FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Modern NoSQL databases research
Student:	Bc. Serhii Holovko
Supervisor:	Ing. Jiří Hunka
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

The goal of this thesis is to familiarize the reader with modern NoSQL approaches focusing on document-oriented and graph database technologies. The study has to provide a real-world demonstration of using NoSQL databases. The resulting application has to be a web platform that makes developers able to manage data through a graphic interface.

Steps to cover:
Analysis of classic RDBMS approaches.
Inspecting generic NoSQL paradigms.
Evaluating document-oriented and graph databases.
Comparing these two approaches.
Design the web platform using document-oriented and graph databases.
Implementing the example application.
Design possible improvements and investigate the potential growth of the resulting product

---

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 3 February 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Database as a service platform using modern architectural techniques

Bc. Serhii Holovko

Department of Web and Software Engineering, Web Engineering specialization
Supervisor: Ing. Jiří Hunka

June 23, 2022

Acknowledgements

I would like to thank my supervisor Ing. Jiří Hunka who helped with everything I needed regarding the thesis and also for the psychological support from him. Also thank you from the bottom of my heart to all people who are defending Ukraine from the russian aggression. This allows me to work, learn and develop myself in a peaceful Europe.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on June 23, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Holovko, Serhii. *Database as a service platform using modern architectural techniques.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Tato práce popisuje hlavní aspekty volby databázové technologie. Vysvětluje se teorie řízení dat, moderní databázová řešení jsou analyzované. Ukázková aplikace demonstruje progresivní databázové stroje v kontextu reálného vývoje, také poskytuje platformu pro použití grafů a dokumentě-orientované technologie jako služby.

Klíčová slova databáze, SQL, RDBMS, NoSQL, Elasticsearch, Neo4j, datový stroj, moderní databáze

Abstract

This thesis describes main aspects of choosing a database technology. Data management theory is described. Modern database solutions are analyzed. The example application demonstrates cutting-edge database engines in the context of a real-world development, providing a platform for using graph and document-base technologies in “as a service” way.

Keywords database, SQL, RDBMS, NoSQL, Elasticsearch, Neo4j, data engine, modern databases

Contents

List of Figures

List of Tables

Introduction

It is almost impossible to imagine a modern web application which would not manage any data about its state using some type of a data engine. In the world of social networks, continuous developing new software products and occurrence of such giant of information systems as LinkedIn, Facebook, Uber, Bloomberg etc. it becomes critically important to choose an appropriate technology to make a result application able to satisfy all technical requirements. As a consequence of such problems existence a sphere of modern and cutting-edge data technologies began to arise what led to the appearance of innovative database techniques.

This thesis is considered to be a complete source of the knowledge base for people interested in contemporary database technologies and paradigms. Problems discussed below are not strongly related to specific implementations what makes the thesis useful for almost all developers, however some details and comparison of solution patterns are described based on specific data engines. Concerning the real-world usage of this work, the result application which has to facilitate a data management process is fully implemented and ready to be used by engineers.

The theme of this thesis was motivated by the reason of the fact that it is still fairly hard for developers to choose an appropriate data management system even considering that many engineers are already using modern software solutions. Moreover, such issue as a database impropriety for an information system may arise in the middle of a project by cause of new requirements. Therefore it is definitely transparent that choosing a proper data engine is a comprehensive and significant task.

The aim of the work is to accomplish a complete research in the context of NoSQL models as well as their specific implementations. Being aware of contemporary database technologies has 2 consequences. The first one is that a reader becomes competent enough to potentially analyze his development requirements and choose the most suitable database. The second benefit of having a solid knowledge in the NoSQL field brings in the possibility to ap-

preciate all advantages of the data management platform implemented in the scope of this thesis.

The thesis content consists of two huge parts, the first one describes contemporary database paradigms in theoretical way, while the second demonstrates the process of using specific data engines. First of all the theoretical material explains concepts of relational database approach, then problems which the NoSQL field faces are demonstrated. After becoming acquainted with the database world issues, 4 non-traditional models are explained and their representatives are introduced. Elasticsearch and Neo4j databases are clarified afterwards. The second part of the thesis illustrates the process of implementing a web application using graph and document technologies. The final product's benefits and potential improvements are also discussed.

Chapter 1

Relational Database Management System

In this section the classic relational database approach is explained. The purpose of this chapter is not to dive deeply into the context of RDBMS but to review the main strengths and weaknesses of the traditional data engines design pattern. The awareness of the traditional database design approach makes a reader possible to determine if it makes sense to build an application based on NoSQL technologies or it is rather wiser to choose a relational technique.

1.1 Historical context

In 1970's the world of database management systems was almost blank. Up to that time only several IBM frameworks were developed. In 1970 Edgar Codd presented a revolutionary concept which started the epoch of relational databases [1]. His theory consisted in perception a database engine on a high level of abstraction. Thus users did not have to understand the computer structure for managing data [2].

In the web sphere the RDBMS technique became a truly innovative approach in generating dynamic websites and users' interaction with HTML forms. During the first era of World Wide Web pages were mostly either static or generated using files stored on the operation system of a server [1]. Therefore rising of such a significant technique of storing data became a real breakthrough in the Internet world.

The impact of relational ideas on the software community was extremely wide. Nowadays data engineers debate about weaknesses of RDBMS and how to cope with them using NoSQL approaches. Nevertheless, dealing with RDBMS based projects is still very common. This fact emphasizes benefits of relational data approach which are described in the next section.

1.2 Relational databases properties

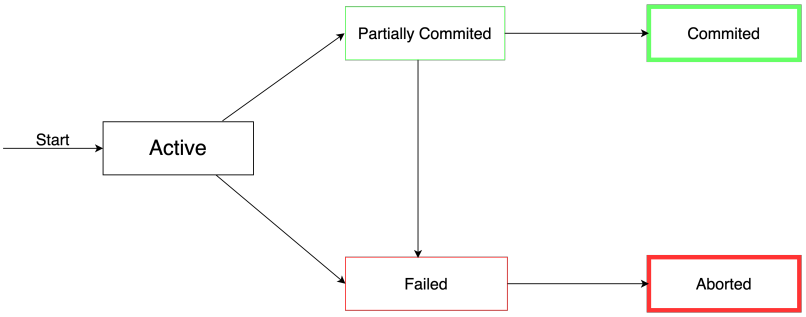

Figure 1.1: Transaction states

The most fundamental relational database principles are concurrency control and error recovery. In this section, these concepts are explained and related terms such as transaction, ACID properties, and log file are described.

1.2.1 Transaction

The term transaction refers to a sequence of database operations. Such a set of actions has one essential feature. They are not executed separately, the collection is completed as a whole or not executed at all. This characteristic leads to the fact that a transaction may result only in two states: committed or aborted [3].

In case of a successful scenario, the transaction starts in the state called Active, then after all operations are performed a transaction moves to the Partially Committed state. Only after that, the final Committed point can be achieved which denotes that a whole set of database activities is confirmed. However, a Partially Committed state may also lead to a Failed state which is depicted in Figure 1.1. Moreover, a transaction may even transfer directly from Active to Failed state and then obviously to Aborted [3].

The most important collection of transaction features is called ACID. This is an abbreviation of such characteristics:

- Atomicity
- Consistency

- Isolation
- Durability

Atomicity reflects the idea that a transaction can be executed either as a whole or not executed at all. This concept can be observed while inspecting how relational databases achieve error recovery. In case of a data or system failure during transaction execution, a ROLLBACK operation is performed and all uncommitted changes do not persist [3].

Before describing a second feature the "consistent data" term has to be explained. Data consistency means that data in a database do not violate integrity constraints defined by a schema. That leads to the RDBMS concept called Consistency which declares that both database states before and after any transaction have to be consistent. Nevertheless, this characteristic does not restrict data to fulfill integrity constraints while a transaction is being executed [4].

Isolation Level	Dirty read	Non-repeatable read	Phantom Read
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Table 1.1: Isolation levels table

A third property from the ACID collection is called Isolation. This characteristic claims that a transaction during its execution does not see uncommitted changes caused by another transaction that modifies data simultaneously [4]. Moreover, relational databases make users able to establish something called isolation level. Level 0 is called READ UNCOMMITTED and makes a transaction able to read uncommitted (dirty) data which may be rollbacked. Such property ensures no transaction isolation, however, has a positive impact on performance characteristics. The next level 1 is READ COMMITTED, it provides transactions a constraint to read-only confirmed data. The next isolation mode is REPEATABLE READ, it ensures the previous READ COMMITTED restriction and denies such aspects as non-repeatable read. This means that during a transaction row values retrieved twice can not differ even in case of new data were committed. Another level of isolation is SERIALIZABLE. This is the strictest transaction mode, it collects all REPEATABLE READ restrictions and makes it impossible to occur a phantom read anomaly. Such constraint means that collections of rows retrieved in two points of time are identical. It may seem that the SERIALIZABLE

level is the best choice, however, performance drawbacks have to be taken into consideration. The stricter isolation rules are, the less efficiently the database engine operates [5]. The table presentation of all isolation levels and their characteristics are depicted in Table 1.1.

The last RDBMS property is Durability. This characteristic ensures that as soon as a transaction is committed, its impact on data is persisted even in case of any kind of failure. Durability feature is provided by using the technique of transaction log file. When a transaction is confirmed, its change vector is stored in the secondary memory [4]. Such a concept also ensures the Atomicity feature which we have discussed before.

1.2.2 Concurrency control

A relational data engine does not handle requests sequentially, transaction operations are often interleaved. Consecutive behavior would lead to performance inefficiency. For instance, when transaction A is loading a data block from the secondary memory, transaction B can be executed without a need to wait for transaction A. Another example of sequential drawbacks is that a short transaction would have to wait for a long one, which is highly inefficient [1].

Thus database engine has to organize transactions interleaving. Such an issue is solved by the concept of scheduling. The idea is to set up transactions' operations interleaving order in such a way that the impact on data is the same as in the case of sequential executing [4].

1.2.3 Error recovery

Recovering from errors is a crucial aspect of a relational data engine. At first categories of errors have to be specified and explained.
Transaction failure: database engine cannot proceed in executing transaction operations due to a deadlock, an integrity constraint violation, or resource unavailability [6].
System Crash: the hardware failure or the operating system error are encountered [6].
Disk Failure: a physical crash of disc components [6].

When a database encounters a failure it has to perform several actions to restore the last consistent state before a crash. This process consists of taking the last valid data from the disc and applying changes from a log file. That brings a database to contain committed as well as uncommitted transaction modifications. After that actions which were not part of confirmed transactions are reverted which produces the last consistent state before a crash [7].

1.3 Advantages and disadvantages summary

After a long discussion about RDBMS properties, it may become hard to realize if relational data engines have any drawbacks. In this section, all pros and cons are explained and the final concluding opinion is declared.

1.3.1 Advantages

When talking about RDBMS advantages the popularity ranking has to be taken into consideration. In 2021 relational database implementations remain the most attractive way of storing and managing data [8]. Such a popularity trend is not a surprise concerning that middle and big companies still have at least a small amount of projects based on relational databases. For firms, RDBMS continues to be the clearest approach because of its wide expansion and a huge knowledge base in the software engineering community.

Another relational advantage is its ability to cover all use cases of data storage. The concept of tables, transactions, and relations implemented by foreign keys has its disadvantages but can fulfill all business requirements except performance.

One of the most fascinating RDBMS features is concurrency control. Modern informational systems require databases to manage multiple requests at the same time. Such OLTP behavior affects the performance characteristic of relational databases and makes them greatly efficient.

1.3.2 Disadvantage

Relational database implementations have several disadvantages, however, this thesis was inspired by the most critical one. The most crucial drawback relates to performance. Relational databases have implemented many techniques to be efficient enough in many cases, however in 2022 RDBMS still cannot scale horizontally as much as NoSQL instances which makes relational databases less profitable when dealing with huge amounts of data or a high load. The PostgreSQL engine offers some options for being a multi-node system [9], nevertheless, configuring a distributed logic for this RDBMS is much more complicated than in the case of modern non-relational databases. Such a drawback makes scalable NoSQL engines very promising and popular. Some of them are described in the next chapters. The impossibility of relational data engines to scale horizontally may not seems to be a serious issue, nonetheless, in the world of auto-scaling cloud techniques and Big Data RDBMS often becomes a bottleneck when trying to provide users with efficient information systems.

1.3.3 Conclusion

In this section, mostly subjective opinion is provided to prove that relational databases still have their niche and can be used in many cases. At the same time, the main disadvantage of the traditional RDBMS approach is introduced to focus on the extremely intriguing concept of horizontal scalability. This technique will be discussed in more detail in the next sections to demonstrate some strengths of NoSQL data engines.

Chapter 2

NoSQL theory

The main idea of this chapter is to focus on traditional RDBMS dilemmas and manifest the importance of using unusual data-storing patterns. Some theoretical concepts are explained to form a solid understanding of NoSQL's main problems, advantages, and drawbacks. The aim of this chapter is also to make the specific NoSQL models reviewing more clear as a reader would already be aware of non-relational database core concepts.

2.1 Big Data

Before explaining the NoSQL term and related problems the Big Data concept has to be discussed. First of all, it has to be mentioned that there is no academic definition of what Big Data is. However, the most popular opinion is that this term refers to what is called "3 V"–high **V**olume, high **V**elocity and high **V**ariety [10]. Data that fulfills mentioned criteria and requires extraordinary methods for its processing is considered Big Data. There are many other definitions, however, this study focuses on the one depicted above.

"High Volume" term characterize that modern database techniques have to cope with an enormous amount of bytes. Another aspect called "High Velocity" emphasizes that such a colossal information stream is received not once a day but continuously and has to be processed efficiently. The last aspect is "High Variety". This characteristic accentuates the fact that data in the digital world can be presented in different formats (XML, JSON, YAML, text, audio, unstructured data, etc.) thereafter such information streams may require some classification techniques to perform a data-processing in a clever way [11].

2.2 CAP theorem

Non-traditional approaches to managing data are often related to concepts of distributed systems. This term signifies a multi-node approach to store data [12]. Such a concept was not discussed in the previous chapter and may be slightly confusing for engineers having experienced only relational databases. At the time of making this thesis research, the RDBMS approach is considered to be non-scalable in a horizontal way thus the relational idea is not suitable for storing data on multiple machines [13]. Nevertheless, it can be possible that some nameless RDBMS exists but the personal opinion of the author is that traditional ACID relational databases are not successful in horizontal scaling. The reason for that is already discussed transaction features which are very hard to implement in the distributed model. Such a long introduction is extremely crucial because such a non-traditional design of storing data on multiple nodes brings in the idea of the CAP theorem.

An understanding of the CAP theorem is highly necessary when talking about unusual database principles. This concept makes it possible to classify various distributed models. The majority of this term's explanations begin with a similar illustration see Figure 2.1. Therefore the CAP concept depicted in this image has to be explained.

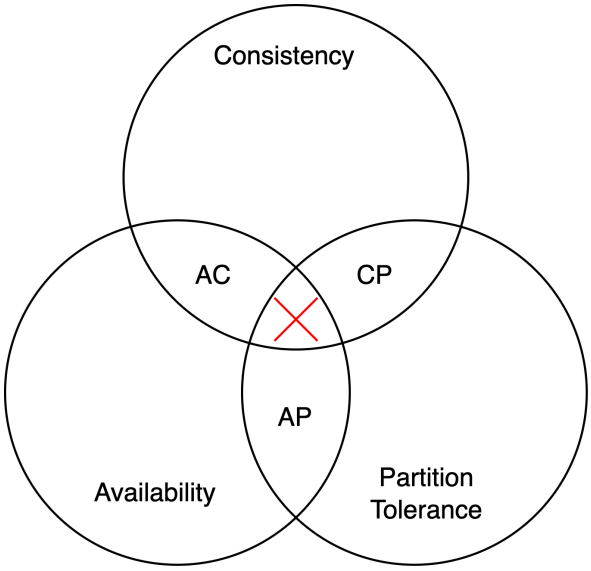

Figure 2.1: CAP theorem

The CAP theorem describes 3 main distributed system characteristics: Consistency, Availability, and Partition Tolerance [14]. These three aspects are explained below which makes an understanding of the CAP theorem idea less complicated.

Consistency term means that each node when performing a READ operation receives the last written data. Therefore after any node is affected by new data, all nodes respond with the updated information [12].

Availability feature describes the idea that any "alive" node of a distributed system often returns a non-error response [12].

Partition Tolerance concept represents the theory that even in case of communication failures between nodes the system continues to operate. This characteristic is highly fascinating when talking about distributed systems because such multi-node applications bring the problem of communication errors between individual computing units thus sets of isolated nodes may appear. After becoming more familiar with 3 CAP aspects the most important CAP idea has to be clarified: a distributed system can only implement 2 CAP features at most [14]. Therefore a multi-node model can only have 1 or 2 CAP characteristics. It would be highly absurdly for a distributed system to have only 1 of the CAP features in case it can have 2. That is why this study does not analyze such approaches with only characteristics but focuses on such models:

- Consistent-Available (CA)
- Consistent-Partition Tolerant (CP)
- Available-Partition Tolerant (AP)

The Consistent-Available model is the most suitable system design for implementing ACID properties. However, in case of communication errors between nodes, a system has to stop accepting requests and declare that the whole system is stopped [14]. Such behavior describes a partition intolerant concept.

Consistent-Partition Tolerant idea declares that a system has to stay consistent and tolerate nodes isolation, but these isolated nodes may become "blocked" and thus not be always **Available** [14]. Such a locking technique makes it possible to ensure consistency even in case of communication failures. Nevertheless, this method is just an example of **Partition Tolerance** strategy for consistency assuring.

Available-Partition Tolerant concept set aside the **Consistency** aspect for the sake of performance. This model is considered to refer to the term BASE [14] which is discussed in later sections. The main idea is not to force every node in a cluster to hold the most actual data. For example, Apache Cassandra distributed database which follows the AP model makes its nodes available at any time and tries to make them as much consistent as possible but without constant guarantees [12].

2.3 BASE

The CAP theorem classification and BASE concept has to be explained to make a reader familiar with fundamental database issues. We have already discussed that relational databases almost always follow ACID principles, thus primarily focusing on consistency [15]. However, as we already mentioned being consistent does not always have the highest priority for a system. Therefore, temporary inconsistency may extremely increase the distributed system performance metrics. Such behavior is described by the BASE concept.

The BASE abbreviation means three multi-node system characteristics:

- Basically Available
- Soft State
- Eventual Consistency

Basically Available characteristic stands for the system feature of being always available for requesting regardless of some nodes' failures or communication errors [16]. **Soft State** aspect declares that a node's state can change for some time. Moreover, changes do not have to be encouraged by users' interactions with a software application. **Eventual Consistency** term highly relates to the previous Soft State concept and claims that after some updates being made to the system, it tries to make every node consistent but this may not be achieved instantly; thus the read request does not have to return the most recent data [16]. The opposite concept is called Immediate Consistency, a system that follows this approach always returns the last written information [17].

2.4 Sharding and replication

Another important concept from the NoSQL world is sharding. The idea of sharding is to divide a data set and spread it evenly across all computing instances [18]. Such an approach makes it possible to store a colossal data block that can not be stored on a single machine. The concept of dividing data also relates to the idea of horizontal scalability.

The sharding concept is highly connected with a replication theory. The main idea consists in making a copy of a shard and to its store on a different node. Such an approach ensures a database's ability to work even after a node failure. In case of a machine loss, another instance could handle requests by using a shard's copy [19]. Even in the case of a shard's replica (copy) absence, a distributed system may proceed to operate for the reason that a node failure would lead only to one shard's loss but not to a whole database crash [19].

Moreover, querying data which are located on several nodes could be more efficient than in the case of a one-machine database. A multi-node system with

shards and replication can parallelize a request; therefore, processing a query may become less time-consuming [18].

2.5 NoSQL term

After becoming more familiar with general distributed systems theory it becomes highly important to introduce the NoSQL term. This concept was used for the first time in 1998 as the name of the relational database which had the significant difference in that it did not support SQL. Then in 2009, a conference took place in San Francisco. Several non-traditional data management systems we demonstrated and the new label for such unusual approaches was declared as NoSQL. This term became worldwide famous, however, has not received any scientific declaration and began to indicate "Not only SQL" or "Non SQL" [20]. As we can see, NoSQL is not a specific technology or a paradigm, it is rather a label for non-relational software products which aim on coping with Big Data. The NoSQL concept does not imply a database to implement a multi-node model, however, this thesis binds these two ideas together because 2 NoSQL data engines discussed below involve such distributed principle.

When explaining the NoSQL term its main distinctions have to be mentioned. The extremely important set of characteristics called ACID has been already discussed. These characteristics are greatly profitable for relational databases which use only 1 node. However, the NoSQL theory focuses on managing data in a distributed way [21]. The greatest advantage of this approach is performance improvement. Bringing more than 1 node into operation makes a system possible to parallel request processing much more than just performing simultaneous actions on one CPU. There are many models in the NoSQL world but it is considered that non-relational databases follow BASE principles instead of ACID thus acquiring availability and partition tolerance benefits [15].

Another NoSQL advantage is the ability to use a "free schema". The classic approach of storing data in relational databases lies in the concept of normalization [22]. This means that data are grouped into tables and relations between records are managed by foreign and primary keys. This approach aims to avoid data redundancy and ensure total consistency [23]. NoSQL models focus on maintaining data in many ways. One of the most common patterns is storing records as documents which are in practice collections of JSONs or XMLs. In such models data becomes more intuitive and easy to scale, however, a redundancy characteristic appears. This is not the only non-relational model, other denormalization patterns are described in later chapters but the main idea is that NoSQL techniques do not follow normalization rules but try to make data schemes flexible thus not forcing them to define strict tables and relations between them.

2.6 NoSQL databases division

A reader is considered to be already a little bit familiar with the NoSQL context. Given the above the NoSQL models' division idea can be introduced. The idea to divide non-relational approaches is described in the *Learning Neo4j* book [24]. The author borrowed this view from Martin Fowler's text *NOSQL Distilled*. The division idea consists in separating non-traditional technologies into two groups.

The first one is a collection of so-called aggregate stores. These are NoSQL databases that follow the principle of storing data and related information in the same place. This category contains document-oriented, key-value, and wide-column databases. Such databases follow the pattern of storing a document/row and "linked" information together. Therefore such engines eliminate the need of connecting data through foreign keys as we know it from relational databases [24]. Databases from this category follow the doctrine of saving data in the format in which they are later required instead of following the RDBMS concept of dividing data into entities that could be then linked in several possible ways. An example of such behavior is illustrated in the next chapter where a document containing a person's information also collects a man's relatives in the same document 3.1.1.

The second paradigm focuses on graph-oriented databases. The author of the *Learning Neo4j* book Rik Van Bruggen claims that the graph technique of storing data is an improvement of the method invented by RDBMS creators. Enhancements of the graph model lie in upgrading the way data are stored and improving the querying approach [24].

Chapter 3

Introduction to NoSQL databases models

This chapter aims to make a reader more familiar with various NoSQL models. The reason is that choosing the right database always implies choosing a model which will fulfill all desired requirements, thus, an engineer, first of all, has to consider which data management model he is going to work with before choosing a specific database. There is no deep dive into the exact implementation, this chapter does not cover specific product comparison in detail either. However, 4 NoSQL models are introduced, and 2 of them are explained in detail in later parts of the thesis.

3.1 Document-oriented approach

When talking about developers and database specialists with RDBMS experience only it is not a trivial task to apply a document engine model to a software product. The reason for that is that operating with document databases requires a specific paradigm understanding. Thus, some theoretical material is provided at the beginning and then specific engine instances are discussed to make a specific document-oriented product easier to choose.

3.1.1 General theory

The first NoSQL idea to be explained is the Document-oriented model. This concept of storing information is considered to be one of the most popular [25]. The main idea of such an approach is to perceive a data unit as a JSON or an XML record which are called “documents”. However, it has to be mentioned that the document-oriented approach does not suppose a database to implement BASE or ACID principles [26].

Another peculiarity of the document model is that a data schema can be implemented in a flexible way [27]. Thus, for example, JSON documents would not have mandatory fields. If the is a requirement for a document to contain an unknown field, such "schema-violating" document can be naturally inserted into a database with a new field, and previously stored documents would simply not have a new property. Such behavior may seem too obvious for newbies in a database world but in the case of RDBMS adding a new field to a table cause all rows to become affected with such a property initializing it with a NULL value. Thereby the data unit pattern of any table is extremely strict ensuring that all rows have the same properties which is not the case in the document model.

The self-description of data units is the next feature of document-oriented databases. Such a model leads to the advantage that a schema does not need to be explicitly defined but can be obtained from JSONs or XMLs. Considering that information is denormalized and is not distributed by tables it is more intuitive to work with because any document contains all data related to it.

The example of a single document is depicted on Listing 3.1. It is noticeable that the "relatives" attribute does not act as a foreign key or any type of reference but contains all necessary information directly in its value. Such a technique of storing data in one place refers to the denormalization term which is explained above but demonstrated here in a simple example. The absence of any type of relations makes it possible to implement the horizontal scaling procedure by distributing independent documents on individual nodes.

Taking into consideration the advantages and an attraction of the document-oriented model it has to be mentioned that the business case of storing JSONs is not something unusual. Furthermore, it is not a reason to reformat the whole application initially based on a relational database. The solution could be to use PostgreSQL. This data engine has a significant advantage in that its JSON datatype is extremely useful in storing this kind of data [28].

3.1.2 Document-oriented database products

As already mentioned it is incorrect to declare any specific feature of document-oriented data engines regarding ACID principles, BASE paradigm, CAP classification, etc. The reason for that is that the concept of storing JSONs/XMLs does not imply any definite characteristic but is considered to be just an abstract database model. The evaluation of the most popular document data engines has to be clarified for the sake of understanding the benefits of the Elasticsearch database which are discussed in the next chapter.

The first distributed document database to introduce is **MongoDB**. This data engine is considered to be the most popular document-oriented software product for storing data [29]. MongoDB uses concepts of sharding and replication to provide a better performance, possibilities to recover from errors, and achieve storing a large amount of data [30]. Another feature that arouses

```
{
    "_id": 5,
    "name": "John",
    "surname": "Doe",
    "gender": "male",
    "relatives": [
        {
            "name": "Richard",
            "surname": "Doe",
            "gender": "male"
        },
        {
            "name": "Jane",
            "surname": "Doe",
            "gender": "female"
        },
        {
            "name": "Michael",
            "surname": "Doe",
            "gender": "male"
        }
    ]
}
```

Listing 3.1: JSON document example

from the distributed characteristic is High Availability. It is achieved by storing data redundantly on several nodes; thus, in case one node is overloaded another is able to handle a request [31]. MongoDB also supports transaction techniques over multiple documents. All these features make MongoDB a multipurpose database that may be used as an unmistakable choice for integrating a NoSQL technology.

Another document approach implementation is the **DynamoDB** database. This data engine is not limited only to document paradigm but can also operate with the key-value data format. DynamoDB applies concepts of sharding and replication as well as previously explained database [32]. Compared to MongoDB which can be used either in an on-premise way or as a service, DynamoDB can be used only through the AWS platform. Its data types are more limited compared to MongoDB as well as document size limits. While MongoDB can ensure some schema constraints, DynamoDB does not have any such ability [33]. DynamoDB is a perfect choice for cases when a NoSQL database needs to be leveraged fast and a company does not have an already set up infrastructure. It also uses a distributed model to cope with huge data

sets and to be highly efficient. However, the subjective opinion is that MongoDB may be a better choice because of its benefits range and its ability to deploy it in any environment.

The third document database to introduce is **Couchbase Server**. This database uses replication and sharding ideas, as well as data engines, discussed above [34]. Couchbase has several advantages as the extremely efficient cache protocol or the integrated web interface for performing maintenance tasks [35]. It is considered an easily scalable database that is able to automatically rearrange a cluster's data after adding a node [36]. Compared to MongoDB Couchbase Server offers some unique features, however, it is not obvious which one is better. Probably deeper dive into data engine architectures could make the comparison clearer but such research does not belong to the thesis scope.

The last NoSQL document storing engine to introduce in this section is Elasticsearch (ELS). It is not considered as a document database but as a search engine [37]. However, its main data units are JSON documents which inspire to claim that Elasticsearch is a NoSQL distributed document-oriented database with an additional range of capabilities. The reason why ELS is often related to a search engine term is that this database offers a huge set of analysis functionalities that makes a process of text search highly efficient. ELS is built on top of the Apache Lucene search engine, Elasticsearch also uses sharding and replication techniques to ensure efficient scalability, availability, and reliability [38]. It is quite hard to compare ELS with databases introduced before, because of its enormous search competence which makes it more than just a NoSQL database. Its related products and their functionalities are described in detail in the next chapter.

3.2 Graph approach

The previous section aimed at the document-oriented theory which is quite common in the NoSQL world. Nevertheless, this part of the thesis explains a quite exotic concept even in the non-relational context. The section's idea is to introduce a graphic data management theory.

3.2.1 Introduction

A reader may consider the beginning of the thesis quite verbose. Nevertheless, the advantage of the graph database model is not that obvious without comparing it with the traditional relational approach. RDBMS has been with us for decades and has evolved that much to be capable to handle a tremendous set of software development requirements. Relational databases deal mainly with data in the form of tables and rows, the foreign keys concepts enable to implement of some sort of relations between entities [39]. We have described many aspects which make relational data engines incredibly useful and efficient, however, connections between rows are not the case. The study

written by Emil Eifrem(Neo4j co-founder and CEO) and co-authors dedicates a section in which they compare an example social network RDBMS of 1.000.000 persons and then compare it with the same network implemented in Neo4j. The comparison is focused on retrieving friends-of-friends in particular depth and analyzing the results. The contrast between a relational and graph approach is colossal, the test outcome demonstrated the fact that retrieving friends-of-friends on the depth 5 is was not even completed by the RDBMS, at the same time Neo4j required only 2 seconds to do the operation [40]. Taking into consideration such comparison we can affirm that a relational approach is not suitable for coping with data that are modeled in a way of a dense graph.

When discussing a document-oriented paradigm the denormalization concept was introduced. This idea influenced the way in which data are stored. The processed and retrieving idea is that entities and their related information are stored together to avoid joins as we know them from RDBMS. That is the most unsophisticated approach to managing links in a document database.

Non-graph databases have a possibility to connect some rows to another through some sort of an additional field that acts as a foreign key [41]. Thus, it is not correct to declare that only graph data engines are capable of managing relations. However, retrieving foreign keys and then fetching corresponding data by identifiers is extremely inefficient. This method also forces to update all documents relation fields in case a document is deleted [41].

3.2.2 General theory

The main point of graph data engines is to provide an interface to storage that would enable users to manipulate data as it would be a graph. However, such databases do not have to guarantee that they use efficient traversing techniques, their underlying data-management methods could be simply object-oriented, relational, or other inefficient in the context of big data techniques [42].

Graph databases use the model of nodes and edges which connect them [43]. This thesis is not going to dive into the theory of graph algorithms, it is enough to understand what a node is, and that they are connected by edges that are able to be directed. This model introduced is nothing but an abstract concept that is actually divided into two specific design patterns. These specifications are **Property graphs** and **RDF graphs** [44]. RDF (Resource Description Framework) is a way to represent a graph. The main idea behind RDF is to represent data as a set of triples: *subject - predicate - object* [45]. Subjects and predicates are mainly represented using the IRI pattern, objects can be IRIs or literals (strings, integers, etc.) Subjects and objects which follow the IRI scheme are considered nodes in a graph and a predicate as a directed edge. RDF brings in the concept of blank nodes, these are nodes (subjects, objects) that have unique identifiers inside a graph but do not follow the IRI pat-

tern and are usually represented in such way: *_:1234*. RDF, however, is not a specific data management system, it is rather an approach to describing and exchanging graph data.

The second model is the Labeled Property graph which is interchangeably used with the Property graph term in the scope of this thesis. The Labeled Property approach focuses on nodes and the relationships between them. Moreover, this model brings in the concept of properties and labels [46]. The labeling technique assigns a value to node/link which makes it possible to group them in some way accordingly to the roles they play in the data model. For example, a node in a social network can have a Person, Group, or Post label which specifies its function in a database. Nodes and relations are also able to store properties [47] which enables them to act as key-value holders. The Neo4j graph database uses the Labeled Property model. The rich analysis of the mentioned data engine is presented later and makes the understanding of the Labeled Property approach more distinct.

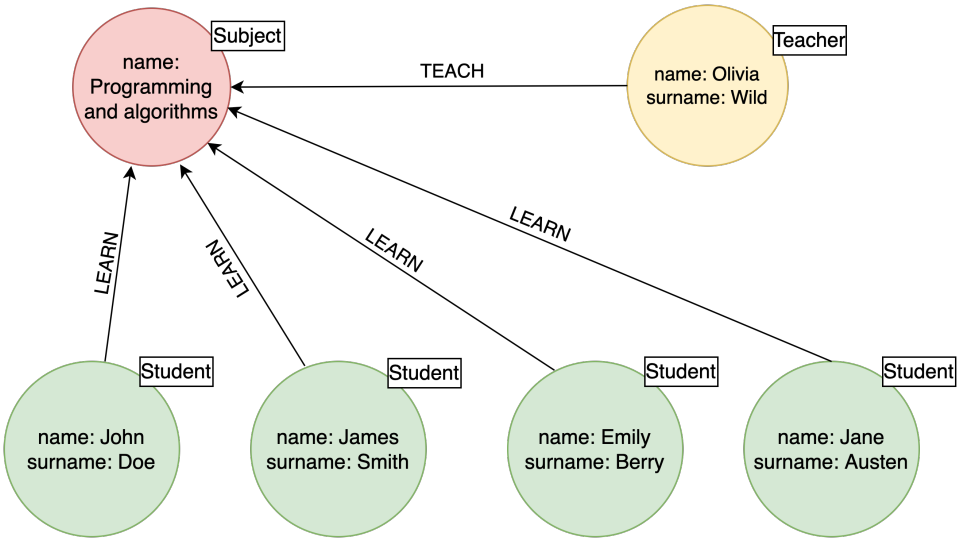

Figure 3.1: Labeled Property graph

The theoretical style of describing the graph-oriented approach in databases implementation could be fairly confusing. However, the main idea which has to be realized is what a Labeled property graph looks like. Therefore, the example of applying such a model is demonstrated. Let us imagine an application that has to store information about classes in the university. The visualization of the Labeled Property graph would be similar to such illustration 3.1. On the image, three labels are introduced: Subject, Teacher, and Student. Thus, all nodes are divided into these 3 categories which group nodes

based on their roles in a database. Properties feature is also demonstrated on the visualization, each student has a *name* and a *surname* fields, a teacher also has such characteristics, the subject node contains only *name* property. Relation links bring in information about node relationships. Each student has a link with a *LEARN* label directed to a subject node. The teacher connects to the subject node with a *TEACH* link. The introduced example does not cover all gains of a graph model design, for instance, relations properties were not explained in a detail as well as the importance of links direction, the reason of that is a desire to keep the example graph as simple as possible. The directed relations were demonstrated in the example to illustrate that the Labeled Property model support this instrument of connecting nodes, however, undirected links are not considered a forbidden technique; thus, they can be also used in the example.

3.2.3 Graph model disadvantages

Some graph engine advantages are mentioned in the introduction section 3.2.1, however, to create a more comprehensive opinion of the paradigm its disadvantages have to be mentioned as well. It has been already mentioned that graph databases are extremely efficient in querying a dense graph, however, it can not be declared that graph engines are more performant than others regardless of context. Thereby, this model's disadvantages have to be introduced.

3.2.3.1 Horizontal scalability

First of all, the graph paradigm brings in scalability issues. Thus, the case when a graph database becomes terribly large brings in non-trivial questions regarding horizontal scalability [48]. Each graph data platform implements scalability solutions on its own; nevertheless, a basic idea has to be demonstrated.

We have already introduced the idea of sharding. Graph databases also face the problem of the necessity to store a dataset that needs more memory than a machine provides. Concepts and an illustration explained in this section are inspired by a DZone article [49]. The main advantage of the graph model is a fast traversal process. After executing a query an engine is passing through matching nodes and acquires necessary data. Graph databases are considered to be extremely efficient in traversing from a node to its connected neighbors. However, the consequences of dividing a connected graph into several machines do not always have only higher availability or performance benefits. The data distribution method may also lead to unpredictable execution times. This embarrassment is caused by the fact that traversing through an edge between nodes located on different servers requires developers to take into consideration network latency obstacles.

Let us describe the scalability theory with an illustration. Imagine we want to scale a graph storage platform horizontally, for that reason the graph data are settled on two servers instead of one 3.2. Let us assume that queries are going to be concentrated on disjunctive data, that is to say, queries would not use relations that would lead to network communication between servers i.e. there would be no traversing using the V3 -> V4 edge. In such a scenario, the data distribution is going to make sense and is going to have performance and availability benefits. Nevertheless, this exact style of querying is hard to predict. The illustration 3.2 is trying to accentuate that a process of retrieving nodes' neighbors is 5.000 times slower in case nodes are located on separate servers than if they are loaded in the joint machine memory. Taking this into consideration it can be affirmed that a naive distribution of data onto several computation units may lead to the network hop problem and have high-performance losses.

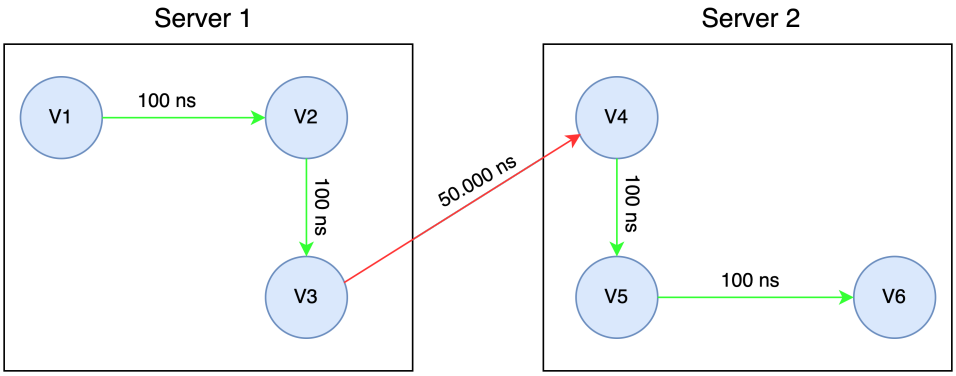

Figure 3.2: Scalability issue

3.2.4 Query languages

Another highly important graph engine weakness is the absence of a query language which would be common for all databases of a graph type [48]; thus, switching a specific graph database requires rewriting all queries used in a project. In contrast, relation databases offer the SQL language which can be easily adapted to any specific RDBMS.

The set of dissimilar query languages used to retrieve data from graph stores is fairly rich. An adept of graph platforms could possibly contradict this statement. The fact that almost every relational database has its own "SQL dialect" can possibly be an argument; however, the subjective opinion of the author is that the difference between query languages of different relational platforms is not that considerable.

3.2.5 Graph-oriented database products

After becoming more familiar with the concept of the graphs-oriented model, specific databases have to be illustrated and evaluated. Two graph sub-models are introduced in the previous section which is RDF and Labeled Property graph approaches. These models, however, do not specify the design of data storing and processing; thus databases that handle, for example, an RDF engine does not imply a database to work in a specific way, but only the ability to store RDF data.

The most popular database which is described in detail in the following chapters is **Neo4j** [50]. It is a highly scalable data platform with Cypher query language support. Neo4j also ensures ACID and transactional operations. Neo4j also supports a "schema free" principle as well as schema constraints [51]. Thus, a developer can leave a data model relaxed to enable a schema to evolve more effortlessly. On the other side, Neo4j makes possible schema restrictions to make data more standardized [51].

The second most popular database due to the *DB-ENGINES* rating is **Cosmos DB** [50]. This platform's advantage is the possibility to store data using all 4 core NoSQL models: document, graph, key-value and wide column. Cosmos DB implements the SQL-like query language, limited ACID transactions, and a sharding technique [52]. It is available only through the Azure platform compared to Neo4j which offers an open-source Community version.

ArangoDB is the third most famous graph database according to *DB-ENGINES* rating [50]. This is also a multi-model database being able to handle data using all 4 NoSQL core approaches. ArangoDB supports ACID properties, sharding, and free-schema principles [53]. ArangoDB also works with data using its own query language called *ArangoDB Query Language* (AQL) [54].

The idea of this section is not to compare mentioned databases nor to describe all their specifics in detail. The purpose is to introduce the fact that there is not only Neo4j on the market but a lot of other worthful databases.

3.3 Wide-column approach

This part of the thesis is going to clarify the wide-column (also known as column family) NoSQL, model. It has to be mentioned that this approach as well as the next one (key-value) is not going to be discussed as carefully as already explained ones. The majority of ideas explained in this section highly relate to the most popular wide-column database called Apache Cassandra.

3.3.1 General theory

To make an understanding of the wide-column databases more effortless the interconnection and comparison with the relational approach are going to be

used. First of all, let us introduce the term *Keyspace* which refers to what we know as a *Database* i.e. a collection of tables in the RDBMS context. The *Column Family* concept can be comprehended as something similar to a table in a relational database. Another technique used by wide-column databases is the *Row Key*. That can be also compared with the traditional relational *Primary Key* idea [55]. After becoming more familiar with the terminology it becomes possible to dive into column family stores basis.

The main idea behind the wide-column model can be described in several possible ways. The book *NoSQL Distilled* emphasizes the dissimilarity of ideas of storing data using column family and relational approaches [56]. Authors Martin Fowler and Pramod J. Sadalage claim that in contrast with the RDBMS paradigm which stimulates a developer to perceive data in the form of tables, the wide-column model enforces a programmer to think in the way of two-dimensional maps.

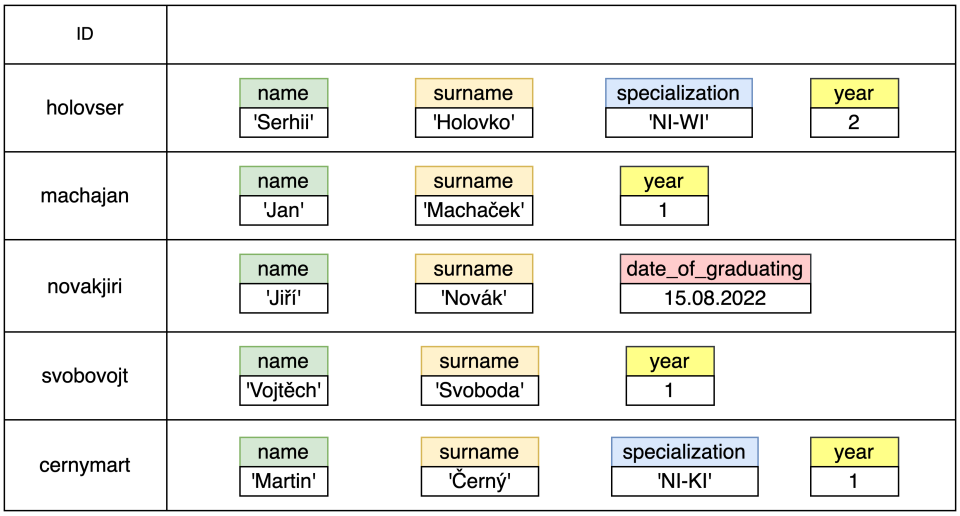

Figure 3.3: Students' Keyspace

Let us imagine a use case of managing a Keyspace (table) of students see Figure 3.3. The first dimension of the data retrieval is a map where keys are IDs (*holovser*, *machajan*, *novakjiri*, etc.) and rows act as values. The obtained row can be also interpreted as a map. The secondary structure (row) can be considered as a map with columns as keys (*name*, *surname*, *specialization*, etc.)

It is possible to maintain many types of values inside a row. There can be primitives, tuples, collections, or values of a custom type [57]. Cassandra, also, provides a possibility to manage the metadata of each column value. These additional data can be **Timestamp** and **Time-to-live** information [57]. The

first parameter specifies the time of the last modification. The second one defines a moment when a column value has to be eliminated.

Another fascinating concept used by the Cassandra project to ensure some sort of consistency tuning is the **Consistency Level** and the **Replication Factor**. The problem of maintaining data on several nodes is not that trivial. Cassandra brings in the technique of storing the same data on different machines (replication). Thus, the retrieval process from a distributed system can become more efficient. The Replication Factor (RF) is the number of nodes on which the "write data" has to be duplicated. The Consistency Level (CL) in the context of the write/update process is the number of nodes which has to confirm that data were written to the node. Therefore, if the write is successful it is possible to declare that at least CL nodes manage the data. Nevertheless, the system would continuously try to achieve the state when RF nodes store the written information [17]. The Consistency Level in relation to READ operations refers to the number of instances that have to confirm that a process was completed. It is also important to emphasize that a higher read/write Consistency Level affects the efficiency of the read/write operations.

Immediate Consistency may not be a requirement when dealing with a NoSQL database, however, the "consistency tuning" theory claims that even a distributed system is able to implement it. The dilemma is which aspect of the application is more important for an application: write or read. The distributed architecture design has to consider a formula that ensures Immediate Consistency. This rule describes the concept of tuning the Read and Write Consistency Levels is such way: $ReadCL + WriteCL > RF$ [17]. This formula emphasizes the correlation between the time cost of reading and writing actions. Thus, it becomes more clear how to configure the read and write efficiency concerning application requirements.

3.3.2 Wide-column database products

In this section the thesis also refers to the *DB-ENGINES* website which claims that the most popular databases in the context of storing wide-column data are **Cassandra**, **HBase** and **Microsoft Azure Cosmos DB** [58]. However, the first one leads significantly. That is the reason why main column family features were discussed based on the Cassandra architecture.

A highly interesting difference between all these 3 databases has to be mentioned. The ACID theory was already discussed, therefore it is fairly fascinating if wide-column projects are able to implement these transaction features. Cassandra in contrast with another two databases does not implement ACID characteristics. HBase is able to perform ACID operations on the level of an individual row. Cosmos DB has the ability to execute multi-item transactions in case items are located on the same node [59]. All these databases

use sharding and technique. Cassandra and Cosmos DB use SQL-like query languages, while HBase does not support such kinds of mechanisms.

3.4 Key-value approach

The last NoSQL theory to introduce in this chapter is the key-value approach. After becoming familiar with 4 core non-traditional concepts it may be fairly complicated to find out the difference between the previously introduced and the key-value one. This model is not explained such attentively as document-oriented or graph databases. However, the main idea and its characteristics and specific software solutions are introduced.

3.4.1 General theory

The key-value model is treated as the most understandable approach of all 4 NoSQL concepts explained in this thesis [60]. The previous section introduced the idea of managing data in a 2 dimensional way. This approach, on the other hand, can be illustrated as a traditional 1D key-value structure or a hash map. The keys of such databases could be some sort of entity identifiers: generated IDs, emails, etc. The value part is not analyzed as in the case of wide-column databases; thus its individual fields are not able to be queried. The key-value model requests are only focusing on retrieving the value unit with the key knowledge.

The main benefit of such a simple model is scalability and easiness. Each key-value pair is treated as a single independent unit. Therefore, these records can be easily distributed over different shards which are usually located on separate machines [61]. There is also an opportunity to manage links between values which makes it possible to slightly improve a data model to be more powerful than pure key-value storage.

Key-value databases are often designed to implement BASE principles and not to be 100% consistent. However, as already discussed, the complete consistency can be achieved by assigning specific values to write and read consistency level parameters [60]. There are many other distributed system concepts that key-value data stores use but many of them are extremely similar or even the same as already explained ones.

3.4.2 Key-value database products

Referring again to the *DB-ENGINES* ranking it can be claimed that a total leader in the universe of key-value databases is **Redis** [62]. Another 2 in the TOP-3 range are **Amazon DynamoDB** and **Microsoft Azure Cosmos DB**, but only the first one is a purely key-value database. The other 2 are multi-model products, for that reason, they are also mentioned in previous sections. The scope of this thesis does not cover the comparison of popular

key-value databases, due to the necessity to dive very deeply into specific products' internal details.

Chapter 4

Elasticsearch and Neo4j

After becoming familiar with such concepts as a CAP theorem, BASE, sharding, and replication it has the sense to demonstrate how all these techniques are implemented by ELS and Neo4j. This chapter is not focusing on technical details in the context of setting up a database instance, configuring proper VM memory amount, or other DevOps topics. The aim of this part of the thesis is to familiarize a reader with the ELS/Neo4j architecture and the main ideas that stand behind its efficiency. Such knowledge about these 2 modern databases is crucial because of the fact that the database platform which is discussed in the practical part is built on top of Neo4j and ELS. Thus to understand the project's benefits it is necessary to get some knowledge about several core principles.

4.1 Elasticsearch

Previous chapters can be considered as a fairly long preface to the exceptionally amusing technology called Elasticsearch (ELS). In this section, a reader is going to dive deeply into some internal details which can explain the reason for the hype around the Elasticsearch technology stack. Some information used in this section is taken from a paid course on the Udemy platform called *Complete Guide to Elasticsearch*.

4.1.1 Architecture

ELS is a distributed NoSQL engine, which is designed to be greatly efficient in performing operations on many virtual or physical machines. Each of these nodes is considered as a separate ELS instance that in cooperation with other machines forms an Elasticsearch cluster see Figure 4.1.

Elasticsearch does not support transactional techniques. *Kartik Gautam*– the author of the "Elasticsearch- Introduction,Basic Concepts,Features & implementation" article, in his work he emphasizes the necessity to store data

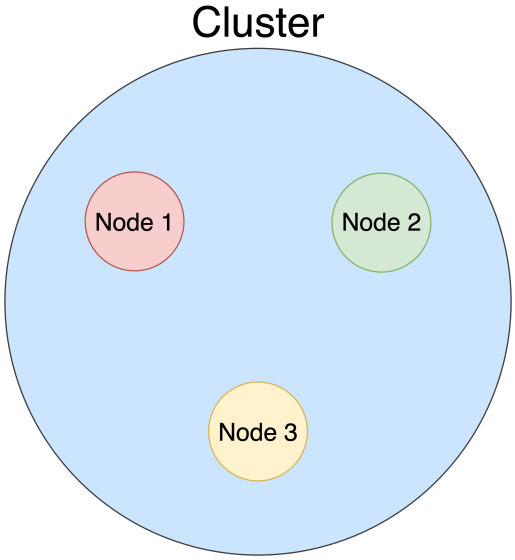

Figure 4.1: Cluster

in an ACID database before indexing it in ELS to ensure that data could be recovered or and a source of consistency would be ensured [63].

However, ELS supports a versioning technique that makes it possible to retrieve a document (for instance in a backend service), then perform some business logic, and send an update request with a version parameters. When ELS receives such a query it compares the version of the document stored in the system and versioning parameters sent in the request, and in case versions are equal it admits the update. This technique makes it possible to guarantee at least a highly limited transactional behavior.

Therefore, the distributed architecture of Elasticsearch is either a benefit or a drawback. Compared to MongoDB, ELS is not able to support ACID principles; thus, it may not be the best choice when dealing with an application that has to ensure a transactional behavior or a strong consistency.

4.1.2 Indexes

It is already mentioned that ELS is a search engine and not a database. However, the difference between those 2 concepts is very unnoticeable. Therefore, in this thesis, Elasticsearch is considered both a search engine and a document-oriented database.

Data units that ELS takes care of are JSON documents. Such records are organized in a structure called an index. An index is an entity similar to a database in the RDBMS world [64]. Nevertheless, it is not a common practice to store documents of different nature in the same index. The subjective

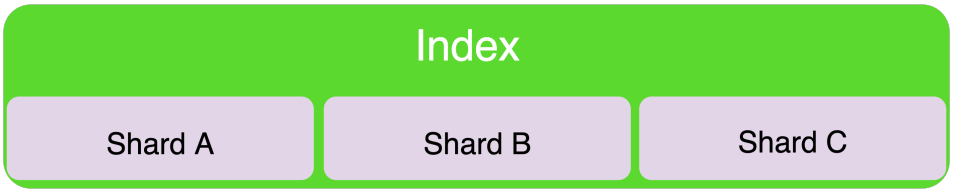

Figure 4.2: Sharding

opinion of the author is that indexes in practice are treated more like relational tables.

Before explaining index-level configuration parameters it is necessary to understand and demonstrate how ELS implements the concept of sharding and replication. An index is a set of document, however, an index is not the aspect that an ELS node manage. The data set that Elasticsearch focuses on is shard. A shard in a nutshell is just a part of an index [65], consequently, an index can be considered as a set of shards see Figure 4.2. An index formed by 3 shards is depicted in the illustration.

The idea of splitting an index into shards relates to the concept of managing data on separate machines 4.1.1. An individual node may not be always able to handle the whole index which can be thousands of GBs huge. Thereby, an index is usually divided into many parts (shards) which are distributed and replicated based on the cluster's capabilities.

Elasticsearch designed a cluster using the primary and replica shards technique. The primary shard is the main shard which is usually duplicated by creating a copy called a replica shard [66]. ELS usually separates primary and replica shards on individual nodes to ensure that a machine collapse would not lead to a data loss and also to improve the efficiency characteristic [66].

Such distributed design with shards replication is demonstrated in Figure 4.3. In this image, a cluster of 3 nodes (ELS instances) is demonstrated. Each node manages 1 primary shard, for instance, Node 1 is responsible for primary shard A; thus, all replica shards are located on other nodes due to already discussed reasons. The logic of locating shards on a node depends on a cluster's capacity, considering the same illustration 4.3, in case of the nodes' impossibility to manage all three shards a machine could handle also 1 primary and 1 replica shard or even only the primary one.

Each Elasticsearch index has several configuration parameters. Most fascinating ones are *number of shards* and *number of replicas* [67]. These parameters obviously specify shards and replicas amount. A fairly interesting aspect of index settings is the fact that these parameters are divided into 2 categories. The first group specifications can be only set at the index creation moment, the second category systematizes parameters that are able to be re-

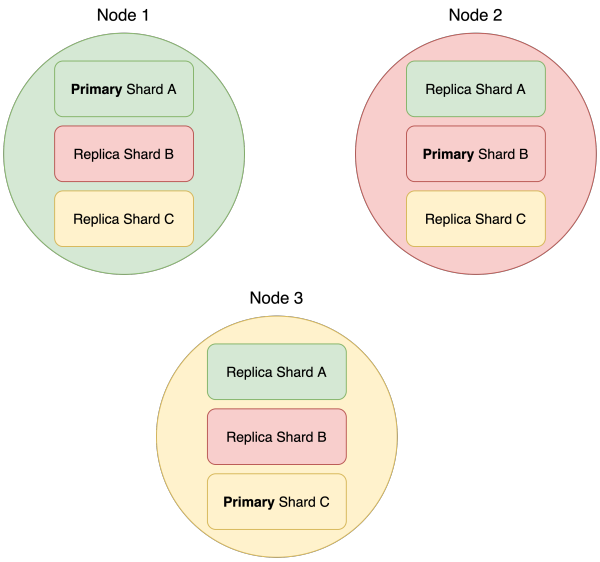

Figure 4.3: Replication

defined after an index was established [68]. For instance number of shards is a static setting and can be established at the moment of creating an index, while the number of replicas can be reconfigured at any time.

4.1.3 Analysis

Let us imagine a requirement of storing product documents in ELS, each product would only have a name, a description, and an interest level. A software developer would perceive an Elasticsearch storage as an engine storing documents in the same way he uses them in an application see Listing 4.1.

However, when ELS receives a request to store a document it performs an operation called *analysis*, as a result, the document's text field transforms into the array of keywords that are detected in the string. A developer or a database engineer is responsible to specify a value type or an analyzer and based on this specification Elasticsearch determines if a field would be passed through the analysis process or not. ELS enables a user to create a custom analyzer or configure existing ones. Nevertheless, any custom or a built-in analyzer consists of 3 parts: character filters, tokenizer, and token filters [69]. A text value that has to be processed goes through a character filter where all irrelevant symbols are removed or changed. After that, an "filtered" text goes to the tokenizer section in which a string is split into substrings. Finally, a generated array of tokens is processed by token filters which remove/transform generated in the previous section elements.

The discussed above technique is not a trivial one, thus, an illustration of

```
[
{
    "name": "computer",
    "description": "A modern PC with Intel Core i7 chip.",
    "interest": "high"
},
{
    "name": "Call of Duty",
    "description": "A PC video game made by Activision.",
    "interest": "medium"
},
{
    "name": "laptop",
    "description": "An Old HP laptop.",
    "interest": "low"
}
]
```

Listing 4.1: Product JSON

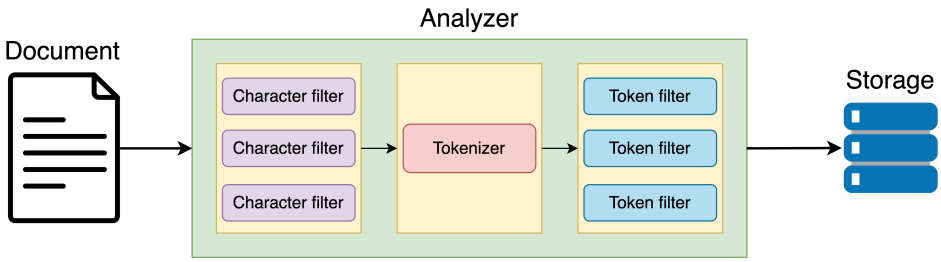

Figure 4.4: Analysis

the whole process is provided 4.4. A document on the image gradually goes through different analyzer's components and then is inserted into the storage. An analyzer demonstrated on the image is a depiction of analysis architecture, the specific analyzer may have another number of character and token filters.

The analysis technique is used when a text field has a corresponding type or an analyzer specification. For instance, a parameter *interest* see Figure 4.1 is not a value that has to be processed for later text search, because values stored under the *interest* key would be queried in the same way they are depicted on the products' JSON array.

Let us dive a little bit deeper into how the build-in analyzer called *Standard* works. This analyzer does not have any character filer, its tokenizer is called

Standard Tokenizer. The default token filter is the *Lower Case Token Filter* which lowercase all tokens. The standard analyzer can be also configured by enabling the *Stop Token Filter* [70].

The example message "A modern PC with Intel Core i7 chip." is a good illustration of a text which would be usually analyzed to perform an efficient searching afterwards. The standard analyzer would transform the string into this array of substrings:

```
["a", "modern", "pc", "with", "intel", "core", "i7", "chip"]
```

An analysis technique is an extremely powerful method. For instance, a token filter could be configured to transform different synonyms of the same word into an equal token. For example, words *deeply*, *extremely*, *highly* and *greatly* could be transformed to the term *very*. The connection between an analysis process and how this can help with searching may not be obvious. An investigation of the retrieving procedure is presented in the next section where all previously discussed concepts are coupled together and form a complete view of the Elasticsearch efficiency.

4.1.4 Searching

The analysis theory introduces the concept of diving a text field into many substrings, but that is not clear how such segmentation helps in improving search efficiency. In this section ideas for analyzing a query and inverted indexes are discussed which makes it possible to understand why ELS is the best choice when dealing with a text search.

At this point, a reader should be familiar with the fact that ELS does not store values in the way presented in the example 4.1. It actually uses some sort of trivial structure to return documents to the client but this is the secondary method of managing data. An extremely fascinating technique used by ELS to store tokens retrieved after the analysis phase is inverted indexes.

The inverted index is a data structure that works as a map where values are represented as a set of entities. In the context of ELS keys of such structure are terms/tokens and values are documents that contain given terms see Figure 4.5. This data structure makes it possible to answer a question "Which documents contain the term?" remarkably fast.

But how does an inverted index relates to the analysis? After ELS performs a text processing it stores tokens in the inverted index [71]. Thereby, it becomes possible to search for documents not by the whole phrase but only with the knowledge of some part of the string. Imagine a case of an online bookstore, clients of such site/application would probably like to look for a book they have already read for instance by main characters' names they remember, or by specific words used in this book. Thus, it would be unwise to manage books' content as a huge string. Nevertheless, the idea of creating an inverted index of a text would enable to efficiently search inside it.

Term	Document 1	Document 2	Document 3
a	✓	✓	
modern	✓		
game		✓	
activision		✓	
hp			✓
core	✓		
laptop			✓
chip	✓		
pc	✓	✓	

Figure 4.5: Inverted index

The last theory which has to make a reader assured about the basic idea of how Elasticsearch performs searching is the concept of analyzing a query. There are different querying methods ELS provides. The most suitable technique for text searching and also the most interesting one is the full-text querying which refers to the option called **match** query. The main idea is that a text in the query is analyzed by the exactly same analyzer as the text in the field we are querying. On the other hand, ELS also supports non-analyzed requests also known as **term** queries [72]. Text values passed in these requests are not analyzed and are compared to values stored in the inverted index.

Thereby it is highly important for a developer to realize if the documents field he is querying has been analyzed. Based on this information and an application use case the engineer would be able to construct a correct query. It may be a valid desire–trying to match a non-analyzed request with an analyzed field or vice versa, however, it has to be mentioned again that a developer must clearly understand how were documents he is working with are processed and how is his query handled by ELS. The illustration of how match and term concepts differ is demonstrated in Figure 4.6.

4.1.5 Kibana

Kibana is visualization software that could be only connected with an Elasticsearch cluster. Working with ELS storage could be done through a command line and the cURL software. Nevertheless, constructing complex queries without typos check and code completion may be extremely hard while working on any bigger project. Thus, it becomes necessary for any developer working with ELS stack to be familiar with the Kibana project.

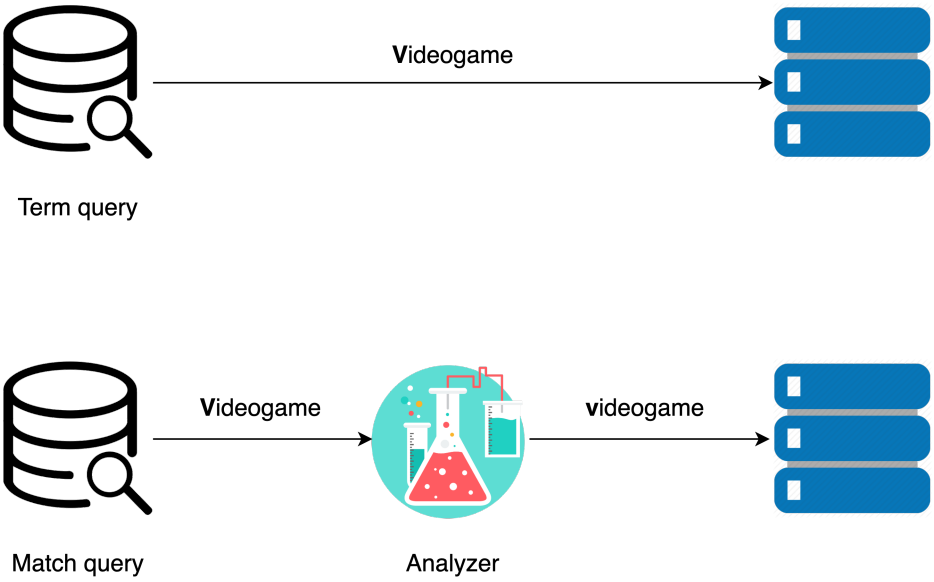

Figure 4.6: ELS Querying

First of all, Kibana provides a console window from which all kinds of operations can be executed. However, all activities done in the Kibana console are still very manual, it is considered to be a workspace for developers where they can debug, or retrieve data from their Elasticsearch environments.

Kibana is also capable of creating graph visualizations, heat maps, pie charts, and others. These data depiction methods can be organized in dashboards where already configured visualizations could be accessed at any point in time. Dashboards are often used not only by developers but also by analysts, project managers, and other code-skeptic people. To make a reader aware of what a Kibana dashboard could look like and how many various visualization options it has, the illustration from the official website is provided see Figure 4.7.

4.1.6 Summary

Elasticsearch is a prominent software. All its functionalities, details, and related products are a topic for an independent dissertation. However, its main features and some internal details were provided and explained.

ELS is designed to be highly scalable but may is not the best choice when consistency is a requirement. Elasticsearch is also not capable of making multi-document transactions. The subjective opinion is that ELS drawbacks are very hard to find. Compared to RDBMS Elasticsearch is able but not

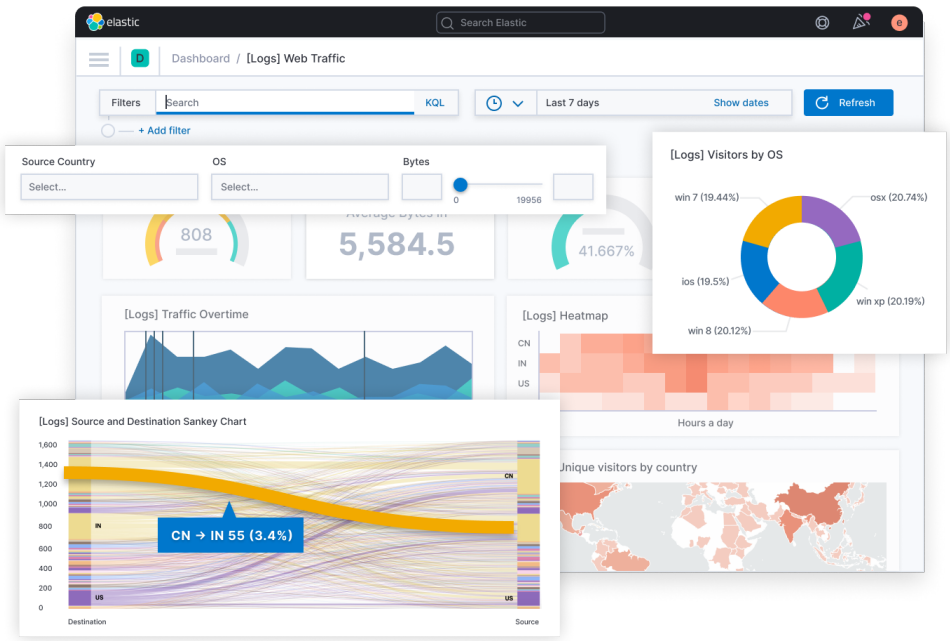

Figure 4.7: Kibana dashboard, from [73]

highly recommended for managing relations. It may not cover all business requirements, which traditional databases can do. However, it is still a great product.

The outstanding power of ELS is obviously its searching proficiency. The ability to convert a user's request to something which Elasticsearch could understand, transform synonyms, configure or create its own analyzers and multiple languages support is what ELS is known for. Elasticsearch scalability is incredible, it is often managed on hundreds of machine clusters. ELS-related products are also awesome: Kibana, Logstash, and Beats. Elasticsearch is not a product that would make your application fast a reliable by itself but a solution that can solve many problems in the case is operated by intelligent developers and enthusiastic DevOps engineers.

4.2 Neo4j

In the previous section, the Elasticsearch product was accurately discussed. It has to be repeated that ELS is a great software. However, its impossibility to efficiently work with relations is critical in some cases. That is the reason why this thesis pays huge attention to Neo4j software. The aim of this chapter is to introduce the second database used in the practical part of the thesis;

thus a reader would be able to improve the implemented data management system or at least realize the platform advantages when using it as a data layer for an own application. Moreover, after reading the section it may become apparent which database to choose when configuring storage using the *database as a service* product implemented in the scope of this thesis.

4.2.1 Overview

Neo4j is a graph database created by Neo Technology company [74]. Neo Technology provides several modes of using Neo4j database:

- Community Edition
- Neo4j Aura
- Enterprise

Information used in this paragraph is taken from the Udemy course called *Neo4j: GraphDB Foundations with Cypher* [75]. A community edition is a limited approach that is not restricted to use by any rules. There are however some limitations compared to the Enterprise Edition, for instance, graph nodes amount, the number of relations, and properties are reduced. Community Edition does not make it possible to use any roles-based security mechanisms, and a possibility to restrict various properties' uniqueness is not provided as well. Neo4j Aura is the cloud version of the Community version. The Enterprise is the paid model which supports high cluster scalability, does not restrict the volume of a graph, and provides qualified support.

Being a graph database does not imply using an efficient graph-native store under the hood. Similarly, Neo4j at the beginning of its history implemented graphs serializing them into the MySQL database [76] and other not graph-native databases. Such an approach was definitely insufficient to process large graphs and traverse to a great depth; thus, Neo Technology invented and is still improving techniques to enable accessing a node's neighbors in a constant time.

Databases can be divided into 2 big categories: OLAP (Online analytical processing) and OLTP (Online Transaction Processing). The first type focuses on creating reports and running complex queries which are allowed to take some time. Databases from the OLTP category are responsible to fulfill simple business requirements and responding fast. Neo4j is able to run OLAP tasks and its team intends to improve such functionalities, however, this database is considered a pure OLTP engine [77].

The query language which Neo4j uses is called Cypher. This is a declarative way of working with graph data. Being inspired by SQL Cypher makes the process of manipulating data extremely easy and intuitive [78]. Let us recall once more an example of managing a graph of students who learn some subject

and a subject is taught by a teacher see Figure 3.1. If there is a requirement to retrieve all students who study the "Programming and algorithms" course the query would look like so see Code sample 4.2.

```
MATCH (s:Student)-[:LEARN]->(subj:Subject)
WHERE subj.name = 'Programming and algorithms'
RETURN s
```

Listing 4.2: Cypher query example

As demonstrated from the query example, the Cypher syntax is fairly easy to figure out in case a developer is familiar with SQL or SPARQL. Queries are created in the declarative style, thus, it is not necessary to manage the logic of data manipulation but only to declare the desired result.

One of the various Neo4j advantages is the ability to execute transactions. Moreover, Neo Technology developers built their graph database to implement all ACID features. The book *Graph Databases* points out that Neo4j does not lose its scalability efficiency that much as a consequence of following ACID principles [79].

4.2.2 Internal details

Neo4j Community Edition is open source, therefore its details in implementation can be found. However, there is still a lack of resources that would explain how traversal operations are accomplished so fast. In this section, some main techniques are introduced but the thesis does not dive into all architectural details.

As already mentioned, Neo4j optimizes its storage to be highly efficient to work with data in the graph format. One of the main terms which relate to these boost techniques is *index-free adjacency*. The idea of this practice consists in that each node points to its neighbors and no foreign keys approaches are used to implement relations between elements [80]. In case a graph database does not rely on native storage its traversal time complexity continues to grow accordingly to the graph expansion.

Neo4j implements its storage using files for nodes, links, properties, and labels [80]. Each node record uses a fixed-size memory chunk in a file, they are organized in the offset structure; thereby a node's location in a file can be retrieved only with the knowledge of its index. A node points to corresponding relations which then point to a node on the other side of a link [81].

4.2.3 Scalability and Federation

The previous chapter did not focus that much on how ELS implements a scalability concept. The reason for that is the relatively obvious techniques that

Elasticsearch uses. As already discussed document databases can simply divide an index as each document is an independent record. On the other hand, the Neo4j data model is highly connected and that is why horizontal scalability becomes a very fascinating problem and brings in many interesting ideas.

One of the clustering principles is storing a complete graph on each computing machine. Neo4j uses a master-slave paradigm when managing a cluster. One of the various possible configurations is assigning the write responsibilities to a master node, and write operations to slaves [82]. This topology ensures high read efficiency but does not focus that much on the performance of write operations.

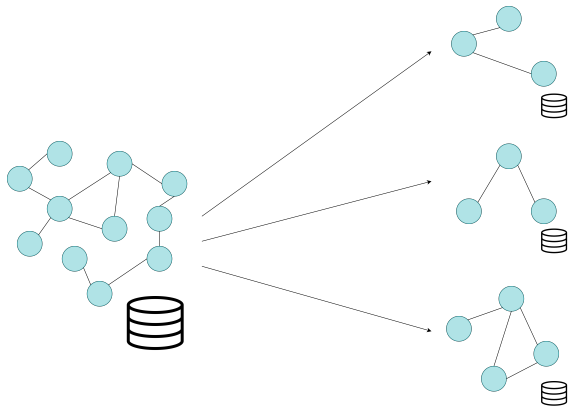

Figure 4.8: Graph sharding

When explaining ELS the thesis has paid attention to the idea of sharding. Neo Technology engineers have also developed their own technique of creating shards of a graph and distributing them based on the custom logic see Figure 4.8. By sharding a Neo4j instance it becomes possible to store location-oriented data closer to regions they would be requested; thus a latency could be reduced. Another sharding feature is that a large graph can be split into smaller ones; thereby higher throughput is ensured [83].

Neo4j allows not only to distribute a graph on several machines but also to "merge" graph databases into one virtual huge graph which could be then queried as a singular Neo4j instance see Figure 4.9. This technique is called *Federation* and enables a company to store data in separate Neo4j databases which could solve separate application requirements, but an ability to retrieve information from all Neo4j graph databases (thus from all company's graph data sources) remains [83].

Summarizing this section it can be declared that the scalability feature in the context of the graph data model is a much more complicated problem compared to the documentary approach and Elasticsearch especially. Therefore, it is important to honor the work Neo Technology engineers have made

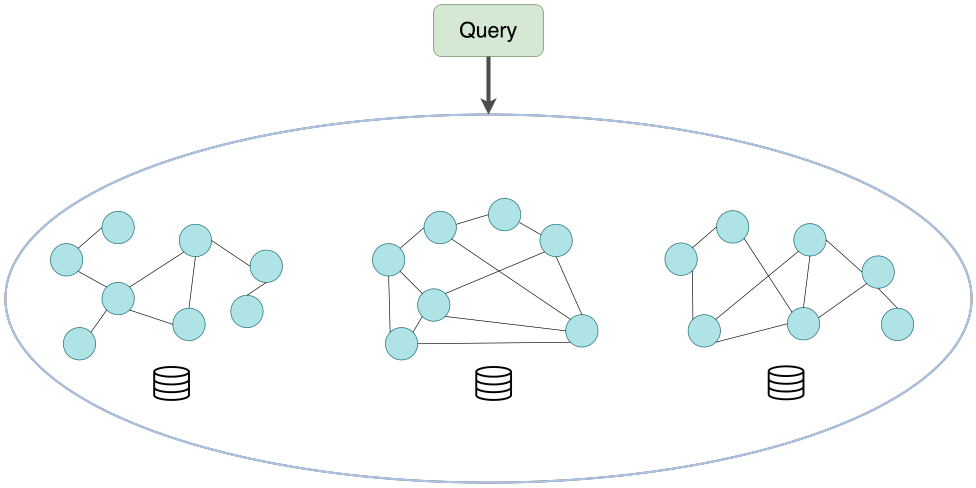

Figure 4.9: Graph federation

to make their database horizontally scalable. Also, the federation feature has to be admired as it brings much more flexibility to companies' data analysis.

4.2.4 Neo4j Browser

Neo Technology developers have implemented a solution called Neo4j Browser. This is a web application that connects to a running Neo4j server and manages it. Neo4j Browser is conceptually similar to Kibana, it solves similar problems and also produces data visualizations. The popularity of the graph paradigm stimulated the inventions of various visualization software as well as libraries for creating custom graphical solutions.

Neo4j Browser is fairly interesting to discover as it is more than just a tool for making requests. It is a good idea for beginners to play a bit with a graph using Neo4j Browser to become more familiar with the concept of nodes, labels, relation types, Cypher, and so on. The illustration of what a workspace of this visualization looks like is demonstrated in Figure 4.10.

4.2.5 Summary

In the previous chapter, Elasticsearch was discussed very conscientiously. The reason for that is that Elasticsearch design principles are often fairly original and sometimes quite complicated. Neo4j instead does not require such massive knowledge to use its features.

The detailed comparison of Neo4j and ELS is clarified in the next section, however, it is important to emphasize that Neo Technology invented a notice-

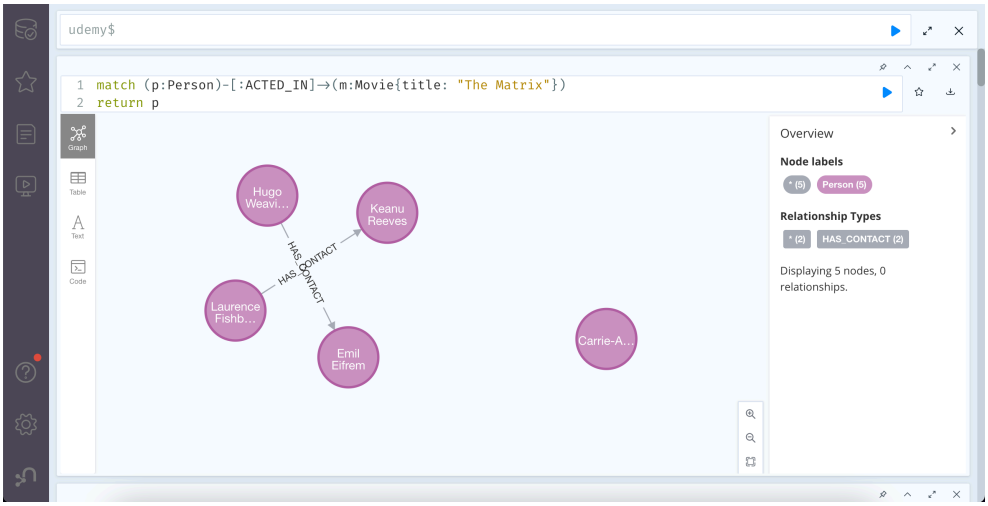

Figure 4.10: Neo4j Browser

ably powerful graph-native storage that is capable of managing graph data much more efficiently that aggregate NoSQL engines and RDBMS.

4.3 Elasticsearch versus Neo4j summary

The subject of comparing Elasticsearch with Neo4j can be treated as a comparison of document and graph models. However, as already mentioned, the graph database model does not imply any efficient native graph processing, as well as not every document data engine has all the set of features that ELS has. Taking this into consideration, this section analysis is focusing on these specific 2 databases.

The first Elasticsearch advantage is its ability to efficiently scale. Neo4j also implements some techniques to distribute its data over computing instances but the whole idea of storing data into a Neo4j database implies the concept of managing connected information which is possible to scale but is not as easy as in an ELS cluster.

Another ELS dominance over Neo4j is its searching functionality. If a developer has a requirement to search inside an unstructured text or to perform some kind of aggregations, Neo4j is evidently not the best candidate to do such kind of work. On the other hand, Elasticsearch uses many advanced data structures to handle these tasks.

Finally, to clarify Neo4j benefits over ELS it is necessary to mention that Neo4j supports all ACID properties while the document database does not support any of them. Elasticsearch implements a feature that enables storing of a document's version and to execute an update action only if versions would

match at the execution time, however, such behavior is much less powerful than ACID properties.

The second Neo4j advantage relates to its query language. Compared to ELS which uses a fairly complicated syntax applicable only to Elasticsearch, Neo4j supports Cypher which is extremely intuitive and can be used for working with many other graph data storages.

To summarize the material explained in this chapter it can be clarified that there is no better or worse database when talking about Neo4j and Elasticsearch. Each one is good for its purposes. When data is perceived as a graph and it is necessary for a business to constantly traverse this graph, then Neo4j is the obvious choice. On the other hand, if data can be decoupled, and stored in separate records and there is no need to work with any ACID property, then ELS is more suitable. Each case is different, the conclusion of this chapter is that it is highly important to identify actual and potential requirements to choose the most applicable data engine which will fulfill all needs of a system during its whole life cycle.

Chapter 5

Practical part

The practical part of the thesis navigates a reader through the process of building a web application using Elasticsearch and Neo4j. The aim of this chapter is to ensure that working with non-traditional databases is not complicated at all and can be much more convenient than building an information system using a relational approach.

5.1 Motivation

Being mobile or a frontend developer is cool, however, such a job implies several disadvantages. For instance, an ambition to embed modern database solutions may be extremely hard to implement. The reason for that is the necessity to spend some time figuring out how to configure a Neo4j instance or an Elasticsearch cluster. Furthermore, an engineer also has to gain some theoretical knowledge about the ELS query language, datatypes, analysis, and much more.

The implemented application is a Database as a service product. A reader may notice such a drawback that this system does not has all Elasticsearch and Neo4j capabilities. For instance, there is no possibility to add an Elasticsearch instance and reindex all data to become distributed over all instances. It is also impossible to define a strict schema to restrict adding new fields to an index.

However, this data management service enables all basic functionalities which a developer may require. Exposed REST endpoints are capable of all CRUD operations. The huge strength of this product is that a user does not has to really care about which database he is manipulating, the main choice has to be made is selecting connected or non-connected records type. Therefore creating a database for a frontend or a mobile application becomes extremely easy and separates a developer from server-side programming, connecting a database to a backend and making him/her ability to focus on client-side programming and managing data through an intuitive set of REST

endpoints.

The core idea of this data management platform is to make a user's interaction with modern databases absolutely simple. In the previous chapter, we discussed several NoSQL models and corresponding database products. That was fairly important because the database platform which is described in this chapter is built on top of Elasticsearch and Neo4j. Therefore only a competent reader may realize this system's advantages and drawbacks. Moreover, this database service forces a user to select a database due to his requirements; thus, being aware of different models and specific databases are highly recommended.

5.2 Design

The application architecture consists of 3 logical parts: frontend side, backend, and a database layer. Additionally, the data management component is divided into Elasticsearch and Neo4j parts see Figure 5.1.

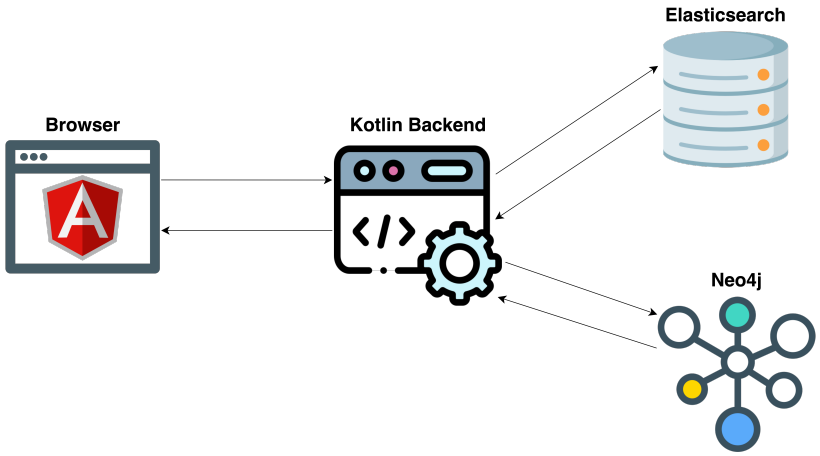

Figure 5.1: Data management application design; Kotlin Backend icon [84], Neo4j graph icon [85]

A user is able to create 2 personal databases using the visual layer of the system. The main benefit of the application is the possibility to configure and create document and graph storage without any interactions with containers, virtual machines, or any kind of Linux server configuration. All setting up procedures are made by working with the browser UI.

The frontend communicates with a server-side component that handles requests and allocates Elasticsearch indexes, configures schema, or authenticates a user. This part of the system is written in Kotlin using the Spring framework. After a configuration process is completed a backend exposes REST API which makes the communication with personal data storage possible.

The authorization mechanism is implemented by creating 3 types of tokens: access tokens for frontend-backend communication, access tokens for accessing Elasticsearch API, and tokens for working with Neo4j API. The frontend layer communicates with the backend using access tokens, refresh tokens, and a *time to live* parameter which specifies the date and time when a token should expire.

After a user completes its configuration through a web UI a backend enables a client to manipulate data through a set of REST API endpoints. The contract that a server exposes for data management is explained attentively in a later section. The intention was to make communication rules as intuitive as possible to create a platform that every developer could use without reading any kind of documentation or a help page.

The distribution of database resources that are allocated to the final users is implemented in the *Shared Everything* way. The system uses 1 Elasticsearch cluster and 1 Neo4j database which is logically divided between all clients. This model has its disadvantages which are discussed later.

5.3 Frontend layer

In this section, the motivation for configuring storage through a graphic interface is demonstrated. After that, more technical details of the frontend implementation are explained.

5.3.1 Introduction

One of the most essential ideas of this project is to create a "clickable" visual layer that enables a user to configure a database through a graphic interface. The advantage of this approach is that there is no necessity to perform any configuration actions using REST endpoints or connecting to a server via SSH, a client has the ability to make all needed operations as easy as possible.

The visual layer of the platform is a web application. Implementing the whole application in pure Javascript would be an extremely irrational decision as modern frameworks allow engineers to focus on implementing application logic and make a frontend architecture much more modular. As a result, implemented components can be easily used in different parts of a project. Furthermore, many common functionalities are already prepared in popular Javascript frameworks. For instance, there are many libraries for managing an application state, a lot of components for making HTTP calls, and much more.

The web interface for this project is programmed very qualitatively. The latest version of Angular is used and the whole architecture is built using the enterprise development best practices. However, the design of the GUI has to be rebuilt as the style does not has a high priority in the scope of the thesis.

5.3.2 Architecture

The frontend part of the project is built using a popular Javascript framework called Angular. The scope of this thesis does not include discussions and comparisons of Javascript technologies. However, the subjective opinion of the author is that Angular is a robust, complex, and extremely well-developed framework containing many related libraries which makes a SPA (Single Page Application) development very structured.

The principal aspect of any Angular application is custom components. These pieces of a UI interface represent independent HTML blocks with assigned Typescript logic and CSS styles. Custom components may be reused in different parts of an application. Another interesting part of the framework is services. These are singleton objects created from the programmer's defined classes, they are accessible from any component or another service by constructor injection. Considering that services and components are independent pieces of functionality, this brings in the issue of sharing data between them. This problem can be resolved in different ways, nevertheless, the next paragraph clarifies the most widely used and the most fundamental concept.

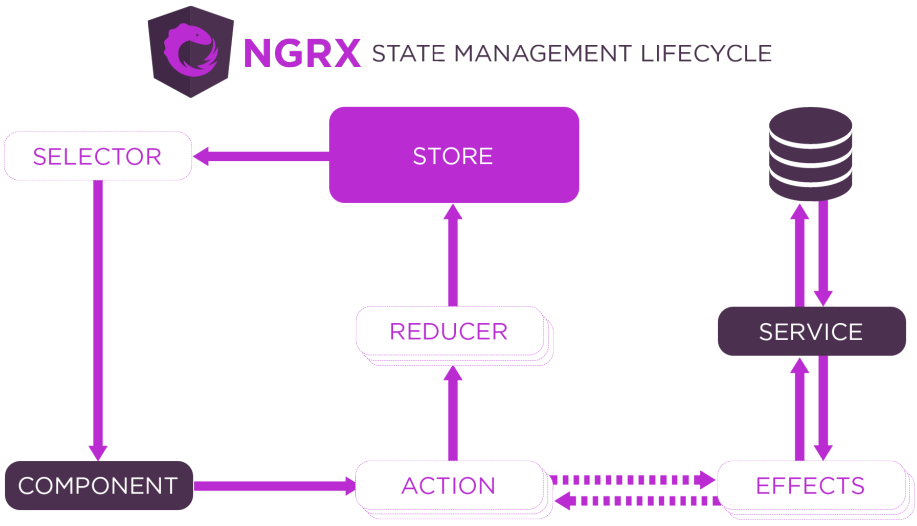

Figure 5.2: NgRx state management architecture, from [86]

The main client-side problem faced during the implementation is managing a client's state. Therefore the Angular application is built using the **NgRx** library. This technology operates with the concepts of actions and reducers see Figure 5.2. The diagram may seem to be overly complicated considering the fact that the library just stores and share a complex JSON between different Angular services and components. However, following this design

approach force a developer to write a much cleaner code, thus maintaining and developing new functionalities becomes an easier task. A programmer defines **actions** and their parameters which are just function objects with some data chunks. After that a **reducer** part has to be implemented, this can be perceived as a switch-case block that receives an action, determine the action type and process received data due to defined logic. A receiver usually updates a state based on data collected from an action. When a component or a service needs to change

5.3.3 Functionality

The first functionality a user experiences when working with this platform is the authentication process. There is a login/sign-up form that communicates with the backend. After that the application store all tokens in case of successful authentication. The Angular application stores an access token and refreshes one, the frontend is programmed in such a way that there is no necessity to log in every 5-50 minutes. Instead, an automatic refresh functionality is implemented to provide an authenticated user with current tokens. There is also a possibility to log out with one click and return to the initial application state.

After being authenticated a user is able to configure two data storage. The extremely important benefit of this project is that a user does not need to know anything about Elasticsearch or Neo4j. The UI offers a user to work with *non-connected* and *highly-connected* data. Thus, this platform is suitable not only for engineers who have a requirement to elevate an Elasticsearch/Neo4j storage because of these technologies' advantages but also for those ones who identified their necessity to use any graph or document-oriented database system.

An authenticated user has the capability to configure a *non-connected* data source. The main aspect of such configuration is setting up a data schema. The graphic interface enables a client to define and change a schema. After submitting a schema user receives an index name, access token for managing *non-connected* data, and a set of endpoints with some description about how to use them. A client also has the ability to redefine or change its schema even after a database establishing and filling it with some data but it is necessary to take into consideration that after changing a schema all stored records would be lost.

The second data source is not that configurable. As the *highly-connected* data source uses a Community version of Neo4j under the hood it is impossible to define any schema on the database level. Thus, a user has only an opportunity to create a set of endpoints with an access token to work with a personal schema-free graph.

5.4 Backend layer

In this section, some internal details are provided. The complete REST API definition is also discussed to facilitate working with it for potential service consumers.

5.4.1 Architecture

The server-side component of the system is implemented as a monolithic Kotlin application. The main task of the implemented backend is to handle HTTP requests, communicate with a data source, and return a response. Spring is the framework that was chosen to perform this kind of work. It is famous to be highly popular in the Java world, as a result, this is a very mature technology with a huge community and many related libraries.

The backend application can be logically divided into 2 parts: the first one works with REST API used by a frontend to configure data sources, and the second component is responsible for exposing REST API for accessing data through established endpoints. A detailed description of the API is provided in the next sections.

One important limitation is that a space that is allocated for a client in the Neo4j database is not a separated resource, nodes and relations are divided logically by the *owner* property which each node and relation has to have. Nodes and relations of all tenants are located in the same graph. A client does not have to be aware of this design, it is not necessary to change queries. All transformations are made on the backend side, queries are reconstructed to return, change, create or delete only graph components to which a client has access. After this transformation, a query is executed and results are returned to a platform user.

All queries' transformations are performed using regular expressions. That would be unfair to declare that this solution is easily maintainable are intuitive to read. A much cleaner way of implementing the data separation would apply some sort of database isolation with roles assigned to clients. More detailed discussions about potential improvements are demonstrated in later sections.

Electing space in the Elasticsearch cluster is a much more trivial task than solving the same issue in a Neo4j database. Creating a non-connected database on the frontend leads to creating a new index with provided schema. The implemented REST API for client consumption does not cover all possible Elasticsearch functionalities, however, all basic operations are supported. After receiving a request the backend service checks the access token and builds a query that is executed on a specific index. The details of a defined ELS contract are demonstrated below.

There are for sure some defects in the server-side architecture. For instance, the backend can be divided into microservices, there is also a lack of CI/CD, and much more. Nevertheless, the system is completely functional

and operates in the real Cloud environment. All deficiencies are analyzed to prepare the system to be fixed in the future.

5.4.2 Implementation details

It was already mentioned that an ELS cluster has two purposes. The first one is storing users' data to manage data sources access permissions. For this intention, a *user* index was created. The "test" user's data is stored in such was see Figure 5.1. This JSON structure is fairly intuitive, a user with configured data sources has a username, password, and access/refresh tokens which secure the frontend-backend communication together with a *tokenTimeToLive* value. There is also stored information for communicating with Elasticsearch REST API which are an index name, an access token for accessing a document data source, and a serialized schema. A *schemaLocked* field has the purpose of preventing schema changes, this parameter is mainly used to lock the "test" user's schema. The last property is the Neo4j *accessToken* which shares the same idea as the token for an ELS data source access.

```
{
    "_class" : "cvut.fit.holovser.model.els.User",
    "id" : "MpxkfIEBKxXdnAjjKTJN",
    "password" : "test",
    "auth" : {
        "accessToken" : "t7Bg9WEGsNGlDiAOOOC9XExfKAuP-hQ2",
        "username" : "test",
        "tokenTimeToLive" : 1655747310,
        "refreshToken" : "t9zTn8eT7BkCNbGfPGNbJS8Z9EUl_X5T"
    },
    "els" : {
        "index" : "test",
        "accessToken" : "erFSgOY9Pq-pParFZCUK8X-u4wzweGah",
        "schemaLocked" : false,
        "schema" : """{"name":"as-is text","age":"integer"}"""
    },
    "neo4j" : {
        "accessToken" : "DVxT7txJVLCnqgLU-UK6cCHtp4IzbGgp"
    }
}
```

Listing 5.1: User's data in the Elasticsearch

Working with the Elasticsearch cluster from a Kotlin/Spring environment is an effortless task. To access the *user* index the Spring Data framework was used, which enables a developer to define a DAO interface see Figure 5.2

and Spring provides the specific objects which are accessible from the Spring container in the runtime.

```
interface UserRepository: ElasticsearchRepository<User, String>
{
    fun findByAuth_Username(username: String): User?
    fun findByAuth_AccessToken(accessToken: String): User?
    fun findByNeo4j_AccessToken(accessToken: String): User?
    fun findByEls_AccessToken(accessToken: String): User?
}
```

Listing 5.2: Kotlin DAO interface

The REST API exposed to clients does not use the DAO concept explained before. The reason for that is that Spring Data requires a developer to specify the Kotlin/Java class to which a response would be mapped. However, in our case, the data platform is going to be used by many users. Each one would define its own schema; thus it is impossible to map an ELS response to an object. Given the above access to a cluster is implemented using a fairly intuitive builder API developed by the Elasticsearch team see Figure 5.3.

```
var boolQuery = QueryBuilders.boolQuery()
...
boolQuery.must(
    QueryBuilders
        .rangeQuery(key).gt(valueObj?.gt).lt(valueObj?.lt)
)
...
val builder: SearchSourceBuilder =
SearchSourceBuilder().query(boolQuery)
val searchRequest = SearchRequest(index)

searchRequest.searchType(SearchType.DFS_QUERY_THEN_FETCH)
searchRequest.source(builder)
val response: SearchResponse =
client.search(searchRequest, RequestOptions.DEFAULT)
```

Listing 5.3: Elasticsearch builder example

The Neo4j cluster access is realized with the Driver object from the *org.neo4j.driver* package. The Spring Data library also provides the DAO functionality for dealing with Neo4j. Due to the system's design aspects queries are constructed by converting original Cypher queries and that is the reason why all requests are sent in such an uncommon way see Figure 5.4.

```
driver.session().use { session ->
    session.run(query)
}
```

Listing 5.4: Neo4j Driver example

5.4.3 Frontend REST API description

POST /user: the endpoint for registering a new user. A request body has to contain a JSON with username and password see Example 5.5. Returns an access token that enables a frontend to communicate with a backend and configure data sources.

```
{
    "username": "holovser",
    "password": "mysecurepassword"
}
```

Listing 5.5: Register/Login JSON example

POST /session: the endpoint for logging a user in and returning an access token to perform further configurations. The request body remains the same as in the previous endpoint see Example 5.5.

POST /index: creating a non-connected data source with a schema defined in the request body. The JSON example is depicted on Listing 5.6. The request has to contain an "AccessToken" header.

```
{
   "elsSchema": {
        "name": "as-is text",
        "age":"integer"
    }
}
```

Listing 5.6: Schema JSON example

GET /index: fetching information about a non-connected data source. The response example is demonstrated on the Listing 5.7. The request has to have an "AccessToken" header.

POST /graph: creates a highly-connected data source. The request body should be empty. The "AccessToken" header is mandatory.

5.4.4 Data sources REST API

All endpoints described below have 1 important restriction. Each of them has to have the *AccessToken* header with a token generated during the process of

```
{
   "index":"1",
   "accessToken":"IBVO1BaoWpeg3gVlkjNV4wAQnMx9ra4p",
   "schemaLocked":false,
   "schema":"{\"name\":\"as-is text\",\"age\":\"integer\"}"
}
```


Listing 5.7: Non-connected data source GET response

establishing a data source. Otherwise, a request would receive a 403 response.

POST /api/els/index/{index}: retrieve records from a non-connected storage. A client has to specify an index name as a path variable. This endpoint violates W3C rules for web services contracts as it uses the POST method for a fetching method which is also safe and idempotent, however, the reason is that implementing a GET function with a query as an URL parameter also has several limitations. Such a method with a long and complex URL is not easy to read, debug, and a query becomes restricted by length. Given the above, the endpoint is a POST method that consumes a query in the request body see Listing 5.8.

```
{
    "name": {
        "value": "Mike",
        "type": "match"
    },
    "age": {
        "gt": 20,
        "lt": 40
    }
}
```

Listing 5.8: Non-connected data retrieval POST request

The first-level keys are a document keys whose value has to match. A complex objects can be queried by specifying a key with the dot notation, for example: *job.salary* to query an object with such structure `{"name":"John", "job": {"position":"manager", "salary":"3000"}}`. Each key in a query has a corresponding value, a value can be of 2 types: string or numbers querying. In case of a string a value part has to be in the format: `{"value": "Mike", "type": "match/term"}`, the *match* part specifies if the requested text has to analyzed or not. The second type of a queried value is the "number object": `"age": {"gt": 20, "lt": 40}`. "gt" field stands for greater than and "lt" for less than. The response contains a JSON of documents array.

PUT /api/els/index/{index}/doc/{docId}?fieldName="name" &fieldValue="value": changing a *fieldName* of a document. Supports a dot notation for a *fieldName*. Index and docId path variables are mandatory.

POST /api/els/index/{index}/doc: creates a new document in the *index*. The backend does not assume receiving any body in such a request.

DELETE /api/els/{index}/doc/{docId}: deletes a document by its id from a specified index. There should not be any body in a delete request.

5.4.5 Graph REST API

Similar to the previous section each request which tries to access a graph data source has to contain a generated AccessToken for a highly-connected records storage. The logic of diving a graph space into separate areas is explained in the next section. It is important to mention that compared to document storage, REST API implemented for highly-connected data can be considered as a proxy service that runs sent Cypher queries on a Neo4j instance. However, there are still some important features, described in the next section, which make the platform extremely valuable.

POST /api/neo4j: this endpoint receives a request which contains a JSON body with a *query* field and executes a MATCH query:

```
{
    "query": "match (p:Person)--(m:Movie) \n return m.title, m"}
}
```

POST /api/neo4j/record: creates new data in Cypher language:
`{"query": "CREATE (n:Person {name: 'John'})"}`

DELETE /api/neo4j: deletes records and relationships of a graph:
`{"query":  "MATCH (p:Person{name: 'Andy'}) \n detach delete p"}`

PUT /api/neo4j: updates nodes/relations using the SET Cypher keyword:

```
{
    "query": "match (p:Person) \n where p.born < 1940 \n
    set p.status='old' \n
    return p"
}
```

5.5 Database layer

In this section, internal details of how REST calls to a backend are processed to cooperate with a Neo4j and Elasticsearch database are processed. Also, the design of separating a space in these two engines is explained.

5.5.1 Elasticsearch

The Elasticsearch used for this thesis has two purposes. The first one is managing information about users, their passwords, and indexes. The second function of the cluster is to handle client data.

A reader may notice that non-connected storage REST API functionality does not cover all Elasticsearch capabilities. This limitation is made deliberately to make a user's interaction with it as easy and intuitive as possible.

The approach used for dividing clients' data sources across a cluster is creating an index for each user in case he needs non-connected storage. Such a technique makes it possible for each client to define its own mapping (schema) as this restriction is configured on the index level.

Assigning an index for each client is a quite scalable approach. As already discussed in the theoretical part, indexes can be easily divided into shards and distributed across several ELS instances, thus even users with a requirement of storing high volume data can use the platform.

There is also an important issue that, however, has the ability to be solved. As for now, there are no system load restrictions which means that one client can overwhelm the whole cluster by actively working with its index. This issue can be solved by assigning and regulating requests per hour quota of each user.

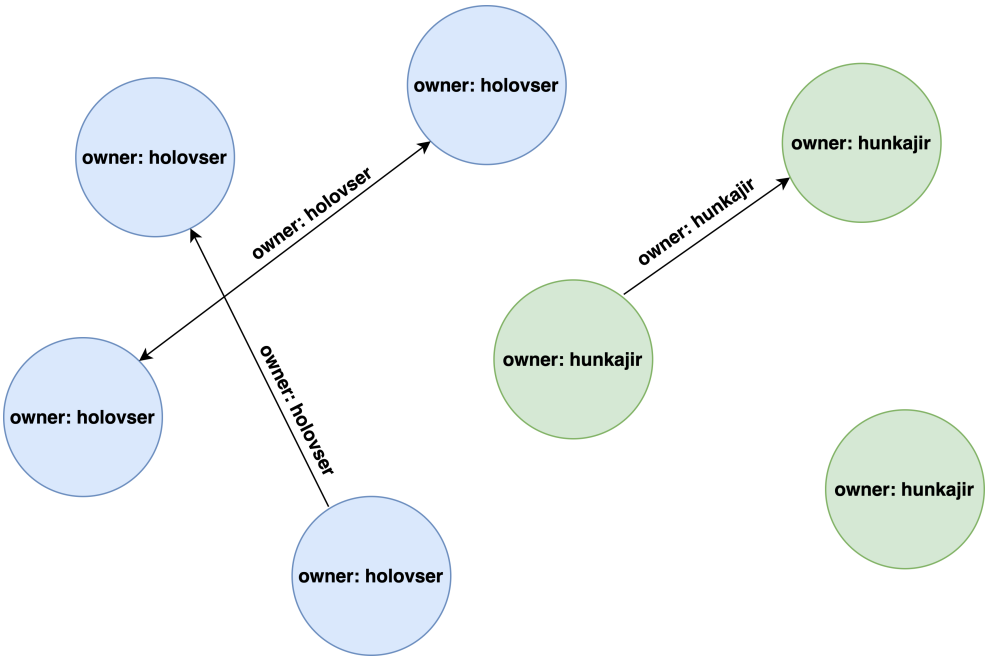

Figure 5.3: Graph sharing illustration

5.5.2 Neo4j

One of the biggest problems experienced during the platform implementation is the issue of sharing a Neo4j cluster with many tenants. The initial idea was to create a separate graph database at the runtime for each client. Then it was discovered that this approach is not possible to implement as it was decided to manage the cluster in the Cloud service called Neo4j Aura. Its users receive 1 free database after a registration process but there is no API to create another paid one at a time when our database platform clients would need some. Another thought was to create separate graphs inside one Neo4j database and this option is also not possible in Neo4j. There are for sure solutions how to divide a graph space by working with containers or isolated VMs, such redesign strategies are discussed in later sections, however, that is not a task one engineer can handle in a short period of time, thus the problem was handled using simpler strategy.

The graph layer of the platform consisted only of 1 Neo4j instance. At this point in time, a reader has to realize that a graph in Neo4j contains nodes, each one having labels and properties. There are also edges with can hold key-value data as well as nodes. This fact was a motivation for assigning each node and edge a property *owner* which keeps the information about a client to whom every graph component belongs, see Figure 5.3.

Speaking about the potential scalability the platform has 2 options. The first one which is discussed below consists of creating a separate Neo4j container for each user who will need a highly connected data source. This option is considered the main one because of its clients' isolation benefit. However, there is another way how to scale this solution, concerning that the Neo4j database is a collection of disconnected graphs the sharding technique can be potentially used to distribute nodes and corresponding links based on their *owner* property.

5.6 API access

Another important subject is how to access, test, and consume a configured REST API endpoints set. It is assumed that a final user would create a mobile or a frontend application that would manage its data via our database as a service implemented interface.

There was created a Postman collection of 2 folders that contains prepared requests for a graph and document API to see Figure 5.4. The user with a *test* username and a *test* password was created and his access tokens are filled in the Postman requests' headers. This collection is exported and provided in the GitLab repository as well as on the SD card. A reader may import requests into a personal environment and confirm that a system is truly operating by running them.

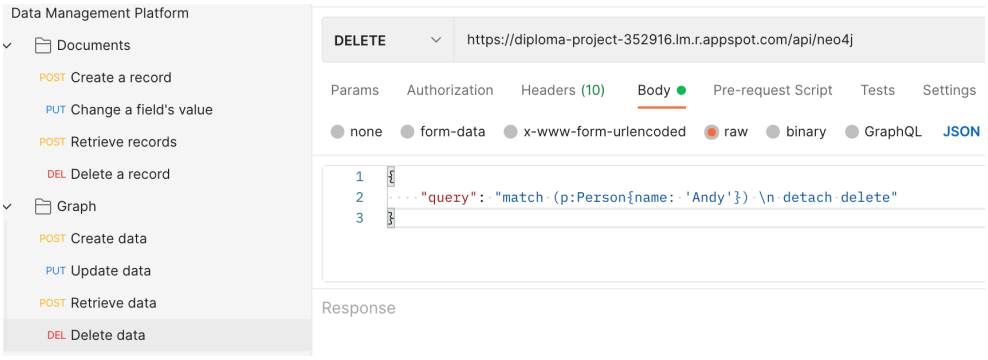

Figure 5.4: Postman requests illustration

5.7 Deployment

The biggest victory of the practical part is the fact that the whole project was deployed and is running in the real Cloud environment. Moreover, different parts of the system use different providers which for sure deserves attention. It is essential to emphasize the fact that comparing different Cloud providers is not a subject of this thesis, only essential details and some impressions are shared in this section.

The first thing to mention is that the Elasticsearch employees are great. The support team gave this project a 150 days license which makes it possible to run the cluster for free. ELS infrastructure is operating in the Elastic Cloud environment, however, under the hood clusters are still using AWS/Google Cloud/Azure. For this project, Google Cloud was chosen.

For the reason that the backend was implemented in Kotlin, it was required to run the service in JWM. The App Engine solution was chosen. It is a product that is developed in the scope of the Google Cloud. The App Engine provides a CLI for deploying new versions of an application in one click which does not bother a developer with connecting to a server by SSH, transferring a JAR or a WAR to the file system, and stopping/starting a JWM manually. Everything a programmer has to do is to build a project with any assembling tool he/she likes (Maven in the case of this project) and run the deploy command.

The last part of the project is the frontend application. Despite that it was developed in Angular the compiled product is just a connected set of HTML pages, Javascript, and CSS. The frontend was deployed to the Firebase environment because of the ability to host web pages for free. The deployment process is also configured to be performed by executing a deploy command from the Firebase CLI package. This hosting service is also a Google product, therefore the whole project is operating in the Google environment. The URL

assigned to the frontend service is: application-link

The important point to mention is source code access. This thesis does not focus on GIT details or versioning platforms. Source files of this project are hosted using the GitLab software. The link to the project's resources is–Git-Repository.

5.8 Potential improvements

Implementing a production-ready database as a service platform is a complex task that is usually realized by a team of developers. That is a reason why the final product can be considered either as a proof of concept or as a minimum viable product. In this section improvements of different project parts are explained to specify necessary actions to release a more advanced version of the project.

5.8.1 Frontend improvements

The front-end improvements necessity is truly obvious. The first weakness of the GUI component is its visual part. As this thesis focuses on the databases theory, designing an attractive interface does not have a high priority. There is also a lack of mobile screen adjustment. The potential solution could be buying a design through a freelance platform and implementing it in Angular.

Another GUI drawback is that no usability analysis is provided. The potential steps to realize are completing a heuristic evaluation and to perform the cognitive walkthrough process with at least 7 users of different "personas" types.

There is also an improvement opportunity regarding the authorization process. That would be highly engaging if a user can sign up or log in with his Google or Facebook account. Thereby the registration process would be more pleasant for clients and more people would start using the platform.

5.8.2 Backend improvements

The server-side logic of the platform is implemented using popular and fairly promising technologies: Kotlin, Maven, Spring Boot, and Spring Data. Nevertheless, there are still some drawbacks that have to be remodeled.

The first aspect of potential improvements is dividing the backend into several services based on their responsibilities. There is for sure a necessity to create a separate component that would handle only authentication functionality and work with its own database of users' credentials. It is also important to build a service whose responsibility would be only handling requests related to the database configuration process. The next microservice purpose would be processing Elasticsearch REST API requests, similarly, the last backend component could handle an HTTP interface for working with Neo4j. Such

microservice architecture would be much more scalable, easily maintainable, and simpler to develop. It has to be also considered that a distributed system of such kind would need a lot of DevOps engineering capabilities to cope with microservices communication, networking issues, a CI/CD functionality for each service, and much more.

It was already mentioned but that would be great to emphasize this problem again that the platform lacks CI/CD automation. Even though the server-side functionality is built in a monolithic way it uses GitLab which makes it potentially possible to set up the build and deployment process as an instant operation after pushing a commit to a specific branch.

A reader may also notice that a REST API contract defined for working with the Elasticsearch cluster does not cover all functionalities which ELS provides. The backend interface was designed based on the author's experience in web applications development, thus the requirement was to support 90% small/middle projects needs. The intention for a future evolution is to collect feedback from different engineers and define the most critical functionality absence and implement it afterward.

The last potential upgrade which can make the further development process more efficient is establishing at least 3 deployment environments, each having its own frontend service, backend service, and databases. The first one should provide an ability for developers to check and briefly test an implemented logic. The seconds one can be dedicated to a testers team and can be also perceived as a last-before-production environment. The last deployment space can be an infrastructure for a well-tested and production-ready product.

5.8.3 Databases improvements

The biggest data layer improvement potential corresponds to the issue of dividing data between multiple tenants in Neo4j. As for now nodes and edges are logically separated by the *owner* property which each graph element has. This method forces working with regular expressions and query transformations that have the potential to execute a query injected code. The intention for future development is to deprecate this way of data separation. The initial plan consists in creating a complex infrastructure that would make it possible to set up a container with a Neo4j instance at the runtime. Thus a client could request a highly-connected data source which will cause a completely isolated data source creation. This approach also solves the problem that now a platform client may simply overload a system and make other users' operations less efficient than expected.

The Elasticsearch infrastructure configured for this project is just a one-instance cluster that does not have any replication capabilities because of such a simple configuration. The first ELS improvement would be setting up an auto-scale feature to adjust the shards amount and size accordingly to users' requirements growth. The second Elasticsearch improvement was

already introduced, and the quota restrictions have to be implemented to restrict users from overloading the platform. The *requests per hour* value will be assigned to each user to prevent all system's clients from unexpected response delays.

5.9 Summary

The first thing to mention after finishing this project is that in 2022 putting ELS or Neo4j into operation is a fairly easy task. The existence of Cloud services makes it the software company possible to establish its NoSQL infrastructure without the need to invest a tremendous amount of money. Employing these contemporary data engines into code is not a complicated issue as well, many intuitive libraries have been implemented by ELS and Neo4j teams; thus backend developers' job does not become more sophisticated in case there would be a necessity to start implementing new business logic using these non-traditional databases.

Another aspect that was experienced during the platform completion is that Neo4j and Elasticsearch technologies indeed offer a great scalability power. Even though the first version of an application can be developed as a system built on a single instance of an ELS/Neo4j database, it still can be scaled in the future without a huge effort. That is a great benefit compared with traditional RDBMS approaches which can be scaled only vertically.

The contradictory aspect of using these graph and document databases is the necessity to employ a DevOps engineer or even a team in case of a big project. The set of configuration details that stands behind setting up an efficient ELS/Neo4j cluster is tremendous. Small or medium-size projects can easily ignore the majority of them, but big companies have to ensure their DevOps engineers comprehend specific aspects that a modern database brings in.

Given all the above, companies could consider NoSQL products before building their IS with relational databases. This thesis does not claim that Elasticsearch, Neo4j, or any other NoSQL database is a must for every project. It rather accentuates the fact that many companies use an SQL model even though the architecture design process shows from the beginning that a NoSQL model would be more suitable.

Conclusion

The IT industry is continuously evolving fulfilling emerging requirements. Databases play a key role in building information systems. That is why NoSQL innovations play such a prominent role. This thesis carries a reader through the database theory, from the relational approach, through the NoSQL technologies to their practical implementation.

The goal of this thesis was to clarify how are present-day database technologies evolving. Thereby their relational predecessor is introduced.
RDBMS is also explained to introduce issues that database systems face as transactions, ACID properties, and horizontal scalability. These topics are demonstrated in the context of non-traditional data engines afterward.

After becoming familiar with a relational approach the theory behind NoSQL databases is explained. Many modern concepts such as Big Data, CAP theorem, sharding, and replication are defined. Such academic comprehension makes a reader possible to dive into the NoSQL paradigm and perceives contemporary database models in the next sections in a more fundamental way.

The next part of the thesis carries a reader through 4 modern database models. Ideas and a theory behind each approach are described. The most popular implementations of each model are demonstrated. This chapter connects specific databases with problems that this data engine can solve. As a result, a reader becomes qualified enough to potentially take into consideration a specific database technology and identify the system's capabilities.

In previous chapters, a reader acquires a solid knowledge about problems NoSQL databases are capable of. After that, the substantive explanation of the 2 main documents and graph-oriented data engines are demonstrated. These NoSQL products are Elasticsearch and Neo4j. Each technology is carefully analyzed and its characteristics are evaluated. The comparison of these 2 databases is clarified as well.

Elasticsearch and Neo4j are demonstrated in the real project example. The system is designed and its alpha version operates in the Google envi-

ronment. Such demonstration assured us that working with contemporary database technologies is not a complicated process. The second benefit of this project is that it provides an alpha release of the NoSQL data management platform. This product enables programmers to incorporate contemporary database capabilities into their frontend/mobile applications. It is also admitted that the final platform is imperfect. That is the reason why several sections focus on aspects of the potential project's improvement.

The databases academic field is tremendous. It evolves constantly and brings new features and modern points of view to the subject of managing data. However, this thesis does not accentuate the fact that the traditional SQL approach is outdated. Because relational management systems cover most of the information systems requirements, the SQL paradigm remains the number one choice in the database world.

At the same time, many innovative solutions have already emerged and make IS systems capable of fulfilling various requirements in situations where SQL databases are out of their capacity. The thesis call a user's attention to the fact that many NoSQL products are fairly mature; thus their integration into a project is supported by plenty of supplemental libraries and tutorials. Even though contemporary non-relational data management systems are not a panacea for all database problems, they are often a good fit for many demanding scenarios.

Appendix A

Acronyms

SQL Structured Query Language

NoSQL Not only SQL

HTML HyperText Markup Language

ACID Atomicity, Consistency, Isolation, Durability

OLTP Online Transaction Processing

XML Extensible Markup Language

JSON JavaScript Object Notation

YAML YAML Ain't Markup Language

CAP Consistency, Availability, Partition tolerance

BASE Basically Available, Soft State, Eventually Consistent

AWS Amazon Web Services

ELS Elasticsearch

RDF Resource Description Framework

IRI Internationalized Resource Identifier

DevOps Development Operations

OLAP Online Analytical Processing

CRUD Create Read Update Delete

REST Representational state transfer

API Application Programming Interface

UI User Interface

CLI Command-Line Interface

SSH Secure Shell

HTTP Hypertext Transfer Protocol

GUI Graphical User Interface

CSS Cascading Style Sheets

CI Continuous Integration

CD Continuous Delivery

DAO Data Access Object

URL Uniform Resource Locator

SD Secure Digital

JWM Java Virtual Machine

JAR Java Archive

WAR Web Application Archive

Appendix B

Contents of enclosed CD

- `README.md` ..... the file with SD contents description
- `Serhii_Holovko_Thesis.pdf` ..... the Master's Thesis text
- `be` ..... the directory of backend source code
- `be_documentation` ..... the directory of backend documentation
- `fe` ..... the directory of frontend source code
- `postman` ..... the directory with the exported Postman collection

Bibliography

1. RAMAKRISHNAN, Raghu; GEHRKE, Johannes. Database Management Systems: NEW Material on Database Applications. In: [online]. 3rd ed. 2003, pp. 6–7, 524 [visited on 2022-02-04]. ISBN 0-07-246563-8. Available from: https://github.com/pforpallav/school/raw/master/CPSC404/Ramakrishnan%20-%20Database%20Management%20Systems%203rd%20Edition.pdf.
2. Relational Database. *IBM100* [online] [visited on 2021-11-19]. Available from: https://www.ibm.com/ibm/history/ibm100/us/en/icons/reldb/.
3. POKORNÝ, Jaroslav; VALENTA, Michal. Databázové systémy. In: 1st ed. Czech technical university in Prague, 2020, pp. 186–188. ISBN 978-80-01-06708-6.
4. VALENTA, Michal. Transakce a transakcní zpracování [online] [visited on 2021-11-23]. Available from: https://courses.fit.cvut.cz/BI-DBS/materials/slides/hand-les08-transakce.pdf. Czech Technical University in Prague.
5. ABADI, Daniel J. Demystifying Database Systems, Part 1: An Introduction to Transaction Isolation Levels. *fauna* [online]. 2019 [visited on 2021-11-23]. Available from: https://fauna.com/blog/introduction-to-transaction-isolation-levels.
6. DBMS - Data Recovery. *tutorialspoint* [online] [visited on 2021-12-06]. Available from: https://www.tutorialspoint.com/dbms/dbms_data_recovery.htm.
7. VALENTA, Michal. ACID implementation in RDBMS [online] [visited on 2021-12-06]. Available from: https://courses.fit.cvut.cz/NI-PDB/materials/acid-implementation-lesson/rdbms-architecture-acid-implementation.pdf. Czech Technical University in Prague.

8. DB-Engines Ranking. *DB-ENGINES* [online]. 2021 [visited on 2021-12-07]. Available from: https://db-engines.com/en/ranking.

9. REHMAN, Asif. Horizontal Scalability Options in PostgreSQL. *HIGH GO* [online]. 2021 [visited on 2022-02-20]. Available from: https://www.postgresql.org/docs/9.4/datatype-json.html.

10. SVOBODA, Martin. Advanced Database Systems [online] [visited on 2022-02-01]. Available from: https://www.ksi.mff.cuni.cz/~svoboda/courses/201-MIE-PDB/lectures/MIEPDB16-Lecture-01-Introduction.pdf. Czech Technical University in Prague, Charles University.

11. UNDERSTANDING THE 3 VS OF BIG DATA – VOLUME, VELOCITY AND VARIETY. *Coforge* [online]. 2017 [visited on 2022-02-02]. Available from: https://www.coforge.com/salesforce/blog/data-analytics/understanding-the-3-vs-of-big-data-volume-velocity-and-variety/.

12. *IBM Cloud Education* [online]. 2019 [visited on 2022-02-13]. Available from: https://www.ibm.com/cloud/learn/cap-theorem.

13. ALLEN, Matt. Relational Databases Are Not Designed For Scale. *MarkLogic* [online] [visited on 2022-06-22]. Available from: https://www.marklogic.com/blog/relational-databases-scale/.

14. SVOBODA, Martin. Basic Principles [online] [visited on 2022-02-06]. Available from: https://www.ksi.mff.cuni.cz/~svoboda/courses/201-MIE-PDB/lectures/MIEPDB16-Lecture-08-Principles.pdf. Charles University, Czech Technical University in Prague.

15. MEIER, Andreas; KAUFMANN, Michael. SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management. In: [online]. 2019, pp. 134, 139–140 [visited on 2022-02-17]. ISBN 978-3-658-24549-8. Available from: https://dokumen.pub/sql-amp-nosql-databases-models-languages-consistency-options-and-architectures-for-big-data-management-1nbsped-3658245484-978-3658245481.html.

16. SARAVANAN, Nandhini. The basics of NoSQL databases—and why we need them. *freeCodeCamp* [online]. 2019 [visited on 2022-02-18]. Available from: https://www.freecodecamp.org/news/nosql-databases-5f6639ed9574.

17. GATTERMAYER, Josef. NoSQL, Apache Cassandra, úvod a clusterování [online] [visited on 2022-04-14]. Available from: https://owncloud.cesnet.cz/index.php/s/JrIq5CP2OxyQIcA. Czech Technical University in Prague.

18. FOWLER, Adam. NoSQL For Dummies. In: [online]. 2015, p. 214 [visited on 2022-03-18]. ISBN 978-1-118-90578-4. Available from: https://www.pdfdrive.com/nosql-for-dummies-d52326329.html.

19. Database Sharding: Concepts and Examples. *MongoDB* [online] [visited on 2022-03-18]. Available from: https://www.mongodb.com/features/database-sharding-explained.

20. HANEL, Filip. *Comparison of SQL and noSQL application development*. 2020. Bachelor's thesis. Charles University.

21. What Is a NoSQL Database? *redis* [online] [visited on 2022-02-18]. Available from: https://redis.com/nosql/what-is-nosql.

22. What is NoSQL? *AWS* [online] [visited on 2022-02-19]. Available from: https://aws.amazon.com/nosql/.

23. Description of the database normalization basics. *Microsoft* [online]. 2022 [visited on 2022-02-19]. Available from: https://docs.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description.

24. BRUGGEN, Rik Van. Learning Neo4j: Run blazingly fast queries on complex graph datasets with the power of the Neo4j graph database. In: Birmingham: Packt Publishing Ltd., 2014, pp. 32–33. ISBN 978-1-84951-716-4.

25. DEKA, Ganesh Chandra; RAJ, Pethuru. Advances in COMPUTERS: A Deep Dive into NoSQL Databases: The Use Cases and Applications. In: [online]. Elsevier Science, 2018, chap. 2. Document-oriented databases, p. 159 [visited on 2022-02-04]. ISBN 978-0-12-813786-4. Available from: https://books.google.fr/books?id=LnFZDwAAQBAJ.

26. HARRISON, Guy. Next Generation Databases: NoSQLand Big Data. In: [online]. Apress, 2015, chap. 4: Document Databases, p. 53 [visited on 2022-02-04]. ISBN 978-1-4842-1329-2. Available from: https://books.google.fr/books?id=q6pPCwAAQBAJ.

27. What is a Document Database? *MongoDB* [online] [visited on 2022-02-20]. Available from: https://www.mongodb.com/document-databases.

28. Chapter 8. Data Types. *PostgreSQL 9.4.26 Documentation* [online] [visited on 2022-02-20]. Available from: https://www.postgresql.org/docs/9.4/datatype-json.html.

29. FURTADO, Preethica. Best Document Databases. *G2- Business Software Reviews* [online] [visited on 2022-02-20]. Available from: https://www.g2.com/categories/document-databases.

30. TAYLOR, David. What is MongoDB? Introduction, Architecture, Features & Example. *Guru99* [online]. 2022 [visited on 2022-03-15]. Available from: https://www.guru99.com/what-is-mongodb.html.

31. Introduction to MongoDB. *MongoDB* [online] [visited on 2022-03-15]. Available from: https://docs.mongodb.com/manual/introduction/.

32. Amazon DynamoDB System Properties. *DB-ENGINES* [online] [visited on 2022-03-16]. Available from: https://db-engines.com/en/system/Amazon+DynamoDB.

33. WICKRAMASINGHE, Shanika. MongoDB vs DynamoDB: Comparing NoSQL Databases. *bmc* [online]. 2021 [visited on 2022-03-16]. Available from: https://www.bmc.com/blogs/mongodb-vs-dynamodb.

34. System Properties Comparison Couchbase vs. MongoDB. *DB-ENGINES* [online] [visited on 2022-03-16]. Available from: https://db-engines.com/en/system/Couchbase%3BMongoDB.

35. The Essential Difference Between Couchbase & MongoDB. *Cubet Techno Labs* [online]. 2018 [visited on 2022-03-16]. Available from: https://db-engines.com/en/system/Couchbase%3BMongoDB.

36. OBJELEAN, Alexandru. Introduction to Couchbase - NoSQL Document Database. *TODAY SOFTWARE MAGAZINE* [online] [visited on 2022-03-16]. Available from: https://www.todaysoftmag.com/article/1506/introduction-to-couchbase-nosql-document-database.

37. What is Elasticsearch? *Elasticsearch* [online] [visited on 2022-03-17]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html.

38. PERRY, Yifat. Elasticsearch Architecture: 7 Key Components. *NetApp* [online]. 2018 [visited on 2022-03-18]. Available from: https://cloud.netapp.com/blog/cvo-blg-elasticsearch-architecture-7-key-components.

39. ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. Graph Databases: NEW OPPORTUNITIES FOR CONNECTED DATA. In: [online]. 2nd ed. Sebastopol: O'Reilly Media, Inc., 2015, pp. 11–12 [visited on 2022-03-21]. ISBN 978-1-491-93089-2. Available from: https://books.google.cz/books?id=wbsaAAAAQBAJ&printsec=frontcover&dq=nosql+graph+database&hl=en&sa=X&redir_esc=y#v=onepage&q=nosql%20graph%20database&f=false.

40. ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. Graph Databases: NEW OPPORTUNITIES FOR CONNECTED DATA. In: [online]. 2nd ed. Sebastopol: O'Reilly Media, Inc., 2015, pp. 20–21 [visited on 2022-03-21]. ISBN 978-1-491-93089-2. Available from: https://books.google.cz/books?id=wbsaAAAAQBAJ&printsec=frontcover&dq=nosql+graph+database&hl=en&sa=X&redir_esc=y#v=onepage&q=nosql%20graph%20database&f=false.

41. ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. Graph Databases: NEW OPPORTUNITIES FOR CONNECTED DATA. In: [online]. 2nd ed. Sebastopol: O'Reilly Media, Inc., 2015, pp. 15–16 [visited on 2022-03-21]. ISBN 978-1-491-93089-2. Available from: https://books.google.cz/books?id=wbsaAAAAQBAJ&printsec=frontcover&dq=nosql+graph+database&hl=en&sa=X&redir_esc=y#v=onepage&q=nosql%20graph%20database&f=false.

42. ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. Graph Databases: NEW OPPORTUNITIES FOR CONNECTED DATA. In: [online]. 2nd ed. Sebastopol: O'Reilly Media, Inc., 2015, p. 5 [visited on 2022-03-21]. ISBN 978-1-491-93089-2. Available from: https://books.google.cz/books?id=wbsaAAAAQBAJ&printsec=frontcover&dq=nosql+graph+database&hl=en&sa=X&redir_esc=y#v=onepage&q=nosql%20graph%20database&f=false.

43. The graph database defined. *AWS* [online] [visited on 2022-03-22]. Available from: https://aws.amazon.com/nosql/graph/.

44. Graph Database Defined. *Oracle* [online] [visited on 2022-03-23]. Available from: oracle.com/autonomous-database/what-is-graph-database/.

45. ING. MILAN DOJČINOVSKI, Ph.D. Semantic Web & Knowledge Graphs: Lecture 2: Resource Description Framework [online] [visited on 2022-03-23]. Available from: https://courses.fit.cvut.cz/NI-SWE/lectures/files/swe-lecture02.pdf. Czech Technical University in Prague.

46. ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. Graph Databases: NEW OPPORTUNITIES FOR CONNECTED DATA. In: [online]. 2nd ed. Sebastopol: O'Reilly Media, Inc., 2015, p. 4 [visited on 2022-03-21]. ISBN 978-1-491-93089-2. Available from: https://books.google.cz/books?id=wbsaAAAAQBAJ&printsec=frontcover&dq=nosql+graph+database&hl=en&sa=X&redir_esc=y#v=onepage&q=nosql%20graph%20database&f=false.

47. ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. Graph Databases: NEW OPPORTUNITIES FOR CONNECTED DATA. In: [online]. 2nd ed. Sebastopol: O'Reilly Media, Inc., 2015, pp. 26–27 [visited on 2022-03-21]. ISBN 978-1-491-93089-2. Available from: https://books.google.cz/books?id=wbsaAAAAQBAJ&printsec=frontcover&dq=nosql+graph+database&hl=en&sa=X&redir_esc=y#v=onepage&q=nosql%20graph%20database&f=false.

48. Graph databases explained. *Digital Guide IONOS* [online] [visited on 2022-03-23]. Available from: https://www.ionos.com/digitalguide/hosting/technical-matters/graph-database/.

49. STUECKE, Jan. Do Graph Databases Scale? *DZone* [online]. 2020 [visited on 2022-03-30]. Available from: https://dzone.com/articles/do-graph-databases-scale.

50. DB-Engines Ranking of Graph DBMS. *DB-ENGINES* [online] [visited on 2022-03-28]. Available from: https://db-engines.com/en/ranking/graph+dbms.

51. Neo4j - Overview. *tutorialspoint* [online] [visited on 2022-03-30]. Available from: https://www.tutorialspoint.com/neo4j/neo4j_overview.htm.

52. Microsoft Azure Cosmos DB System Properties. *DB-ENGINES* [online] [visited on 2022-03-31]. Available from: https://db-engines.com/en/system/Microsoft+Azure+Cosmos+DB.

53. ArangoDB System Properties. *DB-ENGINES* [online] [visited on 2022-03-31]. Available from: https://db-engines.com/en/system/ArangoDB.

54. AQL v3.9.0 Documentation. *ArangoDB* [online] [visited on 2022-03-31]. Available from: https://www.arangodb.com/docs/stable/aql/index.html.

55. GATTERMAYER, Ing. Josef. Cassandra Data Modeling, New, Legacy [online] [visited on 2022-04-01]. Available from: https://owncloud.cesnet.cz/index.php/s/2tq6jLSm3BzxzG5. Czech Technical University in Prague.

56. PRAMOD J. SADALAGE, Martin Fowler. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. In: Addison-Wesley Professional, 2012, pp. 32–33. ISBN 978-0321826626.

57. SVOBODA, Martin. Wide Column Stores: Cassandra [online]. 2020 [visited on 2022-04-12]. Available from: https://www.ksi.mff.cuni.cz/~svoboda/courses/201-MIE-PDB/lectures/MIEPDB16-Lecture-11-Cassandra.pdf. Charles University, Czech Technical University in Prague.

58. DB-Engines Ranking of Wide Column Stores. *DB-ENGINES* [online] [visited on 2022-04-15]. Available from: https://db-engines.com/en/ranking/wide+column+store.

59. System Properties Comparison Cassandra vs. HBase vs. Microsoft Azure Cosmos DB. *DB-ENGINES* [online] [visited on 2022-04-15]. Available from: https://db-engines.com/en/system/Cassandra%3BHBase%3BMicrosoft+Azure+Cosmos+DB.

60. SVOBODA, Martin. Key-Value Stores: RiakKV [online] [visited on 2022-04-17]. Available from: https://www.ksi.mff.cuni.cz/~svoboda/courses/201-MIE-PDB/lectures/MIEPDB16-Lecture-09-RiakKV.pdf. Charles University, Czech Technical University in Prague.

61. MEIER, Andreas; KAUFMANN, Michael. SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management. In: [online]. 2019, p. 203 [visited on 2022-04-21]. ISBN 978-3-658-24549-8. Available from: https://dokumen.pub/sql-amp-nosql-databases-models-languages-consistency-options-and-architectures-for-big-data-management-1nbsped-3658245484-978-3658245481.html.

62. DB-Engines Ranking of Key-value Stores. *DB-ENGINES* [online] [visited on 2022-04-17]. Available from: https://db-engines.com/en/ranking/key-value+store.

63. GAUTAM, Kartik. Elasticsearch- Introduction,Basic Concepts,Features & implementation. *Medium* [online]. 2019 [visited on 2022-04-23]. Available from: https://medium.com/@kartikgautam1710/elasticsearch-introduction-basic-concepts-features-implementation-2a28d656290.

64. TONG, Zachary. What is an Elasticsearch Index? *Elasticsearch* [online]. 2013 [visited on 2022-04-23]. Available from: https://www.elastic.co/blog/what-is-an-elasticsearch-index.

65. ANDERSEN, Bo. Sharding and scalability. In: *Udemy* [online] [visited on 2022-04-23]. Available from: https://www.udemy.com/course/elasticsearch-complete-guide/learn/lecture/16287978#overview. Complete Guide to Elasticsearch.

66. Scalability and resilience: clusters, nodes, and shards. *Elasticsearch* [online] [visited on 2022-04-23]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/scalability.html#scalability.

67. Create index API. *Elasticsearch* [online] [visited on 2022-04-23]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-create-index.html.

68. Index modules. *Elasticsearch* [online] [visited on 2022-04-24]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules.html#index-modules-settings.

69. ANDERSEN, Bo. Introduction to analysis. In: *Udemy* [online] [visited on 2022-04-25]. Available from: https://www.udemy.com/course/elasticsearch-complete-guide/learn/lecture/18848504#overview. Complete Guide to Elasticsearch.

70. Standard analyzer. *Elasticsearch* [online] [visited on 2022-04-26]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-standard-analyzer.html.

71. INGEBRIGTSEN, Morten. Indexing for Beginners, Part 3. *Elasticsearch* [online] [visited on 2022-04-27]. Available from: https://www.elastic.co/blog/found-indexing-for-beginners-part3.

72. ANDERSEN, Bo. Full text queries vs term level queries. In: *Udemy* [online] [visited on 2022-04-27]. Available from: https://www.udemy.com/course/elasticsearch-complete-guide/learn/lecture/7585378#overview. Complete Guide to Elasticsearch.

73. *Your window into the Elastic Stack* [online] [visited on 2022-04-27]. Available from: https://static-www.elastic.co/v3/assets/bltefdd0b53724fa2ce/blta64f0cb4ab8c7c1c/5fa31cc94a4abb73ff79b0d5/illustrated-screenshot-hero-kibana.png.

74. SVOBODA, Martin. Graph Databases: Neo4j [online] [visited on 2022-05-03]. Available from: https://www.ksi.mff.cuni.cz/~svoboda/courses/201-MIE-PDB/lectures/MIEPDB16-Lecture-12-Neo4j.pdf. Charles University, Czech Technical University in Prague.

75. SAYERS, Louis. Neo4j Editions. In: *Udemy* [online] [visited on 2022-05-03]. Available from: https://www.udemy.com/course/neo4j-foundations/learn/lecture/18243882#content. Neo4j: GraphDB Foundations with Cypher.

76. BRUGGEN, Rik Van. Learning Neo4j: Run blazingly fast queries on complex graph datasets with the power of the Neo4j graph database. In: Birmingham: Packt Publishing Ltd., 2014, p. 44. ISBN 978-1-84951-716-4.

77. BRUGGEN, Rik Van. Learning Neo4j: Run blazingly fast queries on complex graph datasets with the power of the Neo4j graph database. In: Birmingham: Packt Publishing Ltd., 2014, pp. 46–47. ISBN 978-1-84951-716-4.

78. Cypher Query Language. *Neo4j* [online] [visited on 2022-05-06]. Available from: https://neo4j.com/developer/cypher/.

79. ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. Graph Databases: NEW OPPORTUNITIES FOR CONNECTED DATA. In: [online]. 2nd ed. Sebastopol: O'Reilly Media, Inc., 2015, p. 162 [visited on 2022-03-21]. ISBN 978-1-491-93089-2. Available from: https://books.google.cz/books?id=wbsaAAAAQBAJ&printsec=frontcover&dq=nosql+graph+database&hl=en&sa=X&redir_esc=y#v=onepage&q=nosql%20graph%20database&f=false.

80. CHAO, Joy. Graph Databases for Beginners: Native vs. Non-Native Graph Technology. *Neo4j* [online] [visited on 2022-05-04]. Available from: https://neo4j.com/blog/native-vs-non-native-graph-technology/.

81. ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. Graph Databases: NEW OPPORTUNITIES FOR CONNECTED DATA. In: [online]. 2nd ed. Sebastopol: O'Reilly Media, Inc., 2015, p. 155 [visited on 2022-03-21]. ISBN 978-1-491-93089-2. Available from: https://books.google.cz/books?id=wbsaAAAAQBAJ&printsec=frontcover&dq=nosql+graph+database&hl=en&sa=X&redir_esc=y#v=onepage&q=nosql%20graph%20database&f=false.

82. ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. Graph Databases: NEW OPPORTUNITIES FOR CONNECTED DATA. In: [online]. 2nd ed. Sebastopol: O'Reilly Media, Inc., 2015, pp. 164–165 [visited on 2022-03-21]. ISBN 978-1-491-93089-2. Available from: https://books.google.cz/books?id=wbsaAAAAQBAJ&printsec=frontcover&dq=nosql+graph+database&hl=en&sa=X&redir_esc=y#v=onepage&q=nosql%20graph%20database&f=false.

83. The Fastest Path to Graph Scaling and Flexible Development. *Neo4j* [online] [visited on 2022-05-05]. Available from: https://neo4j.com/product/neo4j-graph-database/scalability/.

84. [online] [visited on 2022-06-22]. Available from: https://www.flaticon.com/premium-icon/coding_2010957?term=backend&page=1&position=6&page=1&position=6&related_id=2010957&origin=tag.

85. [online] [visited on 2022-06-22]. Available from: https://www.flaticon.com/free-icon/server_689319?term=database&page=1&position=8&page=1&position=8&related_id=689319&origin=search.

86. *@ngrx/store* [online] [visited on 2022-06-18]. Available from: https://ngrx.io/guide/store.