Storage and Transformation for Data Analysis Using NoSQL

Lagring och transformation för dataanalys med hjälp av NoSQL

Christoffer Nilsson
John Bengtson

Supervisor : Zlatan Dragisic
Examiner : Olaf Hartig
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.
Abstract

It can be difficult to choose the right NoSQL DBMS, and some systems lack sufficient research and evaluation. There are also tools for moving and transforming data between DBMS’ in order to combine or use different systems for different use cases. We have described a use case, based on requirements related to the quality attributes Consistency, Scalability, and Performance. For the Performance attribute, focus is fast insertions and full-text search queries on a large dataset of forum posts. The evaluation was performed on two NoSQL DBMS’ and two tools for transforming data between them. The DBMS’ are MongoDB and Elasticsearch, and the transformation tools are NotaQL and Compose’s Transporter. The purpose is to evaluate three different NoSQL systems, pure MongoDB, pure Elasticsearch and a combination of the two. The results show that MongoDB is faster when performing simple full-text search queries, but otherwise slower. This means that Elasticsearch is the primary choice regarding insertion and complex full-text search query performance. MongoDB is however regarded as a more stable and well-tested system. When it comes to scalability, MongoDB is better suited for a system where the dataset increases over time due to its simple addition of more shards. While Elasticsearch is better for a system which starts off with a large amount of data since it has faster insertion speeds and a more effective process for data distribution among existing shards. In general NotaQL is not as fast as Transporter, but can handle aggregations and nested fields which Transporter does not support. A combined system using MongoDB as primary data store and Elasticsearch as secondary data store could be used to achieve fast full-text search queries for all types of expressions, simple and complex.
Acknowledgments

We begin by thanking Berkant Savas and Mari Ahlqvist at iMatrics, for making this thesis project possible. It has been the start of an exciting journey. Carrying on, we would also like to thank Olaf Hartig for guiding us through this thesis. You have managed to both challenge and encourage us.

I, John Bengtson, would like to thank my girlfriend Sofie, my father Lage, and my mother Eva, for supporting me throughout my education. You have always believed in me and helped me become the person I am today.

I, Christoffer Nilsson, thank my wonderful girlfriend Anna. Both for her support and many necessary distractions. I would also like to thank my entire family for their support and encouragement.
## Contents

**Abstract** iii

**Acknowledgments** iv

**Contents** v

**List of Figures** vii

**List of Tables** vii

### 1 Introduction

1.1 Motivation ........................................... 2
1.2 Aim .................................................. 3
1.3 Research Questions ................................... 4
1.4 Delimitations ......................................... 4

### 2 Theory

2.1 Server Clusters ...................................... 5
2.2 NoSQL ................................................ 5
2.3 CAP-Theorem, ACID and BASE ......................... 6
2.4 Full-Text Search ..................................... 7
2.5 Document-Oriented Databases ......................... 7
  2.5.1 MongoDB ........................................ 8
  2.5.2 Elasticsearch ..................................... 8
2.6 Data Modelling ...................................... 9
2.7 Data Transformation .................................. 10
  2.7.1 NotaQL ........................................... 10
  2.7.2 Transporter ....................................... 11
2.8 Exploratory Testing .................................. 12
2.9 Research Methodology ................................ 13
2.10 Evaluating NoSQL Systems ......................... 14
2.11 Related Work ....................................... 16

### 3 Method

3.1 Use Case ........................................... 17
  3.1.1 Evaluation ....................................... 18
3.2 Data Modeling ....................................... 19
3.3 Java Client ......................................... 20
3.4 Hardware Specification ............................. 20
3.5 Quantitative Evaluation of MongoDB and Elasticsearch
  3.5.1 Exploratory Testing Plan ......................... 21
  3.5.2 MongoDB Test Environment .................... 22
  3.5.3 Elasticsearch Test Environment ............... 23
# List of Figures

1.1 System component and scope of the thesis ........................................... 2
3.1 A NoAM block with entries ............................................................... 20
3.2 MongoDB test environment ............................................................. 22
3.3 Elasticsearch test environment ......................................................... 23
3.4 Transformation test environment ....................................................... 27
4.1 Insertion Strong Query Router .......................................................... 33
4.2 Insertion Weak Query Router ............................................................ 33
4.3 Replication Strong Query Router ....................................................... 34
4.4 Replication Weak Query Router ........................................................ 35
4.5 Weak Query Router Simple Queries ............................................... 36
4.6 Strong Query Router Simple Queries .............................................. 36
4.7 Weak Query Router Complex Queries .............................................. 37
4.8 Strong Query Router Complex Queries ............................................ 37
4.9 Weak Query Router Three Nodes ..................................................... 38
4.10 Transformations on data size small ................................................. 43
4.11 Transformations on data size medium .............................................. 43
4.12 Transformations on data size large .................................................. 44
C.1 Insertion Small Dataset ................................................................. 71
C.2 Insertion Medium Dataset ............................................................... 71
C.3 Insertion Large Dataset ................................................................. 72
C.4 Insertion Small Dataset ................................................................. 72
C.5 Insertion Medium Dataset ............................................................... 72
C.6 Insertion Large Dataset ................................................................. 73
List of Tables

3.1 NoAM concepts .................................................. 19
NoSQL has risen as a new concept for storing and managing data \[1\]. It is often stated that it implements features that traditional database management systems (DBMS), such as relational DBMS, do not. Primarily NoSQL systems support techniques to overcome problems concerning unstructured data, and horizontal scalability. There are many different NoSQL systems which differ from each other and are developed for different purposes. They have both strengths and weaknesses, which can make it difficult to find a solution that fits your needs. Trade-offs are often made to strengthen what is considered most important.

1.1 Motivation

iMatrics is a Linköping based start-up company that develops large scale text mining algorithms for various information technology applications. Part of their work involves large volumes of data which consist of many semi-structured documents and they are interested in a database solution for storing this data in a way that allows for nearly instant retrieval of all data relating to a search. Because of the large volumes of data, and the semi-structured nature of the data, NoSQL is considered as an interesting area to explore.

iMatrics is in need of a system that can be described in the following way. Their large amounts of data need to be stored in a database. Then, given a set of search requirements, the stored data will be searched and extracted for post-processing. Post-processing includes analysis of text in different ways. The system’s workflow can be seen below in Figure 1.1.

![Figure 1.1: System component and scope of the thesis]
1.2 Aim

Input, Post-processing, and Output are outside the scope of this thesis. The focus will be on storage of data and full-text search, as can be seen in Figure 1.1 above. The basic functional requirements of the system are the following:

- Inserting semi-structured text-documents in the form of JSON
- Retrieving inserted documents in the form of JSON
- Horizontal scalability
- Full-text search using:
  - Index upon word stem (stemming)
  - Remove stop words when searching
  - Regular expressions

The aim of the thesis is divided into three major parts. The first part includes an evaluation and comparison of two different solutions for storing semi-structured text-documents and performing full-text search operations. These solutions are MongoDB\(^1\) and Elasticsearch\(^2\). MongoDB is a document-oriented NoSQL DBMS that has support for text search operations. Elasticsearch is a search engine that can also be configured to act as a document-oriented NoSQL DBMS. These two systems are two of the most popular software solutions in their respective categories \(^3\) and could be used independently, to some extent, for both storage and full-text search operations.

Since MongoDB was developed to act as a DBMS, and Elasticsearch was developed to act as a search engine, there may be challenges in using them independently to fulfill the needs related to both storage and full-text search functionality. MongoDB is recommended as an option for a primary data store \(^4\), while Elasticsearch is recommended to be used together with another data store as primary \(^4\).

In addition to this, MongoDB does not have the high-end full-text search capabilities as Elasticsearch \(^5\), \(^6\), \(^7\). On the other hand, Elasticsearch seems weaker when it comes to the Consistency part of the CAP (Consistency, Availability, Partition tolerance) theorem and less trustworthy when it comes to the Availability part \(^3\), \(^4\), \(^5\), \(^6\). Users of both tools say that MongoDB is more reliable, in the sense that it is less likely to lose data and more likely to remain reachable despite the absence of nodes \(^4\), \(^8\). However, with an increasing amount of data reaching several GB in size, Elasticsearch seems faster at performing full-text search operations \(^8\), \(^9\).

Our hypothesis is that a combined solution, where MongoDB is used as an underlying database and Elasticsearch for full-text search operations, will result in a reliable database system while also maintaining fast retrieval speeds for full-text search. In such a combination, data needs to be transferred between MongoDB and Elasticsearch. In order to transfer and transform data from MongoDB to Elasticsearch we have chosen two tools, Compose’s Transporter\(^3\) and NotaQL\(^4\).
1.3 Research Questions

The Compose Transporter allows for transformation of data between several different systems, including MongoDB and Elasticsearch. NotaQL is a cross-system transformation language that currently has support for MongoDB, HBase, Redis, CSV files, and JSON files. In order to use NotaQL, we first need to extend it to support transformations into Elasticsearch, which is the second major part of the thesis. The final major part is to evaluate and compare Compose’s Transporter and NotaQL for transferring and transforming data from MongoDB to Elasticsearch.

In addition to the basic functional requirements introduced above, the evaluation will be based on a use case, which was created together with iMatrics. The evaluation of the NoSQL systems, and the transformation tools, will be both quantitative and qualitative, and software quality attributes will be derived from the use case.

The workflow of the thesis will be as following:

1. MongoDB and Elasticsearch will be setup, compared and evaluated.
2. NotaQL will be extended to support transformations into Elasticsearch.
3. Compose’s Transporter and NotaQL will be setup, compared and evaluated.

1.3 Research Questions

1. Do MongoDB and Elasticsearch fulfill the aforementioned system requirements?
2. Can Compose’s Transporter and NotaQL handle transformations on semi-structured text data from MongoDB and insert into Elasticsearch?
3. What are the advantages and disadvantages of using the combined solution as compared to separately using MongoDB or Elasticsearch?

1.4 Delimitations

Because of time limitations, there are some delimitations:

- Ultimately, the system is intended to function on data from different kinds of information sources, but in this thesis, the type of data considered is discussion posts from an Internet forum.
- The system will not be tested on requests by multiple clients simultaneously.
- The number of software quality attributes included in the evaluation is limited to the three most relevant derived from the use case, namely scalability, consistency and performance.
This chapter introduces the theory needed in order to understand the thesis.

2.1 Server Clusters

A server cluster is a group of computer systems. They are connected in order to act as a unit and can thus split the workload among themselves. A single system within the cluster is referred to as a node [6], [10]. A few more related and relevant concepts are Shard and Replica. A Shard is a part of the entire data of the database. These are used in order for a single node to work on a smaller part of data in order to overcome obstacles such as insufficient storage or processing power. A Replica is an exact copy of a shard that is placed on another node and used to achieve high availability, through the means of redundancy [6].

2.2 NoSQL

NoSQL [1] is a term that describes a new set of DBMSs that works differently compared to relational DBMSs. What the term ”No” actually means is a bit vague. Some propose ”Not Only” SQL, others ”No relational” SQL [1]. What NoSQL refers to are new ways of storing and managing data, compared to the traditional relational DBMSs.

The reason NoSQL appeared is because relational DBMSs are not suitable for all use cases. Relational DBMSs are built on the assumption that the structure of the data is known in advance, and hence can be stored in a well-structured way [1]. This assumption did not work well with the increasing use of the Internet. Data suddenly appeared in many different forms and it was difficult to structure the data. NoSQL offers different ways of managing unstructured data or semi-structured data and therefore they often have a flexible schema, which helps address and solve this problem.

Another problem was the amount of data [11]. Traditionally, relational DBMSs use the concept of vertical scaling, which means that if you need more storage space on a database node, you add it directly to that node. If storage space keeps getting added, they will eventually reach their limit and data needs to be stored somewhere else. Distributing data on
different nodes in relational databases is not ideal because they were not initially built for that and it often increases the complexity of the system \cite{11}, \cite{12}. Several features, such as table joins and transactions, become more difficult to perform and often result in decreased performance \cite{11}, \cite{12}. NoSQL however, uses the concept of horizontal scaling, which means that if more storage space is needed, an additional node is added to the system, and since it was designed for this it can actually help increase performance instead of decreasing it.

### 2.3 CAP-Theorem, ACID and BASE

The CAP-theorem \cite{13}, \cite{14} is an important concept concerning distributed database systems. CAP stands for “Consistency”, “Availability” and “Partition-tolerance”. If a shard and its replicas all contain the same data, they are in a consistent state. Availability aims to describe that if a node is fully functional and receives a request, it will return a response. Simply put, if working nodes exist, every request will receive a response. Due to different types of network failures, nodes within a cluster can be partitioned. With a partition-tolerant system, nodes should be able to function properly and respond to requests, even if some kind of partition occurs. However, this might lead to problems concerning consistency if nodes can not communicate with each other.

The theorem states that only two out of these three guarantees can be achieved in distributed systems \cite{14}. The possible outcomes are CA, AP, and CP. Developers can trade off one property for another and also in different ways. However, it is sometimes difficult to achieve these, especially CA. Focusing on consistency and availability, and not on partition-tolerance could be ill-advised. This choice could mean that you are vulnerable to split brain situations since you are relying on the network communications within the system to never fail, which is infeasible at best \cite{11}.

The CAP-theorem tells us that systems focus on different properties and that there is a possibility to perform trade-offs. This leads us to two other important concepts when working with databases, ACID and BASE. ACID stands for Atomicity, Consistency, Isolation and Durability \cite{15}. It is a set of properties that wants to be guaranteed in order to achieve strong consistency when performing transactions. Relational DBMSs often focus on ACID, and there are also some NoSQL databases that do that as well \cite{15}. The ACID properties are strongly connected to CA and CP from the CAP properties.

Achieving strong consistency when using NoSQL databases might not always be the goal, especially since the Internet demands that a lot of users should be able to access large volumes of data at the same time. This is where the BASE properties are interesting to look at. BASE is an acronym for Basic Availability, Soft-state and Eventual Consistency \cite{16}, \cite{15}. It is built on the idea that data does not always has to be in a consistent state. It focuses more on eventual consistency and NoSQL systems that achieve the BASE properties also often tries to increase availability at the cost of consistency \cite{11}. It is strongly connected to AP from the CAP properties. With Basic Availability, data is distributed on several nodes with shards and replication, and if a failure occurs on one node, there are still other nodes containing accessible data. Simply it means that the service is basically available all the time, even though the entire dataset is not. When allowing reads from replicas the concept of Eventual Consistency comes into play. With Eventual Consistency, insertions and updates are not replicated directly, but they will eventually be. Exactly when, is configurable and up to the developers of the system. For example, if the workload is high, replication can be postponed until the workload is low. Soft state means that the state of the system can change over time. Regardless of user interaction, the state of the system can change, for example when a shard and its replicas reach a consistent state.
2.4 Full-Text Search

Full-text search refers to techniques regarding searches of entire documents in a collection of documents. It is a common concept of information retrieval and there are many tools available for performing such searches. A platform implementing such techniques is commonly referred to as a search engine.

Search engines are complete software platforms designed to retrieve and filter information from data stores. Examples of popular search engines are Google, Bing, Yahoo! and Ask. In this thesis we will use an open source search engine called ElasticSearch which is suitable for performing full-text searches on a wide variety of different information. ElasticSearch is part of a software stack called the Elastic Stack which also offers extra functionality such as visualization, security, and performance monitoring.

2.5 Document-Oriented Databases

One category of NoSQL databases is document-oriented databases [17], [11]. There are some terms used for describing how a document-oriented database works. First of all, the data stored in a document-oriented database is called a document. You can think of a document as a key-value item, where the document itself, along with an identifier, is the key, and the content of the document is the value [17]. How the content is structured is different between databases, but common ways of doing it is using XML, JSON or BSON [11]. Documents are also stored and managed in a collection, which basically is a set of documents [17].

A document contains fields, which describe the content of the document. A field can also be seen as a kind of key-value item. A simple way of storing data as a document is by creating a field named "content" and store data there, for example the text from a forum discussion post. This can be compared to a row in a relational database with two columns named "id" and "content". Document-oriented databases are schema less, which means that users can define any number of fields they want. Documents within a collections may have different number of fields, which can be defined during the design phase, but also added and removed during usage [11]. Document-oriented databases are good to use when storing texts, and need to query specific content in the text. To get a grip on how a document might look like, there is an example in Listing 2.1 where a discussion post from an Internet forum has been structured.

Listing 2.1: A document example

```json
{
    title: "Food review",
    author: "Chef_1",
    publication date: 2015-11-12,
    timestamp: 18:40,
    body: "The other night, I made a lasagna with extra cheese...."
}
```

Since documents contain fields to describe the content of the data, it is classified as at least semi-structured [18], [19]. The possibility to add and delete fields during a document’s lifetime makes it possible to evolve and optimize the document structure. When having documents with a detailed structure, it is possible to construct relevant and optimized queries. For example, this can be achieved by querying a single field.
2.5.1 MongoDB

MongoDB is a popular open source document-oriented DBMS [2]. It is schema free and structures its documents as BSON (Binary JSON) [11]. If we connect it to the CAP properties, it is by default a CP database [13]. Replication in MongoDB is performed using replica sets. A replica set consists of a shard node called the primary and replicas called secondaries. The primary manages all write and read operations, and replicates data to the secondaries asynchronously [19]. This is the default setting and ensures high consistency. It is possible to configure the secondary nodes to respond to reads as well. However, this makes the consistency weaker since a secondary might return outdated data, while it instead increases the availability and tolerance for network partitions. A replica set has support for failover, which means that if the primary is partitioned from its replicas, or crashes, the secondaries select a new primary. This increases the availability of the entire system.

Since the replication is asynchronous, and secondaries are allowed to respond to reads even if they do not have the latest data, MongoDB implements the BASE properties. MongoDB offers different types of configurations, which for example can make it more CP than AP and vice versa.

The document fields in MongoDB are key-value fields. The key is a string, while the value can be of several different types. It is possible to index any field in MongoDB [18]. In "Data Modeling in the NoSQL World" [20], they categorize the value types as the following:

- Basic types
  - Strings
  - Integer
  - Dates
  - Boolean
- Arrays
- Documents

In MongoDB it is possible to set up a sharded cluster. In such a setup, there are three main components used. Firstly, we have the shards. As described in Section 2.1 shards are parts of the database, which are usually distributed on different nodes. A shard can either be set up on a single node or configured as a replica set using several nodes. Secondly, we have the config server. The config server keeps track of all shards within the cluster. A config server can also be set up either on a single node or configured as a replica set using several nodes. Last we have the query router (in MongoDB terms also called mongos), which manages the communication with clients. The query router retrieves and responds to all queries. It also makes use of the config server in order to find out which shards it should use.

2.5.2 Elasticsearch

Elasticsearch is a fast and versatile search engine that uses multiple types of indexes for search optimization and clusters for horizontal scalability. Elasticsearch is the most popular search engine according to DB-Engines [2]. It was built on top of Apache Lucene, which is an open source search engine library [6].

Apache Lucene makes use of four basic concepts: Document, Field, Term, and Token. A Document is an object containing the data in the database. The entire database can consist of many documents. A Field is a part of a document and contains two parts, a name and a value.
A Term is a word that has been indexed using an inverted index and is thus searchable. An inverted index is a list consisting of all terms, as keys, and all documents that contain them, as values. Lastly, a Token is a single occurrence of a term. Using these four basic concepts along with inverted indexes as well as many additional information retrieval techniques, Apache Lucene allows for powerful full-text searches. The additional techniques include concepts such as term vectors, synonym filters, and multiple language stemming filters. The process where Elasticsearch analyses the text and uses these techniques is called Analyze.

Term vectors are mathematical representations of how a term relates to a document, the entire database, and other terms. Synonym filters are used to recognize synonyms to terms in order to account for this during for example a full-text search. Stemming is used to combine different word extensions/flexion under the same stem. These are available for many different languages and this is necessary due to the different characteristics of languages.

Elasticsearch uses indexes to store data. It is important to separate these indexes from for example Lucene’s inverted indexes. Elasticsearch uses Lucene as an underlying library, but the indexes used to store data are more closely related to either a MongoDB database or collection, depending on its use, than a Lucene inverted index. An index is sharded, which means that the data is split across a number of shards. A shard is, as presented for MongoDB, a part of the database. The database may also have replicas and in such a case each shard is replicated once per database replica. The documents in an Elasticsearch index are restricted in the way that all common fields, fields shared by several documents, have to be of the same datatype. Fields are always of a datatype and Elasticsearch has different ways for handling the different datatypes. The datatypes can be manually specified for a specific field or automatically determined by Elasticsearch upon insertion into the index. Two example datatypes are string which stores text, and integer which stores whole numbers. For a full list of available datatypes refer to the Elasticsearch reference guide.

2.6 Data Modelling

Data modelling in traditional relational databases is an area that is well established. EER-diagrams are often used to model the data and display relationships. However, modelling in NoSQL databases is not as well established. There are no standard methods on how to model the data, and one reason for this is the flexible schema. Even though this thesis does not focus on how to best model data, a modeling technique is good to use for several reasons, which are mentioned below.

In "Modeling and querying data in NoSQL databases", the authors highlight the importance of modeling data as it increases communication between different stakeholders, such as people responsible for design and implementation. Data models are good to describe the rules of the data and the system. The introduced approach for modelling data is built on what queries will be executed. The reason to base it on queries is because the data is unstructured (or semi-structured), which makes it difficult to model based on the content.

The authors first model a relational database with an ER-diagram and then later translate the diagram for a document-oriented database. To represent document-oriented databases they use a class diagram where each document is a class. Fields inside a document are class attributes. Relationships between documents are modelled with class relationships using references. An example of a relationship could be between two documents in which the first document is a blog post, and the other document is information about the person that has
written the blog post.

Since NoSQL systems can not perform join operations as in relational DBMS, queries needs to visit the database several times. So when queries are executed, it is desired to visit the database as few times as possible. The model introduced in “QODM: A query-oriented data modeling approach for NoSQL databases” [27] is called query-oriented data modeling, which base the modeling on types of queries that will be executed.

In the article, the authors give the example of storing content from a web application. In the example there are user information, blog posts by users, and comments on blogs. Say that user information is stored as one document, blog posts as another, and comments as a third, with relationships between them. In order to extract these three documents in one query, the database will be visited three times. However, if all data is stored in one document, the database only need to be visted once. This supports the choice to base modelling on queries. UML is used to represent the modeled data.

In “Data Modeling in the NoSQL World” [20], the authors model data with a model called NoAM (NoSQL Abstract Data Model). A document is seen as a unit that is represented as a block, and also called a block. A field in a document is called an entry and a collection is called a collection. This leads to the following definition of the NoAM data model. A database, in NoAM, is a set of collections, which is called a database. As in all document-oriented databases, a collection has a name. A collection contains a set of blocks, which all have a unique identifier for the collection. A block consists of at least one entry, and each entry has a key called entry key (ek) and a value called entry value (ev). The key is unique for that block. This is a general model which can be used on any type of NoSQL database. The idea here is to model the data without initially having to take the targeted system into account. The collections, blocks, and entries are defined before selecting a NoSQL system.

2.7 Data Transformation

Since NoSQL systems are built for performance, they are usually based on context-specific optimizations. This means that there are different advantages and disadvantages depending on the system used and therefore it can be necessary to switch between them at times [28]. There are many different data operations which can be necessary in order to facilitate the use of multiple databases.

2.7.1 NotaQL

NotaQL is a transformation language built for cross system NoSQL data transformation [29]. NotaQL comes with a tool that supports the language. The tool is built in Java using Antlr to easily define and generate the language constructs. Currently, NotaQL supports the following stores; MongoDB, HBase, Redis, CSV files, and JSON files. However, it was constructed to be easily extendable and uses an underlying Apache Spark instance [30]. Therefore all engines supporting Spark can easily be added to NotaQL. Others can also be added, but may not be as easy and/or receive the same benefits as when using Spark. NotaQL supports the following transformation types: Renaming of fields, Projections, Filters, and Aggregations. Projections are used to extract parts of the data onto chosen fields. Filters are used to remove documents using conditions. Aggregations are used to combine fields.

NotaQL is designed for extendability and therefore based on a data-store independent data model and language [30]. There are also extension points in the grammar of the NotaQL language which allows for store specific path traversal and custom functions [30]. For every engine it is necessary to define how to process input and output path expressions. Therefore
each engine will have its own specific path traversal. There is also the possibility to define engine-specific functions and therefore extend the NotaQL grammar for that engine. These functions will have to be implemented in the NotaQL extension for them to work and allow for customized functionality.

A NotaQL transformation expression can be divided into several parts. The first part specifies the input engine (Where the data should be transformed to), and the output engine (Where the data should be transformed from). An example using MongoDB as both input and output engine can be seen below:

\[
\begin{align*}
\text{IN-ENGINE}: & \text{mongodb} \left( \text{database} \leftarrow \text{database1} \right), \\
\text{OUT-ENGINE}: & \text{mongodb} \left( \text{database} \leftarrow \text{database2} \right),
\end{align*}
\]

It is also possible to specify filters. Below we can see an example in which a filter is specified on a field, which only transform documents in which the field ‘category’ is equal to ‘A’.

\[
\begin{align*}
\text{IN-FILTER}: & \text{IN.category} = 'A'
\end{align*}
\]

After engines and filters, the transformation of fields is specified. For example which fields should be transformed, and what the field names should be. Below we can see an example in which the field ‘name’ is transformed with the same name, but the field ‘alias’ is transformed and renamed to ‘nickname’.

\[
\begin{align*}
\text{OUT.name} & \leftarrow \text{IN.name}, \\
\text{OUT.nickname} & \leftarrow \text{IN.alias}
\end{align*}
\]

The full expression of the examples above would look like following:

\[
\begin{align*}
\text{IN-ENGINE}: & \text{mongodb} \left( \text{database} \leftarrow \text{database1} \right), \\
\text{OUT-ENGINE}: & \text{mongodb} \left( \text{database} \leftarrow \text{database2} \right), \\
\text{IN-FILTER}: & \text{IN.category} = 'A', \\
\text{OUT.category} & \leftarrow \text{IN.category}, \\
\text{OUT.name} & \leftarrow \text{IN.name}, \\
\text{OUT.nickname} & \leftarrow \text{IN.alias}
\end{align*}
\]

\[\text{2.7.2 Transporter}\]

The Compose Transporter is an open source software used for transforming data between different types of data stores, such as databases or files. It is a simple tool that does not require much more than source and destination paths to the data stores. When these paths are specified, the data is extracted from the source, converted to messages in the form of JavaScript data objects, and sent to the destination. Before inserted into the destination, the messages are converted to the format used by the destination. Transformations are specified by writing JavaScript code, which opens up for many possible transformations. The actual transfer can be performed within the same system or between different systems, and also from one system to multiple others. There is a possibility to perform this transformation
in a one-time action, or using a tail function in which changes at the source are synchro-
nized and reflected at the destination. Currently, the Transporter supports the following
data stores: Elasticsearch, MongoDB, PostgreSQL, RethinkDB, RabbitMQ, and files. The
supported transformation types are split into two categories, JavaScript transformers and
Native transformers. JavaScript transformers are based on JavaScript pipelines and are very
flexible, but requires scripting expertise. Native transformers are called through functions
and the supported native transformations are: projections, filters, field rename and pretty
print.

Below we can see an example of how a JavaScript could look like when transforming
every field and document from the input stores. In this example, MonogoDB is used as both
input and output.

```javascript
var source = mongodb({"uri": "mongodb://localhost/database1"})
var sink = mongodb({"uri": "mongodb://localhost/database2"})
t.Source("source", source, "namespace", "collection1").Save("sink", sink, "namespace", "collection2")
```

Below we can see an example of how to apply a filter, which in this example filters on a field
‘category’ that matches the value ‘A’. The method skip() only transforms the documents that
matches the requirement inside the method.

```javascript
t.Source("source", source, "namespace", "collection1").Transform(
  skip({"field":"category","operator":"==","match":"A"})).Save("sink", sink, "namespace", "collection2")
```

Below we can see an example of how to transform only two fields and how to rename the
field ‘alias’ to ‘nickname’.

```javascript
t.Source("source", source, "namespace", "collection1").Transform(
  pick({"fields":["name","alias"]}).Transform(rename({"field_map":
    {"alias":"nickname"}})).Save("sink", sink, "namespace", "collection2")
```

The complete JavaScript of the examples above would look like following:

```javascript
var source = mongodb({"uri": "mongodb://localhost/database1"})
var sink = mongodb({"uri": "mongodb://localhost/database2"})
t.Source("source", source, "namespace", "collection1").Transform(
  skip({"field":"category","operator":"==","match":"A"})).Transform(
  pick({"fields":["name","alias"]}).Transform(rename({"field_map":
    {"alias":"nickname"}})).Save("sink", sink, "namespace", "collection2")
```

2.8 Exploratory Testing

Testing systems during and after development can be challenging. There are several testing
methods that can be used and one approach is called exploratory testing. In "Exploratory
Testing Explained" [32], exploratory testing is defined as the following: "Exploratory testing
is simultaneous learning, test design and test execution”. The tester continuously gathers information about the system by executing tests in order to answer questions concerning the system. The answers then provide new information, which is used to ask new questions and create new test cases. This procedure is repeated until the tester has achieved good test cases and is satisfied with the results. Initially, this means that the tester in advance does not need to know much about the system. Another testing method that exploratory testing can be compared with is scripted testing. In scripted testing, tests are executed automatically instead of manually as in exploratory. These tests are based on scripts created by the tester. Compared to exploratory, which does not demand much knowledge in advance, scripted testing demands knowledge of what should be tested, how it should behave and what results should be expected. There is also not as much interaction from the user as in exploratory.

How exploratory the testing should actually be is up to the tester. Pure exploratory testing means that everything is manually tested and decisions are made all the time. This is called freestyle exploratory testing [32]. However, the tests can be structured in different ways, depending on how much is known in advance. Exploratory testing can be combined with scripted testing, where scripts are used to test certain scenarios and new scripts are created based on new information.

Several factors influence the exploratory testing, such as the functionality of the system under test, how much the tester knows about the system under test, what tools are available for use, and what the goals of the system are [32]. During a test session, the tests should be guided by the use of a charter, which is a type of test plan [32]. What the charter looks like depends on all influencing factors of the system under test and which use case the system is tested for. However, the charter should include one or more goals. After each iteration of the test, the charter can be updated based on new information gathered until the goals are satisfied.

2.9 Research Methodology

When conducting empirical research and experiments, it is important that the entire process, from planning to conclusion, is structured and well-defined so that the process and the results can be considered reliable and valid. In the paper “Preliminary Guidelines for Empirical Research in Software Engineering” [33], the authors propose a set of guidelines for six areas when conducting empirical experiments. These are:

"• Experimental context,
• Experimental design,
• Conduct of the experiment and data collection
• Analysis,
• Presentation of results, and
• Interpretation of results."

The context guidelines discuss the importance of putting the experiment in a proper context and which information is suitable for such a task [33]. Two types of studies are presented: observational studies and formal experiments. Observational studies are based on simply observing a behaviour or process in its natural environment, while formal experiments are about recreating a behaviour or process in order to test them in a controlled environment.
Observational studies are used when gathering information directly from the industry and can therefore be good to gain insight into the industrial process. Some of the difficulties of such a study is how to define entities and attributes and how to consistently measure them. Formal experiments are performed by setting up a test environment. An important aspect of such experiments is to not oversimplify the industrial process when recreating a behaviour or process. If the process is oversimplified and the experiments performed in an incorrect context, the experiments might have little or no value at all. Another aspect that is mentioned is the exploratory aspect of a study, which refers to how much can be scripted and how much can be exploratory.

The experimental design section describes concepts important when designing an experiment as shown below:

- the population being studied,
- the rationale and technique for sampling from that population,
- the process for allocating and administering the treatments (the term “intervention” is often used as an alternative to treatment), and
- the methods used to reduce bias and determine sample size.”

The “conducting the experiment and data collection” section presents guidelines for the data collection, such as defining all measures, and introduce methods for quality control. In summary, the guidelines primarily state the importance of definitions for all aspects, and to have a critical view on related circumstances which may have affected the study.

The analysis section brings up two approaches, namely classical analysis and Bayesian analysis. The main differences between classical and Bayesian analysis is that a Bayesian analysis uses prior information in order to interpret and analyze the experimental results. Classical analysis mostly use only the current results and not prior information. The section also presents guidelines for the analysis, such as using sensitivity analysis, and ensuring assumptions are not violated. In short, the guidelines are for data quality assessment and result verification.

The presentation of results section contains examples and rules for how the data should be properly described. The interpretation of results holds guidelines for proper use of the results. It is important to show the differences of theoretical implications and practical situations, and clearly show or relate to the limitations of the study.

2.10 Evaluating NoSQL Systems

Comparing and evaluating different NoSQL systems demands a plan that describes how the system will be used and what to actually evaluate and compare. An evaluation can be both quantitative and qualitative. A quantitative evaluation is usually based on numbers, for example measurements concerning insertion and retrieval speeds. A qualitative evaluation is more subjective than a quantitative evaluation and is usually based on words, for example reviewing literature and documentation.

When choosing an appropriate system and evaluating if it satisfies particular needs, requirements are needed as a reference point. In “The Case for Application-Specific Benchmarking”, emphasis is put on the importance to perform benchmarking within a specific application context. Therefore, using requirements and goals is a good approach to describe and set
2.10. Evaluating NoSQL Systems

up the application context. In “Quality Attribute-Guided Evaluation of NoSQL databases: A Case Study” [35], the authors evaluate a number of NoSQL system candidates for a system. The authors specify requirements and create a use case based on that, which demonstrate how the system will be used and function correctly [36]. The opposite to use cases are misuse cases, which demonstrate how the system should not function. A use case includes story telling which aims to capture the basic requirements and the goals of the system. It does not explain how things will be developed or tested, but such things can be derived from the case. One can derive and form the basis for many parts of the development such as how the architecture should look like, how the software should be designed, how users should experience the system, and which tests should be created in order to investigate if the requirements are fulfilled. In “Quality Attribute-Guided Evaluation of NoSQL databases: A Case Study” [35], the requirements and use cases are created and categorized into a quantitative part and a qualitative part, which helps form the basis for the evaluation.

Evaluating NoSQL systems can be done by evaluating software quality attributes. A qualitative evaluation of these attributes may include an analysis and discussion of how well the systems satisfy the attributes. This demands a review of the systems, in which possibilities and limitations are discussed and compared. In “Choosing the Right NoSQL Database for the Job: A Quality Attribute Evaluation” [37], the authors list a number of software quality attributes relevant when evaluating the software quality of NoSQL systems. By using a use case, the most important attributes can be chosen for evaluation.

• Availability: Aims to describe how available the system is in terms of delivering a correct service.
• Consistency: Aims to describe the data consistency within a system. If a shard and its replicas all have the same data at the same time, they are in a consistent state.
• Durability: Aims to describe the validity of data after transactions.
• Maintainability: Aims to describe the degree to which it is possible to maintain a system. Is it possible to repair or update the system? Is it possible to modify it?
• Performance: Aims to describe the performance of different types of operations in the system. It could for example concern insertion, retrieval, update, and removal of data.
• Reliability: Aims to describe to which degree a system can function properly and deliver a correct service in the presence of failures. Which failures can it or can it not handle and what is the probability that critical failures occur?
• Robustness: Aims to describe how the system handles errors during execution.
• Scalability: Aims to describe how the system reacts to and handles an increasing amount of data and workload.
• Stabilization Time and Recovery Time: Aims to describe how long it takes for the system to recover after a node has failed and how long it takes to rebalance after a node has joined/rejoined the system.

As mentioned, a quantitative evaluation usually consists of quantifiable measurements, for example measurements of insertion and retrieval speeds, or low delay and waiting times during failover. The attributes that could be included in the quantitative evaluation are the same attributes as for the qualitative listed above [35].
2.11 Related Work

There exist a few performance evaluations on NoSQL systems, and mostly, the evaluations compare several independent solutions, for example MongoDB compared to Elasticsearch. In *Performance Evaluation of NoSQL Systems Using YCSB in a resource Austere Environment* [38], four different NoSQL systems have been compared concerning insert, read, and update operations, and MongoDB and Elasticsearch are two of them. The comparison is made with the Yahoo Cloud Serving Benchmark framework. The time measurements for the different operations are executed using up to 100 000 records at a time. The results show that with an increasing amount of data, MongoDB is faster at inserting data. However, concerning reads and updates, Elasticsearch is faster.

In *Performance optimization of applications based on non-relational databases* [7], performance tests on reads are performed on MongoDB and Elasticsearch. The results from *Performance Evaluation of NoSQL Systems Using YCSB in a resource Austere Environment* [38] are referred to in this article. The evaluation is performed reading a maximum of 900 documents from a database containing 3 500 documents. The results show that Elasticsearch is faster reading data than MongoDB, but no further evaluation is performed concerning inserts.

The results in these two articles are similar, in the sense that Elasticsearch is faster reading data than MongoDB. However, the second article is limited in the evaluation on inserts, and how insert performance stands in relation to read performance. The actual execution time is different in the results from these two articles, but they are using different tools for benchmarking, which might affect the outcome. Because of this, it might be difficult to compare these results. Both articles evaluate the performance on smaller datasets, the highest on 100 000 records.

When conducting performance tests, it is important that the tests are performed with a realistic workload, otherwise it may not reflect how well the system will perform in a real life setting. As mentioned in Section 2.10 “The Case for Application-Specific Benchmarking” [34] argues that performance must be measured within a context of a specific application. Just measuring performance because it is needed, provides little or perhaps no value at all. The article presents three ways of performing benchmarks with regards to the context of an application. In *Cutting Corners: Workbench Automation for Server Benchmarking* [39], the authors highlight the importance of relevant workloads for specific tests. A framework for workbench automation is introduced, which will ease and help developers perform benchmarking tests.

There is not much scientific material concerning a combined solution. MongoDB is suggested as a primary data store on its official website [3] and Elasticsearch is suggested as a secondary data store on its official website [4]. However, neither of the suggestions explicitly suggest combining MongoDB and Elasticsearch. Although, there are such explicit suggestions from companies who work with combinations of these tools [9] [8]. It is the absence of open research on the matter combined with the company interest which forms the basis for our hypothesis in section 1.2.
3 Method

In this chapter, the method used to carry out the work is described.

3.1 Use Case

In order to get an understanding of what is desired from the system, we created a use case together with iMatrics. This use case was then used as a reference point when evaluating the solutions and formed the basis for deciding which solution would best fit iMatrics' needs.

The system is intended to store large amounts of textual data. This data should be stored in a way that allows for full-text search queries in order to extract relevant information. The data storage is not supposed to be geographically distributed across the world at this point.

The system should be able to store large-size datasets which are managed separately. The number of datasets are going to increase over time. Each dataset can include several millions of documents (for example ten million) and be at most several gigabytes in size. Insertion of data is allowed to take some time before it should be available for querying, at most one day. The querying of data is based on full-text search queries. The full-text search queries has to be fast and be done in the matter of just a few seconds, ideally under one second. It should be possible to query the entire data as it is, or just parts of it that are relevant for further processing. The source data is intended to be inserted and queried, but not modified. Therefore, an operation such as an update is not a highly prioritized functionality. Since the end-result of the system is about performing analysis on large datasets, minor deviations during querying such as a few missing documents are allowed as this usually does not affect the end result of the analysis.

Data transformations are of interest and can be helpful for the analysis part. What kind of transformations that are of most importance depends on the structure of the data, and also on what kind of analysis is going to be performed. Since the analysis could be based on certain parts of a document, it is not always necessary to transfer every field. Projections are therefore of interest, since it could ease the retrieval and analysis of data. Depending on the structure of data, it could sometimes be appropriate to filter out certain documents, for
example if documents are categorized. The possibility to rename fields in a more appropriate way during transfer is also of interest.

3.1.1 Evaluation

From the use case, we could derive the most important aspects and attributes to evaluate when comparing each solution. The purpose of deriving attributes from the use case was to identify which attributes are relevant for evaluation. So even if the use case includes specific details for that solution, the general aspects of the attribute was considered, especially in the qualitative evaluation.

If we begin with the quantitative evaluation for the MongoDB and Elasticsearch, the following could be derived:

- Insertion and retrieval performance should be evaluated, especially since there are specific requirements relating to these metrics.
- The relation between insertion and retrieval is of interest, especially since a longer time for insertion than retrieval is allowed.
- Since the type of query that will be executed is full-text search, this is the query of interest when configuring the databases for insertion and retrieval. It is also the query of interest when performing measurements on retrieval.

If we then take a look at the qualitative evaluation for MongoDB and Elasticsearch, the following could be derived from the use case:

- Since a few missing documents during extraction usually do not affect the end result of the analysis, it is possible to have less constraints on the consistency. Therefore consistency is relevant for evaluation, in the sense of evaluating how configurable it is.
- Given that it should be possible to store several large datasets, with an increasing number of datasets over time, the solution should be able to scale in order to handle a greater workload. Therefore scalability is relevant for evaluation.
- The performance attribute is among the most important and the qualitative analysis in this case will mostly be used to explain and understand the results of the quantitative evaluation.
- In addition to the listed quality attributes, the full-text search functionality of MongoDB and Elasticsearch is also relevant for evaluation.

If we then take a look at the aspects of interest for transformation evaluation, the following could be derived:

- The time for transformation from MongoDB to Elasticsearch is going to be measured. The faster the better. This performance aspect is included in both the quantitative and the qualitative evaluation.
- Since what kind of transformation is interesting depends on the purpose of the analysis and the structure of data, it is important to investigate what transformations are supported by both NotaQL and Transporter. This aspect is included in the qualitative evaluation.
3.2 Data Modeling

Modeling the data in a sound and consistent way will help describe how both an individual document and an entire dataset is structured. The data considered in this thesis is a subset of a dataset that contains one month of discussion posts from an Internet forum\footnote{https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/}. The subset contains the first 10 000 000 posts from the larger dataset. Each post in the dataset consists of several information fields, such as author, information about the author, publication date, publication time, text body, and also some additional metadata fields.

Since MongoDB’s and Elasticsearch’s logical data models are not entirely equivalent in their syntax and structure, we used the NoAM model as the general model to describe the data structure, instead of using each system’s respective model. Using NoAM also allows for a neutral schema, instead of creating a schema biased towards one of the systems. Compared to the other models described in section 2.6 the NoAM model focuses more on abstract modeling, without initially taking the targeted system into account. This was preferable when evaluating several systems and needing a unified schema for both.

As described in Section 2.6 NoAM uses the concepts entry, block, collection and database to model the data. We have translated these concepts for the specific systems MongoDB and Elasticsearch in order to achieve specific data models that are as close to the general NoAM model as possible. An entry in NoAM has been translated to a field in both MongoDB and Elasticsearch and a block has been translated to a document for both MongoDB and Elasticsearch. Collections did not need any translation for MongoDB, but for Elasticsearch we have chosen to model it as a type. Similarly databases did not need any translation for MongoDB, but in Elasticsearch they have been modelled as indexes. A table with the translated concepts can be seen below in table 3.1.

<table>
<thead>
<tr>
<th>NoAM</th>
<th>MongoDB</th>
<th>Elasticsearch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database/Set of collections</td>
<td>Database</td>
<td>Index</td>
</tr>
<tr>
<td>Collection</td>
<td>Collection</td>
<td>Type</td>
</tr>
<tr>
<td>Block</td>
<td>Document</td>
<td>Document</td>
</tr>
<tr>
<td>Entry</td>
<td>Field</td>
<td>Field</td>
</tr>
</tbody>
</table>

Table 3.1: NoAM concepts

For the data used in this thesis, a NoAM entry represent an information field from a post, for example the text body. A block represent a post. A collection represent all posts, and a database is a set of collections, which in this case is just one collection that contains all posts. An example of a post, modelled as a NoAM block with entries, can be seen in Figure 3.1 below.
3.3 Java Client

MongoDB offers its own client, which can be connected to using a terminal. It also has its own Java library. Elasticsearch is provided with a HTTP web interface, which makes it possible to communicate with it from for example a terminal. As for MongoDB, Elasticsearch also has its own Java library. A Java client was written in order to simplify the test procedure. Two managers were created, one for MongoDB and one for Elasticsearch. The managers can perform insert operations and full-text search queries. The managers were created according to their respective Java API\(^2\)\(^3\).

The dataset used for insertion, is contained in a single file with all forums posts. Each line in the file is an individual post, and each post is formatted as a JSON object. During the insertion, we read from this file, and added the JSON objects to the bulk insertion request. For MongoDB, these JSON objects were first parsed into BSON format, since MongoDB uses BSON in their underlying storage mechanism. Once a bulk was finished, it was sent to the current system and processed.

The full-text search queries were listed in files, in which each line contained one query. When executing these queries, the Java client read one line at a time, and hence executed one query at a time. Both MongoDB and Elasticsearch returned the number of matching documents.

3.4 Hardware Specification

A total of four server nodes were available. Three with identical hardware specifications called server 1, 2 and 3, and one weaker server node called server 0. All were running Ubuntu 16.04.2 LTS. All servers were connected in a wired local network with TCP communication using Gigabit ethernet capped at one Gbit/s.

\(^2\)https://api.mongodb.com/java/3.2/
\(^3\)https://www.elastic.co/guide/en/elasticsearch/client/java-api/5.2/index.html
Server 0 has the following hardware specification:

- **CPU**: Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz
  - 2 cores, 2 threads
- **Memory**: 4x2GB DIMM DDR2 Synchronous 800 MHz (1,2 ns)
- **HDD**: 80GB Seagate ST380815AS

Server 1, 2 and 3 have the following hardware specification:

- **CPU**: Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
  - 4 cores, 8 threads
- **Memory**: 2x4GB DIMM DDR3 Synchronous 1333 MHz (0,8 ns)
- **HDD**: 500GB Western Digital WDC WD5000AAKX-7

3.5 **Quantitative Evaluation of MongoDB and Elasticsearch**

In order to perform a quantitative evaluation, experiments were performed to measure insertion and full-text search query speeds. Focus in this section is the experimental design, how the experiment was carried out, how the measurements were collected, analyzed, presented, and finally on what basis the interpretation was carried out.

### 3.5.1 Exploratory Testing Plan

MongoDB and Elasticsearch are highly configurable. Some of the settings are hardware specific and therefore exploratory testing was used to determine the correct values for these settings. The exploratory testing was also used to verify unscientific statements and test the impact of some of the more important settings as presented by various optimization guides. The settings and guides are included in the results.

An exploratory approach in combination with scripted testing was selected. Scripts were used in order to test different insert and full-text search query scenarios, and the scenarios were specified and updated during the test sessions. A test charter was created, which included the main goals, and some initial approaches. The charter acted as a basis for the test sessions and is listed below:

- Separate test sessions for MongoDB and Elasticsearch.
- The main goal is to achieve fast insertion and full-text search query speeds and investigate the impact of different system configurations.
- Investigate the effects on document insertion speed when changing the bulk size.
- Review and analyze the setup for a sharded cluster and try to find configurations that affect insertion and full-text search query speed.
- Increase the amount of documents inserted/queried when testing and review the changes in speed.
- Review different approaches available in Elasticsearch and MongoDB when searching for documents.
3.5.2 MongoDB Test Environment

The test environment for MongoDB was set up using a sharded cluster with a maximum of three shards. How many shards used during testing is defined by the scenario under test. Each shard was placed on its own node. The query router and the config server was placed on a server called the application server, which could be either weak or strong, depending on the hardware used (see Section 3.4). When replication was used, the replica was placed on its own node. The MongoDB version used was 3.2. An overview of the setup can be seen in Figure 3.2.

Figure 3.2: MongoDB test environment

Default Settings MongoDB

All settings not mentioned are set to their default values. All settings which have been identified as relevant for our tests are presented with their default values.

Sharded cluster balancer:

- The balancer is by default enabled. The balancer monitors all shards. On each shard, the data is split up in several chunks. The size of the chunks, and how many to use, is configurable. The balancer also monitors the chunks, and when the number of chunks are uneven among the shards, it will rebalance the chunks with the goal of an even distribution.

- The default number of chunks created per shard is 2.

- The default chunk size is 64 MB. The minimum size is 1 MB and the maximum is 1024 MB.

- When a chunk exceeds its maximum size, MongoDB automatically splits that chunk into several new chunks. This is not the work of the balancer. The balancer might however start balancing these new chunks.
3.5. Quantitative Evaluation of MongoDB and Elasticsearch

General settings:

- The default storage engine used is WiredTiger.
- Default number of shards in a cluster is 0. Shards need to be added manually into the query router and the config server.
- When inserting with bulks, the insertion is by default done ordered (in serial).

3.5.3 Elasticsearch Test Environment

The test environment for Elasticsearch was also set up using a sharded cluster using a maximum of three shards, each placed on its own node. How many shards that are used during testing is defined by the scenario under test. On the application server, an Elasticsearch query router was placed, which keeps track of all shards and queries. Compared to MongoDB, the Elasticsearch query router acts as both a query router and a config server. When replication was used, the replica was placed on its own node. The Elasticsearch version used was 5.2. An overview of the setup can be seen in Figure 3.3.

![Figure 3.3: Elasticsearch test environment](image)

**Default Settings Elasticsearch**

All settings not mentioned are set to their default values. All settings which have been identified as relevant for our tests are presented with their default values.

Sharded cluster balancer:

- If one or more shards receive more data than others, the system can perform rebalance operations to even it out. By default this function is turned on.

General settings:

- By default Elasticsearch stores the source JSON data in a special field called _source. It also stores a special _all field which allows for searching through all fields at the same time. These fields can be turned off.
3.5. Quantitative Evaluation of MongoDB and Elasticsearch

• By default a database is created using 5 shards, 1 replica set and an index refresh interval of 1 second.

• Thread pools are enabled by default and are based on the number of available cores on each node.

3.5.4 Test Procedure

The comparisons were carried out based on the metrics of insertion and full-text search query speed, set in different scenarios. There are several parameters that affect the number of scenarios. First of all, we have the data volume. As introduced in Section 3.2, the dataset consists of 10 000 000 discussion posts from an Internet forum. Each post is roughly the same size, but there are some that are larger than others. This means that we have a total number of 10 000 000 documents, which is 5.4GB in size. The decision was made to divide the data into three sizes. The largest size is the total number of documents. The next size, which is called medium, consists of the first 10% of the total number of documents, and is 540MB in size. The third size, which is called small, consists of the first 1% of the total number of documents, and is 54MB in size. (Note that the 10% and 1% are the percentages of the total number of documents, and not the percentages of the actual total data size 5.4 GB.)

\[ D_1 : \text{Small data} = 100 000 \text{ posts, 54MB} \]
\[ D_2 : \text{Medium data} = 1 000 000 \text{ posts, 540MB} \]
\[ D_3 : \text{Large data} = 10 000 000 \text{ posts, 5.4GB} \]

The next parameter is the number of nodes. For the experiments we had a total of four nodes available (see Section 3.4). Three of these nodes had identical hardware specification, while the fourth was less powerful. The hardware specification of a node is an important aspect. Using nodes with less powerful hardware might reduce performance as compared to using a node with more powerful hardware. Therefore, we tested using both different numbers of nodes and different hardware specifications. The fourth weaker node is only used as a query router. One of the other three are in some scenarios used as a stronger query router, and in others as a shard node. The other two are only used as shard nodes. This means that we have either a weak query router or a strong query router.

\[ Q_1 : \text{Weak query router} \]
\[ Q_2 : \text{Strong query router} \]

We then get a maximum of three shard nodes.

\[ N_1 : \text{One shard} \]
\[ N_2 : \text{Two shards} \]
\[ N_3 : \text{Three shards} \]

Another parameter that might affect the performance is replication. Using replicas is usually done in order to increase the availability, since the replicas can take over and manage reads and writes if the primary fails. Therefore we performed tests with and without replication.

\[ R_1 : \text{Without replication} \]
\[ R_2 : \text{With replication} \]

With the mentioned parameters above, we end up in the following test environment scenarios, which test both the difference in hardware specification and in number of shard nodes used.
3.5. Quantitative Evaluation of MongoDB and Elasticsearch

$S_1$: One shard + weak query router + without replication ($N_1 + Q_1 + R_1$)

$S_2$: Two shards + weak query router + without replication ($N_2 + Q_1 + R_1$)

$S_3$: One shard + strong query router + without replication ($N_1 + Q_2 + R_1$)

$S_4$: Two shards + strong query router + without replication ($N_2 + Q_2 + R_1$)

Due to the limited amount of hardware, using replication with two shards was not possible. However, measuring the effects of replication was still interesting, even if it is done with one shard. This leads to the following scenarios, which test the effects of replication and how it is affected by having a weak and a strong query router:

$S_5$: One shard + weak query router + with replication ($N_1 + Q_1 + R_2$)

$S_6$: One shard + strong query router + with replication ($N_1 + Q_2 + R_2$)

Since we have the possibility to use three shard nodes together with a weak query router, this results in an additional scenario:

$S_7$: Three shards + weak query router + without replication ($N_3 + Q_1 + R_1$)

**Insertion**

For the insertion tests, the test environment scenarios $S_1$, $S_2$, $S_3$, $S_4$, $S_5$ and $S_6$ were used. These scenarios tested how the performance was affected by increasing the number of nodes, using a stronger query router, and replication. These tests are also connected to the scalability aspect. All scenarios were used to execute tests on all three data sizes $D_1$, $D_2$ and $D_3$, and resulted in 18 test cases.

All tests were executed using MongoDB and Elasticsearch, and also repeated 20 times. The repetition of tests was in order to mitigate large effects caused by uncharacteristic behavior, for example network deviations.

**Full-Text Search**

For full-text search queries, there were two types of queries. The first was a simple query type containing a list with single search words (See Appendix A.1). The second was a more complex query type containing a list with multiple search words and four different kind of regular expressions: 1) AND, 2) OR, 3) Term exclusion and 4) Exact phrase (See Appendix A.2 - A.3). Each list consisted of 100 queries.

$T_1$: Simple query type

$T_2$: Complex query type

The test environment scenarios used were $S_1$, $S_2$, $S_3$, $S_4$ and $S_7$. These scenarios tested how the performance was affected by an increasing number of nodes and a stronger query router. All scenarios were used to execute tests on both types of queries, $T_1$ and $T_2$, and on all three data sizes $D_1$, $D_2$ and $D_3$. This resulted in 30 test cases. Before executing each query list, another simple query list was executed in order to warm up both MongoDB and Elasticsearch. That list also included 100 queries, and those queries were not included in any of the other two lists.

In order to get a wide range of queries, we used Google Trends[4] as a basis to retrieve popular searches from the same month the forum posts were written. Common random words were also added to fill up some gaps.

3.6 Qualitative Evaluation of MongoDB and Elasticsearch

To evaluate the attributes scalability, consistency and performance, we compared the API and other documentation for each system. We also include related research and relevant quantitative results. Even though these attributes cover different software quality aspects, there exist relationships between them. Each attribute was analyzed and evaluated.

There are only a few scientific attribute evaluations on NoSQL systems. For Elasticsearch, "Mining Modern Repositories with Elasticsearch" [40] was found. For MongoDB, "Choosing the right NoSQL database for the job: a quality attribute evaluation" [37] and "Quality Attribute-Guided Evaluation of NoSQL Databases: A Case Study" [35] were found, and these evaluations were used as a basis for the qualitative evaluation. The comparison with Elasticsearch was performed using the API, manual pages, performance experiments and also blog posts based on user experience.

3.6.1 Consistency

Consistency is an essential part of NoSQL systems. In addition to consistency being derived as a quality attribute interesting to analyze, it is also included as a property in the CAP-theorem, ACID and BASE. The API for each system was used to compare theoretical functionality and limitations, and these were then compared with previous research and blog posts. The evaluation of consistency was focused on what degree of consistency is possible and at what cost.

MongoDB ensures consistency through its core functionalities regarding isolation, consistency and recency. These are described in the manual "Read Isolation, Consistency, and Recency" [41]. The architecture of MongoDB is described in "MongoDB Architecture Guide" [42]. Elasticsearch’s Alex Brasetvik has written a blog entry on using Elasticsearch as a NoSQL DBMS [4] in which he describes it as a CP system, but describes the consistency using the term "for a fairly weak definition of "consistent"". The two previously mentioned articles "Choosing the right NoSQL database for the job: a quality attribute evaluation" [37] and "Quality Attribute-Guided Evaluation of NoSQL Databases: A Case Study" [35] cover parts of MongoDB’s consistency aspects.

3.6.2 Scalability

The scalability aspect is highly relevant for NoSQL systems, especially since NoSQL offers horizontal scalability. The API for each system were used to compare the offered functionality. The focus here lies on how scalability is implemented in each solution, what configurations are available, and how it affects other parts of the system, for example how already existing data is affected by adding and removing nodes. Performance tests were used to measure the effects of increasing the number of nodes.

MongoDB advertizes that it can handle scaling for big data [43] and Elasticsearch has an entire section of their guide dedicated to scale [44]. However Elasticsearch’s guide on scale is only for version 2.x at this moment and therefore some of it may be outdated since we used version 5.2.

3.6.3 Performance

The performance aspect was primarily evaluated using the quantitative experimentation, but these results were also compared with previous research. The results from the experimentation was also discussed in relation to the application context, settings used, and the API documentation [45] [46]. The related performance evaluation articles which we used to compare
3.7 NotaQL Extension

The extension of NotaQL for Elasticsearch is based on documentation on Apache Spark for Elasticsearch and a combination of the already existing NotaQL engines for JSON and MongoDB. The Apache Spark API contains specific methods for the use of Elasticsearch as a connected data store. The language structure for document-oriented databases was already in place for MongoDB and JSON.

3.8 Quantitative Evaluation of NotaQL and Transporter

For the transformation testing, we first of all identified possible transformations that could be performed using both NotaQL and Transporter. Both NotaQL and Transporter have a few native transformations (See Section 2.7.1 and 2.7.2). There are three types of transformations that could be used in both NotaQL and Transporter for the type of data considered in this thesis. These are rename, projections, and filters.

3.8.1 Transformation Test Environment

The test environment can be seen in Figure 3.4. MongoDB’s query router and config server, along with Elasticsearch’s query router, NotaQL, and Transporter, were all placed on a strong application server. Elasticsearch’s and MongoDB’s databases were each placed on its own node. All nodes have the same technical specifications.

3.8.2 Test Procedure

The comparison between NotaQL and Transporter was carried out by executing the three identified common transformations, and combination of these transformation. The execution
times were measured, and executed on all the different data sizes $D_1$, $D_2$ and $D_3$. The chosen transformation tests are listed below:

$TR_1$: Projection: This transformation transfers all documents, but only projects two fields.

$TR_2$: Filter: This transformation matches on a specific value on a field, and filters out the documents that do not match.

$TR_3$: Filter + Projection: This transformation matches on a specific value on a field, filters out the documents that do not match, and projects two fields.

$TR_4$: Filter + Rename: This transformation matches on a specific value on a field, filters out the documents that do not match, and renames one of the fields.

$TR_5$: Projections + Rename: This transformation transfers all documents, projects two fields, and renames one field.

$TR_6$: Filter + Projections + Rename: This transformation matches on a specific value on a field, filters out the documents that do not match, projects two fields, and renames one field.

$TR_7$: Multiple Filters: This transformation filters according to two criteria, instead of one.

$TR_8$: Rename: This transformation transfers all documents and renames all fields.

Each transformation was executed on each data size and repeated 20 times in order to mitigate large effects caused by uncharacteristic behavior, for example network deviations. The expressions for the NotaQL transformations can be seen in Appendix B.1, and the scripts for the Transporter transformations can be seen in Appendix B.2.

### 3.9 Qualitative Evaluation of NotaQL and Transporter

The main quality attribute evaluated in NotaQL and Transporter is the performance. For this, we compared the API for each software and also included relevant quantitative results. There are some scientific articles about NotaQL, which mainly include extensions and different use cases. We have however not found any performance evaluation on NotaQL. For Transporter, we have not found any scientific articles, neither some performance evaluations. The literature basis for Transporter is the API and blog posts from Compose’s website.

In addition to the performance attribute, we have also evaluated the possibility to create and execute additional transformations and what the main differences are in implementing them in NotaQL and Transporter.
In this chapter, the results are presented. The chapter starts with the quantitative evaluation of MongoDB and Elasticsearch, followed by the qualitative evaluation of MongoDB and Elasticsearch. After that, the extension of NotaQL is presented. Finally, the evaluation of NotaQL and Transporter is presented, starting with the quantitative evaluation followed by the qualitative evaluation.

4.1 Quantitative Evaluation of MongoDB and Elasticsearch

The quantitative evaluation of MongoDB and Elasticsearch is divided into two major parts. First, the exploratory testing is presented, which shows how the systems were configured according to the use case. The exploratory testing ends up with a list of test settings used when executing all test cases. The second part presents the quantitative measurements from the test cases.

4.1.1 Exploratory Testing

The exploratory test goals were presented in the method in section 3.5.1. Originally the exploratory testing started with default settings.

Initial Insertion Speed

As a first step we tested insertion speed when inserting a single document at a time. These insertions took a long time to execute on large datasets. An example test case took one and a half hour when inserting ten million documents into MongoDB, with a strong query router, and one shard node. The next step was to try and speed up the insertion by using bulks [47][48]. This resulted in a significantly reduced insertion time and several million documents could be inserted in a matter of minutes. Different bulk sizes were tested in order to find one with the fastest insertion speed. In the Java client, the bulk size for MongoDB was in number of documents, while it for Elasticsearch was in MB. For MongoDB, the bulk size was set to 5 000 documents, and for Elasticsearch, the bulk size was set to 3 MB, which is roughly equal to 5 000 documents.
4.1. Quantitative Evaluation of MongoDB and Elasticsearch

In order to do a full-text search in MongoDB it is necessary to have a text-index on the content that is going to be searched. Text-indexes are not part of MongoDB’s default settings, but it is included in Elasticsearch’s equivalent method called analyze. The next step was to investigate the impact on insertion when adding a text-index for MongoDB. MongoDB text-index tests showed that indexing during insertion was slower than indexing the entire collection afterwards. In one test case, when inserting 1 000 000 documents with a strong query router and one shard node, inserting with indexation during insertion took 197 seconds. Inserting and indexation after insertion took 111 seconds in total. This shows that indexation after insertion was 44% faster. The next step was to add more nodes and replicas. The collections were sharded across three nodes with three replica sets. Elasticsearch has an automatic strategy for distributing data across shards. Elasticsearch routes documents based on a mathematical formula, $\text{shard} = \text{hash}(\text{routing}) \% \text{number of primary shards}$. The default routing value is the document’s _id field [49]. In MongoDB, a shard key has to be chosen. After that, MongoDB shards the data automatically with the help of the balancer. The key is based on the value of a field. There are two strategies in MongoDB for sharding based on the key. Either the shard key is the actual value of the field, and the data is sharded based on the range of these values, or the shard key is the hashed value and data chunks get a range of the hashed key. Since Elasticsearch hashed on id-field values, MongoDB was also set up with a hashed shard key based on id-field values. MongoDB seems to use the MD5 hash function [50] and Elasticsearch uses the Murmur3 hash function [51].

Large Insertions on Multiple Nodes

In order to test large insertions on multiple nodes, a weak query router with three shard nodes were initially used. The tests showed that MongoDB slowed down while performing a lot of insertions and that the number of documents fluctuated afterwards. There were at first more documents afterwards than had been inserted. The slow insertion speed was partially caused by the creation of new data chunks. By default, MongoDB has a few data chunks with a size limit of 64 MB. With an increasing amount of documents, new chunks have to be created. The fluctuation of data occurred because of MongoDB’s balancer, which migrates data chunks. The balancer creates copies of documents and move them to another node in order to achieve an even distribution [52]. When the copies have been moved, the original documents are removed and the total number of documents goes back to normal. With a large dataset, the creation and migration of chunks becomes an issue due to the increased overhead related to full chunks. In order to repeat the tests, we had to wait for the rebalancing to finish. As rebalancing is part of the default settings for MongoDB, we tested switching off the rebalancing, which removed this migration. Switching off the balancer resulted in an uneven distribution of data, which would never be evened out and that is a problem as it meant that some nodes would be used less than others. Since the balancer did not migrate chunks, a majority of the collection was stored on one shard, while the other shards had a much smaller part of the collection. Therefore a strategy for shard allocation based on a field containing categories was implemented, instead of id-field values. On each shard, we also created more chunks in advance, and split the shard key value range among these chunks, in an attempt to evenly distribute documents based on category [53]. This shard allocation strategy resulted in a distribution of roughly 35% of the inserted documents on the first two shard nodes and the remaining 30% of the documents on the last shard node. We considered this distribution to be sufficiently even. Elasticsearch did not experience this behavior, and each test in a series of multiple consecutive insertion tests had roughly the same execution time. This means that either the distribution or the rebalancing worked better in Elasticsearch. However even if Elasticsearch’s hash function was better, we did not deem it to be within our scope to change the hash function in MongoDB. This decision was based on the fact that we did not find any documentation related to switching out the hash function in MongoDB without changing the source code.
At this point MongoDB had started to take up the entire RAM on all nodes when inserting just a few million documents, which affected the overall performance on the servers. Elasticsearch was still able to perform its tasks. Inserting documents when the RAM was full took a lot longer time to complete.

**Replication**

When investigating the issue of the RAM being filled up very quickly, we came to the conclusion that replicas were a part of the problem. Therefore replicas were not placed on the same shard nodes as the primaries during performance tests. Settings to throttle replica actions exists, but is left for future studies and/or live deployments.

**Full-Text Search**

Elasticsearch was discovered to not have a stemmer enabled by default and therefore we enabled a simple snowball stemmer using the provided English language option [54]. Tokenization and regular expressions were enabled by default for string queries. MongoDB has tokenization, stemming, and regular expressions enabled by default for its searches on text-indexed documents [55]. MongoDB also uses the snowball stemmer [56].

**Optimization/Fine-Tuning**

Following the tips of several guides for speeding up the performance of both MongoDB and Elasticsearch, we performed several tests to fine-tune the settings for both systems. Unordered insertions were switched on for MongoDB to allow for multiple concurrent operations across shards while performing bulk insertions [47]. Concurrent operations were activated for Elasticsearch and the number of allowed concurrent [48] operations that improved performance was tested and determined to be 8, as many as the number of logical cores on the data nodes. The allocated RAM for Elasticsearch was increased. Since the full-text search is primarily interesting for fields with a lot of text, MongoDB’s text-index and Elasticsearch’s analyze was configured to only process the field "body" which contains the text body of the documents. Elasticsearch uses a refresh interval setting in order to schedule the preparation of new operations in order to allow for new data to be searched in near real-time. During insertion this refresh interval is disabled [48].

The insertion speed of MongoDB was found to have a bottleneck (at least) in the Java client API in the form of a need to parse the JSON strings to BSON before insertion. This operation is performed on the query router and therefore the benchmark tests include parse measurements when using both a strong query router and a weak query router. To compare the strong and weak query routers equally, the number of shard nodes were limited down to two, instead of the three nodes used in parts of the exploratory testing.

**Test Settings**

Here we present a summary of non-default settings used for the benchmark tests.

- Bulk insertions are used in accordance with the use case.
- Bulk size is set to 5000 documents for MongoDB and 3 MB for Elasticsearch.
- Text-index for MongoDB is added after insertions.
- Replicas are not placed on the same shard nodes as the primaries in the benchmark tests.
• The MongoDB balancer is disabled.

• MongoDB chunks are split based on a field which contains forum categories, which results in three shards having a roughly 35%, 35% and 30% split.

• Elasticsearch’s refresh_interval is set to -1 during insertion, which means that Elasticsearch does not prepare new data for search at that time.

• The Java Virtual Machine (JVM) was granted a heap of 4 GB, which is equal to half the available system memory for our hardware.

• We enabled 8 concurrent operations for Elasticsearch’s bulk insertion.

• We enabled unordered operations for MongoDB’s insert many (bulk insertion) to allow for concurrent operations.

• MongoDB’s text-index is only enabled for the field body. In Elasticsearch this is called analyze, but is roughly equivalent and also enabled for just the field body.

• The number of chunks enabled before insertion in MongoDB is 202.

• A snowball stemmer was added to the Elasticsearch analyze settings.

4.1.2 Test Results

In this section, the measurements for insertion and full-text search are presented.

Insertion

The insertion procedure for MongoDB and Elasticsearch were not identical. First of all, before inserting data into MongoDB using Java, the data (which consists of JSON strings) needs to be parsed to BSON format. Elasticsearch on the other hand, could insert these JSON strings without any parsing. Secondly, as mentioned in section 4.1.1, text indexation after insertion into MongoDB was much faster than text indexation during insertion. Therefore we measured the indexation time after insertion. Elasticsearch analyze on the other hand, was performed during insertion. In Figures 4.1 - 4.2 below, we can see the difference between using a strong query router and a weak query router when inserting documents. The insertion times presented in the graphs are the average of 20 test executions. There are two types of lines for MongoDB in the graphs. There is one line that only includes insertion and indexation. Then there is another line that includes insertion, indexation, and parsing.
4.1. Quantitative Evaluation of MongoDB and Elasticsearch

Figure 4.1: Insertion Strong Query Router

Figure 4.2: Insertion Weak Query Router

We can first of all see that Elasticsearch is in average faster than MongoDB for all data sizes $D_1$, $D_2$ and $D_3$, and all scenarios $S_1$, $S_2$, $S_3$ and $S_4$. Using two shard nodes instead of one, increased the performance for both MongoDB and Elasticsearch. If we compare the two graphs and look at the time difference between strong and weak query router, there are differences. However, relative to the respective insertion time for Elasticsearch and MongoDB without the parsing time, these differences can be considered minor. The largest difference in those four cases is for Elasticsearch with two shard nodes and the large dataset ($S_2$ and $S_4$ with $D_3$). The tests with a strong query router is in average 45 seconds faster than with a weak query router. In Appendix C.1, we have illustrated the maximum and minimum insertion times for all tests. When using both MongoDB and Elasticsearch with two shard nodes on the small and medium datasets ($S_2$ and $S_4$, with $D_1$, $D_2$), the maximum insertion time using
4.1. Quantitative Evaluation of MongoDB and Elasticsearch

A strong query router was faster than the minimum insertion time using a weak query router. In these two test cases the strong hardware was always faster, but this was not the case for the other tests. Similar results were found when comparing the number of nodes which can be seen in Appendix C.2. These results indicate that more powerful hardware does not always guarantee better performance.

If we then take a look at the lines for MongoDB with the parsing time included, we can see the impact when parsing JSON strings to BSON format. First of all, the total time for insertion in MongoDB is much longer with the parsing time included. When the data size increases, so does the time it takes to parse the JSON strings. This is expected since an increasing number of documents demands more parsing operations. There are no major differences between using one or two shard nodes. However, there is a major difference between using a strong or a weak query router. Using a strong query router is faster, which is also expected since the hardware is more powerful and should be able to manage parse operations at a faster pace.

Below in Figures 4.3 - 4.4, we can see the effects of replicating data during insertion using a strong and a weak query router. The time for MongoDB in the graphs includes parsing, insertion and indexation.

![Figure 4.3: Replication Strong Query Router](image)

Figure 4.3: Replication Strong Query Router
4.1. Quantitative Evaluation of MongoDB and Elasticsearch

The replication of data is performed actively after inserting into the primary, which means that the replica node has to finish writing data before an acknowledge could be returned. It is the primary that manages the replication of data to the secondary. Replicating data increases the insertion time for both MongoDB and Elasticsearch. This is clearly shown for the large data size. The actual increase in insertion time is larger for Elasticsearch than MongoDB, but Elasticsearch is still faster. For MongoDB, it is the insertion time that has increased, while the parsing and indexation times are roughly the same.

There are no major differences between using a strong and a weak query router, except for the MongoDB parsing which is affected in the same way as for previous tests.

Full-Text Search

The full-text search tests were executed on two types of queries, simple $T_1$ and complex $T_2$ (See Section 3.5.4). If we start by looking at Figures 4.5-4.6, we see the differences between the strong and weak query router when executing simple queries.

![Figure 4.4: Replication Weak Query Router](image)

Figure 4.4: Replication Weak Query Router
Several observations can be made by looking at these graphs. First of all, we see that the execution time for MongoDB with a weak query router and one shard node ($S_1$ and $S_3$), exceeds the graph bounds when executed on the large dataset ($D_3$). The time it took executing these queries were in average 8 seconds for the weak query router and 11 seconds for the strong query router. The likely explanation is that MongoDB exceeds maximum memory usage at 8GB. MongoDB stores as much data as possible in memory, and when exceeding maximum memory usage, MongoDB swaps data from the harddrive. The large dataset was 5.4GB in size, and in addition to that, MongoDB also adds several GB of indexes. However, when using more than one shard node ($S_2$ and $S_4$), only half of dataset is stored on each shard node, which is enough to not exceed the maximum memory usage.
Elasticsearch stores data in a different way as compared to MongoDB. Elasticsearch primarily stores data on the harddrive, but keeps mappings of data in memory. As can be seen from the tests, these mappings seem efficient and allow for fast full-text searches. Another observation that can be made is that Elasticsearch becomes faster than MongoDB on the large dataset when using a strong query router and two shard nodes ($S_4$). In Figures 4.7 - 4.8 below, we see the difference between using a strong and a weak query router when executing complex queries.

![Weak Query Router Complex Queries](image)

**Figure 4.7: Weak Query Router Complex Queries**

![Strong Query Router Complex Queries](image)

**Figure 4.8: Strong Query Router Complex Queries**

For complex queries, MongoDB with one shard node ($S_1$ and $S_3$), still takes a long time to
execute on the large dataset. However, MongoDB now also experiences this behavior using two shard nodes \((S_2 \text{ and } S_4)\). Executing these queries with one shard node took 98 seconds in average when using a weak query router and 91 seconds when using a strong query router. With two shard nodes, it took 24 seconds in average when using a weak query router and 34 seconds when using a strong query router. It also takes longer time for MongoDB to execute the queries on small and medium datasets, compared to simple queries. Elasticsearch however, is actually faster in some cases, at executing complex queries than simple ones.

An additional noteworthy observation in Figures 4.5 - 4.6 as well as in Figures 4.7 - 4.8 is that for MongoDB it does not seem to matter whether the query router is on a weak node or on a strong node. In contrast, for Elasticsearch, the computer power of the node that runs the query router makes an observable difference, both positive and negative. Elasticsearch with one shard node using a strong query router, is getting slower than using a weak query router on the large dataset. Meanwhile, Elasticsearch with two shard nodes using a strong query router, is faster than using a weak query router.

The last full-text search test performed was using a weak query router and three shard nodes \((S_7)\). The average time it took to execute simple and complex queries in MongoDB and Elasticsearch can be seen below in Figure 4.9.

One observation which can be made from these tests is that MongoDB still takes a long time to execute complex queries when compared to Elasticsearch, but with three shard nodes, the time has decreased significantly. However, when executing simple queries, MongoDB is faster than Elasticsearch for all tested data sizes.

**4.2 Qualititative Evaluation of MongoDB and Elasticsearch**

In this section, the qualitative evaluation of MongoDB and Elasticsearch is presented, which evaluates the quality attributes consistency, scalability and performance.
4.2.1 Consistency

MongoDB supports different levels of consistency, which depend on replication settings [42]. In a replica set, there is one primary and one or more secondaries, which all have a replica of the data. There are two settings used to configure the consistency and availability, 1) when an acknowledgement for a write operation should be returned, and 2) the read responsibilities within a replica set. A write operation can be acknowledged either when the primary is done, or when all secondaries are done, or even when a subset of the secondaries are done. Read responsibilities within a replica set can be configured in a way so that only the primary responds to reads, or so that the secondaries also can respond to reads. The default setting in MongoDB is that only the primary responds to reads.

Elasticsearch is similar to MongoDB in this aspect, and also supports different levels of consistency. The settings are also the same, 1) how many secondaries that should finish writing data before acknowledging the write operation, and 2) the read responsibilities between a primary node and its secondary nodes [57] [58].

In order to achieve strong consistency, only the primary node should be allowed to respond to reads, and a write operation should be acknowledged when all secondaries are done. However, if the secondaries are allowed to respond to reads, while a write operation is acknowledged when the primary is done, there is a possibility that the primary and the secondaries could return different data. In this case, the primary and its secondaries are in an inconsistent state, but they will eventually reach a consistent state. This is a typical characteristic of NoSQL systems, known from the BASE properties as Eventual Consistency. Here, we can also see the system behavior known as Soft State, that the state of the systems changes over time, for example when a replica set reaches a consistent state.

Another aspect connected to consistency is that of ACID transactions [37]. MongoDB supports ACID transactions at document level, which means that a write or update operation concerning one document is either fully complete, or not made at all [42]. If an error occurs, the system will perform a rollback and the operation will not be performed. However, if the operation concerns several documents, it will not be performed as an ACID transaction. For example, if an error occurs during an update operation of six documents, and only the first three documents have been updated. The other three will have invalid data. There are however procedures that could be used to achieve ACID-like transactions. For example, MongoDB has an operator called isolated, which prevents other processes to interfere and does not allow other processes to read any changes until all operations are complete [41]. The flaw with the isolated operator is that if an error occurs, it does not perform a rollback. It could help solve the problem of reading different data, but it would not solve the problem of invalid data. Unfortunately, the isolated operation does not work in a sharded cluster.

Another way of achieving ACID-like transactions in MongoDB is by using the Two-Phase Commit procedure [59]. Simply put, the Two-Phase Commit procedure logs the status of all write or update operations in fields at the targeted documents or in fields at supporting documents. In this case, the transaction is divided into several ACID transactions, which are executed serially. By monitoring the status of the operations, this procedure allows for recovery after failures, both in the form of a rollback or in the form of a resumption. It could also be used to check the validity of the data in scenarios where a read operation is executed during the write operation.

Elasticsearch also supports ACID transactions at document level, but not for operations concerning multiple documents. There are however ways of achieving ACID-like transactions, for example with the use of locks [60]. It is possible to lock data and prevent other
processes from interfering, for example locking the entire database, a certain collection or individual documents. Based on how the Two-Phase Commit procedure works in MongoDB, it should be possible to implement it for Elasticsearch as well, even though no documentation was found on that specific procedure.

Given the different levels of consistency offered by MongoDB and Elasticsearch, we can see some possible CAP combinations. Settings in which the primary responds to all writes and reads, and a write or update is acknowledged when all secondaries are done, results in stronger consistency at the cost of weaker availability. This means that both MongoDB and Elasticsearch can be configured as CP systems. However, if secondaries are allowed to respond to reads, and a write or update is acknowledged when the primary is done, this results in stronger availability at the cost of weaker consistency. This means that both MongoDB and Elasticsearch can be configured as AP systems.

The authors in "Choosing the right nosql database for the job: a quality attribute evaluation" [37] have classified MongoDB’s consistency as Great on a scale: Unknown, Bad, Mediocre, Average, Good or Great. The classification is based on how much the system supports ACID semantics and the possibilities to configure consistency. According to the authors, MongoDB is the one system of all the evaluated NoSQL systems in the article, that is most similar to RDBMS when it comes to its strengths, which are reliability, durability, and consistency. Its weaknesses are availability and scalability. The weaknesses are mostly based on the slow stabilization time MongoDB has when nodes rejoin a cluster. MongoDB also states, in the white paper "MongoDB Architecture Guide" [42], that they are building functionality based on RDBMS techniques.

### 4.2.2 Scalability

MongoDB offers horizontal scalability, and mainly through the means of automatic sharding across several shards. As we have seen in the exploratory testing part of the thesis, MongoDB balances and migrates data chunks across all shards in order to achieve an even distribution of data. This makes MongoDB’s scalability good, since a user does not necessarily have to focus on this part. However, we have noticed that this automation can definitely reduce the performance when working with large datasets, especially write performance. MongoDB offers different sharding strategies, in which it is possible to plan the sharding distribution in advance. Such a strategy can increase the performance, while it at the same time removes a lot of the automation and makes the setup procedure more complex [37].

Elasticsearch also offers horizontal scalability through automatic sharding. One of the main differences between automatic sharding in MongoDB and Elasticsearch is that MongoDB requires manual adding of shards to the cluster, while Elasticsearch finds available shards on its own. According to "Mining Modern Repositories with Elasticsearch" [40], scalability is one of Elasticsearch’s strengths. Elasticsearch will also balance the data between existing shards when inserted, with the purpose of an even data distribution. During the exploratory testing part of the thesis, we found it difficult to configure the sharding strategy for Elasticsearch. The data was evenly distributed with the default strategy and did not noticeably affect the write performance. This supports the statement that the scalability aspect might be one of Elasticsearch’s strengths.

There is another difference between MongoDB and Elasticsearch concerning scalability. In Elasticsearch, the number of primary shards are set when the database is created, and there is no possibility to add additional primary shards to an already existing cluster [61]. A workaround to this problem is to re-index the entire database. This procedure can be expensive, depending on how much data there is in the database. Elasticsearch can however
add additional secondaries to an already existing cluster. The purpose of adding another secondary could be to increase the throughput on read requests, since secondaries are allowed to respond to reads.

MongoDB however, can add additional primary shards to an already existing cluster. When an additional primary shard is added, the balancer starts migrating data chunks in order to achieve an even distribution of data. This might be expensive to perform and can take a long time to complete. Enough resources are needed or else the entire service might severely decrease in performance.

4.2.3 Performance

If we start with the quantitative evaluation in this thesis, the tests showed that Elasticsearch was faster at inserting data, for every scenario. Elasticsearch was also faster for some scenarios, at performing full-text search. However, MongoDB was faster concerning simple queries, especially on smaller data sizes and with an increasing number of shard nodes. Since MongoDB was slower than Elasticsearch inserting data, but in some scenarios faster at performing full-text search, there might be a connection between these two aspects. A faster full-text search with simple queries might be achieved at the cost of a slower insertion and indexation speed.

The faster full-text search on smaller data sizes could also be an effect by MongoDBs way of storing as much as possible in memory. This could then also explain why an increasing number of nodes resulted in better performance, since an additional node effectively doubles the maximum memory. If the same amount of memory used on three shard nodes, was used on one shard node, that node might be as fast as three. However, adding resources to a single machine is the way vertical scalability handles the problem with an increasing amount of data. The tests show that horizontal scalability works fine as well.

As described in the qualitative evaluation of consistency and scalability, better performance might be possible to achieve with weaker consistency and more complex configuration steps. We have seen that the default automatic sharding in MongoDB affects the performance significantly when working with larger datasets.

In “Performance Evaluation of NoSQL Systems Using YCSB in a resource Austere Environment” [38], MongoDB was faster at inserting documents when inserting 1000, 20000, 40000, 60000, 80000 and 100000 documents. Of these different sizes, we have only tested on 100000, in which Elasticsearch was faster. It is however not clear enough what settings were used in previous research, in order to compare them equally. However, the tests in this thesis show that certain settings might very well change the performance of both MongoDB and Elasticsearch. The read tests in the same article is about retrieving specific amounts of documents, which we have not tested and therefore it is not possible to compare the results with the full-text search results. The same goes for “Performance optimization of applications based on non-relational databases” [7], which only performs tests on reads and on much smaller datasets.

In “Choosing the right nosql database for the job: a quality attribute evaluation” [37], the authors states that document-oriented stores are in general considered to be more read efficient and optimized. The authors refer to several performance evaluations, in which MongoDB sometimes is seen as better compared to other NoSQL systems. The authors also classify MongoDB’s read performance as Great and its write performance as Mediocre, which might be reflected in this thesis as well. In some scenarios, MongoDB was good at full-text search, but concerning writes, Elasticsearch outperformed MongoDB.
4.3 NotaQL Extension

NotaQL was successfully extended in order to support transformations into Elasticsearch. NotaQL already had support for document-oriented stores, which made the extension process less complicated. We based the Elasticsearch extension on the NotaQL files for the already supported engines MongoDB and JSON. Both those engines are compliant with the NotaQL document-store syntax, and both uses the JSON format for storage, which Elasticsearch does as well. Strictly speaking, MongoDB uses BSON (Binary JSON). BSON is an extension of JSON and is not an issue. Elasticsearch also had an Apache Spark extension that was officially supported and since it also uses JSON it was considered compatible with NotaQL.

The NotaQL grammar for Elasticsearch is the same as for MongoDB as it is based on the common grammar for document-oriented stores. The transformations shown in Section 2.7.1 are therefore applicable for Elasticsearch in the same way as shown for MongoDB. However, there are still two differences in the syntax. Firstly, how you specify the engine, and secondly, a workaround related to an issue we encountered. The encountered issue with the extension was related to the use of metadata fields in Elasticsearch. Metadata fields are fields that are not part of the original dataset and are instead stored by Elasticsearch in order to help other functionality. Below we present how to specify the engine for Elasticsearch using the NotaQL tool:

```plaintext
elasticsearch (index_name <- '...', type_name <- '...')
```

For the use case in this thesis, we wanted to transfer MongoDB’s document identification field. This field is named _id and the goal was to set it with the same name in Elasticsearch, like this: `OUT._id <- IN._id`. Elasticsearch also has a field named _id, which is a metadata field, and used as an identifier for the document. This field is by default auto-generated when a document is inserted. Instead of storing MongoDB’s _id field as a non-metadata field in Elasticsearch, and hence having two identification fields, we wanted to set Elasticsearch’s _id field. Elasticsearch 5.2 through Apache Spark allows for modifications on metadata fields, but not by default. To modify such fields, additional mapping rules need to be added during the transformation. However, even with a mapping rule, we did not manage to send MongoDB’s _id field with that explicit field name, because Elasticsearch still complained that it was associated with a metadata field. If this is a bug or not, we do not know. This could however be solved by sending MongoDB’s _id field to Elasticsearch by another name, for example mid. First when mid had arrived at Elasticsearch, it was possible to use it to set Elasticsearch’s _id field. Unfortunately this workaround only works for this specific scenario and any other metadata transformations would require further extension. The workaround is shown below:

```java
JavaEsSpark.saveToEs(output, indexName+/"+typeName, ImmutableMap.of("es.mapping.id", "mid", "es.mapping.exclude", "mid");
```

4.4 Quantitative Evaluation of NotaQL and Transporter

The results from Transporter and NotaQL can be seen in Figures 4.10 - 4.12 below.
4.4. Quantitative Evaluation of NotaQL and Transporter

Figure 4.10: Transformations on data size small

Figure 4.11: Transformations on data size medium
4.5. Qualitative Evaluation of NotaQL and Transporter

If we first look at the differences between the data sizes, we can see that Transporter is faster than NotaQL when executing all transformations on the small data size. However, if we look at medium and large, NotaQL is faster for transformation 2, 3, 4, 6, and 7, while Transporter is still faster for transformation 1, 5, and 8. The difference between these two groups of transformations is that number 2, 3, 4, 6, and 7 all include a filter, which means that not all documents are transformed and inserted into Elasticsearch. For transformations 2, 3, 4, and 6, the same filter was used. For data size large, 0.28% of the documents passed the filter. For data size medium, 3.8% passed the filter. For data size small, 0.37% passed the filter. For transformation 7, two filters were used. For data size large, 0.09% of the documents passed the filter. For data size medium, 0.13% passed the filter. For data size small, 0.14% passed the filter. In transformation 1, 5, and 8, all documents are transformed and inserted, and in transformation 8, all fields as well. So that those with a filter would result in a faster transformation time, was not unexpected. However, the fact that NotaQL is faster than Transporter with the filter queries, even though they transform the same amount of documents, is an interesting observation. A likely explanation for this is that Transporter has less overhead on its filter operations, but is not as efficient as NotaQL when handling filter operations on larger datasets.

Besides the mentioned observations, we can see that transforming all documents and all its fields in transformation 8, takes a lot longer time compared to the others, especially for NotaQL.

4.5 Qualitative Evaluation of NotaQL and Transporter

In this section, the qualitative evaluation of NotaQL and Transporter is presented, which evaluates the aspects of performance and transformations.

4.5.1 Purpose and Foundation

The use of NotaQL, along with design decisions, and grammar, is well-documented in several scientific reports. While Transporter is documented on its github page and a few
4.5. Qualitative Evaluation of NotaQL and Transporter

Compose articles, it lacks examples for many use cases, and details on many of the design decisions can only be found within the source code. Both engines are built on abstraction principles and are suitable for extensions.

NotaQL was constructed as a science project with the goal of constructing a language which could support data transformations and transfers across many different storage engines. It was also built on top of Apache Spark which means that extensibility for all their supported storage engines is fairly straight-forward. Another benefit of Apache Spark is the ability to easily scale horizontally.

Compose’s Transporter was built in a production setting and is used to transfer entire or parts of databases to and from specific systems in order to simplify engine migrations. We have not found any indications of horizontal scalability and therefore it has probably not been deemed necessary for their use of it.

There are also significant language differences since NotaQL is a declarative language as opposed to Transporter which uses transformations written in an imperative programming language. This would mean that NotaQL could be simpler to use for a non-programmer, while Transporter could be seen as a flexible tool for anyone used to writing programming scripts.

4.5.2 Performance

Besides the mentioned observations in the results above, we can see that NotaQL and Transporter are suitable for different use cases. Transforming all documents and all its fields, as in transformation 8, takes a lot longer to complete compared to the other transformations and is particularly slow for NotaQL. In such cases, in which the amount of documents is known in advance to be high, Transporter might be the proper choice for such a task. However, when using filters, and the number of documents up for transformation is not the entire document set, NotaQL would seem to be the most efficient choice.

Another important aspect is horizontal scalability and since NotaQL is built on Apache Spark it is fairly easy to scale out. While Transporter does not show any signs of having been built for horizontal scalability. This difference could be quite significant and allow for large performance gains in NotaQL’s favor, however this is not something we test in this thesis.

4.5.3 Transformations

NotaQL and Transporter do not support the same native transformations. Both NotaQL and Transporter support projections, filters, and rename. However, for Transporter, we can not draw any conclusions regarding the use of the logical operators '!', and '||' (For example 'FILTER1' || 'FILTER2') since these have not been tested and we have not been able to find any documentation on their usage. NotaQL however, supports these operators. In addition to these three transformations, NotaQL also supports aggregations, which Transporter does not. Transporter on the other hand, supports pretty print, which NotaQL does not.

One of the most important difference between NotaQL and Transporter is that Transporter can only execute its native transformations on top-level fields, while NotaQL can execute on nested fields. This limits the number of use cases in which Transporter could be used, especially since nested fields are commonly used with JSON objects. The data used in this thesis only contains top level fields.

Using Transporter, it is possible to transform the data in other ways by creating JavaScript
4.5. Qualitative Evaluation of NotaQL and Transporter

pipelines, and also by adding native transformations in the source code. For NotaQL, additional transformations require additions to the source code.
This chapter presents the discussion and contains the following sub-headings.

- Comparing and Combining MongoDB with Elasticsearch
- Method
- Hardware Limitations
- Work in a Wider Context
- Future Research

5.1 Comparing and Combining MongoDB with Elasticsearch

By examining the results, we can see that there are different use cases in which MongoDB is preferable to use instead of Elasticsearch, and vice versa. Concerning Consistency, MongoDB and Elasticsearch are very similar. However, MongoDB has more documented approaches for achieving different levels of consistency and has been subject to more evaluations than Elasticsearch. With this in mind, MongoDB can be considered a safer choice than Elasticsearch, even though Elasticsearch might be able to implement the same functionality using similar approaches.

Concerning scalability, MongoDB and Elasticsearch are also similar. The major difference is that MongoDB allows for additional primary shards to be added to an already existing cluster, while Elasticsearch needs to re-index the entire dataset. Elasticsearch however, seems to have a more effective automatic sharding process than MongoDB. This means that Elasticsearch might be preferable if you are quickly deploying a cluster which immediately should be able to handle large amounts of data. While MongoDB would be more suitable for a scenario where the data increases over a longer period of time.

Based on the quantitative evaluation, Elasticsearch outperformed MongoDB in several tests. Elasticsearch performed better than MongoDB when inserting data, both with and without replication, and also when executing complex full-text search queries on larger datasets. MongoDB however, performs better when executing simple full-text search queries, especially
with an increasing number of shard nodes. This would indicate that MongoDB scales well 
with additional nodes, but would require further tests to confirm.

Based on the performance results of MongoDB and Elasticsearch, there are situations where 
a combination might be considered as an option. However, there are also situations where 
MongoDB and Elasticsearch could be used on their own, separately. A situation, in which 
using only MongoDB could be considered as an option, would be when executing only 
simple full-text search queries on large data sizes. Even though the insertion time is slower 
than Elasticsearch, MongoDB might be preferable to use if the data size has grown large 
over time. Another reason to use MongoDB instead of Elasticsearch is as mentioned, when 
reliability and stability has to be guaranteed.

If a lot of complex full-text search queries need to be executed, Elasticsearch is the pre-
ferred choice. However, MongoDB’s performance seem to scale better when adding more 
nodes. This would seem to indicate that MongoDB is better when a large number of nodes 
are available, even in the case of performing complex full-text search queries. We can how-
ever, neither confirm nor deny this statement since the evaluation in this thesis is limited 
to three nodes. A combination could also be considered as an option, in order to rely on 
MongoDB’s stability and reliability, and Elasticsearch’s speed. Especially if both simple 
and complex full-text search queries are going to be executed. Another important aspect is 
the customizability available when configuring the full-text search. MongoDB’s text-index 
comes pre-configured to work for a lot of full-text search queries, but the customization 
options are very restricted. Elasticsearch’s analyze needs a bit more configuration to activate 
all functionality that MongoDB’s text-index offers. Therefore Elasticsearch could require a bit 
more work before it is ready, however it allows for a near endless number of configurations 
to fine-tune the search for specific needs.

In order to combine MongoDB and Elasticsearch, a tool such as NotaQL or Transporter must 
be used. The results show that NotaQL and Transporter are good for different use cases. 
When working with small data sizes, Transporter performs better in the transformation tests, 
but with larger data sizes, NotaQL performs better in test cases involving filter operations. 
The results show that Transporter is the preferable option when transforming and trans-
ferring all documents within a large dataset. However, if a smaller amount of documents 
need to be filtered from a large dataset, NotaQL seems to be the better option. Another as-
pect to remember is the fact that NotaQL and Transporter were evaluated on single-machine 
setups. This is an important detail for NotaQL as it was built to support horizontal scalability.

The question whether to use NotaQL or Transporter, depends on the situation. However, 
there are two important aspects that could make NotaQL a preferable choice, over Trans-
porter. First of all, NotaQL can execute aggregation transformations, which Transporter can 
not. These types of transformations could be useful when for example analyzing data and 
obtaining statistics. Generally, this limits the number of use cases for Transporter. Secondly, 
NotaQL can execute its native transformations on both top-level fields and nested fields. 
Transporter can only execute its native transformations on top-level fields. This could be 
considered as a major flaw, since it is not possible to execute transformations on all types of 
data structures. This further limits the number of uses cases for Transporter.

Using NotaQL or Transporter to transform data into Elasticsearch in order to enable exe-
cution of fast full-text search queries in Elasticsearch would take some time, which depends 
on the data size. If the data size is large this operation would be best suited as an offline 
preparation step and not used on-the-fly during live deployments. Another approach is 
to continously transfer updates in the source (in our case MongoDB) to Elasticsearch and 
therefore build up the databases from scratch in both database engines. This would be suited
for live deployments, but would mean extra overhead. In such a case, Elasticsearch could also be seen as a replica, that is, if the data was simply transferred and not transformed. Such a combination would allow for both fast simple and complex full-text search queries. It would also allow you to keep the stability and reliability of MongoDB. Although we have not tested transformations from Elasticsearch to MongoDB, we do not see any significant architectural differences in performing the transformations in reverse direction and present this as a possible starting point for continuing the work presented in this thesis. Based on the results that we do have, we would say that it is possible that an approach where Elasticsearch is used as the primary storage engine and MongoDB is used for replicas could result in good overall speed along with reliable replicas.

5.2 iMatrics

As the results and the discussion shows, there are different solutions that fulfill iMatrics’ basic functional requirements from Section 1.2 and the derived performance and software quality aspects from the use case in Section 3.1. Both MongoDB and Elasticsearch are highly configurable and can be used for many different contexts. For the use case presented in this thesis, we conclude that an effective NoSQL solution is possible to achieve. Storing the data in one of these systems, allows for convenient management of full-text data even if it is unstructured or semi-structured, and the data can be inserted as plain JSON. The scalability aspect of the systems seem promising, and with enough available computer nodes it is possible to achieve the desired performance for both insertion and full-text search, as mentioned in Section 3.1. A combination also offers ways to optimize certain parts of the system, such as performing simple full-text search queries using MongoDB and complex full-text search queries using Elasticsearch. Using both NoSQL systems along with a transformation tool demands more computation power, and hence more hardware. We recommend actively choosing a transformation tool each time a transfer should be carried out and there are a few guidelines for this. If you need to perform transformations on nested fields, you have to use NotaQL. If not, then you should choose Transporter if you have a small dataset or if you will not use any filters. Therefore NotaQL should be used when transferring large datasets using filters. Another benefit to NotaQL is also the possibility of easily performing aggregations and therefore advanced transformations including aggregations should most likely be carried out using NotaQL.

5.3 Method

The evaluation has not been performed on completely equal terms. We have used a general data model and used as similar functionality as possible without actually compromising their inherent strengths. Since the tools are developed and used for different purposes, there are many functionalities that are not quite the same and they would lose much of their purpose if forced inside a predefined context. The comparison has been performed with a real context in mind, but more scenarios would be interesting to investigate.

5.3.1 Source Criticism

Finding scientific literature for certain areas, such as for performance and software quality attribute evaluation, was difficult. The reason for that is because for some areas, there are not much to be found, and sometimes nothing at all. Most of the used scientific literature was found for MongoDB, while little was found for Elasticsearch, NotaQL and Transporter. However, efforts have been made to increase the credibility of the thesis, by frequently using the official documentation, together with blog posts written by developers and users. Related and relevant scientific material has been used when possible.
In addition to the type of material used, we have also used an exploratory testing procedure to verify and test how hardware specific fine-tuning impacts the performance. This should strengthen the methodology in terms of both reliability and openness.

5.3.2 Hardware Limitations

The limited hardware available meant that we had to limit the extent of some resource heavy tests due to time constraints. It also meant that we could not test the impact of more than three shard nodes. The insertion tests were limited to a maximum usage of two nodes, and full-text search queries were limited to a maximum of three nodes. Replication tests were limited to only one replica. Since horizontal scaling is an essential part of NoSQL, further research using more nodes could be interesting to perform. The same goes for the transformation tests using NotaQL and Transporter, since the number of shard nodes in their tests were limited to one.

From the results, we could observe that the time it took to parse JSON strings to BSON format, was faster using a strong query router compared to a weak. However, in our opinion, it was not as much faster as we had expected. Especially if we compare number of cores and threads on the processors. A possible explanation could be that all threads are not used. However we did not have the resources to fully investigate the possibilities and limitations related to MongoDB’s JSON parse.

5.4 Work in a Wider Context

A possible connection to ethical aspects would be the issue of handling sensitive and personal information. When handling sensitive information it is important to always store and process these according to common law, best practices and appropriate security considerations. These aspects are important to take into consideration, especially when using new technologies, for example new NoSQL systems.

Societal impacts could include issues regarding longer work schedules for personnel using ill-fitting technology. The correct choice of tools can also help reduce the number of support issues and incorrect use which leads to misunderstandings or abuse. An example of a difficult situation would be the lack of ACID transactions on multiple documents when working with bank account transactions.

The use of properly optimized computational tools could reduce the need of computing power and therefore help reduce the demand of computers and/or electricity consumption.

More efficient systems for information storage and retrieval could also have societal impacts in the form of more accessible and available information, which could help make information easily accessible to the public. This would in turn open up for new business opportunities. An example is Facebook and their massive amounts of data. If they only relied on traditional RDBMS technologies their IT-costs would be much higher and therefore their business might not have been viable to sustain at such a large scale.

5.5 Future Research

An important part of the purpose of this thesis was to evaluate MongoDB and Elasticsearch on larger data sizes than previous research had done (See Section 2.11). This was interesting, since the large data size used in this thesis was a breakpoint for some performance tests (See
Section [4.1]. Therefore, it would be interesting to investigate the impact of increasing the data size even more.

Horizontal scalability has also shown to impact insertion and full-text search queries. Because of the limited use of hardware in this thesis, it would be interesting to investigate the impact of using more hardware in the form of more shard nodes. It would also be interesting to add multiple clusters, measure the impact of network issues and see how MongoDB and Elasticsearch handles commonly problematic issues such as split brain.

In addition to the suggestions above, it would also be interesting to evaluate other DBMS than MongoDB and Elasticsearch, and compare how they stand in relation to the results in this thesis. It does not necessarily have to be NoSQL DBMS, it could for example also be Relational DBMS that supports horizontal scalability.
Do MongoDB and Elasticsearch fulfill the aforementioned system requirements? The basic functional requirements listed in Section 1.2 were:

- Inserting semi-structured text-documents in the form of JSON
- Retrieving inserted documents in the form of JSON
- Horizontal scalability
- Full-text search using:
  - Index upon word stem (stemming)
  - Remove stop words when searching
  - Regular expressions

In addition to these requirements, the use case in Section 3.1 mentions performance requirements, along with the possibility to allow for weaker consistency, and good horizontal scalability. The full-text search queries need to be fast and therefore executed in a matter of seconds or even milliseconds. The insertion however, is allowed to take some time, at most one day. Both MongoDB and Elasticsearch fulfill these requirements, which means that both systems are qualified for use. More specifically, Elasticsearch outperforms MongoDB when it comes to both insertion speed and complex full-text search queries. MongoDB however, performs better for simple full-text search queries. Concerning horizontal scalability, MongoDB and Elasticsearch are similar in the big picture, but when examined closely we note that they have different strengths and weaknesses which makes them suitable for specific use cases, as discussed in Section 5.1. MongoDB for example, has the possibility to add additional primary shards to an already existing cluster, which is not possible with Elasticsearch. Elasticsearch however, has more effective automatic sharding than MongoDB. As mentioned in Section 1.2, MongoDB has been recommended as a stable primary storage engine and we do confirm the reliable nature of MongoDB. However when it comes to speed, it generally can not compete with Elasticsearch.
Can Compose’s Transporter and NotaQL handle transformations on semi-structured text data from MongoDB and insert into Elasticsearch?

Both Transporter and NotaQL work well for performing projections, filters and rename transformations from MongoDB to Elasticsearch. However, there are significant time differences and the right tool can only be chosen based on the specific scenario. Neither of the tools works best for all use cases. Since NotaQL can transform nested fields, while Transporter can not, the possible transformations that can be executed with Transporter are only a subset of the possible transformations that can be executed using NotaQL. When working with unstructured data, or semi-structured data, it could be difficult to predict if transformations on nested fields will be required. Therefore, NotaQL has more use cases than Transporter. However, since the data structure in this thesis shows that Transporter works for this use case, it can also be considered a suitable tool. The main issue is that if the data structure would change along the way to include nested structures in a project originally using Transporter, then this would mean that the tool is no longer a viable choice.

What are the advantages and disadvantages of using the combined solution as compared to separately using MongoDB or Elasticsearch?

Based on the results, and the discussion in Section 5.1, we deem it possible and in some cases even useful to combine MongoDB and Elasticsearch. One combined approach is to use MongoDB as a primary storage, and Elasticsearch for replicas, in which certain queries are executed on MongoDB, and others on Elasticsearch. The advantages of such a solution is the ability to reliably store documents and perform simple full-text search queries in MongoDB along with the extensive functionalities, customizability and speed for performing complex full-text search queries in Elasticsearch. The disadvantages include the need to configure multiple storage engines, the necessity of having a tool for transformations between the engines, the added resource consumption of having to send all documents through both the engines, and a transformation pipeline in the middle. This means that the overall complexity of the system would increase, and could be considered too complex for many use cases. Another important aspect of this, is that you would be stuck with MongoDB’s slow insertion speed. Another approach, which we have not tested in this thesis, would be to use Elasticsearch as the primary storage and MongoDB on replicas. This could result in good speeds along with stable replicas which could also be used to answer simple full-text search queries.
Bibliography


A.1 Simple Full-Text Search Queries

ford
car
honda
toyota
toyota
cars
dodge
nissan
jeep

tires

googol
chevrolet
mercedes
mustang
ebay
autozone

subaru

audi
hyundai

quotes

books

fanfiction

sparknotes

quote

poems

frases

poetry

facebook

shakespeare

mac
A.1. Simple Full-Text Search Queries

apple
xbox
chrome
drive
microsoft
java
paypal
adobe
ipad
dell
sony
canon
rent
apartments
homes
houses
real
florida
home
apartment
orlando
bank
credit
chase
calculator
money
fidelity
finance
new
business
staples
access
netflix
coupons
nike
black
pandora
lego
gamestop
furniture
tile
box
target
pool
sears
clock
refrigerator
dog
lottery
dogs
cat
puppies
fish
cats
horse
camera
flowers
kayak
bowling
flower
heart
diabetes
marijuana
walmart
anxiety
measles
youtube
depression
amazon
herpes

A.2 Complex Full-Text Search Queries Elasticsearch

mail shutterfly maps frozen
iphone college immigration panera
verizon jobs express subway
android elementary movies baby
tumblr test bottle girls
twitter education games hot
msn indeed imdb pregnancy
samsung resume disney cup
gmail blackboard redbox love
sprint academy batman horoscope
internet craigslist bachelor karma
apps work empire jesus
linkedin powerschool pizza ass
wallpaper harvard recipes india
torrent army food friends
gif unemployment restaurants news
images obama cake cnn
women navy pizza hut accuweather
face act dominos espn
chat passport starbucks bbc
background visa tea drudge
wallpapers safety sushi translate
backgrounds constitution mcdonalds definition
instagram justice chipotle map
ford AND apple AND cat AND books AND quote AND sparknotes
car AND xbox AND puppies AND fanfiction AND cats
honda AND chrome AND fish
toyota AND drive
chevy AND microsoft AND horse AND poems
sony AND orlando AND google
canon AND bank AND chevrolet AND pandora AND black
rent AND credit AND mercedes AND lego
apartments AND chase AND mustang AND gamestop
homes AND calculator AND ebay AND furniture AND measles AND youtube
clock AND staples AND camera
refrigerator
dog AND netflix AND kayak AND depression
lottery AND coupons AND bowling AND amazon
dogs AND nike AND flower AND herpes AND finance AND nissan
autozone AND heart AND java AND houses
subaru AND diabetes AND paypal AND real AND flowers
audi AND marijuana AND adobe AND florida
hyundai AND walmart AND ipad AND home AND access
quotes AND anxiety AND dell AND apartment
tile AND frases AND money AND cars
box AND poetry AND fidelity
target AND facebook
pool AND shakespeare AND new AND jeep
sears AND mac AND business AND tires AND dodge
library powerball -horses -violin
dictionary mario -animals -cello
calendar minecraft -husky -accordion
history skyrim -lotto -economy
translation wow -boat -beep
translator steam -record -exchange
bing nfl -surf -rate
synonyms football -yacht -currency
hotel basketball -charter -control
flights patriots -dive -import
hotels nba -marina -export
vegas seahawks -sailboat -market
southwest sports -sail -financial
beach cowboys -harbor -tree
flight packers -ymca -bush
united superbowl -wireless -grass
delta colts -clip -stairs
travel ufc -producer -lake
train animal -paper -river
expedia puppy -guitar -spring
marriott petsmart -bass -winter
hilton bird -drums -summer
cruises pitbull -piano -fall
game bulldog -tuba -watch
pokemon shark -saxophone -washer
"I am never at home"
"the broken glass"
"says goodbye"
"than I initially thought it would be"
"was getting dark"
"I can find everything"
"different from each other"
"I don’t know"
"my hand"
"Let me help"
"I will never be"
"not the right thing"
"I’d rather be"
"I want more detailed information"
"Check back tomorrow"
"that was her reason for"
"in the middle of the night"
"I hear that"
"I went to the movie"
"they are clean and"
"is very pretty"
"it was a lovely sight"
"many years ago"
"speaks to him"
"it is better"

A.3 Complex Full-Text Search Queries MongoDB

mail shutterfly maps frozen
iphone college immigration panera
verizon jobs express subway
android elementary movies baby
tumblr test bottle girls
e-mail classroom music bible
twitter education games hot
msn indeed imdb pregnancy
samsung resume disney cup
gmail blackboard redbox love
sprint academy batman horoscope
internet craigslist bachelor karma
apps work empire jesus
linkedin powerschool pizza ass
wallpaper harvard recipes india
torrent army food friends
gif unemployment restaurants news
images obama cake cnn
women navy pizza hut accuweather
face act dominos espn
chat passport starbucks bbc
background visa tea drudge
wallpapers safety sushi translate
backgrounds constitution mcdonalds definition
instagram justice chipotle map
\"ford\" \"apple\" \"cat\" \"books\" \"quote\" \"sparknotes\"
\"car\" \"xbox\" \"puppies\" \"fanfiction\" \"cats\"
\"honda\" \"chrome\" \"fish\"
\"toyota\" \"drive\"
\"chevy\" \"microsoft\" \"horse\" \"poems\"
\"sony\" \"orlando\" \"google\"
\"canon\" \"bank\" \"chevrolet\" \"pandora\" \"black\"
\"rent\" \"credit\" \"mercedes\" \"lego\"
\"apartments\" \"chase\" \"mustang\" \"gamestop\"
\"homes\" \"calculator\" \"ebay\" \"furniture\" \"measles\" \"youtube\"
\"clock\" \"staples\" \"camera\"
"refrigerator"
"dog" "netflix" "kayak" "depression"
"lottery" "coupons" "bowling" "amazon"
"dogs" "nike" "flower" "herpes" "finance" "nissan"
"autozone" "heart" "java" "houses"
"subaru" "diabetes" "paypal" "real" "flowers"
"audi" "marijuana" "adobe" "florida"
"hyundai" "walmart" "ipad" "home" "access"
"quotes" "anxiety" "dell" "apartment"
"tile" "frases" "money" "cars"
"box" "poetry" "fidelity"
"target" "facebook"
"pool" "shakespeare" "new" "jeep"
"sears" "mac" "business" "tires" "dodge"
library powerball -horses -violin
dictionary mario -animals -cello
calendar minecraft -husky -accordion
history skyrim -lotto -economy
translation wow -boat -beep
translator steam -record -exchange
bing nfl -surf -rate
synonyms football -yacht -currency
hotel basketball -charter -control
flights patriots -dive -import
hotels nba -marina -export
vegas seahawks -sailboat -market
southwest sports -sail -financial
beach cowboys -harbor -tree
flight packers -ymca -bush
united superbowl -wireless -grass
delta colts -clip -stairs
travel ufc -producer -lake
train animal -paper -river
expedia puppy -guitar -spring
marriott petsmart -bass -winter
hilton bird -drums -summer
travels pitbull -piano -fall
game bulldog -tuba -watch
pokemon shark -saxophone -washer
"I am never at home"
"the broken glass"
"says goodbye"
"than I initially thought it would be"
"was getting dark"
"I can find everything"
"different from each other"
"I don’t know"
"my hand"
"Let me help"
"I will never be"
"not the right thing"
"I’d rather be"
"I want more detailed information"
"Check back tomorrow"
"that was her reason for"
"in the middle of the night"
"I hear that"
"I went to the movie"
"they are clean and"
"is very pretty"
"it was a lovely sight"
"many years ago"
"speaks to him"
"it is better"
B Transformations

B.1 NotaQL Expressions

The transformation expressions for NotaQL, formated in JSON according to the NotaQL tool.

```
[  
  
  "name": "TR1 Projection",  
  "IN–ENGINE": {  
    "engine": "mongodb",  
    "database_name": "database",  
    "collection_name": "collection"  
  },  
  
  "OUT–ENGINE": {  
    "engine": "elasticsearch",  
    "index_name": "database",  
    "type_name": "collection"  
  },  
  "transformation": "OUT.mid < IN._id, OUT.body < IN.body"
},

[  
  "name": "TR2 Filter",  
  "IN–ENGINE": {  
    "engine": "mongodb",  
    "database_name": "database",  
    "collection_name": "collection"  
  },  
  
  "OUT–ENGINE": {  
    "engine": "elasticsearch",  
    "index_name": "database",  
    "type_name": "collection"  
  },  
  "transformation": "IN–FILTER: IN.subreddit = 'Games',  
  OUT.mid < IN._id, OUT.$(IN.*.name()) < IN.@"
]`
```
B.1. NotaQL Expressions

```
},

"name": "TR3 Filter+Projection",
"IN-ENGINE": {
  "engine": "mongodb",
  "database_name": "database",
  "collection_name": "collection"
},

"OUT-ENGINE": {
  "engine": "elasticsearch",
  "index_name": "database",
  "type_name": "collection"
},

"transformation": "IN-FILTER: IN.subreddit = 'Games',
OUT.mid <- IN._id, OUT.body <- IN.body"
},

},

"name": "TR4 Filter+Rename",
"IN-ENGINE": {
  "engine": "mongodb",
  "database_name": "database",
  "collection_name": "collection"
},

"OUT-ENGINE": {
  "engine": "elasticsearch",
  "index_name": "database",
  "type_name": "collection"
},

"transformation": "IN-FILTER: IN.subreddit = 'Games',
OUT.mid <- IN._id, OUT.full_text <- IN.body"
},

},

"name": "TR5 Projection+Rename",
"IN-ENGINE": {
  "engine": "mongodb",
  "database_name": "database",
  "collection_name": "collection"
},

"OUT-ENGINE": {
  "engine": "elasticsearch",
  "index_name": "database",
  "type_name": "collection"
},

"transformation": "OUT.mid <- IN._id, OUT.full_text <- IN.body"
},

},

"name": "TR6 Filter+Projection+Rename",
"IN-ENGINE": {
  "engine": "mongodb",
  "database_name": "database",
  "collection_name": "collection"
},

"OUT-ENGINE": {

```
B.1. NotaQL Expressions

```json
{
    "engine": "elasticsearch",
    "index_name": "database",
    "type_name": "collection"
}
{
    "transformation": "IN-FILTER: IN.subreddit = 'Games',
    OUT.mid <- IN._id, OUT.full_text <- IN.body"
}
{
    "name": "TR7 Multiple Filters",
    "IN-ENGINE": {
        "engine": "mongodb",
        "database_name": "database",
        "collection_name": "collection"
    },
    "OUT-ENGINE": {
        "engine": "elasticsearch",
        "index_name": "database",
        "type_name": "collection"
    },
    "transformation": "IN-FILTER: IN.subreddit = 'Games' && IN.ups >= 4,
    OUT.mid <- IN._id, OUT.$(IN.*.name()) <- IN.""
}
{
    "name": "TR8 Rename",
    "IN-ENGINE": {
        "engine": "mongodb",
        "database_name": "database",
        "collection_name": "collection"
    },
    "OUT-ENGINE": {
        "engine": "elasticsearch",
        "index_name": "database",
        "type_name": "collection"
    },
    "transformation": "OUT.mid <- IN._id, OUT.field_1 <- IN.author_flair_text,
    OUT.field_2 <- IN.gilded, OUT.field_3 <- IN.author_flair_css_class,
    OUT.field_4 <- IN.retrieved_on, OUT.field_5 <- IN.ups,
    OUT.field_6 <- IN.edited, OUT.field_7 <- IN.subreddit_id,
    OUT.field_8 <- IN.controversiality, OUT.field_9 <- IN.parent_id,
    OUT.field_10 <- IN.subreddit, OUT.field_11 <- IN.body,
    OUT.field_12 <- IN.created_utc, OUT.field_13 <- IN.downs,
    OUT.field_14 <- IN.score, OUT.field_15 <- IN.author,
    OUT.field_16 <- IN.archived, OUT.field_17 <- IN.distinguished,
    OUT.field_18 <- IN.id, OUT.field_19 <- IN.score_hidden,
    OUT.field_20 <- IN.name, OUT.field_21 <- IN.link_id"
}
]```
B.2 Transporter Scripts

The transformation JavaScript for Transporter.

TR1 Projection

```javascript
var source = mongodb({
    "uri": "mongodb://localhost/database"
})

var sink = elasticsearch({
    "uri": "http://localhost:9200/database"
})

t.Source("source", source, "namespace", "collection").
Transform(pick({"fields":[_id, "body"]})).
Save("sink", sink, "namespace", "collection")
```

TR2 Filter

```javascript
var source = mongodb({
    "uri": "mongodb://localhost/database"
})

var sink = elasticsearch({
    "uri": "http://localhost:9200/database"
})

t.Source("source", source, "namespace", "collection").
Transform(skip({"field":"subreddit", "operator":"==", "match":"Games"})).
Save("sink", sink, "namespace", "collection")
```

TR3 Filter+Projection

```javascript
var source = mongodb({
    "uri": "mongodb://localhost/database"
})

var sink = elasticsearch({
    "uri": "http://localhost:9200/database"
})

t.Source("source", source, "namespace", "collection").
Transform(skip({"field":"subreddit", "operator":"==", "match":"Games"})).
Transform(pick({"fields":[_id, "body"]})).
Save("sink", sink, "namespace", "collection")
```

TR4 Filter+Rename

```javascript
var source = mongodb({
    "uri": "mongodb://localhost/database"
})

var sink = elasticsearch({
    "uri": "http://localhost:9200/database"
})

```
B.2. Transporter Scripts

```
var sink = elasticsearch({
  "uri": "http://localhost:9200/database"
})

t.Source("source", source, "namespace", "collection").
Transform(skip({"field":"subreddit","operator":"=","match":"Games"})).
Transform(rename({"field_map":{"body":"full_text"}})).
Save("sink", sink, "namespace", "collection"
)

TR5 Projection+Rename

var source = mongodb({
  "uri": "mongodb://localhost/database"
})

var sink = elasticsearch({
  "uri": "http://localhost:9200/database"
})

t.Source("source", source, "namespace", "collection").
Transform(pick({"fields":["_id","body"]})).
Transform(rename({"field_map":{"body":"full_text"}})).
Save("sink", sink, "namespace", "collection"
)

TR6 Filter+Projection+Rename

var source = mongodb({
  "uri": "mongodb://localhost/database"
})

var sink = elasticsearch({
  "uri": "http://localhost:9200/database"
})

t.Source("source", source, "namespace", "collection").
Transform(skip({"field":"subreddit","operator":"=","match":"Games"})).
Transform(pick({"fields":["_id","body"]})).
Transform(rename({"field_map":{"body":"full_text"}})).
Save("sink", sink, "namespace", "collection"
)

TR7 Multiple Filters

var source = mongodb({
  "uri": "mongodb://localhost/database"
})

var sink = elasticsearch({
  "uri": "http://localhost:9200/database"
})
```
B.2. Transporter Scripts

```javascript
var source = mongodb({
    "uri": "mongodb://localhost/database"
});
var sink = elasticsearch({
    "uri": "http://localhost:9200/database"
});

t.Source("source", source, "namespace", "collection").
Transform(rename({"field_map": {"author_flair_text": "field_1", "gilded": "field_2", "author_flair_css_class": "field_3", "retrieved_on": "field_4", "ups": "field_5", "edited": "field_6", "subreddit_id": "field_7", "controversiality": "field_8", "parent_id": "field_9", "subreddit": "field_10", "body": "field_11", "created_utc": "field_12", "downs": "field_13", "score": "field_14", "author": "field_15", "archived": "field_16", "distinguished": "field_17", "id": "field_18", "score_hidden": "field_19", "name": "field_20", "link_id": "field_21"})).Save("sink", sink, "namespace", "collection")
```
C Query Routers and Number of Shards Comparison

C.1 Comparing Query Routers

(a) Elasticsearch

(b) MongoDB

Figure C.1: Insertion Small Dataset

(a) Elasticsearch

(b) MongoDB

Figure C.2: Insertion Medium Dataset
C.2 Comparing Number of Shards

Figure C.3: Insertion Large Dataset

(a) Elasticsearch

(b) MongoDB

Figure C.4: Insertion Small Dataset

(a) Elasticsearch

(b) MongoDB

Figure C.5: Insertion Medium Dataset

(a) Elasticsearch

(b) MongoDB
C.2. Comparing Number of Shards

(a) Elasticsearch

(b) MongoDB

Figure C.6: Insertion Large Dataset