Hermann Mørkrid

Replacing Elasticsearch in a Data Analytics Platform

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg (NTNU)
Co-supervisor: Valdemar Edvard Sandal Rolfsen (Ignite)
February 2024

Master's thesis

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

NTNU
Norwegian University of Science and Technology

ignite

Hermann Mørkrid

Replacing Elasticsearch in a Data Analytics Platform

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg (NTNU)
Co-supervisor: Valdemar Edvard Sandal Rolfsen (Ignite)
February 2024

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

Norwegian University of
Science and Technology

Acknowledgements

I would like to thank Valdemar Rolfsen and Erik Bøe of Ignite Procurement AS, for their invaluable technical expertise and guidance provided throughout my work on the thesis. I would also like to thank my supervisor from NTNU, Svein Erik Bratsberg, for his counseling and friendly conversations.

Abstract

This thesis compares two analytical databases, Elasticsearch and ClickHouse, for the use case of building a generic data analytics platform. We delve into the design of Elasticsearch, and how it is used in the case of Ignite, a platform for analyzing procurement data. The thesis presents a set of challenges faced by Ignite in their use of Elasticsearch, and then explore potential alternatives to it, choosing ClickHouse as the database to compare further.

We then set up an experiment to compare the two databases, implementing a generic HTTP service as an abstraction layer to compare them equally. A set of benchmarks are performed for this service, finding that ClickHouse outperforms Elasticsearch in data ingestion, but that it performs worse at our specific query execution, though this finding has its limitations. In addition, a set of qualitative findings are presented, describing the challenge of achieving correctness in results from Elasticsearch, and the issue of "object-columnar impedance mismatch" for ClickHouse.

The thesis concludes that ClickHouse is a viable alternative to Elasticsearch for the use case of a generic data analytics platform, but that the mixed results and limitations of the experiment make it not the obvious choice.

Sammendrag

Denne avhandlingen sammenligner to analytiske databaser, Elasticsearch og ClickHouse, til formålet av å bygge en generisk plattform for dataanalyse. Vi gjør et dypdykk i hvordan Elasticsearch er designet, og hvordan databasen brukes i Ignite, en plattform for å analysere innkjøpsdata. Avhandlingen presenterer et sett med utfordringer som Ignite har støtt på i deres erfaring med Elasticsearch, og utforsker potensielle alternativer. ClickHouse velges som database til å sammenligne videre.

Vi setter så opp et eksperiment for å sammenligne de to databasene, og implementerer en generisk HTTP-tjeneste som et abstraksjonslag for å sammenligne dem likt. Et sett med ytelsesmålinger blir så utført for denne tjenesten, og finner at ClickHouse-databasen utpresterer Elasticsearch i datainntak, men at den yter verre i utføring av vår spesifikke spørring. I tillegg presenteres et sett med kvalitative funn, som beskriver utfordringen med å oppnå nøyaktighet i resultater fra Elasticsearch, og utfordringen med å forene objekt-orienterte modeller med den kolonne-orienterte strukturen til ClickHouse.

Avhandlingen konkluderer med at ClickHouse er et levedyktig alternativ til Elasticsearch for en generisk dataanalyse-plattform, men at de blandede resultatene og begrensningene i eksperimentet gjør det til et ikke åpenbart valg.

Contents

1 Introduction **1**
1.1 Research Goal 2
1.2 Thesis Structure 2

2 Background **3**
2.1 Elasticsearch 3
2.1.1 Distributed 3
2.1.2 Document Store 4
2.1.3 Search and Analytics 4
2.1.4 The Elastic Stack 5
2.2 The Ignite Data Analytics Platform 7
2.2.1 Ignite's Elasticsearch Architecture 7
2.2.2 The Data Management System 8
2.2.3 Ignite's Analysis Service 10
2.3 Key Challenges With Elasticsearch 14
2.3.1 Time to Index 14
2.3.2 Incorrectness in Aggregation Results 15
2.3.3 Memory Use 16
2.3.4 Configuration and Maintenance 16
2.4 Other Analytical Databases 16
2.4.1 Apache Cassandra 17
2.4.2 ScyllaDB 18
2.4.3 Bigtable 19
2.4.4 ClickHouse 20

3 Design and Implementation of Experiment **22**
3.1 The Choice of ClickHouse 23
3.2 Choice of Programming Language 24
3.3 System Architecture 26
3.4 API of the Analysis Service 28
3.4.1 Table Schema Format 29
3.4.2 Query Format 30
3.5 Internal Structure of the Analysis Service 35
3.5.1 The `api` package 36
3.5.2 The `csv` package 36
3.5.3 The `db` package 37
3.5.4 The `clickhouse` package 38

3.5.5 The `elasticsearch` package 39
3.5.6 The `config` package 40
3.6 Experiment Methodology 40
3.6.1 Reproducing Benchmark Results 41

4 Results and Discussion **43**
4.1 Performance 43
4.2 Correctness 44
4.3 Developer Ergonomics 46
4.3.1 Object-Columnar Impedance Mismatch 46
4.3.2 Non-Descriptive Error Messages 47
4.4 Limitations of the Experiment 48

5 Conclusion **50**
5.1 Further Work 51

References **52**

A Benchmark Results **56**

Chapter 1

Introduction

In a world where companies increasingly digitalize their operations, the amount of data available to them is ever-increasing. In this context, there is a growing demand for tools to help companies understand and analyze their own data. Hiring dedicated software engineers can be expensive, so to fill this demand, "software-as-a-service" companies have emerged as general solutions to the digital requirements of companies. But such companies face new challenges of scalability: their service must no longer just handle the data of a single company, but of all its customers, the combined data of which can be truly massive. This puts further demands on the databases used by software-as-a-service platforms.

One such software-as-a-service platform is the Ignite data analytics platform, developed by Ignite Procurement AS. It provides tools for companies and organizations to manage and analyze their data on spending, contracts, suppliers, carbon emissions and so forth. A central part of the platform's architecture is the Elasticsearch database. It provides a lot of functionality for Ignite, but as we shall see, it also presents its own set of challenges. Although the Ignite platform handles these challenges well enough, the required workarounds are less than ideal. Elasticsearch is only one of a multitude of different analytical databases available today, and so it is interesting to examine if an alternative database might better serve a data analytics platform such as Ignite.

In this thesis, we explore how Elasticsearch works as a database, how it is used in the Ignite platform, and the landscape of alternative analytical databases. We then go on to design an experiment comparing Elasticsearch with one such alternative database, ClickHouse. From this experiment, we examine both quantitative and qualitative results, to compare how the databases fit the use case of a data analytics platform like Ignite's.

1.1 Research Goal

The research goal of the thesis is to examine if an alternative analytical database can provide similar capabilities to Elasticsearch for a data analytics platform, while alleviating its performance and correctness challenges.

1.2 Thesis Structure

Chapter 2 covers the background for the thesis, which includes how Elasticsearch is used both generally and by Ignite, key challenges with it, and potential alternatives. Chapter 3 goes on to detail how the thesis experiment was implemented, and the reasons behind choices made. Chapter 4 then discusses both the quantitative and qualitative results of the experiment, referencing benchmarks from Appendix A. Finally, chapter 5 concludes the thesis, revisiting the research goal and suggesting further work.

The implementation of the thesis experiment has been made open-source in agreement with Ignite, released under the MIT license. The source code is available at `https://github.com/hermannm/analysis`. Where appropriate, the thesis links directly to source code in this repository to provide context when discussing the implementation.

Finally, the thesis uses the term *database* as equivalent to a *database management system* (DBMS). A distinction is typically made between these two when discussing the internals of a database system. But when developing an application that uses a database system, it is more common to just refer to the whole system as "the database". The distinction between a database and DBMS has not been relevant in this thesis, since it is more concerned with the application layer, and thus, we just use the term database.

Chapter 2

Background

This chapter aims to explain the background for the thesis, providing context for the experiment detailed in the next chapter. Section 2.1 describes the design of Elasticsearch as a database, and the context in which it is generally used. Section 2.2 then describes the architecture of the Ignite data analytics platform, and how Elasticsearch fits into it. Next, section 2.3 presents key challenges faced by Ignite in their use of Elasticsearch. Finally, section 2.4 explores potential alternatives to Elasticsearch, describing what sets them apart from other databases.

Sections 2.1, 2.2 and 2.3 are based on the specialization project prior to the master's thesis [1].

2.1 Elasticsearch

Elasticsearch is a **distributed document store** built for **search and analytics** [2] [3]. The following sections aim to explain Elasticsearch by going in-depth on each of these terms. Subsequent sections describe the broader context in which Elasticsearch is used.

2.1.1 Distributed

Elasticsearch distributes data and query load across a *cluster of nodes* (i.e. servers). The number of nodes per cluster, called the *cluster capacity*, is configurable, and may grow and shrink over the lifetime of the system. When nodes are added or removed, Elasticsearch performs *rebalancing*: migrating data between nodes to ensure continued availability and redundancy [4].

An *index* is the primary unit of data organization in Elasticsearch, similar to the concept of a table in traditional databases. Each document (i.e. row) in an index is replicated across multiple *shards*. These shards are distributed across different nodes in the cluster [4]. This gives redundancy: if a node with an index's shard goes down, the index can get the data from a shard on a different node.

Similarly to nodes, the number of shards is also configurable, and can grow and

shrink while the system is running. The number of primary shards in an index is fixed when it is created, but the number of replica shards can be changed at any time. The size of each shard can also be configured. There are important tradeoffs to consider here: larger shards are more expensive to move around when rebalancing the cluster, but a larger number of smaller shards makes it more expensive to maintain indices, as the index must replicate data in more physical locations [4].

In addition to shard replication, Elasticsearch also offers replication of entire clusters through *cross-cluster replication*. This automatically synchronizes indices from one cluster, called the *primary cluster*, to a secondary cluster. Typically, these clusters will be far physically removed from each other, so that if disaster strikes at one cluster site, the other can take over. The secondary cluster can be used as a *hot backup* in this way, but can also be used to improve geographic colocation with users, by serving read requests of users that are physically closer to it than the primary cluster. However, the primary cluster still handles all write requests to its indices, which are called the *active leader indices* [4].

2.1.2 Document Store

Elasticsearch may be called a *NoSQL* database, i.e. not a traditional relational database management system queried through SQL. It eschews rows and columns in favor of *documents*, in JSON format [3]. As described in section 2.1.1, data in Elasticsearch is organized into indices, and so the process of adding documents is called *indexing*.

Documents in Elasticsearch may or may not have a defined schema. The process of defining a schema for documents is called *mapping*, and has several variations. *Explicit mapping* is when the user explicitly defines the data type for each field name in the document. *Dynamic mapping* allows documents to be indexed without specifying how to handle each field – instead, Elasticsearch can automatically detect and add new fields to the index, and attempt to deduce the field type from the data. The way in which Elasticsearch deduces data types from documents can be configured through *dynamic field mapping rules* [5].

Getting the correct data type for fields, whether through explicit or dynamic mapping, enables Elasticsearch to do data-specific optimizations. Elasticsearch indexes every field in documents it receives, using a data structure that depends on the field type. For text fields, an inverted index is most efficient, but if Elasticsearch knows that the field is numeric, it can instead use an even more efficient *BKD tree* [3].

2.1.3 Search and Analytics

Although Elasticsearch is a distributed database in its own right, it was not really built to serve as a primary database. Rather, the main use case of Elasticsearch is to serve as an *engine* for search and analytics. The reason that every document field is indexed, as described in section 2.1.2, is to optimize for fast full-text search [3]. For example, the inverted index used for text fields allows search queries to quickly find all documents containing a given word.

Search queries take different forms in Elasticsearch. *Structured queries* are similar to SQL, allowing matching and sorting on specific fields. *Full-text queries* instead aim to find matches to a given query string, and sort them by relevance. *Complex queries* combine these two categories – for example, to find query string matches on a specific field. A number of additional queries are available for specific data types [6].

The main query language of Elasticsearch is the custom *Query DSL* [7]. Similarly to the documents themselves, DSL queries are submitted as JSON – in fact, the query text goes through the same analysis that documents go through during indexing [3].

Beyond just searching, Elasticsearch also offers *aggregations*: summaries of data based on given aggregation parameters. *Metric* aggregations calculate some number result, such as an average, based on field values. *Bucket* aggregations, meanwhile, group documents into buckets based on values. Aggregations can also be *pipelined*, so the result of one aggregation can be fed into another one. Additionally, the Query DSL allows the user to perform a query along with an aggregation, which makes the aggregation only apply to results of the query [8].

Apache Lucene

Elasticsearch builds on top of the *Apache Lucene* (hereby Lucene) search engine library. Lucene provides a variety of different full-text search algorithms [9]. Each node in an Elasticsearch cluster runs Lucene, executing search algorithms on stored documents when it receives queries. In this sense, Elasticsearch can be viewed as an API and distributed architecture that wraps Lucene. The actual search implementation is handled by Lucene, while Elasticsearch takes care of the distribution of data and the processing of queries.

Lucene is an internal component of Elasticsearch, and so users do not directly interact with it through the API. However, the abstraction can be seen to leak in *Kibana*, which is used as a control center for managing an Elasticsearch cluster [4]. It offers a *Lucene query syntax* [10] in order to utilize Lucene's full feature set for searches.

2.1.4 The Elastic Stack

Looking at Elasticsearch in isolation is not sufficient to fully describe how it is used. Elasticsearch is typically used in conjunction with other tools. One set of such tools is referred to as the *Elastic Stack*, which includes *Beats*, *Logstash* and *Kibana*. The following sections will describe these components, to give a more complete picture of the context in which Elasticsearch is used.

Beats

Beats is a term used in the Elastic stack for a variety of different *data shippers*. Their role is to capture data, and then send it on to the rest of the stack. Different *Beats* exist for different types of data: *Filebeat* for log files, *Heartbeat* for availability data, *Packetbeat* for network traffic, and more. Beats can either

send data directly to Elasticsearch to be indexed, or through Logstash for processing on the way.

Logstash

As described in section 2.1.3, Elasticsearch is not meant to serve as a primary database on its own. This logically entails that the data indexed in Elasticsearch is coming from some other data source. This could for example be another database, or the API of some other service.

Logstash exists to serve as a middle man between Elasticsearch and sources of data to be indexed. It does this through pipelined processing: a Logstash instance has a number of pipelines, running in parallel, each of which process incoming data through a series of steps. The first step is receiving or collecting data from the ingested data source, through *input plugins*. This data is placed on a queue, where it is picked up by a set of *filter plugins*, which parse, process and enrich the data. Finally, the data moves to the *output plugin*, which formats the data and forwards it – to Elasticsearch, when used as part of the Elastic stack. This whole process is called *ingesting* data [11].

Kibana

As mentioned in section 2.1.3, Kibana is a control center for managing the cluster in Elasticsearch. Moreover, it provides an interface for analyzing and visualizing the data in the cluster [12]. Whereas the structured Query DSL of Elasticsearch is great for applications, Kibana provides administrators and analysts with a more ergonomic interface for exploring data interactively.

On the database workload scale of *online transaction processing* (OLTP) vs. *online analytical processing* (OLAP), we can place the workloads supported by Kibana firmly on the OLAP side. However, Elasticsearch as a whole, with its focus on search and analytics, is also weighted towards OLAP. This illustrates how OLTP-OLAP is a sliding scale. In a way, Kibana *extends* the OLAP support of Elasticsearch to provide even better tools for detailed analysis of the data.

The Stack

When viewed together, Elasticsearch, Beats, Logstash and Kibana can be seen as steps in a data flow. Data is first captured by Beats, then filtered through Logstash, then indexed and stored in Elasticsearch, then finally analyzed and visualized with Kibana. Not all systems will use all these together, but most systems using Elasticsearch will have a similar-looking flow: component(s) on the *ingest* side which provide data to be indexed in Elasticsearch, and component(s) on the *consume* side to analyze the data. Figure 2.1 shows an illustration of this architecture.

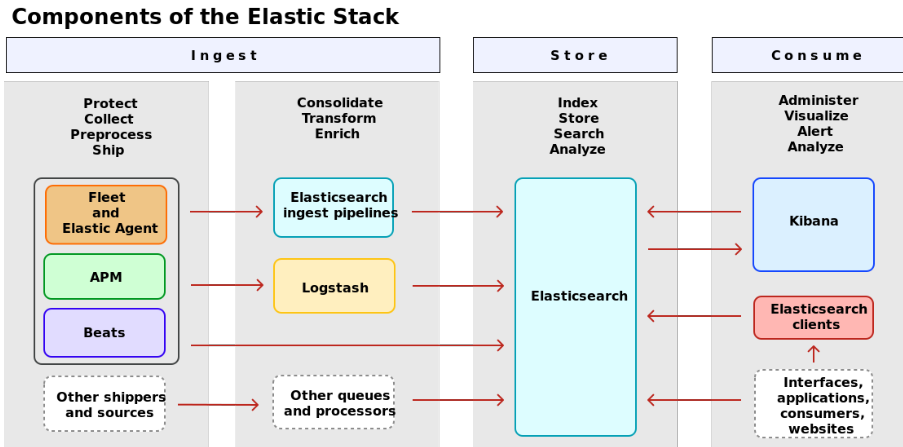


Figure 2.1: Illustration of the Elastic Stack [13]

2.2 The Ignite Data Analytics Platform

Ignite Procurement (hereby “Ignite”) is a software company that develops the Ignite Procurement platform (hereby “the Ignite platform”), which allows companies to manage and analyze their data on spending, contracts, suppliers, carbon emissions and so forth. Elasticsearch plays a central part in the architecture of the Ignite platform.

The findings in these sections are based on conversations with the Chief Technical Officer (Valdemar Rolfsen, personal communication, Mar. 13, 2023) and a software engineer from Ignite (Erik Bøe, personal communication, May 3, 2023), as well as my own experience working as a developer of the Ignite platform for a year.

2.2.1 Ignite’s Elasticsearch Architecture

To understand the architecture of the Ignite platform, one must first understand microservices. Microservice architecture is defined by splitting a monolithic server application into separate applications, running on separate servers, and communicating with each other over the network. Ignite runs a plethora of microservices for different modules of the platform: one for contracts, one for suppliers, one for file management, et cetera. These services communicate over *gRPC*, an efficient communication and serialization protocol. Users access the platform through a web application, which sends all requests to a “gateway” service, which in turn forwards requests to other services as appropriate.

One of Ignite’s microservices is the Data Management System (DMS). It is built as an abstraction layer on top of a MongoDB database, and provides an API for other parts of the platform to insert and manage data. This is the primary database for users’ data – thus, the first step for new users of the Ignite platform is to get their data into the DMS.

The DMS is also one of the components in Ignite’s architecture that connects

to Ignite's Elasticsearch instance. All the data that is inserted into the DMS is *also* indexed into Elasticsearch. The purpose of this is to make analytical queries faster: when indexing data from the DMS into Elasticsearch, the data is transformed to make analytical queries on it more efficient (more on this in section 2.2.2). In addition, Elasticsearch's built-in functionality around search and aggregation provides greater support for analytical queries than that of MongoDB.

Another one of Ignite's microservices that interacts with Elasticsearch is the Analysis service. Whereas the DMS takes care of inserting data into Elasticsearch, Ignite's Analysis service provides an API for querying this data from Elasticsearch. It builds on the existing query API of Elasticsearch, and extends it with additional data aggregation capabilities, primarily through Ignite's Pivot library (further explained in section 2.2.3).

As we take a broad look at these components interacting with Elasticsearch in the Ignite platform, we see similarities to the Elastic Stack, as described in section 2.1.4. To illustrate this, Figure 2.2 shows the different components in Ignite's architecture surrounding Elasticsearch, in the same way as Figure 2.1 did for the Elastic Stack. On the *ingest* side, we have the process of getting users' data into the DMS, transforming it to optimize for analysis, and then indexing it into Elasticsearch. On the *consume* side, we have the Analysis service and its API, which is used by other services in Ignite's microservice architecture.

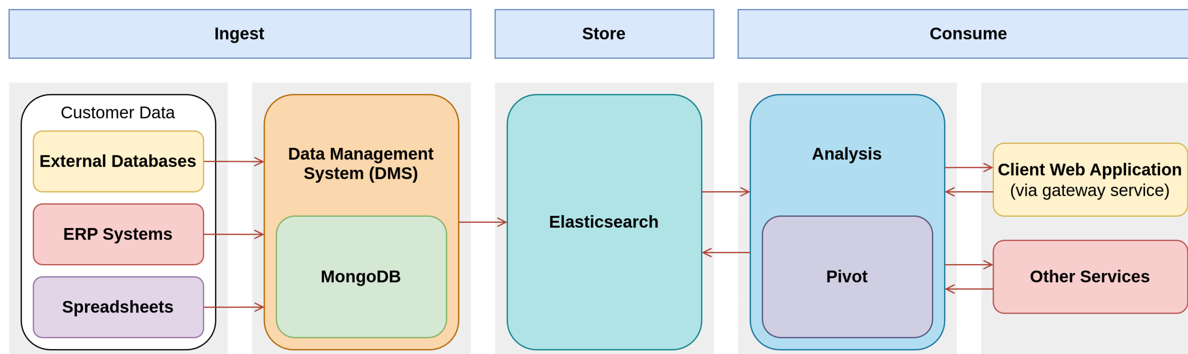


Figure 2.2: Diagram of Ignite's architecture surrounding Elasticsearch

The following sections will further explain these various components in Ignite's architecture.

2.2.2 The Data Management System

As previously mentioned, Ignite's Data Management System (DMS) is responsible for indexing data into Elasticsearch, placing it on the *ingest* side of the architecture. This process consists of two steps: first, the DMS needs to ingest data itself, then it can go on to index this data into Elasticsearch.

Data ingestion in the Ignite platform starts by building a *data source*. This typically consists of uploading a file, with multiple different formats supported, such as Excel spreadsheets or CSV ("comma-separated values") files. After

building a data source, users next define a *data table*, in which users describe the structure of their data to make it more useful in the Ignite platform. This process can also involve intermediate transformations of the original data – for example, renaming a field called `s_no` in the original data to the more descriptive `Supplier number`. These transformations form a *pipeline*, leading from a data source to the new data table.

Fields in the data table also have corresponding data types, i.e. text, number, et cetera. The DMS can infer data types from values in the original data, but the user may sometimes want to change this. For example, a number in the original data may in fact refer to an ID in a different data set. In this case, once the user has set up tables for the two data sets, they can mark a field in the first table as a reference to a field in the second table, creating a relation between the two.

Once users have ingested their data into a data source and built a data table, the DMS can index this data into Elasticsearch. This in turn allows modules of the platform to use the analytical query capabilities of Elasticsearch to provide analytical views of the data. The DMS makes a number of transformations on the data it sends to Elasticsearch, in order to optimize it for analytical queries:

- **Materialized relations.** Data tables that are connected to each other through references have the data in their corresponding related documents fetched and stored in-place. For example, a *spend* table with data on expenses, and a supplier ID reference for each expense, would fetch the actual supplier object with that ID from the supplier table and store that. By doing this, later analytical queries on the table and its relation do not need to join the two tables.
- **Flattening empty data.** Any "null-ish" values are not indexed. Since the DMS uses JSON as its format, such values include `null`, `false` and empty strings.
- **Keyword indexing and normalization.** All text fields are indexed with the *keyword* Elasticsearch mapping, and have a *normalizer* applied to them. A keyword normalizer in Elasticsearch ensures that variations in case and accents on keywords resolve to the same keyword. Ignite uses an upper-case normalizer, which means that e.g. "Supplier, inc." and "supplier, INC." both resolve to "SUPPLIER, INC.".

The DMS – Showing the Limits of NoSQL?

With the DMS's data sources, tables and types, we see that Ignite has essentially built its own database paradigm on top of MongoDB. This may be indicative of the limitations of NoSQL databases. MongoDB is a document-oriented NoSQL database, which essentially allows the storing of any type of JSON document, without caring for its structure. This flexibility is one of MongoDB's strengths, but the lack of structure and corresponding type safety can create a need to build structure on top of it. This is in contrast to traditional relational databases, which require data to follow a strict schema, but in turn offer more powerful capabilities when querying and analyzing data. Thus, one may argue that the

lack of strict structure provided by MongoDB makes it less useful for data analysis.

MongoDB may however be the perfect fit for Ignite's use case, precisely because of the duality between flexibility and structure of data. Each of Ignite's users brings their own data source, with its own structure, fields and types. It is valuable to be able to ingest and store this data in its original form, as it can then be later retrieved as-is. Here, the flexible nature of MongoDB is an advantage, as it allows the storage of data sources in their raw form. When it comes time to define structure for the data, Ignite's pipeline builder provides an intuitive interface for data transformation. If a user later decides to change a pipeline, updating the data is simply a matter of running the original data source through the updated pipeline. In a relational database, this would typically involve a data migration through SQL scripts, which is error-prone and could lead to data loss. By separating the *source* and *structure* of data, the Ignite platform thus provides data integrity.

2.2.3 Ignite's Analysis Service

The Analysis service is Ignite's query interface between Elasticsearch and their other services. Whenever some other module in the platform needs to make an analytical query on user-provided data, it goes through the Analysis service (note: some services still query Elasticsearch directly, but Ignite's goal is that all these should eventually use Analysis). One may ask why such a service would be necessary, when Elasticsearch already provides a query API. The reasons are twofold: first, to provide a more ergonomic and human-readable query language, and second, to better control *how* Elasticsearch is queried.

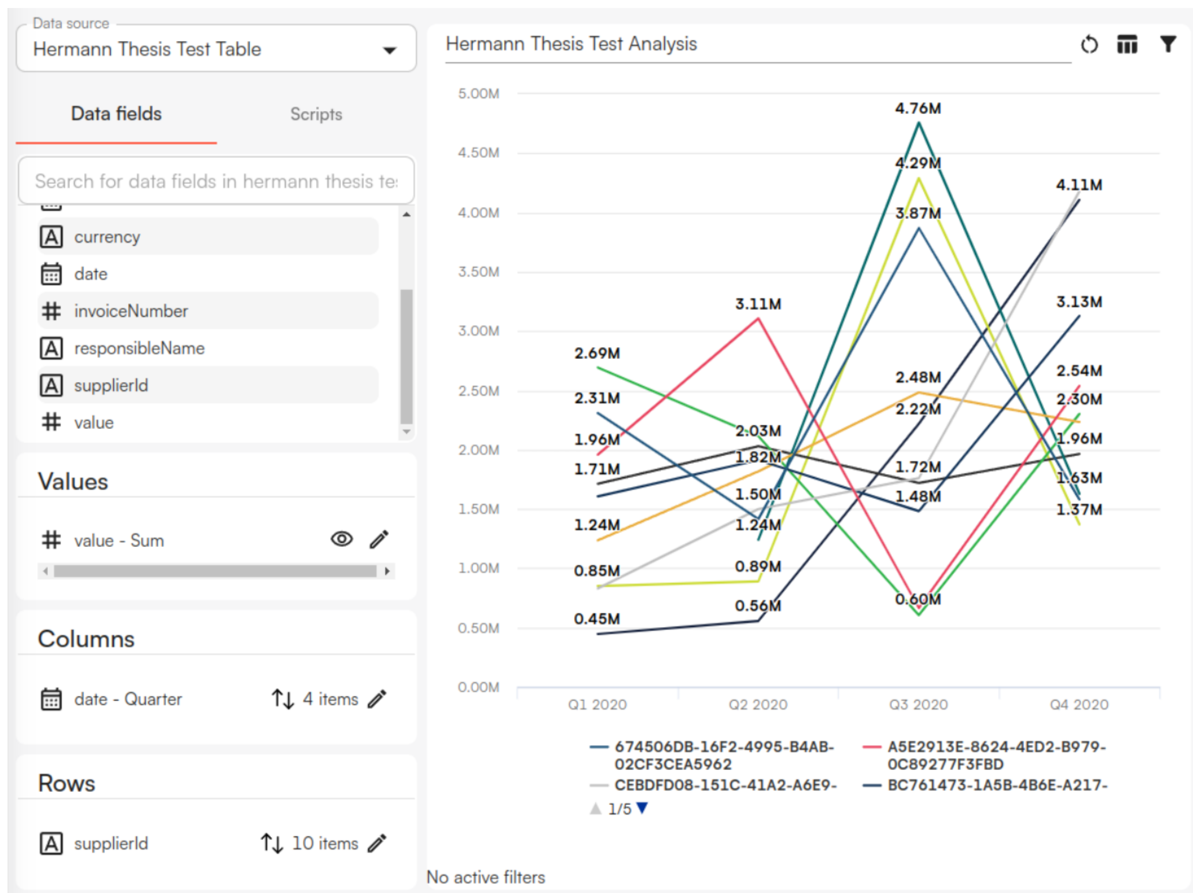


Figure 2.3: Screenshot of a custom analysis made in the Ignite platform.

One of the modules of the Ignite platform uses the Analysis service to enable users to build custom graphs based on their data. Figure 2.3 shows an example of this. The graph is based on a CSV test data file, where each row represents a hypothetical transaction made between a company and one of its suppliers. The fields of the data, as shown on the left in the figure, include the currency the transaction was made in, the date it was completed, the invoice number, the name of the person responsible for the transaction, an ID for the supplier, and the transaction's value (i.e. how much was spent). Having built a data table from this data source, we can now analyze it. As we see in the bottom left of the figure, we choose the `date` field as columns, splitting it into quarters. We use the `supplierId` field for rows, limited to 10, and then sum the `value` of transactions. This gives us a graph of the top 10 suppliers for each of the last 4 quarters, by the sum of our spend for each supplier in each quarter.

The Pivot Query Format

In order to support the type of queries needed to build custom analyses like the one in Figure 2.3, Ignite's Analysis service has its own query format called *Pivot*.

Pivot is based on the concept of "pivot tables", which originates from spreadsheet software in the late 1980s [16]. A pivot table is a way to display data from a data source – whether that be a different table, or a database – where rows of the original data have been grouped together and aggregated. This process is called "pivoting" [17], as it involves moving rows to columns, or columns to rows, to view the data from different "angles".

Similarly to Elasticsearch's Query DSL, a Pivot query uses JSON as its format. It consists of three fields: an array of *value aggregations*, an array of *row splits* and an array of *column splits*, all of which are JSON objects. A *split* in this case is similar to a `GROUP BY` clause in traditional SQL – a way to group data together based on some field, before performing aggregation. One can create "nested splits" by including multiple objects in the split arrays – for example, doing a column split on a date field for both years and quarters.

Listing 2.1 shows the generated Pivot query for the graph in Figure 2.3. The value aggregation, row split and column split objects correspond to the *Values*, *Columns* and *Rows* sections of the graph interface, respectively. We see that each object contains metadata about the queried fields, that the value aggregation specifies `"aggregation": "sum"`, and that the splits specify sort order, size and interval.

Listing 2.1: Pivot query for the graph in Figure 2.3.


```
{
  "value_aggregation_items": [
    {
      "id": 21942,
      "type": "float",
      "field": "data_column_248",
      "aggregation": "sum",
      "cumulation": null,
      "window_size": null,
      "script": null,
      "index": 0,
      "filters": [],
      "visible": true,
      "data_column": null,
      "group_type": null,
      "uuid": "db7ce857-09e1-4f60-b591-d6d7ec149e91"
    }
  ],
  "row_split_items": [
    {
      "filters": [],
      "aggregation": "cardinality",
      "id": 18804,
      "direction": "row",
      "index": 0,
      "analysis_widget": 12328,
      "largest_field_widget": null,
      "supplier_source_configuration": null,
```

```
      "procurement_lever": null,
      "field": "data_column_252",
      "type": "keyword",
      "size": 10,
      "search_size": null,
      "exclude_others": true,
      "interval": null,
      "intervals": null,
      "unique": true,
      "sort_order": "desc",
      "sort_agg_index": 0,
      "data_column": null,
      "group_type": null,
      "aggregated_filters": [],
      "uuid": "dc88ee25-f787-457a-b2c2-165dda2116cd"
    }
  ],
  "column_split_items": [
    {
      "filters": [],
      "aggregation": "max",
      "id": 18803,
      "direction": "column",
      "index": 0,
      "analysis_widget": 12328,
      "largest_field_widget": null,
      "supplier_source_configuration": null,
      "procurement_lever": null,
      "field": "data_column_249",
      "type": "date",
      "size": 4,
      "search_size": null,
      "exclude_others": false,
      "interval": "quarter",
      "intervals": [],
      "unique": true,
      "sort_order": "asc",
      "sort_agg_index": null,
      "data_column": null,
      "group_type": null,
      "aggregated_filters": [],
      "uuid": "3efcaf3d-5e09-4784-b947-7aa8d15390f3"
    }
  ]
}
```

Another part of the query language of Ignite's Analysis service is *filters*. These are additional query objects passed alongside the Pivot query, which filter the data before it is aggregated. Filter queries support a variety of different operations: filtering on the presence or absence of specific field values, filtering on a range of

values, filtering on a search term, et cetera. These filters can be nested together in a *conditional* filter, which chains filters together with either `AND` or `OR`.

Query Translation and Execution

The parsing, translation and execution of Pivot queries is done by Ignite's Pivot library, written in C++. When the Analysis service receives a query, it calls Pivot, which then translates the given query to equivalent Elasticsearch queries. A single Pivot query may produce multiple Elasticsearch queries in order to satisfy the semantics of the original query.

The Pivot library also alleviates some of the inherent inaccuracy in results from certain queries in Elasticsearch. Section 2.3.2 will go into more detail on this.

2.3 Key Challenges With Elasticsearch

As we have seen, Elasticsearch plays a central part in the architecture of the Ignite platform. However, Ignite has had multiple challenges with adapting Elasticsearch for their use case. The following sections explore these challenges, and how Ignite has taken measures to alleviate them.

2.3.1 Time to Index

One of the key problems with Ignite's use of Elasticsearch is the time it takes to re-index updated data. When a customer makes an update in some module of the platform, that change propagates through to the Data Management System (DMS). It updates its MongoDB database, and then triggers re-indexing of Elasticsearch.

The issue here is that re-indexing is a heavy operation. Elasticsearch does not support partial updates of indices – this is a fundamental part of its design – so this process involves a lot more data than just the relevant update. This only becomes worse with larger indices: the more documents in the index, and the more fields on those documents, the longer it takes to re-index.

The effect of this delay is *stale reads*. When a user makes an update in the Ignite web application, they may navigate to a module that uses this data. If that module makes an analytical query that goes to Elasticsearch, and the re-indexing process has not completed, then the user will see a previous version of the data, without their update. This can lead to user confusion and frustration, and diminish the usability of the platform by essentially requiring users to leave and come back later to view their updates. Even worse, the user may not realize that the data analysis is based on old data, and use this flawed view of the data as the basis for a decision.

One of the ways that Ignite tries to alleviate this is by reducing the amount of data in Elasticsearch, so there is less to re-index. We noted some of these techniques in section 2.2.2, such as not indexing empty data, and only indexing fields that the user has marked for data analysis. However, some of the other techniques that Ignite employs when ingesting data into Elasticsearch, namely

materializing relations, work to the contrary: they increase the amount of data to index. All in all, this is a critical challenge for the Ignite platform today.

2.3.2 Incorrectness in Aggregation Results

One of the issues that can be encountered when using Elasticsearch is the tradeoff between performance and correctness (i.e. the accuracy of query results). Many of Elasticsearch's APIs do not guarantee that the returned results are completely correct, and may only give approximations. For example, the *Terms aggregation*, a type of query that places data into buckets for further aggregation, may produce inaccurate results [18]. This is because Elasticsearch only looks at a certain number of records on each shard to answer the query, in order to increase performance. This can produce incorrect results, depending on how the data is distributed across shards. Say that we want to aggregate some data for the top 10 suppliers in some transaction data we have stored in Elasticsearch. The `size` of our query will then be 10, and Elasticsearch then chooses a default `shardSize`, i.e. the number of records to examine from each shard, of `size * 1.5 + 10` [18], i.e. 25. But if one of the shards contains data for a top 10 supplier that falls outside those 25 on a shard, it will not be included, which may produce wrong aggregation results, and even the wrong suppliers being included in the top 10.

The Ignite platform must contend with the issue of correctness, as its users expect correctness in custom analysis results. This issue emerges in particular for queries that combine aggregation with sorting, such as the "top 10 suppliers" query exemplified above. For such a query, Ignite's Pivot library first makes a naive query against Elasticsearch to get the top 10 suppliers, but knowing that the results may be incorrect due to records falling outside the query's shard size. Then, it makes another Elasticsearch query that filters on just those top 10 suppliers, to guarantee correct aggregated values for them. Now, Pivot can compare the aggregation results for the first, inaccurate query to the results of the second, correct query. If the difference between these values is above a certain threshold, that indicates that there was a large error in the original query, and that the results may in fact contain the wrong top 10 suppliers. In this case, Pivot repeats the query, looking at the top 20 suppliers next, and if the error margins are still too large, goes on to look at the top 30, 40 or even 50 until the error margin is low enough. This technically does not *completely* guarantee correctness, but it provides a good enough guarantee in practice.

This process of retrying queries to achieve correctness is rather complicated. It reflects a dissonance between a data analytics platform that requires correctness, and a database that mostly provides approximations by default. This increases the maintenance burden of the Ignite platform, as it must essentially maintain a "correctness layer" on top of Elasticsearch, in the form of the Pivot library. And although Ignite's solution performs well enough in practice, making multiple round-trips to the database in order to increase correctness is not ideal for performance. These issues suggest that a database that is designed for correctness by default, while still maintaining acceptable performance, might be a better solution.

2.3.3 Memory Use

Databases are heavy systems, and typically have high memory requirements. This is especially true when they want to deliver higher performance, as it is much more efficient to fetch data from memory than from disk. Elasticsearch does this a lot, which gives it a high baseline memory consumption. In addition, Elasticsearch is built in Java, running on the Java Virtual Machine (JVM), and thus uses garbage collection instead of manual memory management. While this is generally a safer approach, as it avoids memory safety vulnerabilities, it comes at a significant performance overhead. Memory is not freed until the JVM's garbage collector has checked that it is unused, which typically happens at a delay after it actually became unused. This further contributes to the memory usage of Elasticsearch.

For Ignite, this high memory use comes at a cost. Their Elasticsearch instance has a baseline memory consumption of 500 GiB in production [19]. This in turn leads to a higher hosting bill for Ignite, which is not ideal.

2.3.4 Configuration and Maintenance

Elasticsearch is a complex system, with a lot of configuration options. Ignite has 1000's of lines of configuration files for Elasticsearch, which can be hard to debug. This in turn makes it harder to maintain Elasticsearch as part of the technology stack, as it requires intimate knowledge of Elasticsearch to understand all this configuration.

2.4 Other Analytical Databases

As we have seen, Ignite has experienced several different issues with their use of Elasticsearch. Thus, it is useful to look at what alternative databases exist. Such an alternative would have to fulfill certain criteria:

1. It must provide the same capabilities for analytical queries that Elasticsearch does, i.e., its query language must be just as flexible and powerful, or at least enough to match how Ignite already uses it.
2. The database's analytical query capabilities must also include support for aggregate queries, and ideally give better correctness guarantees than Elasticsearch, to avoid the need for Ignite's current process of retrying queries to achieve correctness (see section 2.3.2).
3. It must allow for the mass ingestion of data in an efficient manner, ideally more performant than Elasticsearch, in order to reduce the time required to index new data (see section 2.3.1).
4. It must be deployable in the cloud, to be able to integrate with the rest of Ignite's microservice architecture.
5. It should ideally be more memory efficient than Elasticsearch, to alleviate Ignite's issues with high memory consumption from their Elasticsearch instance. Thus, it would be relevant to investigate alternatives written in more memory-efficient programming languages than Java.

6. It should ideally require less configuration than the thousands of lines that Ignite currently maintains for Elasticsearch (see section 2.3.4).
7. It should ideally be cheaper to run in production than Elasticsearch, or at least in the same price range. An alternative that performs marginally better than Elasticsearch, but costs ten times more, would not actually be a solution for Ignite.

These criteria restrict the options that should be investigated for alternative databases. In the following sections, we shall explore various databases designed for analytical workloads, and their characteristics in relation to the above criteria.

2.4.1 Apache Cassandra

Cassandra is a distributed, wide-column, NoSQL database management system, built by Facebook in 2008 for their Inbox Search feature [20]. Its main design goal is to ensure reliability and scalability in a massive distributed environment. Like Elasticsearch, it is written in Java, running on the Java Virtual Machine (JVM).

Facebook found that traditional relational database management systems did not sufficiently meet their requirements for scalability, as they handle network partitions (i.e. nodes in a database cluster going offline) poorly. This is the problem described in Brewer's conjecture, also known as the CAP theorem: that a distributed system may only choose two of the attributes consistency, availability and partition tolerance [21]. In Facebook's case, traditional databases' prioritization of consistency was insufficient for their workloads.

Data in Cassandra is organized into tables with rows and columns, though it does not use a relational model [20]. Row keys are strings, provided by the user when inserting data. All data is indexed on this key. Columns can be grouped together into column families, which can be further grouped together to form a *super* column family. Keys and column names are used to interact with Cassandra's API, which consists of the following operations:

- `insert(table, key, rowMutation)`
- `get(table, key, columnName)`
- `delete(table, key, columnName)`

The `columnName` can either be the name of a column family, a super column family, a column within a column family (on the format `column_family:column`) or a column within a super column (on the format `column_family:super_column:column`) [20, p. 36].

Read and write requests go to any node in a Cassandra cluster [20, p. 36]. The receiving node finds the replicas that hold the data for the requested key. For writes, the request is then forwarded to all replicas that hold data for that key, and then a quorum is performed with all those nodes to confirm successful writes. Reads, meanwhile, are routed to the closest replica for efficiency. Cassandra may be configured to make reads function like writes, using a quorum with all replicas, for a stronger consistency guarantee.

Data is partitioned between nodes by using *consistent hashing* [20, p. 36]. Each node is given responsibility for a range of values in the possible outputs from the hash function - this node is called the *coordinator* of keys in that range. It is responsible for replicating its data across $N - 1$ other nodes in the cluster, where N is a configurable replication factor. When a request is received for a given key, that key is hashed with the same hash function to determine which node it should be assigned to.

For local persistence on each node, Cassandra uses write-ahead-logging (WAL), an in-memory data structure and file system disk persistence [20, p. 38]. Every change is first written to the commit log, and if successful, the change is then persisted in the in-memory data structure. Once this structure reaches a certain size threshold, the data is flushed to a file on disk. When a read request is received, it first queries the in-memory data structure, then files on disk from newest to oldest. As time goes on, multiple files will accumulate on disk - to alleviate this, a background process runs *compaction* periodically, i.e. merging files into one.

Since the initial paper on Cassandra, several developments have been made to the database, the most significant of which is the introduction of the Cassandra Query Language (CQL) [22]. It provides an SQL-like syntax for data definition and manipulation in Cassandra.

2.4.2 ScyllaDB

To understand why ScyllaDB exists, one must first understand the context of Apache Cassandra, as Scylla was created as a response to some key challenges with Cassandra. In their whitepaper on Cassandra's shortcomings [23], the creators of Scylla identify four main issues:

1. **Inefficient utilization** of the resources offered by modern, multi-core computers. Production deployments of Cassandra are recommended to limit storage capacity to 1-2 terabytes per node in a cluster [24] – which can waste a lot of capacity when modern cloud computers offer up to 60 terabytes of storage [25].
2. **Unpredictable and unbounded latency** due to Cassandra being written in Java. Java manages memory with garbage collection, which is typically safer and more ergonomic to work with, but sacrifices some performance that could have been gained through manual memory management.
3. **Team intensive**, as operating Cassandra at scale is a big task which requires dedicated experts. This is exacerbated by issue 1, since limited storage capacity per node leads more nodes in a cluster, which makes the cluster harder to manage, which puts further requirements on the team to operate it.
4. **Manual tuning** is often required to manage clusters, as well as the impacts of expensive storage compaction and garbage collection. One example of this is the balance between "background" operations, such as storage compaction and data streaming between nodes, and "foreground" operations that respond to queries. All these operations typically require disk I/O, which can cause contention between them and thus impact response times.

To alleviate this, Cassandra allows setting a cap on background operations – but tuning this right to your database workload is difficult to get right, and can cause even further slow-downs if one gets it wrong.

ScyllaDB aims to solve these issues. It is the same type of database as Cassandra, i.e. distributed and wide-column, using a NoSQL data format. In fact, Scylla aims to be compatible with the API of Cassandra, meaning that it uses the same Cassandra Query Language (CQL) [26]. However, Scylla does a number of things differently from Cassandra to improve its performance [23]:

- **Written in C++** – ScyllaDB is written in C++, which grants precise control over lower-level details such as memory management. This allows performance optimizations that are essentially impossible in Java.
- **User-space I/O scheduling** – Scylla implements its own disk I/O scheduler, instead of letting the operating system kernel handle it. This allows Scylla to automatically prioritize and balance different types of operations that require I/O, which avoids the challenge of having to manually tune this (see issue 4 above).
- **Unified cache** – Cassandra has its own in-memory cache, but falls back to the OS file system cache. Having multiple layers of cache is not ideal, as they may interact in unexpected ways, and puts further pressures on manually tuning the database. To alleviate this, Scylla implements a single, unified cache, which automatically tunes itself to observed workloads.

The result of these techniques is that ScyllaDB can withstand 5x greater traffic than Cassandra, with lower latency for every type of workload, both read-heavy and write-heavy [23, p. 7]. It completes administration tasks 2.5 to 4 times faster than Cassandra, and can achieve the same performance on a smaller cluster that is 2.5 times less expensive. These performance gains, in addition to the auto-tuning features of Scylla that make it easier to operate, could make it a better alternative to Elasticsearch than Cassandra.

2.4.3 Bigtable

Bigtable is a "distributed storage system for managing structured data" [27, p. 205], built by Google. Its main design goals are wide applicability, scalability, high performance and high availability.

Bigtable's data model resembles that of Apache Cassandra (see section 2.4.1): it consists of tables, which map row keys, column keys and timestamps to uninterpreted arrays of bytes [27, p. 206]. These byte arrays may themselves have structure, e.g. JSON blobs, though this is all left in the hands of applications. Row and column keys are arbitrary strings. Similarly to Cassandra, column keys are grouped together into *column families*, and are accessed with the same `column_family:column` syntax.

The client API of Bigtable is also similar to Cassandra's. It supports basic operations to write or delete values in a row, look up individual row values, or iterate over a subset of a table's data [27, p. 207]. Additionally, Bigtable supports the use of a scripting language developed by Google called *Sawzall*, which executes on Bigtable's servers and allows data transformation, filtering and

aggregation. Finally, Bigtable can also be integrated with Apache MapReduce, a framework for big data processing.

Storage in Bigtable uses the Google File System [27, p. 207]. Data is stored in files using Google's *SSTable* format: an ordered, immutable map from keys to values, where both are arbitrary arrays of bytes. Like Cassandra, Bigtable uses a write-ahead commit log, and first stores writes in-memory. When the in-memory data reaches a certain threshold, it is written to an SSTable on disk. A background compaction process runs periodically to merge SSTable files.

Tables are divided into *tablets*, which each hold a range of rows – these are distributed across nodes in a Bigtable cluster. A master node makes periodic health checks on other nodes in the cluster, and reassigns tablets from nodes that go down, to ensure availability. The cluster is coordinated through the use of the *Chubby* distributed lock service, which runs the Paxos algorithm to enforce consistency between nodes.

Rows in Bigtable are sorted alphabetically by key, which allows users to optimize for data locality, by formatting their row keys in such a way that rows which are frequently accessed together are also stored together. In the original Bigtable paper, they present an example table which stores data for crawled web pages, where the row keys are reversed domains [27, p. 206] – i.e. `com.google.maps` instead of `maps.google.com`. This places rows from similar domains together, which improves data locality for more efficient fetching. Users of Bigtable can also enable compression of SSTables, which is further helped by data locality, since similar data close to each other compresses better. This allows Bigtable to achieve up to 10-to-1 space reduction [27, p. 211].

2.4.4 ClickHouse

ClickHouse is a relatively new OLAP database, first launched in 2012, and made open-source in 2016 [28]. It was initially developed by Yandex, an internet service company from Russia, but the development team was spun out into its own company in 2021 [29]. ClickHouse was made as an answer to the data processing challenges faced by Yandex, which required exceptional performance. Thus, the database has been designed with data processing performance in mind from the beginning.

The main distinguishing feature of ClickHouse is its column-oriented storage [30]. This typically performs better than row-oriented storage for analytical query workloads, which often consists of aggregating data together from columns, since all the data needed for an aggregation is stored together. This in turn allows for optimizations such as vectorization (i.e. SIMD instructions). Additionally, column-oriented storage also enables better data compression, since a column often has similar data across rows, and storing these values together allows for better compaction. This in turn improves the scalability of the database, as database nodes can store more data before reaching their limit and requiring more nodes to be deployed. A further consequence of this is reduced operating cost, as fewer nodes deployed is generally cheaper.

Another contributing factor to ClickHouse's performance is its implementation

language: C++. As we have seen, Elasticsearch's Java implementation can lead to outsize memory consumption. A lower-level language such as C++ provides greater control of memory use, which is critical for an analytical database that may have to handle large amounts of data in memory.

The query language of ClickHouse is SQL. It supports most of the ANSI SQL standard, though it is not fully compliant with all of it due to its design [31]. ClickHouse also offers additional aggregate functions beyond those found in most other relational databases, and *combinators* that change how an aggregate function processes its data [32]. Thus, ClickHouse's SQL is its own dialect, like most other SQL-based databases.

Finally, ClickHouse is designed to be deployed as a distributed system. Multiple ClickHouse nodes can be deployed in a cluster, using a centralized system for coordinating and synchronizing distributed operations between nodes. Initially, Apache ZooKeeper was used as this central system [33]. However, the resource efficiency of ZooKeeper's Java implementation was found to be inadequate, so ClickHouse developed its own solution: ClickHouse Keeper, a drop-in replacement for ZooKeeper, but implemented in C++. This brought a 46x reduction in memory consumption compared to ZooKeeper [33], further demonstrating the efficiency gains that are possible with a lower-level language.

Chapter 3

Design and Implementation of Experiment

As explained in section 1.1, the research goal of this thesis is examine potential alternatives to Elasticsearch for a data analytics platform. In section 2.3, we saw the challenges that Ignite has faced with Elasticsearch in building their own such platform, so we now know better the issues that a potential alternative should solve.

In order to fully evaluate an alternative database, we design an experiment that aims to emulate the data analytics platform use case. For this, an HTTP server has been implemented, providing a generic analytical query API to clients, which abstracts over a backing analytical database. This implementation is adapted to support two databases: Elasticsearch and ClickHouse. The server can be configured to connect to either database, while the generic API stays the same. This allows us to run test queries against the server configured for Elasticsearch, then reconfigure it to use ClickHouse and run the exact same queries again, after which we can compare the performance of both databases.

The name chosen for the implementation is "the Analysis service", after the service of the same name from the Ignite platform (see section 2.2.3). Our implementation will be referred to as the canonical "Analysis service" (or "our Analysis service" where appropriate to disambiguate), whereas Ignite's will be referred to as "Ignite's Analysis service". Our service fulfills much of the same role as that of Ignite's, providing a generic API over a backing database, though mine takes on additional responsibilities. Mainly, our service can be seen to also consume the role of Ignite's Data Management System (see section 2.2.2), in that it also provides data ingestion as part of its API. Thus, our service sits at both the "Ingest" and "Consume" sides of the model of the Elastic Stack (see Figure 2.1), while either Elasticsearch or ClickHouse fulfills the "Store" role. This was done to keep the implementation limited to a single service, as a multi-service architecture would have complicated the system for no real reason in this case.

Although the purpose of the Analysis service is to support tests to compare

Elasticsearch and ClickHouse in this thesis, an attempt has been made to design it to be generic and extensible as well. For example, while our service only supports data ingestion from CSV files, it has defined data ingestion interfaces in such a way that support for other formats can be added relatively easily (more on this in section 3.5.2). The main reason for this is to make the implementation more realistic, as one would not design a real service like this to only support a few specific test cases.

The following sections will go into detail on the design and implementation of the Analysis service, as well as the experiment to compare Elasticsearch and ClickHouse. Section 3.1 will explain why ClickHouse was chosen as the alternative database to evaluate against Elasticsearch. Section 3.2 will explain my choice of programming language to implement the service. Section 3.3 will show the overall system architecture of the implementation, i.e. how the different major components of the system interact. Section 3.4 will document and explain the API of the service, which includes both data ingestion and analytical queries. Section 3.5 will examine the internals of how the Analysis service is implemented, by explaining the structure of the codebase and the choices made in the implementation. Finally, section 3.6 will show the methodology behind the experiment using the Analysis service to compare Elasticsearch and ClickHouse.

3.1 The Choice of ClickHouse

As mentioned in the chapter introduction, the database chosen to compare with Elasticsearch in the thesis experiment is ClickHouse. The following paragraphs will explain the motivations for this choice.

Performance. As described in section 2.4.4, ClickHouse was designed from the beginning with performance in mind. Its column-oriented storage model, data compression and C++ implementation may help to solve the performance and memory use challenges of Elasticsearch (see section 2.3). The performance of a system is ultimately dependent on the context in which it is used, but since ClickHouse is designed for analytical workloads, it looks at a first glance like it should fit the use case of a data analytics platform.

Proven record. ClickHouse is already emerging as a potential viable alternative to Elasticsearch. For example, Uber (the ridesharing company) migrated its log analytics platform from the ELK stack (Elasticsearch + Logstash + Kibana, as described in section 2.1.4) to ClickHouse in 2021 [34]. One of the reasons cited by Uber for this migration was hardware cost, mainly due to the overhead of indexing fields in Elasticsearch. As described in section 2.3.1, the cost of indexing is one of the issues Ignite has also had with Elasticsearch. Uber also describes performance issues with aggregation queries, as Elasticsearch is "not designed to support fast aggregations across large datasets". Since Ignite essentially only uses the aggregation part of Elastic's API, this is another shared motivation for exploring alternatives. Uber's new ClickHouse solution "reduced the hardware cost of the platform by more than half compared to the ELK stack", and "the ingestion latency is capped under one minute". This shows that ClickHouse is a

promising alternative for the data analytics platform use case.

Other success stories with ClickHouse include Cloudflare, one of the biggest content delivery networks in the world, which migrated their HTTP Analytics service to ClickHouse in 2018 [35]. Their new solution "improved API throughput and latency" and is "easier to operate", both of which are relevant in solving the challenges described in section 2.3. Sentry, a performance monitoring and error tracking service, also built their new search infrastructure around ClickHouse in 2019 [36]. They evaluated a number of other analytical databases, but chose ClickHouse because it's "operationally simple", and "Data can be queried as soon as it is written in real time" - i.e. mitigating the problem of stale reads described in section 2.3.1. These cases, albeit anecdotal, further suggest ClickHouse as an appropriate solution for analytical use cases.

SQL. Another reason for choosing ClickHouse as an alternative is its use of SQL as its query language (see section 2.4.4). The choice of query language does not directly affect the challenges presented in section 2.3, but it has a great impact on the experience for the developer working with the database. A part of the thesis experiment was to evaluate the developer ergonomics of working with the different databases, as will be discussed in section 4.3. In this aspect, it is interesting to compare Elasticsearch's custom JSON-based API, to the more traditional SQL format offered by ClickHouse. Thus, ClickHouse's use of SQL was another motivation for choosing it.

3.2 Choice of Programming Language

The chosen programming language for the implementation of the thesis experiment is Go. Go is a relatively new language, developed by Google and released in 2009 [37]. It is a compiled, statically typed, garbage-collected language, placing it around the same level of abstraction as Java and C#. What separates Go from those languages is that it compiles to native machine code in a static binary, as opposed to a bytecode format that requires an installed runtime (the JVM for Java, or the CLR for C#). This has made Go popular in "cloud-native" development, as it is typically easier to deploy a static binary to a web server than an application that requires a runtime to also be installed.

The following paragraphs list the reasons why Go specifically was chosen for the Analysis service.

Familiarity. The microservice architecture of the Ignite Platform (see section 2.2) features multiple different programming languages, and one of the most used ones is Go. In particular, Ignite's Analysis service (see section 2.2.3) is written in Go, which is the service that my own implementation is most inspired by. When I previously worked on the Ignite platform myself, I mostly developed microservices in Go, and so I have experience with how to structure such services.

Although Ignite's Analysis service is written in Go, it depends heavily on Ignite's *Pivot* library, which is written in C++ (as mentioned in section 2.2.3). This library does most of the more complex interactions with Elasticsearch

in the Ignite platform, so that might argue in favor of using C++ for my own service as well. However, my experience with C++ is minimal compared to my experience with Go, so this would drastically slow down development speed. Additionally, C++'s manual memory management increases the risk of introducing security vulnerabilities (for example, it accounts for 70% of Microsoft's yearly vulnerabilities [38]). This risk would be compounded by my lack of experience with the language, and so a memory-safe language such as Go is more appropriate in this regard.

Performance. Go's garbage collector means that it will not match the maximal performance of the likes of C and C++. However, being statically typed and compiled to machine code makes it significantly faster than dynamic, interpreted languages such as Python or JavaScript. In addition, Go's semantics for constructing objects on the stack, as well as the ability to keep more data within a single allocation, can reduce pressure on the garbage collector and thus out-perform similar languages such as Java [39].

Furthermore, Go's native support for lightweight concurrency makes it a natural fit for web server development. To achieve scalability, most web servers will use multithreading in some way in order to process multiple incoming requests at the same time. The problem with threads provided by the operating system is that they are relatively heavy-weight: each thread is allocated around 2 megabytes of memory for its stack [40]. This limits the amount of threads that a web server can run simultaneously, which restricts how many requests it can serve at once. Go's solution to the scalability issue of OS threads is to implement its own kind of lightweight threads in the language's runtime, called *goroutines*. Go runs a number of OS threads equal to the machine's number of CPU cores [41], and multiplexes goroutines onto these threads to allow for parallelism. Each goroutine starts with just 2 kilobytes of memory for its stack [42] – 1000x less than the default for OS threads – and whenever a goroutine makes an IO call, it yields to the runtime so other goroutines can execute while it waits for the IO operation to complete. This allows a server to run millions of goroutines concurrently, thus providing the scalability that high-traffic web servers require.

Since the service implemented for the experiment is a web server designed to be a part of a micro-service architecture, with support for high traffic, Go's performance characteristics and concurrency support makes the language a good fit.

Explicit error handling. The most recognizable part of typical Go code is its error handling. Go embraces "errors as values" as opposed to the exception and try/catch model seen in most other modern programming languages. This takes the form of functions returning multiple values, and if it can fail, the final return value is an error. For example, take the `GetTableSchema` function from the `AnalysisDB` interface (declared in `db/db.go`):

```
GetTableSchema(
    ctx context.Context,
    table string
) (TableSchema, error)
```

This function signature signals that it will either succeed and return a `TableSchema`, or fail and return an error (types are placed after variable names in Go). When calling this function, one can check the returned error to see if it failed. For example, in `api/schema.go`:

```
schema, err := api.db.GetTableSchema(req.Context(), table)
if err != nil {
    sendServerError(res, err, "failed to get table schema")
    return
}
```

You will see this pattern repeated across the codebase of the Analysis service. Some consider these repeated checks for `if err != nil` overly verbose and tedious, but the benefit is that it makes it very clear where code may fail. Thrown exceptions, on the other hand, make errors implicit: any function may throw an exception, and your only way of knowing is to check its documentation, if it exists. In implementing the Analysis service, this explicit error handling was found to be quite helpful, as one often encounters errors when developing against a database. This was another reason for choosing Go as the implementation language.

3.3 System Architecture

To get an overview of how the Analysis service is implemented for the thesis experiment, this section will explain the overarching architecture of the system.

The Analysis service can be configured to use either Elasticsearch or ClickHouse. At startup, the program reads a configuration file to see which database to connect to. It then establishes connections to the appropriate database, expecting the database to already be running. The service then exposes an HTTP API, which consumers can send requests to. The API is generic and independent from the specific database the service is connected to, so it works the same whether configured for Elasticsearch or ClickHouse. Section 3.4 details the format of this API.

Figure 3.1 models the various components in the architecture around the Analysis service. The “API Consumer” in the diagram can be anything making requests to the Analysis service, but in a microservice architecture it would typically be another application server that uses this service to store or analyze data as part of some larger operation.

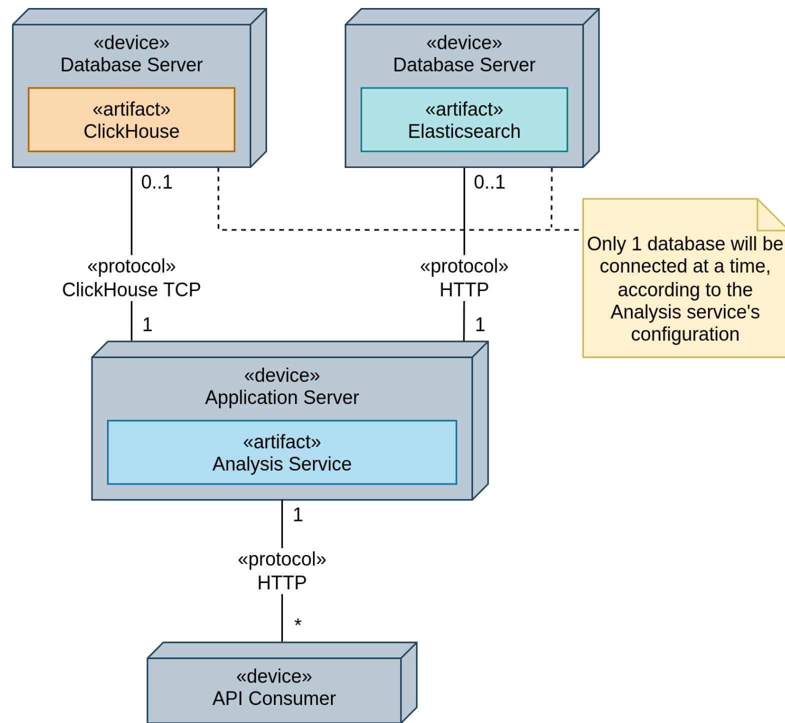


Figure 3.1: Deployment diagram of the Analysis service and its database connections.

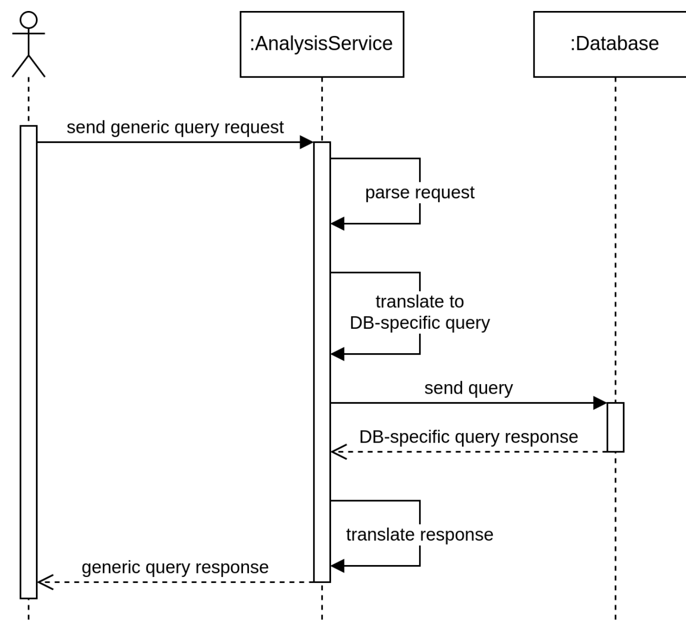


Figure 3.2: Sequence diagram following a request's path from an API consumer through the Analysis service and its connected database. *DB* stands for database.

When the Analysis service receives an API request, it parses the request body into its expected format, and then translates it into a corresponding query for its configured database. Once a response is received from the database, it is then parsed into a generic response format, and passed back to the consumer. If the request had an invalid format, or the database failed to process the query, an error is returned instead. Figure 3.2 models a request's path through the Analysis service (though only looking at the happy path, assuming a valid request and successsful processing by the database).

3.4 API of the Analysis Service

The Analysis Service interacts with the outside world by exposing an HTTP interface. What this means in practice is that the service has a set of *endpoints* defined, i.e. sub-routes for whichever URL the service is hosted on. For example, when running the service locally on `localhost:8000`, the `/run-query` endpoint is accessed by sending an HTTP request to `localhost:8000/run-query`. Each endpoint expects input from either the request body, typically encoded as JSON, and/or query parameters.

This type of API differs from the gRPC protocol used between microservices in the Ignite platform (see section 2.2.1). This choice was made for practical reasons. One of the benefits of gRPC is its reduced serialization overhead due to its binary format. This is great when it is used to communicate between services, but not as practical when manually interacting with the API as a human. Throughout the development process of the service, being able to manually construct the JSON messages sent to the API made debugging much simpler. Furthermore, Go's native support for this type of API reduced the need for external dependencies. If the Analysis service were to actually be deployed as part of a microservice architecture, it would probably be beneficial to refactor it to use gRPC, but the current form of the API serves as a practical compromise.

The endpoints exposed by the Analysis service are as follows:

- `/create-table-from-csv`: This endpoint serves to let users create a new database table with data from a CSV ("comma-separated values") file. It expects a request body with content type *multipart form-data*, which allows the request to contain both a file upload and additional fields. In this case, it expects a field called `csvFile`, containing a CSV file with the data to ingest. It also expects a field called `tableSchema`, a JSON object that names the fields of the data along with their data types (see section 3.4.1 for more detail on this format). It creates a new table in the database with the fields from the given table schema, and then inserts the data in bulk from the CSV into the created table. On success, it returns an empty HTTP `200 OK` response.
- `/ingest-data-from-csv`: This endpoint is identical to `/create-table-from-csv`, except it ingests data into an existing table (given by the table name in the table schema provided in the request).
- `/deduce-csv-table-schema`: The purpose of this endpoint is to let users automatically deduce the data types and fields contained in their CSV

data. It also expects a multipart form-data request body, with a `csvFile` field. It returns a JSON-encoded table schema with the fields and data types deduced from the values found in the CSV, but with a blank table name.

- `/get-table-schema`: This endpoint returns the table schema of a previously created table. It expects a query parameter named `table`, and returns the JSON-encoded schema for that table, if found.
- `/run-query`: This is the main endpoint offered by the service, once data has been ingested. It expects a query parameter named `table` for the table to run a query on, and a request body with a JSON-encoded *Analysis query* (see section 3.4.2 for more detail on the format) to run against the database. It responds with a JSON-encoded *Analysis result* (more detail on this also in section 3.4.2).

The general usage of this API is to send a CSV file with data to the `/deduce-csv-table-schema` endpoint, then set the table name on the returned table schema. This can then be used, along with the CSV file again, to create a table with a request to the `/create-table-from-csv` endpoint. After that, one can query the table by passing the table name and an appropriately constructed query to the `/run-query` endpoint. If one later has the need to ingest more table data, one can get the table schema again from the `/get-table-schema` endpoint, and pass that along with new CSV data to the `ingest-data-from-csv` endpoint.

The reason for separating the table schema deduction endpoint from the table creation endpoint is to give more flexibility to the API consumer. The schema deduction algorithm is not perfect, and may deduce the wrong data type for certain fields. By separating these operations, the user has the opportunity to fix errors in the schema before creating the table.

3.4.1 Table Schema Format

All of the endpoints of the Analysis service, except the `/run-query` endpoint, either expect a table schema as input or return one as output. Thus, it is central to the service. Its purpose is to provide a common abstraction over the schema formats used by ClickHouse and Elasticsearch, so that clients can use the API in the same way regardless of the backing database.

The Analysis table schema is a JSON object, with the following fields:

- `tableName`: A string with the name of the table. Must be unique.
- `columns`: An array of objects for each column in the table. The fields of each column object are listed below.
  - `name`: A string with the name of the column.
  - `dataType`: Must be one of `TEXT`, `INTEGER`, `FLOAT`, `DATETIME` or `UUID` (Universal Unique Identifier, a common ID format).
  - `optional`: A boolean, representing whether unset values are allowed for fields in the column.

Listing 3.1 shows an example of how a JSON-encoded table schema looks. This is the schema for the test data that was also used for the custom analysis example in the Ignite platform (see section 2.2.3).

When translating a table schema to a query against ClickHouse or Elasticsearch to create the table, these data types are translated into the appropriate data types for the database.

Listing 3.1: Example of the table schema format used by the Analysis service.

```json
{
  "tableName": "supplier_spend",
  "columns": [
    {
      "name": "currency",
      "dataType": "TEXT",
      "optional": false
    },
    {
      "name": "value",
      "dataType": "INTEGER",
      "optional": false
    },
    {
      "name": "date",
      "dataType": "DATETIME",
      "optional": false
    },
    {
      "name": "invoiceNumber",
      "dataType": "INTEGER",
      "optional": false
    },
    {
      "name": "responsibleName",
      "dataType": "TEXT",
      "optional": false
    },
    {
      "name": "supplierId",
      "dataType": "UUID",
      "optional": false
    }
  ]
}
```

3.4.2 Query Format

The query format of the Analysis service is the expected JSON object in the request body passed to the `/run-query` endpoint. It is essentially a simplified version of the Ignite platform's *Pivot* queries (see section 2.2.3).

The Pivot query format has the concept of *value aggregations*: fields to aggregate data on, with a variety of operations supported. Furthermore, a Pivot query has *splits*, i.e. fields used to partition the data before aggregating. Pivot allows an arbitrary number of splits, in both row and column dimensions. In addition, Ignite's Analysis service has the concept of *filters*, a separate query object, with nesting allowed, to filter data before it is queried.

In order to emulate the types of queries that the Ignite platform processes, the query format of the thesis experiment also has *splits*, for both rows and columns. However, the number of splits is limited to one in each dimension, in order to simplify the implementation logic. Queries also only allow a single value aggregation (just called "aggregation" in this format). Finally, our Analysis service does not have the concept of filters, as this would further complicate the implementation, when it's the aggregation of Pivot that is the core of Ignite's queries.

The fields and sub-fields expected from the queries passed to our Analysis service are as follows:

- `aggregation`
  - `kind`: The kind of aggregation to perform. Must be one of `SUM`, `AVERAGE`, `MIN`, `MAX` or `COUNT`.
  - `fieldName`: The name of the field on which to aggregate. Must be present in the queried table.
  - `dataType`: The data type of the queried field. For aggregations, the only supported data types are `INTEGER` and `FLOAT`. While this field is not strictly necessary, as the Analysis service could query the database for the type matching `fieldName`, that would require an extra round-trip to the database before executing the query. It is assumed that clients will typically already have the table schema when making a query, so providing this should be trivial – and if not, they can fetch it from the `/get-table-schema` endpoint. This field is also included in Ignite's Pivot queries.
- `rowSplit`
  - `fieldName`: The name of the field to split the data on.
  - `dataType`: The data type of the queried field. The logic for including this in the query is the same as for the `dataType` field of the aggregation object. However, splits allow any of the data types (i.e. all of `TEXT`, `INTEGER`, `FLOAT`, `DATETIME` and `UUID`).
  - `limit`: The maximum number of results to include for the split. When combined with `sortOrder`, this can be used to get the "top K" results for some query, which is a common type of analytical query.
  - `sortOrder`: The order by which to sort the returned results. Must be either `ASCENDING` or `DESCENDING`.
  - `integerInterval`: Optional, and only applicable to fields of the `INTEGER` data type. If provided, each field value is rounded to the nearest value in the given interval.

  - `floatInterval`: Same as `integerInterval`, except for fields of the `FLOAT` data type. It is kept in a separate field, as it makes it easier for the implementation's JSON parser to deserialize into the appropriate type.
  - `dateInterval`: Optional, and only for splits on `DATETIME` fields. It must be one of `YEAR`, `QUARTER`, `MONTH`, `WEEK` or `DAY`.
- `columnSplit`: The same type of object as the `rowSplit` field.

Listing 3.2 shows an example of how Analysis queries look in practice. The query here is semantically equivalent to the example used for Ignite's Pivot query format (see section 2.2.3): it gets the top 10 suppliers for each of the last 4 quarters, by the sum of our spend for each supplier in each quarter. Similarly to the Pivot query, we see that each object in the query specifies the queried fields, that the aggregation specifies its kind (`"SUM"`), and that the splits specify sort order, limit and interval. The main difference between our format and Pivot is that the top-level fields in the query are singular objects instead of arrays, since as explained above, our implementation only allows a single aggregation and a single split in each dimension.

Listing 3.2: Example of a JSON-encoded Analysis query.

```
{
  "aggregation": {
    "kind": "SUM",
    "fieldName": "value",
    "dataType": "INTEGER"
  },
  "rowSplit": {
    "fieldName": "supplierId",
    "dataType": "UUID",
    "sortOrder": "DESCENDING",
    "limit": 10
  },
  "columnSplit": {
    "fieldName": "date",
    "dataType": "DATETIME",
    "sortOrder": "ASCENDING",
    "limit": 4,
    "dateInterval": "QUARTER"
  }
}
```

The result of an Analysis query is also a JSON object, with the following fields:

- `rows`: An array of objects, each representing a result for the query's row split, with the below specified fields. The number of objects are at most the value of `rowSplit.limit` in the given query.
  - `fieldValue`: This row result's value for the field that was split on (given by `fieldName` in the query). If the split had an interval (`integerInterval`/`floatInterval`/`dateInterval`), the value is

rounded to the closest value in that interval, and includes all records within that interval's range.

  - `aggregationsByColumn`: An array of values resulting from aggregating the query's `aggregation.fieldName` for each column split and this row split, aggregated according to the query's `aggregation.kind`. The array contains one aggregated value for each column split of the query, so the first element is the aggregation for the first column split, the second element for the second column split, et cetera.
  - `aggregationTotal`: The sum of `aggregationsByColumn`.
- `rowsMeta`: The same object as the `rowSplit` field from the query. It provides metadata that is useful to parse the returned results (such as the data type of `fieldValue`).
- `columns`: An array of objects, each with a single field as specified below. The array's length is at most the value of `columnSplit.limit` in the given query.
  - `fieldValue`: Same as `fieldValue` from the row results, but for the column split.
- `columnsMeta`: Similarly to `rowsMeta`, this field copies the `columnSplit` object from the query, to provide metadata about the results.
- `aggregationDataType`: The data type of the aggregated values, useful for parsing the results. Its value is the same as the `aggregation.dataType` field from the query, i.e. `INTEGER` or `FLOAT`.

The `rowsMeta`, `columsMeta` and `aggregationDataType` fields are not strictly necessary to include in the result object, as the client can get this metadata from their original query objects. However, this requires clients to keep their query objects around for parsing results, which makes the API less ergonomic to use. Returning this metadata as part of the results is also more in line with Ignite's Pivot format on which this is based (see section 2.2.3), which indicates that it is useful to return for this use case.

Listing 3.3 shows an example Analysis query result, in this case the result of the query shown in Listing 3.2. We see that 10 rows are returned, as specified in the query, sorted in descending order by the aggregation total for each row. Furthermore, the result includes 4 columns, where each field value is the date at the start of a quarter for that year, since we specified a `dateInterval` in the query. The `aggregationsByColumn` field in each row has 4 elements, one for each column – so the first aggregation value corresponds to the first quarter, the second value to the second quarter, et cetera.

Listing 3.3: Example of a JSON-encoded Analysis query result.

```
{
  "rows": [
   {
     "fieldValue": "674506db-16f2-4995-b4ab-02cf3cea5962",
     "aggregationTotal": 9170496,
     "aggregationsByColumn": [2308458, 1414463, 3868625, 1578950]
```

```
  },
  {
    "fieldValue": "a5e2913e-8624-4ed2-b979-0c89277f3fbd",
    "aggregationTotal": 8268271,
    "aggregationsByColumn": [1958798, 3106917, 664589, 2537967]
  },
  {
    "fieldValue": "cebdfd08-151c-41a2-a6e9-e9d2f0af25a0",
    "aggregationTotal": 8267693,
    "aggregationsByColumn": [829246, 1498596, 1760983, 4178868]
  },
  {
    "fieldValue": "bc761473-1a5b-4b6e-a217-0b4556f2ad65",
    "aggregationTotal": 8125914,
    "aggregationsByColumn": [1606117, 1911257, 1481250, 3127290]
  },
  {
    "fieldValue": "4554370b-692b-465a-9193-35e77f023288",
    "aggregationTotal": 7767498,
    "aggregationsByColumn": [1235287, 1816908, 2482320, 2232983]
  },
  {
    "fieldValue": "cd01ba31-5752-47ec-bcf5-7177f55f700f",
    "aggregationTotal": 7713269,
    "aggregationsByColumn": [2692553, 2114652, 604948, 2301116]
  },
  {
    "fieldValue": "00d0fb1e-6f4b-48d3-829e-b7616179318f",
    "aggregationTotal": 7625860,
    "aggregationsByColumn": [0, 1239843, 4757961, 1628056]
  },
  {
    "fieldValue": "fe0757ce-9503-4ef0-9765-dd1c6f0b7e75",
    "aggregationTotal": 7424424,
    "aggregationsByColumn": [1712395, 2029003, 1719183, 1963843]
  },
  {
    "fieldValue": "d9abd1f0-a9b0-4e13-ac30-00c8ce877e60",
    "aggregationTotal": 7399902,
    "aggregationsByColumn": [850080, 889134, 4290620, 1370068]
  },
  {
    "fieldValue": "b30f9302-3a15-465d-a8f2-34928f8ef111",
    "aggregationTotal": 7326747,
    "aggregationsByColumn": [445084, 555393, 2218789, 4107481]
  }
],
"rowsMeta": {
  "fieldName": "supplierId",
  "dataType": "UUID",
```

```
    "limit": 10,
    "sortOrder": "DESCENDING"
  },
  "columns": [
    { "fieldValue": "2020-01-01T00:00:00Z" },
    { "fieldValue": "2020-04-01T00:00:00Z" },
    { "fieldValue": "2020-07-01T00:00:00Z" },
    { "fieldValue": "2020-10-01T00:00:00Z" }
  ],
  "columnsMeta": {
    "fieldName": "date",
    "dataType": "DATETIME",
    "limit": 4,
    "sortOrder": "ASCENDING",
    "dateInterval": "QUARTER"
  },
  "aggregationDataType": "INTEGER"
}
```

3.5 Internal Structure of the Analysis Service

To understand the internal structure of the Analysis service, it is first useful to understand the structure of source files in the Go programming language. A Go codebase is called a *module*, typically identified by a URL - in our case, `hermannm.dev/analysis` (defined in the file `go.mod`), corresponding to a page on my personal website. Every directory in a module is a *package* in that module. For example, the files in the `db` directory comprise the `db` package, identified by the full path `hermannm.dev/analysis/db`. When other packages want to use this package, they import it by using the full path, and use identifiers from the package by prefixing them with the package name – e.g. `db.AnalysisQuery`.

One important aspect of packages in Go is that they forbid *circular dependencies*, meaning that dependencies between packages in Go only go one way. For example, if the `api` package in the Analysis service imports the `db` package, it would be a compilation error for the `db` package to then import `api`. Thus, a dependency tree of Go packages is a directed acyclic graph. This affects how one structures the codebase, as it effectively forces the developer to place code with mutual dependencies in the same package.

To create an executable Go program, there must exist a package called `main`, with a function called `main` within it that serves as the entry point of the program. In the Analysis service, this package is at the top level, in the `main.go` file. This orchestrates the other packages to run the service. Figure 3.3 shows the structure of these packages, and the relationships between them. The following sections go through each one of the Analysis service's packages and explain their role.

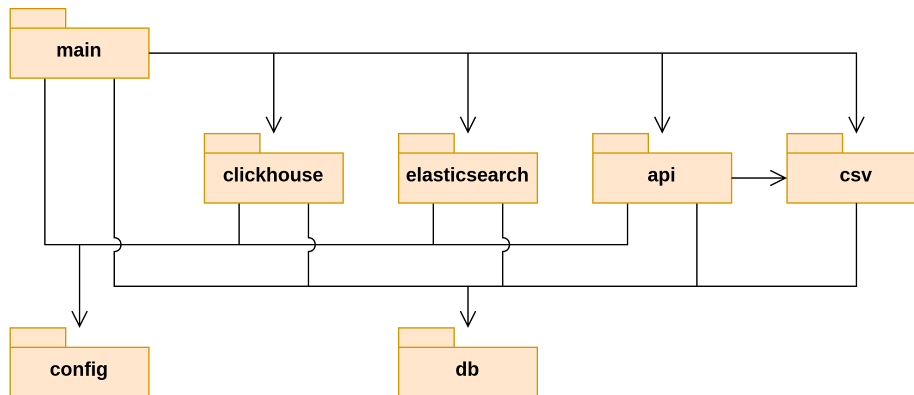


Figure 3.3: Package diagram of the Analysis service. Each arrow represents the source package importing the target package.

3.5.1 The `api` package

The role of the `api` package is to set up and serve the HTTP API described in section 3.4. It uses Go's `net/http` package from the standard library to map each API route to an HTTP handler function (in `api/api.go`). Each handler function parses the HTTP request body into its expected format, and handles it appropriately (implemented in `api/analysis.go`, `api/ingestion.go` and `api/schema.go`). This package contains no database-specific logic, since it uses the common `db.AnalysisDB` interface, which is further detailed below.

The file names `analysis.go`, `ingestion.go` and `schema.go` are reused across packages. This is to signal that these files handle similar concerns, though at different application layers. For example, `api/analysis.go` implements the HTTP handler for running Analysis queries, `db/analysis.go` defines types and utilities used for Analysis queries, and `db/clickhouse/analysis.go` and `db/elasticsearch/analysis.go` implement translation and execution of Analysis queries for ClickHouse and Elasticsearch, respectively. `ingestion.go` files handle data ingestion (i.e. bulk inserts - a common term used in contexts such as Elasticsearch, see section 2.1.4), while `schema.go` files handle table schemas.

3.5.2 The `csv` package

The `csv` package implements CSV parsing. It is a wrapper around the Go standard library package `encoding/csv`, and provides some utilities on top of it. This package is used by `api` in the `/create-table-from-csv`, `/ingest-data-from-csv` and `/deduce-csv-table-schema` endpoints, and by the `main` package in its benchmark tests (further detailed in section 3.6).

First of all, `csv` implements automatic field delimiter deduction. Although CSV stands for "comma-separated values", that separator does not in fact have to be a comma. For example, the test data used for the experiment (see section 3.6) is CSV delimited by semicolon. In order to accept all kinds of CSV files with different delimiters, the Analysis service starts CSV parsing by looking at a

limited number of rows in the beginning of the document, and finds the best match for the field delimiter (implemented in `csv/delimiter.go`).

Secondly, the package implements table schema deduction from CSV files (in `csv/schema.go`). Similarly to field delimiter deduction, it does this by looking at a limited number of rows at the beginning of the document. It tries to parse field values in each row to one of the service's supported data types, falling back to `TEXT` if it was not parsable to any other type.

Finally, the `Reader` type (defined in `csv/reader.go`) implements the `db.DataSource` interface. This allows the CSV reader to be passed to the `db` package as a data source, without that package concerning itself with the fact that it's CSV. This means that if the Analysis service were to be expanded to accept other formats than CSV, one would only have to implement the `db.DataSource` interface for the new format, and it would work the same with all of `db`'s functionality.

3.5.3 The `db` package

The `db` package is the central package of the service, as it is used by every other package (except `config`). The most important part of the package is the `AnalysisDB` interface (defined in `db/db.go`). It represents the abstraction over the different databases implemented in the service. The methods of the interface are implemented for both ClickHouse and Elasticsearch in the `clickhouse` and `elasticsearch` packages (further detailed below). This is what allows the `api` package to implement HTTP handlers that work for both databases, as the `AnalysisAPI` type holds a reference to `AnalysisDB`, without knowing what the underlying implementation is.

`db/schema.go` defines the table schema format as specified in section 3.4.1. It also implements data type deduction from an unknown row, used by the `csv` package in its schema deduction. This allows any future supported input formats other than CSV to reuse this logic from `db`. Furthermore, the file implements conversion of raw input rows to appropriate data types according to a schema, used for the actual data ingestion into the database. Finally, it defines a `StoredTableSchema` type, which is the format used for storing the schema in ClickHouse and Elasticsearch. It turns out that it is easier to represent the stored schema columns as a "struct-of-arrays" rather than an "array-of-structs". This is discussed further in section 4.3.1.

`db/analysis.go` defines the Analysis query and result formats as specified in section 3.4.2. In addition, the file contains utilities for constructing query responses from generic database results, through the `ResultHandle` type. It allows users to iterate over a database result set, and pass results to a `ResultHandle`, which then handles all the logic of validating the results and adding them to the full `AnalysisResult`. The implementations for ClickHouse and Elasticsearch use this to share logic for result parsing, which lets them focus on only the database-specific handling.

Two types are crucial for the generic result parsing of the `db` package: `DBValue` (in `db/db_value.go`) and `AggregatedValues` (in `db/aggregated_values.go`). These solve the problem of type casting correctly when dealing with data types

not known at compile time. The Analysis service encounters this problem since it accepts arbitrary table schemas, and must deal with all different schemas in a generic manner. The solution found in the implementation was to define an interface, i.e. `DBValue`, with methods that are not bound to a specific data type. This interface is then implemented by a generic type, i.e. `dbValue[T]`, which has the actual underlying data type as a type parameter (generic type parameters in Go use square brackets). When calling `NewDBValue`, a `DBValue` is constructed through an appropriate type-parameterized implementation. This gives method implementations access to the generic type parameter `T`, which makes it easier to do type casting and generic operations. In a way, the generic implementations act as "closures" over the type. This pattern helped to greatly reduce verbosity and redundant repeated type-checking when parsing database results.

The remaining files in the `db` package define various enumerated types used in the schema and query formats. `AggregationKind` (in `db/aggregation_kinds.go`) defines the different kinds of aggregations offered for queries, `DataType` (in `db/data_types.go`) defines the supported data types for columns in a schema, `DateInterval` (in `db/date_intervals.go`) defines the possible intervals for splits on `DATETIME` fields, and `SortOrder` (in `db/sort_orders.go`) defines the available rankings for split results. These all share the same structure: a type defined as an `int8` (signed byte), the enumerated values for that type, a map of their string representations, and methods for validation and JSON serialization. These all use my package `hermannm.dev/enumnames` for mapping enumerated values to string names – this package is separate from the Analysis service, as it is one I use across different hobby Go projects. It allows these types to use efficient `int8`s internally in the service, while using more human-readable strings for the JSON API and debugging.

3.5.4 The `clickhouse` package

The key part of the `clickhouse` package is the `ClickHouseDB` type (in `db/clickhouse/clickhouse.go`), which implements the `db.AnalysisDB` interface for ClickHouse. It uses the official `clickhouse-go` database driver package [43] to communicate with ClickHouse.

Table creation (in `db/clickhouse/ingestion.go`) is implemented by translating a given schema into an appropriate `CREATE TABLE` SQL statement. Data ingestion (in the same file) uses the `PrepareBatch` function from the ClickHouse package in order to bulk insert from the given data source until it is read to completion. It uses a batch size of 10 000 rows, which is within the range recommended by ClickHouse's documentation [44].

Storage, fetching and deletion of table schemas is implemented in `db/clickhouse/schema.go`. The schemas are stored in a specified table created when the server starts up (by calling `CreateStoredSchemasTable`), unless it already exists. Storing and fetching converts between the `db.TableSchema` and `db.StoredTableSchema` types mentioned in section 3.5.3.

Analysis query execution is implemented in `db/clickhouse/analysis.go`. It consists of three steps: translating the query to SQL, sending that query to Clickhouse, and then parsing the returned result rows. As we see in `parseAnalysisResultRows`, it uses the `ResultHandle` type explained in section 3.5.3,

thus off-loading most of the result parsing logic from here.

All SQL construction in the package uses the `QueryBuilder` type (implemented in `db/clickhouse/query_builder.go`), a wrapper around Go's standard library string builder. It provides methods for adding query parameters in a safe manner, to avoid SQL injection.

3.5.5 The `elasticsearch` package

Like `clickhouse`, the `elasticsearch` package implements the `db.AnalysisDB` interface through the `ElasticsearchDB` type (in `db/elasticsearch/elasticsearch.go`). It uses the official `go-elasticsearch` client package [45]. `ElasticsearchDB` holds two different clients: a `TypedClient` and an "untyped" `Client`. These provide different ways of interacting with Elasticsearch from Go. The `TypedClient` is more integrated with the Go language, giving better type safety. `Client`, on the other hand, gives more "raw" access to Elasticsearch, and as we shall see, it is required for one of the APIs used in this package.

As explained in section 2.1, Elasticsearch uses slightly different terminology from traditional databases: tables are called *indices*, and schemas are called *mappings*. For consistency with the rest of the codebase and the generic API of the Analysis service, the traditional terminology (i.e. tables/schemas) are used where possible, though Elasticsearch-specific terms leaks through in the APIs used from the `go-elasticsearch` package. For example, `CreateTable` first translates the given generic table schema to Elasticsearch mappings, then creates an index with the given table name and mappings.

Similarly to the ClickHouse implementation, `elasticsearch` implements data ingestion through bulk insert (in `db/elasticsearch/ingestion.go`). It uses the `BulkIndexer` type from Elastic's `esutil` package, which provides a higher level of abstraction than the `clickhouse-go` equivalent. Rather than the developer choosing a batch size and sending batch-by-batch on their own, one adds all records to the `BulkIndexer`, and it internally maintains a number of workers that will flush data to Elasticsearch at certain thresholds [46]. This makes for a more ergonomic API, though it may come at the cost of the developer's control of exactly how bulk insertion is executed. One less ergonomic part of this API is that the `BulkIndexer` requires an `elasticsearch.Client` instance, as opposed to the `elasticsearch.TypedClient` used elsewhere – which is why `ElasticsearchDB` must hold two different client references.

Schema handling (in `db/elasticsearch/schema.go`) uses a dedicated index for storing schemas, with a mapping corresponding to the `StoredTableSchema` type from the `db` package. Store, get and delete operations then operate on this index to persist table schemas.

Finally, Analysis query execution is implemented in `db/elasticsearch/analysis.go`. It first translates the given generic Analysis query to the equivalent Elasticsearch format. This involves creating an *aggregation*, of which there are a great variety of different types, and nesting *sub-aggregations* as appropriate to construct the query. Aggregation types include operations such as sum, average, min and max, but also *bucket aggregations* which essentially function like the `GROUP BY` SQL clause, grouping values into "buckets" that can then be aggregated

separately. For example, Analysis queries that include a row/column split with a `dateInterval` (see section 3.4.2) will use Elastic's `DateHistogramAggregation`, placing values within the same date interval into the same buckets.

After translating and executing the query, the next step is to parse the returned JSON from Elasticsearch. The response format is defined by the `analysisQueryResponse` struct, which is then translated to the generic `AnalysisResult` type using the utilities provided by the `db` package.

3.5.6 The `config` package

The `config` package (in `config/config.go`) defines the available configuration parameters for the Analysis service, and provides a function for reading and validating service configuration from system environment variables. It uses the `joho/godotenv` package to read environment variables from a `.env` file, and the `caarlos0/env` package to parse environment variables into structs in a declarative manner. When starting the Analysis service (in `main.go`), the configuration is passed to the `api` package and the `clickhouse` or `elasticsearch` package (depending on which database is configured) for initialization.

3.6 Experiment Methodology

In order to perform a quantitative comparison between ClickHouse and Elasticsearch, the Analysis service includes a set of automated benchmarks for various aspects of the database interfaces. These are located in the top-level `main` package, in `benchmark_test.go`. The tests use Go's standard library `testing` package, which supports benchmarking [47].

Setup for the benchmarks is done in the `TestMain` function, which reads the service's configuration and initializes the database, much like when launching the Analysis service normally. The difference here is that we do not serve the API, since the benchmarks go directly to the database in order. This avoids the added time of sending requests to the Analysis HTTP API – since this API is the same for ClickHouse and Elasticsearch, it would only add noise to the benchmarks. Once the database is initialized, it sets the global `database` variable, so that it can be used by the benchmark tests. This variable is an instance of the `db.AnalysisDB` interface, which allows our benchmarks to work for both ClickHouse and Elasticsearch, while the underlying implementation goes to the configured database.

The benchmarks use test data from the `test-data.csv` file at the root of the Analysis service repository. This file contains 200 000 rows of hypothetical transactions made between a company and its suppliers, and is the same data as the one used for the custom analysis example in the Ignite platform (see section 2.2.3). The schema of the data is defined by the `testDataColumns` variable, and is equivalent to the example table schema shown in Listing 3.1.

Three different benchmarks are implemented for the Analysis service. `BenchmarkIngestion` sets up a test table and benchmarks the time used for ingesting the test data CSV file. `BenchmarkQuery` tests the time taken for Analysis queries

against the database, using the same query as the example shown in Listing 3.2. Finally, `BenchmarkCreateTable` tests the time used for table creation.

While the benchmarks provide quantitative performance comparisons between ClickHouse and Elasticsearch, another important part of the thesis research goals is to examine the developer ergonomics of working with the two databases. This requires a more qualitative approach. In the process of implementing the Analysis service, I therefore kept a "developer diary". I took notes of observations I made in the development process, especially when it came to issues of data correctness, and APIs that were particularly troublesome to work with. These notes provided concrete examples for describing the developer ergonomics of working with the databases, and form the basis for the qualitative findings to be presented in chapter 4.

3.6.1 Reproducing Benchmark Results

The steps below detail how to set up and run the benchmarks for the Analysis service, in order to reproduce the quantitative results of the thesis experiment.

1. Download, install and run Docker, a tool for standardizing application environments [48], used by the Analysis service to simplify running of ClickHouse and Elasticsearch. One way to install it is through the Docker Desktop application, from https://docs.docker.com/desktop/.

2. Install the Go programming language toolchain (minimum version 1.21.1) from https://go.dev/dl/.

3. Clone the Git repository of the Analysis service:

   ```
   git clone https://github.com/hermannm/analysis.git
   ```

4. Copy the `.env.example` file at the root of the Analysis repository to a new file called `.env`. Change the `DATABASE` field to the database we wish to test (`clickhouse` or `elasticsearch`).

5. Navigate into the Analysis service directory in the terminal.

6. Run the previously chosen database through Docker.

   - For ClickHouse:

     ```
     docker compose up clickhouse
     ```

   - For Elasticsearch:

     ```
     docker compose up elasticsearch
     ```

7. In a new terminal shell, navigate back to the Analysis service directory, and run benchmarks through Go. The `-benchtime` flag allows ut to set the number of iterations to run (by default, the number of iterations is quite low due to the time taken for each iteration in these cases). We use 1000 iterations for the query benchmark, and 100 for the ingestion and table creation tests (since they already run for minutes).

   - To benchmark data ingestion:

```
go test -bench=Ingestion -benchtime=100x
```

- To benchmark Analysis queries:

```
go test -bench=Query -benchtime=1000x
```

- To benchmark table creation:

```
go test -bench=CreateTable -benchtime=100x
```

8. Stop the database:

```
docker compose down
```

9. To test the other database, change the `DATABASE` field in the `.env` file accordingly, and redo from step 6.

Chapter 4

Results and Discussion

This chapter will present and discuss the results of the thesis experiment. Section 4.1 discusses the results of the benchmarks performed for various aspects of the implementations for ClickHouse and Elasticsearch. Section 4.2 goes on to discuss the issue of correctness in query results, and observations made regarding it through the development process of the experiment. Then, section 4.3 discusses the aspect of developer ergonomics from working with the different databases. Finally, section 4.4 presents limitations of the results.

4.1 Performance

Appendix A details the results of running the benchmarks detailed in section 3.6. One of our findings is that ClickHouse ingests data 2.9 times faster than Elasticsearch. This is a promising result for ClickHouse as a potential alternative to Elasticsearch, since it should help to alleviate the "time to index" challenge with Elasticsearch (see section 2.3.1).

In addition, ClickHouse performs table creation 10.7 times faster than Elasticsearch. This is not as important a finding as the ingestion result, as table creation makes up a smaller part of the overall ingestion process. Nevertheless, it suggests that ClickHouse can perform substantially better for common database operations.

Finally, our benchmark of Analysis queries shows that Elasticsearch in fact performs 3.2 times faster than ClickHouse for the same type of query. This result is surprising, as ClickHouse's claimed performance was one of the main reasons for examining it closer (see section 3.1). Revisiting the case of Uber migrating their log analysis platform from Elasticsearch to ClickHouse, we see that Elasticsearch's poor query performance was one of their main reasons for migrating, and that ClickHouse provided a substantial speedup [34]. This suggests that we could expect faster query execution from ClickHouse than Elasticsearch.

Alibaba, the Chinese e-commerce company, provides further reasons to be surprised by our query performance results. In benchmarks performed by

Alibaba's Cloud Database OLAP Product department, ClickHouse out-performs Elasticsearch in a variety of metrics across multiple query types [49]. One crucial difference here is that the data sets used for Alibaba's benchmarks contain hundreds of millions of records, as opposed to the 200 000 records used in our experiment. It may be that ClickHouse's performance advantages only start to show for truly massive datasets. But the stark contrasts between our service's results and those of Alibaba and Uber may also suggest a flaw in our experiment. Section 4.4 will explore some of the possible flaws of our experiment.

One possible explanation for the differences in query performance is that the queries are run with different correctness guarantees. As discussed in section 2.3.2, Elasticsearch queries by default make a tradeoff in favor of performance over complete correctness. In ClickHouse, on the other hand, complete correctness is the default, while opt-in tools are provided to trade accuracy for performance [50]. These opt-in tools were not used in our Analysis service, and so our ClickHouse implementation guarantees correct results, while our Elasticsearch implementation technically could return incorrect results, if we're unlucky with how data is distributed across shards (see section 2.3.2). Guaranteeing correctness for Elasticsearch may require us to implement the same logic for retrying queries as Ignite's Pivot library (also explained in section 2.3.2), which would involve more round-trips to the database, degrading performance. Thus, our query benchmarks may not be an entirely fair comparison, as Elasticsearch is allowed to risk sacrificing correctness in order to optimize query execution, while ClickHouse is not. As we shall see in section 4.2, correctness guarantees are an important part of the difference between the two databases.

4.2 Correctness

The most significant qualitative observation made through the development process of the thesis experiment was the negative impact of working with a database with reduced correctness guarantees. As explained in section 2.3.2, Elasticsearch sacrifices some accuracy in query results in order to improve performance. The consequence of this tradeoff was encountered in the initial implementation of Analysis query execution for Elasticsearch, in which a *bucket sort aggregation* was used. This seemed like the correct choice for the type of query processed by the Analysis service, since it lets us specify sort orders and size for adjacent aggregations in the query (what Elasticsearch calls a *pipeline* aggregation) [51].

This initial query implementation ran without errors, and returned results that appeared reasonable at first glance, as the numbers seemed in line with the results previously observed for the query in the Ignite platform and for our ClickHouse implementation. However, after comparing the results exactly, it became clear that the Elasticsearch query had missed some data. The query in question was the "top 10 suppliers" query shown in Listing 3.2, and the effect of Elasticsearch's inaccuracy was that the results missed one of the top 10 suppliers entirely, and returned the wrong aggregated spend values for several of them, leading to incorrect rankings.

Once the errors in the returned results were discovered, the implementation had

to change. The first attempt was to change `size` and `shardSize` parameters in the query, in order to force Elasticsearch to examine more documents and thus improve correctness. This did change the results, but they still did not match those from Ignite and ClickHouse. When these parameters were adjusted too high, it caused out-of-memory errors in Elasticsearch, and so it became obvious that they were not the solution. After more experimenting, it became clear that it was in fact the bucket sort aggregation that was the real issue. The way that this aggregation type interacted with the other aggregations in the query (namely for the row and column splits) made it almost impossible for Elasticsearch to return correct results, since it would require examining most documents in every shard, which Elasticsearch by design tries not to do (as explained in section 2.3.2).

The solution was to move away from the bucket sort aggregation, and instead use "ordering by a sub-aggregation" [52]. This finally produced correct results for our Analysis query against Elasticsearch, and also significantly improved the performance of the query. However, this solution was not obvious. Firstly, the documentation for this feature was buried deep in a subsection on the documentation page for a general aggregation type. Secondly, its documentation warned that it "generally produces incorrect ordering" [52]. This made it a less obvious choice than the bucket sort aggregation, which had its own dedicated documentation page [51], with examples that appeared to match the semantics of an Analysis query. But in this case, the less obvious choice was the correct one. This illustrates how Elasticsearch's API can make it difficult to find the right choice for a given query, due to the large variety of different aggregation types that interact through nesting and pipelining. When combined with "inaccuracy-by-default", precious developer time may end up being spent on navigating the API to produce correct results.

Another problem with Elasticsearch's reduced correctness guarantees is the issue of *discoverability*. As explained, the inaccuracies in the results from the initial query implementation were only discovered after comparing them with previous results from the same query in the Ignite platform and our ClickHouse implementation. In other words, the errors were only discovered because we already knew the answer to the query. If we had not already had these previous results, the implementation might have been left as-is, and continued to return incorrect results. This would be the normal state of affairs; most applications do not implement their queries for multiple databases, and neither do they typically have an existing platform such as Ignite that they can cross-reference with semantically equivalent queries.

This issue of discoverability can cause problems of incorrectness to arise in non-ideal ways. In a data analytics platform such as Ignite, users may use the results of their custom analyses to make business decisions. If, for example, the "top 10 suppliers" query exemplified in section 2.2.3 were to be used as a basis for which of a company's suppliers to make further deals with, incorrect results can change the decision made, with potentially dire consequences. The user may not know that the underlying database powering their analytics does not in fact guarantee correctness. Ignite's process of retrying queries to improve correctness (see section 2.3.2) was in fact implemented in response to a user complaint about incorrect analysis results (Erik Bøe, personal communication,

May 3, 2023). Such a problem being encountered by a user can weaken the credibility of a data analytics platform, and cause other users to distrust their data in it. Thus, correctness of query results is of great importance to this type of platform, and as we have seen, ClickHouse provides better correctness guarantees than Elasticsearch by default.

4.3 Developer Ergonomics

As explained in section 3.6, a developer diary was kept through the development process of the Analysis service, to record observations on the developer ergonomics of working with ClickHouse and Elasticsearch. The following sections present the key observations made for this qualtitative aspect of the databases.

4.3.1 Object-Columnar Impedance Mismatch

When developing applications with traditional relational databases, a well-known challenge is that of "object-relational impedance mismatch", which is the mismatch encountered between the object-oriented data model of most application software and the relational model of a database [53]. This mismatch requires a mapping between the object-oriented and relational models, costing time and effort for the developer.

In the development of the ClickHouse implementation for our Analysis service, a similar mismatch was observed between the object-oriented structures of the Go program, and the column-oriented storage model of the database. The most concrete example of this is the `StoredTableSchema` type (in `db/schema.go`). This type is equivalent to the `TableSchema` type used in the API of the Analysis service, except it turns the "array of structs" for the schema's columns into a "struct of arrays" for each field of the column. This was implemented in order to fit in with ClickHouse's columnar model.

Wen storing and fetching schemas (in `db/clickhouse/schema.go`), the application has to convert back and forth between the stored representation of the schema and the format used for the API. Although this mapping is not that complex in this example, it nonetheless costs time and effort for the developer, and thus presents a similar mismatch to the object-relational one. We call this the "object-columnar impedance mismatch", representing the mismatch encountered when object-oriented applications interact with *columnar* databases, such as ClickHouse. In a larger application, such as the full extent of the Ignite platform, the time and effort required to address this mismatch could become a greater problem than what we observe in our Analysis service.

One solution to this mismatch is to fully embrace the column-oriented model. In the example of table schemas in the Analysis service, this would involve using the `StoredTableSchema` type for the API as well. Essentially, all data that ends up in the database would use a "struct of arrays" layout. However, exposing this in the API of the service would require all API clients to also adopt this model, essentially moving the mismatch along. This is not necessarily a problem in itself, but it does present a *leaky abstraction*. Part of the point a service such as Analysis is to abstract away the underlying database, and present an intuitive API to clients. The column-oriented model is not the model that most developers

would opt for in traditional object-oriented environments, so the columnar model of the database that the Analysis service abstracts over would "leak" through the API. In a microservice architecture such as that of the Ignite platform, such a decision could permeate throughout multiple services and frontends, and thus leak through multiple layers of the application. A developer that works multiple layers removed from the Analysis service may encounter this object-columnar impedance mismatch, and struggle to work with this model when the motivation behind it is hidden multiple layers away. Thus, this impedance mismatch of ClickHouse presents challenges with abstractions and API design for a data analytics platform.

The object-columnar impedance mismatch may also partly explain the relatively poor query performance we observed for ClickHouse in section 4.1. The `clickhouse-go` client package used to interact with the database partly abstracts away the column-oriented nature ClickHouse, in order to provide row-oriented semantics that are more intuitive to the object-oriented developer [54]. Essentially, this package pays the cost of the object-columnar impedance mismatch, so that the application itself does not need to tackle it. However, ClickHouse also offers the lower-level `ch-go` client package, which provides a real column-oriented interface, avoiding the overhead of pivoting data between row- and column-oriented formats. If the Analysis service used the `ch-go` client package instead of the higher-level alternative, it might have improved performance. However, the overhead of converting between rows and columns could still have manifested itself, just at a different layer of the application, so it is difficult to say if this would make a real difference. Nonetheless, it presents yet another potential issue of the object-columnar impedance mismatch of ClickHouse.

4.3.2 Non-Descriptive Error Messages

One of the reasons for choosing Go as the implementation language for the Analysis service was its explicit error handling, which would help when encountering errors from the database during development (see section 3.2). However, this is less helpful when the error messages returned by the database provide very little information themselves. This was the case for the early development of the Elasticsearch implementation in the Analysis service. On multiple occasions, queries against Elasticsearch failed for one reason or another, but the error messages returned by the database gave no indication of the actual underlying cause of the failure.

In one instance, Elasicsearch returned "all shards failed" – which is essentially the equivalent of "something went wrong". The actual underlying error was that the provided mapping for an index did not actually match the ingested data. In another case, the Elasticsearch client simply returned "EOF" (i.e. end-of-file) when the actual error was unrelated to IO. These are just two anecdotes of an overall trend observed in the initial development process with Elasticsearch: that database errors encountered required a lot of manual debugging, increasing the time taken to resolve them. This differed from the developer experience with the ClickHouse implementation, in which the database mostly provided useful error messages. Although database error messages cannot be perfect in all cases, as the database does not necessarily understand the developer's intention, ClickHouse's superior ergonomics in this case suggested that Elasticsearch could

improve in this area.

Later in the development process, it was discovered that Elasticsearch did in fact provide a way to get more descriptive error messages. Most of Elasticsearch's error responses include a `reason` field, with a longer error message that provides more context for the error. This drastically improved the developer experience of debugging errors from the database, but it was not obvious how to discover this part of the API. The `reason` of an error is not included in its error message by default, and requires the developer to manually extract it. In the Analysis service, custom logic was implemented to handle this (in `db/elasticsearch/errors.go`), requiring type casting of the initial error from Elasticsearch, and inspection of the source code of the official `go-elasticsearch` client package in order to see the types of errors one could really expect. Although this greatly improved the error messages from Elasticsearch, the poor discoverability of this feature and the extra implementation logic required to use it is far from ideal. Compared to ClickHouse, whose errors provided useful context without requiring additional parsing, Elasticsearch's developer ergonomics were poor in this regard. When developing a complex application such as a data analytics platform, this can increase the time taken to fix errors, and thus negatively impact users.

4.4 Limitations of the Experiment

There are multiple limitations to the thesis experiment as implemented, which may help to explain surprising results such as ClickHouse's relatively poor query performance (as detailed in section 4.1). The following paragraphs present such limitations.

Local environment. The benchmarks of the thesis experiment were run on a personal computer (see Appendix A), with ClickHouse and Elasticsearch running through Docker on that same computer. This does not accurately represent the environment that this type of system would run on in the real world. For example, the Ignite platform runs on Google Cloud, deployed with Kubernetes (a system for managing containerized applications [55]). The benchmark results may have differed if they were run in such an environment, as that would typically involve the database running on a separate server from the Analysis service. This would introduce network latency between the application and the database, which might have affected the two databases differently due to differences in formats used across the network. Having said that, services in a cloud architecture are typically deployed close together in a data center, which minimizes network latency. Nonetheless, the possible effects of the environment remain a limitation of the thesis experiment.

Limited data set and query variety. The data set used for the benchmarks was the single `test-data.csv` file, with 200 000 rows of hypothetical supplier spend data. The first limitation of this is that it's not *real* data; the test data was adapted from a data set generated by Ignite for tests of their Data Management System. The distribution of values in this data set may not represent how values are distributed in data from real users. How the data is distributed may affect how the databases perform on it, and so this may make the benchmark results

less applicable. A second limitation is the fact that this was the only data set used. It may be that pecularities of this specific data set are advantageous to one database over the other, thus affecting benchmark results.

Additionally, the query benchmarks were run for just a single type of query (the "top 10 suppliers" query shown in Listing 3.2). Similarly to the lack of variance in data sets, it may be that this specific query is handled better by one database over the other. To achieve more generalizable results, multiple different kinds of queries should be benchmarked.

Chapter 5

Conclusion

The research goal of this thesis was to examine whether a potential alternative can replace Elasticsearch in a data analytics platform. The fact that we were able to implement the same generic analytical query API for both ClickHouse and Elasticsearch, suggests that ClickHouse is at least *viable* as an alternative. Since the Analysis query format used in this experiment mimics that of Ignite's Pivot format, this shows that ClickHouse's analytical query capabilities should be sufficient to serve such a data analytics platform.

Additionally, the performance gains made by our ClickHouse implementation in data ingestion presents it as a candidate for solving the challenge of long indexing times with Elasticsearch, which causes stale data in the Ignite platform. However, ClickHouse's poorer query performance in the benchmarks is a caveat to this, as it may be that Elasticsearch serves these types of queries better. That being said, we have seen multiple limitations and potential problems with this query performance finding, so further work is needed to make a conclusion on this point.

Finally, we have seen the challenge that result correctness poses when using Elasticsearch. First, Ignite's need for a process of retrying queries to improve correctness is already a sign that Elasticsearch's guarantees are insufficient here. Secondly, the observations made during the development process of our Elasticsearch implementation, showing the issues of discoverability and distrust in query results, illustrate the negative impacts of these lacking correctness guarantees. ClickHouse's correct-by-default model, where trading inaccuracy for performance is opt-in, may provide a better foundation for a data analytics platform where result correctness is a priority.

Overall, this thesis finds that ClickHouse may be a suitable alternative to Elasticsearch in a data analytics platform, though the limitations of the experiment and ClickHouse's own challenges mean that it is not the obvious solution.

5.1 Further Work

As discussed in section 4.4, the environment in which the thesis experiment was run may make the benchmark results less valid. Thus, deploying the Analysis service in a real production environment would be useful to see how it affects performance. In addition, one could gather an expanded data sets and produce a greater variety of queries to solve the limitation of the experiment's lack of variety in this aspect.

Next, this experiment only compares two analytical databases. As we saw in section 2.4, there are multiple other potential alternatives. Using the modular design and generic API of the Analysis service, one could implement adapters for more databases for a broader set of comparisons.

Finally, the benchmarks made as part of this experiment are limited in that they only examine the latencies of database operations. As we saw in section 2.3, memory use is one of the key challenges that Ignite has experienced with Elasticsearch. Thus, it would be interesting to measure the memory footprint of the different databases when faced with various workloads. This type of benchmark was considered for the thesis, as Docker provides tools for measuring memory use of running containers (i.e. the databases in this case). However, this was abandoned due to the way that Elasticsearch uses system memory, since it reserves a large static portion of memory for its Java Virtual Machine, making it difficult to observe the actual memory from outside a container. But tools exist to profile memory within the JVM of Elasticsearch, so this type of benchmark could be performed with further work.

References

[1] Hermann Mørkrid. *Computer Science Specialization Project: Challenges with Running Elasticsearch in Production.* Norwegian University of Science and Technology, 2023.

[2] Elastic. *What is Elasticsearch?* URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html (visited on Jan. 30, 2023).

[3] Elastic. *Data in: documents and indices.* URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/documents-indices.html (visited on Jan. 30, 2023).

[4] Elastic. *Scalability and resilience: clusters, nodes, and shards.* URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/scalability.html (visited on Jan. 30, 2023).

[5] Elastic. *Dynamic field mapping.* URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/dynamic-field-mapping.html (visited on Feb. 15, 2023).

[6] Elastic. *Information out: search and analyze.* URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-analyze.html (visited on Jan. 30, 2023).

[7] Elastic. *Query DSL.* URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html (visited on Feb. 17, 2023).

[8] Elastic. *Aggregations.* URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html (visited on Feb. 17, 2023).

[9] Apache Lucene. *Lucene Features.* URL: https://lucene.apache.org/core/ (visited on Mar. 1, 2023).

[10] Elastic. *Aggregations.* URL: https://www.elastic.co/guide/en/kibana/current/lucene-query.html (visited on Mar. 1, 2023).

[11] Elastic. *A Practical Introduction to Logstash.* URL: https://www.elastic.co/blog/a-practical-introduction-to-logstash (visited on Mar. 3, 2023).

[12] Elastic. *Kibana — your window into Elastic.* URL: https://www.elastic.co/guide/en/kibana/current/introduction.html (visited on Mar. 3, 2023).

[13] Elastic. *Learn about the Elastic Stack*. URL: https://www.elastic.co/guide/index.html (visited on Mar. 7, 2023).

[16] WebPivotTable. *Pivot Table*. URL: https://webpivottable.com/doc/pivot-table/ (visited on May 3, 2023).

[17] Microsoft. *Overview of PivotTables and PivotCharts*. URL: https://support.microsoft.com/en-us/office/overview-of-pivottables-and-pivotcharts-527c8fa3-02c0-445a-a2db-7794676bce96 (visited on May 3, 2023).

[18] Elastic. *Terms aggregation*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/8.12/search-aggregations-bucket-terms-aggregation.html (visited on Feb. 4, 2024).

[20] Avinash Lakshman and Prashant Malik. “Cassandra — A Decentralized Structured Storage System”. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[21] Seth Gilbert and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. In: *ACM SIGACT News* 33.2 (2002), pp. 51–59.

[22] Apache Cassandra. *The Cassandra Query Language (CQL)*. URL: https://cassandra.apache.org/doc/latest/cassandra/cql/index.html (visited on Aug. 29, 2023).

[23] ScyllaDB. *Where Cassandra Falls Short, and Why*. URL: https://lp.scylladb.com/cassandra-falls-short-wp-offer (visited on Aug. 31, 2023).

[24] DataStax (Apache Cassandra Enterprise Solution). *DataStax Enterprise Capacity Policy*. URL: https://docs.datastax.com/en/dseplanning/docs/dse-capacity-planning.html#datastax-enterprise-capacity-policy (visited on Aug. 31, 2023).

[25] Amazon Web Services. *Amazon EC2 I3en Instances*. URL: https://aws.amazon.com/ec2/instance-types/i3en/ (visited on Aug. 31, 2023).

[26] Scylla Documentation. *CQL - Apache Cassandra Query Language*. URL: https://opensource.docs.scylladb.com/stable/cql/ (visited on Aug. 31, 2023).

[27] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2006, pp. 205–218.

[28] ClickHouse. *Our story*. URL: https://clickhouse.com/company/our-story (visited on Feb. 4, 2024).

[29] Alexey Milovidov. *Introducing ClickHouse, Inc.* Sept. 20, 2021. URL: https://clickhouse.com/blog/introducing-click-house-inc (visited on Feb. 4, 2024).

[30] ClickHouse. *Why is ClickHouse so fast?* URL: https://clickhouse.com/docs/en/concepts/why-clickhouse-is-so-fast (visited on Feb. 4, 2024).

[31] ClickHouse. *ANSI SQL Compatibility of ClickHouse SQL Dialect*. URL: https://clickhouse.com/docs/en/sql-reference/ansi (visited on Jan. 27, 2024).

[32] ClickHouse. *Aggregate Function Combinators*. URL: https://clickhouse.com/docs/en/sql-reference/aggregate-functions/combinators (visited on Feb. 4, 2024).

[33] Tom Schreiber and Derek Chia. *ClickHouse Keeper: A ZooKeeper alternative written in C++*. Sept. 27, 2023. URL: https://clickhouse.com/blog/clickhouse-keeper-a-zookeeper-alternative-written-in-cpp (visited on Feb. 4, 2024).

[34] Uber Engineering Blog. *Fast and Reliable Schema-Agnostic Log Analytics Platform*. URL: https://www.uber.com/en-US/blog/logging/ (visited on Jan. 27, 2024).

[35] The CloudFlare Blog. *HTTP Analytics for 6M requests per second using ClickHouse*. URL: https://blog.cloudflare.com/http-analytics-for-6m-requests-per-second-using-clickhouse/ (visited on Jan. 27, 2024).

[36] Sentry Blog. *Introducing Snuba: Sentry's New Search Infrastructure*. URL: https://blog.sentry.io/introducing-snuba-sentrys-new-search-infrastructure/ (visited on Jan. 27, 2024).

[37] Robert Griesemer et al. *Hey! Ho! Let's Go!* URL: https://opensource.googleblog.com/2009/11/hey-ho-lets-go.html (visited on Jan. 8, 2024).

[38] Microsoft Security Response Center. *A proactive approach to more secure code*. URL: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (visited on Jan. 16, 2024).

[39] Rob Pike. *Go at Google: Language Design in the Service of Software Engineering*. URL: https://go.dev/talks/2012/splash.article#TOC_14. (visited on Jan. 16, 2024).

[40] Linux manual. *pthread_create*. URL: https://man7.org/linux/man-pages/man3/pthread_create.3.html#NOTES (visited on Jan. 16, 2024).

[41] Go programming language documentation. *GOMAXPROCS*. URL: https://pkg.go.dev/runtime#GOMAXPROCS (visited on Jan. 18, 2024).

[42] The Go Programming Language. *stack.go*. URL: https://github.com/golang/go/blob/f296b7a6f045325a230f77e9bda1470b1270f817/src/runtime/stack.go#L72 (visited on Jan. 18, 2024).

[43] ClickHouse. *Golang driver for ClickHouse*. URL: https://github.com/ClickHouse/clickhouse-go (visited on Feb. 2, 2024).

[44] ClickHouse. *Bulk Inserts*. URL: https://clickhouse.com/docs/en/cloud/bestpractices/bulk-inserts (visited on Jan. 24, 2024).

[45] Elastic. *The official Go client for Elasticsearch*. URL: https://github.com/elastic/go-elasticsearch (visited on Feb. 2, 2024).

[46] Elastic. *go-elasticsearch: BulkIndexerConfig*. URL: https://github.com/elastic/go-elasticsearch/blob/a58e89f/esutil/bulk_indexer.go#L56 (visited on Jan. 26, 2024).

[47] Go. *Standard library testing package documentation: Benchmarks*. URL: https://pkg.go.dev/testing@go1.21.6#hdr-Benchmarks (visited on Feb. 2, 2024).

[48] Docker. *Docker Overview*. URL: https://docs.docker.com/get-started/overview/ (visited on Feb. 2, 2024).

[49] Alibaba Cloud Database OLAP Product Department. *ClickHouse vs. Elasticsearch*. July 21, 2021. URL: https://www.alibabacloud.com/blog/clickhouse-vs--elasticsearch_597898 (visited on Feb. 4, 2024).

[50] ClickHouse. *Support for Approximated Calculations*. URL: https://clickhouse.com/docs/en/about-us/distinctive-features#support-for-approximated-calculations (visited on Feb. 4, 2024).

[51] Elastic. *Bucket sort aggregation*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/8.12/search-aggregations-pipeline-bucket-sort-aggregation.html (visited on Feb. 5, 2024).

[52] Elastic. *Terms aggregation: Ordering by a sub-aggregation*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/8.12/search-aggregations-bucket-terms-aggregation.html#_ordering_by_a_sub_aggregation (visited on Feb. 5, 2024).

[53] Christopher Ireland and David Bowers. "Exposing the Myth: Object-Relational Impedance Mismatch is a Wicked Problem". In: *DBKDA 2015, The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications* (2015), pp. 21–26.

[54] ClickHouse. *ClickHouse Go Client*. URL: https://clickhouse.com/docs/en/integrations/go#clickhouse-go-client (visited on Feb. 5, 2024).

[55] Kubernetes. *Kubernetes Documentation*. URL: https://kubernetes.io/docs/home/ (visited on Feb. 5, 2024).

Appendix A

Benchmark Results

Figures A.1, A.2, A.3, A.4, A.5 and A.6 show the results of the benchmarks described in section 3.6. The benchmark tests were run on a laptop, with specifications shown in the figures, following the steps from section 3.6.1. Docker, the tool used to run the databases, was configured to run with 16 CPU cores and 12 GB of RAM.

Comparing the benchmark results, we get the following ratios between the benchmarks for the different databases:

- **Ingestion:** ClickHouse is $\frac{2794904997 \text{ ns}}{959848403 \text{ ns}} \approx 2.9$ times faster than Elasticsearch.
- **Analysis queries:** Elasticsearch is $\frac{18758777 \text{ ns}}{5927975 \text{ ns}} \approx 3.2$ times faster than ClickHouse.
- **Table creation:** ClickHouse is $\frac{200939482 \text{ ns}}{18758777 \text{ ns}} \approx 10.7$ times faster than Elasticsearch.

```
goos: linux
goarch: amd64
pkg: hermannm.dev/analysis
cpu: AMD Ryzen 7 PRO 6850U with Radeon Graphics
BenchmarkIngestion-16            100    2794904997 ns/op
PASS
ok      hermannm.dev/analysis    291.557s
```

Figure A.1: Ingestion benchmark result for Elasticsearch.
Result: `2794904997 ns/op` $\approx$ `2.79 s/op`
Output from: `go test -bench=Ingestion -benchtime=100x`

```
goos: linux
goarch: amd64
pkg: hermannm.dev/analysis
cpu: AMD Ryzen 7 PRO 6850U with Radeon Graphics
BenchmarkIngestion-16              100     959848403 ns/op
PASS
ok      hermannm.dev/analysis    98.968s
```

Figure A.2: Ingestion benchmark result for ClickHouse.
Result: `959848403 ns/op` ≈ `0.96 s/op`
Output from: `go test -bench=Ingestion -benchtime=100x`

```
goos: linux
goarch: amd64
pkg: hermannm.dev/analysis
cpu: AMD Ryzen 7 PRO 6850U with Radeon Graphics
BenchmarkQuery-16            1000       5927975 ns/op
PASS
ok      hermannm.dev/analysis    15.195s
```

Figure A.3: Analysis query benchmark result for Elasticsearch.
Result: `5927975 ns/op` ≈ `5.93 ms/op`
Output from: `go test -bench=Query -benchtime=1000x`

```
goos: linux
goarch: amd64
pkg: hermannm.dev/analysis
cpu: AMD Ryzen 7 PRO 6850U with Radeon Graphics
BenchmarkQuery-16            1000      18758777 ns/op
PASS
ok      hermannm.dev/analysis    20.650s
```

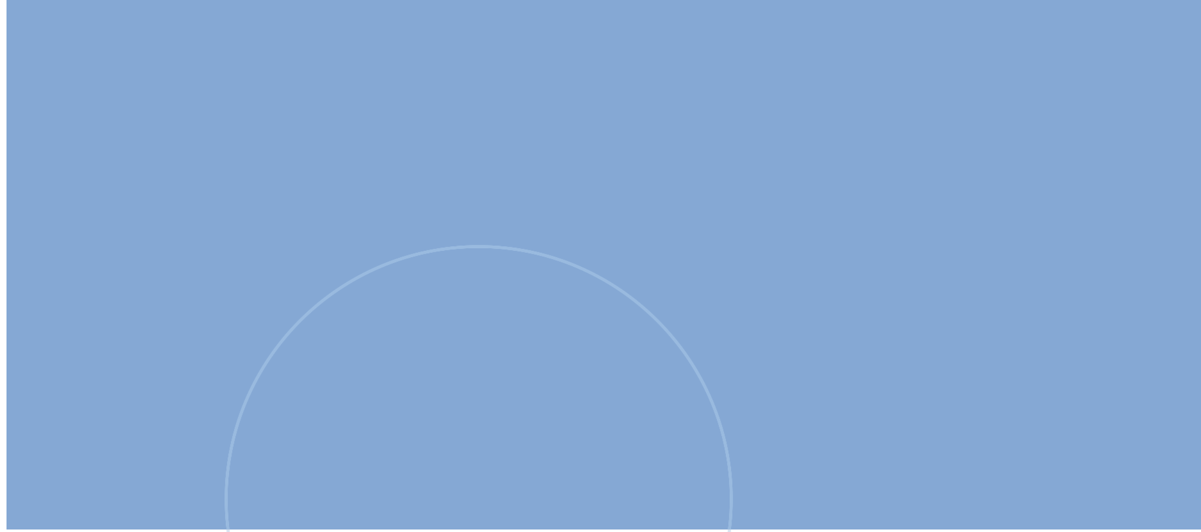
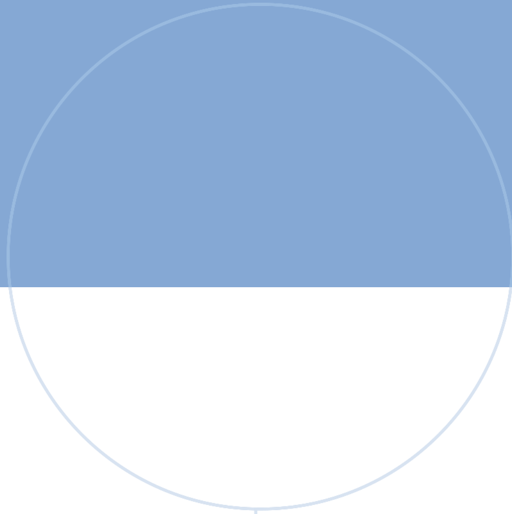
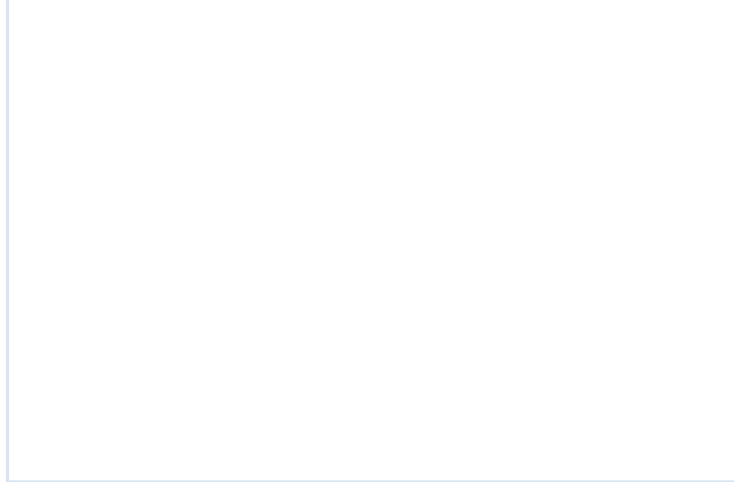
Figure A.4: Analysis query benchmark result for ClickHouse.
Result: `18758777 ns/op` ≈ `18.76 ms/op`
Output from: `go test -bench=Query -benchtime=1000x`

```
goos: linux
goarch: amd64
pkg: hermannm.dev/analysis
cpu: AMD Ryzen 7 PRO 6850U with Radeon Graphics
BenchmarkCreateTable-16             100     200939482 ns/op
PASS
ok      hermannm.dev/analysis    24.339s
```

Figure A.5: Table creation benchmark result for Elasticsearch.
Result: `200939482 ns/op` ≈ `200.94 ms/op`
Output from: `go test -bench=CreateTable -benchtime=100x`

```
goos: linux
goarch: amd64
pkg: hermannm.dev/analysis
cpu: AMD Ryzen 7 PRO 6850U with Radeon Graphics
BenchmarkCreateTable-16             100      10549878 ns/op
PASS
ok      hermannm.dev/analysis    1.238s
```

Figure A.6: Table creation benchmark result for ClickHouse.
Result: `10549878 ns/op` ≈ `10.55 s/op`
Output from: `go test -bench=CreateTable -benchtime=100x`

NTNU
Norwegian University of
Science and Technology

ignite