Daniel Liberman

Migration from NoSQL to SQL

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

24 05 2023

Abstract

Author:	Daniel Liberman
Title:	Migration from NoSQL to SQL
Number of Pages:	42 pages + 2 appendices
Date:	24.05.2023
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Smart Systems
Supervisors:	Antti Piironen, Principal Lecturer
	Anne Pajala, Senior Lecturer

The thesis was under Nokia's premises, and the goal was to research and find methods for migrating from NoSQL to SQL. Moreover, part of the research identifies and analyses flaws in the NoSQL to learn from and act upon when shifting toward the new database. Furthermore, the research should also identify the functionalities of the new database, which would provide efficiency, maintainability, scalability, and security.

The research involves experimenting with different technologies like Python, JavaScript, SQL, and Docker. The programming languages provide the ability to create scripts and analyse the data flow and performance in real-time. Docker simplifies the process of experimenting with new technologies, such as PostgreSQL, as it provides a plug-and-play service to ease the development process.

The migration involves two significant components, NoSQL and SQL databases. Hence, the research splits into three different sections, consisting of an overview of the current state of MongoDB, the migration process, and PostgreSQL as the new database. Each section strives to provide insights into possible ways of structuring and developing toward best practices.

The study provides an overview of the migration process and its outcome. These insights can support others as a guideline for migrating from NoSQL to SQL. Although, each application is an individual case. However, there are core concepts that can assist in simplifying the process. The act of migration assists with the productization of Nokia's project, providing the necessary security, maintainability, and scalability to provide an exceptional product for its customers.

1 Introduction 1

2 Nokia’s Project NoSQL Architecture 1

2.1 Overview of Database Design Principles 2
2.2 Nokia’s Database Design Pattern Flaws 6
2.3 MongoDB Licensing Limitation 8

3 Theoretical Background 9

3.1 MongoDB Database Specifications 9
3.2 MongoDB to PostgreSQL Data Migration 16
3.3 PostgreSQL Optimization 18
3.4 PostgreSQL Features 22
3.5 PostgreSQL Security 23

4 Implementation 24

4.1 MongoDB Best Practices 24
4.2 Migration Script 27
4.3 PostgreSQL 32
4.3.1 PostgreSQL Optimization 32
4.3.2 PostgreSQL Features 33
4.3.3 PostgreSQL Security 35

5 Conclusion and Discussion 38

6 References 39

7 Appendix 42

List of Abbreviations

DBMS: Database management system. Software for maintaining, querying, and updating data and metadata in a database.

ORM: Object-relational mapping. The set of rules for mapping objects in a programming language to records in a relational database, and vice versa.

ODM: Object data manager. Data manger which intendeds for storing and maintaining information as objects with associated characteristics.

SSL: Secure socket layer. Internet security protocol which ensures privacy, authentication, and data integrity in internet communications.

TLS: Transport layer security. The modern version of SSL, it is a security protocol which ensures privacy and data protection in communication over the internet.

UUID: Universal unique identifier. It is a 128-bit value ensuring a unique identity to an object or entity in a software.

GUI: Graphical user interface. Interface which utilizes different components to enables human to machine interaction, such as, icons, menus, and a mouse.

ERD: Entity relationship diagram. A diagram that represents in a graphical manner the relationship between tables in a database.

API: Application programming interface. Software interface that allows communication between software.

1 Introduction

This thesis explores data migration in telecommunication testing software. Nokia's project is a test automation tool that uses test automation frameworks. The software supplies a manner to automate telecommunications testing in a lab environment.

The software is currently used by telecommunication experts, where the tester can customize the test environment. The software provides the tester with a user-friendly, fast, and powerful tool for testing, as it is highly configurable to work with other software or hardware.

The software is undergoing a productization process for commercial release. As part of the productization, it requires improvements to ensure the best product for its future customers. The improvement consists of different fields, with the backend being one of them.

The improvements needed within the backend have pushed the project to migrate from NoSQL to SQL, and the reasoning behind it is as follows. Primarily, the data within the database has a relational nature. Secondly, the data requires a more effective data structure for maintainability. Thirdly, the database must be secure and scalable. Lastly, the current NoSQL database, MongoDB, changed its licensing, preventing the usage of MongoDB commercially.

This thesis aims to recommend improvements to Nokia's project to ease its commercial release. The thesis explores the current flaws, which would assist in identifying possible changes for SQL's data structure. Moreover, the thesis explores the concepts of migrating data from NoSQL to SQL. Finally, the thesis explores PostgreSQL as a possible SQL database to migrate to and its features.

2 Nokia’s Project NoSQL Architecture

It is critical to have a solid foundation for the software architecture when developing software as a product. The front end of the software is the content presented to the client. On the other end of the spectrum is the backend, which handles heavy-duty

operations and database communication. The importance of having a suitable database architecture is to reduce the complexity of the logic handled by the server. Furthermore, it ensures a logical way of storing the data, which affects the overall performance of the application.

2.1 Overview of Database Design Principles

A fundamental aspect of building an application is structuring the database architecture before the implementation of the application. Several factors should be considered, such as what problem the database structure is trying to solve, the rate of read or write operations to the database, estimation of the scale of the amount of data the database will contain, and lastly, if the application is using real-time processes or batch processes. **(Edward and Sabharwal, 2015: 1.)**

Nokia's software requires flexibility and data accessibility, as data updates with real-time operations. In real-time operations with complex querying, a normalized structure is a recommendation. Normalization, also known as normal form, is a technique used to reduce data redundancy in a relational database. However, MongoDB can use a similar implementation. **(Edward and Sabharwal, 2015: 1.)**

Normalization is the process of organizing data in a database. It is essential to establish tables based on the identified relationships between the entities when organizing the database. Data relationships consist of three types. Firstly, a one-to-one relationship is between two entities where only one entity can belong to another. The following table depicts a one-to-one relationship between a person and a passport. **(Edward and Sabharwal, 2015: 1.)**

Person				
ID	First Name	Last Name	Age	Passport ID
1	John	Doe	42	12345

Passport			
ID	Expiration	Country Issued	Person ID
12345	12/12/2022	Finland	1

Table 1. One-to-one relationship between Person and Passport entities

Table 1 shows that one person can own one passport, and one belongs to another. The second relation is one-to-many, which is a relationship between two entities in which many belong to one entity. The following table shows an example of a one-to-many relationship. **(Edward and Sabharwal, 2015: 1.)**

Car Manufacturer		
ID	Name	Country
1	Ford	USA
2	Hyundai	KOR
3	Nissan	JPN

Car Models		
ID	Model	Manufacturer ID
1	Mustang	1
2	Focus	1
3	Accent	2
4	Sentra	3
5	Quest	3
6	Altima	3

Table 2. One-to-many relationship between Car manufacturer and Car models

Table 2 shows that one car manufacturer can produce several car models, whereas only one can be manufactured by one manufacturer. Finally, many-to-many refers to entities that belong to many other entities and vice versa. Courses and students are two examples of many-to-many relationships. The following table illustrates the many-to-many relationship. **(Edward and Sabharwal, 2015: 1.)**

Student		
ID	First Name	Last Name
1	John	Doe
2	Johnny	Doe
3	Joe	Doe

Course	
ID	Name
1	Mathematicas
2	C-programming
3	Physics

Enrollments		
ID	Student ID	Course ID
1	1	1
2	1	2
3	1	3
4	2	1
5	2	3
6	3	1
7	3	3

Table 3. Many-to-many relationship between Student and Course entities

Table 3 shows that a student can enroll in multiple courses, each with multiple students. When many-to-many relationships occur, they require using an intermediary table into one-to-one or many-to-one relationships. A table that maps between two many-to-many relationships is known as an intermediary table. **(Edward and Sabharwal, 2015: 1.)**

Normalization in MongoDB implies reducing data redundancy by splitting many-to-many relationships into two possible collection designs using embedding and referencing. Inserting relational data into the same document is known as embedding. The document is analogous to a table row. The following figure represents two collections with many-to-many relationships. **(MongoDB, n.d.: 2.)**

```
// Student - Collection
[
  {
    "id": 1,
    "firstName": "John",
    "lastName": "Doe"
  },
  {
    "id": 2,
    "firstName": "Johnny",
    "lastName": "Doe"
  },
  {
    "id": 3,
    "firstName": "Joe",
    "lastName": "Doe"
  }
]
```

```
// Course - Collection
[
  {
    "id": 1,
    "name": "Mathematics"
  },
  {
    "id": 2,
    "name": "C-programming"
  },
  {
    "id": 3,
    "name": "Physics"
  }
]
```

Figure 1. JSON representation of Student and Course collections

Figure 1 displays a similar data structure to Table three. MongoDB recommends using an embedded document structure. Embedded documents are more efficient than referencing because all associated data is stored together. There is an existing relationship between the student and course collections. The following illustration shows an example of the embedded strategy. **(MongoDB, n.d.: 2.)**

```
// Student - Collection
[
  {
    "id": 1,
    "firstName": "John",
    "lastName": "Doe",
    "courses:" [
        {
          "id": 1,
          "name": "Mathematics"
        },
        {
          "id": 2,
          "name": "C-programming"
        },
        {
          "id": 3,
          "name": "Physics"
        }
      ]
  }
]
```

Figure 2. Courses embedded into Student collection.

Figure 2 displays an embedded design for storing data in MongoDB. Another application embedding data is the extended reference pattern. The extended reference pattern entails appending the most-accessed information from a document to the associated document. The following figure displays an example of the extended reference pattern. **(MongoDB, n.d.: 2.)**

```
// Student - Collection
[
  {
    "id": 1,
    "firstName": "John",
    "lastName": "Doe",
    "courses": [
      {
        "id": 1
      },
      {
        "id": 2
      },
      {
        "id": 3
      }
    ]
  }
]
```

Figure 3. Extended reference pattern

Figure 3 represents the extended reference strategy. The core concept is utilizing the least parameters to reference documents between collections. In Figure three, the student document uses the id parameter of the course document inside the student collection to simplify the interaction querying between multiple collections. **(MongoDB, n.d.: 2.)**

2.2 Nokia's Database Design Pattern Flaws

The database used by Nokia's project is MongoDB. MongoDB is an open-source NoSQL database that does not adhere to SQL standards and stores non-relational data. MongoDB has the characteristics of a relational database while also serving as a general-purpose NoSQL database, which is one of its advantages**. (Giamas, 2017: 3.)**

MongoDB is a schema-less database. Due to MongoDB being schema-less, it was not considered a scalable solution for the cost of a database design. Furthermore, MongoDB cannot confirm the successful transaction completion between the server and the database. MongoDB is a write-and-forget database, as the implementation focuses on fast data insertion with minimal error handling. **(Giamas, 2017: 3.)**

Nokia's project case elucidates the previously mentioned MongoDB flaws. The database architecture is fundamentally a relational database. MongoDB, on the other hand, was chosen because it is developer-friendly and meets the requirements for providing a functional database for the software. **(MongoDB, n.d.: 2.)**

As a result, it is necessary to have a method to overcome the limitations imposed by its relational nature in a non-relational database. Nokia's project overcomes relational difficulties by utilizing a manual reference design pattern. The referencing method is referencing a document between collections using a UUID parameter. The following figure depicts an example of the reference implementation. **(MongoDB, n.d.: 2.)**

Figure 4. Manual reference between documents

Due to the lack of cascade operations, referencing a document from another collection as a parameter could be better. Cascade operations have an impact on entities that

rely on other entities. When removing a dependent document in Nokia's project, the database does not automatically delete all entities that are dependent on the document. The application requires multiple access to the database to remove the dependent entities. These operations may cause data integrity issues and be costly to the application's performance. **(Edward and Sabharwal, 2015: 1.)**

MongoDB is schema-less, meaning that the application accepts inserting any data regardless of its data type. Accepting data types without data type validation or illogical deviation between each document can cause several issues. For example, the following shows examples of different data types for the same parameter within the same collection. **(Edward and Sabharwal, 2015: 1.)**

```
_id: ObjectId('6360ced56c2af3c188f1f210')
logstorage_root: "C:\logstorage"
id: "1ed59b94b07d620"
    _version: "3.22.11.160"
workspace_name: "test-abort"
user_name: "abc"
start_time: 1667288789
delta_time: 2
end_time: 1667288791
utc_offset: 120
verdict: 1
modified_verdict: 0
directory_size: 476090
```

```
_id: ObjectId('6360ce726c2af3c188f1f20b')
logstorage_root: "C:\logstorage"
id: "1ed59b90ec189e0"
    _version: "3.22.11.160"
workspace_name: "test-abort"
user_name: "abc"
start_time: 1667288690
delta_time: 2
end_time: 1667288692
utc_offset: 120
verdict: 1
modified_verdict: null
directory_size: 476090
```

```
_id: ObjectId('60a4d7b44c06a3714c44be62')
logstorage_root: "/mnt/bfat_log/bfat"
id: "1ebb883024a9180"
    _version: "3.21.5.231"
workspace_name: "TA2"
user_name: "youjianping"
start_time: 1621415858
delta_time: ""
end_time: ""
utc_offset: 480
verdict: 0
modified_verdict: ""
skip_cleanup: true
```

Figure 5. Example of different data types of the parameter "modified_verdict"

Figure 5 shows different data types for the same parameter. The value varies between null and empty string. As a result, there will be a lack of consistency across the data, which may cause data parsing issues in the backend. Furthermore, across all the collections, there needs to be more usage of the _id field in Nokia's project application. The _id field is a built-in parameter in MongoDB appended to every document created. Documents in Nokia's project use UUID in most cases. However, there are cases where the documents are random strings. In both cases, the two use the parameter id. **(MongoDB, n.d.: 2.)**

The parameter _id is automatically set as a primary key, making it a unique index. As a unique index, it would prevent document duplication. MongoDB's _id is also optimized to provide a faster query lookup. Nokia's project did not use the _id parameter, resulting in a lack of a unique index. The lack of a unique index resulted in duplicate documents and possible performance loss, as illustrated in the following figure. **(MongoDB, n.d.: 7.)**

Parameters

```
_id: ObjectId('637782be963f968e23f0bf08')
uuid: "f26f5831-c399-4338-afb9-aabd71f5ce62"
name: "id"
description: "Basestation ID.
              Must be unique in current environment.
              Must be digit…"
type: "string"
default: ""
path: ""
elementSpecific: true
regExp: "^\d+$"
elementUuid: "bd6ebeb5-da40-4886-86d0-140511f78ac8"
position: 20
```

```
_id: ObjectId('637782be963f968e23f0bf1f')
uuid: "f26f5831-c399-4338-afb9-aabd71f5ce62"
name: "id"
description: "vCU ID.
              Must be unique in current environment.
              Must be digits."
type: "string"
default: ""
path: ""
elementSpecific: true
regExp: "^\d+$"
elementUuid: "c996f919-154d-4f9b-ac51-95163b363465"
position: 20
```

Figure 6. UUID duplication within the same collection

Duplications in a database can have disastrous consequences because changes or calls to the document based on its UUID result in the manipulation of another document in addition to the desired one. Furthermore, a race condition may occur when functionality uses the UUID as a parameter to fetch data to a duplicated UUID. A race condition may occur when the behavior of functionality is dependent on the outcome of an event. For example, when two elements try to change or return data simultaneously, resulting in random behavior as to which element returns the data first. **(MongoDB, n.d.: 9.)**

The data structure of Nokia's project application could be improved when migrating data between MongoDB and PostgreSQL. Furthermore, identifying flaws may assist with de-structuring the database, possibly simplifying the normalization of the data. Finally, addressing the flaws is crucial to open the possibility of creating a better application data structure, which may result in improved performance, scalability, maintainability, and enhanced data integrity.

2.3 MongoDB Licensing Limitation

MongoDB announced they were switching from an open-source project to a closed project in 2018. The new license declares that using version 3.6 and earlier versions

will remain open to the public. The reason MongoDB changed its licensing is due to enterprises and businesses abusing MongoDB's platform. **(SSPL, 2018: 22.)**

The new license is the server-side public license, which specifies conditions for utilizing MongoDB as a service. The license specifies that the utilization version of MongoDB released before 2018 is still available for everyone. However, MongoDB will maintain these versions of MongoDB databases until their end-of-life, which already occurred in 2021. Moreover, any party utilizing any version MongoDB released after 2018 must have the software source code open to the public. **(SSPL, 2018: 22.)**

Since MongoDB's latest version before 2018, version 3.6, has already reached the end of its life, it has become a security risk **(MongoDB, n.d.: 24).** MongoDBs' developers will not maintain these versions anymore, which turns these versions into a potential attack vector. Threat actors can take advantage of the fact that the software will not be updated anymore, giving them more time to identify new attack methods. Threat actors can use different tools to perform an exploit easily, as every exploit is simple to find and use **(Hoffman, 2020.: 22).** From the point of view of corporate software, continuing to use MongoDB version 3.6 is an unnecessary risk.

3 Theoretical Background

3.1 MongoDB Database Specifications

MongoDB is schemeless in nature, unlike SQL databases, whereas declaring a table and the datatypes are necessary to insert data. This flexibility eases document mapping as there could be different document variations inside a collection. However, the challenge occurs when the data model needs to match the application's needs regarding data usage, such as data retrieval patterns, updates, and data processing. **(Giamas, 2017: 3.)**

Mongoose is an ODM (Object-Document Modeler) that defines data models. Mongoose excels in interaction with structured and repeatable data, as it reduces the complexity of server-to-database operations. Although MongoDB is schemeless, there is an option to provide schema and data validation to MongoDB to reinforce the database with a powerful tool to sustain data integrity while supplying a high-performance database. It is important to emphasize that the following data structures

should not use Mongoose, schemeless, random documents, or pure key-value pairs. 4. **(Holmes, 2013: 4.)**

Mongoose schemas are one of the core functionalities of Mongoose. The schema describes the data that will construct the document instance in the database, which includes the key and the expected value data type. The following illustration is an example of a simple Mongoose schema. **(Holmes, 2013: 4.)**

```
import mongoose from 'mongoose';
const { Schema } = mongoose;

const blogSchema = new Schema({
  title: String, // String is shorthand for {type: String}
  author: String,
  body: String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs: Number
  }
});

```

Listing 1. Mongoose's schema example. **(Mongoose, n.d.: 5.)**

Listing 1 shows a key-value structure to the document. The key-value defines the data type or other possible validation methods for the data. Schema increases data integrity and supplies the developer with an idea of the database's data types. Additionally, what makes schemas so powerful is their helper function extensions and additional methods. **(Mongoose, n.d.: 5.)**

Schemas provide the possibility to set rules for the data to follow, such as specifying the expected data type of the parameter, the parameter to be unique, the default value for the parameter, and other built-in functionalities given by Mongoose. Mongoose models are another core functionality of Mongoose. Models have compiled schema versions, and one model instance is equivalent to one document in the database. **(Holmes, 2013: 4.)**

Mongoose provides basic CRUD operations to the model. The CRUD operations provided by Mongoose are as such find, create, update, and remove. The CRUD operations as similar to MongoDB's shell operations, simplifying migration from MongoDB's module to Mongoose. Furthermore, Mongoose, in some cases, provides a more efficient querying method than MongoDB. **(Mongoose, n.d.: 6.)**

Data validation is a crucial part of the backend, as the application needs to provide data integrity. Mongoose provides built-in validators, also giving the possibility to the developers to build their custom validator. One basic validator is the unique specification in the schema, which checks for other existing documents with a similar value for the specified parameter. The validators can be global or data type specific. Global validators consist of unique or required validators. Mongoose's validation process determines whether the data validation requires MongoDB for an in-database check or whether Mongoose can handle it with built-in methods. **(Holmes, 2013: 4.)**

The required validator does not require MongoDB's in-database elements to assess whether the field is mandatory. For example, the unique validator checks for value duplications at the MongoDB level. Additionally, Mongoose has data-specific validators, such as the number data type validator consisting of min and max validators. Lastly, the string validators consist of string match or array matching validators. **(Holmes, 2013: 4.)**

To place a custom validator into the schema, create a validator function. The validate built-in method will take the custom validator into effect. A good practice would be defining the function names based on the validation action and field of validation. Another good practice is creating reusable validators. **(Holmes, 2013: 4.)**

Building complex data can use the population or subdocument strategies in the schema. The population is a referencing system that one model that pulls the information of a second model. Using the population method is as follows. Firstly, define the desired link to the referenced model in the primary schema by passing the object containing the name of the linked schema object. Only ObjectID, Number, String, and Buffer are valid for reference, and it is possible to declare them as array referencing. Referencing as an array would equal a one-to-many relationship, while referencing a single value would be a one-to-one relationship. Secondly, by utilizing Mongoose's CRUD operations with the built-in method populate, Mongoose will follow

the schema rules and handle the referencing. The following listing shows an example of the population strategy. **(Holmes, 2013: 4.)**

```
1. { "projectName" : "Population test",
2.   "createdBy" : ObjectId("5126b7a1f8a44d1e32000001"),
3.   "_id" : ObjectId("51ac2fc4c746ba1645000002"),
4.   "contributors" : [
5.     ObjectId("5126b7a1f8a44d1e32000001"),
6.     ObjectId("5126b7a1f8a44d1e32000002") ] }
```

Listing 2. Example of populate strategy structure. **(Holmes, 2013: 4.)**

The second strategy, Subdocuments, are like ordinary documents in nature. However, subdocuments are documents saved within their parent document. Subdocuments are more efficient than the populate strategy in terms of performance since it reduces the number of operations with the database. An example of the subdocument is as follows. **(Holmes, 2013: 4.)**

```
 1. {
 2.   "projectName" : "Project 2",
 3.   "tasks" : [
 4.       "taskName" : "A task please",
 5.       "taskDesc" : "A short description of the task",
 6.       "createdBy" : ObjectId("5126b7a1f8a44d1e32000001"),
 7.       "_id" : ObjectId("51ad7d6cfa492a174a000005"),
 8.       "createdOn" : ISODate("2013-06-04T05:38:52.847Z")
 9.     },
10.   ]
11. }
```

Listing 3. Example of the subdocument strategy structure. **(Holmes, 2013: 4.)**

In a relational database, subdocuments traditionally would have been separated into another table. However, in a non-relational database, it is possible to create subdocuments within a document, which avoids the unnecessary JOIN operations in the database. Concluding in better performance for read and write operations. **(Mongoose, n.d.: 6.)**

Indexing is another operation that could improve the performance of read operations in the database, though indexing is costly in a high write-to-read ratio. Indexing is another implementation of the B-tree data structure. The B-tree data structure allows CRUD operations executions in logarithmic time. The logarithmic time applies for the average

and worst case of possible performance, which brings stability to applications. The following diagram emphasizes the core idea of logarithmic time. **(Giamas, 2017: 3.)**

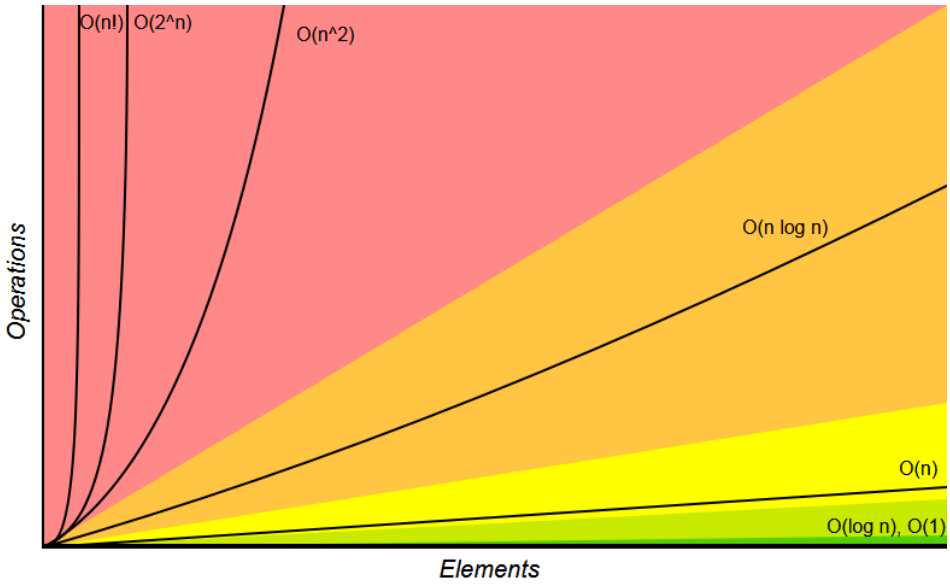


Figure 7. O-time of operations in respect to elements. **(Giamas, 2017: 3.)**

Time performance is consequential in applications that rely upon algorithms for fetching the data as fast as possible. As can be learned from Figure 7, operations that take logarithmic time is superior to other time complexities in terms of operations concerning elements. Additionally, B-tree is self-balancing, which means it will self-adjust the data to keep the search algorithm the in a manner that will approximately halve a search space with every step, thus maintaining its properties of having a log n time complexity. **(Giamas, 2017: 3.)**

There are various indexing strategies to construct proper indexes for different collections. Additionally, MongoDB provides tools to analyze and identify inefficient queries. When indexing, there are several elements to consider, the read-to-write ratio, the amount of free memory in the system, and the level of understanding of the application's queries. Understanding the application's queries will ease constructing an index by understanding the frequency of each query and if it justifies an index. Once an index is justifiable, a general strategy is to test different indexes and utilize the query analyzing tool to choose the best-performing index. **(MongoDB, n.d.: 7.)**

An index works as follows when a query executes, then the query scans the index, and if the index fulfills the requirement of the query, then the query will return the elements based on the index, which concludes in avoiding unnecessary visits to the collection. Indexes can be on a single-key or multiple keys, depending on the desired query. Using a single key should take place only when a query requires a single key from the collection. When the query requires more than one key, then using multiple keys is a better approach since the query uses the required index from the allocated indexes. **(MongoDB, n.d.: 8.)**

There are a variety of possible indexes to use, starting from a single field index. Singe field indexes refer to a simple case of an index on one field or key, such as the default index in every MongoDB collection, which is the index on the _id key. When creating indexes, it is possible to set query rules, such as the order for the data retrieval. However, it is only sometimes necessary as indexing is efficient, and setting rules would not significantly impact the overall performance. **(Giamas, 2017: 3.)**

Another indexing system is embedded field indexing. Since MongoDB supports nested documents, naturally, it allows indexing the embedded document fields. The index will pinpoint the embedded parameter by utilizing the method createIndex. The following code block presents a document with embedded data and the method to create the index. **(Giamas, 2017: 3.)**

```
1. {
2.   "_id" : ObjectId("5969ccb614ae9238fe76d7f1"),
3.   "name" : "MongoDB Indexing Cookbook",
4.   "meta_data" : {"page_count" : 256 }
5. }
6. db.books.createIndex( { "meta_data.page_count": 1 }
```

Listing 4. Embedded document and embedded index. **(Giamas, 2017: 3.)**

However, it is important to note, when the collection holds a vast amount of data, and the application requires frequent access, then it is possible to use the index in the background. Background indexing runs operations in the background allowing the database to run while creating the index. Nonetheless, there is a downside, as the mongo shell will be busy with index creation. Overcoming the blockage is as easy as opening a new connection to the database, as the new incoming operations will take place on top of the index creation, although, creating an index in the background has

an impact on the performance as it utilizes the database resource for the sake of the index creation. **(MongoDB, n.d.: 21.)**

Another indexing strategy is compound indexes. MongoDB supports multiple-field indexing. In compound indexes, the order of the indexes matters, as the order will determine the order of the retrieved data. However, it is essential to know that compound indexes cannot accept arrays as a field. **(Giamas, 2017: 3.)**

Although indexes can be beneficial for the performance of database operations, they can also lead to challenges. Indexes that are created and remain unused can impact the performance poorly. Similarly, using many indexes harms the database performance. Thus, it is essential to understand the application's query architecture. **(MongoDB, n.d.: 9.)**

Assessing the enhancement that indexing brings can be done using the performance measurement feature of MongoDB. The performance measure is a method named explain. The method provides the analysis from the query planner, and the developer can choose how detailed the analysis will be. An example of the performance measurement feature is as follows. **(Giamas, 2017: 3.)**

```
1. db.keys.find( { x : { $in : [ 3, 4, 50, 74, 75, 90 ] } } ).explain(
"executionStats" )
```

Listing 5. Example of the explain method in MongoDB. **(MongoDB, n.d.: 10.)**

The query optimizer processes queries and searches for the most viable query plan which fits the query shape. If the query planner cannot find matching candidates, it will generate possible plans for evaluation over a trial period. The query planner then elects the winning plan and caches the entry containing the winning plan to generate the result documents. If the query planner finds matching candidates, it creates a plan based on the cached entry and evaluates its performance through a replanning mechanism. The replanning mechanism assesses the performance of the entry with a pass-or-fail evaluation. If the decision fails, the replanning mechanism will remove the cache entry, and the query planner will select a superior plan for executing the query **(MongoDB, n.d.: 11.)** The diagram illustrating the query planner execution in appendix 1.

3.2 MongoDB to PostgreSQL Data Migration

Linux is a popular operating system with a vast range of utilization, from mobile devices to cloud computing. The operation system operations, in short, are as follows, manages the memory, handles real-time operations, handles processes with I/O devices, manages files, and configures. Nokia's software runs only on Linux. Hence the migration script requires Python or Shell scripting. **(Hausenblas, 2022: 12.)**

Python is a multipurpose programming language commonly used for web development, scientific computing, data analysis, artificial intelligence, and automation. Python is a high-level programming language that met the market in 1991. Moreover, Python's popularity has grown steadily over the years due to its flexibility, ease of use, and various libraries and frameworks. Additionally, Python is highly portable, meaning Python code can run on different platforms, such as Windows, Linux, and macOS. This portability makes it easy to deploy anywhere, reducing the cost of software development and maintenance. **(van Rossum, n.d.: 13)**

Using Python enables the use of the package PyMongo. Pymongo is a Python library that provides a simple and intuitive API for working with MongoDB. Pymongo has various features for interacting with MongoDB, such as querying, updating, data manipulation, and managing indexes and collections. One of the core features of Pymongo is its support of BSON, a binary representation of JSON used by MongoDB to store and exchange data. Pymongo also supports aggregation, a flexible way to query and manipulate data in MongoDB. Aggregation allows for performing complex queries and data transformations, such as grouping, sorting, and filtering. Pymongo eases the complexity of working with MongoDB in Python and can leverage the power of the MongoDB document-oriented data model. **(Pymongo, n.d.: 14.)**

Another valuable component for the data migration process is MongoCompass. MongoCompass is a GUI for MongoDB database. MongoCompass visualizes MongoDB's data structures, simplifying the navigation and interaction with the data. MongoCompass provides some built-in features for working with MongoDB, which include the ability to view and edit data, run queries, manage indexes, and analyze data. It also provides several tools for managing MongoDB instances, such as managing users and roles and configuring security settings. **(MongoDB, n.d.: 15.)**

Prisma is introduced to the application to simplify interaction with the PostgreSQL Database. Prisma is an open-source ORM that simplifies database access by providing type-safe API for accessing databases. Prisma supports various databases, including PostgreSQL, MySQL, and SQLite. Prisma supports table creations utilizing the schema method. Creating a schema for PostgreSQL requires installing Prisma CLI and initializing a new Prisma project using the following commands. **(Prisma, 2023: 13.)**

```
1. npm install prisma --save-dev
2. npx prisma init
```

Listing 6. Prisma's initialization commands. **(Prisma, 2023: 13.)**

The database schema uses Prisma's schema definition language. Prisma's definition language is the data model. The data model is a set of objects defining the structure of a database schema, including the relationships between the tables. Prisma's schema definition language provides a variety of built-in scalar types, such as String, Int, Float, Boolean, and DateTime, which define the data types of the columns in the database. Prsima's schema definition also supports constraints and validations. For example, the @unique directive specifies that a column should be a unique value across all rows in the table. The @relation keyword is another example that defines the relationships between tables. Another feature of Prisma's schema definition language is the ability to define enums, which represent a fixed set of values that a column can take. This feature makes it easier to understand and work with the data in the database. The Prisma schema sample is as follows. **(Prisma, 2023: 14.)**

```
 1. datasource db {
 2. provider = "postgresql"
 3. url = env("DATABASE_URL") }
 4. generator client {
 5. provider = "prisma-client-js"
 6. }
 7. model User {
 8. id Int @id @default(autoincrement())
 9. createdAt DateTime @default(now())
10. email String @unique
11. name String?
12. role Role @default(USER)
13. }
```

Listing 7. Prisma's schema definition language sample. **(Prisma, 2023: 14.)**

Another tool that is beneficial for the data migration process is Psycopg 2. Psycopg 2 is a database adapter for Python. It provides a Python interface for working with

PostgreSQL databases, allowing easy access and manipulation of data stored in PostgreSQL. Psycopg 2 supports all the features of PostgreSQL, including transactions, stored procedures, and user-defined types. It provides a high level of compatibility with Python's database API. **(Di Gregorio and Varrazzo, n.d.: 18.)**

3.3 PostgreSQL Optimization

Performance may refer to any modifications resulting in improved system performance. Optimization should be an integral part of database development, as it is a more productive development method. Writing queries without analysing query performance may create problems in the future, and avoiding it is as simple as considering optimization from the beginning of development. Optimization of the database is understanding how the database engine processes a query and how a query planner decides its execution path. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

SQL is a declarative language in contrast to imperative programming languages. Unlike programming languages, which define the steps, declarative language specifies the desired result without writing the steps to obtain it. Applying similar imperative logic of programming languages to SQL may result in suboptimal performance. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

Optimization may imply a reduction in the execution time. However, the definition is vague as the execution time may differ between application goals. The framework SMART exists to assist in setting up optimization goals. The core concept of the SMART framework is understanding the business needs within reason and creating a database that would deliver the needs of the company and the usability of the company's clients. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

Database design and optimization should consider the application development actions integral to the database design, as database design relies on the application's functionality. For example, the type of information presented in the web UI may affect the normalization of the database's tables. Normalizing the database may harm the performance and increase complexity rather than increase efficiency and improve the performance. Changes in the application over time, such as data volume, could impact performance. Therefore, optimization requires constant checks of the database

performance, as automated performance checks are not always alerted. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

The algorithm cost model is an algorithm to compute the resources needed for executing a query or a single physical operation. The optimizer considers two crucial points, CPU cycles and the number of I/O operations. Then, the optimizer combines those into a single cost function. The low cost of the cost function means a better plan. Cost models assess the resources required to execute the operation, and it depends on the tables. The representation of the cost model is by the number of rows with respect to the storage blocks occupied by the table. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

The algorithms for reading data from database objects are called data access algorithms. The algorithm used from the data access algorithms depends on a ratio called selectivity. The selectivity ratio is constructed from the retained rows to the total rows in the stored table. Heaps are the rows in a table in a data structure. Another way to describe a heap is a row that utilizes any block with enough space without a specific order. Table rows, indices, and records are considered database objects and represent the data stored in the database. The database divides database objects into blocks of the same length. Blocks represent the unit that transfers between the hard drive and the main memory, and each block has 8192 bytes. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

Full scan is a data access algorithm. The full scan consists of the database engine reading through all the rows in a table and checking for any filtering conditions for every row. The following formula is to estimate the cost of a full scan.

$$c1 * BR + c2 * TR + c3 * S * TR$$

Formula 1. Formula to estimate full scan costs. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

Formula 1 variables are as follows. C1, C2, and C3 represent the properties of hardware. BR represents the number of I/O accesses that are required, while TR represents the total number of iterations of filtering condition checks. Additionally, the cost of operations producing the output needs to be estimated, and this cost depends on selectivity, denoted as S. The overall cost is calculated by taking the product of c1

and BR, adding it to the product of c2 and TR, and then adding the product of c3, S, and TR. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

Index-based table access is yet another data access algorithm. Indexes are a unique way to represent which row contains a specific value and are considered redundant database objects. Indexes store no more information than the source table. In addition, indexes provide additional data access paths. As a result, it is possible to determine the values stored in a table row without reading the table. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

When executing a filtering condition on a table with an index already in place, it initiates an algorithm to extract a group of pointers to blocks containing rows that meet the filtering criteria. The only blocks that require access from the table correspond to these pointers. The selectivity values determine the cost model for this algorithm. When dealing with low selectivity values, the cost is proportional to the number of rows returned. When using larger selectivity values, the cost roughly equals the total number of blocks. When allocating resources to access the index in such cases, the cost may exceed that of a full scan. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

Another data access algorithm is the index-only scan. This algorithm utilizes the data from the index and applies the remaining filtering conditions to it, hence does not access the table data. The cost model for the index-only algorithm is lower than a full table scan for large selectivity values and proportional to the number of returned rows in small selectivity values. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

There are three factors to identify an index. Firstly, an index is a redundant data structure, as indexes can be dropped and reconstructed elsewhere without impacting the data. Secondly, indexes are invisible to the application, which means that often it requires specifying which index to use for efficient data retrieval. Thirdly, indexes optimize query performance by enabling efficient data retrieval based on specific criteria. Although indexes may be called redundant data structures, indexes are beneficial due to performance enhancement, as they execute quick checks of filtering conditions in a query. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

The most common index type is the B-tree index in database systems. The B-tree structure is a balanced tree consisting of hierarchically organized nodes associated

with blocks stored on a disk. B-tree leaf nodes contain an index record constructed of an index key and a pointer to a table row. In comparison, the non-leaf nodes contain data of the smallest key and the largest key in the block located at the next level and a pointer to that block. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

The root node initiates the search of the B-tree. When the algorithm cannot find the largest key, the search moves on to the block pointed by the key's associated pointer. The search continues until it reaches the leaf node, where the pointer points to table rows. B-tree can also utilize the range search. A range search begins with the node having the lower range bound and moves up the tree in order until it reaches the node containing the upper range bound. A sequential scan gathers all index keys inside the defined range in this traversal. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

PostgreSQL includes various index structures to support a variety of data types and filtering conditions. Hash index is part of these indexes, the hash index uses a hash function to find the block containing the index key, and its performance is superior to B-tree. In contrast to B-tree, the hash index underperforms in-range queries. Nevertheless, another index is the R-tree index, the R-tree index focuses on spatial data, and its index key always represents a rectangle in a multidimensional space. When utilizing an R-tree index, it returns all records having an intersection of the multidimensional rectangle. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

The manifestation of relational databases is utilizing the relationships across tables to combine the data. Database engines use various algorithms to achieve the goal of data combination. The database engine utilizes the algorithm based on its need. For example, large tables utilize the sort-merge and hash algorithms as they are efficient. On the other hand, the nested loop algorithm is better for small index-based combinations. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

The PostgreSQL query planner's task of generating a query plan consists of a strategy to execute a query that specifies the order and method of retrieving data from the database. The query planner first analyses the query's syntax and builds a query tree, which represents the logical steps required to execute the query. To optimize the query and generate a query plan, the query planner applies a set of rules to the query tree. This optimization process includes selecting the best algorithms for each step of the query execution based on the available data statistics and metadata about the

database. The query planner selects the algorithms based on several factors, including the size of the tables involved, the selectivity of the query predicates, the available indexes, and the available CPU and memory resources. Moreover, it considers the cost of different algorithms based on the estimated number of disk I/O operations required to execute a query. The query planner assigns a cost to each possible query plan and chooses the one with the lowest overall cost. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

PostgreSQL generates an execution plan based on the query planner. The execution plan outlines the database's steps to retrieve the data. Once the execution plan is in place, PostgreSQL executes the query following the outlined steps. PostgreSQL provides several built-in methods for analysing execution plans, such as the EXPLAIN command and EXPLAIN ANALYZE command. The EXPLAIN command displays a text-based representation of the execution plan. The EXPLAIN ANALYZE is more detailed, as it performs a real-time execution and displays the information on each stage of the query tree. **(Dombrovskaya, Novikov and Bailliekova, 2021: 19)**

3.4 PostgreSQL Features

In PostgreSQL, there is a built-in operation which is a view. The view is a virtual table representing query results from one or more tables in the database. Views help combine data from multiple tables into a single view. Users can query the view instead of manually joining the tables in every query by defining a view that combines the data from multiple tables. Views simplify and streamline the data access process, reducing the risk of errors and improving query performance. **(The PostgreSQL Global Development Group, 2023: 20.)**

Views can provide a layer of abstraction that limits the data that users can see or modify. View's limitation can be beneficial when users need access to specific data but cannot view or modify other data in the underlying table. Furthermore, views can hide sensitive or present data in a different format than the underlying tables. **(The PostgreSQL Global Development Group, 2023: 20.)**

The materialized view is a feature that contains the results of a pre-computed SQL query. Unlike a standard view, which provides a virtual representation of the data stored in the database, a materialized view caches the results of the underlying query,

allowing for much faster access to the data. Materialized views are beneficial when dealing with large amounts of data or executing complex queries involving multiple joins and aggregations. By pre-computing the query results and storing them in a materialized view, it is possible to avoid repeatedly executing queries against the database. The materialized view is accessible as any other table in the database. **(The PostgreSQL Global Development Group, 2023: 20.)**

One benefit of materialized views is that they can improve query performance by reducing the time required to compute the results of a query. By caching the results in the materialized view, subsequent queries can be executed against the pre-computed data rather than against the underlying tables in the database. However, it is essential to note that materialized views also have some limitations. Since they store pre-computed data, they may only sometimes reflect the most up-to-date information in the database. In addition, materialized views require additional disk space to store the cached results, which may concern large datasets. **(The PostgreSQL Global Development Group, 2023: 20.)**

3.5 PostgreSQL Security

PostgreSQL offers several security features to protect sensitive data stored in databases. Some of the security features of PostgreSQL are audit logging, TLS communication, and others. PostgreSQL provides built-in methods for audit logging. Audit logging in PostgreSQL is the process of recording all events and activities taking place in the database. The logs include user logins, data access, schema modification, and other activities relevant to monitoring the security compliance of the database. Audit logging is beneficial in security incidents and forensic analysis cases. Additionally, audit logging can generate alerts and notifications when specific events or activities occur, which is beneficial to ensure that potential security threats are identified and addressed on time. **(The PostgreSQL Global Development Group, 2023: 20.)**

TLS is a security protocol to encrypt data sent over a network connection. Establishing a TLS connection requires a TLS certificate and private key. In PostgreSQL, TLS can secure database and application communications by encrypting data transferred between the database and application. PostgreSQL supports TLS using the OpenSSL library, which provides certificates for encrypting and decrypting data sent over the

network. TLS is essential for database security, particularly for databases accessed over the internet or other unsecured networks. By encrypting data sent over the network, TLS helps to ensure that sensitive information remains confidential and is not intercepted or tampered with by attackers. **(The PostgreSQL Global Development Group, 2023: 20.)**

4 Implementation

The implementation consists of three different sections. Each section focuses on other aspects of the migration process and experimenting with various technologies. The first section explores MongoDB's ODM Mongoose and the differences in maintainability and efficiency. The second section visits the migration script and provides an approach for writing a NoSQL to SQL migration script. Lastly, the third section examines different aspects of PostgreSQL and how to execute them in practice.

4.1 MongoDB Best Practices

As a start, it is essential to revise the original data structure of MongoDB to detect the flaws and provide suggestions for them. The data in the thesis is a sanitized version of the software's original dataset. The data consists of three main collections, test_execution, test_suite, and test_case. MongoCompass is used to simplify the interaction with the data and makes the data more readable by developers.

In the thesis, MongoCompass is the tool utilized. To utilize MongoCompass, the developer can download it from the official MongoDB website. It is also the location to create a new cluster to host the data on MongoDB's cloud. Also, a developer can host the data locally. The developer can use a URI provided by MongoDB'S cloud or the local server to connect to the database, as seen in the following illustration.

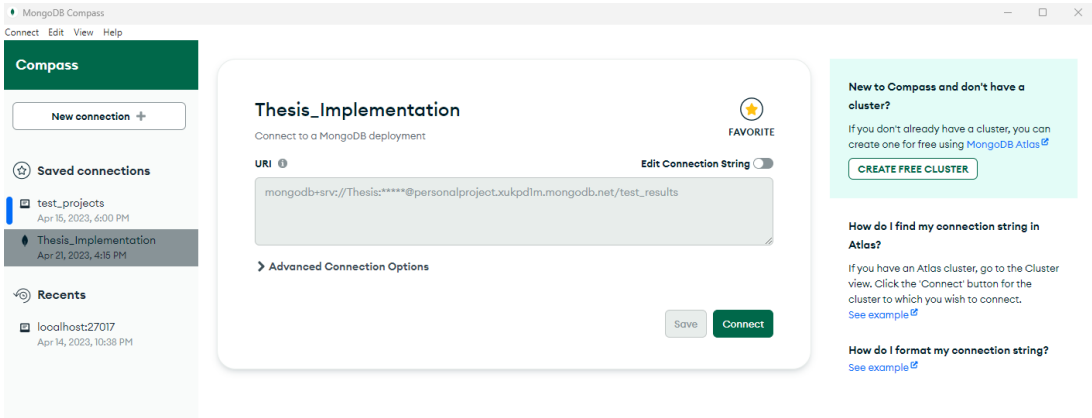


Figure 8. MongoCompass connection page.

MongoCompass's dashboard contains multiple features facilitating the interaction with the data, for example, simple interaction with databases, aggregation tools, schema, explain plan, indexes, and validation. These tools can also be used for query optimization. However, it is out of the scope of the thesis. An example of a MongoCompass dashboard is as follows.

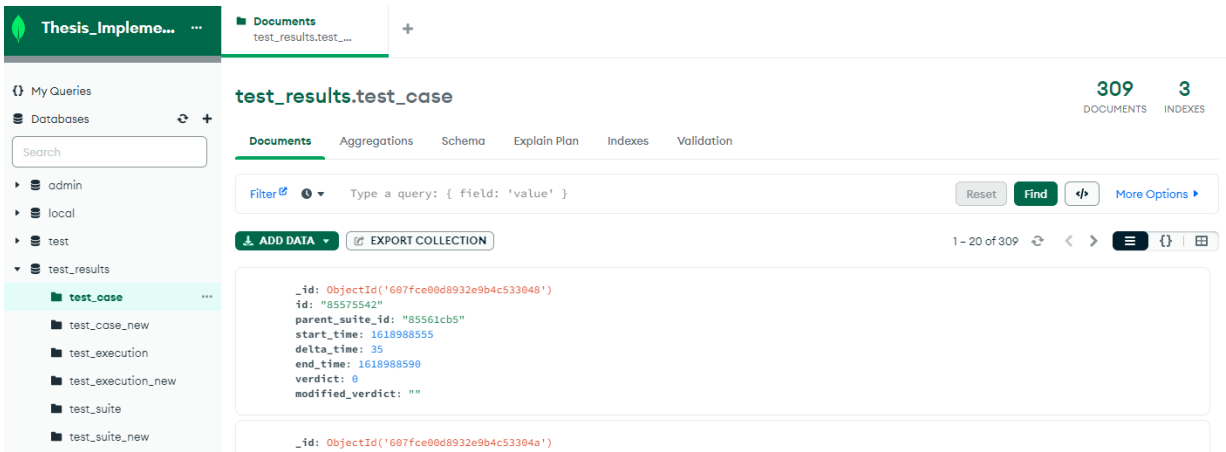


Figure 9. MongoCompass’s dashboard.

The test result collection has a hierarchical data structure, where test execution is at the top. The test execution collection consists of documents inserted after finalizing a test execution in the software. A sample of the test execution's data structure is as follows.

```
_id: ObjectId('6360ce726c2af3c188f1f20b')
id: "1ed59b90ec189e0"
start_time: 1667288690
delta_time: 2
end_time: 1667288692
utc_offset: 120
verdict: 1
modified_verdict: null
directory_size: 476090
```

Figure 10. Test execution's data structure.

Figure 10 shows that each execution got a starting time, end time, the time elapsed, verdict, and possibly modified verdict. Test executions can contain one or more test suites. Test suites represent a group of tests, the tests run as a unit, and the failure or success of unit tests depends on each of the group's tests. Test suites can be inserted into other test suites, meaning a test suite can reference another. The test suite is the child of test execution and the parent of other test suites or test cases. A representation of the test suite's data structure is as follows.

```
_id: ObjectId('5ec23510398b00f40aaf35ea')
id: "c128cc3a"
parent_execution_id: "1ea98d6c09a6a70"
start_time: 1589785872
delta_time: 601
end_time: 1589786473
verdict: 0
modified_verdict: ""
suite_repeat: ""
set_repeat: ""
```

Figure 11. Test suite's data structure.

Figure 11 shows the relation between a test suite to a test execution or another parent within the test suite collection. Additionally, test suites include the start time of the test suite execution, end time, and the difference between them. Additionally, the test suite contains a verdict and modified verdict and the repetition of a suite or a test.

The test case is at the bottom of the hierarchical data structure. The parent of the test case is the test suite. Each test case is a group of keywords that will run later in the Robot framework. Robot framework is a keyword Python framework for test automation

purposes. The test case collection is the results of the Robot's automated tests. A sample of the test case data structure is as follows.

4.2 Migration Script

Migrating the data from MongoDB to PostgreSQL requires mapping MongoDB's data structure to fit PostgreSQL's relational data structure, starting with choosing the suitable data types in PostgreSQL for the attributes to utilize. The data types between MongoDB and PostgreSQL are different. However, they do share core similarities which would simplify the data type selection.

Returning to MongoDB Compass, utilizing the schema tab to analyse the data types in each collection, is possible. Analysing the data types would supply the necessary information to create the tables in PostgreSQL. An example of an analysed collection is as follows.

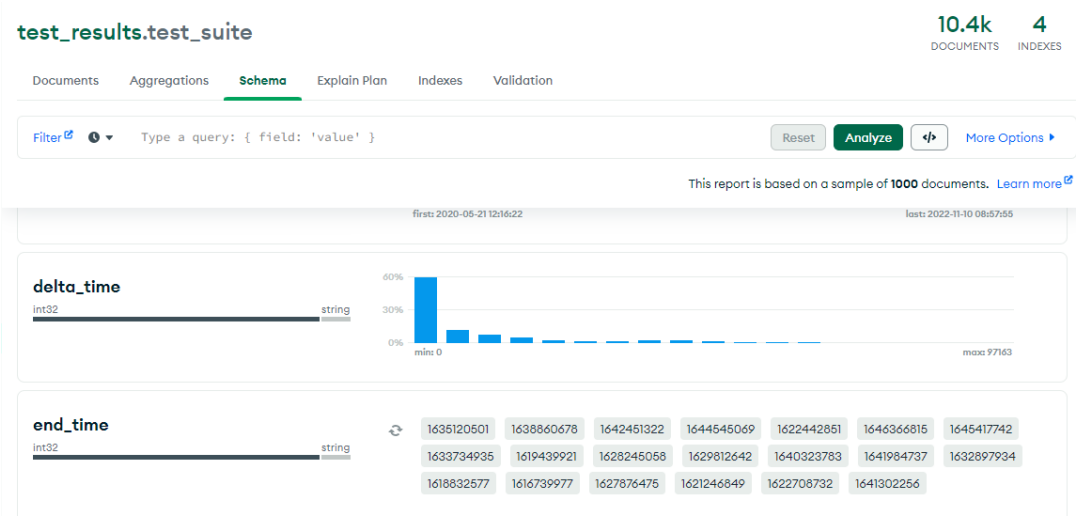


Figure 12. MongoCompass's schema tab.

Figure 12 shows an analysis of the collection test suite. The analysis provides much information. For example, the delta time and end time are not consisting of integer only data type but also from string data type. Those parameters were likely from the string data type in some previous software versions. This information is significant, as it is possible to identify irregularities in the data before starting with the migration script. An example of a possible structure of the data in PostgreSQL is as follows.

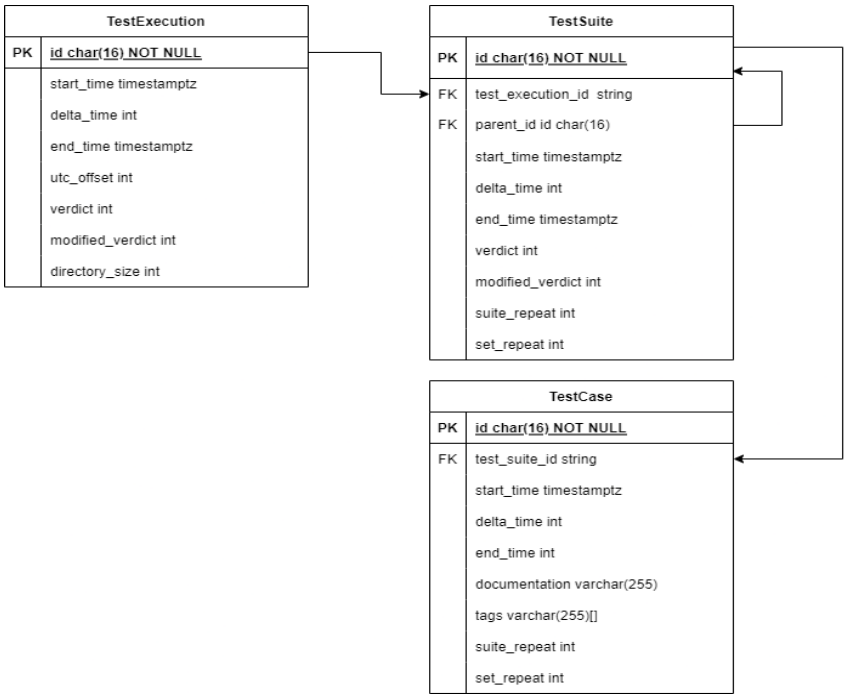


Figure 13. ERD of the Nokia's application sample data.

Figure 13 shows the proposed data structure of the migrated data. Creating the proposed data structure can be done using Prisma. Prisma is an ORM that supports the communications of different SQL databases, including PostgreSQL. Prisma simplifies the table creation process, utilizing its schema functionalities. Install Prisma in the project requires a few simple commands and to be in the backend folder of the project.

```
1. npm install Prisma
2. npx init prisma
```

Listing 8. Prisma installation and initialization. **(Prisma, n.d.: 16.)**

Code block 8, installs Prisma, installing Prisma creates a Prisma folder and env file. The Prisma folder contains a schema template. The template file fetches the variables from the env file to connect to the desired database, which the developer needs to specify in the env file. The schema file is where the developer must create the tables. Prisma's schema language is a simplified way to create tables in SQL. Each table is within curly brackets, and each row separated by a comma is equivalent to an entity in SQL. An example of a test execution schema is as follows.

```
 1. import mongoose from 'mongoose';
 2.
 3. const TestExecutionSchema = mongoose.Schema({
 4.   id: String,
 5.   start_time: {
 6.     type: Date,
 8.   },
 9.   end_time: {
10.     type: Date,
11.   },
13.   utc_offset: Number,
14.   verdict: Number,
15.   modified_verdict: String,
16.   directory_size: Number
17. })
18.
19.
20. const TestExecution = mongoose.model('test_execution',
TestExecutionSchema, 'test_execution')
21.
22. export default TestExecution
```

Listing 9. Test execution's schema.

Code block 9 shows the structure of a schema in Prisma. The schema uses the test execution table from the ERD. After finalizing the test suite and test cases schema, two more items are a must to create the tables in PostgreSQL. These items consist of having a functional PostgreSQL server and having the database created in the PostgreSQL server.

The PostgreSQL server can be installed locally or utilizing Docker or Kubernetes. For the sake of simplicity, the thesis is utilizing Docker to create the PostgreSQL server. The reason is that it is simple to configure the PostgreSQL server utilizing Docker. Also, it is efficient when it comes in terms of upgrading or wiping out the server. The docker image for the thesis is as follows.

```
1. FROM postgres:14
2.
3. ENV POSTGRES_USER=postgres \
4.   POSTGRES_PASSWORD=password
5.
6. COPY ./scripts/ /usr/local/bin/scripts/
7.
8. COPY ./configurations/db-config.sql /docker-entrypoint-initdb.d/
9.
10. EXPOSE 5432:5432
11.
12. RUN cd /usr/local/bin/scripts && \
13.   chmod 777 ./certs.sh && \
14.   ./certs.sh && \
15.   chown postgres:postgres /var/lib/certs/cert.crt
/var/lib/certs/priv_key &&\
16.   chmod 600 /var/lib/certs/cert.crt /var/lib/certs/priv_key && \
17.   ls -ll /var/lib/certs/
18.
19. USER postgres
20.
21. ENTRYPOINT [ "docker-entrypoint.sh" ]
22.
23. CMD [ "-c", "ssl=on" , "-c",
"ssl_cert_file=/var/lib/certs/cert.crt", "-c",
"ssl_key_file=/var/lib/certs/priv_key"]
```

Listing 10. Postgres Dockerfile.

Code block 10 shows the construction of the PostgreSQL docker image. The Postgres image uses the official Postgres image, which requires a user, a password, a port, and the docker-entrypoint script as the Dockerfile entry point. The Dockerfile also contains a line that copies a SQL query to docker-entrypoint-initdb, a unique directory that runs the query on container creation. The SQL query contains the query to create the database in which the tables will reside. The following commands are building and running the server.

```
1. docker build -t test .
2. docker run -d test:latest
3. docker exec –it <container name> /bin/bash
```

Listing 11. Building, running, and interacting in Docker.

Code block 11 initiates the PostgreSQL server and creates the database. Once the server is running, it would be possible to run the command to create the tables in the database through Prisma. The command is as follows.

```
1. npx prisma migrate dev
```

Listing 12. Initiate tables in PostgreSQL using Prisma.

After the initialization of the tables, it is possible to start working on the migration script and check the results on the server. The central concept of the migration script is having a clear idea of the relationships between the collections, data types, and possible irregularities of the parameter's data types. The creation of the migration script will consist of extensive experimentation, as the data type irregularities will affect the script creation. The following appendix 2 illustrates the mapping of test suite data.

Code block 12 shows the core ideas specified earlier, such as understanding the relationships, data types, and dealing with irregularities. The functions follow a simple logic consisting of, First, executing a query to fetch all the test executions ids and store them. Next, iterate through the data gathered from MongoDB's database and initialize a variable with a fallback value. Then, compare the stored execution id with each one of the data points and execute a simple SQL insert query with fallback values. If the comparison fails, it is most probable that the test suite has a test suite as its parent. That would be the second check in the iteration. If the test suite does not have both, it will not be stored in the database, as a test suite cannot exist without a parent. Test execution and test case follow a similar logic, although test execution can exist without a parent, so the mapping is straightforward.

The software requires writing the script in Python, as Python is executable across different operating systems, which would make the script work across different operating systems. After the execution of the script, the data from MongoDB will be present in PostgreSQL, as shown in the following figures.

```
postgres=# \c test_results
You are now connected to database "test_results" as user "postgres".
test_results=# select * from test_execution;
        id        |        start_time        | delta_time |        end_time        | utc_offset | verdict | modified_verdict | directory_size
------------------+--------------------------+------------+------------------------+------------+---------+------------------+----------------
 1ed59b94b07d620  | 2022-11-01 09:46:29+00 |          2 | 2022-11-01 09:46:31+00 |        120 |       1 |                0 |         476090
 1eba26f651f45f0  | 2021-04-21 10:02:20+00 |         53 | 2021-04-21 10:03:13+00 |        480 |       0 |                  |              0
 1ed59b90ec189e0  | 2022-11-01 09:44:50+00 |          2 | 2022-11-01 09:44:52+00 |        120 |       1 |                  |         476090
 1ed59b9abf0f2f0  | 2022-11-01 09:49:14+00 |          2 | 2022-11-01 09:49:16+00 |        120 |       1 |                  |         476090
 1ed59b9e4680ec0  | 2022-11-01 09:50:47+00 |          3 | 2022-11-01 09:50:50+00 |        120 |       1 |                  |         476092
 1ed59ba2edb8180  | 2022-11-01 09:52:52+00 |         33 | 2022-11-01 09:53:25+00 |        120 |       1 |                  |         475877
 1ed59bb16ef6ea0  | 2022-11-01 09:59:21+00 |          5 | 2022-11-01 09:59:26+00 |        120 |       1 |                  |         492609
 1ed59bb20293000  | 2022-11-01 09:59:36+00 |          6 | 2022-11-01 09:59:42+00 |        120 |       1 |                  |         492613
 1ed59bca4b78e10  | 2022-11-01 10:10:28+00 |        604 | 2022-11-01 10:20:32+00 |        120 |       1 |                  |         492699
 1ed59bd7d1bac50  | 2022-11-01 10:20:34+00 |         33 | 2022-11-01 10:21:07+00 |        120 |       1 |                  |         492529
 1ed59c005d85550  | 2022-11-01 10:34:40+00 |         33 | 2022-11-01 10:35:13+00 |        120 |       1 |                  |         492531
 1ed59f531f2f8e0  | 2022-11-01 16:55:20+00 |         16 | 2022-11-01 16:55:36+00 |        120 |       1 |                  |         476503
 1ed59f568307860  | 2022-11-01 16:56:48+00 |         15 | 2022-11-01 16:57:03+00 |        120 |       1 |                  |         476418
 1ed59f640a38ac0  | 2022-11-01 17:02:53+00 |          2 | 2022-11-01 17:02:55+00 |        120 |       1 |                  |         476048
 1ed5aa02367dd30  | 2022-11-02 13:18:59+00 |         17 | 2022-11-02 13:19:16+00 |        120 |       1 |                  |         476418
--More--
```

Figure 14. Test execution table in PostgreSQL.

```
test_results=# select * from test_suite;
        id        |        start_time        | delta_time |        end_time        | verdict | modified_verdict | suite_repeat | set_repeat |     parent_id     | test_execution_id
------------------+--------------------------+------------+------------------------+---------+------------------+--------------+------------+-------------------+------------------
 85561cb4         | 2021-04-21 10:02:25+00 |         47 | 2021-04-21 10:03:12+00 |       0 |                  |              |            |                   | 1eba26f651f45f0
 11332702         | 2022-11-01 09:44:51+00 |          1 | 2022-11-01 09:44:52+00 |       1 |                  |            1 |          1 |                   | 1ed59b90ec189e0
 4c264198         | 2022-11-01 09:46:29+00 |          2 | 2022-11-01 09:46:31+00 |       1 |                  |            1 |          1 |                   | 1ed59b94b07d620
 ae7b738c         | 2022-11-01 09:49:14+00 |          2 | 2022-11-01 09:49:16+00 |       1 |                  |            1 |          1 |                   | 1ed59b9abf0f2f0
 e655709e         | 2022-11-01 09:50:48+00 |          2 | 2022-11-01 09:50:50+00 |       1 |                  |            1 |          1 |                   | 1ed59b9e4680ec0
 3131cecc         | 2022-11-01 09:52:54+00 |         31 | 2022-11-01 09:53:25+00 |       1 |                  |            1 |          1 |                   | 1ed59ba2edb8180
 18b9a5e8         | 2022-11-01 09:59:22+00 |          4 | 2022-11-01 09:59:26+00 |       1 |                  |              |            |                   | 1ed59bb16ef6ea0
 21ed9998         | 2022-11-01 09:59:38+00 |          4 | 2022-11-01 09:59:42+00 |       1 |                  |              |            |                   | 1ed59bb20293000
 a67c8f0a         | 2022-11-01 10:10:30+00 |        602 | 2022-11-01 10:20:32+00 |       1 |                  |              |            |                   | 1ed59bca4b78e10
 f2d4506          | 2022-11-01 10:20:35+00 |         32 | 2022-11-01 10:21:07+00 |       1 |                  |              |            |                   | 1ed59bd7d1bac50
 7cd2014          | 2022-11-01 10:34:41+00 |         32 | 2022-11-01 10:35:13+00 |       1 |                  |              |            |                   | 1ed59c005d85550
 351b8378         | 2022-11-01 16:55:21+00 |         15 | 2022-11-01 16:55:36+00 |       1 |                  |            1 |          1 |                   | 1ed59f531f2f8e0
 6942381a         | 2022-11-01 16:56:48+00 |         15 | 2022-11-01 16:57:03+00 |       1 |                  |            1 |          1 |                   | 1ed59f568307860
 4325a222         | 2022-11-01 17:02:54+00 |          1 | 2022-11-01 17:02:55+00 |       1 |                  |            1 |          1 |                   | 1ed59f640a38ac0
 263dd400         | 2022-11-02 13:19:00+00 |         16 | 2022-11-02 13:19:16+00 |       1 |                  |            1 |          1 |                   | 1ed5aa02367dd30
 f05c0e74         | 2022-11-07 16:10:31+00 |          5 | 2022-11-07 16:10:36+00 |       1 |                  |            1 |          1 |                   | 1ed5ea5ed6e0a60
 f5eb850a         | 2022-11-07 16:10:40+00 |          6 | 2022-11-07 16:10:46+00 |       1 |                  |            1 |          1 |                   | 1ed5ea5f507fe70
 1198a576         | 2022-11-07 16:11:27+00 |         50 | 2022-11-07 16:12:17+00 |       1 |                  |            1 |          1 |                   | 1ed5ea610bda700
 94b7a862         | 2022-11-07 16:15:07+00 |         50 | 2022-11-07 16:15:57+00 |       1 |                  |            1 |          1 |                   | 1ed5ea69394d4a0
 b3583588         | 2022-11-07 16:15:58+00 |         50 | 2022-11-07 16:16:48+00 |       1 |                  |            1 |          1 |                   | 1ed5ea69ef93e30
 def7647a         | 2022-11-07 16:17:11+00 |         51 | 2022-11-07 16:18:02+00 |       1 |                  |            1 |          1 |                   | 1ed5ea6ddc2d9f0
 fda359ee         | 2022-11-07 16:18:03+00 |         50 | 2022-11-07 16:18:53+00 |       1 |                  |            1 |          1 |                   | 1ed5ea6f5ce1820
 bec4c37a         | 2022-11-08 19:14:03+00 |         51 | 2022-11-08 19:14:54+00 |       1 |                  |            1 |          1 |                   | 1ed5f88bc2ba000
 f94a3986         | 2022-11-08 19:15:42+00 |         50 | 2022-11-08 19:16:32+00 |       1 |                  |            1 |          1 |                   | 1ed5f88f84ab530
 1800a776         | 2022-11-08 19:16:33+00 |         51 | 2022-11-08 19:17:24+00 |       1 |                  |            1 |          1 |                   | 1ed5f88fc299d10
```

Figure 15. Test suite table in PostgreSQL.

4.3 PostgreSQL

PostgreSQL provides various features. The features consist of ways to enhance efficiency within the database, moreover, methods for analysing the existing queries to optimize them. Furthermore, PostgreSQL provides tools to harden the security of the database. These features provide powerful tools for developers to create a maintainable, scalable, and secure database.

4.3.1 PostgreSQL Optimization

PostgreSQL optimization consists of analysing queries and optimizing based on the analysis results. PostgreSQL provides some methods to analyse a query and comprehensive documentation of how to read the analysis. To run the analysis, use the keyword EXPLAIN or EXPLAIN ANALYZE before the query that requires optimization.

The EXPLAIN ANALYZE provides better accuracy of the query analysis. The query analysis specifies how the query planner is planning the execution of a query and then provides the execution time of the query. An example of a query analysis is as follows.

```
test_results=# EXPLAIN ANALYZE  SELECT tc.* FROM test_execution tc JOIN test_suite ts ON ts.test_execution_id = tc.id
;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=5.76..21.40 rows=446 width=53) (actual time=0.243..0.479 rows=167 loops=1)
   Hash Cond: (ts.test_execution_id = tc.id)
   ->  Seq Scan on test_suite ts  (cost=0.00..14.46 rows=446 width=17) (actual time=0.060..0.251 rows=446 loops=1)
   ->  Hash  (cost=3.67..3.67 rows=167 width=53) (actual time=0.133..0.134 rows=167 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 24kB
         ->  Seq Scan on test_execution tc  (cost=0.00..3.67 rows=167 width=53) (actual time=0.010..0.023 rows=167 loops=1)
 Planning Time: 9.023 ms
 Execution Time: 0.573 ms
(8 rows)
```

Figure 16. Output of EXPLAIN ANALYZE in PostgreSQL.

Figure 16 shows the analysis of joining between the tables of test execution and test suite, filtering out the rows where test execution is not the parent of a test suite. The analysis provides beneficial insights into the cost of each query stage and what kind of algorithm the query planner utilizes. This tool may assist in optimizing different queries that software utilizes.

4.3.2 PostgreSQL Features

The thesis provides an insight into the view and materialized view creation. These are primary cases where it is possible to create a view and some considerations that a developer should consider before engaging with the view or materialized view feature. To create a view, utilize the following command.

```
1. CREATE VIEW table_case_view AS SELECT * FROM test_case;
```

Listing 13. Create view example in PostgreSQL.

Views are beneficial when a table requires abstractions, so fields would be easy to add and remove without modifying the underlying schema. Moreover, views can provide higher efficiency for complex joins, as the complex queries would not require executing on multiple occasions. Furthermore, views assist with implementing backward compatibility, as views may present no changes in their table while the underlying schema is changed. An example of a view is as follows.

```
test_results=# select * from table_case_view;
      id        |       start_time       | delta_time |        end_time        | verdict | modified_verdict | documentation | tags | suite_repeat | set_repeat | testExecutionId |  test_suite_id
----------------+------------------------+------------+------------------------+---------+------------------+---------------+------+--------------+------------+-----------------+----------------
 1134fb9c       | 2022-11-01 09:44:51+00 |          1 | 2022-11-01 09:44:52+00 |       1 |                  |               |      |              |            |                 | 11332702
 4c26fc9e       | 2022-11-01 09:46:30+00 |          1 | 2022-11-01 09:46:31+00 |       1 |                  |               |      |              |            |                 | 4c264198
 ae7c34c8       | 2022-11-01 09:49:15+00 |          1 | 2022-11-01 09:49:16+00 |       1 |                  |               |      |              |            |                 | ae7b738c
 e65989f4       | 2022-11-01 09:50:48+00 |          2 | 2022-11-01 09:50:50+00 |       1 |                  |               |      |              |            |                 | e655709e
```

Figure 17. Sample of the view table.

Figure 17 shows a simple case of the test case view table, which represents the data in the test case table. Views are beneficial for software. However, it is crucial to understand when it is beneficial to use a view. Understanding the expectations of the queries in the software would make it easier to pinpoint where a view might be relevant. Views present some drawbacks, such as losing the information about relationships between tables, and it needs to be apparent when using view would be more beneficial than harmful.

On the other hand, there are materialized views, which can be beneficial with data that does not require constant updating. Materialized views are similar to views in their concepts. However, materialized views do not update the data unless requested. Materialized views can improve performance when querying for extensive data or complex queries. Creating a materialized view is as follows.

```
1. CREATE MATERIALIZED VIEW table_execution_view AS
2. SELECT * FROM test_execution;
```

Listing 14. Commands to create materialized view in PostgreSQL.

Similarly, to views, materialized views should not be utilized if not necessary. It is crucial to fully understand the software architecture to know when having a materialized view would be beneficial. Materialized views might be beneficial in cases where the query would present the result set, storing the results of a complex query. An example of the materialized view is as follows.

```
test_results=# select * from table_execution_view;
        id        |       start_time       | delta_time |        end_time        | utc_offset | verdict | modified_verdict | directory_size
------------------+------------------------+------------+------------------------+------------+---------+------------------+----------------
 1ed59b94b07d620  | 2022-11-01 09:46:29+00 |          2 | 2022-11-01 09:46:31+00 |        120 |       1 |                0 |         476090
 1eba26f651f45f0  | 2021-04-21 10:02:20+00 |         53 | 2021-04-21 10:03:13+00 |        480 |       0 |                  |              0
 1ed59b90ec189e0  | 2022-11-01 09:44:50+00 |          2 | 2022-11-01 09:44:52+00 |        120 |       1 |                  |         476090
 1ed59b9abf0f2f0  | 2022-11-01 09:49:14+00 |          2 | 2022-11-01 09:49:16+00 |        120 |       1 |                  |         476090
 1ed59b9e4680ec0  | 2022-11-01 09:50:47+00 |          3 | 2022-11-01 09:50:50+00 |        120 |       1 |                  |         476092
 1ed59ba2edb8180  | 2022-11-01 09:52:52+00 |         33 | 2022-11-01 09:53:25+00 |        120 |       1 |                  |         475877
 1ed59bb16ef6ea0  | 2022-11-01 09:59:21+00 |          5 | 2022-11-01 09:59:26+00 |        120 |       1 |                  |         492609
```

Figure 18. Sample of materialized view.

4.3.3 PostgreSQL Security

SSL/TLS encrypts the data in data transmission, providing a layer of security to the database communication. SSL/TLS can use asymmetric or symmetric cryptography. In the thesis implementation, PostgreSQL utilizes the asymmetric strategy, which consists of using a public key and a private key. The public key oversees the data encryption, which can be decrypted only by the private key.

The implementation for utilizing SSL/TLS certificates in the PostgreSQL server served in Docker consists of creating the certificates on container creation using the OpenSSL package in Linux. OpenSSL provides a method to create a certificate and a private key based on a configuration file, which includes the necessary fields for the certificate and private key to work as a valid SSL/TLS certificate. The configuration is as follows.

```
[req]
default_bits = 2048
encrypt_key = no
default_md = sha256
default_keyfile = priv_key
distinguished_name = req_distinguished_name
prompt = no
x509_extensions = x509_extensions
[req_distinguished_name]
C = FI
ST = Uusima
L = Espoo
O = N/A
OU = N/A
CN = localhost
[x509_extensions]
subjectAltName = @alternate_names
keyUsage=digitalSignature,keyAgreement,keyEncipherment
extendedKeyUsage=serverAuth,clientAuth
[alternate_names]
DNS.1 = localhost
DNS.2 = 127.0.0.1
IP.1 = 127.0.0.1
```

Listing 15. Configuration settings of the SSL/TLS certificates.

Code block 15 shows the configuration for the OpenSSL method. It is crucial to address a few fields from the configuration, such as default bits, distinguished_name, and x509_exetensions. The default number of bits should be at least 2048 as a base requirement for asymmetric cryptographic strategy. Next is the distinguished name parameter, which is the origin of the certificate creation. Finally, x509 extensions

specify the DNS address, IP, and methods the certificate should follow. The script to create the certificates is as follows.

```
1. #!/bin/bash
2.
3. openssl req -config ./config.txt -newkey rsa -x509 -days 365 -out
./cert.crt
4. cp ./cert.crt ./root.crt
5. mkdir /var/lib/certs
6. mv -t /var/lib/certs ./cert.crt ./root.crt ./priv_key
7. echo "Done"
```

Listing 16. Certificate generation script.

Code block 16 generates the certificate. The script runs as part of the Dockerfile, so the certificate will exist when PostgreSQL tries to fetch the certificate. The initialization of PostgreSQL occurs in the docker-entrypoint script. After the ENTRYPOINT keyword, it is possible to specify to PostgreSQL to modify its configuration as the following code block shows.

```
1. ENTRYPOINT [ "docker-entrypoint.sh" ]
2.
3. CMD [ "-c", "ssl=on" , "-c", "ssl_cert_file=/var/lib/certs/cert.crt",
"-c", "ssl_key_file=/var/lib/certs/priv_key"]
```

Listing 17. Docker entrypoint and command Keywords example.

The certificate acts as the public key in the asymmetric cryptographic strategy. The private key decrypts the data which the certificate encrypts. As code block 17 shows, the CMD keyword executes the command, and the command specifies the use of SSL/TLS and the path toward the private key and the certificate. A service or software using the certificate would have its data encrypted in the communication between the service and PostgreSQL. Once PostgreSQL takes into action the certificate and the private key, all the services that want to interact with the PostgreSQL server will require the certificate.

The second component in the security hardening of PostgreSQL is audit logging. Audit logging refers to logging the activity occurring on the PostgreSQL server, such as logging in, logging out, queries, abnormalities, fatal errors, and warnings. The developer can specify what to log, how extensive the logs should be, and how to rotate

of the logging. For example, the following code block specifies many different functionalities for logging.

```
1. ALTER SYSTEM SET log_statement TO 'all';
2. ALTER SYSTEM SET log_destination TO 'stderr';
3. ALTER SYSTEM SET logging_collector TO on;
4. ALTER SYSTEM SET log_directory TO '/var/log/postgresql/';
5. ALTER SYSTEM SET log_filename TO 'postgresql-%Y-%m-%d_%H%M%S.log';
6. ALTER SYSTEM SET log_rotation_size TO '100MB';
7. ALTER SYSTEM SET log_min_duration_statement TO -1;
8. ALTER SYSTEM SET log_min_duration_sample TO -1;
9. ALTER SYSTEM SET log_statement_sample_rate TO 0.0;
10. ALTER SYSTEM SET log_transaction_sample_rate TO 0.0;
11. ALTER SYSTEM SET debug_print_parse  TO off;
12. ALTER SYSTEM SET debug_print_rewritten TO off;
13. ALTER SYSTEM SET debug_print_plan  TO off;
14. ALTER SYSTEM SET debug_pretty_print TO on;
15. ALTER SYSTEM SET log_autovacuum_min_duration TO -1;
16. ALTER SYSTEM SET log_checkpoints  TO off;
17. ALTER SYSTEM SET log_connections TO on;
18. ALTER SYSTEM SET log_disconnections  TO on;
19. ALTER SYSTEM SET log_duration TO off;
20. ALTER SYSTEM SET log_error_verbosity TO 'terse';
21. ALTER SYSTEM SET log_line_prefix TO '%t [%d-%u] ';
```

Listing 18. Setting up logs in PostgreSQL.

The functionalities that code block eighteen shows are in PostgreSQL's documentation. The functionalities provide some insight into the server's activity, such as who is logging into the server, who is logging out, and what queries the user queried. Additionally, it specifies that if the logs reach 100 Megabytes, it should create a new log file and continue logging there. A sample of the logging is as follows.

```
2023-04-30 12:24:18 UTC [-] LOG:  database system was interrupted; last known up at 2023-04-24 17:53:1
2 UTC
2023-04-30 12:24:19 UTC [-] LOG:  database system was not properly shut down; automatic recovery in pr
ogress
2023-04-30 12:24:19 UTC [-] LOG:  redo starts at 0/1FAB088
2023-04-30 12:24:19 UTC [-] LOG:  invalid record length at 0/1FAB170: wanted 24, got 0
2023-04-30 12:24:19 UTC [-] LOG:  redo done at 0/1FAB138 system usage: CPU: user: 0.00 s, system: 0.00
 s, elapsed: 0.00 s
2023-04-30 12:24:19 UTC [-] LOG:  database system is ready to accept connections
2023-04-30 12:24:32 UTC [[unknown]-[unknown]] LOG:  connection received: host=[local]
2023-04-30 12:24:32 UTC [postgres-postgres] LOG:  connection authorized: user=postgres database=postgr
es application_name=psql
2023-04-30 12:24:34 UTC [postgres-postgres] LOG:  statement: SELECT pg_catalog.quote_ident(datname) FR
OM pg_catalog.pg_database  WHERE substring(pg_catalog.quote_ident(datname),1,3)='tes'
        LIMIT 1000
2023-04-30 12:24:35 UTC [[unknown]-[unknown]] LOG:  connection received: host=[local]
2023-04-30 12:24:35 UTC [test_results-postgres] LOG:  connection authorized: user=postgres database=te
st_results application_name=psql
2023-04-30 12:24:35 UTC [postgres-postgres] LOG:  disconnection: session time: 0:00:02.571 user=postgr
es database=postgres host=[local]
2023-04-30 12:24:43 UTC [test_results-postgres] LOG:  statement: select * from test_executon;
2023-04-30 12:24:43 UTC [test_results-postgres] ERROR:  relation "test_executon" does not exist at cha
racter 15
2023-04-30 12:24:43 UTC [test_results-postgres] STATEMENT:  select * from test_executon;
2023-04-30 12:24:50 UTC [test_results-postgres] LOG:  statement: select * from test_execution;
2023-04-30 12:25:04 UTC [test_results-postgres] LOG:  statement: select * from test_suite;
2023-04-30 13:08:00 UTC [test_results-postgres] LOG:  statement: EXPLAIN ANALYZE  SELECT tc.* FROM tes
t_execution tc JOIN test_suite ts ON ts.test_execution_id = tc.id;
2023-04-30 19:05:51 UTC [test_results-postgres] LOG:  disconnection: session time: 6:41:16.479 user=po
stgres database=test_results host=[local]
```

Figure 19. Logs output sample.

Figure 19 shows a sample of the activity in the PostgreSQL server. Although the logs can provide insight into abnormalities in the server, they can also be damaging. If an attacker gets a hold of the logs, the attacker might be able to build up the software's data structure and user information, such as the username and the IP. Hence, it is crucial to understand what should be in the logs and keep them in a protected area with limited access.

5 Conclusion and Discussion

Migrating data from NoSQL to SQL requires extensive research, as many factors depend on it. Firstly, it requires understanding the data flow of the NoSQL database and identifying possible relationships, noting each parameter's data type, and finding possible data types irregularities. Secondly, identifying possible flaws in the approach to the NoSQL data structure may assist in creating the new data structure for SQL or ease up the process of identifying abnormalities and possible relationships in the database. Thirdly, when creating the SQL database, it is crucial to consider many factors, such as security, performance, limitations, and the software architecture for query building.

MongoDB shows some limitations when it comes to its licensing and its compatibility with different tools. These aspects are possible risks for the productization of the software. Additionally, it is essential to note that MongoDB's module for Node.js is also limited in its functionality, and there is a more efficient which is Mongoose. Mongoose provides all the methods that MongoDB's module provides and more. Moreover, Mongoose may assist in identifying possible flaws and a more efficient manner to interact with the data.

Migrating the data will require extensive experimentation, as dealing with large data sets is complex and bound to have abnormalities to some extent. Moreover, some changes in the data structure require handling the data in a challenging manner. Furthermore, the more significant number of relationships across collections increase the script's complexity.

PostgreSQL provides many features. Some features may enhance the performance of the software, and it is possible to analyse the performance by utilizing PostgreSQL's tools. At the same time, other features, such as audit logging and TLS/SSL, may

enhance database security. However, it requires an extensive understanding of the software architecture to utilize the features efficiently. Otherwise, it is possible to find these features to be more harmful than helpful.

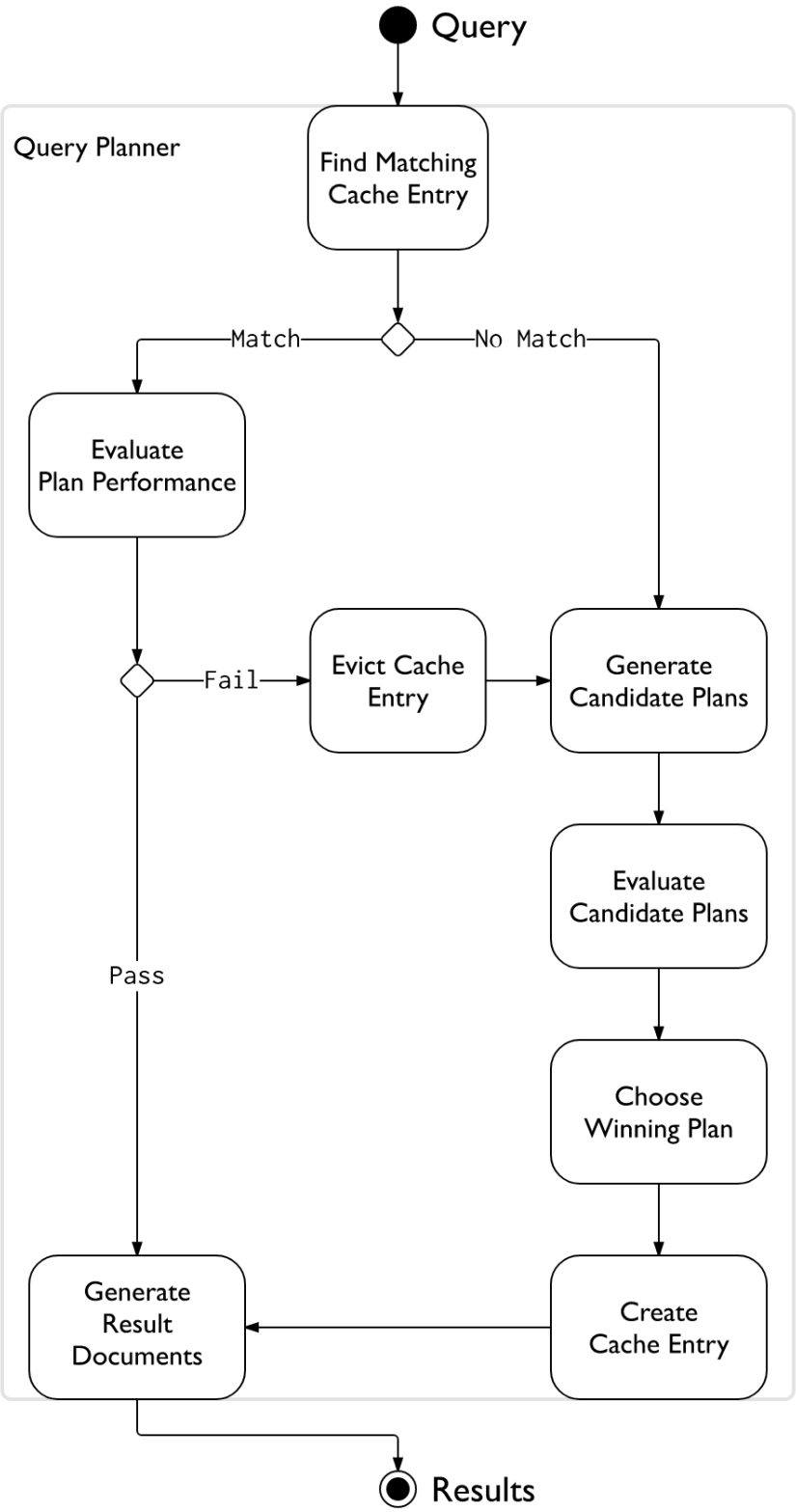
6 References

1. Edward, S.G. and Sabharwal, N. (2015). Practical MongoDB: Architecting, Developing, and Administering MongoDB. 1st ed. [online] Apress, pp.5–100, 160–230. Available at: https://learning.oreilly.com/library/view/practical-mongodb-architecting/9781484206478/ACoverHTML.html [Accessed 5 May 2023].
2. MongoDB (n.d.). Data Model Design. [online] MongoDB. Available at: https://www.mongodb.com/docs/manual/core/data-model-design/ [Accessed 5 May 2023].
3. Giamas, A. (2017). Mastering MongoDB 3.x. 1st ed. [online] Packt Publishing, pp.5–230. Available at: https://learning.oreilly.com/library/view/mastering-mongodb-3-x/9781783982608/ [Accessed 5 May 2023].
4. Holmes, S. (2013). Mongoose for Application Development. 1st ed. [online] Packt Publishing, pp.5–142. Available at: https://learning.oreilly.com/library/view/mongoose-for-application/9781782168195/ [Accessed 5 May 2023].
5. Mongoose (n.d.). Schemas. [online] MongooseJS. Available at: https://mongoosejs.com/docs/guide.html [Accessed 5 May 2023].
6. Mongoose (n.d.). Queries. [online] MongooseJS. Available at: https://mongoosejs.com/docs/queries.html [Accessed 5 May 2023].
7. MongoDB (n.d.). Indexing Strategies. [online] MongoDB. Available at: https://www.mongodb.com/docs/v3.6/applications/indexes/ [Accessed 5 May 2023].
8. MongoDB (n.d.). Create Indexes to Support Your Queries. [online] MongoDB. Available at: https://www.mongodb.com/docs/v3.6/tutorial/create-indexes-to-support-queries/ [Accessed 5 May 2023].

9. MongoDB (n.d.). Remove Unnecessary Indexes. [online] MongoDB. Available at: https://www.mongodb.com/docs/atlas/schema-suggestions/too-many-indexes/ [Accessed 5 May 2023].
10. MongoDB (n.d.). Explain Results. [online] MongoDB. Available at: https://www.mongodb.com/docs/v3.6/reference/explain-results/ [Accessed 5 May 2023].
11. MongoDB (n.d.). Query Plans. [online] https://www.mongodb.com/docs/v3.6/core/query-plans/. Available at: https://www.mongodb.com/docs/v3.6/core/query-plans/ [Accessed 5 May 2023].
12. Hausenblas, M. (2022). Learning Modern Linux. 1st ed. [online] O'Reilly Media, Inc., pp.5–36. Available at: https://learning.oreilly.com/library/view/learning-modern-linux/9781098108939/ [Accessed 5 May 2023].
13. van Rossum, G. (n.d.). What is Python? Executive Summary. [online] Python. Available at: https://www.python.org/doc/essays/blurb/ [Accessed 5 May 2023].
14. Pymongo (n.d.). Tutorial. [online] pymongo.readthedocts.io. Available at: https://pymongo.readthedocs.io/en/stable/tutorial.html# [Accessed 5 May 2023].
15. MongoDB (n.d.). What is MongoDB Compass? [online] MongoDB. Available at: https://www.mongodb.com/docs/compass/current/ [Accessed 5 May 2023].
16. Prisma (2023). What is Prisma? [online] Prisma. Available at: https://www.prisma.io/docs/concepts/overview/what-is-prisma [Accessed 5 May 2023].
17. Prisma (2023). Prisma schema. [online] Prisma. Available at: https://www.prisma.io/docs/concepts/components/prisma-schema [Accessed 5 May 2023].
18. Di Gregorio, F. and Varrazzo, D. (n.d.). Psycopg – PostgreSQL database adapter for Python. [online] psycopg. Available at: https://www.psycopg.org/docs/index.html [Accessed 5 May 2023].

19. Dombrovskaya, H.D., Novikov, B. and Bailliekova, A. (2021). PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries. 1st ed. [online] Apress, pp.5–87. Available at: https://learning.oreilly.com/library/view/postgresql-query-optimization/9781484268858/ [Accessed 5 May 2023].
20. The PostgreSQL Global Development Group (2023). PostgreSQL 15.2 Documentation. [online] The PostgreSQL Global Development Group. Available at: https://www.postgresql.org/docs/15/index.html [Accessed 5 May 2023].
21. MongoDB (n.d.). Index Build Operations on a Populated Collection. [online] MongoDB. Available at: https://www.mongodb.com/docs/v3.6/applications/indexes/ [Accessed 5 May 2023].
22. Hoffman, A. (2020). Web Application Security. 1st ed. [online] O'Reilly Media, Inc., pp.150–250. Available at: https://learning.oreilly.com/library/view/web-application-security/9781492053101/ [Accessed 5 May 2023].
23. Server-Side Public License (SSPL). Available at: https://www.mongodb.com/licensing/server-side-public-license [Accessed 5 May 2023].
24. MongoDB (n.d.). MongoDB Software Lifecycle Schedules. [online] MongoDB. Available at: https://www.mongodb.com/support-policy/lifecycles [Accessed 5 May 2023].

7 Appendix



Appendix 1. Query planner workflow. **(MongoDB, n.d.: 11).**

```
1. def test_suite_mapping():
 2.   cur = conn.cursor()
 3.
 4.   print("Starting test suites mapping")
 5.   cur.execute("SELECT id FROM test_execution")
 6.   test_executions = cur.fetchall()
 7.   test_execution_ids = [str(test_execution[0]).rstrip() for
test_execution in test_executions]
 8.
 9.   for test_suite in db_test_results.test_suite.find():
10.     test_suite_id = test_suite["id"]
11.     start_time = test_suite.get("start_time", "NULL")
12.     end_time = test_suite.get("end_time", "NULL")
13.     delta_time = test_suite.get("delta_time", 0)
14.     suite_repeat = '' if test_suite.get("suite_repeat") in [None,
""] else test_suite["suite_repeat"]
15.     set_repeat = '' if test_suite.get("set_repeat") in [None, ""]
else test_suite["set_repeat"]
16.     verdict = test_suite.get("verdict", -2)
17.     modified_verdict = test_suite.get("modified_verdict", "NULL")
18.
19.     if delta_time == '':
20.       delta_time = 0
21.
22.     if modified_verdict == None:
23.       modified_verdict = ""
24.
25.     if test_suite.get("parent_execution_id") is not None:
26.       if test_suite["parent_execution_id"] in test_execution_ids:
27.         try:
28.           cur.execute(
29.             """ INSERT INTO public.test_suite(
30.               id, test_execution_id, start_time, delta_time,
end_time, suite_repeat, set_repeat, verdict, modified_verdict
31.             ) VALUES (
32.               '{0}', '{1}', '{2}', NULLIF('{3}',''):: INTEGER,
'{4}', NULLIF('{5}',''):: INTEGER, NULLIF('{6}',''):: INTEGER,
NULLIF('{7}',''):: INTEGER, NULLIF('{8}',''):: INTEGER
33.             );
34.             """.format(
35.               test_suite_id,
36.               test_suite["parent_execution_id"],
37.               datetime.fromtimestamp(start_time),
38.               delta_time,
39.               datetime.fromtimestamp(end_time),
40.               suite_repeat,
41.               set_repeat,
42.               verdict,
43.               modified_verdict
44.             )
45.           )
46.         except Exception as e:
47.           print(e)
```

Appendix 2. Sample of test suite mapping from the migration script.