Master’s Thesis

Design and Implementation of a NoSQL Database for Decision Support in R&D Management

Author:
Prem Sagar Jeevangekar

April 23, 2019

Advisors:
Prof. Dr. rer. nat. habil. Gunter Saake
Dipl.-Inf. Wolfram Fenske
Department of Computer Science

Dr. rer. nat. Matthias Plaue
MAPEGY GmbH
Jeevangekar, Prem Sagar:

Design and Implementation of a NoSQL Database for Decision Support in R&D Management

Abstract

The concept of database is introduced in the early 1960s. The relational database came into the picture in the early 1970s. The relational database had a great impact on data storage ever since. However, due to an increase in data in the modern database world, it leads to developing more efficient database technologies to deal with the usage of exponentially increased data. When working with the structured datasets, relational databases are more reliable and efficient. However, the database lacks its efficiency when huge unstructured data is produced from real-world applications. To overcome the problems faced by the relational databases, companies started looking for more reliable, flexible, highly scalable and high-performance databases. In the early 2000s, the NoSQL databases were introduced and gained huge popularity in a short period of time.

The main aim of the thesis is to design and implement the NoSQL database (MongoDB) and investigate its performance in terms of query execution speed. This thesis provides an overview of one of the relational database (PostgreSQL) and investigate the performance of NoSQL database (MongoDB).

The current database is based on a PostgreSQL database that has performance issues. In PostgreSQL database whenever the large complex queries are performed, the performance of the database is low due to a large number of join operations. To investigate the performance of NoSQL database, Document-Oriented MongoDB database is used. It is one of the most popularly used NoSQL databases.

To compare the performance of the MongoDB database. The data from the PostgreSQL is migrated to the MongoDB database. The detailed implementation of data migration procedure is explained in this thesis. The data is extracted from the PostgreSQL database and imported into the MongoDB. By evaluating the performance of both the database. It is possible to take the decision on which database is the best fit to provide high performance for the given data.

The evaluation between the two databases can help in decision making for R&D management.
Acknowledgement

This thesis would not have been accomplished without the support and guidance of

Prof. Dr. rer. nat. habil. Gunter Saake
Dipl.-Inf. Wolfram Fenske

From
Otto von Guericke University Magdeburg

Dr. rer. nat. Matthias Plaue

From
MAPEGY GmbH, Berlin

I would like to thank Computer Science department for providing me an opportunity to learn professional skills by encouraging me to participate in projects and guiding me in my learning stages.

I would like to thank my parents and friends who provided support and encouraged me throughout years of my education. This accomplishment would not have achieved without their prayers and blessings.

Finally, I would like to thank God for giving me strength to reach my goal.

Thank you
Contents

List of Figures viii
List of Tables ix
List of Code Listings xi

1 Introduction 1
   1.1 Goal of the Thesis ............................................ 2
   1.2 Readers Guide .................................................. 3

2 Technical Background 5
   2.1 Decision Support in R&D Management ........................... 5
   2.2 Relational Database Management System ....................... 6
       2.2.1 RDBMS Architecture ....................................... 8
       2.2.2 Database Queries .......................................... 9
       2.2.3 PostgreSQL ................................................ 9
   2.3 NoSQL Databases ............................................... 12
       2.3.1 NoSQL Database Characteristics .......................... 12
       2.3.2 Classification of NoSQL Databases ....................... 13
       2.3.3 MongoDB .................................................. 17
           2.3.3.1 MongoDB Architecture .............................. 18
           2.3.3.2 Schema Design ....................................... 18
           2.3.3.3 Query Model ......................................... 19

3 Requirements and Concept 23
   3.1 NoSQL Database Selection ..................................... 26
   3.2 A Framework for Decision Support in R&D Management ....... 29
   3.3 Data Migration .................................................. 29
       3.3.1 Embeding in MongoDB .................................... 30
       3.3.2 Data Migration Phases .................................... 31
   3.4 MongoDB Query Model .......................................... 33
       3.4.1 Query Language ........................................... 33
           3.4.1.1 Aggregation Pipeline ............................... 35
           3.4.1.2 Text Indexing ....................................... 38

4 Design and Implementation 40
   4.1 Design Process ................................................ 40
   4.2 MongoDB Core Process ......................................... 41
   4.3 Tools Selection ............................................... 42
       4.3.1 NoSQLBooster ............................................ 42
       4.3.2 R framework SHINY ....................................... 43
   4.4 MongoDB Query Optimization .................................. 49
## List of Figures

2.1 Decision support for R&D management ................................. 6
2.2 Codd’s relational model [2589] ........................................ 7
2.3 RDBMS schema model .................................................. 7
2.4 Three-level architecture of relational databases [RU15] ............. 8
2.5 PostgreSQL Schema ..................................................... 10
2.6 Key value stores data mode ............................................. 14
2.7 Wide column stores architecture ...................................... 14
2.8 Graph data model ....................................................... 15
2.9 SQL & NoSQL database classification ................................. 16
2.10 MongoDB document store .............................................. 17
2.11 MongoDB nexus architecture ........................................ 18
2.12 MongoDB denormalized approach example .......................... 19
2.13 MongoDB normalized approach example ............................. 19
3.1 PostgreSQL database model ............................................ 24
3.2 Data model of PostgreSQL database containing required data .... 25
3.3 Functional, Non-Functional requirements, and techniques of MongoDB 28
3.4 Framework to support R&D management ............................... 29
3.5 Embedded data collection structure .................................. 30
3.6 Embedded document structure ....................................... 31
3.7 Data migration process ................................................ 32
3.8 Implementation of data migration process ............................ 33
3.9 Aggregation pipeline .................................................... 36
4.1 Design process .......................................................... 41
4.2 NoSQLBooster Graphical User Interface .............................. 43
4.3 R Framework Shiny Application ...................................... 44
4.4 Output table in Shiny R web application ............................. 48
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Query execution process in SHINY</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>Creating a single text index using NoSQLBooster</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance comparison on 10,000 records</td>
<td>62</td>
</tr>
<tr>
<td>5.2</td>
<td>Performance comparison on 50,000 records</td>
<td>62</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance comparison on 100,000 records</td>
<td>63</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance comparison on 1,000,000 records</td>
<td>63</td>
</tr>
</tbody>
</table>
# List of Tables

2.1 CAP theorem [MH13] ................................. 13
2.2 Terminology difference in MongoDB and RDBMS .......................... 17
3.1 Top ten most popular databases [1] ................................. 27
4.1 MongoDB version and NoSQLBooster Specifications .......................... 42
5.1 Used Machine Specifications ................................. 53
5.2 Statistics of Datasets ................................. 54
5.3 Data Migration results ................................. 56
5.4 Queries for performance comparison from PostgreSQL and MongoDB 61
5.5 PostgreSQL Query results in seconds ................................. 64
5.6 MongoDB Query results in seconds ................................. 65
5.7 Output of the results returned for each query ................................. 65
List of Code Listings

3.1 Finding total number of scientific publications and patents for the institutions ........................................ 26
3.2 Finding data duplication using aggregation ................................................................. 32
3.3 MongoDB embedded document sample ................................................................. 34
3.4 MongoDB query structure and example query ......................................................... 35
3.5 MongoDB aggregation example query ................................................................. 36
3.6 MongoDB aggregation example query output ......................................................... 36
3.7 MongoDB aggregation example query with multiple stages ................................. 37
3.8 MongoDB aggregation example output ................................................................. 37
3.9 MongoDB Text Indexing ................................................................. 38
3.10 MongoDB Text Indexing ........................................................................ 39
4.1 Code for developing a Shiny R application user interface ................................. 45
4.2 Code for developing a Shiny R application server side ......................................... 45
4.3 MongoDB Aggregation Stages for performance optimizations .......................... 50
5.1 MongoDB Data Migration ........................................................................ 55
5.2 Importing data into PostgreSQL database ......................................................... 55
5.3 Importing data into PostgreSQL database ......................................................... 56
5.4 MongoDB query execution ........................................................................ 56
5.5 Aggregation query execution ........................................................................ 57
5.6 MongoDB query ......................................................................................... 58
5.7 MongoDB Ranking number of documents ......................................................... 59
A.1 Query covering some field of interest and get a list of relevant documents 71
A.2 Query covering some field of interest and get a list of organizations rank by number of patents, scientific publications matching the query 72
A.3 Query for an organization and get a list of collaborators, i.e., organizations with common documents; rank them by number of common patents, number of common scientific publications at user interface And at server side shown Listing A.4 on page 73 73
A.4 Query for an organization and get a list of collaborators, i.e., organi-
zations with common documents; rank them by number of common
patents, number of common scientific publications at server side . . . 73
A.5 PostgreSQL query example 1 ..................................... 74
A.6 PostgreSQL query example 2 ..................................... 75
A.7 PostgreSQL query example 3 ..................................... 75
A.8 PostgreSQL query example 4 ..................................... 75
A.9 PostgreSQL query example 5 ..................................... 75
A.10 PostgreSQL query example 6 ................................. 76
A.11 PostgreSQL query example 7 ..................................... 76
A.12 PostgreSQL query example 8 ..................................... 76
A.13 MongoDB query example 1 ..................................... 77
A.14 MongoDB query example 2 ..................................... 77
A.15 MongoDB query example 3 ..................................... 77
A.16 MongoDB query example 4 ..................................... 77
A.17 MongoDB query example 5 ..................................... 77
A.18 MongoDB query example 6 ..................................... 78
A.19 MongoDB query example 7 ..................................... 78
A.20 MongoDB query example 8 ..................................... 79
1. Introduction

In recent days, internet users are gradually increasing. This leads to the exponential growth of the data. The usage of the internet reaches 2.5 quintillions ($10^{18}$) bytes of data on every day \(^1\). From the past 4 decades, relational databases had exclusive control over data storage but some of the applications have performance and scalability issues. So, companies are looking for more reliable database systems. In such a context, NoSQL databases are developed.

The thesis work is done at MAPEGY GmbH company which provides data-driven decision support in the field of life sciences, energy, information & communication systems, industries, and finance & insurance and also provides data products for their customers. Research and Development management plays a key role in supervising and managing the research department of MAPEGY’s customers. The primary objective of the Research and Development is the development of new technology by applying creative ideas to improve knowledge based on patents, scientific publications, and market updates \(^2\). It always focuses on vision and strategy in various perspectives such as financial perspective, customer perspective, internal business perspective, and innovation & learning perspective. This paper mainly focuses on innovation & learning perspective that includes project evaluation ideas for new projects [KvDB99]. R&D management helps to gain knowledge and is used for practical implementation in the future.

Mapegy Gmbh company uses database based on PostgreSQL, which is a relational database and it has some limitations. In this thesis, the data is taken from the company’s PostgreSQL server data warehouse which contains huge datasets of patents, scientific publications, organizations, and so on. The detailed description of the data is given in chapter Chapter 3 on page 23.

In spite of the fact that the relational databases are most common and consistently good in storing and retrieving the data from the database. It has limitations in

\(^1\)URL : https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/

\(^2\)URL: https://www.civilserviceindia.com/subject/Management/notes/r-and-d-management.html
dealing with the data in the relational database. To deal with the data, the relational database needs a predefined schema for data normalization process [KYLTC12]. One of the important limitations is, building a relationship between the entities is complex. For very large queries between interconnected tables, we require JOIN operations to fetch the relevant information. Such cases take long response time which makes querying costly and affects the performance.

This thesis investigates the performance in terms of query execution time between PostgreSQL and one of the NoSQL databases. Although the NoSQL databases are introduced in the early 2000’s it has shown its ability in working with large unstructured and unrelated data. The main reason for its popularity is that it does not require a strict schema structure and provides high performance for large datasets. Unlike relational databases, the NoSQL databases rely on denormalization. It means the data can be retrieved faster that doesn’t involve JOIN operations. In terms of performance, designing and implementing one of the NoSQL databases is selected to decide whether the database is efficient compared to PostgreSQL database. The resultant evaluation between the two databases helps in decision support for R&D management.

NoSQL database is not a single technology. There are dozens of NoSQL databases offer various options they are mainly categorized as Key-value based (example: Redis), wide-column store (example: Hbase), document-oriented databases (example: MongoDB), and graph databases (example: Neo4j) [HJ11b]. The decision in choosing the right database is a key factor for the company. Therefore, R&D management plays an important role in making such decisions. One of the most popular document-oriented databases that is MongoDB database is selected. Because MongoDB provides the abilities of relational database along with the features of high flexibility, high scalability, and high performance.

The thesis contains information about the decision making on selecting the database and the process for selecting the database system for IT managers or engineers for their requirement. The result of the thesis will help to understand the advantages and disadvantage of outcomes of PostgreSQL and MongoDB databases. The data migration and query capabilities of MongoDB are evaluated in terms of performance by comparing it to the PostgreSQL database. In Chapter 3 on page 23 & Chapter 4 on page 40, detailed explanation of selecting the database and its implementation is discussed.

1.1 Goal of the Thesis

The main aim of the thesis is to investigate the performance of NoSQL database by selecting, designing the database and implementing queries and compare its performance to the currently using database (PostgreSQL) in the company. Based on the result of the thesis, why and how the particular database has advantages over the other database is discussed. In the thesis, the document-oriented MongoDB database is used for practical implementation. And compare it with the PostgreSQL database and investigate whether MongoDB can overcome the limitation of PostgreSQL.

We aim to reach the following goals:
1.2. Readers Guide

1. Migrating PostgreSQL database to the MongoDB database.
   (a) Designing a data model which includes at least following entities: scientific publications, patents, essential metadata which must contain the title of the documents and the organizations (companies, research institutions), list of organizations which are extracted from the data warehouse and load it into MongoDB server.

2. To examine the querying capabilities of MongoDB like:
   (a) Enter a query and retrieve the information related to a particular field of interest.
   (b) Enter a query covering some field of interest and get all patents and scientific publications.
   (c) Enter a query covering some field of interest and get a list of organizations and experts projected by document type ‘PATENT’ matching the query.
   (d) Enter a query for an organization and get a list of organization and experts and ranking them by the number of patents and number of scientific publications.

We build an interactive prototype web application using the R framework “shiny” for querying data from MongoDB database\(^3\). From the examination of the output query response time of both the databases, the performance of the database is evaluated.

1.2 Readers Guide

The thesis is structured as:

1. **Chapter: Introduction**
   The first chapter gives the brief introduction of decision support for R&D management along with the limitations of currently existing technology (PostgreSQL) company and introduction to the NoSQL databases. The second part illustrates the goal of the thesis followed by the structure of the thesis.

2. **Chapter: Technical Background**
   This chapter Provides basic technical knowledge of decision support for R&D management, RDBMS. Furthermore, the chapter covers the basic knowledge of schema design and query model of PostgreSQL and MongoDB databases.

3. **Chapter: Requirements and Concept**
   The concept of the thesis is discussed in this chapter. The basic requirements, limitations of the PostgreSQL database, NoSQL database selection, data migration process, and MongoDB query model are the key points that are discussed in this chapter.

\(^3\)http://www.rstudio.com/shiny/
4. **Chapter: Design and Implementation**
   
   This chapter covers design and implementation task. The data migration, tools selected for the implementation, MongoDB query optimization is discussed.

5. **Chapter: Evaluation**
   
   In this chapter, the evaluation setup is discussed. The details of the machine used, the datasets used, the experiments implemented is discussed. Then PostgreSQL and MongoDB query performance are compared and results are evaluated.

6. **Chapter: Related Work**
   
   In this chapter, we discuss the related research papers, their approaches and how the papers are related to the thesis is explained.

7. **Chapter: Conclusion and Future Work**
   
   The chapters provide a summary of the thesis. It’s conclusion and future work is discussed.
2. Technical Background

Detail explanation of NoSQL databases will be described in this chapter. This chapter is divided into multiple sections. It provides a basic understanding of decision support and R&D management, fundamentals of the relational, and non-relational databases (NoSQL).

2.1 Decision Support in R&D Management

To support the R&D decision-making process, all the information is collected from patent documents, scientific publications and, market resources. The decision-making process for selecting the database requires detailed technical knowledge and its features. The decision in selecting the database is their efficiency by comparing with other databases. This section gives a brief introduction to decision support in R&D management.

Innovation process generates technological ideas, these ideas implemented and transformed into a new business model that makes profits for the company and creates a lead in the market place. A tremendous amount of research is carried out to develop new products, technologies, and processes. R&D management plays a vital role in developing the growth of the companies [VGPC+17]. From few years, efforts to build better decisions in R&D management for the companies are implemented. Most of the efforts concentrate on making decision models and improving decision-making methods. To improve the decision-making process, determined attempts are made by the researchers to use computer-based decision-making methods to support R&D activity [TML+02]. Figure 2.1 on the following page depicts the decision-making process for R&D activity.

The Figure 2.1 on the next page shows the decision support for R&D management. The strategy in developing a new project should be designed for achieving the goal. For decision support in R&D management. Some of the factors should be considered as follows.

- The scope of market opportunities is taken into account while designing the project.
2. Technical Background

- The new project is planned considering resources like cost, time, and project development environment.
- The sufficient knowledge to face technical difficulties in the project is needed.
- The lack of technical knowledge can not help in developing a new project. So, missing knowledge should be identified and gain knowledge.

The development of new projects requires a good understanding of the current market needs. The knowledge of design and implementation is necessary to learn for investigating the project efficiently according to market needs.

The decisions are taken by comparing the new technology to the existing technology. Here in the thesis, the relational and non-relational databases are compared in terms of database query performance (query execution speed). This helps in making a decision in selecting the database from the comparison.

2.2 Relational Database Management System

The relational database model was first introduced by IBM employee named Edgar F. Codd [Cod82]. In RDBMS, data is stored in tables. One of the key data integrity features of RDBMS includes the primary key and foreign key constraints. Figure 2.2
shows the combination of tuples (rows) and attributes (columns) form a table (relations) to build a basic block of the relational model. For instance, Figure 2.3 illustrates the relations between the tables using primary and foreign key constraints. Consider an employee table with primary and foreign key indications which are interlinked with other tables. Schema normalization is an important factor in designing a relational database schema. In the data normalization, queries contains joins which make query operations complex especially for large queries for retrieving data from multiple tables.

In Figure 2.3, gives an example of a relational schema model. The tables contain information about employee details. The data is normalized and the relative information is stored in different tables that are connected using primary and foreign key constraints. For instance, the information regarding the annual salary of an

---

employee and the city of the employee is needed. For example, John is an employee living in Berlin and his annual salary is 120000 Euros. The information about the employee is stored in different tables and can be retrieved any time using JOIN operation. It is a simple query that needs less number of JOIN operations. In a real-world scenario, there are a large number of interconnected datasets. To retrieve the required information, a complex query using JOINS is used. In such cases, due to a large number of JOINS, the execution speed decreases. Thus, resulting in a performance problem.

2.2.1 RDBMS Architecture

In relational database, it is important to understand how the data is structured, stores and, authorized. RDBMS supports external, conceptual and internal levels (standard three-level) architecture. As shown in Figure 2.4, Relational database model delivers both physical and logical data independences. The physical and logical data independences isolate the standard three-level architecture.

![Three-level architecture of relational databases](https://www.objectivity.com/is-your-database-schema-too-complex/)

Conceptual level is also known as the data model which describes the structure of the database for the end users. The schema represents the content of the whole database and relationships among the data. The internal level is also known as physical level defines how the data is stored in the database and the layout of the records in files. The external level allows the particular group of authorized users to view the data in the database. To improve the performance, changing the internal schema without affecting the conceptual level is possible in physical data independence [RU15]. Logical data independence is capable of changing the conceptual schema without affecting the external level [RU15]. There are numerous RDBMS like Microsoft SQL Server, Oracle Database, MySQL, and IBM DB2, PostgreSQL and many more. In the thesis, the PostgreSQL database is used for performance evaluation.

---

RDBMS provides a useful feature called an ACID (Atomicity, Consistency, Isolation, and durability) property which provides assurance for reliable transaction [BD04]. PostgreSQL database provides properties of the transaction (that is ACID properties) and the transactions are controlled using transaction control operations (commands used are BEGIN, COMMIT, END, and ROLLBACK). The commands cannot be used when dropping or creating the new tables as they are automatically committed in the PostgreSQL database.

2.2.2 Database Queries

Data operations are done using queries. To retrieve the required information from tables we often use different SQL codes. The query mechanism and different syntaxes are used for different queries to scan and fetch the data are described below.

The conjunctive statement contains the syntax SELECT FROM WHERE which is used to display the required data from the table. Similarly, Functional query further specifies create, manage and control fields used for data manipulation, database management, and access control respectively. OLTP (online transaction processing) query execution is much faster. DSS (decision support system) is used in case of complex queries to retrieve data from a large database. Due to complex queries, DSS is costly in terms of execution time and system resources.

Most of the relational database is based on transactions. A transaction is a logical unit which is used to manage the data such as update, delete, read, and create. These transaction characteristics are called ACID properties which provide accuracy, data integrity, and completeness.

The systems are OLTP when the database structures are designed by focusing on transactional processing. On the database level, the transactional operations aim to fast and powerful queries to the database. OLTP commands INSERT, UPDATE, DELETE are mostly used operations. OLTP is also used in interactive mechanisms like web services. For example, consider the banking transaction. There are many customers using their accounts. The system must execute the operations performed. In case of many concurrent transactions (that is multiple user access) at the same time. The operations must be completed effectively.

DSS is a complex creation that is integrated with the various components. For example, a business company details that has numerous components such as products, customers, orders, timestamp, region and many more. If such company needs to extract the information from the aggregated business information first, the company analyst must model the data accordingly. Then the analyst has to analyze and load the data from various data sources to the data warehouse. The queries are often complex which involves aggregation functions. For the complex queries, the execution speed is low. It can be improved by creating indexes [CDG01].

2.2.3 PostgreSQL

PostgreSQL is an open source relational database. It aims to provide high performance, robustness and reliability to the clients. PostgreSQL stores data in tables

---

7. https://www.postgresql.org/
and generally access the data using SQL language. Since PostgreSQL is a relational database it requires predefined data structure based on the application requirement. Related data is stored in different tables accessed using the JOIN operation. PostgreSQL supports not only system-defined data types but also user-defined data types, indexing types, functional language are used by the user according to their requirements.

**Schema Design**

It is mandatory to design a minimum of one schema for every database. Schema covers the content as listed in Figure 2.5.

![PostgreSQL Schema](https://hub.packtpub.com/overview-postgresql/)

Schema design helps in organizing and identifying the wide range of data into a finely-grained structure providing a unique namespace. Whenever a new database is created it must have one schema minimum. Schema is used for many different reasons. For example, the schema is used for control authorization that means when people using the environment simultaneously one could create rules to access the database schema based on individual roles, organizing the database objects, maintaining third-party SQL code, and efficient performance.

**Tables:** Tables are created specifying the name of the table and by using CREATE TABLE command, providing the column names with data types.

**Range:** It is a data type which usually used for selecting the range of values.

**View:** Suppose we need information by combining two different tables but we do not want to write a query every time. For such situations, we use view command. We create a view and refer it to a query. Consider tables with the city and an employee below example shows how the view is created.

And can execute it by using a simple command as shown below.

```
SELECT * FROM myview;
```

---

8[https://hub.packtpub.com/overview-postgresql/](https://hub.packtpub.com/overview-postgresql/)
9[https://hub.packtpub.com/overview-PostgreSQL/](https://hub.packtpub.com/overview-PostgreSQL/)
CREATE VIEW myview AS
SELECT city, employee_name, employee_salary, employee_address
FROM employees, city
WHERE city = name;

Functions: PostgreSQL provides various functions by the combination of declarations, expressions, and statements. It also has in-built functions for system-defined data types. Functions can be used for different aspects. For instance, developers use functions as an interface for the programming language to conceal the data model, to perform complex logical queries and many other requirements. There are different possibilities to access the data in PostgreSQL. It allows multi-procedural language to write triggers such as default system-defined language like PL/pgSQL and C, C++, Python, R in case of user-defined APIs [Dou05].

Type: It is important to provide the right data type for each column in a table. For large datasets changing the data type is costly. In order to make the table efficient we must pick appropriate data type. Some of the most used data types are numeric type, character type, date and time type. Every relational database supports different types of default data types like text, integers, Boolean and also an array of values with system-defined data types. For designing a schema, the table ensures the allocation of data types for each column. For instance, the database provides the finite number of data types and if it does not permit the use of new data types then it decreases the flexibility of data model [Mak15].

In PostgreSQL this restrictions are improved a little bit by initiation of JSON data type in PostgreSQL documentation release version 9.2 and further development of it. The JSONB data type was introduced in version 9.4. This JSON and JSONB enable storing unstructured data into a PostgreSQL database. The major difference between them is, for JSON type the data is stored as a copy same as the JSON input text. Whereas JSONB data is represented in a binary format that is binary code not in UTF-8 or the ASCII format [Mak15].

Indexes: Indexes are used to improve the performance of the database by scanning the related data pages. Indexing works just like an index page of a book. By following the index, we access the required data much faster which contain the required term. Therefore, we can state that it is easy to fetch the data using indexes instead of scanning the whole book. PostgreSQL database supports different types of indexing from the simplest B-tree index to Geospatial GiST (Generalized Search Tree) indexes. PostgreSQL supports different types of indexes:

1. B-tree index (Balanced-tree): It is a default index when the index is not mere with CREATE INDEX command. Then indexing the data on both sides of

---

13https://hub.packtpub.com/overview-PostgreSQL/
14https://hub.packtpub.com/overview-postgresql/
17http://www.postgresql.org/docs/9.1/indexes-types.html
2. Technical Background

the tree is almost equal. B-tree indexes uses operators like =, <, >, <=, >= whenever, a column involved in comparison.

2. Hash index: Hash index handles equality predicates. It is generally less used because it does not help much in the PostgreSQL database.

3. GiST index: It is an infrastructure inside the GiST index many different plans are implemented. Gist index is generally used in geometric data-types and also supports full-text search.

4. GIN (Generalized inverted index) index: GIN is useful for complex structure for instance arrays, and full-text search.

5. BRIN (Block range index) index: It helps the data to arrange systematically by storing lower to higher values in each block. Partial indexing, multicolumn indexing, and unique indexing are some other kinds of indexing which support PostgreSQL database.

The RDBMS architecture and database query techniques are explained. The queries help to understand how the data is retrieved from the database.

The PostgreSQL database is a relational database that supports unstructured data JSON and JSONB. To retrieve the data SQL (Structured Query Language) is used. PostgreSQL supports full-text search or a phrase search using text indexing. The data model helps in identifying the huge datasets into a proper structure providing a unique namespace. In the case of large complex queries, the database queries need to perform JOIN operations which is costly and slows the execution time.

In the case of complex operations, NoSQL databases work effectively. The NoSQL databases do not require any JOIN operations because of its flexible schema structure. So, for the flexible schema NoSQL databases are the best fit.

2.3 NoSQL Databases

NoSQL databases are designed for large scale data sets. The NoSQL databases are first introduced in the early 2000s, NoSQL databases are well known as non-relational databases which do not contain tables. NoSQL technology is developed for distributed storage on a large set of data, parallel computing across different servers, and to overcome the limitations of RDBMS [MH13]. To interact with the data, every database has its own unique language. Big companies like AMAZON, FACEBOOK, GOOGLE, etc, adopt this type of environment to cope up with highly scalable data for massive read and write requests [HJ11a].

2.3.1 NoSQL Database Characteristics

With the immensely growing data it is important to find a new solution alternative to the existing ACID databases. A new solution is designed under the BASE model (basically available, soft-state, eventual consistency). The consequences of the scale out ACID properties arises a conflict between different aspects of availability in
### 2.3. NoSQL Databases

<table>
<thead>
<tr>
<th>Databases</th>
<th>Consistency</th>
<th>Availability</th>
<th>Partition tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDBMS (MySQL, PostgreSQL etc.)</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Cassandra, CouchDB etc.</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>MongoDB, Hbase etc.</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 2.1: CAP theorem [MH13]

A distributed environment. That is not fully solvable resulting the CAP-theorem [Sim12].

Table 2.1, depicts that every database can achieve only two characteristics out of three characteristics over the distributed network.

**Consistency**: Every user can see the data regardless of updates like delete, create, updates, and so on.

**Availability**: Data should be available to the user in a network even if one of the nodes is down.

**Partition tolerance**: A partition tolerance system can sustain even the entire network is failed. It should maintain its characteristics, and continue its operations.

Some of the NoSQL databases concentrate on availability and Partition tolerance. This result to know BASE (Basic, Available, Soft-State and Eventually Consistent). It is an alternative to ACID properties. The BASE has no transaction properties. The BASE provides high availability, simple, and faster but has weak consistency.

#### 2.3.2 Classification of NoSQL Databases

NoSQL databases are categorized into four types *key value stores, wide column stores, document oriented databases and graph databases*.

**Key Value Stores**

Key value stores are one of the popular NoSQL databases used to store the data. Contrary to some of the relational databases, it supports elasticity, scalability, and easy to manage. The database stores keys in alphanumeric format. The values are text strings, or complex set of strings, arrays, lists. The values are in standalone tables. Data search is performed against key not values. So, the key is always unique [MH13]. Below shows the example Figure 2.6 on the following page.

The different implementations of Key-value stores are Dynamo, Redis, Riak, Voldemort, BerkeleyDB [MH13]. Key-value databases are best suited for fast data access via keys, scalability in fetching the values from an application like retrieving the information of the product, managing user sessions, and so on. There are different advantages to using Key-value databases. They provide flexible data modeling.

---

Contrary to the traditional RDBMS, data retrieval does not involve fetching the data from columns and rows. The architecture involves high performance, it is simple to operate and handles a massive amount of load. There are some drawbacks of Key-value databases, they have no unique query language associated with different executions, mostly support simple databases [Moh16].

Wide Column Stores

The wide column stores also known as extensible record stores. The data stored in tables which contain many rows with unique row key. Consider a single row as

---

18. Figure 2.6: Key value stores data model
19. Figure 2.7: Wide column stores architecture

shown in Figure 2.7 on the preceding page, it shows the first column as row key (the unique identifier of a column). The column in Figure 2.7 on the facing page contains a column name which uniquely identifies every row. In other words, it is a two-dimensional key-value store. The primary usage of the wide column or column family stores is for distributed data storage, large scale data processing like sorting, parsing, conversion between code values (for instance hexadecimal, binary, decimal), and algorithm processing, batch-oriented processing method, and predictive analytics. Wide column databases are generally high performance oriented when querying the data and provide strong scalability. Many organizations like Spotify, Facebook, Big Table model of Google uses Wide column stores [MH13].

Graph Database

Graph databases are first introduced in the early 1990s. At that time, the database community is turning to adapt semi-structured data (in 1990s Graph databases do not have any relation with semi-structured data), the existed technologies are effective for most of the application requirements [AG08]. Today, graph databases gain interest in managing the relationship for a massive set of data by connecting internally between entities. The ideal use case scenarios for using graph databases are, for traversing social networking, pattern detection in forensic investigations, and in biological networking [DS12]. The graph represents the connection between the objects and illustrating the relationship between them [DS12]. Querying in a graph database is traversal [HR15]. Figure 2.8 is a simple example shows the graph model with every node and its relationship has properties. The relationship is stated by dereferencing using pointers. The query can execute with only one index lookup. This approach provides high performance than relational databases. Whereas in the relational database to retrieve information from multiple tables. The database executes the query using foreign keys that have multiple lookups on indexes. Maintaining the performance is more important for every database even when the data volume is increased. It increases performance when querying the interconnected data against any relational or other NoSQL databases. Unlike

\[20\]https://blog.octo.com/en/graph-databases-an-overview
RDBMS, the performance of graph database is stable even the datasets are increased massively [HR15]. Popular Graph databases are Neo4j 21, InfiniteGraph 22, and FlockDB 23.

**Document Oriented Database (DODB)**

DoDBs are JSON or JSON like documents which are encapsulated using key-value pairs. The key is a unique identifier which is also a unique ID in a database collection. The ID helps to identify the document explicitly within the collection. Complex data can be handled effectively. To work in a big data environment, DODB is best suited unless there is any specific database for a specific purpose like graph database [KSM17]. Since DODBs has a flexible schema, it is possible to modify or change anytime. It provides high scalability, efficient performance and it does not depend on any data format. Some of the most popular DODBs are MongoDB 24, CouchDB 25, Riak 26. Figure 2.9 describes the unique characteristics of a different data model,
2.3. NoSQL Databases

a MongoDB database and compare the performance with companies database (i.e. PostgreSQL).

2.3.3 MongoDB

MongoDB is an open source document oriented database. These documents are grouped together in a collection as shown in Figure 2.10.

![Figure 2.10: MongoDB document store](https://www.slideshare.net/mongodb/mongodb-schema-design-practical-applications-and-implications)

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Database</td>
</tr>
<tr>
<td>Tables</td>
<td>Collection</td>
</tr>
<tr>
<td>Rows</td>
<td>Documents</td>
</tr>
<tr>
<td>Indexing</td>
<td>Indexing</td>
</tr>
<tr>
<td>Joins</td>
<td>Embedded documents or lookups</td>
</tr>
</tbody>
</table>

Table 2.2: Terminology difference in MongoDB and RDBMS

A collection is similar to a table in relational databases. Relational database contain rows and columns in a table whereas, MongoDB contains documents in a collection. To design the database schema consider Table 4.1 on page 42 which shows the difference in relational and MongoDB terminology. The documents stored in a collection are BSON documents (Binary encoded JSON). Documents are key value pairs. MongoDB data model is flexible. The data from different tables are embedded in a collection which increases its query performance. MongoDB has its own query

---

28 [https://www.slideshare.net/mongodb/mongodb-schema-design-practical-applications-and-implications](https://www.slideshare.net/mongodb/mongodb-schema-design-practical-applications-and-implications)

29 [https://www.slideshare.net/mongodb/mongodb-schema-design-practical-applications-and-implications](https://www.slideshare.net/mongodb/mongodb-schema-design-practical-applications-and-implications)
2. Technical Background

language which makes it easy to retrieve information from the database. The database Indexing, aggregation, map reduce are some of the powerful query characteristics of MongoDB.

2.3.3.1 MongoDB Architecture

Relational databases served the data world over decades, it shows a very mature approach in dealing with the database. Designing MongoDB includes the proven abilities of RDBMS with newly introduced features of the NoSQL database. Figure 2.11 depicts the architecture blending RDBMS with MongoDB (NoSQL technology) key features. In the MongoDB database, the data is collected as JSON documents. Due to its rich query capabilities, MongoDB provides the abilities of relational databases along with high performance, flexibility, and scalability.

![Figure 2.11: MongoDB nexus architecture](https://www.mongodb.com/white-papers)

2.3.3.2 Schema Design

Before designing a schema it is important to know the technical terminology of MongoDB Section 2.3.3 on the preceding page. The decision in designing an efficient data model is based on the application requirements. The data model can be designed in embedding the data into a single collection or by referencing the data from different collections. It is possible to apply both approaches depending on the application requirements.

The embedded method id generally similar to the denormalization model. Figure 2.12 on the next page shows the embedded document. In the example, the contact and address data are embedded into a single document. The model can be designed one to one, one to many relationship in an embedded documents. The main advantage of embedding is that the database retrieves the information without the use of JOIN operation. As a result their is an increase in performance.

---

30https://www.mongodb.com/white-papers
31https://www.mongodb.com
The referencing data model is also known as a normalized data model. Figure 2.13 shows the example of a normalized data model. In the data model, the data is retrieved by referencing (using the unique object id). This model is best suited to design model for huge data sets.

### 2.3.3.3 Query Model

Some of the important factors related to the query model are discussed in this section. MongoDB supports mongo shell, python, ruby, R, Scala, c#, and many more programming languages to query the data from the MongoDB database.

**Query options**

MongoDB has its own querying language to perform operations like finding the number of documents, matching, ranking, and projecting. MongoDB also supports many ranging queries and other expressions like $gte (greater than equal), $lt (less than), $ne (not equal). To performs the arithmetic operations and to query the data in array $elemMatch keyword is used. The operators are case sensitive.
Indexing

Indexing is one important feature of MongoDB. Indexing provides efficient results upon querying the data in a collection. Indexing is done on a specific field, multiple fields, or on the whole collection. When we query the data without indexing, MongoDB must inspect all the documents in a collection which lowers the execution speed. With indexing, MongoDB restricts the number of documents to examine from the collection. When creating the collection MongoDB has a unique index (i.e. default _id index). This prevents from inserting any identical data with the same value. MongoDB is integrated with various indexing type namely, text search indexing, single, compound indexing. Partial, unique are some of the properties of indexing.\(^{32}\)

By default MongoDB creates default indexing on the _id field. Indexing is done on any one field. With reference to single indexing, Multiple indexing is created by indexing on two or more fields in a collection.

Text indexing

Text Indexing is performed on the string content field on a specific column or multiple columns in a database. MongoDB supports partial and full-text search. Text indexes are applied to the single field, multiple fields, or on a wildcard specifier (i.e. $**$) for indexing every field that contains text content in a collection. There are certain limitations of text indexing. For instance, we create a single text index on a field for the text query.

```javascript
db. <Collection name>.createIndex({“topic” : “text”})
```

Indexing is executed. When we run the compound indexing to the same collection as shown below it throws an error stating that the text search index already exists.

```javascript
db. <Collection name>.createIndex({“topic” : “text”, “abstract” : “text”})
```

So, if we want to create another text index on the same collection we must drop the previously existing indexing.

```javascript
db. <Collection name>.dropIndex({“topic_text”})
```

and create a new one depending on the requirement.

```javascript
db. <Collection name>.createIndex({“topic” : “text”, “abstract” : “text”})
```

So, for creating a text indexing, it should be noted that there are certain rules to perform indexing on a column in a database.

\(^{32}\) [https://docs.mongodb.com/manual/indexes/](https://docs.mongodb.com/manual/indexes/)
2.3. NoSQL Databases

Full Text Search Support

Full-text search provides various features of MongoDB.

1. Stop words: The words like a, an, the, at, etc. are filtered out from the content.

2. Stemming: In stemming the words like standing, happening is reduced to originated words stand, happen respectively.

3. Scoring: Ranking the most relevant word depending on a search result.

With the use of MongoDB text indexing, it is possible to search text on one field (single field text indexing), multiple fields (compound text indexing) or the total text in a collection using wildcard specifier indexing [32]. Using MongoDB it is possible for phrase search. It fetches for the document which contains relevant information based on the given phrase.

Aggregation

Query operations can be performed in different methods in the MongoDB database.

1. Aggregation pipeline.

2. Map Reduce

3. Other aggregation functions.

In aggregation pipeline, the data undergoes set of operation in multiple stages in a pipeline such as $match, $project, $group, etc that computes the data and return the result. Use of pipeline provides efficient data operations. Aggregation pipeline is the preferred method on data in MongoDB [32].

Map Reduce is a data processing model for conducting queries on a large amount of data and return the result. In the map phase, MongoDB maps the key-value pairs. If the same key has multiple values, the reduce phase is applied to the collection and process the aggregated data. The results are stored in a collection. MongoDB map-reduce functions run using javascript so it provides high flexibility [32].

MongoDB provides some other functions like distinct, count, limit, etc [32]. MongoDB provides other important administration and maintenance features like configurations, scalability, persistence, sharding, availability, and security.

In this chapter, the brief introduction of decision support for R&D management is discussed. The R&D activity gives us a basic knowledge of the factors involved in developing a new project. In the next section, the Relational database details are explained. The data model is designed using strict schema structure. To query the data from the database standard SQL language is used. PostgreSQL is one of the most advanced relational databases in today’s world. The schema and the query model along with the types of indexes are discussed.
NoSQL database plays a key role in managing the unstructured data. NoSQL databases are mainly classified into four types. Each type has its unique features. The usage of these databases depends on the application requirement. Furthermore, a rich document-oriented MongoDB database is explained. MongoDB has the abilities of the relational database along with additional features such as flexibility, scalability, and high performance. The data is stored as JSON documents which are represented in a BSON format. There are two possible data modeling techniques in MongoDB. Embedded and the referencing data model. Each of it has a unique purpose. The two approaches can also be used together to develop a database especially in case of large data sets. MongoDB has its own query language to retrieve the relevant information from the database. MongoDB Indexing helps in minimizing the data search which results in high performance.
3. Requirements and Concept

The chapter discusses the structure of the data in the PostgreSQL database server, its limitations, the applicability of NoSQL technology for data migration from the PostgreSQL to NoSQL database, and investigating the query performance on data that is taken from PostgreSQL database. The investigation concludes which of the two databases (PostgreSQL and NoSQL database) are best fit in terms of query performance.

The data used for the thesis is taken from Mapegy’s PostgreSQL database server. The information in the database provides data-driven decision support to the R&D managers. The database provides updates of the data that contains information of organizations, and expert profiles. The data provides innovation insights for R&D managers, investors, and business analysts. The decisions focused on factors such as investment decisions, technology decisions, decision making based on resources, cost, and time. The thesis mainly focused on technology-oriented decision making support. The database contains huge sets of data with patents, science, documents, organizations, experts, metadata, and so on. The database contains information about the date and time, title, country, language and many more. Furthermore, it also contains the information of last updated records time and date. For such huge dataset, the hierarchical structure is modeled in different tables with a primary key and are connected using link tables. The data is derived from millions of research papers, articles, patent offices, social media, and many more innovation-related platforms.

The database is designed with various tables Figure 3.1 on the next page of different types as follows.

1. entity related tables contain the basic information of entities.

2. link tables that contain information about the connection between entity tables.

3. KPI tables (key performance indicator) provide indicators to score e.g assets (table). KPIs are used to evaluate a company’s success rate based on achieving business-oriented goals.
4. Stat tables contain information about statistics like documents count. It monitors the number of records present in the database and the number of deleted records.

5. Trend tables provide information about the number of records count per year (certain time interval) from the database.

The data used in this thesis is taken from three different tables that are entity_docs table, entity_players table, link_player_docs table. The chosen datasets provide information about PATENT and Scientific publications from millions of sources. The data is used by the R&D managers to make a decision on selecting the relevant information depending on the requirement.

- **entity_docs**: In the table the information of the number of patents and scientific publications is shown. It also provides the information regarding the documents such as the title of the document, abstract, metadata, time and date inserted, last updated and so on.

- **entity_players**: The table provides information about the number of institutions, and expert profiles. It also provides the data regarding the address of the institution, experts, the country code where they belong to, global positioning and many more.
• **link_players_docs**: The table provides the connection between the above two tables. Figure 3.2 shows the relationship between the tables. The tables contain information about the number of columns and the connection between the tables.

![Diagram showing the data model of PostgreSQL database containing required data](image)

Figure 3.2: Data model of PostgreSQL database containing required data

PostgreSQL database has a query performance issue. The database contains various tables. The tables are normalized and are linked together using primary and foreign key constraints. When the complex query that requires many JOIN operations are performed to collect relevant information from multiple tables. The JOINS takes time to retrieve the information from the tables decreases the query performance. For better explanation consider a complex query Listing 3.1 on the next page.

The explanation of query syntax is as follows

- selecting the columns to be displayed.
- filtering doc_type (document type) SCIENCE as nb_science, and PATENT as nb_patent
- counting the number of SCIENCE, and PATIENT documents from data_warehouse (PostgreSQL database server) by connecting the entity_docs table, and entity_players table using link_players_docs table for a given phrase.
- group the table by player_id.
3. Requirements and Concept

### Listing 3.1: Finding total number of scientific publications and patents for the institutions

```sql
select
  z.player_id,
  z.player_type,
  z.player_sub_type,
  z.player_name,
  z.country_code,
  z.address,
  count(*) filter (where doc_type = 'SCIENCE') as nb_science,
  count(*) filter (where doc_type = 'PATENT') as nb_patent
from data_warehouse.link_players_docs x
join data_warehouse.entity_docs y on x.doc_id = y.doc_id and y.doc_type in ('PATENT', 'SCIENCE')
join data_warehouse.entity_players z on z.player_id = x.player_id
and player_type = 'INSTITUTION'
where tsv_fulltext @@ phraseto_tsquery('video game console')
group by z.player_id
order by nb_patent desc
limit 500;
```

- Sort the result in descending order.
- executed query output is limited to 500 rows.

The output of the execution retrieve all the INSTITUTIONS who have published the scientific publications and patent information from the given title ('video game console'). From Listing 3.1, it is shown that to retrieve data from multiple tables we required JOINs. This process of retrieving data from multiple tables is a time-consuming process resulting in decreasing the performance. To find an alternate solution to the problems, one of the NoSQL databases is selected. By designing the data model, data migration from the PostgreSQL database and implementing different queries that are mentioned in Section 1.1 on page 2 in the introduction chapter. By investigating the query performance between the existing database and NoSQL database, the decision is taken that which of the databases (PostgreSQL, or NoSQL database) provides high performance.

### 3.1 NoSQL Database Selection

One of the NoSQL databases is selected for implementing the queries to the data that is extracted from the PostgreSQL database. The performance evaluation between the existing and selected database helps in decision making for selecting the database. The evaluation is implemented for the data extracted from PostgreSQL.

NoSQL databases have unique properties that differ in consistency, performance, scalability, reliability [HELD11]. As discussed in section 2.3.2, NoSQL databases are
3.1. NoSQL Database Selection

classified into four different categories. The criteria for selecting the NoSQL database in this thesis work is based on important factors like high performance, marketability, reliability, open source support. Most widely used databases ranking is determined in the DB-Engines ranking. The ranking is based on different aspects like the most used database, frequency in the number of job offers, technical discussions. For instance, consider Table 3.1 that shows the ten most popular databases according to DB-Engine 2019 ranking. From the Table 3.1 it is clear that still RDBMS are most popularly used databases. As discussed from Listing 3.1 on the facing page, performing a large query that involves retrieving information from multiple tables decreases its query performance. In such scenarios, NoSQL databases are useful.

This chapter explains the requirements of the NoSQL database to work with the data which is extracted from the PostgreSQL database. We consider the set of features that should be integrated into the database. NoSQL database is chosen such that it matches the features of PostgreSQL database. The chosen database should have the following important features:

1. The chosen database must be available at any time.
2. It should be flexible to store complex data.
3. The important part of the database is the analysis of stored data. So for data analysis, the database needs to support many data analytic features. NoSQL database should support the query executions similar to the company’s database.
4. Easy query operations that increase the performance effectively.
5. Text indexing support.
6. Documents involving in JSON format.

Table 3.1: Top ten most popular databases [1]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Oracle</td>
<td>RDBMS</td>
<td>1268.84</td>
</tr>
<tr>
<td>2</td>
<td>MySQL</td>
<td>RDBMS</td>
<td>1154.27</td>
</tr>
<tr>
<td>3</td>
<td>Microsoft SQL Server</td>
<td>RDBMS</td>
<td>1040.26</td>
</tr>
<tr>
<td>4</td>
<td>PostgreSQL</td>
<td>RDBMS</td>
<td>466.11</td>
</tr>
<tr>
<td>5</td>
<td>MongoDB</td>
<td>Document store</td>
<td>387.18</td>
</tr>
<tr>
<td>6</td>
<td>IBM DB2</td>
<td>RDBMS</td>
<td>179.85</td>
</tr>
<tr>
<td>7</td>
<td>Redis</td>
<td>Key-value store</td>
<td>149.01</td>
</tr>
<tr>
<td>8</td>
<td>Elasticsearch</td>
<td>Search engine</td>
<td>143.44</td>
</tr>
<tr>
<td>9</td>
<td>Microsoft Access</td>
<td>RDBMS</td>
<td>141.62</td>
</tr>
<tr>
<td>10</td>
<td>SQLite</td>
<td>RDBMS</td>
<td>126.80</td>
</tr>
</tbody>
</table>

1 https://db-engines.com/en/ranking
There are various rich document-oriented databases. MongoDB and CouchDB are the most widely used databases [1]. Query execution speed of MongoDB is faster compared to CouchDB [Bha16]. CouchDB uses an elegant map-reduce syntax for querying the data 2. Whereas, MongoDB has its own query language which is easy to learn for the people who have SQL knowledge. Additionally, it provides map-reduce function 3. It supports rich document-oriented full-text search support, it provides very flexible schema (data model). So, data is represented in a simple way to query data in an efficient way without any join operations. Below Figure 3.3 shows the functional, non-functional requirements, and techniques of MongoDB. The techniques connect functional and non-functional system properties that support MongoDB. Scanning queries, Filtering, Full-text search, Analytics, and Conditional writes are functional requirements. Among the requirements, the queries, indexing, and analytics are our primary requirements for implementing the data in the MongoDB database. The query capabilities of MongoDB guarantees consistency. The data is retrieved using a unique ID. For performing queries efficiently, secondary indexes are employed. The indexes reduce the scanning items which provides high query performance. The full-text search is performed by using text indexing in the database.

Figure 3.3: Functional, Non-Functional requirements, and techniques of MongoDB

---

2 http://couchdb.apache.org/
3 https://www.mongodb.com/white-papers
3.2 A Framework for Decision Support in R&D Management

This thesis investigates the query performance of PostgreSQL and MongoDB databases. We implement a NoSQL databases framework to support decisions for R&D management. The framework aims to address the limitations of the current system and to analyze the performance of the proposed approach. The framework categorized into 4 sections for decision support in R&D management Figure 3.4. The idea for designing a framework is taken from an article [SKC16].

![Diagram](image)

Figure 3.4: Framework to support R&D management

1. **Storage layer:** In this layer, the unstructured data is imported to MongoDB. The data in MongoDB is stored in BSON (binary JavaScript Object Notation) format [SKC16].

2. **Analysis layer:** For the thesis, the queries used for data analysis in MongoDB need features such as indexing and full-text search, sorting, matching, grouping the documents and projecting it and so on. [SKC16].

3. **Application layer:** We used R framework shiny for an interactive web application. Using R programming language data is queried and the result is displayed on the web application API [SKC16].

3.3 Data Migration

PostgreSQL database server contains various huge datasets. According to the requirement, data migration is carried from entity_docs table which contains information related to patents, scientific publications, and metadata. The entity_players
table that provides information about organizations, experts and research institutions. The link_player_docs table contains information about the connection between tables. The schema Figure 3.2 on page 25 contain information about the number of columns and the connection between the tables.

Designing a MongoDB data model varies from the PostgreSQL data model. MongoDB does not need schema design for data modeling in this work. There are two different design patterns for designing a data model in MongoDB. The first option was migrating every database table into different collections to a target database which is a pragmatic approach. The second option is embedding multiple tables into a single collection. Embedding the related data as documents is simple. In this thesis, the data extracted from the PostgreSQL database is embedded into a single collection in MongoDB database. The data extracted consists of relative information that can be embedded into one collection. The advantage of embedding the documents into a collection is that they avoid using references or lookups. The embedded collection is faster in query execution that results in high performance. However, due to the lack of joins, there is data redundancy. This results in high memory usage.

3.3.1 Embedding in MongoDB

For better understanding about embedding in MongoDB. Consider an example, the relational database Figure 3.5 is designed by proper normalization. The data normalization decreases data redundancy, increases data integrity, low space consumption [KGK14]. But in the case of MongoDB, it does not support joins so the data is denormalized. In the Figure 3.5 Table:1 and Table:2 are connected with primary and foreign keys. Whereas, Table:3 shows the denormalization. Here the joins are replaced by merging two tables. Table:4 shows the structure of the embedding document. Embedding is similar to denormalization. In an embedded collection, the foreign key relationship is embedded as an array of documents.

![Figure 3.5: Embedded data collection structure](image-url)
3.3. Data Migration

3.3.2 Data Migration Phases

There are three different phases in the data migration process namely planning, migration, and data validation [MD18].

- **Planning:** In the planning phase, hardware specification, data configurations, and features of the target database (MongoDB) are observed. Proper planning helps to decrease the inherent risk factors like missing data, data duplication, data inconsistency or quality issues.

- **Data migration:** In this phase, the data is migrated by extracting the data from the source database (PostgreSQL) and importing it into the target database (MongoDB) using `mongo shell`.

- **Data validation:** The process of measuring the data quality after data migration. The process is tested by simply querying to count the number of documents (rows and columns) in the target database (MongoDB) and compare it to the source database (PostgreSQL). If the number of documents (rows and columns) is the same in both source (PostgreSQL) and target database (MongoDB) then the data migration is successful.

For migrating data, data should be extracted and restructured from the PostgreSQL database, transform, and load the data to the MongoDB database. Figure 3.7 on the following page shows the data migration process that explains how the data migration is carried out from the PostgreSQL database to MongoDB. The data is
initially copied from PostgreSQL database server in CSV (comma separated value) file format, transformed, imported to the MongoDB, and validated. The CSV data is directly imported using *mongo shell*. This type of data is known as pass-through data. But the data is checked for any data duplication by querying the data using query aggregation Listing 3.2. If the duplicates are present, they are deleted. The data migration design is constructed such that the data extracted from PostgreSQL should match the data in MongoDB. It means, MongoDB database contains the same amount of information with the number of rows equal to the source database (PostgreSQL database).

```javascript
db.doc.aggregate([
#Grouping all doc_id's together
{ $group: {
  _id: {doc_id: "$doc_id"},
  # View documents that have same doc_id and Provides the count that
  # shares same group key
  duplicate: { $addToSet: "$_id"},
  count: { $sum: 1 }
}
},
#To get output only to the group which is greater than 1.
{ $match: {
  count: { "$gt": 1 }
}
},
# sort in a decending order
{ $sort: {
  count: -1
}
}
]);
```

Listing 3.2: Finding data duplication using aggregation

The implementation process of data migration is explained clearly in Figure 3.8 on the facing page.

Following the procedure mentioned in the Figure 3.8 on the next page, the data migration is carried out. The data model specifically focused on entity_docs, entity_players and link_players_docs from PostgreSQL data_warehouse. The sample document after embedding from three tables is shown as Listing 3.3 on page 34.
The data is extracted from the PostgreSQL database as a CSV file and imported to MongoDB using `mongo shell`. After the migration, the data is checked for data duplication. The duplicate documents are removed from the MongoDB database.

MongoDB provides a flexible schema structure. The data is modeled as an embedded collection. The data is denormalized and merged into a single collection. The data migration is carried out in three phases that are planning, data migration, and data validation. The procedure involved in data migration is discussed in this section.

### 3.4 MongoDB Query Model

In this section, the query model, concerning syntax and indexes used for applying business logic are discussed.

#### 3.4.1 Query Language

In MongoDB, queries are expressed as JSON objects. The prototype is built using R programming by developing a user-friendly interactive web application using R framework SHINY. Initially, the data is imported using `mongo shell`. It is an interactive interface that is completely based on JavaScript. The queries used and the time taken to execute queries is discussed in the implementation chapter.

The database structure contains the following basic components to query the data.
A **collection name** which helps to have an idea about where the desired documents are present in the database. Firstly, data is migrated from a single table (entity docs) from PostgreSQL to MongoDB collection (main data) in MongoDB database. Secondly, the data migration is performed by embedding the documents of a collection with 23.7 million documents (entity doc). The data is migrated to compare the query execution speed of MongoDB and PostgreSQL database. The queries are performed on a single table that does not involve in joins (entity docs) and on interlinked tables that are embedded as a single collection in a MongoDB database. The data is migrated single table and embedded collection because to investigate the speed of the query execution that involves no JOIN and JOIN operations respectively.

A **querying method** specifies the method for investigating or retrieving the data from the MongoDB database. Aggregation pipeline is used for querying the document. The detailed explanation of the aggregation pipeline is discussed in Section 3.4.1.1 on the next page.

**Query Structure**

MongoDB has different query methods. That is the MongoDB query using `find()` method, aggregation pipeline, and Map Reduce. The MongoDB query structure is explained with an example in a **Listing 3.4 on the facing page**. It illustrates the query structure with a sample query that fetches the information of 1000 documents.
about the *title* field which is HEATER. The query is prefixed with `db` that contains the collection name where the queries to the data are applied. Query command `find()` is used to find the documents fulfilling the desired output.

```plaintext
#structure of a query
db.Collectionname.Querycommand( Querydocument )
  .Projectiondocument()
  .Limitdocument()

# example

db.data.find( { 'title': 'HEATER' } )
  .projection({})
  .limit(1000)
```

Listing 3.4: MongoDB query structure and example query

### 3.4.1.1 Aggregation Pipeline

In this section, the detail working procedure of the aggregation pipeline is discussed using an example query. The aggregation pipeline discussed because it is used for developing queries in the thesis.

In MongoDB, aggregation pipeline is a preferred method for huge data collection or huge sets of collections. It has a built-in query optimizer, it makes data processing easy across stages and gives optimal results.

Speed and consistency of data access and data retrieval are important factors for evaluating the performance of a database. In aggregation pipeline computation of the result is executed at each stage of operation. The result at each stage is evaluated and returns the output result. Data execution in the aggregation pipeline is based on the proper ordering of the operators and on data dependencies.

Aggregation pipeline gives the output collection by retrieving the required data from a collection. Consider Figure 3.9 on the next page shows the process of data aggregation. When the data is requested by developing an aggregated query as input, MongoDB identifies the requested data including aggregation operators. And it analyzes the data requested and starts executing in different stages depending upon the query. The transformed output data is retrieved following the aggregation procedure.

To understand aggregation pipeline better, consider a collection called docs which have many documents and the sample document is shown in Listing 3.4. The sample query shown in Listing 3.5 on the next page explains how the simple aggregation using a $group operation is performed. The query passing through the aggregation pipeline, groups total number of patents and scientific publications.

The result of the query is Listing 3.6 on the following page. In the collection, the query runs through $group stage. The aggregation pipeline runs through different

---

3. Requirements and Concept

Figure 3.9: Aggregation pipeline

```javascript
db.doc.aggregate([
    {
        "$group": {
            "_id": "$doc_type",
            "number_records": {
                "$sum": 1
            }
        }
    }
]);
```

Listing 3.5: MongoDB aggregation example query

```javascript
{
    "_id": "PATENT",
    "number_records": 798666
},
{
    "_id": "SCIENCE",
    "number_records": 201334
}
```

Listing 3.6: MongoDB aggregation example query output

stages and transforms the documents when it passes through the pipeline. Now consider an example with multiple stages Listing 3.7 on the next page. The aggregation pipeline in the Listing 3.7 on the facing page is an array of different expressions. Every expression is a stage. The stage operators tell about the operation performed on the stage. Aggregation pipeline processes the document through the pipeline. Each stage in pipeline references the output result of the before stages.

1. **First stage**: The pipeline passes through $match operator. The operator finds all the documents related to the particular field of interest. The pipeline executes given text "video" by $match (The text search is possible only after text indexing. The text indexing process is discussed in section Section 3.4.1.2 on page 38). After the text search, it passes through the same stage with another $match operator. Where it finds all the patent documents.
2. **Second stage**: From the output of the $match stages, the pipeline passes through the $group stage. The total number of experts (player_name) and the origin of the country are grouped using a $group operator. The accumulator $sum is used to generate the total number of records for the given query.

3. **Third stage**: The $project is used for projecting the required columns. The choice of fields projection is selected using the $project operator after the output from the first and second stage.

4. **Fourth stage**: The documents from the output is projected in descending order by sorting the number of records using the $sort operator.

```javascript
db.doc.aggregate([{
  "$match": {
    "$text": {
      "$search": "VIDEO"
    }
  }
},

  {
    "$match": {
      "doc_type": "PATENT"
    }
  }

  ,

  {
    "$group": {
      "_id": {
        "player_name": "$player_name",
        "country_code": "$country_code"
      },
      "number_records": { "$sum": 1 }
    }
  }

  ,

  {
    "$project": {
      "player_name": "$player_name",
      "country_code": "$country_code",
      "number_records": "$number_records",
      "_id": 1
    }
  }

  ,

  {
    "$sort": {
      "number_records": -1
    }
  }
]);
```

Listing 3.7: MongoDB aggregation example query with multiple stages

The order of aggregation is important to improve query performance. The output of the above aggregation Listing 3.8.

```javascript
#1
{
  "_id": {
    "player_name": "KIM, YEONG TAEG",
    "country_code": "KR"
  },
  "number_records": 17
#2
{
  "_id": {
    "player_name": "Joo, Young Ho",
    "country_code": "KR"
  },
  "number_records": 17
#3
{
  "_id": {
    "player_name": "Mathew, Manu",
    "country_code": "US"
  },
```
3.4.1.2 Text Indexing

A text index is used for text search on a string field that helps the user to fetch relevant information for a given text. In the thesis, the text indexing is used to text search to fetch relevant information for organizations or experts and rank them by the number of patents and scientific publications. For instance, if a client wanted to know which company has more scientific publications and patents for a topic title HEATERS. The text search on the text indexed field is given as HEATERS in the aggregation pipeline with some aggregation operations. The output of the query finds the relevant information and ranks the number of scientific publications and patents for a given text search in a particular company.

A text index is created by specifying word text which is similar to regular index. Text indexing is created on a single field called title in a collection in MongoDB Listing 3.9. To test the text index on title field, "$text" operator is used. For example, giving some text to search and retrieving the data is shown Listing 3.10 on the facing page. Indexing is possible also for multiple fields known as compound indexing.

There are some restrictions in using text search

- Only one $text expression is allowed for each query.
- Views in text search is not allowed.
- In aggregation pipeline, the stage that includes $text must be in the first place of the pipeline. This increases query performance by scanning the document to the given text. This reduces the scanning time for the other pipeline operators resulting in increased performance.
- $text does not support the operators $or $not in a single expression.

```
1 db.getcollection("doc").createIndex({"title" : "text"})
```

Listing 3.9: MongoDB Text Indexing

In this chapter, the factors for selecting the MongoDB database is discussed. The database schema is designed as an embedded collection from multiple tables. This
brings an advantage of storing relevant information at one place and also having no joins can increase query performance. However, there is a drawback for this approach. The data duplication. The duplicate data is removed from the database by checking the redundancy using the aggregation pipeline. The data in the MongoDB database is stored in BSON format. MongoDB database has its own query language using `find()` which is used for simple queries. The aggregation pipeline and the Map Reduce. For the thesis, aggregation pipeline is used as it is used for large complex queries and easy to develop a query using aggregation pipeline. The aggregation pipeline works by proceeding through stages. The output of before stage is the input next stage. The stages and some of the operators used in the thesis is explained with an example query in this chapter. The usage of text indexing and its restrictions when querying data using text search is discussed in the section.
4. Design and Implementation

In the Chapter 3 on page 23, we discussed the data warehouse structure of the PostgreSQL database, data selection, the concept of data migration, MongoDB data model and the concept of aggregation pipeline in MongoDB using simple and complex queries.

Detailed design and implementation procedure are provided in this chapter. Major steps like tool selection and data analysis procedures are described. In this chapter, we first discuss the design process and the core process of the MongoDB database because it is important to know the basic operations of the MongoDB database. The operations are important to interact with MongoDB server that runs based on JavaScript. Then, we discuss the tool selection to interact with the data in the database. To show the output query performance, we used a user-friendly interactive web interface called R framework Shiny. Lastly, we provide the queries used and its results for evaluation purpose.

4.1 Design Process

The design process is basically an application design which defines the interfaces and its behavior. Before the process begins with the data import to the MongoDB database. After the data migration, all the components from the data query to developing an interactive web application for data retrieval are described. Figure 4.1 on the facing page describes the structure of the whole system to run our applications. The database involves a three-layer process.

1. Storage layer
2. Processing layer
3. Management layer

The storage layer consists of data in a collection. The processing layer operates on the storage layer. All the aggregations, indexing, and query executions happen in this layer. Finally, the management layer which is a high-level job, it consists of a
4.2 MongoDB Core Process

To connect and interact with the data in the MongoDB database, the core processes play an important role. The core processes are discussed in this section. In this thesis, the core processes are used to import data and interact with the data using *mongo shell*.

*mongoreimport* tool is used for importing data from CSV file to MongoDB. Before migration the server must be started, *mongod* is the used to start the server. We cannot perform any database operations without starting the server. *mongo* is used to interact with the data in the MongoDB database using *mongo shell*. The core process of MongoDB is an important part to interact with the database.

The core process of MongoDB involves mongoreimport and mongorexport tools, the GridFS (file storage) tools, and diagnostic tools.

The main components of the core process are described below:

1. *mongod*: mongod command is one of the main components of the core processes. The mongod command starts the server, manages the requests and also support in data format management. It also provides core options such as version, configuration file, verbose, port number, and many more.

---

1[https://docs.mongodb.com/manual/reference/program/](https://docs.mongodb.com/manual/reference/program/)
2. *mongo*: mongo command provides an interactive javascript shell to communicate with the server. With mongo operations, the developers and administrators can manage the data, run queries, and get the report for their business.

3. *mongoimport*: mongoimport tool is used to import data from different file formats such as JSON, CSV, TSV (tab-delimited file).

### 4.3 Tools Selection

In this section, we discuss the tools required for the implementation of MongoDB queries, building interactive web applications and its functionalities.

#### 4.3.1 NoSQLBooster

There are numerous MongoDB management tools that are available in the data world. These tools help to interact with the data in the database with the smart GUI (Graphical User Interface). GUI makes productivity and management tasks easy for developers, and administrators.

In this thesis, NoSQLBooster is selected for query evaluation in MongoDB database Figure 4.2 on the facing page.

<table>
<thead>
<tr>
<th>NoSQLBooster Version</th>
<th>4.7.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB version</td>
<td>4.0</td>
</tr>
<tr>
<td>Machine type</td>
<td>Windows 64-bit</td>
</tr>
<tr>
<td>Size</td>
<td>36.8 MB</td>
</tr>
<tr>
<td>Downloaded on</td>
<td>13.08.2018</td>
</tr>
</tbody>
</table>

Table 4.1: MongoDB version and NoSQLBooster Specifications

NoSQLBooster offers built-in language services, vast number of built-in snippets that helps in writing queries. Whenever the query script writing is started it always pop-up the suggestions as you type a query.

Features of using NoSQLBooster:

1. The query functions can be performed in SQL for MongoDB database. This includes JOINS, expressions, aggregations, and function.

2. NoSQL booster as a unique feature known as chaining syntax. It is possible to retrieve the data with the combination of SQL and MongoDB query language. It works even with the aggregation framework.

3. It explains the query execution plan in a hierarchical structure which is easy to read and understand.

---

4https://www.nosqlbooster.com/
4.3. Tools Selection

Figure 4.2: NoSQLBooster Graphical User Interface

4. It provides a visual query builder feature. Many other features are described in NoSQLBooster website. |

Figure 4.2 shows the user interface of NoSQLBooster. It has multiple types of data retrieval process such as visual query builder, SQL or MongoDB language, querying through the programming languages namely R programming, Python, C++, Java and many more. It also shows the time taken to retrieve the data and number of documents count that is retrieved from the database. The data can be viewed as tables, tree view, and JSON format.

NoSQLBooster helps to develop queries using the query interface. The output of the query gives the information stated in Figure 4.2. The execution time, the number of documents retrieved helps to investigate the query performance of the database and compare it with the PostgreSQL database.

4.3.2 R framework SHINY

The prototype is developed using the R framework SHINY. The technical details and the process of developing of SHINY application for MongoDB query implementation are discussed in this section. R is a programming language built for graphics and statistical computing. Shiny is an R framework. It helps to develop a user-friendly an interactive web application. Shiny is an in-built application in R studio. It is the R package and the SHINY application built is developed using R programming. There are two major parts of the SHINY framework. Firstly, frontend

5URL: https://www.nosqlbooster.com/
6URL: https://www.r-project.org/about.html
7URL: http://www.rstudio.com/shiny/
development providing a user interface. It enables the user to interact with the data even without any programming knowledge. The user can retrieve the data just by text search and some simple search clicks on dialogue boxes, and tabs. Secondly, backend development where the queries are implemented in R programming.

In Figure 4.3, the shiny interface is shown. It has four windows. In window 1 is used as a code editor which is written in R programming (ui.R* & server.R*). Window 2 is a console where it runs execution of the developed program. Window 3 is a global environment where it shows the details of the history of the commands used, connection with the MongoDB database and many more functions. Window 4 contains the information regarding the various R packages, libraries for connecting or interacting with the data.

In this thesis for developing an interactive web application for MongoDB, mongolite and jsonite libraries are used. The libraries are available by default in Shiny R package list.

To work with the Mongodb data, initially we need to connect to the MongoDB database. In order to communicate, first the server must be started using mongod command which is already described in Section 4.2 on page 41.

Consider the query which is used for developing web application Listing 4.1 and Listing 4.2. The steps involved in developing an application is discussed below.

---

8URL: http://www.rstudio.com/shiny/
4.3. Tools Selection

For developing a web application using SHINY, mongolite, jsonlite packages must be installed. The packages help to connect the shiny interface with MongoDB and interact with the data.

In Listing 4.1, the user interface is defined. The page is titled as MongoDB text search Data. In the side-bar panel, the text field for text search and document selection type for the given text search is developed. The output button is used to display the output after selecting the query. In the main panel, two tabs are provided for retrieving the data related to institutions or expert profile. The output of the query is displayed in the main panel as a table.

```r
library(shiny)
library(mongolite)
library(jsonlite)
limit <- 10L

# Define UI for application for mongodb text search
ui <- fluidPage(
  # Application title
titlePanel("Mongodb text search Data"),
sidebarLayout(
    sidebarPanel(
      textInput("query_id", "Title text"),
      selectInput("doc_id", "document", choices = c("PATENT", "SCIENCE")),
      actionButton("act", "output")
    ),
    # Show the mongodb text search output in the main panel
    mainPanel(
      tabsetPanel(
        tabPanel("INSTITUTE", dataTableOutput('table1')),
        tabPanel("EXPERT", dataTableOutput('table2'))
      )
    )
)
```

Listing 4.1: Code for developing a Shiny R application user interface

# defining server side function
server <- function(input, output) {
  # connecting to MongoDB server
  mdb <- mongo(collection = "doc", db = "datasample", url = "mongodb://localhost:27017/?socketTimeoutMS=1200000")
  # Reactivity
  INSTITUTION <- eventReactive(input$act, {
    #Text indexing
    mdb$index(toJSON(list("title" = "text"), auto_unbox = TRUE))
  })
}
```
```r
# Applying query
q <- paste0("["$match" : {
"$text" : {
"$search" : "",
input$query_id , "" } }
 },
{"$match" : {"doc_type" : "",input$doc_id , "" } },
{"$match" : {"player_type" : "INSTITUTION"} }
),
{"$group":
{ "_id":
{ "doc_type" : "$doc_type"},
"number_records": { "$sum": 1 },
"player_name" : {"$first": "$player_name"},
"title" : {"$first": "$title"},
"player_type" : {"$first": "$player_type"},
"country_code" : {"$first": "$country_code"}
}),
{"$sort": {"number_records" : -1}},
{"$limit" : 10
}"
)

jsonlite::validate(q)
query <- mdb$aggregate(
q, '{"allowDiskUse": true}')

# Reactivity
EXPERT <- eventReactive(input$act,{

# Applying query
q <- paste0("["$match" : {
"$text" : {
"$search" : "",
input$query_id , "" } }
 },
{"$match" : {"doc_type" : "",input$doc_id , "" } },
{"$match" : {"player_type" : "EXPERT"} }
),
{"$group":
{ "_id":
{ "doc_type" : "$doc_type","title" ,"$title", "player_name" : "$player_name", "player_type" : "EXPERT", "country_code" : "$country_code" },
"number_records": { "$sum": 1 }
}
```
At backend Listing 4.2 on page 45, input and output the query operations are developed. Initially, The SHINY is connected to MongoDB database. Then the reactive function is used. It is a reactive command that makes the application responsive to the call by the user using the user interface. For the text search from the database, text indexing is created on the title field. The aggregation query is developed. The aggregation process is explained in Section 3.4.1.1 on page 35. finally, the output of the data is displayed as a table in an application Figure 4.4 on the following page.

For instance, if the user needs information related to a particular field of interest. The query scans the related documents in a collection and executes results. The process of query implementation by a user is displayed in Figure 4.5 on the next page.

The user enters the text query in the text field and the document type is selected. The number of organizations or the experts for a selected document type is executed and displayed to the user. With the web application, the user can fetch the information according to the requirements. For instance, it the user need number of scientific publications for every organization for a particular field of interest. The user selects the scientific documents of the organization by giving the text in a text search field. The output of the queries displays, the number of scientific publications for every organization for a given text.
4. Design and Implementation

Figure 4.4: Output table in Shiny R web application

Figure 4.5: Query execution process in SHINY.
4.4 MongoDB Query Optimization

Performance optimization affects the execution speed when data reaches its peak limits or because of large complex queries. In order to improve the performance, it is important to follow proper aggregation pipeline optimization technique.

Creating an index on a single field in a collection reduces the scanning time and increases the data retrieval process faster (Figure 4.6). After indexing, aggregation pipeline optimization is performed. Each stage passing through the pipeline. Proper steps in aggregation reduce the execution time. Projecting the required fields instead of whole data in collection increases the speed of pipeline operation.

When the complex aggregation queries are applied to the database. The database with large queries lowers the speed of execution. To improve the performance the proper aggregation pipeline optimization is required.

The query is used to explain the optimization technique Listing 4.3 on the next page. The query projects the number of PATENT documents fetches the list of organizations, experts and country code that matches the query for a given text. The execution process in each stage is shown.

The execution plan in Listing 4.3 on the following page shows the procedure of how each stage passes through the pipeline. The pipeline first matches the field where doc_type is PATENT and then match the text search which is given in a text indexed field. The document is filtered out and the output after passing through the matching stage it enters to the group operator. With the reference of the output of the match operator, the documents are grouped according to player_name and country_code and fetch the number of records. Then, it projects the fields described in the group stage. Sorts the number of records in descending order and finishes the pipeline after limiting the documents.
#Giving text input as "VIDEO" and projecting the required fields

db.doc.aggregate([{
  "$match": { "$text": { "$search": "VIDEO " } },
  "$match": { "doc_type": "PATENT" } }
],

{ "$group": { "_id": { "player_name": "$player_name", "country_code": "$country_code" }, "number_records": { "$sum": 1 } }
},

{ "$project": { "player_name": "$player_name", "country_code": "", "number_records": "$number_records", "_id": 1}},

{ "$sort": { "number_records": -1 } },

{ "$limit": 500 }], { explain: true});

#Pipeline explanation of the above query

{ "stages": [
  { "$cursor": {
    "query": {
      "$and": [
        { "$text": {
          "$search": "VIDEO "
        },
        { "doc_type": "PATENT" }
      }
    }
  },
  "fields": {
    "country_code": 1,
    "player_name": 1,
    "_id": 0
  },
  "queryPlanner": {
    "plannerVersion": 1,
    "namespace": "sample.doc",
    "indexFilterSet": false,
    "parsedQuery": {
      "$and": [
        { "doc_type": { "$eq": "PATENT" }
      }
    },
    { "$text": {
      "$search": "VIDEO ",
      "$language": "english",
      "$caseSensitive": false,
    }
  },
4.4. MongoDB Query Optimization

"$diacriticSensitive" : false

"
"winningPlan" : {
  "stage" : "FETCH",
  "filter" : {
    "doc_type" : {
      "$eq" : "PATENT"
    }
  }
},
"inputStage" : {
  "stage" : "TEXT",
  "indexPrefix" : {
    "indexName" : "title",
    "parsedTextQuery" : {
      "terms" : [
        "video"
      ],
      "negatedTerms" : [],
      "phrases" : [],
      "negatedPhrases" : []
    },
    "textIndexVersion" : 3,
    "inputStage" : {
      "stage" : "TEXT
MATCH",
      "inputStage" : {
        "stage" : "FETCH",
        "inputStage" : {
          "stage" : "OR",
          "inputStage" : {
            "stage" : "IXSCAN",
            "keyPattern" : {
              "fts" : "text",
              "fts x" : 1
            },
            "indexName" : "title",
            "isMultiKey" : true,
            "isUnique" : false,
            "isSparse" : false,
            "isPartial" : false,
            "indexVersion" : 2,
            "direction" : "backward",
            "indexBounds" : {
              ""}
            "rejectedPlans" : []
          }
        }
      }
    }
  }
}
4. Design and Implementation

The match command should always be on the first staging as it filters the PATENT document by scanning the whole collection. If the search involves in text search, it should be queried in the first line. After the pipeline passes through matching stage grouping the number of document and projecting only the required field can affect the execution time. That leads to an increase in performance.

In this chapter, the design of process flow, MongoDB core process, and implementation process are described. Furthermore, we developed an easy implementation approach to query data by using different tools. The tools used in the implementation are mongo shell (for data importing), NoSQLBooster (for evaluating the performance), and R framework SHINY (developing a prototype) made data retrieval easy and the web application helps to interact with the data even without any programming knowledge. The interactive web application is best suitable for people like business managers, clients, R&D manager's.
5. Evaluation

Different implementation tools and procedures were discussed in the previous chapter. This chapter focuses entirely on the evaluation, it includes results from data migration, query performance in terms of query execution speed and their comparison at the end. To do so, initially, we migrate the data from the PostgreSQL database to MongoDB database. Then, we compare the query performance of MongoDB and PostgreSQL database.

5.1 Evaluation Setup

In order to properly judge the query performance of MongoDB implementation, we need to compare it to PostgreSQL database. Since the primary focus is on query performance, our evaluation is based on query execution time. The results provides the information for decision making in selecting the database for the given data.

5.1.1 Machine Used

For implementing the data migration, writing the queries for data retrieval and designing the web application for the MongoDB database, the local machine is used. The detail description is shown in Table 5.1.

To compare the results, the PostgreSQL database is also implemented with the same data that is used in MongoDB database. The PostgreSQL database implementation is done in the same machine.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Windows version 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Core i5-5200U</td>
</tr>
<tr>
<td>CPU</td>
<td>2.20GHz</td>
</tr>
<tr>
<td>System type</td>
<td>64 bit</td>
</tr>
<tr>
<td>RAM</td>
<td>8 GB</td>
</tr>
</tbody>
</table>

Table 5.1: Used Machine Specifications
5.1.2 Data Characteristics

We use the data from the Mapegy Gmbh company database to compare performance between MongoDB and PostgreSQL systems. We migrated multiple datasets with different size to examine the query performance on different sizes. It is difficult to test the performance with a large amount of data on the local machine. So, we migrated the same data set but with the small data sets. Also, we migrated another dataset that is extracted from the single table that is entity_dogs table see Figure 3.2 from PostgreSQL database. The tables with single table and merged tables (entity_dogs, link_player_dogs, entity_players) are migrated see Figure 3.2. After the migration, we compare the query performance for a single table and the embedded table in PostgreSQL database and the MongoDB database.

In Table 5.2, we provide an overview of tables. In the experiments, we start with the migration of a large table that contains the information of patents and scientific publications for the list of organisations and experts for various countries. The data that contains required information is migrated as a single table (entity_dogs) and from multiple tables (entity_dogs, link_player_dogs, entity_players). For investigating the query performance, the data is migrated with different sizes to the database.

<table>
<thead>
<tr>
<th>No. of records</th>
<th>MongoDB Embedded table</th>
<th>MongoDB Single table</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>13 MB</td>
<td>5 MB</td>
</tr>
<tr>
<td>50000</td>
<td>210 MB</td>
<td>53.3 MB</td>
</tr>
<tr>
<td>100000</td>
<td>420 MB</td>
<td>110 MB</td>
</tr>
<tr>
<td>1000000</td>
<td>3.2 GB</td>
<td>1.40 GB</td>
</tr>
<tr>
<td>23734000</td>
<td>106.2 GB</td>
<td>84.6 GB</td>
</tr>
</tbody>
</table>

Table 5.2: Statistics of Datasets

To examine the query capabilities of MongoDB, we perform queries on single collection and embedded collection. The data is collected from PostgreSQL database.

5.2 Experiments

In this section, we discuss the experiments implemented on MongoDB and its results are shown. Apart from our predefined queries, we perform several other experiments to compare the performance of MongoDB and PostgreSQL with simple and complex queries.

5.2.1 Data Migration

The data is extracted from the PostgreSQL database into a CSV file. The time taken for extracting files from PostgreSQL data into CSV is shown in the Table 5.3. From the result, we observe that the time taken for collecting the data is greater than the time taken in importing the data into MongoDB database. The mongoimport tool is performed efficiently in importing data to the MongoDB database. The mongoimport tool provides an efficient results for the datasets with different sizes.
The data is migrated using the MongoDB shell interface. The migration is performed using following command  

```plaintext
Listing 5.1: MongoDB Data Migration

#syntax for data migration using mongoimport in mongoshell
@<mongoshell>: mongoimport --db <databaseName> --collection <collectionName> --type CSV --file <filepath> --headerline

#Data migration using mongoimport in mongo shell
C:\Programme\mongodb\Server\4.0\bin>mongoimport --db datadocuments --collection data --type csv --file E:mongodb\data-55124365.csv --headerline
```

We run the experiments with different tables that vary in size. Initially, we migrated a large embedded table with 23.7 million records. Due to the large size, it is difficult to run the queries efficiently on local machine. So in order to investigate the query performance, another table with the same columns but with fewer records is migrated. Finally, a single table (entity\_docs) from a PostgreSQL database with 100000 records is migrated. This single table helps to investigate the execution speed without joins.

To compare query performance of both the databases (PostgreSQL and MongoDB), the data extracted from the PostgreSQL tables are initially imported to PostgreSQL database on a local machine in a normalized form. The data is migrated using `COPY` command, to import the data to PostgreSQL database from CSV (comma separated value) file. Before importing the data to the PostgreSQL database, the table is created with `CREATE TABLE` command. The column names and its data type is listed when creating a table. After creating the table the data is imported into the table using the command shown in Listing 5.2. The same procedure is followed to import other tables that are used in the thesis (entity\_players, link\_players\_docs).

```plaintext
Listing 5.2: Importing data into PostgreSQL database

# CSV data importing syntax
COPY <databaseName\.tableName> FROM <FilePath> DELIMITER <'type of delimiter'> CSV HEADER;

# Data imported using SQL shell for a single table.
COPY datawarehouse.entity\_docs FROM 'C:\data\db\mongocsv\entitydocs.csv' "\", "\", HEADER;
```

According to the tasks defined in Section 1.1 on page 2, we need to use text search to fetch the relevant information from the database. So, the tables are indexed using GIN to provide text search indexing. The detail explanation of indexing is given in Section 2.2.3 on page 9. The GIN indexing is implemented on a text field `title` see Listing 5.3 on the next page.
The time taken to import data to the MonogoDB and the postgresQL database is shown in Table 5.3.

<table>
<thead>
<tr>
<th>No. of records</th>
<th>Type of table</th>
<th>PostgreSQL (CSV)</th>
<th>MongoDB (JSON)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Data extraction</td>
<td>Data importing</td>
</tr>
<tr>
<td>23.7 million</td>
<td>embedded table</td>
<td>13200.245 Sec.</td>
<td>9000.192 Sec.</td>
</tr>
<tr>
<td>1 million</td>
<td>embedded table</td>
<td>550.235 Sec.</td>
<td>37.149 Sec.</td>
</tr>
<tr>
<td>1 million</td>
<td>single table</td>
<td>280.685 Sec.</td>
<td>18.281 Sec.</td>
</tr>
<tr>
<td>100 thousand</td>
<td>embedded table</td>
<td>49.568 Sec.</td>
<td>19.214 Sec.</td>
</tr>
<tr>
<td>100 thousand</td>
<td>single table</td>
<td>19.817 Sec.</td>
<td>10.214 Sec.</td>
</tr>
<tr>
<td>50 thousand</td>
<td>embedded table</td>
<td>29.648 Sec.</td>
<td>14.214 Sec.</td>
</tr>
<tr>
<td>50 thousand</td>
<td>single table</td>
<td>24.568 Sec.</td>
<td>12.214 Sec.</td>
</tr>
<tr>
<td>10 thousand</td>
<td>embedded table</td>
<td>15.687 Sec.</td>
<td>9.214 Sec.</td>
</tr>
<tr>
<td>10 thousand</td>
<td>single table</td>
<td>12.549 Sec.</td>
<td>8.214 Sec.</td>
</tr>
</tbody>
</table>

Table 5.3: Data Migration results

5.2.2 Experiment Queries

After importing data into MongoDB database, the queries are executed for the tasks described in Section 1.1. The tasks are selected because, the tasks contain simple and complex queries that helps in evaluating the query performance of MongoDB database. The queries are developed for the tasks and evaluate the query performance of MonogoDB. All the experiments that are shown in this section are also implemented in the R framework SHINY. The listing of the experiments shown in this section is posted in the Chapter A. For evaluating the query performance several other queries are implemented and the details are listed in Table 5.4. From the Table 5.4 the queries 1, 3, 5, and 6 are the queries that are similar to the tasks that are defined in Section 1.1 but the queries 3, 5, 6 are used for different text search.

Retrieve information of all patents:

The query Listing 5.4 is a simple query that is performed using `db.collection.find()` function. In the query the information related to patent documents are retrieved. Listing 5.4 is the implementation of query that finds all the patents with different fields projected. The projected field are `player_id`, `doc_type`, `doc_id`, `title`, `country_code`. The query retrieves 68125 patents from 100000 documents.

```
# simple query finding PATENT documents in doc_type field.
db.mergeddocs.find({
    doc_type: "PATENT"
}).projection {
```
5.2. Experiments

Listing 5.4: MongoDB query execution

The task that are defined in Section 1.1 involves in text search for retrieving the data from MongoDB database. In Section 3.4.1.2, the text indexing is performed Listing 3.9 on every collection that is used in the thesis. Here the index on the _id field is present by default for every collection in a table. In the aggregation pipeline, we use text search (with the focus on document title) for retrieving the relevant data of patents and scientific publications of all the organization and experts. So, text index is executed on single field title that is required for a text search.

**Retrieve all the patents and scientific publications related whose title contains the word 'Complex':**

After creating a text index on title field, the query is executed using text search on the MongoDB database. The queries retrieves all the patents and scientific publications related to the title that contains word 'complex'. The output of the query returns all the documents related to the word complex (736 documents). The output of the given query projects six columns. The projected output columns are shown in Listing 5.5.
Retrieve all the patents related to VIDEO and list all the organizations and experts:

The query in Listing 5.6 is implemented in aggregation pipeline. The query retrieves all the documents (230 documents) whose title contains the word VIDEO. The output result gives all the patents and scientific publications of the organization and the expert from various countries.

Listing 5.5: Aggregation query execution
5.2. Experiments

Listing 5.6: MongoDB query

Retrieve all the organizations and experts related to the SERVICE and ranking them by number of patents:

In the query mentioned Listing 5.7, the aggregation pipeline undergoes different stages. The query ranks the number of patents and scientific publications in descending order for all the organizations and experts. The output projects all the patents and scientific publications of the organization and the expert from various countries whose title contains the word ‘SERVICE’.

```
#Text search input on title column.
# player_type can be selected as EXPERTS or INSTITUTIONS and rank them by number of PATENTS and number of SCIENCE.

db.mergeddocs.aggregate([{
    "$match": {
        "$text": {
            "$search": "SERVICE"
        }
    },
    {
        "$match": {
            "doc_type": "SCIENCE"
        }
    },
    {
        "$match": {
            "doc_type": "PATENT"
        }
    },
    {
        "$player_type": SCIENCE to get list of organization and rank them by number of patents and scientific publications.
        "$group": {
            "_id": {
                "player_type": "player_type",
                "country_code": "$country_code"
            },
            "number_records": { "$sum": 1 }
        },
    },
    {
        "$project": {
            "player_type": "player_type",
            "player_name": ","
        },
        "$number_records": 67
    },
    {
        "$sort": { "number_records": -1 }
    },
    { "$limit": 100000 }
}),
#output of the given query
{
    "_id": {
        "player_type": "EXPERT",
        "player_name": "Eidloth, Rainer",
        "country_code": "DE",
        "doc_type": "PATENT",
        "doc_source": "PATSTAT",
        "doc_source_id": 339191463
    },
    "number_records": 67
},
```
5.3 Comparison between PostgreSQL and MongoDB

For performance evaluation, the query performance is executed on PostgreSQL and MongoDB database. It includes different tables with 10,000, 50,000, 100,000 and 1,000,000 record entries in each database. In PostgreSQL, queries executed in standard SQL language. In MongoDB, queries executed in MongoDB language. For simple queries MongoDB `find()` is used and for complex queries aggregation pipeline is used. Below Table 5.4 shows the queries performed and the projected fields with the code listing references for the query. To get reliable results, the query execution time is calculated by the average over 10 runs. The query average time was calculated with the same procedure on PostgreSQL and MongoDB database. The data in the database are used only for the thesis, that means no additional data operations are performed by any means. The results of the above queries are shown in the Table 5.5 Table 5.6 Table 5.7.

In Figure 5.1 Figure 5.2 Figure 5.3 and Figure 5.4 the query response time for the queries listed in Table 5.4 is compared for PostgreSQL and MongoDB databases. The query number which is listed in Table 5.4 is shown on x-axis. The query response time (in seconds) is measured on y-axis. In Figure 5.1 Figure 5.2 Figure 5.3 and Figure 5.4, the MongoDB and PostgreSQL is denoted with different colour for easy understanding of query comparison between the databases.

For queries 1, 2, and 3, the queries executed on the single table (collection) and retrieves the data almost the same time with slight advantage of PostgreSQL in all the datasets (Figure 5.1 Figure 5.2 Figure 5.3 and Figure 5.4). Query 1 always take longer time than query 2 and 3 because query 1 retrieves all the data from the table (collection) by scanning every row in the table. Query 2 is faster than query 1 because it scans scientific publications from the database and retrieves the data related to scientific publications. In query 3, the query performed text search on the single table retrieves the data faster than the queries 1 and 2. This is because the query only scans the data where the given text is present. This reduces the scanning time resulting in faster query execution.

| "{_id" : { |
| "player_type" : "EXPERT", |
| "player_name" : "Artault, Alexandre", |
| "country_code" : "FR", |
| "doc_type" : "PATENT", |
| "doc_source" : "PATSTAT", |
| "doc_source_id" : 334075617 |
| } |

Listing 5.7: MongoDB Ranking number of documents
### Table 5.4: Queries for performance comparison from PostgreSQL and MongoDB

<table>
<thead>
<tr>
<th>Number</th>
<th>Queries</th>
<th>No. of projected rows</th>
<th>SQL code</th>
<th>MongoDB code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Get all data from single tables</td>
<td>19</td>
<td>Listing A.5 on page 74</td>
<td>Listing A.13 on page 77</td>
</tr>
<tr>
<td>2</td>
<td>Get all the data from single table related to scientific publications</td>
<td>19</td>
<td>Listing A.6 on page 75</td>
<td>Listing A.14 on page 77</td>
</tr>
<tr>
<td>3</td>
<td>Text search in a single table</td>
<td>19</td>
<td>Listing A.7 on page 75</td>
<td>Listing A.15 on page 77</td>
</tr>
<tr>
<td>4</td>
<td>Get all the organizations and experts where scientific publication is selected</td>
<td>59</td>
<td>Listing A.8 on page 75</td>
<td>Listing A.16 on page 77</td>
</tr>
<tr>
<td>5</td>
<td>Get a list of organizations for Patents and scientific publications related to word 'SERVICE' from title field</td>
<td>6</td>
<td>Listing A.9 on page 75</td>
<td>Listing A.17 on page 77</td>
</tr>
<tr>
<td>6</td>
<td>Retrieve all organization related to 'SERVICE' and ranking them by number of patents, and number of scientific publication</td>
<td>6</td>
<td>Listing A.10 on page 76</td>
<td>Listing A.18 on page 78</td>
</tr>
<tr>
<td>7</td>
<td>Get all the players where the type of connection is INVENTOR</td>
<td>59</td>
<td>Listing A.11 on page 76</td>
<td>Listing A.19 on page 78</td>
</tr>
<tr>
<td>8</td>
<td>Retrieve all startup company related to the word 'behaviour' from title for an organization to find the total number of Scientific publications for a type of organization</td>
<td>7</td>
<td>Listing A.12 on page 76</td>
<td>Listing A.20 on page 79</td>
</tr>
</tbody>
</table>
5. Evaluation

Figure 5.1: Performance comparison on 10,000 records

Figure 5.2: Performance comparison on 50,000 records
5.3. Comparison between PostgreSQL and MongoDB

Figure 5.3: Performance comparison on 100,000 records

Figure 5.4: Performance comparison on 1,000,000 records
For query 4 and 7, the datasets involve JOIN operation to retrieve all the patents and scientific publications for all the organizations and experts in the PostgreSQL database. In the MongoDB database, the data is embedded into a single embedded collection and involves no joins. All the data is retrieved faster from the embedded collection because the collection does not involve in join operations. The execution speed of the MongoDB is up to 50% more than PostgreSQL database.

For queries 5 and 6, the query performs faster with great execution speed on both the database because of text indexing. Indexing limits the number of document search. The query scans only the documents related to the given text. For the datasets 10,000 and 50,000 rows, the difference between the performance of the database is less. In Figure 5.3, the data retrieval in MongoDB is up to 50% faster than PostgreSQL database. For Figure 5.4, query 5 is performance is slightly different whereas for query 6 there is a huge difference in the query performance between the databases. This is because the queries are executed on the local machine. For such huge dataset, the behavior of the system is not significant. However, from the datasets, it is clearly evident that the MongoDB database is faster than PostgreSQL database.

For query 8, MongoDB has a better performance compared to the PostgreSQL database. The queries require to join operations and scanning all the tables degrades the query performance. In MongoDB aggregation pipeline, the query is executed in different stages. Each stage output is the response to the next stage. Proper planning in developing an aggregation pipeline provides efficient execution result. This way the complex query output is executed faster.

From these, it is clearly evident that MongoDB is faster than PostgreSQL database for the given queries.

<table>
<thead>
<tr>
<th>PostgreSQL Query Number</th>
<th>10000</th>
<th>50000</th>
<th>100000</th>
<th>1000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.864</td>
<td>2.994</td>
<td>3.891</td>
<td>11.249</td>
</tr>
<tr>
<td>2</td>
<td>1.142</td>
<td>2.137</td>
<td>3.917</td>
<td>9.392</td>
</tr>
<tr>
<td>3</td>
<td>0.533</td>
<td>0.804</td>
<td>1.156</td>
<td>5.632</td>
</tr>
<tr>
<td>4</td>
<td>4.677</td>
<td>9.123</td>
<td>11.213</td>
<td>23.192</td>
</tr>
<tr>
<td>5</td>
<td>0.452</td>
<td>0.522</td>
<td>1.231</td>
<td>4.961</td>
</tr>
<tr>
<td>6</td>
<td>0.562</td>
<td>0.458</td>
<td>0.981</td>
<td>3.798</td>
</tr>
<tr>
<td>8</td>
<td>0.283</td>
<td>0.543</td>
<td>1.124</td>
<td>4.191</td>
</tr>
</tbody>
</table>

Table 5.5: PostgreSQL Query results in seconds

For each dataset, the number of rows returned for each query in PostgreSQL and MongoDB database is shown in Table 5.6. Since both the databases support stemming, the output of the query should return the same number of rows. The output rows count helps to recognize the datasets used in PostgreSQL and MongoDB database are same.
5.4. Discussion

The decision in selecting the NoSQL database depends on the requirement. For instance, if the company needs to manage the relationships between the huge datasets the Graph databases are a better fit. For example, cooperation networks between the organization or the author of the scientific publication.

In this thesis, we used MongoDB database. One of the main advantages of using the MongoDB database is flexible schema. The data model is easy to maintain and modify in the ever-changing environment.

### 5.3.1 Impact of the Size of Datasets

The databases are implemented on the local machine which has a limited amount of RAM (8GB) and permanent storage (1 TB HDD). For the datasets over 100,000 and 1,000,000 records, the machine cannot perform accurately. For the large dataset (1,000,000 records), due to lack of RAM, the execution speed of the databases decreases.

### 5.4 Discussion

The decision in selecting the NoSQL database depends on the requirement. For instance, if the company needs to manage the relationships between the huge datasets the Graph databases are a better fit. For example, cooperation networks between the organization or the author of the scientific publication.

In this thesis, we used MongoDB database. One of the main advantages of using the MongoDB database is flexible schema. The data model is easy to maintain and modify in the ever-changing environment.
From the Section 5.3, the query performance of the database is evaluated. On the single table, the query performance is almost the same between PostgreSQL and MongoDB database. For the embedded collection (multiple tables in PostgreSQL database), MongoDB shows a clear dominance for the given datasets Figure 5.1 Figure 5.2 Figure 5.3 and Figure 5.4. However, the results may vary for the data that contains several millions of records.

To work with several million records, MongoDB provides high scalability, the data is shared over multiple machines and facilitates working with the large datasets \footnote{https://www.mongodb.com/compare/mongodb-postgresql}. In the case of PostgreSQL, there is no native sharding technique for distributing the data across nodes in a cluster\footnote{https://www.mongodb.com/compare/mongodb-postgresql}. Using other horizontal scalability techniques such as manual sharding, using a sharding framework and so on can lead to loss of important relational abilities, data integrity, ACID properties \footnote{https://www.mongodb.com/compare/mongodb-postgresql}.
6. Related Work

“Study the past if you would define the future.”

-Confucius

In this chapter, we discuss the related work that is similar in implementing MongoDB database operations. Their work also describes the performance of the MongoDB database.

Tim Mohring investigates the Tinnitus database project [Moh16]. The Tinnitus database contains information like patient symptoms, ideal treatment method and so on. The Tinnitus database is based on MySQL database which is a relational database and has some disadvantages that can create unacceptable errors in case of mispractice. The author examines different NoSQL databases to overcome problems that occur in MySQL database and concluded that the performance of the document-oriented database is high when the data is retrieved from multiple tables that require joins in the relational database. He used MongoDB for practical implementation and evaluated the results (in terms of performance and schema validation). Due to the flexible schema and easy query capabilities and lack of joins, the query performance of MongoDB is higher than the relational database [Moh16]. In his work, he concluded that the MongoDB database is superior in queries that involve JOIN operations in MySQL database. Similar to this thesis, we also implemented various complex queries in the MongoDB database and evaluate the resultant performance with company’s PostgreSQL database query performance. The query performance of MongoDB is higher than the PostgreSQL database.

Ningthoujam et al [NCP+14] designed a MongoDB data model for Ethnomedicinal plant data. In the article, the authors described the data modeling patterns of MongoDB. There are two options for designing a data model in MongoDB. The two options were embedding or referencing through a unique identifier. For the embedding collection, the related data is merged into a single collection. The second option was connecting the collections using a unique identifier in a collection. They choose to use both the options depending upon the data representation choice. The data sets are imported through Mongoimport tool and tested its performance in terms of scalability, flexibility, extensibility, query execution speed. The authors conclude that the ultimate decisions on MongoDB data model are based on the access
pattern. The MongoDB database queries depend on data usage pattern. Indexing limits the number of scanning documents. The use of indexes in developing a query increases the query execution speed. This results in providing high-performance [NCP+14]. In our approach, we implemented a similar idea of embedding the tables into a single collection and used aggregation pipeline with the use of text indexing. The text indexing fetches the related document that reduces the scanning time. This results in high query performance.

Parker et al [PPV13] compared NoSQL databases and SQL databases. The authors implemented different NoSQL databases. They compare key-value pairs of databases on different operations. The operations are storage, read, write, and deletes. The operations are executed on Microsoft SQL Server Express, MongoDB, CouchDB, RavenDB, Cassandra, Hypertable, and Couchbase. They evaluated results for all the operations for 100000 records. They performed data retrieval query for 10,50,100,1000,10000, and 100000 records. They compared the query response time on all the databases and concluded that Couchbase and MongoDB are the fastest in retrieving data for the given datasets. In our work, we migrated datasets of 23.7 million, 1 million, and 100 thousand records and compared the query performance with the PostgreSQL database. The MongoDB database is superior in fetching the data from the large datasets compared to PostgreSQL.

Chickerur et al [CGK15] used the airline’s database with 1050000 records for comparing a relational database with MongoDB database. Initially, the authors migrated the data from the relational database to MongoDB database. The queries are executed on the MongoDB database and compared its performance with the MySQL database. They implemented various query operations and concluded that the MongoDB provides efficient performance for a big data application compared to the MySQL database. Similarly, in our thesis, we migrated the data from the PostgreSQL database to MongoDB database. After the migration, the various queries are executed. MongoDB database provides an efficient query performance for the big data sets that are extracted from the PostgreSQL database.
7. Conclusion and Future Work

“In three words I can sum up everything I’ve learned about data: it goes on.”
-Robert Frost

7.1 Summary

In this thesis, we compared the performance of the MongoDB database with the PostgreSQL database. To evaluate the performance, the data is migrated to MongoDB database.

In this work, we provide an overview of existing PostgreSQL database where the structure of the PostgreSQL database is shown and addresses the issues that arose in using PostgreSQL database. The PostgreSQL contain multiple tables. The tables contain the information of patents and scientific publications of the organizations and experts from different countries that are taken different data sources.

To retrieve the relevant data from the tables, JOIN operation is used. The usage of multiple JOINS especially in case of complex queries lowers the query execution speed that results in low performance. To overcome the issues, one of the NoSQL databases is selected.

MongoDB database is selected on the factors that support all the characteristics of PostgreSQL. Initially, the data is migrated from PostgreSQL database to MongoDB. The data model of MongoDB is flexible and modelled in an easy way. For evaluating the performance aggregation pipeline is used. Furthermore, the operations involved in aggregation pipeline is discussed.

For developing the prototype a user friendly interactive web application is developed using R framework SHINY.

For our experiments, we used four different sizes of data sets. We implemented all the queries in the local machine (Windows version 10).

7.2 Conclusion

We examined the query performance of small, and large datasets. The results of the work are practically relevant to the developers who use the database. We checked
the results for each query in PostgreSQL and MongoDB database. We also provide the query optimization technique that helps users to develop aggregation pipeline in a proper way that can reduce the query execution time. The results are provided for different queries, ranging from simple queries to large complex queries in case of single table and multiple tables.

- The queries 1, 2, and 3 are performed on the single table (see Table 5.4), where no join operations are required. The PostgreSQL database query performance for a single table is almost the same as MongoDB database.

- To perform queries 5, 6, 7, and 8 (see Table 5.4), multiple tables are denormalized to create an embedded collection in MongoDB database. In PostgreSQL, for the queries that involves multiple tables takes a long response time. MongoDB shows a clear dominance in case of complex queries involved in JOIN operations.

- In case of complex queries where join operations are performed, the query performance is decreased by upto 50% compared to MongoDB database.

In our experiment results, it is clear that MongoDB is dominating in the query execution speed for a single collection and also on the embedded collection. This results proves, MongoDB database provides high performance than PostgreSQL database.

Finally, in conclusion, the thesis proves that the use of NoSQL databases (MongoDB) is beneficial especially in the case of large complex queries where JOIN operations involved. However, the different sizes of the database can impact performance.

### 7.3 Future Work

In the MongoDB database in text search, the text indexing supports features like filtering the stop words such as (it, a, an the and many more), scoring and stemming (reduced words like standing as a stand, stood which has the same base). But, MongoDB not yet supports text search based on synonyms. The development of full-text search on a synonym or similar words will be an interesting task in the future.

In MongoDB indexing, if we need to add indexing the first executed indexing must be dropped. This leads to time consumption. It will be interesting if MongoDB overcomes this restriction in the future.

This thesis evaluated the result for the data upto 1 million records. The possible future benchmark would be evaluating the query performance for a huge datasets (greater than 1 million rows).
A. Code Listings

The chapter contains the queries used in SQL and in MongoDB. The code for developing SHINY application is listed followed by SQL and MongoDB query listings.

A.1 R Framework SHINY

```r
#shiny = (ui.R, server.R)
#libraries used for connecting to MongoDB database
library(shiny)
library(mongolite)
library(jsonlite)

# Define UI for application for mongodb text search
ui <- fluidPage(
  # Application title
  titlePanel("Mongodb text search Data"),
  sidebarLayout(
    sidebarPanel(
      textInput("query_id", "Title text", ""),
      selectInput("doc_id", "document", choices = c("PATENT", "SCIENCE"))
    ),
    # Show the mongodb text search output in the main panel
    mainPanel(
      dataTableOutput("mydata")
    )
  )
)

server <- function(input, output) {

  mdt <- mongo(collection = "data", db = "datadocuments", url = "mongodb://localhost:27017")
  titletext <- reactive({
    mdt$index(toJSON(list("title" = "text"), auto_unbox = TRUE))
    q <- paste0("{""$match" : { "$text" : { "$search" : ", "query_id" : ", "doc_type": ", "doc_id" : "" } },
    {"$group" : 
      {"_id" : 
```
Listing A.1: Query covering some field of interest and get a list of relevant documents

```r
#shiny = (ui.R, server.R)
#libraries used for connecting to MongoDB database
library(shiny)
library(mongolite)
library(jsonlite)

# Define UI for application for mongodb text search
ui <- fluidPage(
  # Application title
titlePanel("Mongodb Data"),
sidebarLayout(
    sidebarPanel(
      textInput("title_id", "Title text", ")
    ),

    # Show the mongodb text search output in the main panel
    mainPanel(
      dataTableOutput("mydata")
    )
  )
)

server <- function(input, output) {
  mon <- mongo(collection = "documents", db = "entitydocuments", url = "mongodb://localhost:27017")

  titleSearchResult <- reactive({
    titleSearchResult <- reactive({
      titleSearchResult <- reactive({
        titleSearchResult <- reactive({
          output$mydata <- renderDataTable(
            titleSearchResult()
          )
        })
      })
    })
  }

  #text search output
  mon$find(toJSON(list("$text" = list("$search" = text)), auto_unbox = TRUE))
)

shinyApp(ui = ui, server = server)
```
Listing A.2: Query covering some field of interest and get a list of organizations ranked by number of patents, scientific publications matching the query.

```r
shinyApp(ui = ui, server = server)
```

Listing A.3: Query for an organization and get a list of collaborators, i.e., organizations with common documents; rank them by number of common patents, number of common scientific publications at user interface And at server side shown Listing A.4.

```r
#libraries used for connecting to MongoDB database
library(shiny)
library(mongolite)
library(jsonlite)

# Define UI for application for mongodb text search
ui <- fluidPage(
  titlePanel("Mongodb text search Data"),
  sidebarLayout(
    sidebarPanel(
      textInput("query_id", "Title text", ""),
      selectInput("doc_id", "document", choices = c("PATENT", "SCIENCE")),
      actionButton("act", "output"),
    ),
    mainPanel(
      tabsetPanel(
        tabPanel("INSTITUTE", dataTableOutput('table1')),
        tabPanel("EXPERT", dataTableOutput('table2'))
      )
    )
  )
)
```

`server <- function(input, output) {
  mdt <- mongo(collection = "data", db = "datadocuments", url = "mongodb://localhost:27017")
  INSTITUTION <- eventReactive(input$act, {
    mdt$index(toJSON(list("title" = "text")), auto_unbox = TRUE))
    q <- paste0('[{"$match" : { "$text" : "$search" : ",
      input$query_id, "" } } ],
      "$match" : {"doc_type": ",input$doc_id , "" } },
    "$match" : {"player_type": "INSTITUTION" },
    {"$project" : { "player_name":1, "title" : 1 , "player_type" : 1, "country_code":1}},
    { "$group" :
      { "_id" : { "player_name" : "$player_name" }},
      "number_records" : { "$sum" : 1 },
      "player_name" : {"$first": "$player_name"},
      "player_type" : {"$first": "$player_type"},
      "country_code" : {"$first": "$country_code"}
  )
}`
Listing A.4: Query for an organization and get a list of collaborators, i.e., organizations with common documents; rank them by number of common patents, number of common scientific publications at server side

A.2 SQL and MongoDB Queries

Listing A.5: PostgreSQL query example 1
### A.2. SQL and MongoDB Queries

#### PostgreSQL query example 2

```sql
# Data selection from single table
SELECT *
FROM public.entity_docs
WHERE doc_type in ('SCIENCE')
```

Listing A.6: PostgreSQL query example 2

#### PostgreSQL query example 3

```sql
# Data selection from single table
SELECT *
FROM public.entity_docs
WHERE tsv_title @@ to_tsquery('Motion');
```

Listing A.7: PostgreSQL query example 3

#### PostgreSQL query example 4

```sql
# Data selection from multiple tables
SELECT *
FROM public.link_player_doc x
JOIN public.entity_docs y ON x.doc_id = y.doc_id AND y.doc_type in ('SCIENCE')
JOIN public.entity_player z ON z.player_id = x.player_id
```

Listing A.8: PostgreSQL query example 4

#### PostgreSQL query example 5

```sql
# Data selection from multiple tables
SELECT z.player_id,
       z.player_type,
       z.player_sub_type,
       z.player_name,
       z.country_code,
       z.address
FROM public.link_player_doc x
JOIN public.entity_docs y ON x.doc_id = y.doc_id AND y.doc_type in ('PATENT', 'SCIENCE')
JOIN public.entity_player z ON z.player_id = x.player_id AND player_type = 'INSTITUTION'
WHERE tsv_fulltext @@ to_tsquery('service')
GROUP BY z.player_id
```

Listing A.9: PostgreSQL query example 5
# Data selection from multiple tables

```sql
select
    z.player_id,
    z.player_type,
    z.player_sub_type,
    z.player_name,
    z.country_code,
    z.address,
    count(*) filter (where doc_type = 'SCIENCE') as nb_science,
    count(*) filter (where doc_type = 'patent') as nb_patent
from public.link_player_doc x
join public.entity_docs y on x.doc_id = y.doc_id and y.doc_type in ('PATENT', 'SCIENCE')
join public.entity_player z on z.player_id = x.player_id where x.player_doc_link_type in ('{INVENTOR}')
where tsv_fulltext @@ to_tsquery('Motion')
```

Listing A.10: PostgreSQL query example 6

# Data selection from multiple tables

```sql
select *
from public.link_player_doc x
join public.entity_docs y on x.doc_id = y.doc_id and y.doc_type in ('PATENT', 'SCIENCE')
join public.entity_player z on z.player_id = x.player_id where x.player_doc_link_type in ('{INVENTOR}')
```

Listing A.11: PostgreSQL query example 7

# Data selection from multiple tables

```sql
select
    z.player_sub_type,
    z.player_name,
    z.player_type,
    y.doc_source,
    z.date_inserted,
    y.meta,
    y.country_code
from public.link_player_doc x
join public.entity_docs y on x.doc_id = y.doc_id and y.doc_type in ('SCIENCE')
join public.entity_player z on z.player_id = x.player_id and z.player_sub_type = ('{STARTUP, COMPANY}')
where tsv_fulltext @@ to_tsquery('behavior')
```

Listing A.12: PostgreSQL query example 8
A.2. SQL and MongoDB Queries

```sql
# Data selection from single table
db.documents.find({})
  .projection({})
  .limit(1000000)

Listing A.13: MongoDB query example 1

```

```sql
db.doctable.find({'doc_type': 'SCIENCE'})
  .projection({})
  .limit(1000000)

Listing A.14: MongoDB query example 2

```

```sql
# Data selection from single table
db.documents.aggregate([{
  "$match": {
    "$text": {
      "$search": "Motion"
    }
  }
}]);

Listing A.15: MongoDB query example 3

```

```sql
# Data selection from multiple tables
db.documents.aggregate([{
  "$match": {
    "$text": {
      "$search": "Science"
    }
  }
}]);

Listing A.16: MongoDB query example 4

```

```sql
# Data selection from multiple tables
db.embedded.aggregate([{
  "$match": {
    "$text": {
      "$search": "SERVICE"
    }
  },
  "$group": {
    "_id": { "player_name": "$player_name", "player_type": "INSTITUTION", "country_code": "$country_code" },
    "number_records": { "$sum": 1 }
  },
  "$project": {
    "player_name": "$player_name", "country_code": "$country_code", "number_records": "$number_records", "_id" :1}
},
  "$sort": { "number_records": -1 } ],
  { "$limit": 1000000 }
});

Listing A.17: MongoDB query example 5
# Data selection from multiple tables

db.embedded.aggregate([{
  "$match": {
    "$text": {
      "$search": "MOTION"
    }
  },
  "$match": {
    "doc_type": "PATENT"
  }
},
  "$match": {
    "doc_type": "SCIENCE"
  }
], # For doc_type = PATENT
  "$match": {
    "doc_type": "SCIENCE"
  }, #For doc_type = SCIENCE
  "$group": {
    "_id": {
      "player_name": "$player_name",
      "country_code": "$country_code"
    },
    "number_records": {
      "$sum": 1
    }
  }
],
  "$project": {
    "player_name": "$player_name",
    "country_code": "$country_code",
    "number_records": "$number_records",
    "_id": 1
  },
  "$sort": {
    "number_records": -1
  }
],
  "$limit": 100000)
)

Listing A.18: MongoDB query example 6

# Data selection from multiple tables

db.embedded.find(
  { "player_doc_link_type": "INVENTOR" }
).projection({})
  .sort({ _id: -1 })
  .limit(1000000)

Listing A.19: MongoDB query example 7
Listing A.20: MongoDB query example 8

```javascript
# Data selection from multiple tables

db.embedded.aggregate([{
  "$match": {
    "$text": {
      "$search": "behaviour"
    }
  },
  "$match": {
    "doc_type": "SCIENCE"
  },
  "$group": {
    "id": {
      "player_sub_type": ("STARTUP,COMPANY")
    },
    "player_name": "$player_name",
    "player_type": "INSTITUTION",
    "doc_source": "$doc_source",
    "date_inserted": "$date_inserted",
    "meta": "$meta",
    "country_code": "$country_code"
  },
  "number_records": {"$sum": 1}
},
{"$project": {
  "player_name": "$player_name",
  "player_type": "INSTITUTION",
  "doc_type": "SCIENCE",
  "date_inserted": "$date_inserted",
  "country_code": "$country_code",
  "number_records": "$number_records",
  "_id": 1
}},
{"$sort": {"number_records": -1 }},
{"$limit": 100000}
]);
```
Bibliography


model for semi-automatic integration of ethnomedicinal plant data from multiple sources. wileyonlinelibrary.com/journal/pca, April 2014. (cited on Page 67 and 68)


Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den April 23, 2019