com	Male	1
2011-01-01T00:49:33Z	ultra.com	Female	1
2011-01-01T00:49:53Z	ultra.com	Female	0

timestamp	domain	gender	clicked
2011-01-01T00:00:00Z	bieber.com	Female	1
2011-01-01T00:00:00Z	ultra.com	Male	3
2011-01-01T00:00:00Z	ultra.com	Female	2

Figure 1: A simple visualization of roll-ups from the official website [1]

Druid internally is based on a microservice-architecture, where the entire system is split into several processes, separated by the different tasks, creating a cluster. These processes, also called *nodes* or services, have their own task, like handling ingestion, querying, or data coordination, and can be put into separate machines for scalability purposes. Druid consists of these main core services:

- **Historical nodes** - a service with the role of loading segments into the cluster, and making them respond to any queries involving these segments. In other words, they handle storage and querying of *historical* data, which are completed and immutable segments.
- **MiddleManager/Indexer** - a service for handling the ingestion of incoming data into the cluster. This service has the responsibility to read data from external data sources, for example a Kafka queue, and publishing and off-loading new immutable segments to be available for the historical nodes. They are also responsible for the real-time features within Druid.
- **Brokers** - a service that works as an interface for querying. The brokers' responsibility is to receive queries from external clients, send them to the historical nodes or the MiddleManagers, merge the result, and build a response to the caller.

- **Coordinator** - a service for managing the historical nodes in the cluster. A coordinator has the responsibility of assigning segments to the different historical nodes such that the historical nodes each manage a well-balanced set of segments.
- **Overlord** - a service for watching over the MiddleManagers, and delegating ingestion tasks to the different MiddleManagers in the cluster. They coordinate Druid's data ingestion and publishing of segments.

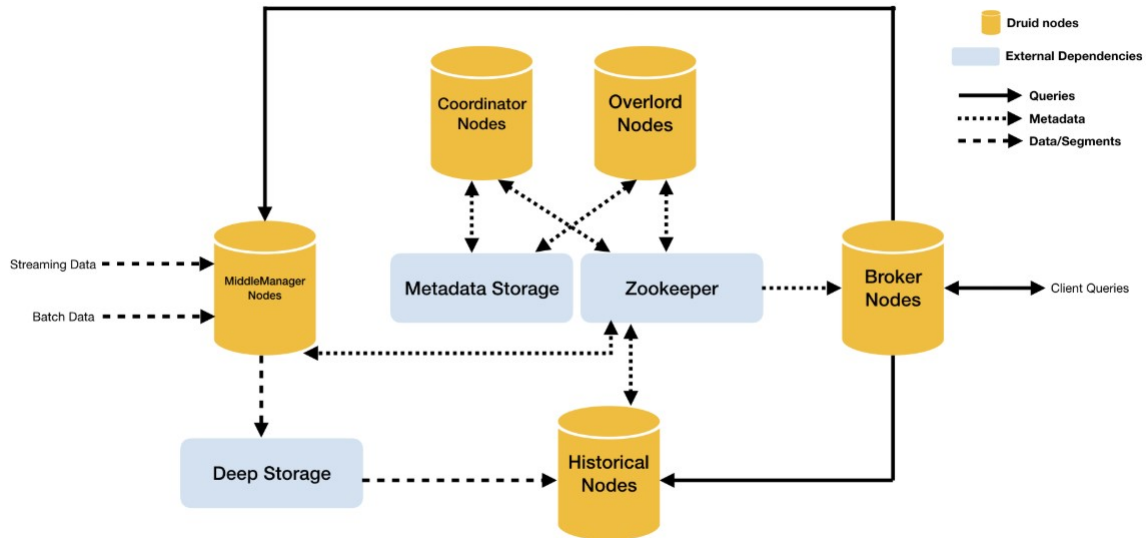


Figure 2: An visualization of Druid's architecture [2]

When executing a query, the job is lead by a Broker process. The Broker starts by identifying which segments to query against based on their time-partitioned identifier and the query's given time-interval filter. After the selection process is done, the query is forwarded to the Historical nodes that are responsible for the selected segments. From here, the queries will be executed in parallel, resulting in a partial result from each Historical, which will be merged by the Broker to produce the final result. This kind of query executing can remind of a small tree, where the Broker is the root node, and the Historicals are the leaves, which can be distributed over different locations. As a whole, this execution-process uses time-partitioning to narrow down the amount of data being processed, in addition to processing that data concurrently, resulting in fast query times. [1]

Druid is currently being used by many big companies, for example like Netflix and Airbnb, which helps to contribute to the opensource database with new features and integrations. Druid itself is defined as an AP database, and comes with a set of web interfaces for configuring data sources, services, and querying data with its own SQL-like query language. Even though Druid seems like a database capable of doing most things, it has a set of limitations, like all databases. There is no possibility to join data between different data sources, but which should not be necessary for analytics, as long as the data is denormalized. There is also no support for *windowing* data, meaning such problems need to be taken care of before ingesting into Druid. One other limitation is the availability of cloud-hosted solutions for Druid, meaning the most common way of running Druid is through self-hosting, making problems like scalability a manually managed task. [2]

The main advantage of using Druid is that it gives a lot of functionality from search systems, timeseries databases, and distributed data stores into one product. Druid is extremely performant when it comes to querying over large datasets, and is quite good at creating aggregations on the fly and compressing data into smaller datasets. It also supports ingestion through streaming and batching, and can offer real-time

querying of event data. If one has a use-case that does not fit into these benefits, Druid might not be the right choice of technology, but otherwise, it is a good fit.

2.1.2.2 BigQuery

BigQuery is one many big data technologies Google has come up with, and according to them, is a "serverless, highly scalable, and cost-effective cloud data warehouse designed for business agility." [5]. A data warehouse is a system built to pull data from many different data sources for analysis, making it possible to build analytical reports from complex queries within the system. Therefore, BigQuery is also a massive parallel query service, capable of running complex SQL-based queries on massive datasets within a short amount of time. It is designed to be able to perform heavy and complex queries on petabyte-scale datasets and give a result within seconds or tens of seconds. Therefore, BigQuery is a good choice for real-time analytics and gives a fully managed and infinitely scalable distributed system.

One aspect that differs modern data warehouses from normal databases is the way they store data. Like most modern data warehouses, BigQuery stores the data in a *columnar storage* fashion, meaning, instead of storing the records together in the same storage volume, it splits the different columns in the record into their own storage volume. One of the benefits with this storage structure is the minimization of scanned values during query execution, meaning instead of scanning through the entire table, the system only needs to do a full scan through the columns that are queried for, reducing query execution times. Another advantage of columnar storage is the possibility to compress the stored data more, compared to row-oriented storage systems. This is because many columns have homogeneous values, making it possible for BigQuery to optimize by applying compressing strategies, reducing the amount of data stored. To summarize, columnar storage is good for performing column-based operations, for example like aggregations and selection of a few columns in the queries, which is good for many analytical use cases. However, the structure is slow on row-based operations, like row-based ingestions, deletions, and updates.[14, 15]

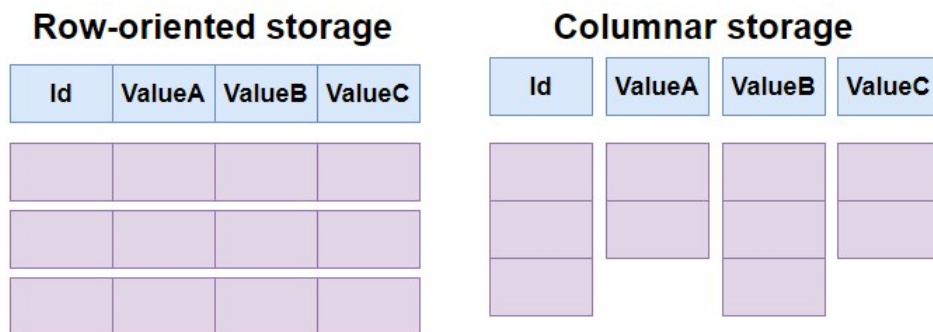


Figure 3: Visual explanation of columnar storage

The underlying storage architecture of BigQuery is based on Google's latest version of *Google File System*, called *Colossus*. Colossus is a distributed file system that consists of clusters of nodes. Each cluster contains one *Master*-node and multiple *Chunkservers*, similar to the master-slave architecture in distributed databases. The chunkservers are servers that contain chunks, where a chunk is a fixed-sized part of a divided file, identified by a 64-bit label assigned by the master-node. With this, the system is creating replications of chunks of frequently used files for performance and availability purposes. Colossus is a file system that guarantees consistency and is capable of distributing data over an enormous amount of machines, and with this, powers BigQuery's storage system.[16]

BigQuery is capable of executing a query on thousands of distributed nodes and collecting the result. This is done by using a *tree architecture*, which is basically capable of pushing a query down a massive parallel distributed tree of nodes, aggregating the results from the leaf nodes, which holds local storage data,

and going upwards towards the root. This way, BigQuery is easy to scale horizontally, execute queries concurrently, and works as a serverless product. It is also worth mentioning that these nodes make up the computational capacity required for executing the query, also called *slots*. How many slots a query execution should consist of is calculated automatically by BigQuery based on the query's complexity, but one can also configure BigQuery to allocate a certain amount of slots for a specific query manually regulation of the slots.

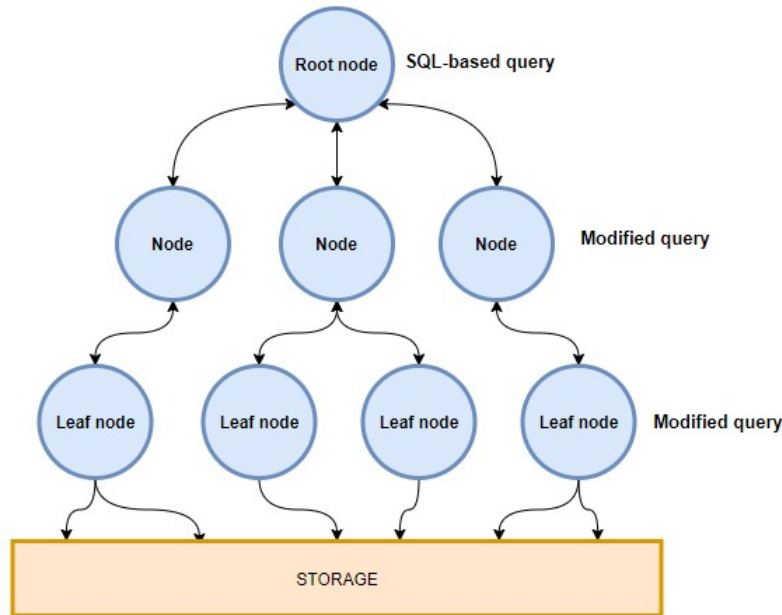


Figure 4: An visualization of BigQuery's tree-architecture

A special feature within BigQuery is partitioning. When creating a table within BigQuery, one can configure the table to be automatically partitioned by a specified column, often a time-based column. This allows for splitting a big table into smaller segments, making it for example possible to perform queries on a smaller subset of the data when filtering on the specified column. This improves query performance, and reduces the amount of scanned data, reducing the cost. Using partitioning is a good idea if one knows on beforehand what kinds of queries one will be executed on the table, especially if one has time-based data.

An important note with BigQuery is the requirement of having structured data. Unstructured data does not fit inside the columnar-oriented structure, making BigQuery not a good choice if one has that kind of data. Another note, BigQuery is only available through Google Cloud Platform (GCP) and has a pricing model that calculates cost per query and gigabytes stored each month. This means one pay per use, meaning one does not get billed when it is not used but can be quite costly when one executes many queries per day. BigQuery is a popular choice in GCP for analytical use cases and integrates with a lot of other external data sources and GCP products.

2.1.2.3 BigTable

BigTable is a distributed storage system developed internally by Google to handle petabyte-scale applications - like Google Maps and Google Analytics. It is a NoSQL database service for dealing with large analytical and operational workloads with low-latency. BigTable is good for applications that need very high throughput for key/value data, and is therefore good for storing for example timeseries data, marketing data, and Internet of Things data. The database is publicly available in GCP and is fully managed by Google.

BigTable itself consists of one or multiple clusters of nodes, and stores all the data in massive tables, hence

the name BigTable. Similar to many other datastores, the tables inside BigTable uses a structure of rows and columns, but differs in that BigTable only allows for one single index - the *row-key*. With this, the tables are a lexicographical sorted key/value map, where the values are extracted by scanning the row-key. The row-key represents an entire row, or entity, which consists of columns grouped into *column family*. The table is also sparse, meaning if a column in a row does not contain any data, it does not take up any additional space. This storage model is what differs BigTable from other databases, it allows for unstructured storage of data by representing by the row-key, making it a good fit for use cases where lookups by key are possible. [17]

To achieve horizontal scaling, BigTable is built out of clusters and nodes. The nodes are responsible for handling subsets of the incoming reads and writes that enter the cluster and has a connection to a subset of the stored data. The tables are stored in Colossus, a highly durable distributed file system, and is divided into *tablets*, a block of key-contiguous rows. The tablets are stored in the format of an immutable lexicographical sorted key-value map, in the form of byte strings, where each table is linked to a specific node. With this, the nodes are responsible for a set of tablets in the form of pointers, meaning the nodes do not store any of the data themselves. Data replication and data availability are all handled by Colossus, meaning the nodes only need to deal with replication and availability of the tablet pointers. The benefit of this type of storage architecture is that recovery from node-failure is quite fast, because only meta-data like the pointers need to be transferred to the replacement node. Also, if a BigTable node crashes, no data is lost, as all the data is stored and managed by Colossus. [17, 18]

BigTable primarily has two kinds of read-operations - read by key and read by a range of keys. This means one can read either a single row or a range of rows with support for filtering on the row-key. BigTable is designed to be extremely fast for executing these read operations by scanning through the row keys. The underlying system uses something called MapReduce, a programming model that allows for distributed and parallel processing of large amounts of data on a cluster [19]. With this, a read operation is distributed between the cluster's nodes but also executed in parallel on the individual tablets. From there, the results are merged and combined within the node responsible for the request, resulting in a highly parallel and distributed execution model, being capable of processing terabyte-scale data with low-latency. [18]

When using BigTable one has to be very careful about how one stores the data. The row-key needs to be designed properly and adjusted to the queries one will perform, especially to avoid a full table scan, which is inefficient. In addition to this, BigTable does not provide support for running complex aggregations, column-filtering, and groupings, and leaves such operations for external processing services. On the other hand, BigTable is extremely good for large amounts of data, especially on key-value pairs. However, BigTable is not suited for small datasets, like less than one terabyte. The reason is BigTable is not able to utilize the benefits of the shards/nodes, because a smaller dataset will result in a small number of nodes and tablets, reducing the parallelism of the database. Therefore, BigTable does not fit all use cases and data sets but is extremely fast on tasks it's suited for, like key-value data on terabyte scales. [18]

2.1.2.4 PostgreSQL

PostgreSQL, or Postgres, is an open-source relational database management system based on SQL. The database is transactional and ACID-compliant, and therefore, guarantees validity in the data. Its use cases are for systems that need to guarantee completeness on success or no effect on failures, like for example banking systems, account services, or system that contains sensitive data. PostgreSQL is also a feature-rich database, having support for things like views, triggers, functions, and full-text search, making it quite useful for various kinds of applications. It is also capable of dealing with many concurrent users and is highly used in web-applications.

Postgres uses a multi-process architecture, with the following types of processes: the *Postmaster process*, *Background process*, *Backend process*, and *Client process*. Overall Postgres is a combination of these processes, some files, and a *shared memory*. Shared memory is a memory that is reserved for caching database results and transactional logs, and can be split into two different parts - *The Shared Buffer* and *WAL Buffers* (write-ahead logging). The Shared Buffer is used to minimize IO-operations, and holds frequently

used blocks of data that is stored in the database. The WAL Buffer is a buffer that temporarily stores logs of transactional-operations, which is used to achieve ACID compliance. The Postmaster process is used to initialize the entire database, basically creating and managing the other running processes in the system, for example the background processes. On the other hand, there are several types of background processes, each with different tasks, for example there are dedicated background processes for logging, for writing buffered data to files and for writing from the WAL-buffer to WAL-files. The Backend processes' job is to execute the queries that the users requests and the Client processes are processes that are dedicated to serving a client-connection, initiated by the Postmaster process. [20]

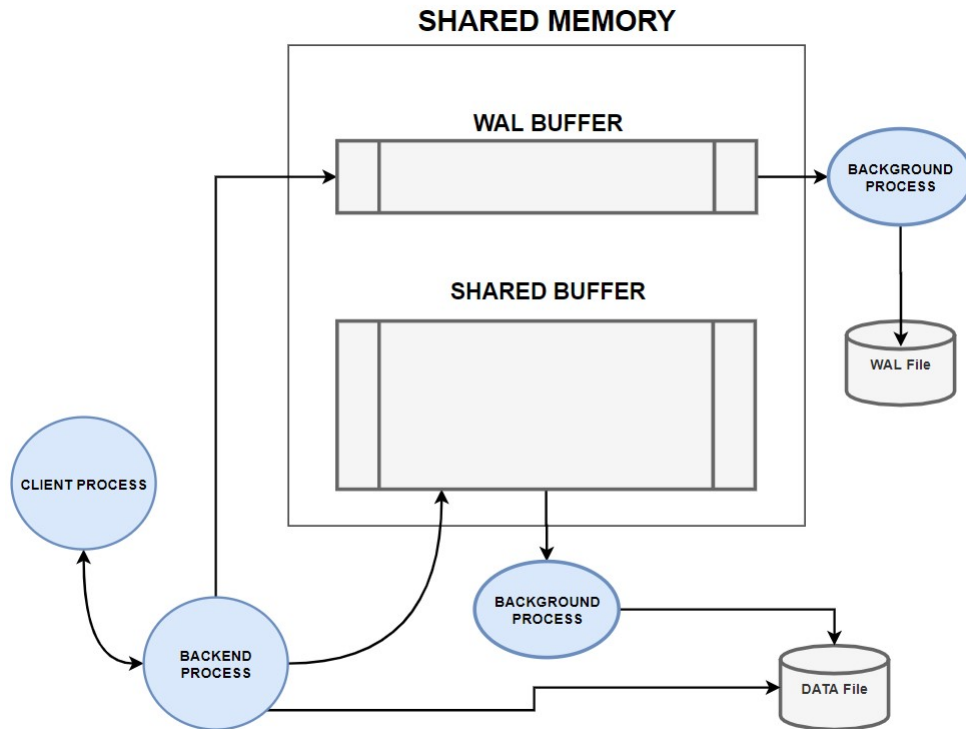


Figure 5: A simplified version of Postgres's underlying architecture

Postgres uses a row-oriented storage model, meaning all the data within a row are contiguously stored together in the same data volume. With such a storage model, the database is fast for row-based operations, like adding, updating, and removing rows. The data in Postgres is stored in tables, where each table is split into gigabyte-sized segments. Inside these segments one finds an array of *pages*, where a page is a fixed-size block, normally 8Kb sized, containing the rows. Inside a page, one also finds something called a *heap file*, which is a collection of pointers referencing the rows inside the page, playing the role as an index for each stored row. The heap file is always placed at the start of the page and the rows are placed at the very end, leaving empty space between them. [21]

For executing a query the job is assigned to the Backend process. The Backend process starts by parsing the query and generates a set of possible methods for executing the query. One of the methods might be to execute a full sequential table scan, scanning through all the rows in the table. Another way might be to do an index-scan, scanning through an index structure that contains indexes that point directly to matching rows. An index-scan is mostly more efficient because one processes far fewer amounts of data to find the matching rows. However, it is also worth mentioning that for some queries they are possible to run in parallel with the help of other Background processes, but these are mainly SELECT-queries that do not affect or cause problems for any transactions. Once the Backend process figures out which of these methods are the least costly to execute in terms of performance, the Backend process executes the method

by directly reading the data-files, and serves the result back to the client process.[22]

Postgres also supports table partitioning, making it possible to create partitioned tables by a selected key column. This can give large increases in query performance, due to fewer data to scan through during execution [23]. It is also worth mentioning that the database supports master-slave replication, meaning Postgres can create a cluster of nodes containing copies of the data for read-operations [24]. All the read-operations can be split between the different replications, increasing the overall performance of the system. In conclusion, Postgres is a pretty popular database, with its rich support of features, ACID compliance, and the possibility to horizontally scale, and as a result can be used in a huge variety of applications with different needs.

2.2 Streaming pipelines

There are two ways of dealing with moving event data into databases for analytics purposes. Moving the data in batches, for example every day or every hour, or continuously moving the data as it comes in. Pipelines that continuously moves the data as it comes in are called streaming pipelines.

2.2.1 Architecture

The architecture of streaming pipelines in a general sense is that the pipeline is split up in several steps, the first being event ingesting which is a step that consumes event data from many clients and queues the data for the next step to read. The next step would be data processing which can include a lot of internal steps and branches to other steps while maintaining the similarity that this step(s) has the potential to alter the data in any way it deems fit. The third and last step is data storage. The importance of having separate steps in the streaming pipeline is that it allows each step to function independently of each other and in the case one step has less throughput than the other steps it can scale up without affecting the other steps.

2.2.1.1 Enrichment

Enrichment is a stage in most pipelines that prepares the event data by mapping it and adding supplementary data from other data sources or aggregating it to be fit for purpose, either by optimizing the data for the queries that the analytics system will run or to fit the data to any format requirements that the destination database has.

2.2.2 Queues

Queues are an important part of any pipeline as they allow event data to be temporary stored in a manner that allows the rest of the pipeline to read event data as they come in, but also buffer messages that cannot be processed right away. Queues are perfect for this application because they offer $O(1)$ (avg & worst case) for insertions, $O(1)$ (avg & worst case) for deletion with $O(n)$ (avg & worst case) on space usage. The various big-O notation attributes of queues mean they scale very well and can be distributed for even larger cross- region/horizontal scaling.

Kafka is Apache's distributed queue system which is a very popular choice when building streaming pipelines. Unlike Google's PubSub Kafka is not managed, so it requires the user to set up a server or cluster to run Kafka, and deal with network setup and maintenance.

PubSub is a managed queue made and serviced by Google that guarantees "at least once" delivery but does not make any promises regarding delivery ordering as the scaling and availability it offers cannot make the necessary checks to provide this. All this considered makes PubSub an easy to use, endlessly scalable solution for ingesting event data into a pipeline.

2.2.3 Stream and batch processing

There are various advantages and disadvantages to choosing streaming over batch processing when creating pipelines. Batch processing allows the pipeline to know how much event data there is to process before it has even started, and can therefore easily scale to fit the need of the user, considering time and size restrictions; By for example running when event data exceeds 10gb or 4 hours have passed. Streaming pipelines, however, allow for near live analytics data, meaning it becomes possible to show updated information all the time and also show metrics that would otherwise become impossible or inconvenient, like for example showing how many active users a site has. Streaming processing has some major difficulties compared to batch processing, a streaming pipeline cannot scale quite as easily because it cannot tell if there has been a sudden spike in activity or a prolonged increase in activity without exchanging information between the first ingest step and the second data processing step which would in turn make it unrealistic to distribute the pipeline processes, and would defeat the purpose of splitting the pipeline into several steps in the first place.

2.2.3.1 Dataflow

Dataflow is a service provided by Google that allows for running managed, distributed Apache Beam pipeline code in the cloud, supporting both batch processing and stream processing. The advantage of using Dataflow is that it can automatically scale Google's Compute Engine VMs to deal with more or less any load, from small personal project all the way to huge enterprises that generate terabytes of data a day. It also allows for easy updating of already running streaming pipelines and Google also provides various libraries to easily integrate a pipeline into other Google services, like databases, caches and long term file storage solutions.

2.2.4 Aggregations & windowing

Aggregations are an integral part of data analysis, especially when near live data is being presented in a dashboard for external users, this is because aggregations allow for optimizing data for specific queries before it is put into databases, meaning query times can be dramatically reduced because there is no longer a need for expensive joins and slow SQL queries.

3 Method

To answer the research question "how is scalability achieved in analytical databases used in big data pipelines?" in the form of design science research the team plans to implement a minimum viable product (MVP) of a presumed solution to a scaleable streaming pipeline, and then iterating on that MVP to continuously improve the pipeline to answer the research question.

3.1 Extracting knowledge

The companies or tech teams that need scalable analytics pipelines tend to be really large enterprise tech companies, with a lot of resources and development time, not a lot of the knowledge the team needs is well documented or recorded in anything other than technical specifications or research papers. Therefore the team will dedicate a lot of time at the start of the project period just to finding and reading research papers with relevant information. The team will find research papers about subjects like among others: "streaming pipelines", "big data" and "database implementations". With the goal being to create enough knowledge among the team to avoid common pitfalls other teams have experienced and create an overview of what should be implemented.

3.2 Implementing a MVP pipeline

To get experience building streaming pipelines and avoid unknown difficulties down the line when implementing the actual streaming pipeline Highered is likely to use, the team started by creating an MVP implementation of a pipeline. This required some overview and planning on what needed to be created, so the team created this illustration showing a general overview of a streaming pipeline and event data movement.

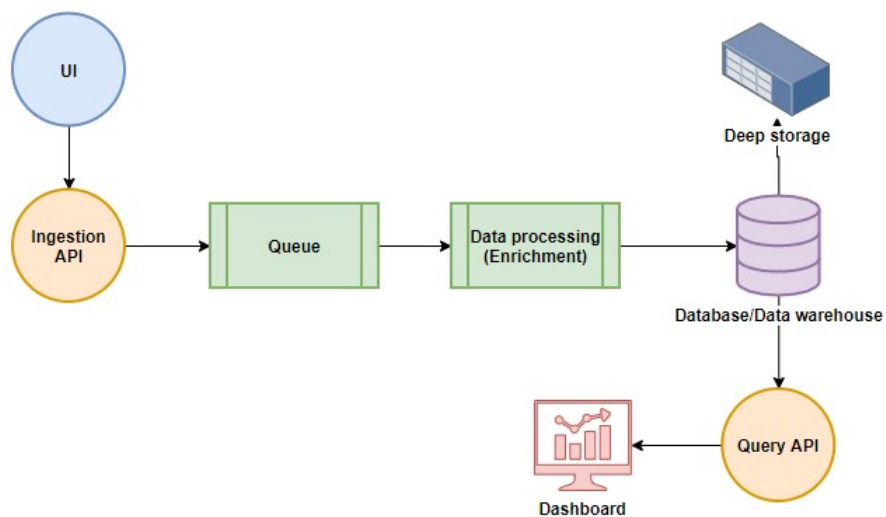


Figure 6: Illustration of the planned streaming pipeline

To ensure scalability of the pipeline and save time when implementing the MVP pipeline the team decided to use managed services as much as possible. So for the ingestion API the team decided to create a simple cloud Function that inputs any input from the UI into a Pub/Sub queue. Both Cloud Function and Pub/Sub are Google Cloud services that are managed and scale automatically to fit more or less any need. For data-processing the team used Dataflow, another managed Google Cloud service. The data-processing stage can change the data however it wishes, through supplementing it with other data from various sources, aggregating it, de-normalizing it, or normalizing it. The plan is to use this stage as an enrichment stage, keeping the data as denormalized as possible in storage, resulting in fewer joins and lower query execution times. The enrichment stage does this by creating several branches of data movement and

can window data to achieve what it wishes. A streaming pipeline wants to do this kind of processing to prepare the data before it reaches storage, resulting in as little computation as possible when extracting information from the storage. The team's first draft and thoughts of the pipeline is documented in attachment C, which is a *design document* for the rest of the Highered's tech team to read and share their thoughts.

3.3 The execution

3.3.1 Database research & experiment

After doing some quick research, the team found that a good way to find out what makes a big data pipeline scale is to test the scalability of the different components in the pipeline. If one can find out what makes the different components scalable or not scalable, one can point out if the different components are essential to the scalability of the entire pipeline or not. By looking at the underlying properties and functionality of different types of the same component, one can answer why some types of component scales and why some types do not after running some tests. By taking the database component as an example - if one performs a test on different types of databases, databases with different properties and functionality, and assuming that some scale and some do not scale, one can draw a conclusion about which aspects in a database is important to achieve scalability. After taking a look at one component, one can perform the same tests on the other components, and see if there is anything related between the results to see what really is important for achieving scalability for the entire pipeline.

It is a good idea to clarify what one means about scalability. In this project, one is most interested in the performance of the system, and how the performance is affected when the system gets increasing workloads. The most important part of the project is knowing how much does the time of extracting information from the storage gets affected when the quantity of these workloads grows. By this, one is most interested in finding out what makes the database-component scale, in terms of query performance. It is also interesting to find out how the data-processing component scale because of its significance in the pipeline's throughput. Sadly, due to constraints in both time and cost, the team is only capable of running tests on only one component. With this restriction, it, therefore, makes the most sense to only dive deep into the scalability of databases, by running mentioned experiments and try to find an answer to the research question from the results.

To test the database-component of the pipeline, one needs a set of databases, with different types of aspects and underlying properties. Today, there exist a lot of different types of databases one can choose, like relational databases, timeseries databases, full-text search engines, and NoSQL databases. For this experiment, one would like to have a database that fits into the world of analytics, and therefore, the team has chosen these four databases - *Postgres*, *BigTable*, *BigQuery* and *Druid*. The team added Postgres to the list because it is a database one can assume with high confidence will not scale well. The other databases are databases built for analytics, with different architectural designs and underlying characteristics, but with some similarities, making it clear which aspects of the databases are more important if some of the scale more than the other.

The team has decided that a good way to test a particular database is to ingest massive amounts of data into the database from the implemented pipeline. By doing that, one can monitor the database's query performance over time, as data gets ingested, by executing queries in given intervals and tracking the time it takes to execute the query. The resulting query times will give a good indication of if the increase of data affects the database's query performance, which expresses its ability to scale. It is also a good idea to monitor the database's ingestion times during the ingestion because it will illustrate if the database's capability of ingesting data gets affected by the increase of stored data. Ingestion times is also important because it affects the pipeline's throughput, the quantity of elements gets outputted from the pipeline every second.

The plan is to generate or find a big data set that can be used for ingestion. For each database, one splits the ingestion into three different waves. In the first wave, one ingests 1 GB of data into the pipeline, the

second wave 10 GB, and the third wave 50 GB. It would be more interesting if one ingests data up to terabyte level, but due to time- and cost-constraint the experiment is limited to gigabytes. The reason for the waves is to see if there are reasons to even proceed to the next wave if the database clearly does not scale from the very beginning, because running these tests can become somewhat costly in the end.

After running the tests on all four databases, one can try to find similarities in architecture and design in the databases that scaled, and find what functionality or property is the databases that did not scale missing compared to those who scaled. This will give signs of what aspects and designs of a database are important for making it scalable. The overall plan and execution for this experiment are documented in attachment D.

3.3.2 Integration and dashboard

To complete the main objective of the project, which is creating a scalable analytics pipeline for Highered, the team needs to find out which of the databases in the experiment fits Highered the best and should be implemented into the final solution. After executing such an experiment, one should have developed a lot of experience and knowledge about the databases in context, and with that, should be able to create reasonable comparisons to decide which database fits Highered the best, at the same time, fulfilling the requirements of the project. The team has also decided it is a good idea to implement an MVP of a dashboard, visualizing the final product, and showing that it is actually working as it should.

3.4 Choice of technologies

3.4.1 Google Cloud Platform

Highered has most of its technology stack and applications running on the Google Cloud Platform (GCP) - Google's cloud computing services. On GCP, one can find a big variety of tools and services for data analytics and building big data pipelines ¹. The team, therefore, found it reasonable to build the data pipeline on the same platform, keeping Highered's stack on the same cloud environment. In addition, the services on GCP are designed to easily integrate with each other, making it a good idea to build the pipeline based on the services GCP provides, even though there exists a lot of other decent software possibilities outside of GCP, like in the world of opensource.

3.4.2 Dataflow

According to the official webpage, Dataflow is described as:

Unified stream and batch data processing that's serverless, fast, and cost-effective.[25]

Since the plan is to utilize enrichment in an attempt for improving performance, the data transformation needs to occur in a processing step in the pipeline before it reaches storage. It is exactly this kind of processing Dataflow is made for, in addition to its streaming and batch processing capabilities. Since Dataflow is also a service in GCP, advertised with automated horizontal scaling, it is highly suitable for this project. With Dataflow, one does not have to bother with setting up and manually scaling the system, since it is all managed by Google, making one only focus on developing and polishing the pipeline itself. In addition, Dataflow also supports windowing, making it possible to calculate aggregations of the data in a given time-interval, pre-processing the data before it reaches storage. With this kind of functionality one can reduce query times, since the data is already in aggregated form, which also opens up for real-time possibilities. Dataflow uses Apache Beam ²for building and developing the pipelines, which introduces Java to the stack.

¹List of GCP Products - <https://cloud.google.com/products>

²Apache Beam - Opensource SDK for defining stream and batch processing workflows - <https://beam.apache.org/>

3.4.3 Cloud PubSub

As mentioned before, a queue is an important part of a big data pipeline, since it can buffer data the pipeline is not able to ingest, due to the restrictions to the number of events the pipeline is able to handle simultaneously. Cloud Pub/Sub is such a queue, managed inside of GCP, advertised to handle enormous amounts of ingesting events at any scale[26]. It is also highly compatible with Dataflow, making it easy to create a pipeline that integrates the two technologies. As a service in GCP, it is also fully managed by Google, which again, results in fewer things for Highered and the team to maintain. With all of this in mind, the team found Cloud Pub/Sub a natural choice for a queue, also due to the fact, an alternative choice would lead to either moving to a different cloud environment or dealing with scaling and managing the queue manually.

3.4.4 Cloud Function

All the events will be created by users of Highered's front-end applications. This means, there will be a need for some sort of API to receive all the events and ingest them into the pipeline. For this reason, the API also needs to handle a huge amount of concurrent requests and should never lose an event, sensing a need for scalability and high availability. Cloud Functions, as stated by Google, *is Google Cloud's event-driven serverless compute platform*[27]. It is a product inside of GCP that offers developers to deploy and expose a single function as an API, running on an autoscaling serverless architecture, resulting in a scalable and highly available API without the need for provisioning. Such a service fits perfectly for the use case of the project, making it a natural choice.

3.4.5 Memorystore (Redis)

For the planned enrichment process, there must be existing data somewhere to be used as enrichment. That data will most likely exist inside a protected database and made available by some API, meaning one will have to send a request to fetch the data. This request may consume a lot of time, due to network and database latency, slowing down the entire pipeline. It would be more efficient to fetch this data from a close-by cache, containing the necessary data. With the usage of such a cache, one would have to ensure it's availability and scalability, otherwise, the pipeline would fail to cache the data.

Memorystore is a fully managed in-memory data store service on GCP, capable of running Redis³ instances, which can be used as a cache. The service guarantees availability and can be scaled manually when needed. To avoid a lot of provisioning and ensure high availability, Memorystore is a good fit for this problem, is the selected choice done by the team.[28]

3.4.6 Apache Druid

Apache Druid is an opensource analytics database, built with high performance and real-time in mind[1]. The database will be used in the experiments as a potentially scalable database and might be a good fit for Highered. The reason for selecting this particular database is the fact that Druid's sole purpose is to function as an analytical database, designed to handle all the different challenges related to analytics. Apache Druid is also a combination of different types of databases, hence the name of a shapeshifter, a Druid, giving it a rich amount of useful functionality from those different types of databases. For this reason, Druid will, therefore, be an interesting database to look at, especially in terms of its ability to scale. It might also be less costly to maintain a database oneself, instead of using a cloud-managed service, making Druid a good opportunity to test that hypothesis.

3.4.7 BigQuery

BigQuery is a technology of a data warehouse that is also a part of Highered's existing solution. The data warehouse is like many of the other technologies on GCP, it is advertised as serverless and highly scalable, removing the need for provisioning the service oneself. It is reasonable to see if Highered should stick with

³Redis - an opensource in-memory data store - <https://redis.io/>

using BigQuery, since it is a piece of technology that the tech employees of the company are already familiar with, lowering the barrier for learning the new system. Besides, BigQuery is designed to be able to execute queries on petabyte-scale in a matter of seconds or tens of seconds, making it interesting to see how well it scales, which also awakes the curiosity of how it is capable of scaling so well. Therefore, BigQuery is a reasonable choice for the experiment and is a technology the team should get more knowledge of.[14]

3.4.8 BigTable

BigTable is another fully managed database on the GCP, built for operating large analytical workloads, powering popular applications like Youtube and Google Earth[29]. BigTable is a NoSQL database, designed to handle data of petabyte-scale at a low-latency level, with seamless scaling and high availability. It is a database that fits well for analytical operations on large-scale data, and has a different type of underlying architectural design compared to BigQuery and Druid, making it an interesting choice for the experiment. The diversity between the databases in the experiment would help to make it more clear which aspects of the databases is important for their scalability, and therefore, the team decided to make BigTable a part of the experiment. With this, one also opens for that BigTable might be a good fit for Highered's new analytics solution.

3.4.9 PostgreSQL

PostgreSQL, or Postgres, is a transactional database, meaning it is architected to be ACID-compliant, which is a critical feature for application-level systems. While being compliant with this standard, the database is provided with a row-based structure, which combined does not result in a good analytical database. The team has a confident assumption that PostgreSQL will not handle reads well when the amount of data stored inside increases, making it unscalable. For the experiment, it is quite a good idea to involve an unscalable database in the tests, making it easier to see what aspects the unscalable database is missing compared to the scalable ones. Therefore, the team has found it reasonable to include Postgres in the experiment, which will also validate the team's assumption.

3.4.10 Dash

For the sake of the experiment, the team will build a simple dashboard for monitoring the query times and the ingestion times during ingestion. A reasonable and quick way to do this would be to use Dash, a Python library for building web-based analytic apps, built by a company called Plotly. Meaning, the timings of the queries will be done through Python-code and visualized with Dash, making it easy to keep track of the result while the experiment is in progress. After each test, the Dash-developed app will export the results, which will be imported into Google Colab for further analysis.

3.5 Process

The project does not have any form of users, since the end goal is to only make a functional streaming pipeline, according to the demands. It was also clear from the very start what the demands were, in addition to the fact that those demands would most likely be static through the entire process. This entire project and paper are also more about the research, than it is about the development of the pipeline. Therefore, using pure Scrum in the development process would not be the best fit for this project, and the team preferred to rather use a variant of a lean inspired scrum process instead.

With this in mind, the team decided to go for running two-week-long sprints - with a sprint planning meeting at the start of every sprint. During such a meeting, there will be set an achievable goal for the sprint and a decision of which epics and corresponding tasks would be necessary to achieve this goal. The epics were all given a deadline, spread across the sprint period, and also written on an online task-board to allow easy access and overview for everyone involved. The team highly values the act of not wasting time on unnecessary and time-consuming tasks, so a lot of emphasis was put on prioritizing what was needed to get done versus what could be delayed. Being an agile process both this prioritization and tasks could

change underway in each sprint. In addition to the sprint planning meeting, the team also had standup-meetings every day, together with Highered’s tech team, resulting in Highered’s CTO was always up to date with the status of the project and it’s plans forwards, creating a continuous feedback loop between the team and Highered.

When it comes to the task-board, the team decided to use a technology like Zenhub for managing the tasks, containing a section for a backlog, blocked tasks, in-progress tasks, and completed tasks. The tasks were also ordered in prioritization, meaning the most prioritized tasks were on the top of that section. This way, one was always taking the tasks that were of most importance, from the top, ensuring that the most valuable tasks were always completed at the end of the sprints. In addition to this, together with the challenges of Covid-19, the team felt that Zenhub was perhaps the not optimal task-tracking software for the team as a whole, and ended up experimenting with Todoist during the last month.

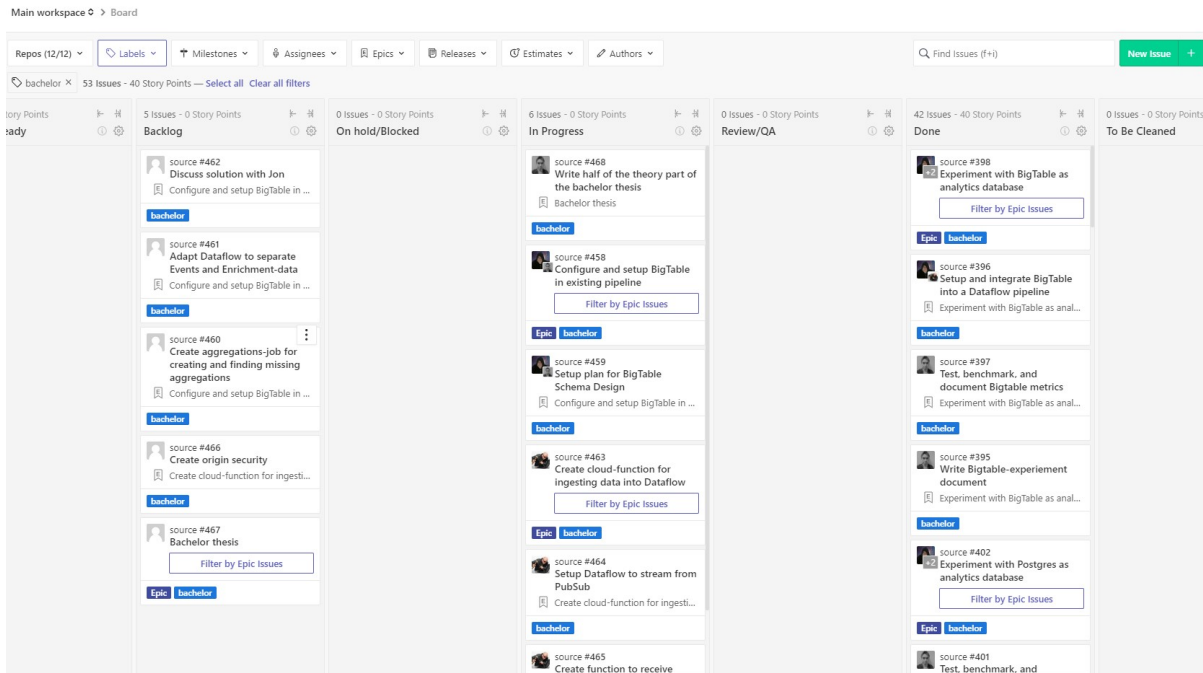


Figure 7: A screenshot of the Zenhub task-board during a sprint

Other than the main development process, the team created various documents to support and reaffirm the development process. These documents consisted of a Gantt diagram, laying out the time-frame of the project, and a technical design specification, describing the team’s various thoughts in preparation for implementation that was explained to and reviewed by Highered’s CTO, Jon Bäcklund. In addition to this, there was also created a work contract between the team, defining core hours, roles, meetings, and rules for violations.

The process described here differs a bit from the process described in the work contract the team signed. This is because not all the items described in the work contract were achievable or feasible amidst the Covid-19 pandemic. The team also had a team member that was struck by illness and hospitalized during the project period, and therefore, could not participate to the extent that the work contract describes, resulting in the team having to adjust the process to fit into a new reality of home-offices and a mostly missing team-member.

3.5.1 Role distribution

For the research and the development of the project, the team wanted everyone to work with all the different types of tasks, and not restrict the workforce into specific areas. With this, the team decided to go

for a flat hierarchy for the development, but instead assigned responsibilities in forms of roles for process-related tasks. As stated in the work contract, the roles for each team member is as following:

- Anders Hallem Iversen: Responsible for delivery - Quality assurance of what is delivered. Anders is responsible that the generated artifacts for the project are delivered on time, and has the responsibility to make sure that the delivery is of a high enough quality to match the team's ambitions. If the delivery is too weak Anders will make sure the team increases the quality before final hand-in.
- Sveinung Øverland: Responsible for meetings - Sveinung should make sure that every meeting with the supervisor is planned and agreed in a reasonable time, and that every member is informed about the upcoming meetings. In addition Sveinung also has to make sure that the team is preparing for and documenting every meeting with the supervisor, the client, or any other people of importance.
- Halvor Fladsrud Bø: Responsible for maintaining documents of importance. Halvor should make sure all important documents are available all the time for the rest of the team and should structure and organize the documents in a reasonable way. This goes for both text-documents and code. In addition Halvor should make sure of the availability of documents that should be shared with external parties, and make sure that these parties are aware of the access to these documents.

4 Results

4.1 Scientific results

This section will mainly elaborate on the results of the experiment, talking about how the tests ended up being performed, what the numerical and metrical results are, what they mean, and how the experiment overall went. The experiment's main focus is to find out which aspects of the databases affect their ability to scale, looking at how they scaled on the different tests and their internal design and architecture.

4.1.1 Overview of the experiment

The tests on the databases ended up being performed quite differently from each other. It is worth to note that these experiments ended up becoming costly, resulting in some tests ending earlier than planned and involved fewer amounts of ingested data compared to other tests. It is also worth noting that these databases are different from each other, meaning some queries do not have an equivalent in the other databases. Especially for BigTable, which was the only NoSQL database in the experiment. During the experiment period, the pipeline had been improved on, resulting in a higher throughput rate from Dataflow on some tests, creating differences there. In addition, the ingested dataset did also change during the tests, due to different setups and configurations, meaning not all databases ingested the same data. With all this said, the results of these tests are not numerical comparable between the databases, which was not the main intention either. The databases have quite different architectural designs, making it not always sensible to compare their numerical results, even though for some databases it does. The intention was to see how each individual database was able to scale on their own, and what aspects made them scale and not scale. Just to make it clear, this experiment is not a benchmarking experiment, with the intention of comparing the different databases.

4.1.2 Recorded metrics

Since some of these databases are fully managed by GCP and some are not, one was not able to track the same metrics for every database the same way. Even for the GCP services one was not always able to get the same types of metrics, resulting in different ways to extract these metrics. The metric most affected by this was the *ingestion time*, which also makes sense because the different databases have different ways of tracking this metric. Another affected metric was *storage utilization* over time, especially since some databases/metric-systems did not update the metric frequently enough. This issue ended up with some metrics being calculated by other metrics, and some metrics extracted directly from GCP's monitoring services⁴, making it not sensible to compare them with each other. Overall, the team ended up trying to extract these metrics from the experiments:

- **Query times** - Monitored and calculated from the Dash-monitoring dashboard.
- **Ingest times** - Extracted from GCP's metric service, and had to exploit SQL's default value functionality to calculate this metric.
- **Dataflow throughput** - Extracted from GCP's metric service, describing how many events Dataflow was able to process and insert into the database each second.
- **Storage utilization** - Sometimes extracted from GCP's metric service, but sometimes calculated with the help of Dataflow's throughput metric.

⁴Monitoring service - GCP have a monitoring system, collecting metrics for most cloud resources running on the platform

4.1.3 The results

4.1.3.1 Postgres

Postgres was the first database to be tested during the experimentation. The database was set up and hosted on GCP, running on a quite powerful machine with 64 virtual CPUs and 240 GB of memory, making sure the database was not bottle-necked by hardware resources. For this database the team found a relevant dataset from Kaggle⁵, containing 1 terabyte of event data. Smaller amounts of this data was ingested directly into Dataflow, which contained an enrichment step that enlarged each event with fake job posting data from HigherEd's development database. The enrichment data was fetched from a nearby Redis-cache, merged with the ingested event data, and from there inserted into Postgres using Apache Beam's support for JDBC. The tests stopped after inserting 15 GB of data into Postgres, since the database ended up not scaling from the very beginning, and there was nothing more to gain by inserting more data.

Postgres summary	
Database:	Postgres
Hosted:	Google Cloud Platform
Total data ingested:	15 GB
GB ingested, first wave:	10 GB
GB ingested, second wave:	5 GB
Ingested time metric:	Calculated using SQL default value
Dataset:	Dataset from Kaggle
Scalable:	No

The data was inserted into a table with a schema defined after the fields of the dataset. Also, there was a timestamp typed column with a default value set to the current time, used to calculate the ingestion time metric. Before ingesting to the database, Dataflow added a timestamp value to every event before they got ingested into the database, making it possible to calculate the ingestion times. This ingestion time is therefore not 100% accurate, but gives a good enough approximation of the metric. The query was adjusted to the schema, and is defined as follows:

```
SELECT "jobAdId", COUNT(*) as c
FROM ingest
WHERE "jobAdId" IS NOT NULL AND
      DATE("timestamp") >= '<RANDOM_DATE>' AND
      DATE("timestamp") <= '<RANDOM_DATE>'
GROUP BY "jobAdId"
ORDER BY c DESC
```

Listing 1: The query

⁵Kaggle - A data science community, offering free datasets and machine learning tools

Wave 1

The test started with the first wave, inserting 10 GB of data, giving the following results:

Wave 1: The query execution times in seconds

Source: The Dash dashboard

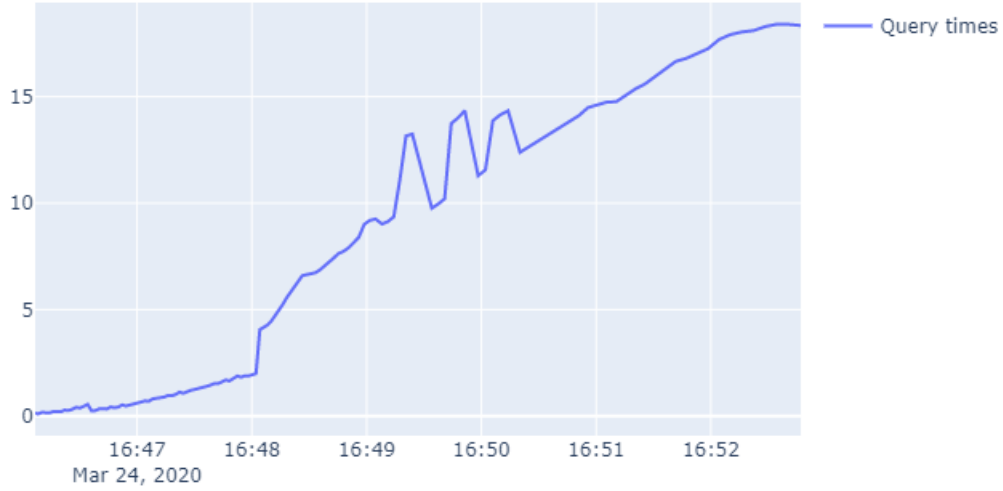


Figure 8: A linear growth of a query execution time

Wave 1: The ingestion times in seconds

Source: Calculated from the column values

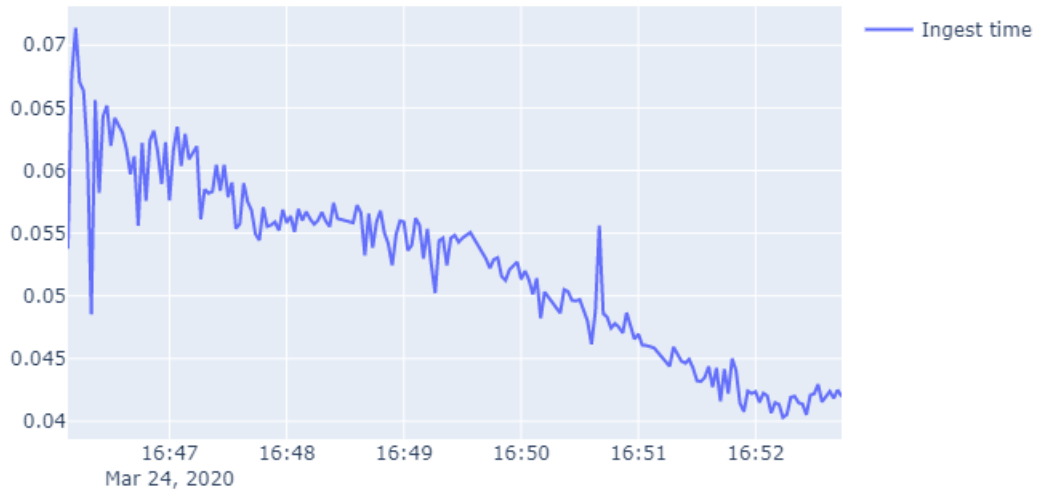


Figure 9: A linear decline of the ingestion time

Wave 1: The storage utilization in bytes

Source: GCP metric service

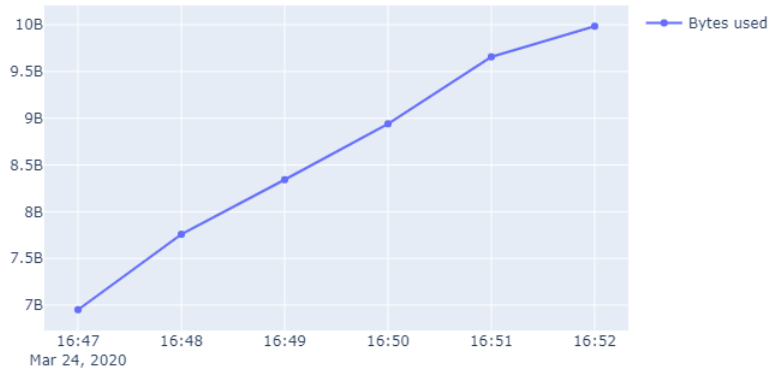


Figure 10: Total of 10 GB inserted

Wave 1: Dataflow throughput

Source: GCP metric service

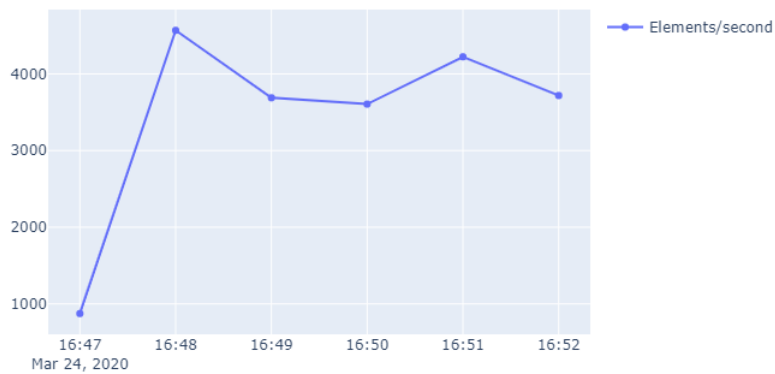


Figure 11: The number of elements inserted per second

Wave 1: All data linearly interpolated between 0 and 10

Description: Creates a better image for analyzation.

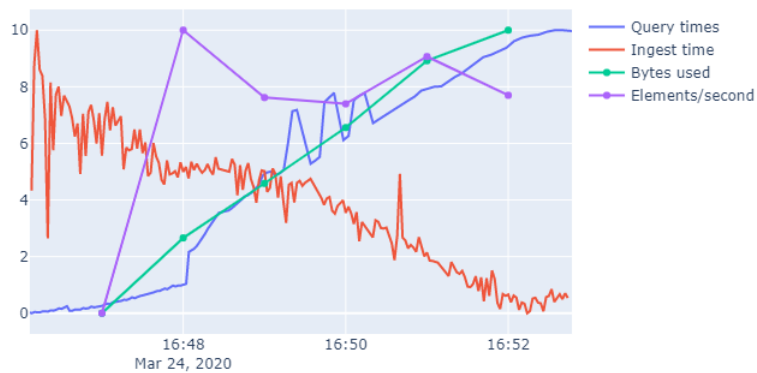


Figure 12: Query times increases when storage utilization increases

Wave 2

For wave 2, 5 additional gigabytes was ingested. This wave also includes a 10 min break in the results.

Wave 2: The query execution times in seconds

Source: The Dash dashboard



Figure 13: Postgres query execution times being unstable.

Wave 2: The ingestion times in seconds

Source: Calculated from the column values

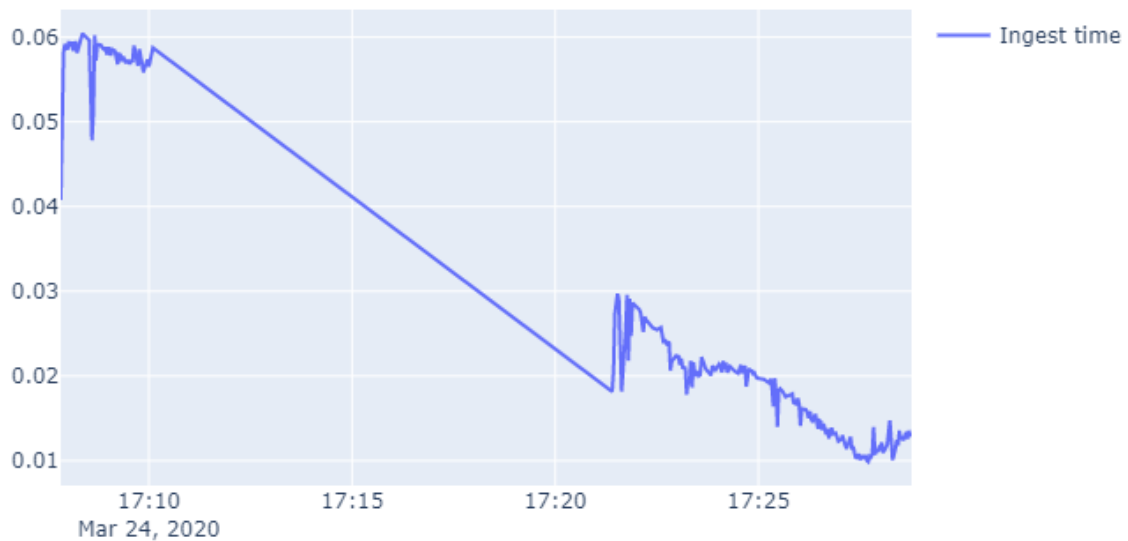


Figure 14: A decline in ingestion times

Wave 2: The storage utilization in bytes

Source: GCP metric service

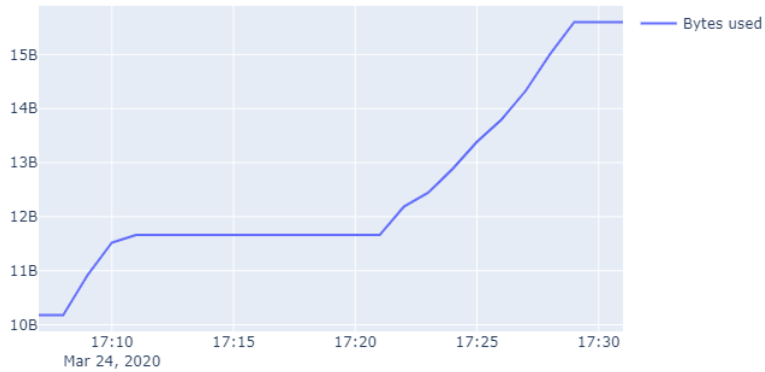


Figure 15: Total of 15 GB inserted

Wave 2: Dataflow throughput

Source: GCP metric service

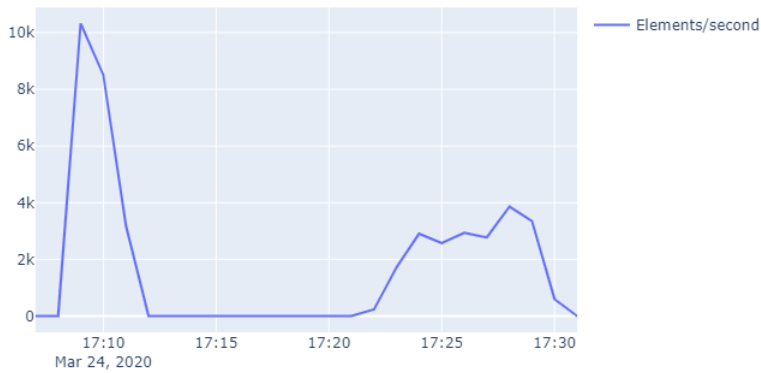


Figure 16: The number of elements inserted per second

Wave 2: All data linearly interpolated between 0 and 10

Description: Creates a better image for analyzation.

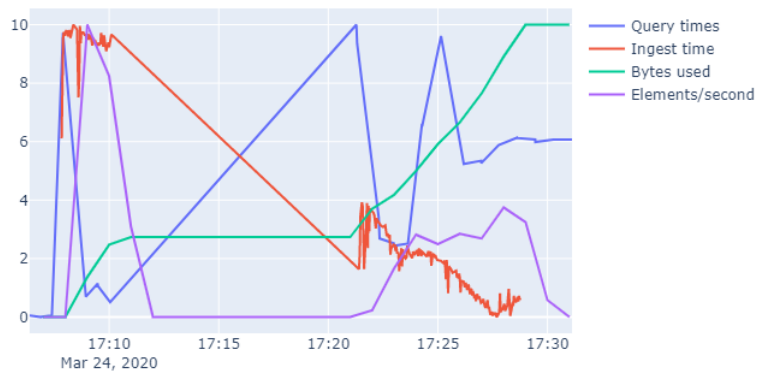


Figure 17: Query times increases when storage utilization increases

4.1.3.2 BigTable

BigTable was the second test subject in the experiment. BigTable is only publicly available as a service on GCP, resulting in the database being hosted in the cloud. From there, one was able to configure the number of nodes each cluster should consist of and to avoid being bottle-necked by hardware, the team created a BigTable instance of one cluster with nine nodes, giving an estimated cost of 4,300 USD a month. For this database, the team used the same dataset as in the Postgres tests, the one from Kaggle. Similar to the Postgres experiment, the dataset was ingested directly into Dataflow, streamed from files stored in Google Cloud Storage ⁶. The Dataflow job was identical to the previous test and consisted of an enrichment step with an external cache, and a database ingestion step for inserting into BigTable based on Apache Beam's built-in support.

Even the setup was quite similar to the Postgres-experiment, the team did one performance change to the Dataflow job's enrichment step. From the Postgres-experiment one noticed that the Dataflow machines were not utilizing their virtual CPUs to the fullest, meaning it was possible to make the enrichment step run faster. By this, the team refactored the enrichment step to work asynchronously, resulting in a huge improvement in Dataflow throughput. With this change implemented, the insertion into BigTable went extremely fast, inserting gigabytes within a minute. This resulted in ingesting way more data than planned, where the first wave ingested 95 GB, and the second wave 255 GB. Due to BigTable's ability to perform data compression, the ingested 355 GB was compressed down to 57 GB in total.

BigTable summary	
Database:	BigTable
Hosted:	Google Cloud Platform
BigTable Instance:	1 cluster of 500 GB SSD, 9 nodes
Total data ingested:	350 GB
GB ingested, first wave:	95 GB
GB ingested, second wave:	255 GB
Asynchronous enrichment:	yes
Ingested time metric:	Extracted from GCP's metric service in form of BigTable's <i>throughput</i> and <i>rows written per second</i>
Dataset:	Dataset from Kaggle
Scalable:	Yes

BigTable's table structure mainly consists of a row key and the columns defined inside the family-columns. Since the row key is the only indexed column, it needs to be designed based on the queries one will execute. Since the use case is just an experiment, the row key was simply designed as a combination of a timestamp and the id of a job posting. The rest of the fields in the dataset was inserted as into separate columns in the same family column.

When it comes to querying BigTable, one can not simply run a traditional SQL-query. BigTable is designed to run full table scans on the row key, meaning the query must be based on the row key. There is also no possibilities for running aggregations in BigTable, putting that responsibility to external processing services, like Dataflow. With this in mind, one decided to execute multiple queries, which extracts different kinds of analytical information. The following queries were defined by using a BigTable client library for Python⁷:

1. **The sample query** - Extracting a random sample rows based on a probability p . Achieved this by using the client library's *RowSampleFilter* filter, and setting the probability to 0.1%. This query fill result in a full-table scan.

⁶Google Cloud Storage (GCS) - GCP's object storage service - <https://cloud.google.com/storage>

2. **The range query** - Extracting max 10000 rows and starting at a random timestamp. This is done by using the client library's *PassAllFilter* and defining a random timestamp within the dataset's time interval.
3. **The all query** - Extracting 50000 rows in reverse lexicographical order using the *PassAllFilter* filter, resulting in the newest 50000 rows.

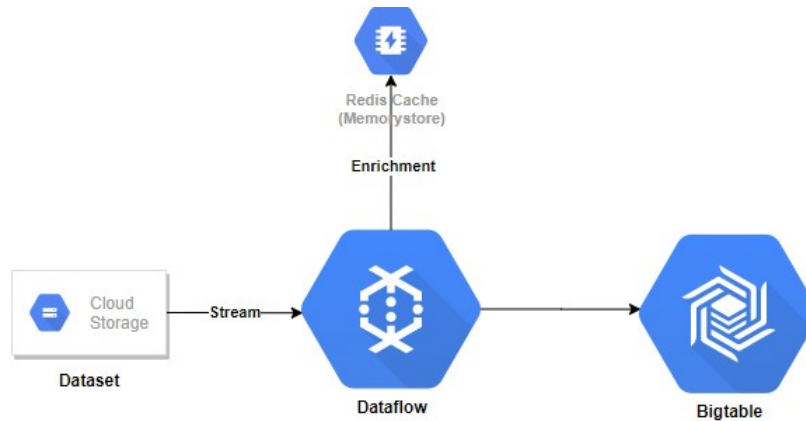


Figure 18: The pipeline for this experiment with BigTable

Wave 1

The first wave started by ingesting 95 GB of data, streamed from GCS:

Wave 1: The query execution times in seconds

Source: The Dash dashboard

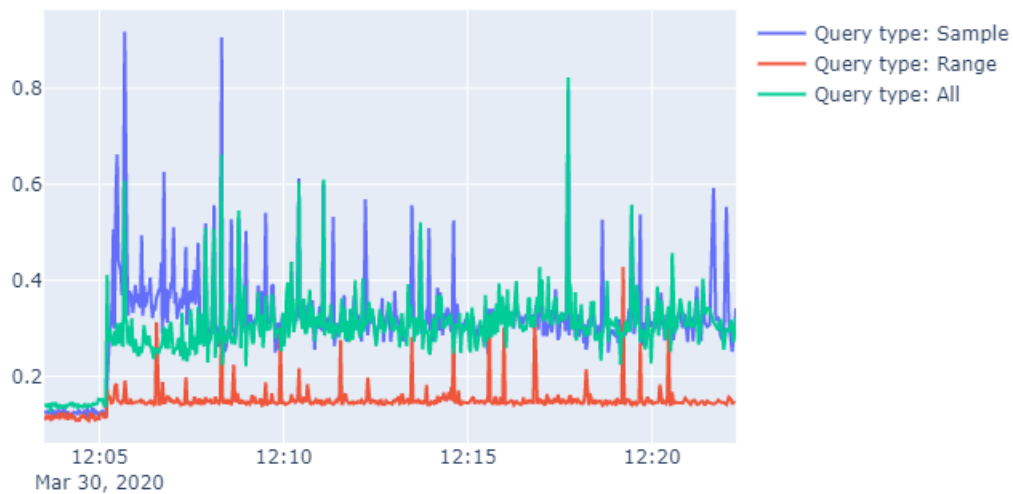


Figure 19: A linear growth of a query execution time

⁷ *google-cloud-bigtable* - BigTable client library for Python - <https://googleapis.dev/python/bigtable/latest/index.html>

Wave 1: BigTable ingestion times in bytes per second

Source: GCP's metric service

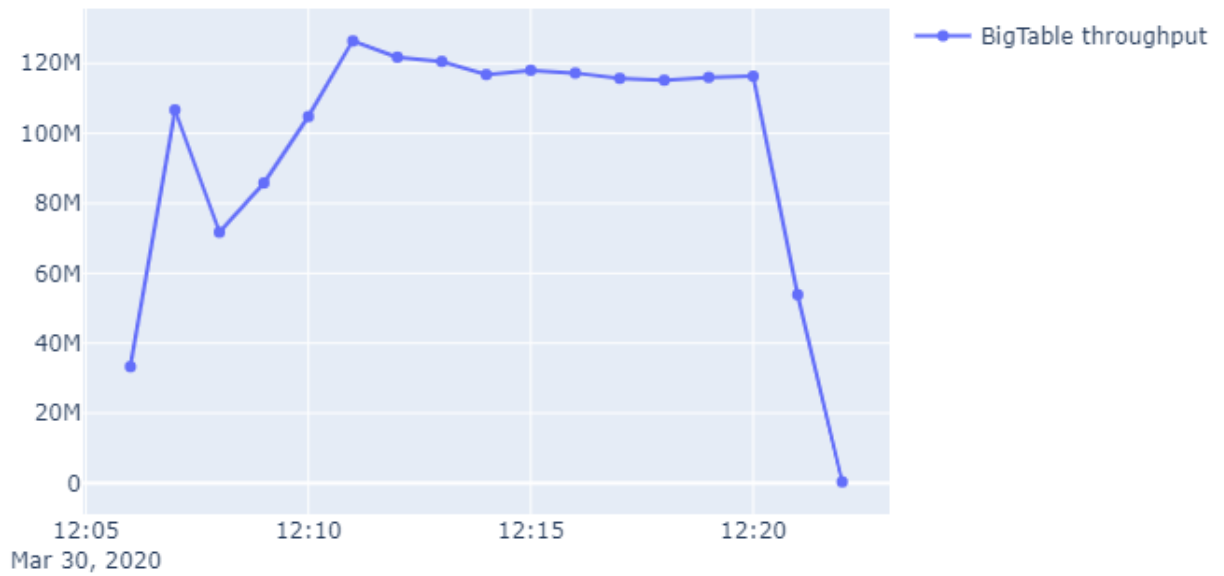


Figure 20: A high ingestion throughput for BigTable

Wave 1: BigTable - rows written per minute

Source: GCP's metric service

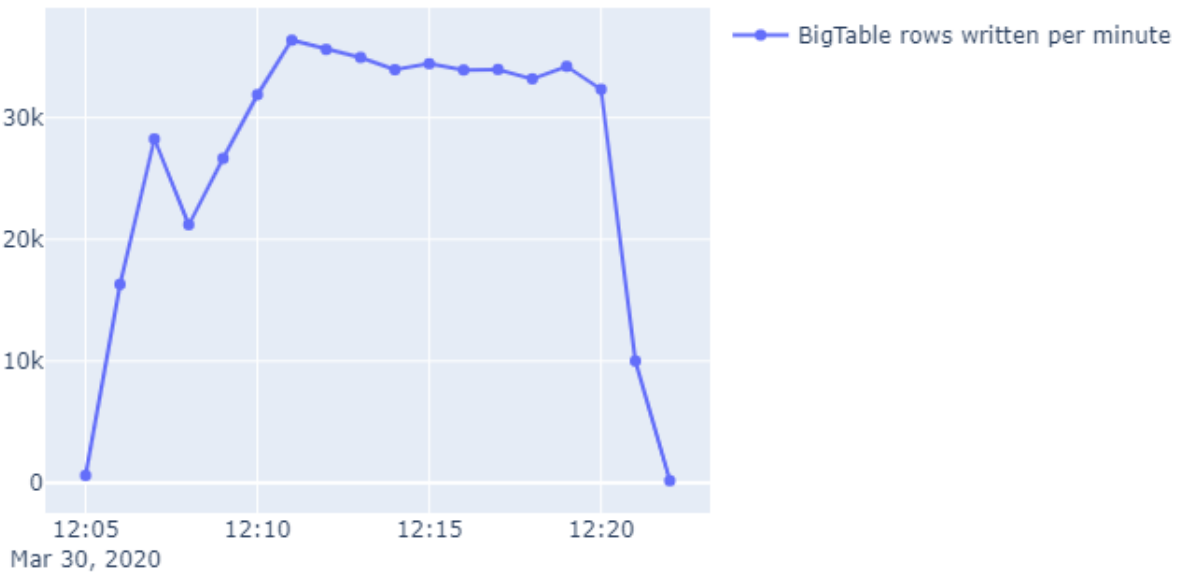


Figure 21: BigTable - wave 01 - rows written per minute

Wave 1: Dataflow throughput - produced elements per minute
Source: GCP's metric service

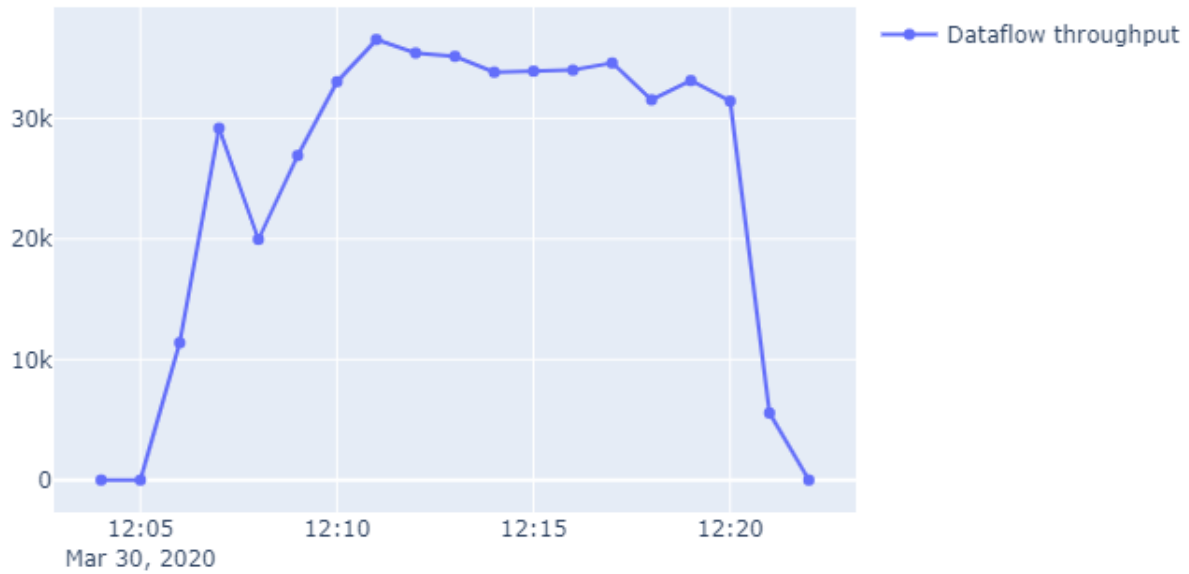


Figure 22: BigTable - Wave 01 - an incredible high Dataflow throughput

Wave 1: BigTable's storage utilization in bytes - compressed
Source: GCP's metric service

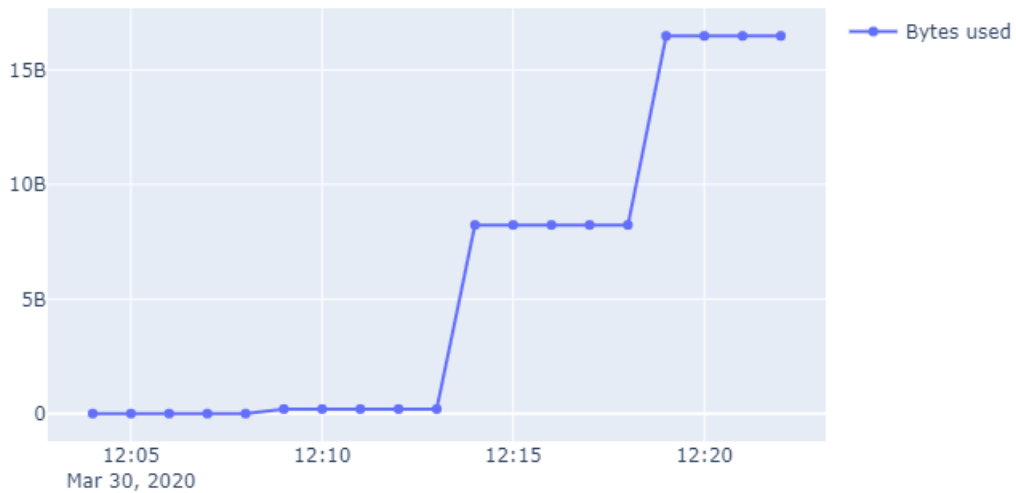


Figure 23: This metric was not updated frequently by GCP's metric service

Wave 1: BigTable's storage utilization in bytes - uncompressed
Source: Calculated using Dataflow's throughput

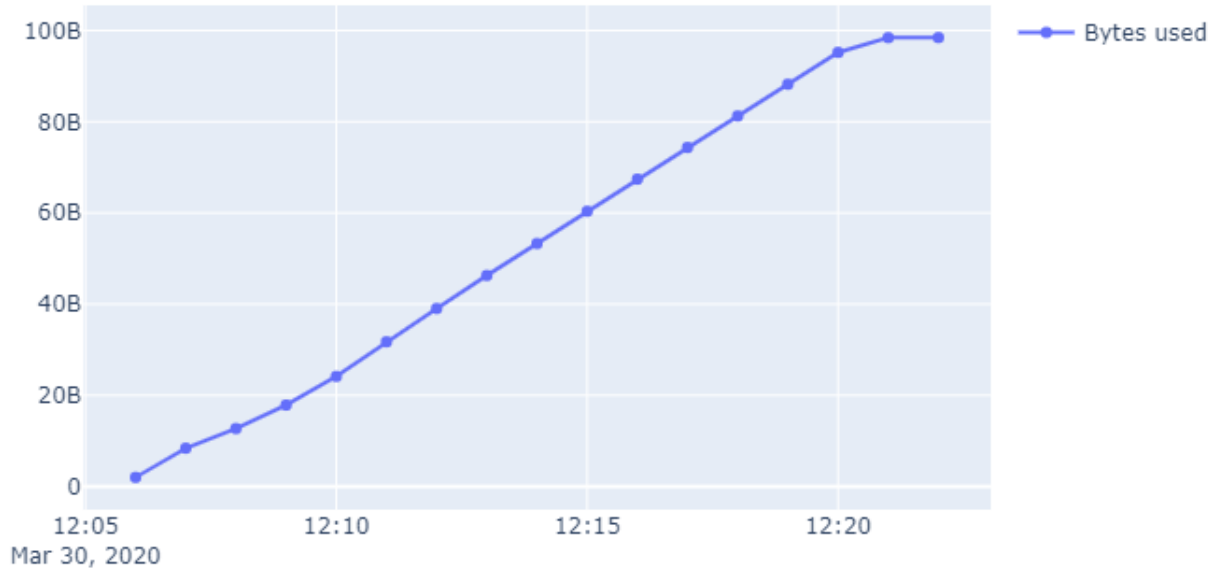


Figure 24: A linear increase of stored data up to 100GB

Wave 1: All data linearly interpolated between 0 and 30
Description: Creates a better image for analyzation

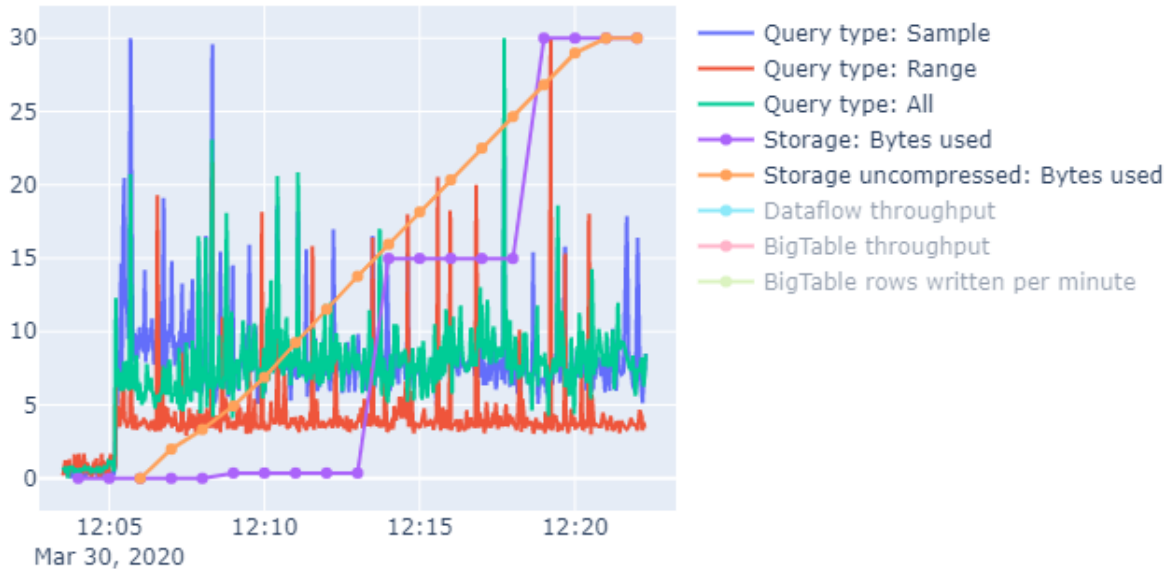


Figure 25: As storage utilization increases, there is no acknowledging change in query times

Wave 1: All throughput-data linearly interpolated between 0 and 30

Description: Creates a better image for analyzation

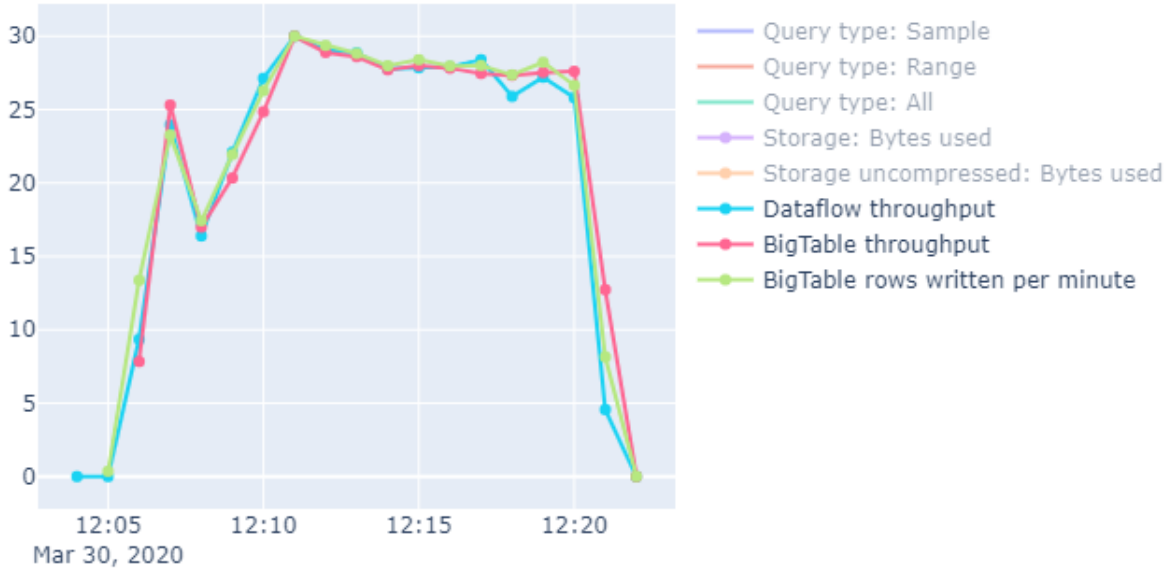


Figure 26: Dataflow's and BigTable's throughput being roughly equal

Wave 2

In the second wave, additional 255 GB was ingested into the pipeline.

Wave 2: The query execution times in seconds

Source: The Dash dashboard

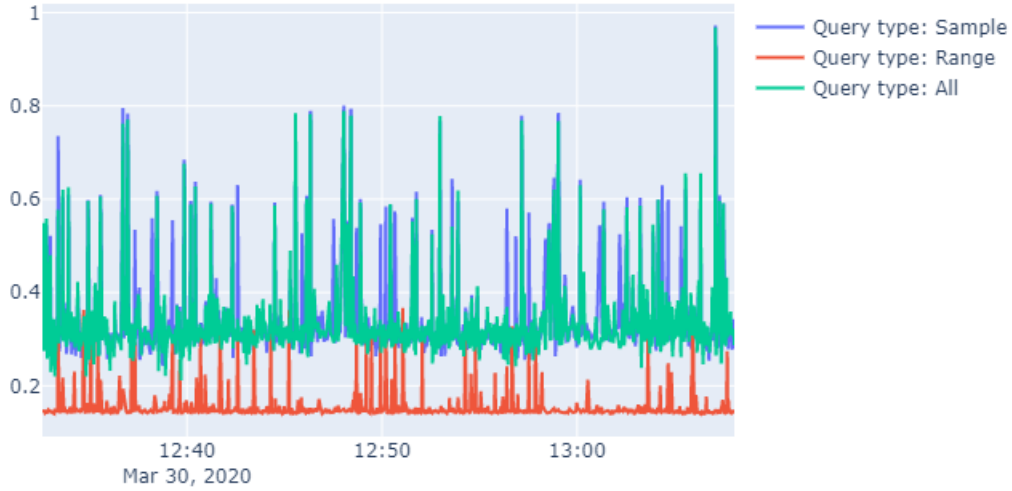


Figure 27: A linear growth of a query execution time

Wave 2: BigTable ingestion times in bytes per second

Source: GCP's metric service



Figure 28: BigTable - Wave 02 - A high ingestion throughput

Wave 2: BigTable - rows written per second

Source: GCP's metric service

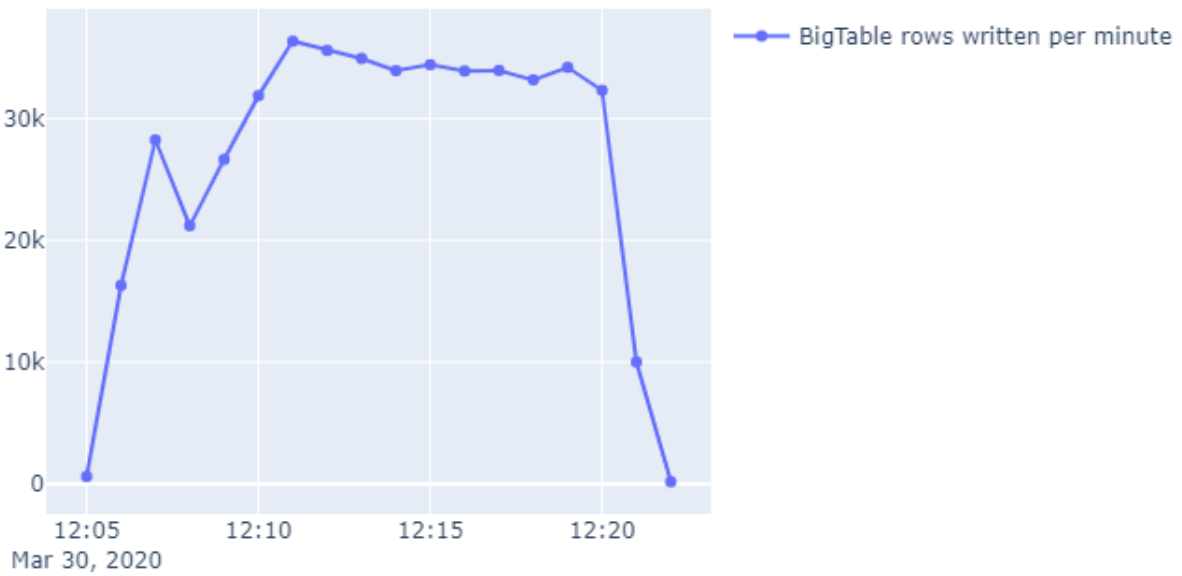


Figure 29: BigTable - Wave 02 - A high Dataflow throughput

Wave 2: Dataflow throughput - produced elements per second

Source: GCP's metric service



Wave 2: BigTable's storage utilization in bytes - uncompressed
Source: GCP's metric service

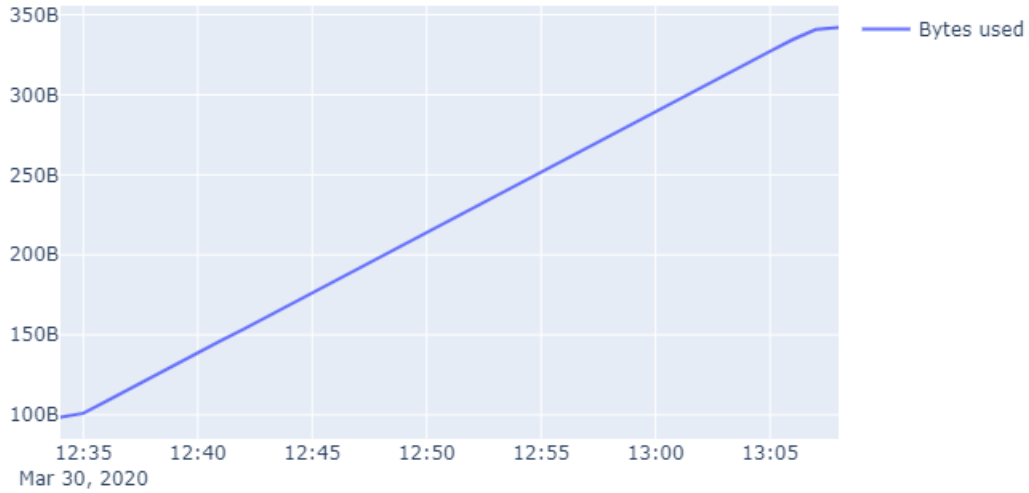


Figure 30: This metric was not updated frequently by GCP's metric service

Wave 2: BigTable's storage utilization in bytes - compressed
Source: Calculated using Dataflow's throughput

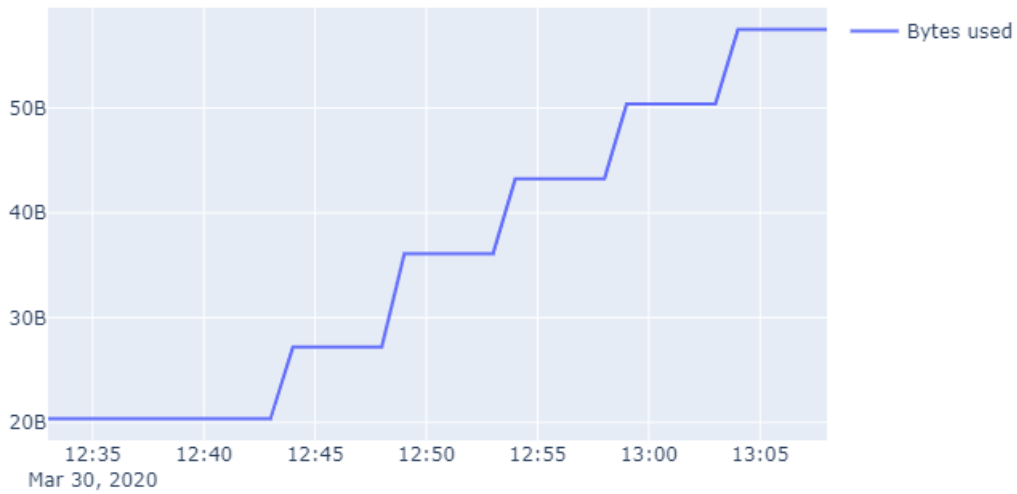


Figure 31: A linear increase of stored data up to 100GB

Wave 2: All data linearly interpolated between 0 and 30

Description: Creates a better image for analyzation

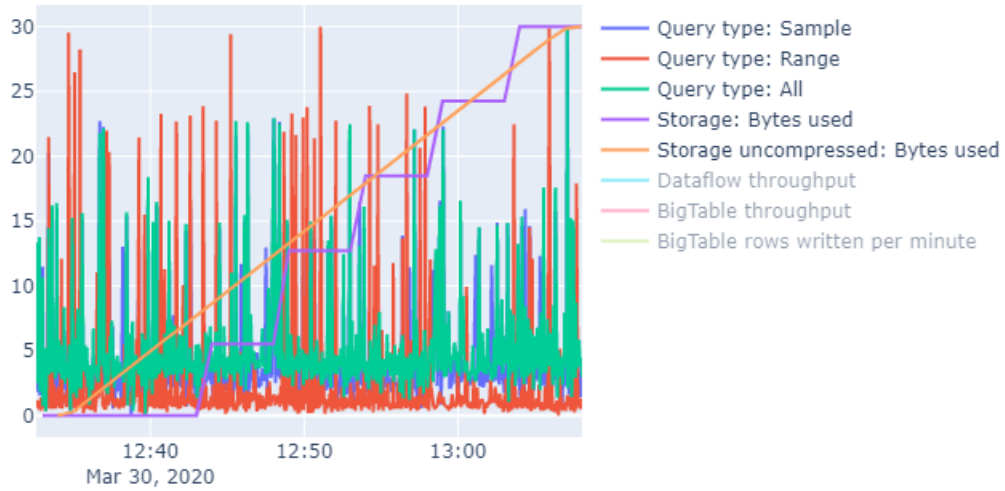


Figure 32: As storage utilization increases, there is no acknowledging change in query times

Wave 2: All throughput-data linearly interpolated between 0 and 30

Description: Creates a better image for analyzation

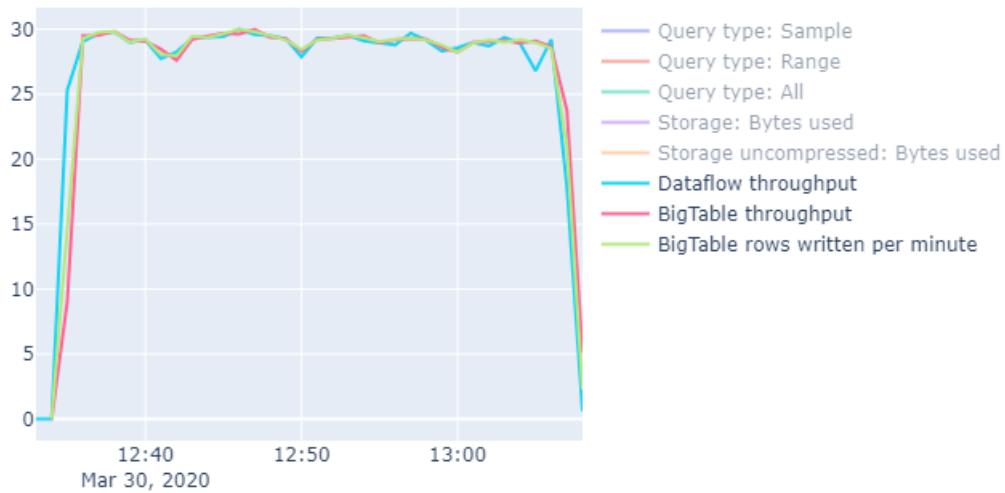


Figure 33: Dataflow's and BigTable's throughput being roughly equal

4.1.3.3 Druid

Apache Druid was the third database to be tested and is also the first database that could not be managed by GCP. This led to the team hosting Druid on a virtual machine on GCP, with 64 virtual CPUs and 240 GB of memory, to yet again avoid being bottle-necked by hardware. Druid itself consists of a cluster of different processes, making it possible to run the Druid on several machines. Due to cost constraints, it was decided to keep the database running on a single machine instance, and instead having that instance be highly rich in hardware resources. With this, Druid comes with predefined configuration sets for single machine deployments [30], making it easy to deploy Druid at different scales. The team decided to go for the *medium* configuration, thinking it would be a good fit for the experiment.

With Druid, one ended up using the same dataset from Kaggle, similar to the Postgres and BigTable experiments, but the pipeline got a little changed. There was no simple and traditional way to ingest data into Druid with Apache Beam, resulting in having to add a queue between Dataflow and Druid, *Apache Kafka*. Kafka is also an open-source Apache-licensed software like Druid and falls into the category of being a *message queue*. By using Kafka, one was able to stream data from Dataflow into Druid, but also got the problem of manually maintaining and provisioning additional software. To avoid this, the team instead decided to find a cloud-provider that was able to host, scale, and manage a Kafka queue, and by this one ended up using *Confluent*⁸. As a result, the team was able to stream data into Druid, but had to find a different way of monitoring ingestion times. With the integration of Apache Kafka, Druid had built-in support for extracting Kafka *consumer lag* [31], an indication of the time it takes for an event to be ingested into the queue and digested from the queue by Druid. The team decided to use this lag for tracking the ingestion time metric.

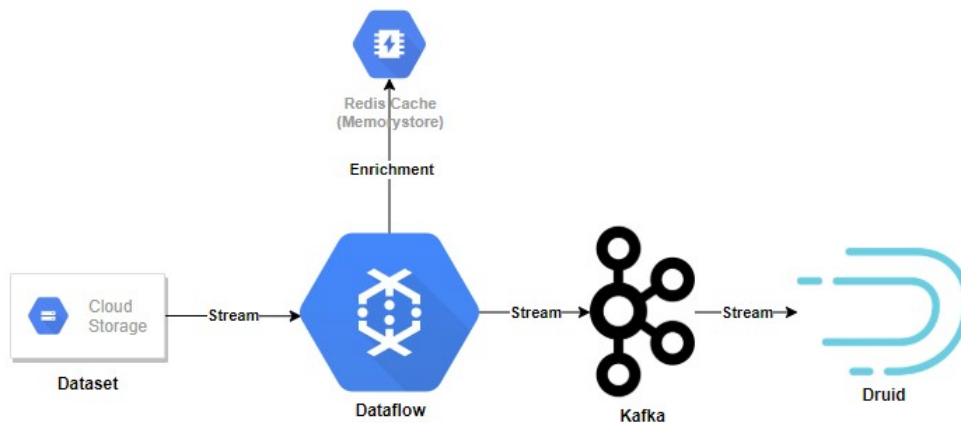


Figure 34: The pipeline for this experiment with Druid

Since Druid supports semi-structured data, one defined *dimensions* for each field in the dataset, which makes Druid open for optimizations. For monitoring query times, one used a SQL-query similar to the Postgres-experiment, which is a query that could be used in analytical contexts. The query is defined as the following:

```
SELECT jobAdId, COUNT(*) as c
FROM ingest
WHERE jobAdId IS NOT NULL AND
      DATE(__time) >= '<RANDOM_DATE>' AND
      DATE(__time) <= '<RANDOM_DATE>'
GROUP BY jobAdId
ORDER BY c DESC
```

Listing 2: The query

Druid summary	
Database:	Druid
Hosted:	Manually hosted on a virtual machine, powered by GCP.
Druid size:	Medium[30]
Total data ingested:	Unknown
Asynchronous enrichment:	yes
Ingested time metric:	Extracted Kafka consumer lag from Druid
Dataset:	Dataset from Kaggle
Scalable:	No/unknown

The entire experimentation of Druid did not go quite as planned. When starting ingesting data, everything seemed normal, but after a while one noticed a drastic increase in the query times. Eventually, the entire Druid cluster crashed due to a *Java heap - OutOfMemoryError* issue. After trying to configure Druid to allocate the running JVM more memory to its heap, one still got the same results, but with a different error. Even with the memory issue, the query times should not get affected by this, and the team noticed that the *segments* was not being published correctly. None of the segments was published to the Historicals, meaning all the data was in memory. In the end, these issues became to be way too time-consuming and ended up being unresolved.

Wave 1

Wave 1: The query execution times in seconds

Source: The Dash dashboard

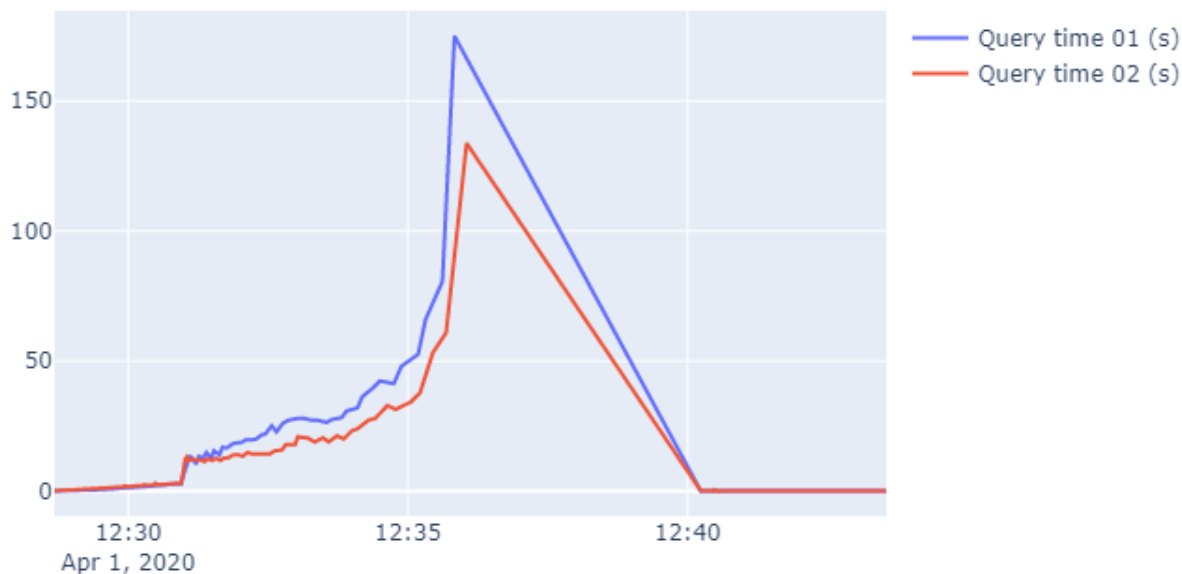


Figure 35: A sudden increase in query times, peaking at over 2.5 minutes

⁸Confluent - a company that offers fully managed cloud services for Apache Kafka - <https://www.confluent.io/>

Wave 1: The kafka producer/consumer lag in nanoseconds

Source: The Dash dashboard - extracted from Druid.

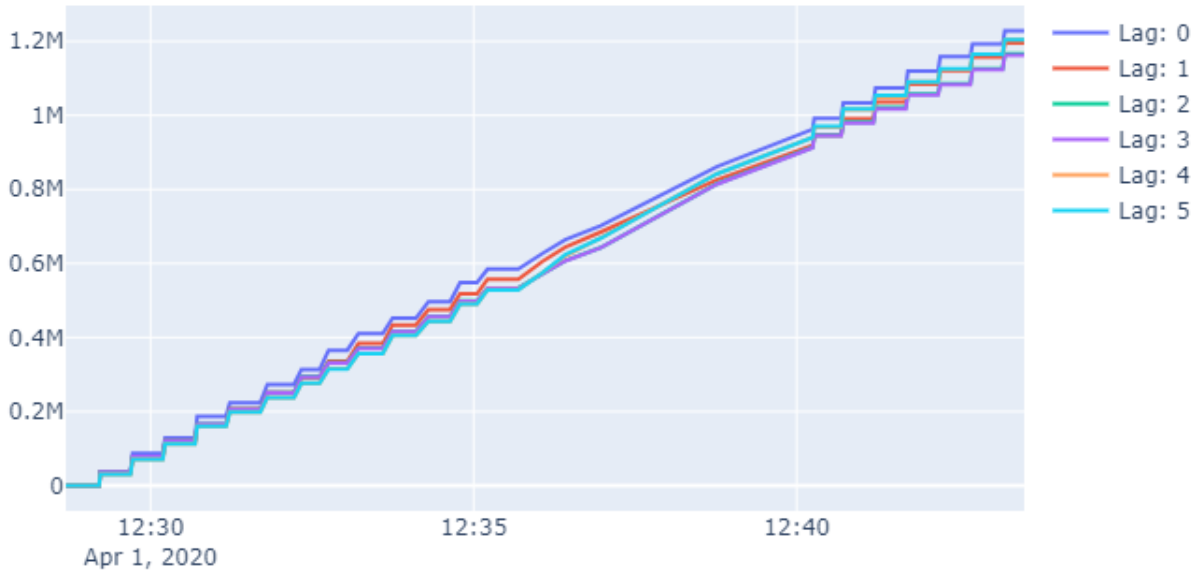


Figure 36: The consumer lag from each partition in the Kafka queue

Wave 1: Dataflow throughput - elements per second

Source: GCP's metric service.



Figure 37: Druid - Wave 01 - An unstable Dataflow throughput

Wave 1: All data linearly interpolated between 0 and 30

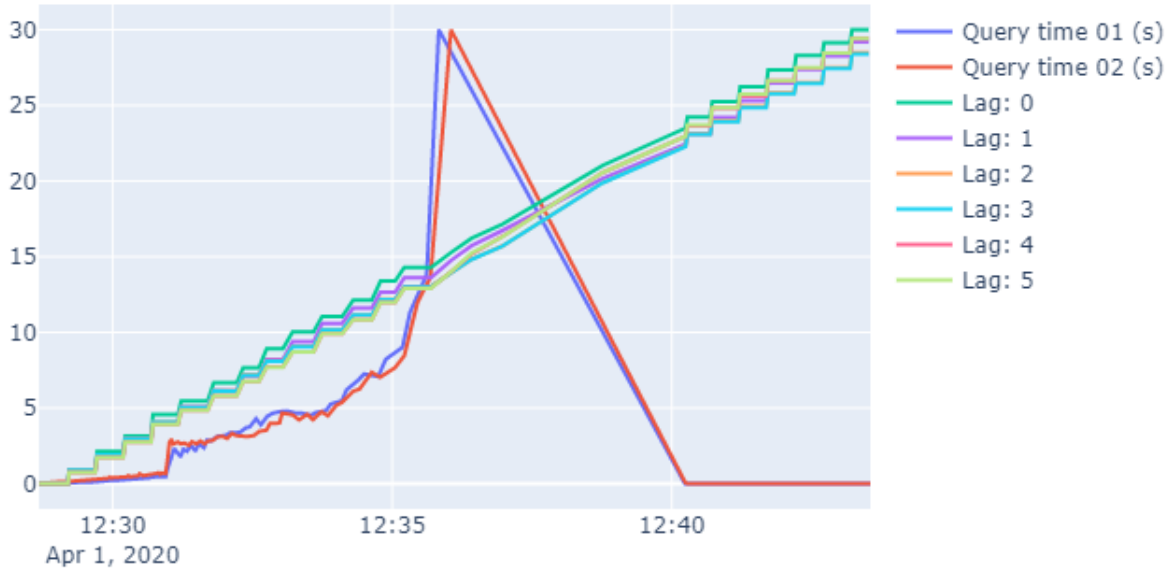


Figure 38: Druid - Wave 01 - Visualization of Druid's unscalability

Wave 2

Wave 2: The query execution times in seconds

Source: The Dash dashboard

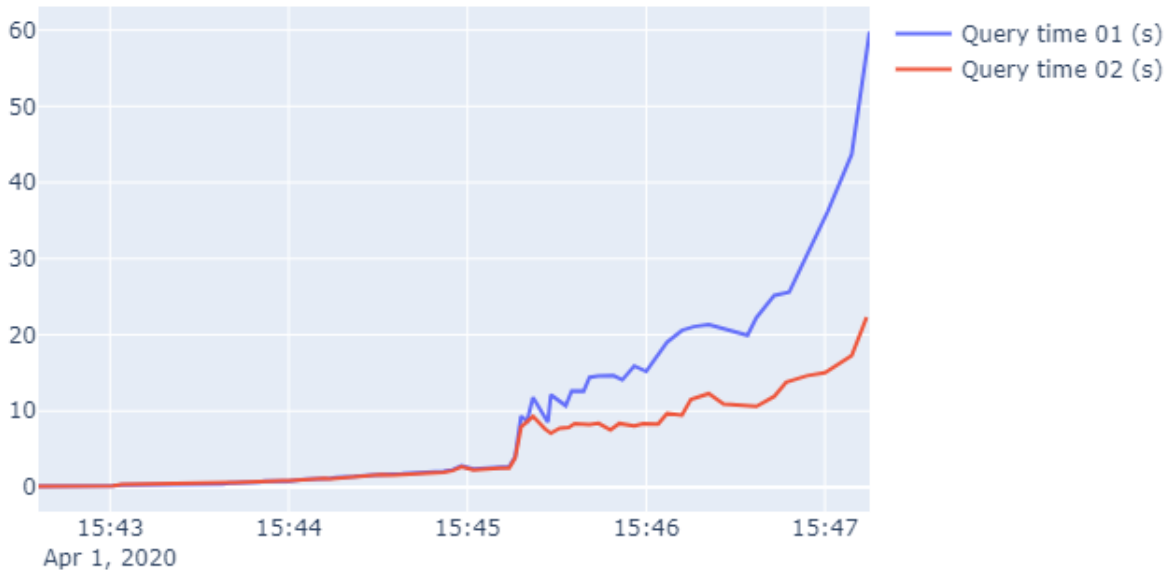


Figure 39: A sudden increase in query times, peaking at over 2.5 minutes

Wave 2: The kafka producer/consumer lag in nanoseconds

Source: The Dash dashboard - extracted from Druid.

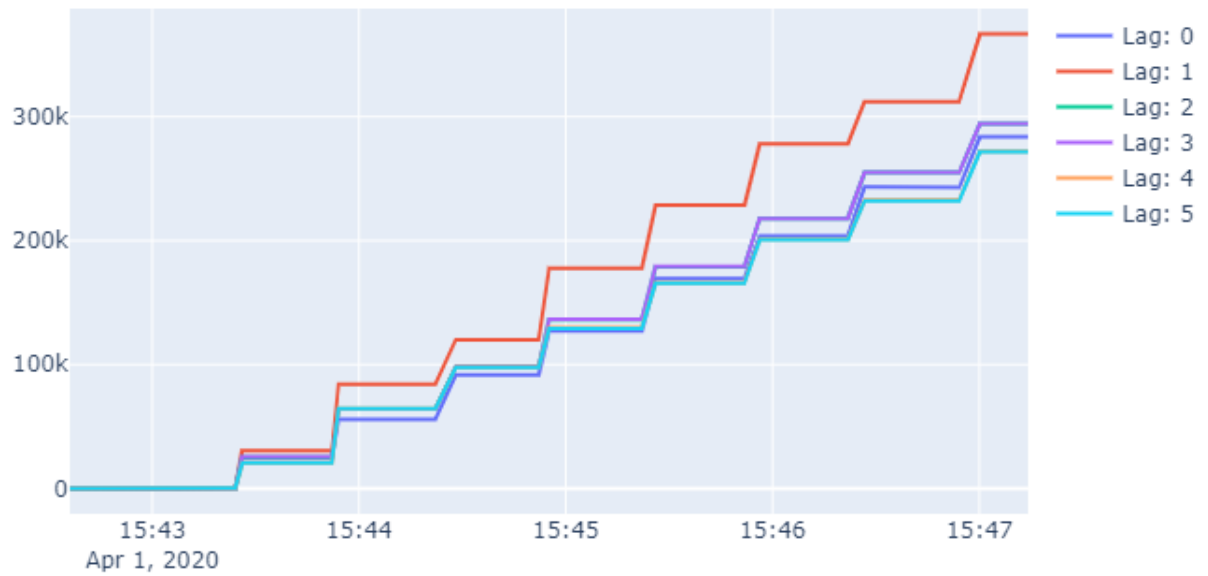


Figure 40: The consumer lag from each partition in the Kafka queue

Wave 2: Dataflow throughput - elements per second

Source: GCP's metric service.

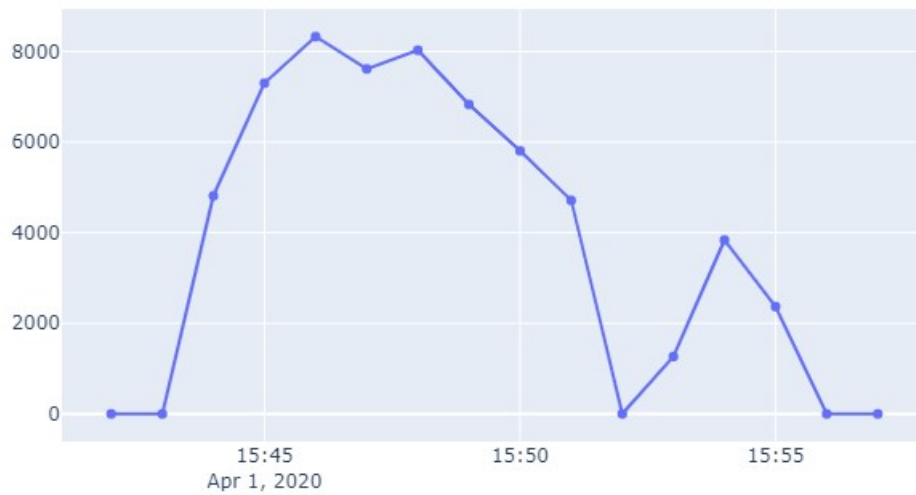


Figure 41: Druid - Wave 02 - Dataflow's throughput suddenly going down to 0.

Wave 2: All data linearly interpolated between 0 and 30

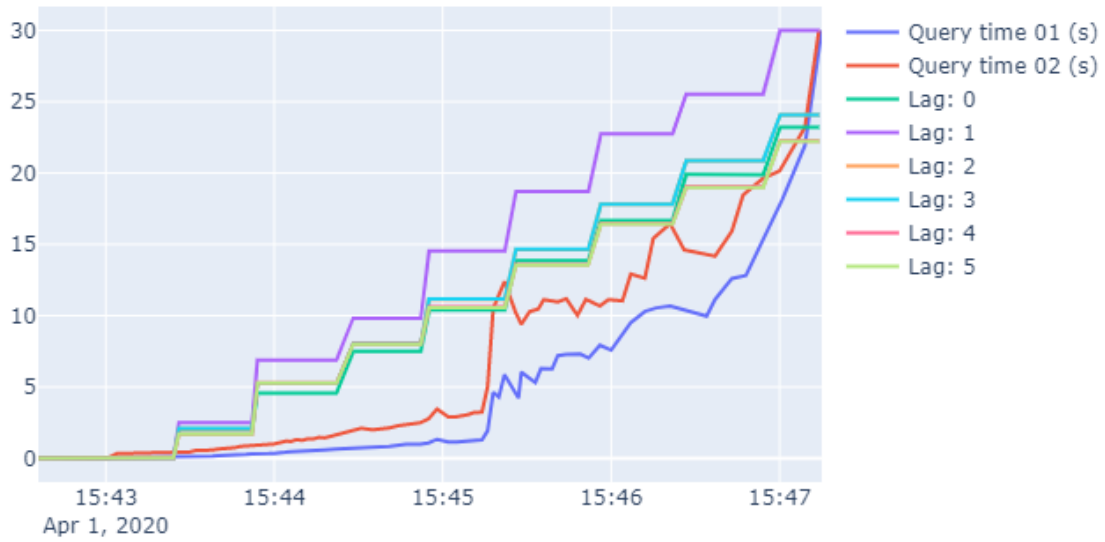


Figure 42: Druid - Wave 02 - An overall increase in ingestion and query times

Wave 3

Wave 3: The query execution times in seconds

Source: The Dash dashboard

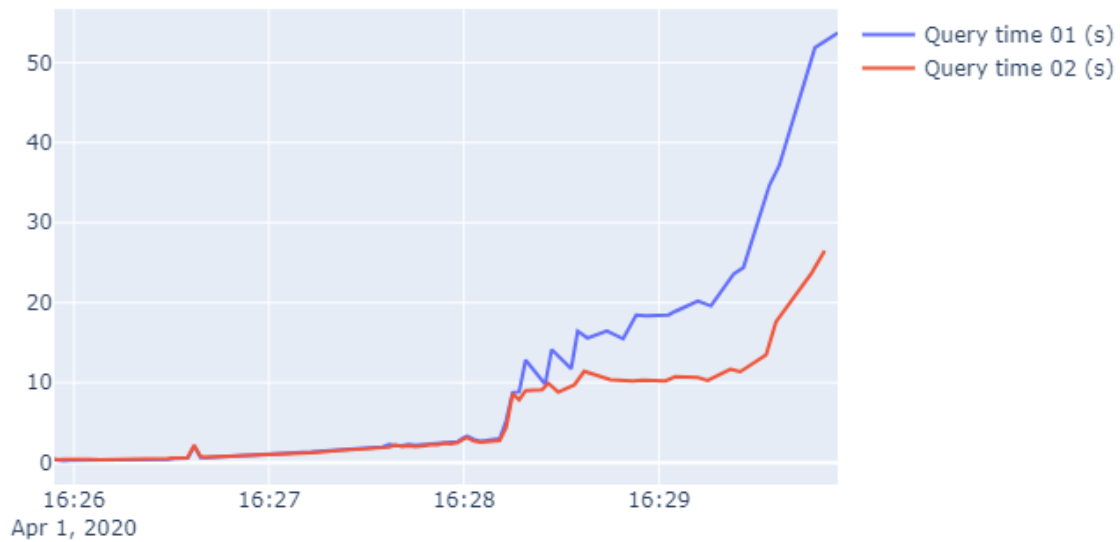


Figure 43: A sudden increase in query times, peaking at over 2.5 minutes

Wave 3: The kafka producer/consumer lag in nanoseconds

Source: The Dash dashboard - extracted from Druid.

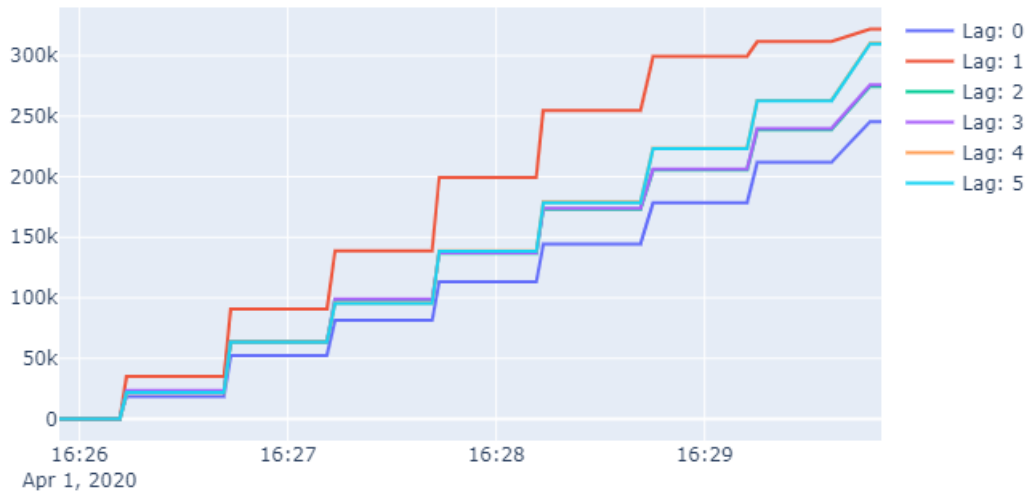


Figure 44: The consumer lag from each partition in the Kafka queue

Wave 3: Dataflow throughput - elements per second

Source: GCP's metric service.



Figure 45: Druid - Wave 03 - A gradually decrease in Dataflow throughput

Wave 3: All data linearly interpolated between 0 and 30

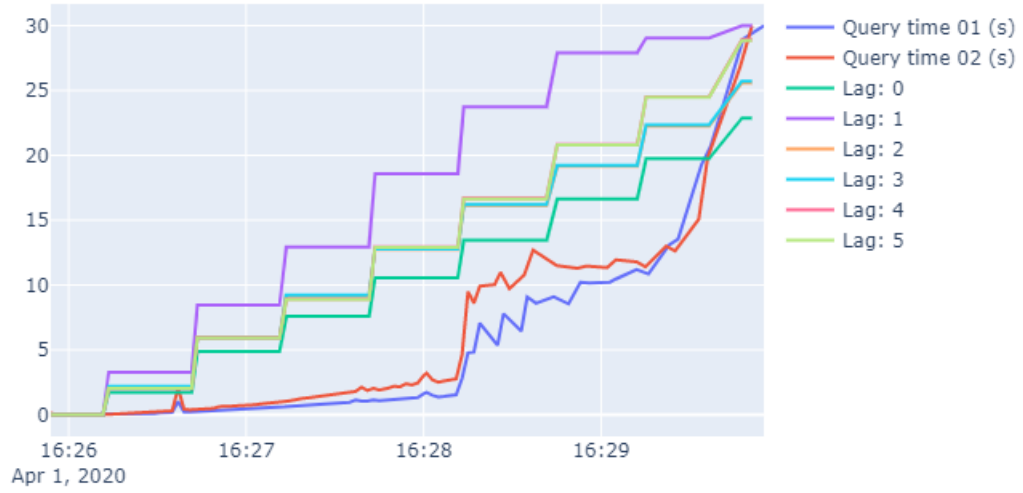


Figure 46: Druid - Wave 03 - An overall increase in consumer lag and query times

4.1.3.4 BigQuery

BigQuery was the last database to be experimented on, which is, similar to BigTable, a service only available through the GCP, fully managed by Google. When the team approached experimenting with BigQuery, one started to create a new dataset, identical to the event data Highered is currently storing. This corresponds to having events like *job posting views* and *application clicks*. The purpose was to set up and configure BigQuery to adapt to Highered’s needs and demands, and as a result, the experiment involved generating a dataset of 4 million events. These events were inserted into Cloud Pub/Sub as the starting point, and then ingested into Dataflow when the experiment started. The Dataflow job included an enrichment step, enriching data like job posting- and company-data, resulting in a bigger data enlargement than before. During the experiment the dataset only included the one type of event, which was ingested into a time-partitioned BigQuery-table from Dataflow.

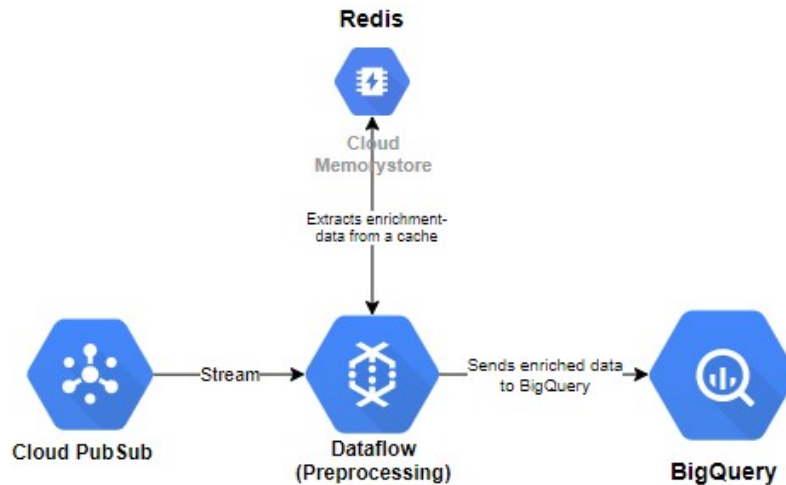


Figure 47: The pipeline for this experiment with BigQuery

BigQuery summary	
Database:	BigQuery
Hosted:	Hosted by GCP.
Total data ingested:	6GB
Asynchronous enrichment:	No
Ingested time metric:	Extracted as <i>rows written</i> from GCP’s metric service
Dataset:	Custom generated - adapted to Highered’s needs
Scalable:	Yes

After experimenting with an asynchronous enrichment step from the previous experiments, the team decided to go back to synchronous processing for the BigQuery-pipeline, due to some restrictions and difficulties with the new demands of the pipeline implementation. This resulted in a slower enrichment step, reducing Dataflow’s throughput. For the sake of the experiment, it is more interesting to see how BigQuery performs with big amounts of stored data, instead of observing how much data it is capable of ingesting simultaneously. With this said, it ended up being more time-consuming and costly to ingest the data into BigQuery and, therefore, had to wrap the experiment up earlier than planned. This led to a total of 6GB being inserted into BigQuery in only one wave, which was compressed down to 2.3 GB inside the data warehouse.

Since BigQuery is capable of storing structured data, the team defined a schema that corresponds to the fields in the generated dataset, in addition to partitioning the BigQuery-table based on a custom timestamp-column. With storing structured data, BigQuery also has support for SQL-like queries, but due to different datasets, the chosen query for this experiment is a little bit different from the ones used in the Postgres- and Druid-experiment. The query is a subquery adapted to an actual analytical insight needed by Highered, and is defined as follows:

```

SELECT
  jobad_id as JobAdId,
  jobad_title as Title,
  jobad_company_name as Location,
  jobad_publish_date as Published,
  jobad_publish_end_date as Deadline,
  COUNT(jobad_id) as Views
FROM `highered.analytics_dev.JOBAD_VIEWS`
WHERE
  DATE(timestamp) >= "RANDOM_DATE" AND
  DATE(timestamp) >= "RANDOM_DATE" AND
  jobad_company_id = 2500
GROUP BY
  jobad_id,
  jobad_title,
  jobad_company_name,
  jobad_publish_date,
  jobad_publish_end_date

```

Listing 3: The query

The results of the first and final wave

The query execution times in seconds

Source: The Dash dashboard

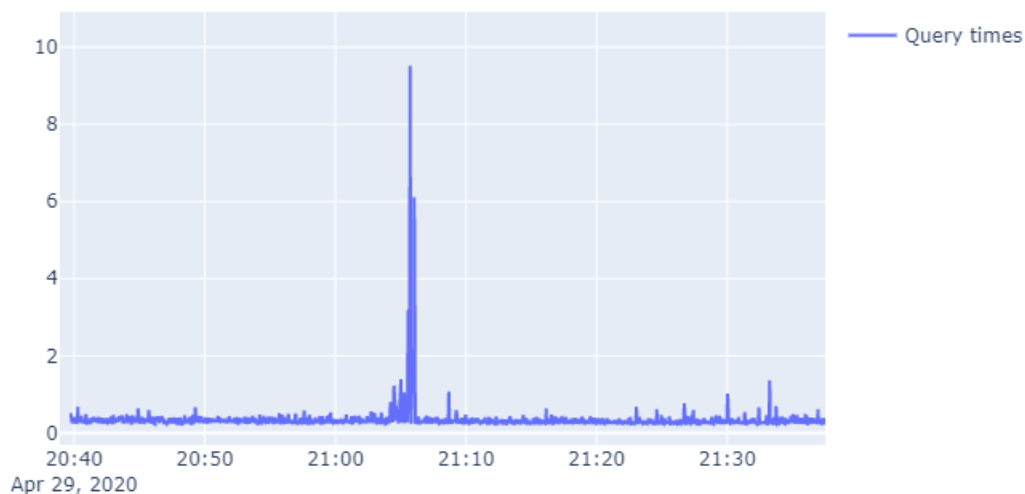


Figure 48: No noticeable change in query times, except for a unexpected spike.

BigQuery - The count of rows written per second

Source: GCP's metric service

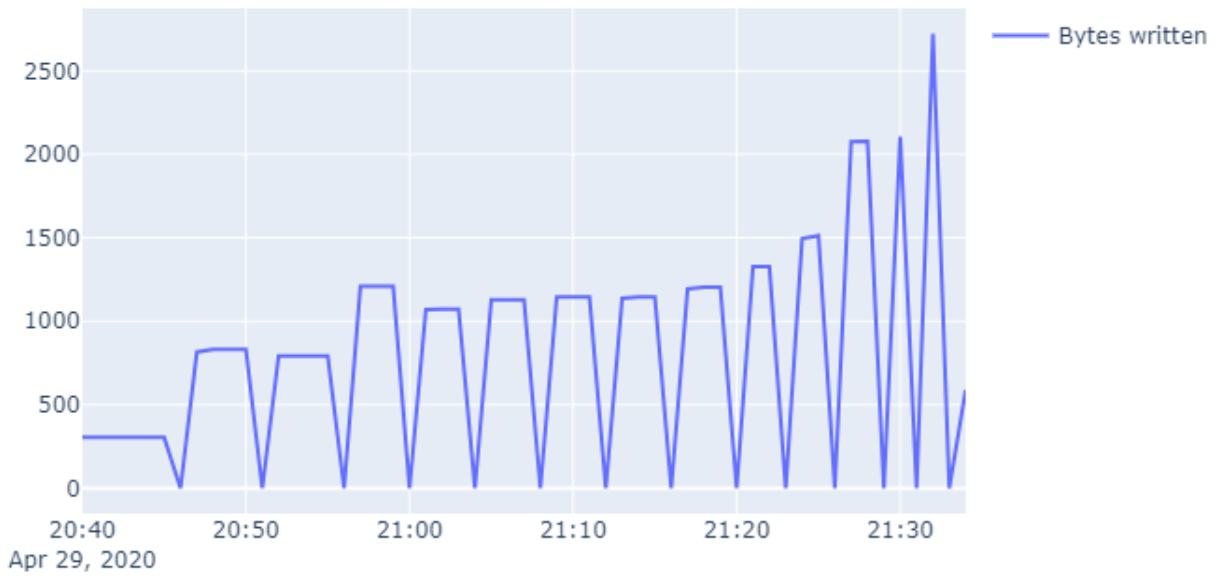


Figure 49: The number of rows written to BigQuery per second.

Dataflow throughput - count of outputted elements per second

Source: GCP's metric service

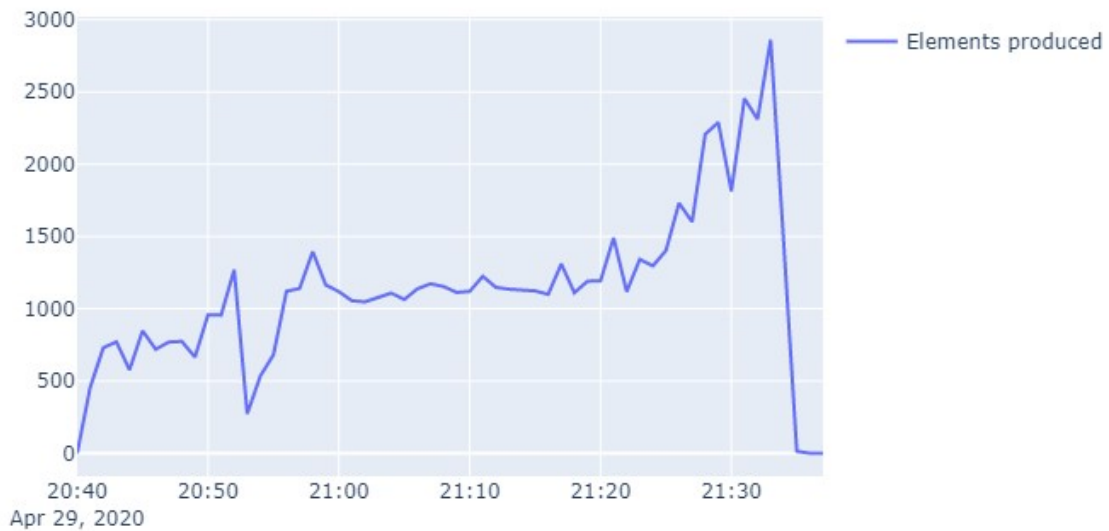


Figure 50: A gradual increase in throughput

BigQuery - uncompressed storage utilization in bytes
Source: GCP's metric service

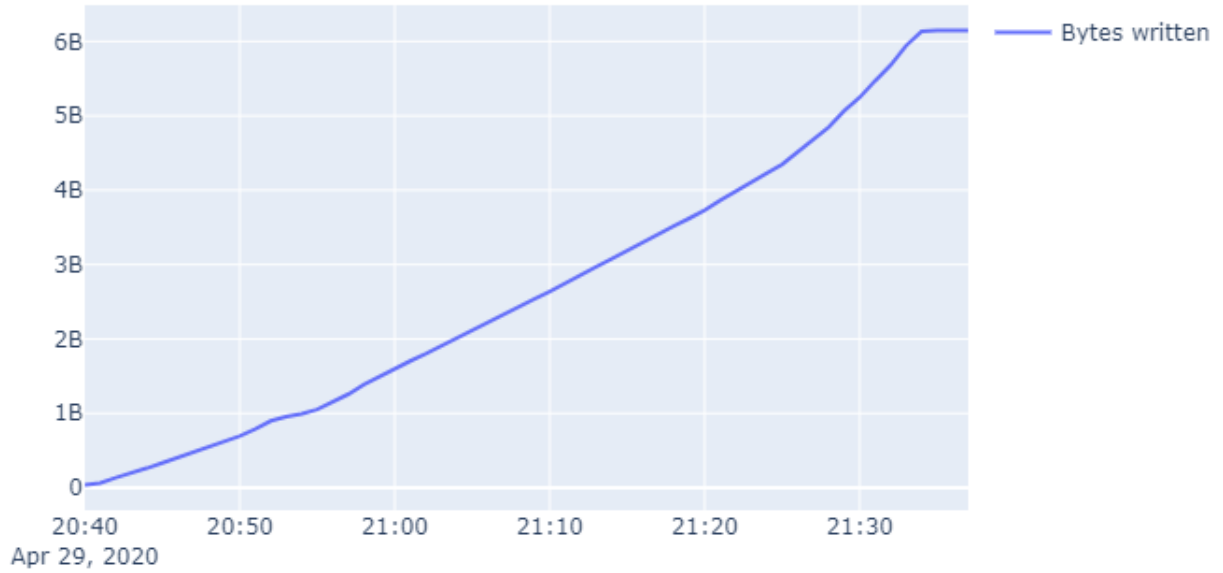


Figure 51: BigQuery - the number of ingested bytes utilized

All data interpolated between 0 and 30

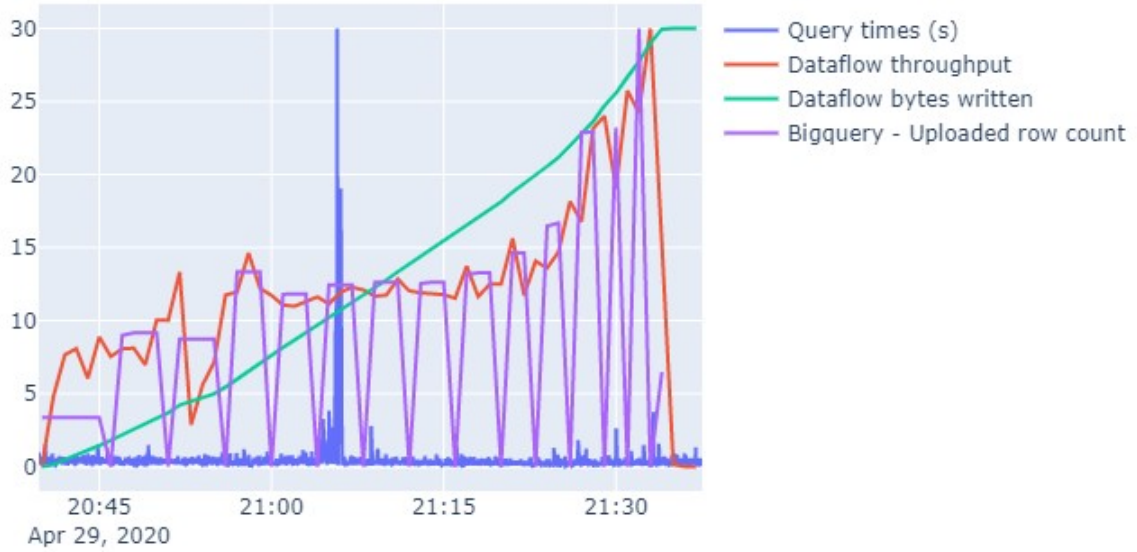


Figure 52: BigQuery - unaffected query times when workload increases

4.2 Engineering professional results

This chapter will describe how the team achieved the various goals described in the vision document by going through each one and commenting on the relevant solutions that the final solution provides.

4.2.1 Final result

The main goal of this project was to build a new analytical solution for Highered according to the requirements and the new demands and should be a solution the company can later build further upon. After running the experiments and getting a better understanding, knowledge, and experience in developing analytical pipelines, the team ended up building the following solution as visualized below:

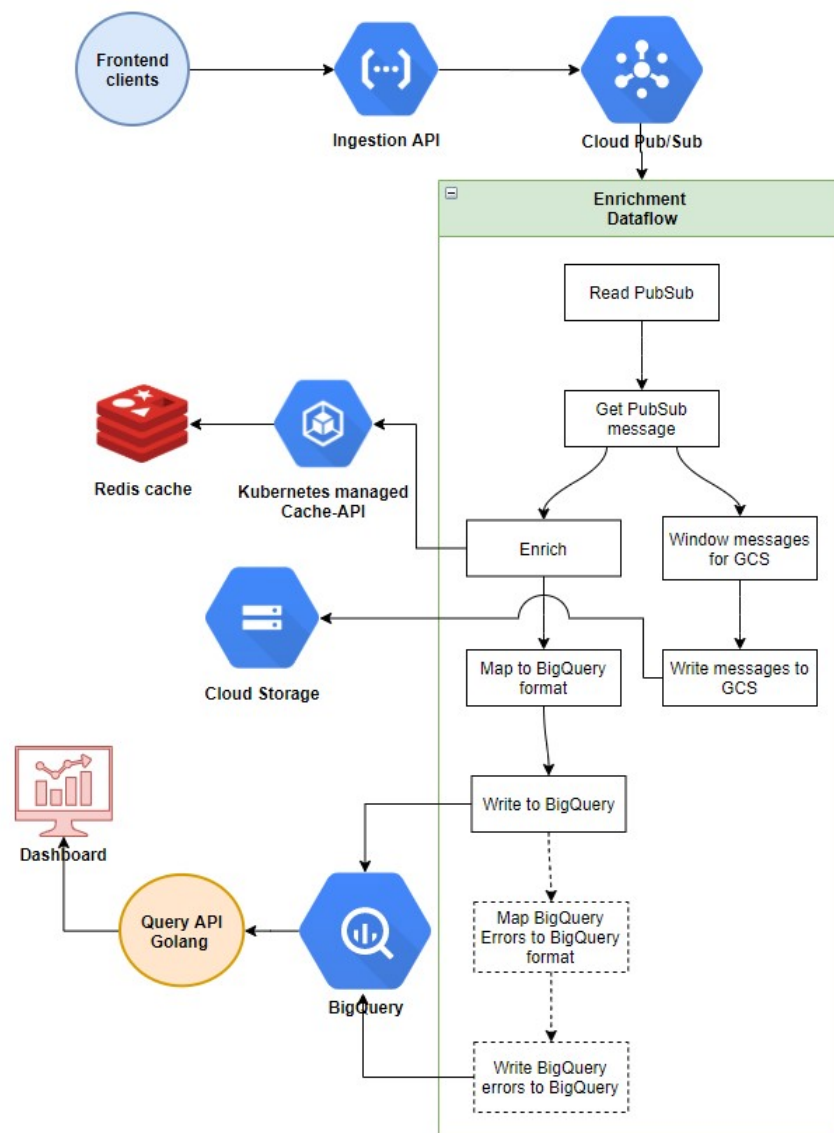


Figure 53: A diagram of the final pipeline result

This final solution builds upon using a variety of GCP services, keeping the entire system inside Highered's existing cloud environment. The pipeline starts by receiving events from Highered's student platforms into an *Ingestion API*, which is simply a Cloud Function that automatically scales horizontally based

on the load. From there, the events get buffered into Cloud Pub/Sub, waiting for getting ingested into Dataflow for further processing. The running Dataflow job consists of two different branches, where one branch ingests into BigQuery, and the other branch windows the raw events into Google Cloud Storage for backup purposes. From there, the data is available in BigQuery, which can be accessed from a *Query-API* written in Go for powering various kinds of dashboards.

It was also decided to store the data denormalized in BigQuery for performance reasons. With this in mind, the Dataflow job includes an enrichment step, enriching the raw events with job posting and organization data, achieving denormalization before the events get ingested into the data warehouse. The enrichment step is based on fetching data from external sources, which is done by using an API with a built-in Redis-cache, hosted and autoscaled by an existing Kubernetes⁹ cluster. The cache is deployed using Memystore, providing high availability and possibilities to scale with a few easy steps. The enrichment step is the biggest bottleneck in the final pipeline implementation, but is under normal circumstances able to fetch the external data within 20-60 milliseconds on average, as shown from GCP's metric service below:



Figure 54: The mean enrichment execution times in milliseconds for an event, from all running tests

In addition to this, the team also decided to make a simple MVP of a dashboard, visualizing the pipeline's produced results. This dashboard ended up being developed by using the mentioned Query API for extracting data from BigQuery, and web technologies like Vue¹⁰ and Google Charts¹¹ for building the dashboard's graphical interface. The purpose of the dashboard is to prove that the pipeline can power a dashboard similar to Highered's existing solution, showcasing the same types of data. The team, therefore, built the dashboard with Highered's existing dashboard in mind, and ended up producing the dashboard visualized below:

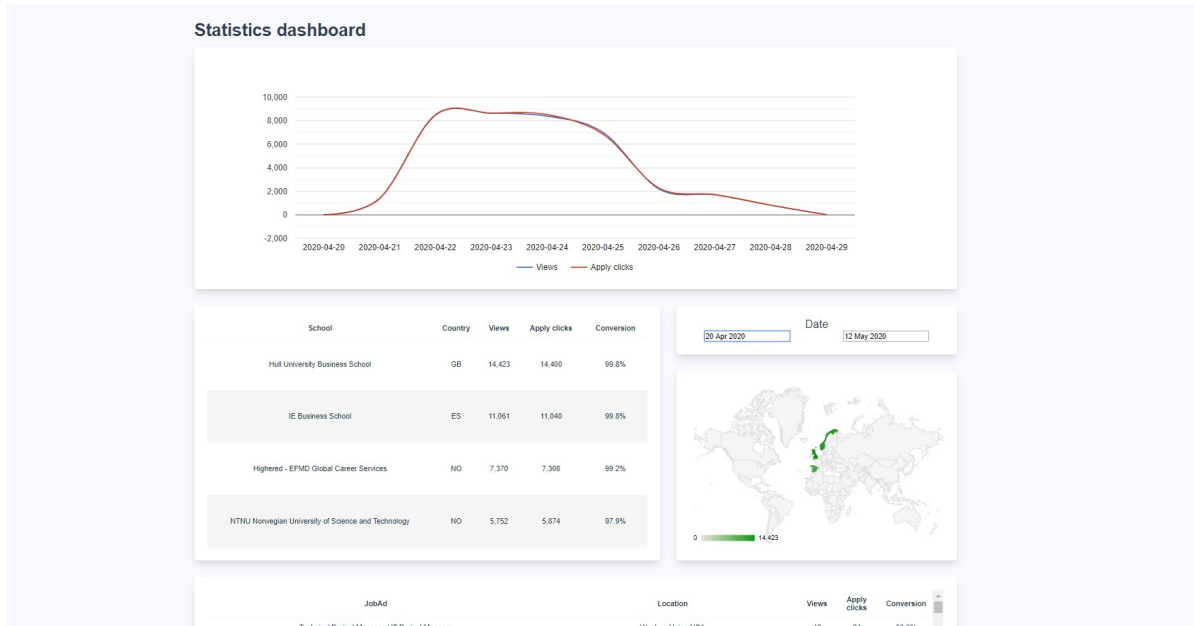


Figure 55: A MVP dashboard, visualizing fake events from the stored BigQuery data

4.2.2 Functional requirements

1. The system should be able to insert an event through an API

The system needs to be able to deal with event data from several front-end clients and thus needs a solid and reliable way of ingesting these events into the analytics pipeline that deals with the data. This ingestion needs to be easily accessible from Javascript running on the front-end clients, and should not be overly complicated.

The team implemented a Cloud Function using Golang instead on a traditional API, this allows the team to not have to worry about uptime or availability for this part of the streaming pipeline, as cloud functions can be hosted in several locations all over the globe (though currently it is only hosted on the same server region as the rest of the pipeline). Cloud functions also have the ability to scale more or less infinitely.

2. The system should be able to handle 1 000 requests per second

The system should be fast enough to both handle the load from the current analytics system experiences at the time of writing but also have lots of headroom for expanding into collecting new types of event data without bogging down.

Under the latest stress test the final pipeline produced a throughput graph on the enrichment stage that clearly shows the pipeline's capability to handle quite a bit more than the goal of 1 000 requests per second.

¹¹Kubernetes - an container orchestration, for deploying, scaling, and managing containerized applications - <https://kubernetes.io/>

¹¹Vue - a javascript web-framework for building progressing web applications - <https://vuejs.org/>

¹¹Google Charts - a Javascript library for creating interactive charts - <https://developers.google.com/chart>



Figure 56: A graph over the average throughput for the enrichment stage under the latest stress test

3. The system should never lose one event

All of the components in the system that are critical to data management, the Cloud Function API, the Pub/Sub queue, the data enricher Dataflow and the database BigQuery all support downtime-less updating and all of them are both managed and hosted by Google, who in turn promises 99.99% uptime. All this put together, plus some simple retry on failure logic for the client UI, means the system can fully guarantee within reason that all events should be handled. The team ran a week-long test of the final streaming pipeline to make sure it could run without crashing or any other bugs, during the test an event simulator was running.

4. The system should be able to distribute events in different types

Highered wants to be able to track different kinds of behavior from their users, meaning storing events of different types, like *job posting views* or *application clicks*.

With this in mind, the team defined a set of event types according to Highered's needs, and made the raw events come with one of the types, describing the action of the event. From there, the final pipeline puts each event type into its own table in BigQuery, distributing the events by their type. Events with types that are not defined by the system are put into a dedicated miscellaneous table, making new and unknown event types effortlessly available from BigQuery for further insight. In addition to this, should there occur an error while processing an event in Dataflow, the system will then ingest the event into a separate error-table, containing all failed events for further investigation. As a result, the system is capable of distributing events by type.

5. The system should be open for "real-time" support

Highered would like to have a system where one could for example know how many users are currently watching a certain job position in real-time. By this, the system would have to be open for serving data within tens of seconds after the occurrence of the event. Since the team has implemented a streaming pipeline that continuously handles the event data, instead of handling it in batches, under normal circumstances the events should be inserted into the database within a minute from being sent from the client.

To ensure this is true, one can take a look at the data inside BigQuery, which contains both the timestamp of occurrence from the client and the timestamp of ingestion from BigQuery. One can for example run a query that calculates the differences between the ingestion time and occurrence time, and then calculate the mean of the differences for each day. This result would give an indication of how fast an event is being available in the database for real-time querying after it's occurrence. Since the team had the implemented pipeline running for a few days while ingesting fake events, one can find the answer to this by running the following query:

```

WITH diff_query AS (
  SELECT
    DATE(timestamp) as day,
    TIMESTAMP_DIFF(
      ingestion_time,
      timestamp,
      MILLISECOND
    ) as diff
  FROM `higherdev.analytics_dev.JOBAD_VIEWS`
  WHERE
    DATE(timestamp) >= "2020-04-22" AND
    DATE(timestamp) <= "2020-04-30"
)
SELECT day, AVG(diff) as mean_ingestion_time
FROM diff_query
GROUP BY day
ORDER BY day;

```

Listing 4: The query

This query gives the following results:

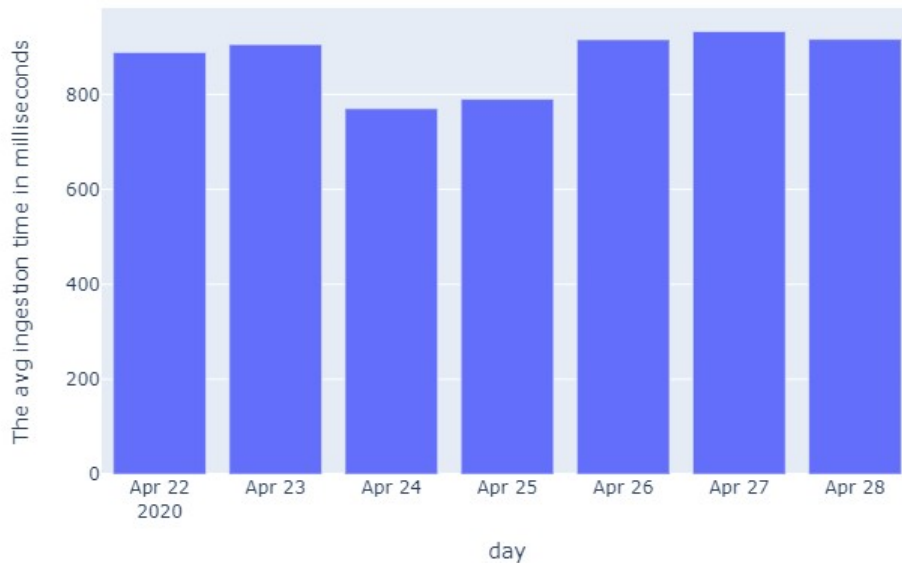


Figure 57: The time it takes in millisecond for an event to be inserted into BigQuery after it's occurrence

These results show that an event would be on average available in BigQuery in less than a second, clearly showing great potential for real-time analytics.

6. Should be able to persist event-data in a long-term storage

The team ended up persisting the raw event data into Google Cloud Storage in the form of files, in addition to storing the data in BigQuery. This was done by using Dataflow's windowing functionality to write event data to Google Cloud Storage every hour, grouping the events for every hour into a dedicated file, storing the files with timestamp-valued names. The result is a big collection of all the raw event data that has ever occurred, persisting the data for a long-term period.

7. Should be able to handle sudden increases in workload without affecting performance

The key here is to ensure scalability in the system. The current pipeline consists of GCP-managed services that guarantee scalability in their product. For the entire pipeline to scale, one needs to make every component in the pipeline to handle the increase in workload. Cloud Pub/Sub, Cloud Functions, Dataflow, and BigQuery is advertised to autoscale. Dataflow for example supports running up to 25 pipelines simultaneously, and can horizontally scale up to 58 virtual machines by monitoring CPU usage. The only component that does not autoscale is the Redis-cache in the enrichment step, which needs to be scaled manually in GCP.

The real question is how increases in workloads affect query performance. By looking at the experiment with BigQuery from figure 50 and 49, one can observe that the Dataflow throughput and the count of rows written to BigQuery per second increases over time, in addition to no change in query performance. This means when the workload increased the query performances stayed the same, when ingesting a total of 6 GB of data.

4.2.3 Non-functional requirements

1. **The system is well documented**

The team has made system documentation and commented the code to achieve this goal. This documentation can be found in attachment E, where it for example describes how to run the pipeline with different configurations or the underlying architecture of the pipeline.

2. **Code is clean and tested**

As a consequence of the process the team used several iterations of the code were written as the team continued to get to know the tools and experiment with different ways of using it. So the final code is the result of several iterations and many considerations by the team to make the code as clean and readable as possible.

3. **Secure, users only see the data they have access rights to**

The query API uses the same OAUTH 2.0 service as all the other API's Highered uses. This means that the data in BigQuery is only available through the Query API, and for receiving information the users have to send an authentication token - containing information on what data the user is authorized to.

4. **Costs are kept lower than previous event data pipeline**

The old solution using PowerBI had a monthly cost of 30 000 NOK. Calculating the cost for the new solution is not quite as cut and dried since the new solution does not have a static upfront cost, but is calculated from resource use. The lowest monthly cost for the new solution is ca. 3 000 NOK (ca. 100 NOK / day), and the team believes that this will cover most days that the pipeline spends in production, at least for the number of users Highered has at the time of writing. Under stress testing, the team observed a cost of 500 NOK / day, which if extrapolated over the entire month would result in a cost of ca. 15 000 NOK / month. This cost estimates only include costs from Dataflow, as that is the most expensive part of the pipeline and all the other parts have yet to generate any cost, as the project has been well within the free-usage plan of the various services.

5. **Query times are kept under 5 seconds**

The figure below shows that excepting outliers the query time is well below 5 seconds for data volumes up to 2.4 GB of data. These query times are a result of optimizations like time-partitioning and data-denormalization.

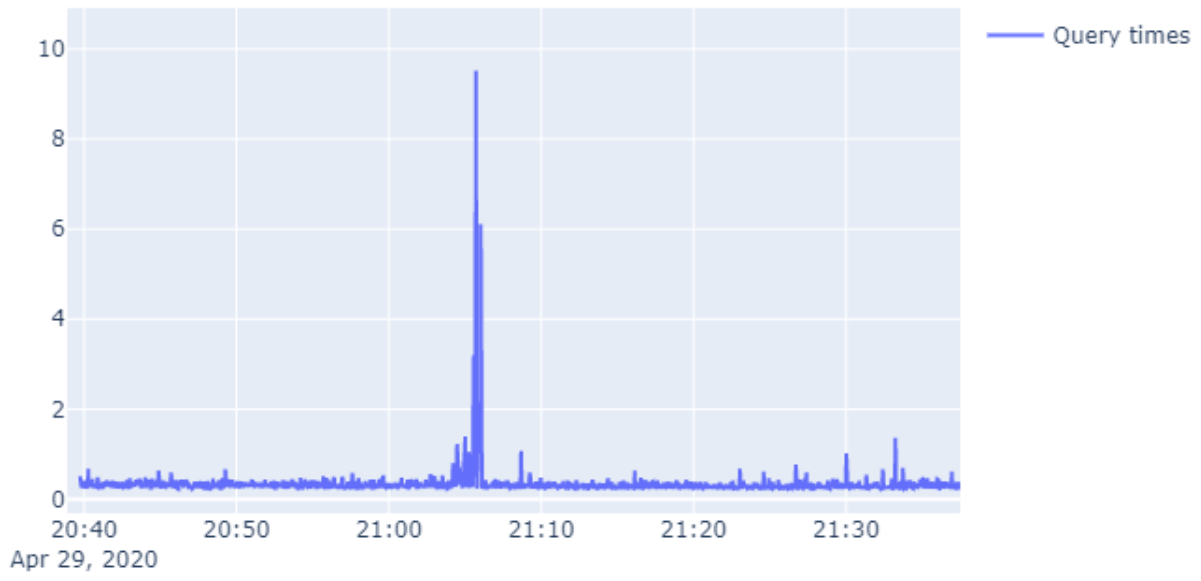


Figure 58: Query times (s) from stress test

4.3 Administrative results

While working on the project and doing the relevant research, one followed an underlying plan in the form of a Gantt diagram and sprints to ensure a systematic process. One was also following the rules and roles stated in the agreed work contract. The documents for the Gantt diagram, the sprints, and the work contract can be found in attachment B, the *Project handbook*.

4.3.1 Work schedule plan

According to the Gantt diagram, seven sprints were planned, and seven sprints were actually executed, with some of them having various lengths. The long term plan in the Gantt diagram was distributed in five phases, where those phases were either connected towards the goal of completing the project for Highered or pointing towards the goal of answering the research question. These phases ended up being roughly followed, except for at the end, when the plan of starting writing on the paper and implementing the final pipeline in parallel failed. The result was that the paper was delayed until the end of April, leaving a huge load of work at the end.

4.3.2 The development process

With some of the team members having work experience in a few businesses, one was inspired by the development process some businesses were practicing, and the team, therefore, ended up using a variant of a lean inspired scrum process. The processes consisted of executing sprints with two-week lengths, having stand-up meetings every workday, and having sprint-planning meetings at the beginning of each sprint. In the sprint-planning meetings, one was reviewing the work of the previous sprint, determining if some tasks were not completely finished or not. If there ended up being some uncompleted tasks, these were automatically added to the current sprint. After the review, one created a goal for the sprint and created needed epics and tasks necessary for achieving the goal, which was later inserted into ZenHub or Todoist. The daily stand-up meetings ended up being held together with Highered's tech team, resulting in the CTO always being up to date with the current status and future plans, creating a daily feedback loop between the team and the project owner.

With the occurrence of COVID-19, the team ended up being split into different locations, resulting in working over the internet in the form of video calls and text messaging. This increased the difficulty of executing some meetings and sustaining good communication between the members. In addition to this, one of the team members ended up going on sick leave at the beginning of April due to mental illness, resulting in his absence for the last two months of this project. With this said, the team's productivity the last two months, from April to and including May, ended up being reduced with roughly 33%. The rest of the team, therefore, went into some stressful situations and focused on being more productive, in the hope of still being able to follow the underlying plan, which in the end may have affected the quality of some work.

5 Discussion

5.1 Scientific results

In this sub-chapter, one will discuss the results of the experiments, their value, meaning, and how they can be connected to the research question. In addition, one will elaborate on the strengths and weaknesses of the experiment, and what could have been done to create a better result.

5.1.1 Overview of the experiment

The overall plan of the experiment is to find out what aspects of their underlying architectural designs and implementation makes a database scalable, and which those aspects are believed to be the most important ones. The experiment consisted of running some tests on various kinds of databases, some with different and some with similar architectural designs. These tests mainly focused on ingesting gigabytes of data into the database, while observing possible changes in query performance and ingestion performance. With the results of these tests, the plan was to determine which of these databases scaled or not, and from there try to point out which aspects of the database are causing those conclusions.

While one was running the experiments, one ended up having different test-parameters from every database, due to changes in the pipeline, different database systems, and restrictions in cost and time. On the other hand, if one had taken these parameters into consideration during the planning of the experiment, one could have prevented these differences from happening. The team did not have any prior knowledge or experiences with these databases, and therefore had little chance of knowing about preventing such a mistake. The reason one would like to have the same testing parameters for all the databases is to make the numerical results more comparable. If those numerical results were comparable, one could for example see that one database was numerically getting better query performance than another database, making it possible to draw conclusions of which query execution method is better than the other, and perhaps is also more scalable. These test parameters can for example be the executed query, the ingested dataset, the amounts of hardware resources available, and the amounts of data ingested. The more equal these parameters were between the different experiments, the more conclusions can be drawn from the results, making it easier to find an answer to the research question. With this all said, the current experiment on the other hand can only draw conclusions from each individual database-test, and from there see which aspects are present or missing within the database compared to the other databases in the experiment. This just means that one would have better results to work with if one did not have these differences in test-parameters.

5.1.2 Recorded metrics

For the experiment, the team decided it was useful to track the following metrics, *the query times*, *ingestion times*, Dataflow's *throughput* and the databases' *storage utilization*. With these metrics, one can observe how the database is performing in terms of extracting and inserting data, and one can additionally observe the pipeline's throughput, basically saying how much data the pipeline is able to output every second. Since one ended up not being able to extract the same type of metrics for the different databases, like ingestion times, one just ended up with another reason for not being able to compare these metrics across the databases. For some databases, one could simply extract ingestion times in the form of *byte-throughput* and *rows written* per second from GCP's metric system, like for BigQuery and BigTable. For Postgres one had to calculate it manually by exploiting Postgres' default value functionality, and for Druid one had to extract the consumer lag from the Kafka Indexer. These metrics have in common that they say something about how fast the database is able to consume the data on changing levels of stored data volumes. The results of these metrics will, therefore, help to determine which underlying implementations and architectural designs for storage perform well for handling big increases in workload and data volumes on the different databases.

5.1.3 Postgres

Postgres is a relational database with a focus on ACID-compliant transactions. In addition to this, Postgres has some built-in features for optimizing performance, like table-partitioning and master-slave replication. Sadly, the team was not aware of these optimizations until after the experiment, and therefore, executed the experiment without them. The result was a database that was not able to scale properly in terms of query performance, but one might have gotten better results if one was utilizing these optimizations. It would be interesting to experiment with how much each of these features optimizes the query performance, which would also give a good indication of their importance. By executing such an experiment one can find out how valuable these optimizations are for the database's scalability, by testing them one by one, but due to time constraints this is only a potential for further work.

The experiment was executed by ingesting in a total of 15 gigabytes of data by running a query that could potentially be used in analytical contexts. From figure 8 and 12, one can observe that the query-execution times increased linearly as data was ingested up to 10 gigabytes. From wave 2, one can notice from 13 and 17 that the query-execution times got more unstable while ingesting up to 15 gigabytes, as the query times were bouncing between 30 and 60 seconds. Since the query is not filtering by index, the execution was most likely a full sequential table scan, resulting in more data to scan through as the data volume increased. If the team had utilized time-partitioning and made the *jobAdId*-column an index the database might have executed the query with far fewer data volumes scanned compared to what was scanned in the experiment. On the other hand, the ingestion times showed more promising results, as the results in figure 9 and 14 shows a linear decrease in ingestion times as the data volume increases. Postgres' storage model is based on row-wise storage, making the database fast for insertions. Why the ingestion times are decreasing is uncertain, but the team assumes it has to do something with the utilization of The Shared Buffer.

The disadvantage of this experiment was the fact that Postgres was not utilized to the fullest, meaning it could have performed better with the supported optimizations. With Postgres one had to the chance to find out how much time-partitioning and master-slave replication could do for the query performance, giving valuable insights about which aspects are necessary for scalability. The current results on the other hand illustrate the fact that a database does not scale well with the implemented aspects of Postgres. As a result, one can use these results to observe which optimization possibilities was missing during the experiment and use those conclusions to find which aspects of a database is important for scalability.

5.1.4 BigTable

BigTable is a database for handling petabyte-scale data volumes with low-latency. Internally, BigTable is built upon a distributed file-system and a cluster of nodes, executing read-operations in a distributed highly parallel manner with MapReduce. With BigTable's underlying architecture, BigTable is extremely fast for insertions and has an overall high throughput rate. However, the database is only able to utilize it's fullest potential with a large amount of data and is therefore not good for small gigabyte-sized data volumes.

The results of the experiment indicate that BigTable is a highly scalable service. Since the Dataflow-enrichment step also was asynchronous - BigTable's throughput was extremely high, capable of ingesting over 30000 elements per minute, as seen at figure ?? and ?. This indicates that BigTable handles ingestion with really low latency, and as a result ended up ingesting a total of 100 gigabytes after the first wave, within only 15 minutes. According to figure 30, BigTable had ingested additionally 255 gigabytes within a total of 30 minutes after wave two, which proves that BigTable's capabilities for ingestion are at least scalable up to hundreds of gigabytes. In total, there were 350 gigabytes of data ingested into BigTable, which was way more than the team planned.

When it comes to query performance BigTable kept showing promising results. From figure 27 and 32 the query times was stable and was always below one second. With a total of 15 gigabytes of compressed data, there were no show signs of changes in query performance during ingestion. With this, one can con-

clude that BigTable is overall scalable up to at least hundreds of gigabytes, but BigTable is advertised to be fully utilized with terabyte-scales. Likely, the results are highly impacted by the number of nodes in the cluster. The team decided to use a cluster of 9 nodes, which is extreme for a total of 350 gigabytes of data. The data was even compressed down to 15 gigabytes, showing BigTable has good compression functionality, but it also makes sense because of a low variety of job postings in the enrichment step.

Overall, the BigTable experiment was quite successful and showed that its underlying architecture and implementation is extremely scalable. However, it is also worth noting that BigTable comes with a set of restrictions on its usage. It does not support SQL and is only able to extract matching rows from a table scans, meaning it is not able to create aggregations for analytical insights on its own. Such logic needs to be done by an external service, like Dataflow, and therefore, these restrictions are contributing to why BigTable is so fast and scalable as it is.

5.1.5 Druid

Druid is a database that is built for analytics and is advertised to handle petabytes of data. One of the key aspects with Druid is its capability to scale horizontally, due to its microservice-based architecture. Each of the core services in Druid is separate processes that can be either deployed on separate machines or on the same hardware, which opens for deploying more of these services and splitting the workloads between them when the amount of work suddenly increases. Another aspect of Druid is how it handles the execution of queries. It splits the work between the Historical nodes due to its segment-based storage design, resulting in concurrently resolving the query, increasing its performance. Besides, Druid also consists of time-partitioning, data compression, roll-ups, and columnar-storage, making Druid the advertised petabyte-scale analytics database.

With these aspects of Druid in mind, one can start concluding why the experiment on Druid went as it did. The experiment did not go as expected, as the database rapidly ended up with query times going over 2 entire minutes, which is not expected from a petabyte-scale database. Also, Druid suddenly started to crash, as one can observe from figure 27 when the query times went from several minutes down to 0 seconds after some minutes of pause. For this reason, the team decided that something was wrong with the configuration of Druid, and noticed through Druid's built-in dashboard that the segments were never published and stored in the Historicals. The Historicals are the fundamental blocks for Druid's fast query execution since they can concurrently resolve queries with the coordination of the Broker-service. Since the segments were not published, the data must be held in memory, resulting in a different method for querying that data inside the Indexer. According to the results of the query times in figure 35, 39, and 43, that method is extremely slow and not scalable. The team unofficially tested ingesting into Druid with 7 gigabytes of data outside of the experiments and did not encounter the same problems, as one observed that the segments were being published. From these unfortunate events, one can conclude that the aspects of Druid that was not present during the query execution inside the Indexer are crucial for Druid's scalability when querying on big data volumes. With this said, the most important missing aspects were *data distribution*, *concurrent query execution*, *time-partitioning*, and *data compression*. Therefore, even if the experimentation of Druid failed, one was still able to extract useful information for Druid's scalability and can use these missing aspects for further discussion.

One can also observe from Dataflow's throughput in figure ??, ??, and ?? that it clearly was Druid that did not scale as it should, as the throughput of the Dataflow job was relatively high when Druid did not crash. The results of Kafka's consumer lag also supports this statement as the lag increases linearly, meaning Druid became slower and slower with handling insertion.

5.1.6 BigQuery

BigQuery is a database that is advertised as highly scalable and capable of handling data at petabyte-scale in a matter of seconds. The core underlying architecture and implementation of BigQuery is based on Colossus, which makes it possible to distribute data over big amounts of machines, giving performance benefits in addition to storing that data in a column-oriented fashion. A different important aspect of Big-

Query is its query model, which basically splits the query into subtasks and distributes them into a tree of nodes, running each task in parallel for better performance. These underlying features are some of the many aspects that make BigQuery a high performance and scalable data warehouse.

BigQuery was the last database that the team experimented on and the results gave the indication that BigQuery is highly scalable as advertised. One problem with the experimentation was that the pipeline currently had a synchronous enrichment step, reducing the throughput from Dataflow. The result of this was a fewer amount of data than planned was ingested into BigQuery. Relative to the other experiments, 6 GB of data is not really that much of data, especially for BigQuery. Despite the low query times from figure 48, the ingested volume of data was not big enough to indicate BigQuery's overall ability to scale, and should instead be at the terabyte level to really get interesting results. The team originally planned to ingest the same amount of data as one did with BigTable, over 350 GB, but sadly due to cost constraints and a synchronous enrichment step it did not happen. With all this said, the current results of the experiment do not really give a trustworthy proof of BigQuery's ability to scale, and the result itself was therefore not surprising.

While looking at figure 48, one can see a sudden spike in the middle of the experiment. Since there was nothing special that happened during that time interval, the team assumes this was just a network lag from the machine running the Dash dashboard, due to a bad network connection. One can also notice from figure 49 that the amount of rows written to BigQuery per second is quite low, especially since that it took over 50 minutes to ingest only 6GB of data. The synchronous enrichment step is partly to blame, but it is also worth mentioning that BigQuery is not high performant for inserts, due to its columnar storage architecture and since the data insertion is row-based.

Based on the results of the experiment one can conclude that BigQuery is scalable up to gigabyte-scale. If one looks at some other benchmarks, one can conclude that BigQuery is capable of scalable up to terrabyte-scale[32]. With this, one can argue that the reason for BigQuery's ability to scale is the way BigQuery handles data storage, data distribution, and query execution. In other words, concurrent query execution model, columnar storage architecture, data compression, and a distributed file system is the main reason BigQuery is capable of scaling.

5.1.7 Summary

In this paper, the term scalability was narrowed down to scalability in terms of query performance, meaning how do sudden increases in workload affect the overall query performance for the system. To find out how the experimented databases achieve scalability or not, one has to look at their internal architectural characteristics and implementations, which this paper has mentioned the following list from the experimented databases:

- **Distribution of data** - distributing data between a cluster of nodes, connected by a network. This means that each node has a subset of the total data.
- **Replication** - This means copying blocks of data to other nodes in the cluster. These blocks can either be read-only, or can handle writes as well, requiring an algorithm of achieving consensus.
- **Parallel query execution** - Executing the query in parallel can lead to big enhancements in query performance. Some implementations utilizes tree-structures to achieve parallelism in a cluster of nodes, like in MapReduce, BigQuery and Druid.
- **Columnar storage** - storing the data by column values, not by row. This allows for faster query times when selecting a few number of columns while querying. It also allows for better compression as column values are often homogeneous, resulting in less data to process during query execution.
- **Table-partitioning** - splitting the data by a certain column, creating better query performance for queries than filters by that column.

When looking at the Postgres experiment, it is interesting to see which of these aspects it was missing for achieving the mentioned scalability in query performance. The experimented Postgres instance did neither have distributed data, replication, or columnar storage. The instance had parallelism for query execution, scanning through the table in parallel. However, parallelism on a single machine is restricted to the hardware's resources, making it only possible to achieve performance benefits up to a certain point. This means one needs to start scaling horizontally to achieve better performance, basically executing the query on several machines simultaneously. As a result one has to involve more machines with access to either all the data or a subset of the data, ending up with distributing the data across several nodes, creating a cluster. In the end, one ends up with a new distributed database that runs queries in a parallel tree-reductional manner, like in the MapReduce-programming model.

The question is how well does this new database handle increases in workloads? Can it handle gigabyte-scales and maybe terabyte-scales of stored data with high query performance? Can it handle sudden increases in incoming read and write requests? The results of the experiments do not hold a trustworthy answer, but logically, the answer is no. When the number of requests increases, one starts getting bottlenecked by the nodes' hardware resources, meaning one has to add more nodes to the cluster. In a distributed system, the data is split between the nodes, making them responsible for a subset of the data. If the incoming requests are attracted to a special subset of the data, meaning some nodes get extra workloads to deal with, these nodes will experience lower performance. To avoid this, one has to replicate the data into the other nodes of the cluster, enabling *data replication*. In the end, just to preserve scalability in query performance one would have to utilize at least three of the listed architectural implementations.

What BigTable, BigQuery, and Druid have in common is that all of them utilizes those three architectural implementations. They are all distributed, utilizes data replication, and execute queries with distributed parallelism. In addition, both BigQuery and Druid have a columnar-storage structure because of their support of SQL, which is oriented around columns. As for table-partitioning, it was utilized by both BigQuery and BigTable, but Druid and Postgres did not get to use it even though it was supported. Both columnar-storage and table-partitioning also enhances query performance, but might not be just as necessary for the overall scalability. As the stored data volume grows, one can always scale by adding more nodes and distribute the data between them. Therefore, columnar storage and table-partitioning might not have equal importance as the other architectural implementations, however, these logical statements are not trustworthy proofs of this conclusion.

The experiments did overall go well, as they were able to produce results that made it possible to draw speculative conclusions for the research question. BigTable scaled the best because of its restricted, but fast, storage model. BigQuery was also quite scalable in terms of query performance but had slow ingestion performance. Postgres and Druid did not scale well because they did not utilize the mentioned architectural aspects. The experiments did not quite go as they were planned, but were a success from the team's point of view. If the team was going to redo the experiments, one would instead start by testing with a database without any of the mentioned architectural aspects, like Postgres, and then start adding them one by one. This way, one would be able to get a better measurement of how these aspects affect the overall scalability of the database, and might, therefore, be able to compare their importance in terms of scalability better.

5.1.8 Choosing a database

Highered needs a database that can power their analytical needs and demands. The company wants to be able to scale when its number of active users increases, in addition to wanting to store more analytical data about them. Postgres is actually a good candidate for analytics, as long as the data volume is not too large and the supported optimizations are enabled. Since Highered wants to be able to scale Postgres will in the end be a restriction. On the other hand, BigTable would easily be capable of dealing with Highered's scalability needs, however, Highered does not have a product that produces the analytical data volume BigTable is designed for. In addition, BigTable introduces extra complexity for calculating the aggregations for the wanted analytical insights. Therefore, Highered might be better fitted with an SQL-database. Based on the team's empiricism from the experiments, Druid was a pain to deal with, as it kept

crashing and needed to manually be managed and monitored. With all this in mind, the team decided that the best fit for Highered would be to use BigQuery, as it is for sure scalable within the data volume Highered wants, supports SQL, and is fully managed by GCP, making it them a perfect pair.

5.2 Engineering professional results

5.2.1 Final result

The team made several versions of the pipeline, all with different success criteria, most of them being made to test and experiment with Dataflow. By iterating like this the team gained experience with how a good, in the teams eyes, pipeline should look and function. The final pipeline is a combination of all the things the team learned. And also the considerations the the team made as to what would be the best pipeline fitting Highered's needs. The team thought that keeping the pipeline simple would be a great strength. It seemed easy to write a lot of code to do a lot of fancy things, but by keeping the pipeline simple, within reason, the team made sure that creating good documentation would be easier, and that understanding both the documentation and the code itself would be considerably easier and less time consuming for Highered's tech team that lack the experience and knowledge that the team gathered while making this project.

Highered had the expectation of having a new solution to their analytics, as the old one was far too expensive and had no more room to grow. The team views the proposed solution, accompanying pipeline and knowledge that the team offers as reaching that expectation and more.

The pipeline ended up consisting of exclusively managed services that all are offered by Google, this was seen as a good choice by the team because it allows the small tech team in Highered to not have to focus on maintaining a complex selection of enterprise software solutions to scalable pipeline components; Having a server that could crash and that requires someone to connect into the VM to restart it would be a very bad choice for Highered because it does not have a tech team that has the capacity for always-on tech support. Considering that the team prioritized having only managed services and also prioritizing GCP, because Highered has a lot of services running on GCP and is currently transitioning the others, it was clear cut to the team that a Pub/Sub to Dataflow solution would be the best option, as Dataflow is the only truly managed scalable analytics pipeline solution Google offers on their platform. So then the only remaining question was what database to use. From the beginning the team considered BigQuery as the obvious answer, because of the usage implications for Highered, but decided that the database question was interesting to look more at. Google even offers several managed database solutions that could have been used. In the end the team landed on using BigQuery, there were several considerations made for this option, but the most important consideration was too Highered's usage. Highered already uses BigQuery and therefore has a lot of experience and knowledge with it.

5.2.1.1 Strengths

In comparison to the old analytics solution, using PowerBI, the strengths of the new one are numerous. The new solution has a more generic handling of event types that allow for experimenting without having to change the schema for countless other old events that are all in one table. And the new solution has a way to enrich the event data, thereby optimizing it for querying, before it is even put into the database; In the old solution any transformation or data aggregation would have to happen at query time, meaning most of the analytics queries combine several SQL views and combine data sources from various servers, resulting in very slow query times. The old solution also relied on running batch jobs every day to update the query views, meaning users of the old dashboard would not see analytics data before the following day. The new solution is also very flexible and customizable in its use, because it is based on an internal code-base and not a drag-and-drop UI for building analytics dashboards that PowerBI uses.

5.2.1.2 Weaknesses

The customizability of the new solution is also a weakness. The solution, while being much more customizable, also restricts that customizing to the tech team, unlike the old solution which used a drag-

and-drop UI that people at Highered outside the tech team could use to change the dashboard and add columns to the dashboard. The team chose to only use managed services instead of writing their own services, to run on Highered's Kubernetes cluster, which has the downside of introducing a lot of new and unknown technologies into Highered's stack, the teams solution also requires more documentation.

5.2.2 Functional requirements

1. The system should be able to insert an event through an API

The team used Golang for the Cloud Function because Golang is a programming language used for most of Highered's APIs and a language that the team has a large degree of proficiency with. Golang is also very suitable for this application because of its simplistic nature and easy syntax. The team decided that the Cloud Function should as simple and generic as possible to make it easy to build new event types without needing to change code in several code bases (enrichment, Cloud Function and database insert).

2. The system should be able to handle 1 000 requests per second

The way the current pipeline is setup everything is handled synchronously, so the CPU utilization during the stress test was very low. The team experimented with asynchronous enrichment but chose not to implement this in the final solution because it limits the type of outputs the enrichment stage can have, usually Dataflow supports multiple outputs for every stage. One can for example output enriched data of various types and errors to completely different branches of the pipeline, but when using async enrichment the system needs to run several enrichment functions inside each of the runners that can output multiple outputs, thus though it might be possible to do, it would require a lot of time and a lot of effort to support multiple outputs from async enrichment. The code for async enrichment still exists and it is quite possible to refactor to use this at a later time should the current throughput become a bottleneck in the system.

3. The system should never lose one event

Highered uses the data gathered by the system to sell the Highered platform to schools and companies that want to higher students to various jobs all over the globe. The data will provide value by allowing companies to see which schools interact and view their job ads and by how much. Schools use this data to see how different companies are performing at their schools and can give insight to the career service about what kind of jobs students at their school prefers. Because of this it is very important that very little event data is lost. The thought process behind the solution to this problem for the team is to remove as many weak points from the teams code as possible especially before the event is stored in long term storage. To do this the team determined it was very important to keep the teams code as simple as possible while maintaining type safety and error handling. Following the event from the front-end client the event moves through a Cloud function written in Golang which provides type safety and easy error handling. After that the event moves through Pub/Sub which is entirely managed by Google the event gets to Dataflow which runs the teams Apache Beam code written in Java which also has type safety, but unlike Golang that has functions that can return multiple items, like a value and an error, Java handles errors by throwing exceptions.

4. The system should be able to distribute events in different types

When the team was thinking about how to make the solution high performant for querying, it was decided to split the different event types into separate BigQuery-tables. The key reason for this is to only scan through the data that is relevant for the query, reducing the workload of the query. While taking a look at the queries that powers Highered's existing analytical dashboard, each queries can be split up by the different types of events, making it possible to store each event type in a separate dedicated table.

The disadvantage by distributing them into separate tables is the need of performing a *SQL-join* if one would like to find relations between two or multiple types of events, increasing the time executing the query. With this in mind, the real question is if one would actually benefit from distributing the events in different tables in cost of having to do additional SQL-joins, instead of just querying

a larger table. The team decided to base their decision on Highered's existing dashboard solution, thinking the amount of SQL-joins needed actually is minimal, and that one would benefit more from scanning through smaller tables than a large table containing all the different types of events. If the team's decision actually gives better query performance is currently unknown, and has not been tested due to time-restrictions.

5. **The system should be open for "real-time" support**

The vision document states that the new solution should be open *real-time* support, meaning it does not necessarily have to be fully implemented, but the system should be open for adding real-time functionality without having to do big system changes or time-consuming refactoring.

With visualizing data in real-time, it is fine with some latency, but the lower the latency the better. To achieve real-time one would have to be able to serve the data within a certain time-constraint, and what that time-constraint is differs from system to system. For some systems, the time constraint might be in millisecond-level, and for Highered that time-constraint is most likely in the level of seconds. One would for example want to be able to answer how many users are currently looking at a certain job posting in real-time within a maximum delay of 10 seconds. This means that the new solution must be able to ingest the event in the database within that time-constraint.

According to the data in BigQuery, the implemented pipeline was able to serve the events within less than a second after their occurrence. This means it took less than a second for an event to go through a Cloud Function, Cloud PubSub, getting enriched in Dataflow, and then inserted into BigQuery. With these statistics alone, one can therefore say that the system is able to serve data in real-time. However, one can argue that these statistics might change if one needs to deal with for example 1 million events simultaneously, even if the experiments have shown that the implemented pipeline is capable of handling a huge amount of events simultaneously. On the other hand, for making the pipeline handle an amount of that scale one can do various optimizations, like for example utilize Dataflow's windowing feature and insert the data directly into a cache. This would speed up Dataflow's throughput a lot more, especially by avoiding the enrichment step by branching, removing the pipeline's biggest bottleneck. With all this said, one can with high confidence conclude that the implemented pipeline is quite open for supporting real-time analytics, without the need of re-designing the system.

6. **Should be able to persist event-data in a long-term storage**

Being able to persist event-data in long-term storage is important and useful for several reasons. It allows Highered to make breaking changes to the analytics pipeline and then put all the old event data through the new pipeline. It does however mean that choosing that the format for event data has to be very generic. In addition, in case of system failures and data loss it is also quite useful to have a backup laying around in a long-term storage.

For this purpose, the new system currently ingests data into two different storage locations, in BigQuery and Google Cloud Storage. Both of them is designed for persisting data for a long-term period, but in this solution, BigQuery is not storing the events raw, meaning that the data is slightly modified to achieve certain requirements. This means, if one would for example want to migrate away from BigQuery or ingestion into BigQuery fails for some reason, having a backup of the raw data is almost a "must-have" thing. One might also one day want to remove data from BigQuery, due to performance and cost, and therefore it is good to have a backup archived somewhere else. As a result of this mindset, the team decided that storing the raw event-data in Google Cloud Storage would be a reasonable idea, and implemented it into the pipeline with long-term storage in mind.

7. **Should be able to handle sudden increases in workload without affecting performance**

The current pipeline has proven to handle the workloads that occurred during the experiment, being able to ingest over 2500 elements per second without affecting query performance. This alone does not necessarily mean the pipeline would be able to handle any increase in workloads, while preserving the pipeline's throughput. If for example the pipeline would suddenly need to handle 100 000

events simultaneously, the result would most likely be a bottleneck in the Dataflow job, due to the enrichment step, leaving most of the events buffering in the Pub/Sub queue. This does not necessarily mean that the pipeline's throughput goes down, but it should instead go up by scaling either horizontally or vertically, depending on the system.

Right now, the Dataflow job's enrichment step is running synchronously, meaning it is not fully utilizing the CPU. Since Dataflow's autoscaling is based on the running machines' CPU usage, the Dataflow job therefore might not actually scale when needed. One is able to configure a minimum number of running machines within Dataflow, but due to cost reasons and the needs of Highered, one would prefer to run only one machine when possible. One can configure the Dataflow job to run the enrichment step asynchronously, which was tested in the experiments and proved to scale up to 58 machines. The team decided to not go for the asynchronous implementation due to both technical challenges and time constraints with the current pipeline, but it can be seen as further work for the project. With this, it is possible to extend the pipeline's scalability limit, making it possible to handle bigger workloads, but for Highered's current needs, the current scalability limit is fine according to their current data ingestions. The important part for Highered is that the pipeline is able to deal with all the data in the end, without affecting query performance, and that is not currently an issue with BigQuery, according to the experiments. Therefore, one can conclude that this criteria is partly achieved, but can be improved in further work.

5.2.3 Non-functional requirements

1. The system is well documented

Documenting the final solution is very important in this projects case because Highered will be using it in production and will build further onto it. Documenting the project is not quite as simple as just writing about the code the team wrote, but also documenting presumptions about the event data and decisions made about technology. The terminology, thought process and code base behind such a analytics pipeline do not exist in Highered from before, as the old solution did not require any code from Highered or met the scaling requirements that this project have defined. Therefore it is extremely important for the team to communicate all the required knowledge to run and maintain the pipeline with all it's nuances and configuration.

2. Code is clean and tested

This goal carries on the thought process from the goal of documenting the system well, by not making the code overly difficult or complicated the team achieves a lower barrier to entry to the code base and components to the pipeline, meaning a faster turn over rate to the rest of Highered's tech team who will continue to build and improve the system. Testing is also important for this purpose, as it allows for runnable examples and edge case assurance that are run independently from the rest of the pipeline.

3. Secure, users only see the data they have access rights to

Because the analytics the pipeline deals with is not publically available and is used as a selling point towards Highered's customers it is very important that the data is not only securely stored but also that users do not see data they have no connection to. Luckily TalentPanel, the front-end users utilize to see the analytics, already has a authorization system, shared by the entire Highered platform, that the teams query API can use to only retrieve data that a specific user has access rights to.

4. Costs are kept lower than previous event data pipeline

Costs are a big part of why Highered wanted to create a new analytics pipeline, because the old solution is very costly and does not cover all the needs Highered has of it. For the old solution Highered was paying for both access to the PowerBI service and also renting a static VM to host PowerBI, this meant that when no-one was using the system there was still a running cost, and when many people were using the system the VM could easily become overcrowded with requests and thus ruined the user experience and usability of the system. The teams final streaming pipeline result fixes this by only using components that either automatically scale according to use or that can be easily scaled manually without needing to shut-down the system, thus allowing near 100% uptime.

5. Query times are kept under 5 seconds

Because the analytics dealt with in the teams system are used as selling points to paying clients a certain expectation is held to the user-experience of the system, especially when it comes to load times of analytics. The old system tended to be very slow and that slowness played a big part in wanting to build a new, entirely from scratch analytics pipeline. The 5 seconds goal was set as a maximum, the idea being one or perhaps two of the queries made could take upto 5 seconds, but no more. Currently the system only makes requests in the 800ms area, so there is a lot of room for adding more complicated queries that perhaps join data from various sources, as a test before the aggregation is built into the pipeline.

5.3 Administrative results

5.3.1 Work schedule plan

The team tried their best to follow the work schedule set out from the start, but allowed it to change underway per the teams agile mindset. This means the final work schedule that is attached to this report is a work schedule that actually shows what the team worked on and not just what the team planned to work on. The team experienced, like most of the world, some major disruptions coming from the Covid-19 pandemic that stopped the team from going to a common office and prevented meeting physically for meetings. This meant the team had to adapt their work process to work in accordance to temporary crisis rules put in place by the Norwegian government.

5.3.2 The development process

The development process chosen was a variant of scrum which was quite alike other processes the team has used with success in other projects, so it was not entirely new to the team. In fact, the team has worked together for more than two years, one year of those working together in Highered, who also uses a scrum like process with stand-ups every working day. This experience meant the team knew what works the the team and what does not, therefore the team chose to adapt a mindset for meetings from a team members experience in another company, by having less meetings (excepting stand-ups) and having them be well planned and as short as possible. All in all the process functioned more or less as expected, with no surprises. With this process not much time is wasted with long meetings that do not answer specific questions that need answers before going into the meetings, this allowed the team to focus much more of their time to the development itself. The daily stand-ups allowed the team to communicate the current status of various tasks and new ideas or problems quickly and efficiently. The team was planning to have review meetings with Highered's CTO, so he could make sure the project was going as planned and was not diverging from Highered's goals, this however did not happen, mostly because of time constraints in the teams core time versus Highered's CTO's work schedule, during the project period he only worked 8:00 to 12:00, whilst the teams core hours were 11:00-15:00, so any review meeting had to be within the hour between 11:00-12:00 and that usually was not a good time for Highered's CTO as he had lots of other meetings with other parts of the business at those times.

Judging just how long time certain tasks will take is always difficult and in the teams experience almost always wrong, so the team decided to not estimate how much time each task would take, but instead work on a set of tasks over the entire sprint and hopefully have every task done at the end. If any tasks were left undone or needed more time at the end of a sprint, the specific tasks would be moved to the next sprint and prioritized over other tasks, unless if it was decided to simply not do the task.

The group as a whole functions well together, and it is not by chance the team worked together on this project, with the teams experience of working together at NTNU and them working together at Highered it was natural to use the chance of writing a bachelor thesis, to do the project at Highered.

But of course the process was affected by the global pandemic, it made it a lot more difficult to entertain new ideas, as the day to day informal communication that happens during lunch or just meeting at the office became more difficult. This resulted in stand-ups to last a lot longer than they are supposed to, some-

times over an hour, and most meetings extending way beyond the planned agenda, as team members had a need to communicate more informally than the planned meetings would usually allow.

6 Conclusion and further work

6.1 Conclusion

This paper has from the start mainly focused on building a streaming pipeline in parallel of answering the following research question:

How is scalability achieved in analytical databases used in big data pipelines?

The experiments themselves indicate that *data distribution*, *replication*, and *parallel distributed query execution* is the key for the scalability of a database, in terms of query performance. Both Postgres and Druid did not scale during the experiment, because neither of them did utilize these aspects. Since BigQuery and BigTable utilize these key aspects and did scale, one can argue that with high confidence that these aspects are crucial for achieving scalability in databases. Besides, Druid is also under normal circumstances advertised to scale by utilizing those aspects.

It is all about being able to utilize hardware enough to handle sudden increases in workloads. A database on a single machine is only able to handle a certain amount of workloads until it starts affecting its performance. Even with parallelism the machine will at some point be bottle-necked by hardware, meaning vertical scaling is no longer an option. To increase performance one would have to leverage hardware on different machines, splitting the workloads between them. For the query execution to be scalable one would have to split the query execution between the different machines, which also requires the machines to have access to the data. This means the data needs to be distributed between the different machines and the query execution needs to be paralleled between them to utilize the overall hardware efficiently. As a result, the database would be utilizing both *data distribution* and *parallel distributed query execution* only for scaling the performance of a single query. This solution would only be able to scale as the data volume increases by adding more machines to the system, but will yet again be bottle-necked by hardware when incoming requests increases - as the distributed blocks of data can only be found at one machine at the time. Therefore, one would have to utilize *replication* for splitting the workload between the different machines, because by keeping parts of the data only on a single machine will restrict the entire system's processing capacity for that data on that single machine. Therefore, to be able to handle increases in both data volumes and requests, and at the same time preserve high query performance, all of these three aspects are necessary. This conclusion is also what the experiments ended up indicating.

Therefore, for achieving scalability in databases *data distribution*, *replication*, and *parallel distributed query execution* is the key. If one does not utilize all three of these aspects at the same time, one would only be able to scale up to a certain point until one runs out of hardware capacity. Scalability is therefore only about being able to leverage and append more hardware for the system to run on. With this conclusion, the team was able to find a suitable database for Highered, and additionally, was able to build a streaming pipeline for powering Highered's next steps towards scalable analytics.

6.2 Further work

Highered intends to use the project in production. To do so, some further work is needed to properly prepare the pipeline to production ready. First and foremost more code needs to be written on the query API, it currently lacks the ability to query anything other than what the team considered would be relevant to this project, in terms of being able to show what the team wanted to show. The query API also lacks functionality in the security layer put in place, which was only very a temporary solution, put in place to save time while also meeting the teams goals. The security layer would make sure that users of Highered's platform only see the data that they are authorized to see and has quite a few different facets of authorization, like for example what schools a user can see data from or what companies. The dashboard that was implemented by the team is, like the query API, made to show what the team deems necessary for this project and has not gone through the vigorous testing or feedback loops to ensure good user experience and functionality that a dashboard ready for production needs. The dashboard also does not currently show any real-time analytics data, even though the pipeline supports it, this was not implemented quite simply because of a lack of time. Lastly, in terms of building on the pipeline, Highered has event data from before that might not fit with the event format the team made for the project and would need formatting, before all the old event data is put through the pipeline so that the new system can deal with it, like it would any new event.

Going beyond what was possible within the time and resource limits of this project the team views more database testing as a natural step forward. Cloud Spanner is a relational and ACID-compliant database, like Postgres, but is also a managed database that checks all the boxes for being a scalable distributed database that might give great results for an analytics pipeline [33]. Doing more comparable experiments and benchmarks between Postgres and Cloud Spanner might be interesting, as Cloud Spanner and Postgres have some similar aspects in terms of consistency and transactions, but differ with Cloud Spanner's underlying distributed architecture. Such an experiment might make it easier to measure and compare the importance of the key aspects for scalability in databases.

It will also in the long term be relevant to consider making the enrichment stage asynchronous. The team experimented earlier in the projects life-cycle to get better throughput with asynchronous enrichment, and managed to push the pipeline upto 35 000 elements/second. By doing so the pipeline would be able to processes more events as one starts utilizing more of available computational capacity, increasing the overall performance of the pipeline. As Dataflow's autoscaling is dependant on CPU-usage, this transition would also make the Dataflow job being more capable to scale horizontally. But doing so would also compromise the code's readability and ability to change easily, and therefore should only be done by the same people who are going to maintain the codebase over a longer time.

Further work to improve the pipeline could also be to write more and better tests, and build out the generic nature of the pipeline to encompass more workflows that Highered could throw at it. Currently the team has only created unit tests to show that the various components of the pipeline work as expected, but when the pipeline is used in production it would become a lot more relevant to also test availability from various regions, and also make end-to-end tests, by for example putting test events through the pipeline but also by for example using a headless browser to click around on the platform and make sure that all the expected events are registered and dealt with by the pipeline. The end-to-end tests could also become a metric for how live the real-time data is. As the pipeline is planned to be put into production and integrated with Highered's existing applications, the importance of integration tests and system tests becomes more essential, especially since the pipeline is something that produces great value for Highered's customers.

The team has focused all their documenting efforts into what the teams thinks is most important to understand what the pipeline is and how to use it. Moving forward it would be relevant to document the various event types that Highered has, and how they are collected and how the pipeline deals with them; Documenting event-types and data handling can mitigate confusion, and also help in future decision making.

7 Attachments

Attachment A - The vision document

Attachment B - Project Handbook

Attachment C - Design Doc

Attachment D - Scalability Experiment Document

Attachment E - System-documentation

Bibliography

- [1] Apache Druid official website. <https://druid.apache.org/>. Accessed 12.04.2020.
- [2] Florian Zysset. An introduction to Druid, your Interactive Analytics at (big) Scale. <https://towardsdatascience.com/introduction-to-druid-4bf285b92b5a>, 2018. Accessed 12.04.2020.
- [3] EFMD Global. <https://www.efmdglobal.org/>. Accessed 05. May 2020.
- [4] Google Analytics Home Page. <https://marketingplatform.google.com/about/analytics/>. Accessed 05. May 2020.
- [5] BigQuery official web page. <https://cloud.google.com/bigquery>. Accessed at 17.05.2020.
- [6] PowerBI Home Page. <https://powerbi.microsoft.com/en-us/>. Accessed 05. May 2020.
- [7] Sanjay Radia Robert Chansler Konstantin Shvachko, Hairong Kuang. The Hadoop Distributed File System. <https://storageconference.us/2010/Papers/MSST/Shvachko.pdf>, 2010. Accessed 17. May 2020.
- [8] Albertas Krisciunas. Benefits of NoSQL. <https://www.devbridge.com/articles/benefits-of-nosql/>, 2014. Accessed 7. April 2020.
- [9] Samir Behara. Designing Highly Scalable Database Architectures. <https://www.red-gate.com/simple-talk/cloud/cloud-data/designing-highly-scalable-database-architectures/>, 2019. Accessed 8. April 2020.
- [10] George Farcasiu. Understanding Distributed Databases. <https://medium.com/@farcasiu.george/understanding-distributed-databases-5e7b30f154c5>, 2016. Accessed 9. April 2020.
- [11] ACID (Atomicity, Consistency, Isolation and Durability). <https://tomyrhymond.wordpress.com/2011/10/01/acid-atomicity-consistency-isolation-and-durability/>, 2011. Accessed 11. April 2020.
- [12] Syed Sadat Nazrul. CAP Theorem and Distributed Database Management Systems. <https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e>, 2018. Accessed 12.04.2020.
- [13] IBM Cloud Education. The CAP Theorem. <https://www.ibm.com/cloud/learn/cap-theorem>, 2019. Accessed 12.04.2020.
- [14] Kazunori Sato. An Inside Look at Google BigQuery. <https://cloud.google.com/files/BigQueryTechnicalWP.pdf>, 2012. Accessed 29.04.2020.
- [15] Mangat Rai Modi. Rowise vs Columnar Database? Theory and in Practice. <https://medium.com/@mangatmodi/rowise-vs-columnar-database-theory-and-in-practice-53f54c8f6505>, 2018. Accessed 29.04.2020.
- [16] Howard Gobioff Sanjay Ghemawat and Shun-Tak Leung. The Google File System. <https://static.googleusercontent.com/media/research.google.com/no//archive/gfs-sosp2003.pdf>, 2003. Accessed at 14.05.2020.
- [17] Overview of Cloud Bigtable. <https://cloud.google.com/bigtable/docs/overview>. Accessed at 14.05.2020.
- [18] Sanjay Ghemawat Wilson C. Hsieh Deborah A. Wallach Mike Burrows Tushar Chandra Andrew Fikes Robert E. Gruber Fay Chang, Jeffrey Dean. Bigtable: A Distributed Storage System for Structured Data. <https://static.googleusercontent.com/media/research.google.com/no//archive/bigtable-osdi06.pdf>, 2006. Accessed at 14.05.2020.

- [19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. <https://static.googleusercontent.com/media/research.google.com/no//archive/mapreduce-osdi04.pdf>, 2004. Accessed at 14.05.2020.
- [20] Chitij Chauhan. Understanding the PostgreSQL Architecture. <https://severalnines.com/database-blog/understanding-postgresql-architecture>, 2017. Accessed at 15.05.2020.
- [21] The Internals of PostgreSQL. <http://www.interdb.jp/pg/pgsql01.html>. Accessed at 15.05.2020.
- [22] Understanding How PostgreSQL Executes a Query. <http://etutorials.org/SQL/Postgresql/Part+I+General+PostgreSQL+Use/Chapter+4.+Performance/Understanding+How+PostgreSQL+Executes+a+Query/>. Accessed at 15.05.2020.
- [23] Table Partitioning. <https://www.postgresql.org/docs/10/ddl-partitioning.html>. Accessed at 15.05.2020.
- [24] High Availability, Load Balancing, and Replication. <https://www.postgresql.org/docs/9.2/high-availability.html>. Accessed at 15.05.2020.
- [25] Dataflow Home Page. <https://cloud.google.com/dataflow>. Accessed 06.05.2020.
- [26] Pubsub Home Page. <https://cloud.google.com/pubsub>. Accessed 06.05.2020.
- [27] Cloud Function Home Page. <https://cloud.google.com/functions/>. Accessed 06.05.2020.
- [28] Memorystore Home Page. <https://cloud.google.com/memorystore>. Accessed 06.05.2020.
- [29] Google's BigTable. <https://andrewhitchcock.org/2005/bigtable.html>, 2006. Accessed 07.05.2020.
- [30] Single server deployment. <https://druid.apache.org/docs/latest/operations/single-server.html>. Accessed 10.05.2020.
- [31] Apache Druid - Kafka lag. <https://druid.apache.org/docs/latest/operations/metrics.html#ingestion-metrics-kafka-indexing-service>. Accessed 10.05.2020.
- [32] Eric O'Connor George Fraser. Cloud Data Warehouse Benchmark: Redshift, Snowflake, Azure, Presto and BigQuery. <https://fivetran.com/blog/warehouse-benchmark>, 2018. Accessed 14.05.2020.
- [33] Spanner: Google's Globally-Distributed Database. <https://static.googleusercontent.com/media/research.google.com/no//archive/spanner-osdi2012.pdf>, 2012. Accessed 17.05.2020.