Graph Databases vs. Relational Databases for Social Web Applications

A Systematic Evaluation

MASTER THESIS

CHRISTIAN FRIES

August 2023

Thesis supervisors:

Prof. Dr. Jacques PASQUIER–ROCHA
Software Engineering Group

UNI
FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Software Engineering Group
Department of Informatics
University of Fribourg
(Switzerland)

Software
Engineering Group

"The difference between theory and practice is that in theory, there is no difference between theory and practice."

- *Richard Moore*

Abstract

Since the advent of Web 2.0, the internet is no longer just a place to consume information. It has become a place to actively participate, to interact with other people. Large social media platforms have emerged with up to 2.3 billion users. The spread of such platforms has changed the requirements for the technologies. It has been realised that it is not only the discrete information that is valuable, but that the relationships between the information are even more valuable. From this realisation, the graph databases as we know them today have gained popularity.

In this thesis, a systematic evaluation of a graph database and a relational database is conducted. Both databases are evaluated based on well-known use cases in the context of social web applications. The two databases are opposed and compared based on a defined set of criteria.

Keywords: Graph Database, Relational Database, MySQL, Neo4j, SQL, Cypher

Preamble

Acknowledgements

First, I would like to express my deepest appreciation to Prof. Dr. Jacques Pasquier-Rocha of the Software Engineering Group at the University of Fribourg. His invaluable expertise and feedback have successfully guided me through this thesis. The interesting discussions have always inspired me to include additional perspectives.

Furthermore, I would like to extend my sincere thanks to my employer, LST AG, for supporting me in writing this thesis by providing me with the necessary time and flexibility.

Last but not least, I'd like to thank my family and friends. Their believe in me has kept my spirits and motivation high throughout this process.

Notations and Conventions

- Figures, tables, and listings
  - Figures, tables, and listings are numbered on a per chapter basis. As an example, the second figure in chapter 3 will be noted as *Figure 3.2*.
- Formatting
  - To put emphasis on one or several words, they're formatted *like this*.
  - URLs are formatted `https://www.like.this`
  - Inline code is formatted `likeThis();`
  - Code blocks are formatted as follows:

```
1    class Thesis {
2        public function __construct() {}
3    }
```

Table of Contents

1 Introduction **1**
1.1 Motivation and Goals ... 1
1.2 Use Cases ... 2
1.2.1 List of Friends ... 2
1.2.2 Friends Recommendation Based on Mutual Friends ... 2
1.2.3 Friends Recommendation Based on Several Criteria ... 2
1.2.4 Auto-Suggest for Search Field ... 3
1.3 Organization ... 3
2 Databases **4**
2.1 Types of Databases ... 4
2.2 Selection Criteria ... 5
2.3 MySQL – Relational Database ... 5
2.3.1 Fundamentals ... 6
2.3.2 Structured Query Language ... 6
2.4 Neo4j – Graph Database ... 7
2.4.1 Fundamentals ... 7
2.4.2 Cypher ... 7
3 Database Implementation **9**
3.1 Creating a Test Dataset ... 9
3.2 Setup of Database Servers ... 11
3.2.1 MySQL Server Setup ... 11
3.2.2 Neo4j Server Setup ... 15
3.3 Import Test Data ... 16
3.3.1 Import Data into MySQL Database ... 16
3.3.2 Import Data into Neo4j Database ... 17
3.4 Optimizing Schema and Data ... 19
3.5 Queries for the Use Cases ... 21
3.6 Validating the Queries ... 27
3.7 Automation ... 28
4 Evaluation **29**
4.1 Evaluation Criteria ... 29
4.1.1 Performance ... 29

4.1.2 Number of Clauses 30
4.1.3 Developer Convenience 30
4.2 Evaluation 30
4.3 Result 31
4.3.1 Result of Performance Evaluation 31
4.3.2 Result of Number of Clauses Evaluation 33
4.3.3 Result of Developer Convenience Evaluation 33
4.4 Discussion 33
5 Prototype **35**
5.1 Architecture 35
5.2 Technology Stack 36
5.3 Implementation 36
5.3.1 REST API 36
5.3.2 Frontend Application 38
5.4 User Interface 38
6 Conclusion and Future Work **42**
6.1 Conclusion 42
6.2 Future Work 42
A Source Code **44**
References **45**
Referenced Web Resources **46**

List of Figures

Figure 2.1: *MySQL tables for relationships between actors and movies* 6
Figure 2.2: *Cyphers ASCII-art type of syntax [3]* 8
Figure 3.1: *MySQL Workbench listing person records* 17
Figure 3.2: *Neo4j Desktop presenting a graph with nodes and relationships* 19
Figure 3.3: *Dataset for query validation* 28
Figure 4.1: *Query performance comparison for list of friends* 31
Figure 4.2: *Query performance comparison for friends recommendation #1* 31
Figure 4.3: *Query performance comparison for friends recommendation #2* 32
Figure 4.4: *Query performance comparison for auto-suggest* 32
Figure 4.5: *Result of number of clauses evaluation* 33
Figure 5.1: *The architecture of the prototype* 35
Figure 5.2: *The left side of the toolbar* 38
Figure 5.3: *The right side of the toolbar* 38
Figure 5.4: *The view of use case 3* 39
Figure 5.5: *The view of a person* 40
Figure 5.6: *The view of a topic* 41

List of Tables

Table 4.1: *Overview of users selected for evaluation* 30
Table 4.2: *Clauses that were counted during the evaluation* 31
Table 4.3: *Result of developer convenience evaluation* 33
Table 5.1: *Overview of REST API endpoints* 37

List of Source Code

Code 3.1: *Configuration options for the script generating the dataset* 10
Code 3.2: *Create a user and define permissions* 12
Code 3.3: *Create a database* 12
Code 3.4: *Create a table for persons* 12
Code 3.5: *Create a table for topics* 13
Code 3.6: *Create a table for friendships* 13
Code 3.7: *Create a table for likes* 13
Code 3.8: *Create indexes for columns used for lookup* 14
Code 3.9: *Create foreign keys* 14
Code 3.10: *Create indexes and constraints* 15
Code 3.11: *Query to import CSV data into a MySQL table* 16
Code 3.12: *Query to import CSV data into a Neo4j database as nodes* 18
Code 3.13: *Query to import CSV data into a Neo4j database as relationships* 18
Code 3.14: *SQL queries to normalize the column country* 20
Code 3.15: *SQL query to insert bidirectional friendship records* 20
Code 3.16: *SQL query to insert bidirectional friendship records* 21
Code 3.17: *SQL query to optimize the table storage* 21
Code 3.18: *SQL query for use case 1* 21
Code 3.19: *Cypher query for use case 1* 21
Code 3.20: *SQL query for use case 2* 22
Code 3.21: *Cypher query for use case 2* 23
Code 3.22: *SQL query for use case 3* 24
Code 3.23: *Cypher query for use case 3* 25
Code 3.24: *SQL query for use case 4* 26
Code 3.25: *Cypher query for use case 4* 27
Code 5.1: *Controller method for use case 1* 37
Code 5.2: *HTTP interceptor to keep track of performance* 38

1
Introduction

1.1 Motivation and Goals 1
1.2 Use Cases 2
1.2.1 List of Friends 2
1.2.2 Friends Recommendation Based on Mutual Friends 2
1.2.3 Friends Recommendation Based on Several Criteria 2
1.2.4 Auto-Suggest for Search Field 3
1.3 Organization 3

1.1 Motivation and Goals

Since the advent of Web 2.0, the internet is no longer just a place to consume information. It has become a place to actively participate, to interact with other people, to share information and to collaborate. Platforms such as Facebook have more than 2 billion users [Est23].

Not only have large social media platforms emerged from this, but various new standards have also developed. For example, a company's static intranet has become an internal communication platform, the fan platform enables direct interaction with the stars, and the residents of a housing estate can organise themselves via a platform to support each other when sugar is missing or a caregiver needs to be found.

With these new habits, the demands on technologies have changed. It has been realised that it is not only the discrete information that is valuable, but that the relationships between the information are even more valuable. From this realisation, the graph databases as we know them today have become popular [Ian13].

The goal of this thesis is to evaluate a graph database and a relational database based on well-known use cases in the context of social web applications. The two databases are contrasted and compared based on a defined set of criteria. A prototype is to be developed that demonstrates the use cases in action.

1.2 Use Cases

When it comes to social web applications, they all have common use cases they need to master. Users of such applications want to manage their profiles, connect with other users, share information about themselves and subscribe to information shared by others.

For this thesis, four common use cases were identified and selected as basis for the evaluation of the database systems and for the prototype.

1.2.1 List of Friends

The simplest use case is about providing a list of friends. As a user, I want to see a list of people I am friends with. Several platforms implement different mechanisms for this use case, sometimes a user can follow another user without the other user's approval, sometimes the other user must approve the request. Also, the platforms differentiate between unidirectional and bidirectional connection. On e.g. Twitter, a user A can follow a user B, but this does not automatically result in user B following user A as well. This is called a unidirectional connection. On Facebook on the other hand, friendships are modelled as bidirectional connection. If user A sends a friendship request to user B and user B approves the request, both users are friends of each other. This is a bidirectional connection.

For this thesis, the concept of a bidirectional connection is used.

1.2.2 Friends Recommendation Based on Mutual Friends

A common use case for social web applications is the recommendation of friends or people one might know. Getting in touch with people is the number one reason to use social media apps. The more people a user is linked to, the more time he spends on the platform. For the operator, this makes it possible to display more advertising, which is often an important source of revenue. In this sense, the recommendation of friends is an important tool.

This use case assumes that people, that have mutual friends, are likely to know each other as well.

1.2.3 Friends Recommendation Based on Several Criteria

The goal of this use case is the same as in the previous use case, recommending possible friends. However, this use case is based on a different assumption: Not only people who have mutual friends should be recommended, but also people who have common interests or live in the same country.

In order to be able to better tune the results, the individual criteria need to be rated. The fact that someone has common interest should be weighted higher than the fact that someone lives in the same country.

1.2.4 Auto-Suggest for Search Field

A search field is a very common concept to allow users of a platform to navigate and find content. If a few letters are entered into the search field, possible results are suggested automatically while typing. The results contain both persons and topics that the user is already connected to, either as friend or as like, and persons and topics the user is not yet connected to. The results must be sorted in an order that takes these factors into account.

1.3 Organization

This thesis is structured in six chapters. The first chapter presents the goals and the motivation for this thesis and explains the relevant use cases. The second chapter provides an overview of types of databases and highlights two database management systems in detail. In chapter 3, the setup of the DBMSs and the development of the queries is explained. The evaluation process, the results and the discussion are presented in chapter 4. Chapter 5 introduces the prototype implemented as part of this thesis and explains the architecture and the technology stack used. Finally, chapter six provides a conclusion and future work.

2 Databases

2.1 Types of Databases 4

2.2 Selection Criteria 5

2.3 MySQL – Relational Database 5

2.3.1 Fundamentals 6

2.3.2 Structured Query Language 6

2.4 Neo4j – Graph Database 7

2.4.1 Fundamentals 7

2.4.2 Cypher 7

A database is an organized collection of information which can be electronically searched and sorted according to its various categories [Ken89]. To interact with a database, a database management system is used.

2.1 Types of Databases

Since the invention of the first database in the mid-1960s, countless database management systems have been developed. One of the first types of databases was the relational database. It introduced data records and relationships between those in a space-efficient way. IBM in the 1970s released System R, a relational database which was the first to use the Structured Query Language (SQL) [Don81].

In the 1980s, object-oriented database management systems emerged. They support the modelling and creation of data as objects and integrate seamlessly with object-oriented programming languages.

In 1998, the term NoSQL was first used for a relational database that did not use SQL but another query language. NoSQL databases are more flexible than traditional databases because they don't follow a rigid schema but instead have more flexible structures to accommodate their datatypes.

Over time, four major types of NoSQL databases emerged:

- *Document databases* store data in documents similar to JSON objects. Each document contains pairs of fields and values.

- *Key-value databases* are a simpler type of database where each item contains keys and values.
- *Column-family databases* or *wide-column databases* store data in tables, rows, and dynamic columns.
- Graph databases store data in nodes and edges. Nodes typically store objects or things, while edges store information about the relationships between the nodes.

In recent years, graph databases have become highly popular, mainly thanks to the fact that relationships are first-class citizens. Traversing relationships is very fast because relationships are not calculated at query times but are persisted in the database. Graph databases have shown to be good solutions for use cases such as recommendation engines, activity analytics, and social networking.

In this thesis, the focus is on graph databases and relational databases. For each of the two database types, a database system is selected and the two are compared based on the use cases presented in section 1.2.

2.2 Selection Criteria

In order to be able to make a fair comparison, some criteria were defined that are decisive for the choice of the database systems.

The first criterion relates to the relevance of the database systems. The systems should have been available for several years and have proven themselves in productive use in well-known products. A continuous development of the systems should be guaranteed.

The second criterion refers to the popularity. Database systems should be used which enjoy great popularity and which have an active community. The DB-Engines Knowledge Base of Rational and NoSQL Database Management Systems [1] provides a ranking of database systems based on the number of mentions in search results, Google Trends, and the number of related questions on platforms such as Stack Overflow and DBA Stack Exchange.

The third criterion refers to the availability. The systems should be available as open-source software. This helps ensuring reliability and transparency.

The fourth criterion refers to the availability of resources. Systems should be used which are well documented and offer a vibrant support community.

Based on these four criteria, the decision was made in favour of MySQL as a relational database and Neo4j as a graph database. Both have proven themselves for several years in platforms such as Facebook or Adobe Behance and enjoy great popularity.

2.3 MySQL – Relational Database

MySQL is an open-source relational database management system first released in 1995 under the GNU General Public License. It is written in C and C++ and works on a wide range of operating systems.

According to DB-Engines, MySQL is the most popular open-source database management system. Platforms such as Facebook or Twitter were built based on MySQL and still make heavy use of it.

2.3.1 Fundamentals

MySQL organizes data in databases and tables. Each table requires a schema defining the columns available to that table and the datatypes to be used for the columns. MySQL knows datatypes such as text, numbers, booleans, dates and more.

Adding a new row to a table expects values for all columns to be defined, otherwise a default value is used if defined in the schema. Adding values for columns that are not defined in the schema is not possible.

Each row in a table can be uniquely identified by the primary key. The key can be an automatically incrementing number or a unique value provided when adding the row.

In MySQL, relationships between records are modelled based on the concept of foreign keys. It allows to create cross-references between tables by defining which column in the original table refers to which column in the target table. MySQL will make sure, that the values inserted into the column on the original table exist in the referenced column on the referenced table. This is called referential integrity.

If multiple rows in one table can refer to multiple rows in another table, an intermediate table needs to be introduced. For example, several actors act in a movie and an actor acts in several movies. These relationships can't be stored in the table *actor* or *movie*, an intermediate table holding these relationships is needed.

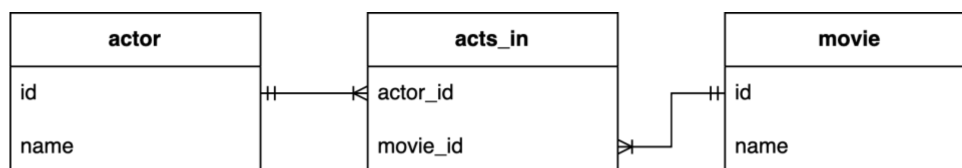

Figure 2.1: ***MySQL tables for relationships between actors and movies***

Indexes can be used to speed up the process of finding rows with specific values. Without an index, MySQL must read through the entire table to find the relevant rows, which gets slower the larger the table gets. With the help of an index, the position of the records in the data file can quickly be determined.

2.3.2 Structured Query Language

The structured query language SQL is a language to operate databases such as MySQL. It has been introduced in the 1970s and became an ANSI standard in 1986. It's used by most relational database management systems but a lot of them don't fully adhere to the standard.

SQL uses the relational model described by Edgar F. Codd. According to that model, all data is represented in terms of tuples and grouped into relations [Edg70].

SQL queries use a set of clauses to interact with the database. A `SELECT` clause can be used to select columns whose values should be returned. The `FROM` clause defines, what table to read from. The `WHERE` clause can be used to define conditions that need to be fulfilled. Ordering can be achieved using the `ORDER BY` clause and limiting can be achieved with `LIMIT`.

A central aspect to SQL is the `JOIN` clause. It is used to combine rows from two or more tables based on a common field between them. MySQL requires joins to traverse relationships between entities. Several different types of joins exist such as `INNER JOIN` which allows conditions that specify how tables are joined, or `CROSS JOIN` which generates a cartesian product. These joins are executed during runtime.

For more complex queries there are further possibilities, e.g. to group and aggregate rows.

2.4 Neo4j – Graph Database

Neo4j is an open-source graph database management system first released in 2010 under the GPL v3 license. It is written in Java and works on a great number of operating systems. According to DB-Engines, Neo4j is the most popular graph database management system. It is used by platforms such as Adobe Behance or eBay.

2.4.1 Fundamentals

Neo4j uses a property graph database model, where a node represents an entity, an edge represents a relationship between entities, and a property represents a specific feature of an entity or relationship [Ren18].

Nodes can have zero or more labels to classify what kind of nodes they are. For example, a node representing a person could be labelled with the label `Person`. With that in place, Neo4j can perform operations only on person nodes.

Relationships connect a source node and a target node. They always have a direction, and they must have one single type to classify what type of relation it is.

To further describe a node or a relationship, properties can be added in form of key-value pairs. Values can hold different data types, such as numbers, strings, booleans or homogeneous lists.

Neo4j is schema optional, meaning that it is not necessary to create a schema up front [2]. Nodes and relationships can be created right away. Indexes and constrains can be introduced at any point in time to gain performance and modelling benefits.

2.4.2 Cypher

Cypher is the graph query language that allows storing, retrieving, and manipulating data stored in Neo4j. Like Neo4j itself, Cypher is open source and backed by a number of companies.

Cyphers visual way of matching patterns and relationships makes it easy to learn and intuitive to use. The ASCII-art type of syntax uses rounded brackets to refer to nodes and square brackets to refer to relationships. Property filtering can be achieved using curly brackets. Writing queries is like drawing a graph pattern as shown in figure 2.2.

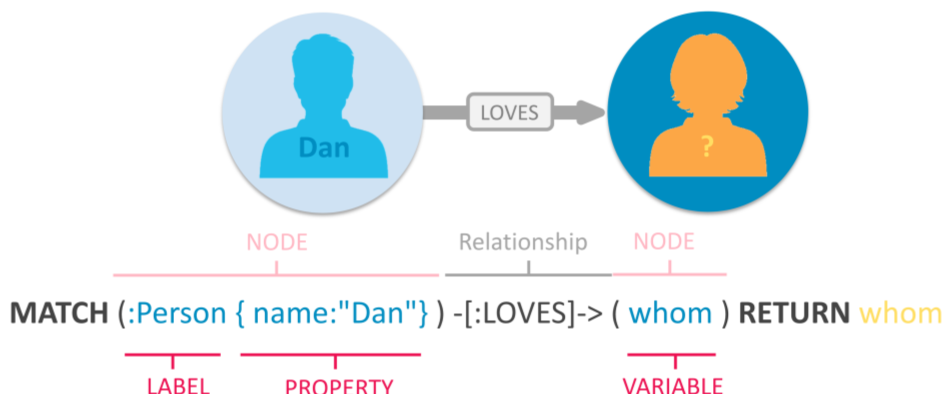

Figure 2.2: ***Cyphers ASCII-art type of syntax [3]***

Cypher was inspired by SQL and works with a similar set of clauses. The `MATCH` clause can be used to select data and the `RETURN` clause defines what data to return from the result set. Conditions can be applied using the `WHERE` clause, ordering can be achieved using the `ORDER BY` clause and limiting can be achieved using `LIMIT`. For persons that already have experience with other database query languages such as the standardized SQL, getting started with Cypher feels familiar.

The power of Cypher comes from the way it handles traversing of relationships. It's almost like using natural language. As an example, finding persons that like the same topics as the person with ID 1 can be achieved with

```
MATCH (:Person {id: 1})-[:LIKES]->(topic)<-[:LIKES]-(persons) RETURN persons
```

The mix of the visual representation and the language used makes it quite obvious what the query does. Traversing a relationship from the node specified before the relationship to the node specified afterwards requires the arrow to point from left to right: `()-[]->()`. To reverse the traversing direction, only the arrow has to be adjusted to point from right to left: `()<-[]-()`.

Similar to SQL, there are other clauses that can be used for more complex queries, to e.g. group and aggregate nodes.

3

Database Implementation

3.1 Creating a Test Dataset **9**
3.2 Setup of Database Servers **11**
3.2.1 MySQL Server Setup 11
3.2.2 Neo4j Server Setup 15
3.3 Import Test Data **16**
3.3.1 Import Data into MySQL Database 16
3.3.2 Import Data into Neo4j Database 17
3.4 Optimizing Schema and Data **19**
3.5 Queries for the Use Cases **21**
3.6 Validating the Queries **27**
3.7 Automation **28**

3.1 Creating a Test Dataset

Comparing two database systems in the context of social web applications requires a dataset, that includes all relevant data for the use cases.

There are several platforms such as Kaggle [4], that offer various types of datasets for free use. The difficulty is to find a dataset that perfectly fits the application purpose. Since no suitable dataset could be found, a custom dataset was developed.

The most central aspect of social web applications are the people that use them. Therefore, the first entity in the dataset is *Person*. A person has a first name, a last name, a gender, and a country in which they live.

The second entity in the dataset is *Friendship*. Each person can be friends with any number of people. In the context of this thesis, the relationship is defined to be bi-directional. If person A is friends with person B, person B is friends with person A, too.

On social web platforms, people can not only connect with each other, they can also connect with stars, clubs, companies, schools and much more. In this work, these entities are aggregated into a single entity called *Topic* where each topic has a name. The relationship between a person and a topic is called *Like*.

These four entities represent the data model to be used for the evaluation.

With the data model specified, test data must be created. To ensure the evaluation delivers a meaningful result, several constraints are placed on the dataset:

- 60'000 people from three different countries
- 100 topics
- Approximately 1'000'000 friendships
- Approximately 200'000 likes

The dataset is generated based on a PHP script developed as part of this thesis. As an input, the script takes a set of configuration options. As an output, it generates CSV files containing the dataset.

```
1     protected array $gender = ['female', 'male'];
2     protected array $countries = ['England', 'France', 'Germany'];
3     protected int $numberOfPersonsPerCountry = 20000;
4     protected int $numberOfRegionalConnectionsMin = 10;
5     protected int $numberOfRegionalConnectionsMax = 40;
6     protected int $numberOfNationalConnectionsMin = 5;
7     protected int $numberOfNationalConnectionsMax = 30;
8     protected int $numberOfInternationalConnectionsMin = 1;
9     protected int $numberOfInternationalConnectionsMax = 10;
10    protected int $numberOfLikesMin = 2;
11    protected int $numberOfLikesMax = 10;
12    protected float $probabilityForLikes = 0.6;
13    protected float $probabilityForRegionalConnections = 0.6;
14    protected float $probabilityForNationalConnections = 0.6;
15    protected float $probabilityForInternationalConnections = 0.05;
```

Code 3.1: ***Configuration options for the script generating the dataset***

Listing 3.1 presents the possible configuration options. Each person is assigned a gender out of `$gender` and a country out of `$country`. For each country, `$numberOfPersonsPerCountry` persons are generated.

Friendships are generated based on probabilities and they're classified into four classes:

- Local: 10 consecutive persons
- Regional: 100 consecutive persons
- National: Persons living in the same country
- International: Persons living in another country

Each class has an upper and a lower limit.

As topics serves a selection of companies from a list of the top 100 most valuable brands in 2022 [5] mixed with local companies and brands.

Likes are generated based on probabilities. With a probability of `$probabilityForLikes`, each person is assigned between `$numberOfLikesMin` and `$numberOfLikesMax` likes.

3.2 Setup of Database Servers

Running tests and comparing the two database systems requires a server instance of both database management systems, MySQL and a Neo4j. Each server needs to have the same set of resources assigned in order to guarantee a fair comparison. This can best be achieved by using virtualization.

As hosting platform serves an entry-level server running Proxmox Virtual Environment [6], an open-source server management platform for enterprise virtualization. Its virtualization technology is based the well-known KVM hypervisor.

The virtual machines for both servers get the same set of resources assigned:

- 4 CPU cores
- 8 GB RAM
- 32 GB SSD storage

This exceeds what MySQL and Neo4j require as minimum. They both require at least 2 CPU cores and 2 GB RAM [7][8].

As a starting point, both machines get a clean install of Ubuntu Server 20.04 with enabled firewall and SSH access. The individual setup for each server is explained in the next two sections.

3.2.1 MySQL Server Setup

MySQL can be downloaded from the official website [9]. It is also available in most package repositories. On Ubuntu Server, it can be installed using the APT package management system with the command `sudo apt install mysql-server`. As soon as the command is finished, the server service has to be started with `sudo systemctl start mysql.service`. Afterwards, the MySQL server is up and running.

Create Database User

By default, the MySQL server comes with a predefined user named *root*. For security reasons, this user should not be used for working with the database, only for management tasks. Therefore, a new database user needs to be created with a restricted set of permissions.

This can be achieved by logging into the database server using the CLI command `mysql -uroot -p`. It asks for the root password which is *root* by default. After the successful login, the server is ready to receive SQL queries.

The code in listing 3.2 shows how to create a new user. The name of the new user is set to *thesis* and the password is set to *MasterThesis23!*.

```
1     CREATE USER 'thesis'@'%' IDENTIFIED BY 'MasterThesis23!';
2     GRANT CREATE, ALTER, DROP, INSERT, UPDATE, INDEX, DELETE, SELECT,
      REFERENCES, RELOAD, FILE on *.* TO 'thesis'@'%' WITH GRANT
      OPTION;
3     FLUSH PRIVILEGES;
```

Code 3.2: ***Create a user and define permissions***

For all future queries and sessions, the user *thesis* should be used instead of the root user. The current session must be closed with `exit;` and a new session must be started, this time with the newly created user: `mysql -uthesis -p`

Create a Database

As a first step, a database is required. It can be created with the `CREATE DATABASE` command:

```
1     CREATE DATABASE thesis;
```

Code 3.3: ***Create a database***

After creating a new database, it can be selected with `USE thesis;` This will make sure all future queries will be executed in that database.

Create Database Tables

After the creation of the database, the tables can be created using the command `CREATE TABLE`. First, the table for persons is created:

```
1     CREATE TABLE person (
2       id int NOT NULL AUTO_INCREMENT,
3       gender varchar(255) NOT NULL,
4       first_name varchar(255) NOT NULL,
5       last_name varchar(255) NOT NULL,
6       language varchar(255) NOT NULL,
7       country varchar(255) NOT NULL,
8       PRIMARY KEY (id)
9     ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
      COLLATE=utf8mb4_0900_ai_ci;
```

Code 3.4: ***Create a table for persons***

For every column of the table, the name and the type must be specified. On line 3, a column called *id* is defined to be of type *integer*. It is not allowed to be *null*, meaning it must always have a value. Using the option `AUTO_INCREMENT` tells the database to always use the next unused integer number if no value for id is provided.

The statement on line 8 defines the column *id* to be the *primary key*. This is a constraint that requires the mentioned column to uniquely identify a person record. The value of this column needs to be unique and is not allowed to be *null*.

The table for topics can be created using a similar query:

```
1     CREATE TABLE topic (
2       id int NOT NULL AUTO_INCREMENT,
3       name varchar(255) NOT NULL,
4       PRIMARY KEY (id)
5     ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
      COLLATE=utf8mb4_0900_ai_ci;
```

Code 3.5: ***Create a table for topics***

The relationship between persons is stored in a table called friendship. It has a person as an origin of the friendship and a person as a target of the friendship. Such a relation is called a self-referencing relationship.

```
1     CREATE TABLE friendship (
2       origin_id int NOT NULL,
3       target_id int NOT NULL,
4       created_at datetime NOT NULL,
5       PRIMARY KEY (origin_id,target_id)
6     ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
      COLLATE=utf8mb4_0900_ai_ci;
```

Code 3.6: ***Create a table for friendships***

While the query is quite similar to the two queries before, it's important to notice the primary key definition on line 5. It is not only referring to one column but it is referring to the two columns *origin_id* and *target_id*. This means, a friendship is uniquely identified by its origin and its target. The combination of these two values must be unique.

The same concept applies to the table for likes:

```
1     CREATE TABLE `like` (
2       person_id int NOT NULL,
3       topic_id int NOT NULL,
4       PRIMARY KEY (person_id,topic_id)
5     ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
      COLLATE=utf8mb4_0900_ai_ci;
```

Code 3.7: ***Create a table for likes***

One important aspect to note for the table *like* is the fact that *like* is a reserved keyword in MySQL. It can be used to compare strings for partial match. A table can still be named *like*, but all usages of the table name must be encoded with backticks. The same rule applies to all reserved keywords.

Create Indexes and Foreign Keys

With the schema created so far, the database would be ready to be used. One important aspect is missing, though: Indexes and foreign keys.

```
1     ALTER TABLE friendship
2         ADD INDEX person_origin_idx (origin_id),
3         ADD INDEX person_target_idx (target_id);
4     ALTER TABLE `like`
5         ADD INDEX idx_person (person_id),
6         ADD INDEX idx_topic (topic_id);
7
```

Code 3.8: ***Create indexes for columns used for lookup***

Indexes can be used to speed up the process of finding rows with specific values. Without an index, MySQL must read through the entire table to find the relevant rows, which gets slower the larger the table gets. With the help of an index, the position of the records in the data file can quickly be determined. Columns defined as primary key are always indexed. As a rule of thumb, all columns used for filtering, joining or grouping should be indexed.

```
1     ALTER TABLE friendship
2         ADD CONSTRAINT fk_person_id_origin
          FOREIGN KEY (origin_id) REFERENCES person (id);
3     ALTER TABLE friendship
4         ADD CONSTRAINT fk_person_id_target
          FOREIGN KEY (target_id) REFERENCES person (id);
```

Code 3.9: ***Create foreign keys***

The concept of foreign keys allows to create cross-references between tables. In listing 3.9, two foreign keys, *fk_person_id_origin* and *fk_person_id_target,* are defined for the columns *origin_id* and *target_id* of the table *friendship*. Both foreign keys reference the column *id* on the table *person*. MySQL will make sure, that for every value that is inserted into one of these columns, a row with the same value exists in the person table. As a result, it is not possible to create a friendship record where the origin or target person does not exist.

Allow Remote Access

By default, a MySQL server can only be accessed locally, because it binds to the IP address `127.0.0.1`. To allow remote connections, the configuration of the server needs to be adjusted to bind to the IP address `0.0.0.0`. Additionally, the firewall needs to be configured to allow TCP traffic on port 3306.

Optimize Memory Usage

Tuning the memory usage of a database server is an art on its own and needs a lot of experience in database server management and profound knowledge about the number and type of access. The settings for a huge number of concurrent queries that only need little memory are completely different from the settings for a low number of queries that require a lot of processing power.

The primary setting that can be optimized on a MySQL server is the setting `innodb_buffer_pool_size`. It defines the amount of RAM that can be used to cache data. By default, it is set to 128 MB. For a server only running a MySQL server, it is recommended to raise the setting to around 80% of the available memory. In the present case, the setting was adjusted to 6 GB.

3.2.2 Neo4j Server Setup

Neo4j can be downloaded from the download center available on the official website [10]. Like MySQL, it can be installed using most package managers, although it is not available in the official repositories. The repository provided by Neo4j first needs to be added to the list of sources as explained in the official install guide [11]. Once the repository is added, Neo4j can be installed using the command `sudo apt-get install neo4j=1:5.4.0`. As soon as the command is finished, the database server service needs to enabled with `sudo systemctl enable neo4j.service` and started with `sudo systemctl start neo4j.service`.

Change User Credentials

By default, the Neo4j server comes with a user called *neo4j* and the password *neo4j*. For security reasons, the user credentials must be changed. This can be achieved by using the CLI tool `cypher-shell`. When first running the cypher shell, it will ask for the username and the password. Afterwards, it will automatically request a new password. In the examples throughout this thesis, the password *MasterThesis23!* is used.

Neo4j also comes with a predefined database called neo4j.

Create Database Schema

As opposed to relational databases, the graph database Neo4j is *schema optional*, meaning that it is not necessary to have a schema defined up front. Nodes and relationships with new labels and properties can be created ad-hoc. To speed up reading and writing queries and to guarantee uniqueness for some properties, it is a good practice to generate indexes and constraints before importing data, though.

```
1     CREATE CONSTRAINT UniquePerson
2         FOR (p:Person) REQUIRE p.personId IS UNIQUE;
3     CREATE CONSTRAINT UniqueTopic
4         FOR (t:Topic) REQUIRE t.topicId IS UNIQUE;
5     CREATE CONSTRAINT UniqueTopicName
6         FOR (t:Topic) REQUIRE t.name IS UNIQUE;
7
8     CREATE INDEX FOR (n:Person) ON (n.personId);
9     CREATE INDEX FOR (n:Topic) ON (n.topicId);
10
11    CREATE TEXT INDEX PersonFirstName
12        FOR (p:Person) ON (p.firstName);
13    CREATE TEXT INDEX PersonLastName
14        FOR (p:Person) ON (p.lastName);
```

Code 3.10: ***Create indexes and constraints***

Notice how constraints and indexes can be defined without defining labels and properties first.

Setup APOC

APOC, short for *Awesome Procedures on Cypher*, is a standard utility library for common procedures and functions. With over 450 well-supported procedures, it is the largest and most-widely used extension for Neo4j. To set it up, the jar needs to be downloaded from the Neo4j

website and placed in the plugins directory. After a restart of Neo4j, the plugin is ready to be used.

Allow Remote Access

By default, a Neo4j server can only be accessed locally, because it binds to the IP address `127.0.0.1`. To allow remote connections, the configuration of the server needs to be adjusted. The configuration option `dbms.default_listen_address` must be set to `0.0.0.0`. Additionally, a firewall rule needs to be created allowing TCP traffic on port 7687.

Optimize Memory Usage

Neo4j comes with a CLI tool `neo4j-admin` that has a built-in command to get an initial memory recommendation for the system it is running on. For the virtual machine used for this thesis, the recommendation is:

```
server.memory.heap.initial_size=3500m
server.memory.heap.max_size=3500m
server.memory.pagecache.size=1900m
```

The recommended values need to be added to the configuration file manually.

3.3 Import Test Data

The test dataset, available as CSV files, needs to be imported into both databases. Both systems provide native support for reading and importing data from CSV files.

3.3.1 Import Data into MySQL Database

The query to import data into the MySQL database is straight forward:

```
1    LOAD DATA INFILE '/var/lib/mysql-files/persons.csv'
2    INTO TABLE person
3    FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED BY '\n'
4    IGNORE 1 ROWS;
```

Code 3.11: ***Query to import CSV data into a MySQL table***

The query expects the path to the CSV file and the name of the table the data should be imported into. In line 3, common CSV settings are defined. The setting for `FIELDS TERMINATED BY` defines the separator between values used in the file. `ENCLOSED BY` defines the type of quotes used to enclose values that might contain a separator and `LINES TERMINAED BY` defines the character(s) used to mark the end of a line. The option `IGNORE 1 ROWS` allows the first row containing the column header to be ignored.

With the database schema introduced in section 3.2.1, MySQL tries to cast the values into the expected format for each column automatically. In case that’s not possible, the system returns an error.

The above query can be repeated for all CSV files of the dataset, only the file name and the table name need to be adjusted.

The best way to verify the successful import and to inspect the result is using a client application with a graphical user interface. MySQL provides such an application, it is called *MySQL Workbench* [14].

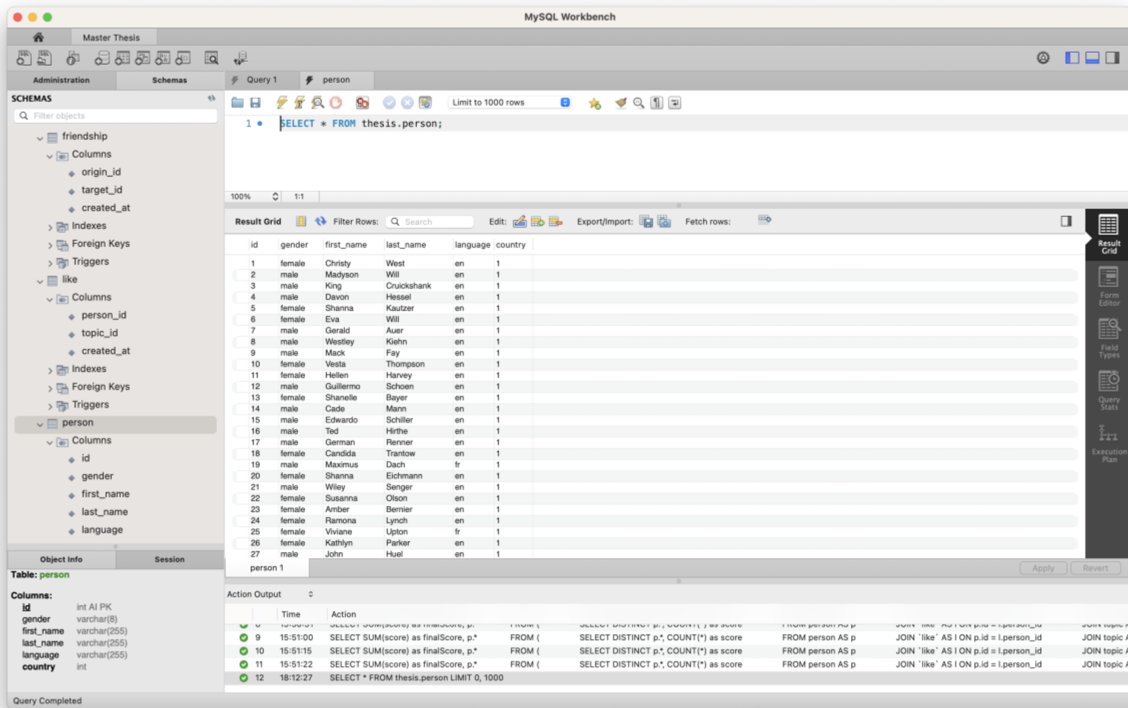

Figure 3.1: ***MySQL Workbench listing person records***

3.3.2 Import Data into Neo4j Database

The process of importing data into the Neo4j database is quite similar to the one for MySQL. The big difference is the missing schema. While the MySQL database has a predefined schema with tables and columns, Neo4j doesn't know what nodes, relations, and properties to expect. Therefore, it cannot cast the values automatically, it needs to be configured manually.

```
1     LOAD CSV WITH HEADERS FROM 'file:///persons.csv' AS row
2     WITH toInteger(row['id']) AS personId,
3          row['gender'] AS gender,
4          row['first_name'] AS firstName,
5          row['last_name'] AS lastName,
6          row['language'] AS language,
7          row['country'] AS country
8     MERGE (p:Person {personId: personId})
9       SET p.gender = gender,
10          p.firstName = firstName,
11          p.lastName = lastName,
12          p.language = language,
13          p.country = country
14    RETURN count(p)
```

Code 3.12: ***Query to import CSV data into a Neo4j database as nodes***

The command on line 1 instructs the system to load the CSV file *persons.csv*, iterate over each row of the file and assign it to the variable `row`. The first row is ignored because of the option `WITH HEADERS`.

Lines 2 to 7 prepare the values of the row for the import. This is where the casting takes place. In the case of a person, all values except for the id are strings and therefore don't require manual casting. The id is expected to be an integer, therefore it is casted to an integer using the function `toInteger()`. Neo4j provides several methods to cast values into a specific type.

Lines 8 to 13 create a new node with a label *Person* and a property *personId* set to the ID prepared in line 2. That is, only if a node with mentioned id does not exist yet. Otherwise, the existing node and its properties would be updated. The reason for this behaviour is the `MERGE` clause. By using `MERGE` instead of `CREATE`, the query doesn't fail in case of a duplicated ID. This is especially interesting when importing data into a database that is already populated with data.

Finally, with the `RETURN` clause the return value of the query can be defined, which here is the number of nodes created.

The query for importing topics is quite similar because topics are nodes as well. As opposed to that, friendships and likes are relationships. They require a slightly different query:

```
1     LOAD CSV WITH HEADERS FROM 'file:///likes.csv' AS row
2     CALL {
3       WITH row, datetime(replace(row['created_at'], ' ', 'T') as cr
4       MATCH (p:Person {personId: toInteger(row['person_id'])})
5       MATCH (t:Topic {topicId: toInteger(row['topic_id'])})
6       MERGE (p)-[:LIKES {createdAt: cr)}]->(t)
7     } IN TRANSACTIONS OF 500 ROWS
```

Code 3.13: ***Query to import CSV data into a Neo4j database as relationships***

Instead of creating a new node, a new relationship needs to be created between a node with label *Person* and a node with label *Topic*. Lines 4 and 5 match the person as *p* and the topic as *t*. On lines 6 to 8, a new relation is created with the additional property *createdAt*.

Because of the large number of friendships and likes, both queries are wrapped into a `CALL` clause. This allows to specify the maximum number of rows to be committed in one transaction. The more rows in one commit, the more memory is required.

Again, the best way to verify the successful import and to inspect the result is using a client application with a GUI. Neo4j provides such an application, it is called *Neo4j Desktop* [15].

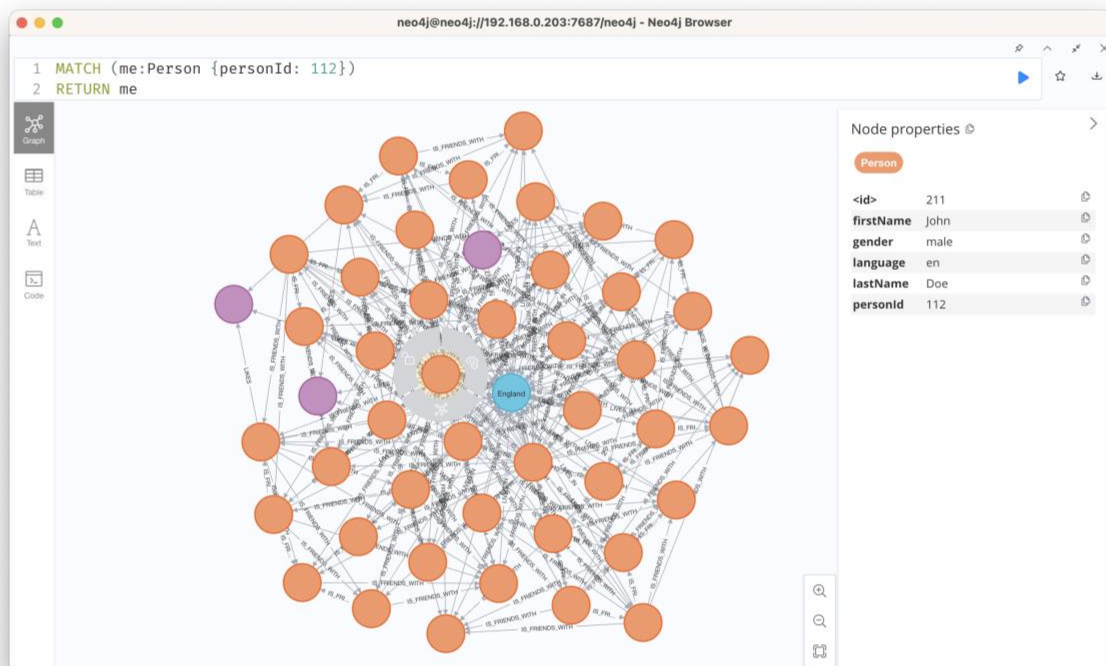

Figure 3.2: ***Neo4j Desktop presenting a graph with nodes and relationships***

3.4 Optimizing Schema and Data

After the import of the test dataset, the databases are ready to be queried. There remains potential for optimization, tough. One obvious optimization relates to the country a person lives in. The column contains the name of a country which is a redundant piece of information. This requires normalization.

For MySQL, a new table called *country* is introduced and instead of storing the name of the country on the person, the ID of the country is stored.

```
1     CREATE TABLE country (
2         id int NOT NULL AUTO_INCREMENT,
3         name varchar(255) NOT NULL,
4         PRIMARY KEY (id)
5     ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
      COLLATE=utf8mb4_0900_ai_ci;
6
7     INSERT INTO country (name)
8     VALUES ('England'), ('France'), ('Germany');
9
10    UPDATE person AS p
11    INNER JOIN country AS c ON p.country = c.name
12    SET p.country = c.id;
13
14    ALTER TABLE person MODIFY country INT NOT NULL;
15
16    ALTER TABLE person
17        ADD CONSTRAINT fk_country_id FOREIGN KEY (country)
18        REFERENCES country (id);
```

Code 3.14: ***SQL queries to normalize the column country***

The approach for Neo4j is very similar. A new node label `:Country` is introduced together with a constraint to guarantee the country name is unique. The property *country* is replaced with a new relationship named `:LIVES_IN`.

```
1     CREATE CONSTRAINT UniqueCountryName
2         FOR (c:Country) REQUIRE c.name IS UNIQUE;
3
4     CREATE (c:Country {name: 'England'});
5     CREATE (c:Country {name: 'France'});
6     CREATE (c:Country {name: 'Germany'});
7
8     MATCH (p:Person)
9     MATCH (c:Country {name: p.country})
10    CREATE (p)-[:LIVES_IN]->(c);
11
12    MATCH (p:Person)
13    REMOVE p.country;
```

Code 3.15: ***SQL query to insert bidirectional friendship records***

When running some experimental queries on the table *friendship*, it becomes obvious, that querying unidirectional self-referencing records is inefficient and results in large queries. If person A is a friend of person B, it means that person B is also a friend of person A. If there is only one record in the table *friendship* with an origin and a target, every query needs to evaluate both directions, from origin to target and from target to origin. It is not only inefficient, but also confusing.

A better approach is to model the relationship bidirectional. Every friendship is stored as two records, one with the origin A and the target B, one with the origin B and the target A. Even though this is a violation of the *domain-key normal form* [Tho88], it is the way to go because it has shown to be twice as fast, and the queries are much simpler.

```
1     INSERT INTO friendship (origin_id, target_id, created_at)
2     SELECT target_id, origin_id, created_at FROM friendship;
```

Code 3.16: ***SQL query to insert bidirectional friendship records***

As a last step, MySQL offers a way to optimize the physical storage of the table data and associated index data. This reduces storage space and improves I/O efficiency when accessing the table [13].

```
1     OPTIMIZE TABLE person, topic, friendship, `like`, country;
```

Code 3.17: ***SQL query to optimize the table storage***

3.5 Queries for the Use Cases

The main element for comparing the two database systems in this thesis is the set of queries for fetching the data. Usually, there are several ways to query the same set of data, but they can differ significantly in terms of performance. Writing efficient queries is a complex, time-consuming, and tedious work. It is a profession on its own.

The queries for the use cases defined in section 1.2 are listed and explained below. It's important to note that they contain a placeholder for the ID of the user called `$userId`. This placeholder is substituted with the actual user ID before the queries are passed to the database management system.

```
1     SELECT p.*
2     FROM person AS p
3       INNER JOIN friendship AS f ON p.id = f.target_id
4     WHERE f.origin_id = $userId
5     ORDER BY p.id;
```

Code 3.18: ***SQL query for use case 1***

```
1     MATCH
2       (me:Person {personId: $userId})-[:IS_FRIENDS_WITH]-(p:Person)
3     RETURN p.personId, p.firstName. p.lastName
4     ORDER BY p.personId;
```

Code 3.19: ***Cypher query for use case 1***

The queries for the first use case are rather simple. They return all persons that have a relationship of type *friendship* with the person with the ID `$userId`.

```
1     WITH RECURSIVE paths (current_target_id, distance, path) AS (
2       SELECT origin_id, 0, CAST(origin_id AS CHAR(100))
3       FROM friendship
4       WHERE origin_id = $userId
5       UNION DISTINCT
6       SELECT friendship.target_id, distance + 1,
7         CONCAT(paths.path, ',', friendship.target_id)
8       FROM paths
9       JOIN friendship ON current_target_id = friendship.origin_id
10      WHERE distance < 3
11      AND NOT FIND_IN_SET(friendship.target_id, path)
12    )
13    SELECT DISTINCT p.*, distance, path, mut.mutual_friends
14    FROM paths
15    JOIN person p ON p.id = current_target_id
16    LEFT JOIN (
17      SELECT mf1.origin_id as personId, count(*) as mutual_friends,
18        2 as dist
19      FROM friendship AS mf1
20      JOIN friendship AS mf2
21      ON mf1.target_id = mf2.origin_id
22        AND mf1.origin_id <> mf2.target_id
23      WHERE mf2.target_id = $userId
24        AND mf1.origin_id NOT IN (
25          SELECT f.target_id FROM friendship f
26          WHERE f.origin_id = $userId
27        )
28      GROUP BY mf1.origin_id, mf2.target_id
29    ) AS mut ON mut.personId = p.id AND mut.dist = distance
30    WHERE current_target_id <> $userId
31      AND p.id NOT IN (
32        SELECT target_id FROM friendship WHERE origin_id = $userId
33      )
34    GROUP BY p.id
35    ORDER BY mut.mutual_friends DESC, distance, p.first_name,
36      p.last_name
37    LIMIT 100;
```

Code 3.20: ***SQL query for use case 2***

The SQL query presented in listing 3.20 makes use of an advanced concept called *recursive common table expression* (CTE). A CTE is a named temporary result set that exists within the scope of a single statement and that can be referred to later within that statement [12]. A recursive CTE can be used for series generation and traversing hierarchical or tree-structured data.

The first `SELECT` statement starting on line 2 is a non-recursive statement. It provides the initial row for the dataset. The second `SELECT` statement starting on line 6 is a recursive statement. It produces the result set iteratively until the condition provided in the `WHERE` clause on line 10 is true.

The recursive CTE generates the temporary result set *paths* containing the paths and the distance (number of hops) for the friendships of the person with the ID `$userId`. The condition to stop the recursion requires the distance to be lower than 3. If more relationships should be traversed, that number could be raised.

The *6 degrees of separation theory* claims, that every person can get to know anyone by connecting though 6 people, or fewer [Fri29]. Raising the distance limit can quickly lead to an enormous result set.

For all the paths, mutual friends are determined.

```
1     MATCH (me:Person {personId: $userId})
2     CALL apoc.path.expandConfig(me, {
3         relationshipFilter: "IS_FRIENDS_WITH",
4         minLevel: 1,
5         maxLevel: 3,
6         uniqueness: "NODE_GLOBAL"
7     })
8     YIELD path
9     WITH me, apoc.path.elements(path) as pathElements, path
10    WITH me, length(path) as distance,
11      pathElements[size(pathElements) - 1] as friend
12    OPTIONAL MATCH
13      mfs=(me)-[:IS_FRIENDS_WITH]-(mf)-[:IS_FRIENDS_WITH]-
      (friend:Person)
14    WHERE distance = 2
15    RETURN friend, distance, count(DISTINCT mfs) AS mutualFriends
16    ORDER BY mutualFriends DESC, distance, friend.firstName,
17      friend.lastName
18    LIMIT 100;
```

Code 3.21: ***Cypher query for use case 2***

The cypher query follows the same approach. It makes use of the APOC library to expand the path of friendship relations starting at the person with the ID `$userId`. The uniqueness setting `NODE_GLOBAL` ensures every node is visited only once. The max level of 3 limits the distance to 3 hops.

Again, for all the paths, mutual friends are determined.

```
1     SELECT SUM(score) as finalScore, p.*
2     FROM (
3       SELECT DISTINCT p.*, COUNT(*) * 5 as score
4       FROM person AS p
5       JOIN `like` AS l ON p.id = l.person_id
6       JOIN topic AS t ON l.topic_id = t.id
7       WHERE t.id IN
8         (SELECT topic_id FROM `like` WHERE person_id = $userId)
9       AND p.id <> $userId
10      AND p.id NOT IN
11        (SELECT target_id FROM friendship WHERE origin_id = $userId)
12      GROUP BY p.id
13      UNION ALL
14      SELECT DISTINCT p.*, 1 as score
15      FROM person AS p
16      WHERE p.country =
17        (SELECT country FROM person WHERE id = $userId)
18      AND p.id <> $userId
19      AND p.id NOT IN
20        (SELECT target_id FROM friendship WHERE origin_id = $userId)
21      UNION ALL
22      SELECT DISTINCT p.*, mut.mutual_friends * 0.25 AS score
23      FROM person AS p
24      JOIN friendship AS f1 ON p.id = f1.target_id
25      JOIN (
26        SELECT mf1.origin_id, mf2.origin_id as personId,
27          count(*) as mutual_friends
28        FROM friendship AS mf1
29        JOIN friendship AS mf2 ON mf1.target_id = mf2.target_id
30          AND mf1.origin_id <> mf2.origin_id
31        WHERE mf1.origin_id = $userId
32          AND mf2.origin_id NOT IN
33            (SELECT f.target_id AS friend_id_2 FROM friendship f
34             WHERE f.origin_id = $userId)
35        GROUP BY mf1.origin_id, mf2.origin_id
36      ) AS mut ON mut.personId = f1.target_id
37      WHERE f1.origin_id IN
38        (SELECT f.target_id AS friend_id FROM friendship f
39         WHERE f.origin_id = $userId)
40      AND f1.target_id NOT IN
41        (SELECT f.target_id AS friend_id_2 FROM friendship f
42         WHERE f.origin_id = $userId)
43      AND f1.target_id <> $userId
44    ) AS p
45    GROUP BY p.id
46    ORDER BY finalScore DESC, p.first_name, p.last_name
47    LIMIT 100;
```

Code 3.22: ***SQL query for use case 3***

The SQL query for use case 3 makes use of a subquery. The sub query combines persons based on different criteria and provides a score for each criterion. The main query groups the persons by their ID and calculates the final score based on the sum of the individual scores.

The individual scores are determined based on the number of common relationships. The score for common interests is boosted by 500% and the score for mutual friends is reduced to 25%.

```
1     MATCH (me:Person { personId: $userId })
2     CALL {
3         WITH me
4         OPTIONAL MATCH pMutualFriends=(me)-[:IS_FRIENDS_WITH]-
5           (mf:Person)-[:IS_FRIENDS_WITH]-(friend:Person)
6         WHERE NOT EXISTS ((me)-[:IS_FRIENDS_WITH]-(friend))
7         RETURN friend, count(DISTINCT pMutualFriends) * 0.25 AS score
8       UNION ALL
9         WITH me
10        OPTIONAL MATCH pSameCountry=(me)-[:LIVES_IN]->
11          (c:Country)<-[:LIVES_IN]-(friend:Person)
12        WHERE NOT EXISTS ((me)-[:IS_FRIENDS_WITH]-(friend))
13        RETURN friend, count(DISTINCT pSameCountry) AS score
14      UNION ALL
15        WITH me
16        OPTIONAL MATCH pTopics=(me)-[:LIKES]->
17          (topic:Topic)<-[:LIKES]-(friend:Person)
18        WHERE NOT EXISTS ((me)-[:IS_FRIENDS_WITH]-(friend))
19        RETURN friend, count(DISTINCT pTopics) * 5 AS score
20    }
21    RETURN DISTINCT friend, sum(score) as score
22    ORDER BY score DESC
23    LIMIT 100;
```

Code 3.23: ***Cypher query for use case 3***

The cypher query for use case 3 makes use of a subquery as well. It follows the same algorithm that's used for the SQL query and applies the same weights.

The queries for use case 4 not only use the placeholder `$userId` but also the placeholder `$s` containing the string input from the search field.

```
1     (SELECT
2       CONCAT(p.first_name, ' ', p.last_name) AS label,
3       (CASE WHEN f.origin_id IS NOT NULL THEN true ELSE false END)
4         AS connected,
5       (CASE
6         WHEN CONCAT(p.first_name, ' ', p.last_name) LIKE '$s' THEN 10
7         WHEN CONCAT(p.first_name, ' ', p.last_name) LIKE '$s%' THEN 5
8         WHEN CONCAT(p.first_name, ' ', p.last_name) LIKE '%$s' THEN 1
9         ELSE 3
10      END) AS score, 'Person' as type, p.id
11    FROM person AS p
12    LEFT JOIN friendship AS f
13      ON p.id = f.target_id AND f.origin_id = $userId
14    WHERE
15      CONCAT(p.first_name, ' ', p.last_name) LIKE '%$s%'
16    ORDER BY
17      (CASE WHEN origin_id IS NOT NULL THEN 1 ELSE 2 END),
18      (CASE
19        WHEN CONCAT(p.first_name, ' ', p.last_name) LIKE '$s' THEN 1
20        WHEN CONCAT(p.first_name, ' ', p.last_name) LIKE '$s%' THEN 2
21        WHEN CONCAT(p.first_name, ' ', p.last_name) LIKE '%$s' THEN 4
22        ELSE 3
23      END), first_name, last_name
24    LIMIT 5)
25    UNION
26    (SELECT
27      t.name AS label,
28      (CASE WHEN l.person_id IS NOT NULL THEN true ELSE false END)
29        AS connected,
30      (CASE
31        WHEN t.name LIKE '$s' THEN 10
32        WHEN t.name LIKE '$s%' THEN 5
33        WHEN t.name LIKE '%$s' THEN 1
34        ELSE 3
35      END) AS score, 'Topic' AS type, t.id
36    FROM topic AS t
37    LEFT JOIN `like` AS l ON t.id = l.topic_id
38      AND l.person_id = $userId
39    WHERE
40      t.name LIKE '%$s%'
41    ORDER BY
42      (CASE WHEN l.person_id IS NOT NULL THEN 1 ELSE 2 END),
43      (CASE
44        WHEN t.name LIKE '$s' THEN 1
45        WHEN t.name LIKE '$s%' THEN 2
46        WHEN t.name LIKE '%$s' THEN 4
47        ELSE 3
48      END), t.name LIMIT 5)
49    ORDER BY connected DESC, score DESC
```

Code 3.24: ***SQL query for use case 4***

Both, the SQL query, and the cypher query, make heavy use of string comparison for use case 4. To return the best possible result for the search string provided as input, several iterations of string matching are required. First, records are selected that contain the search string `$s` at any position in the name. Second, based on the position of the search string in the name, a score is assigned. An exact match gets the highest score, a match at the beginning of the name gets the second-highest score, a match at the end of the name gets the lowest score.

This algorithm is applied for persons and for topics. Finally, the results are combined using the `UNION` clause and then ordered, first by the fact if a connection exists (friendship in case of a person, like in case of a topic), then by score.

```
1     CALL {
2       MATCH (f:Person)
3       WITH toLower(f.firstName + ' ' + f.lastName) AS s, f
4       WHERE s CONTAINS '$s'
5       WITH s, f, exists((f)-[:IS_FRIENDS_WITH]-
6         (:Person {personId: $userId})) as connected, 'Person' AS type
7       RETURN f.personId AS id,
8         f.firstName + ' ' + f.lastName AS label, connected, type,
9         (CASE WHEN s = '$s' THEN 10
10        WHEN s STARTS WITH '$s' THEN 5
11        WHEN s ENDS WITH '$s' THEN 1
12        ELSE 3 END) AS score
13      ORDER BY
14        connected DESC,
15        (CASE WHEN s = '$s' THEN 1
16        WHEN s STARTS WITH '$s' THEN 2
17        WHEN s ENDS WITH '$s' THEN 4
18        ELSE 3 END), f.firstName, f.lastName
19      LIMIT 5
20      UNION
21      MATCH (t:Topic)
22      WITH t, toLower(t.name) AS searchField
23      WHERE s CONTAINS '$s'
24      WITH t, s, exists((t)<-[:LIKES]-
25        (:Person {personId: $userId})) AS connected, 'Topic' AS type
26      RETURN t.topicId AS id, t.name AS label, connected, type,
27        (CASE WHEN s = '$s' THEN 10
28        WHEN s STARTS WITH '$s' THEN 5
29        WHEN s ENDS WITH '$s' THEN 1
30        ELSE 3 END) AS score
31      ORDER BY
32        connected DESC,
33        (CASE WHEN s = '$s' THEN 1
34        WHEN s STARTS WITH '$s' THEN 2
35        WHEN s ENDS WITH '$s' THEN 4
36        ELSE 3 END), s
37      LIMIT 5
38    }
39    RETURN id, label, connected, score, type
40    ORDER BY connected DESC, score DESC
```

Code 3.25: ***Cypher query for use case 4***

3.6 Validating the Queries

Validating the result of a query on a large dataset is hard to do, because it's difficult to reason about such a huge amount of data. That's the reason why a second, much smaller dataset was developed. With such a small dataset, it becomes easy to validate if a query is returning the expected result or if some parts are missing out.

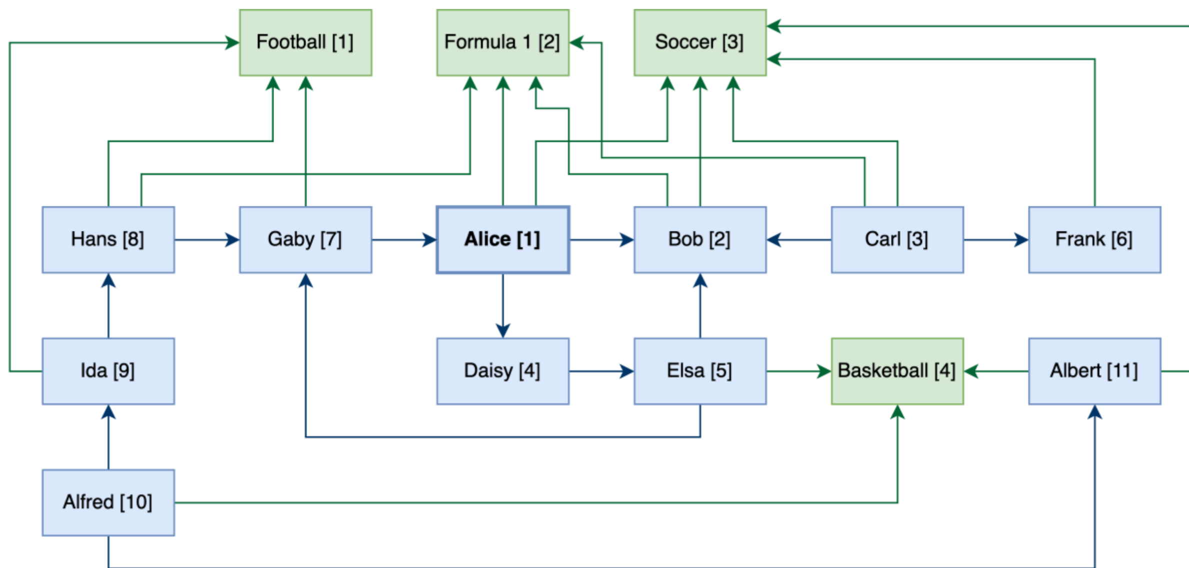

Figure 3.3: *Dataset for query validation*

To test the queries, they were executed in the context of the person Alice with ID 1.

3.7 Automation

Running predefined database queries is not a difficult task, but it involves some steps:

1. Open a terminal window
2. Connect to the database server
3. Enter the user credentials
4. Enter the query
5. Execute the query

While this is fine for an occasional test, it is too much of an overhead when running a systematic evaluation for several use cases with several repetitions.

To simplify the process of executing a series of queries and storing metadata such as the execution time for each query, a custom CLI command was implemented. A CLI command is a piece of code, that can be executed from the command line. It runs a sequence of tasks.

The custom CLI command implemented as part of this thesis runs the queries for all the use cases on both database systems for a selected set of users, measures the performance and logs the results.

4 Evaluation

4.1 Evaluation Criteria 29
4.1.1 Performance 29
4.1.2 Number of Clauses 30
4.1.3 Developer Convenience 30
4.2 Evaluation 30
4.3 Result 31
4.3.1 Result of Performance Evaluation 31
4.3.2 Result of Number of Clauses Evaluation 33
4.3.3 Result of Developer Convenience Evaluation 33
4.4 Discussion 33

4.1 Evaluation Criteria

4.1.1 Performance

The first criterion for the evaluation of the database systems is the performance of the queries. The performance is measured as time it takes to run the query and fetch the result in milliseconds.

Both databases have caching mechanisms in place, therefore, simply running the same query repeatedly is not a fair evaluation. Disabling the cache would not be a fair evaluation, too, because in a production environment, caches will always be enabled.

A fair way of conducting performance measurement in the context of a web application is to run a large series of queries, like it happens in a production environment. The performance of each query is logged for evaluation.

The order in which the queries are executed is the same for both databases.

4.1.2 Number of Clauses

The second evaluation criterion is the number of clauses required to write the queries. This number serves as an indicator for the effort that needs be put into writing queries for each of the systems.

While some clauses are essential for the result of the query, others are optional. The optional clauses are not counted for this evaluation.

4.1.3 Developer Convenience

The third evaluation criterion is the developer convenience. Each query is classified into one of the four categories *easy*, *intermediate*, *hard*, and *ultra-hard* based on the cognitive load required to understand the query and the effort it took to write it. This classification serves as an indicator of maintainability.

This criterion is largely subjective but serves as an interesting addition to the second criterion.

4.2 Evaluation

To perform the performance evaluation, four persons were selected from the dataset. Care was taken to ensure these persons have a varying number of relationships.

User ID	Number of friends	Number of likes
112	43	3
2624	3	0
19143	44	0
55745	56	10

Table 4.1: ***Overview of users selected for evaluation***

Based on the CLI command presented in section 3.7, the performance evaluation was conducted.

To perform the number of clauses evaluation, the clauses were counted manually, and the result was tabulated. Table 4.2 lists which clauses were counted for the number of clauses evaluation and which ones were ignored.

Database	Clauses counted	Clauses not counted
MySQL	`SELECT, FROM, JOIN, ORDER BY, WITH RECURSIVE, UNION, GROUP BY, LIMIT, IN, NOT IN, CASE, LIKE, WHEN/THEN, ELSE, CAST(), CONCAT(), FIND_IN_SET(), SUM()`	`AS, ON, DISTINCT`
Neo4j	`MATCH, OPTIONAL MATCH, RETURN, ORDER BY, CALL, YIELD, WITH, WHERE, LIMIT, EXISTS, NOT EXISTS, CONTAINS, length(), count(),`	`AS, DISTINCT`

	apoc.path.expandConfig, apoc.path.elements, sum(), toLower(), {}-Filter, -[]-()	

Table 4.2: *Clauses that were counted during the evaluation*

For the developer convenience evaluation, the queries were studied and classified.

4.3 Result

4.3.1 Result of Performance Evaluation

Figures 4.1 to 4.4 present the result of the performance evaluation as box plot diagrams.

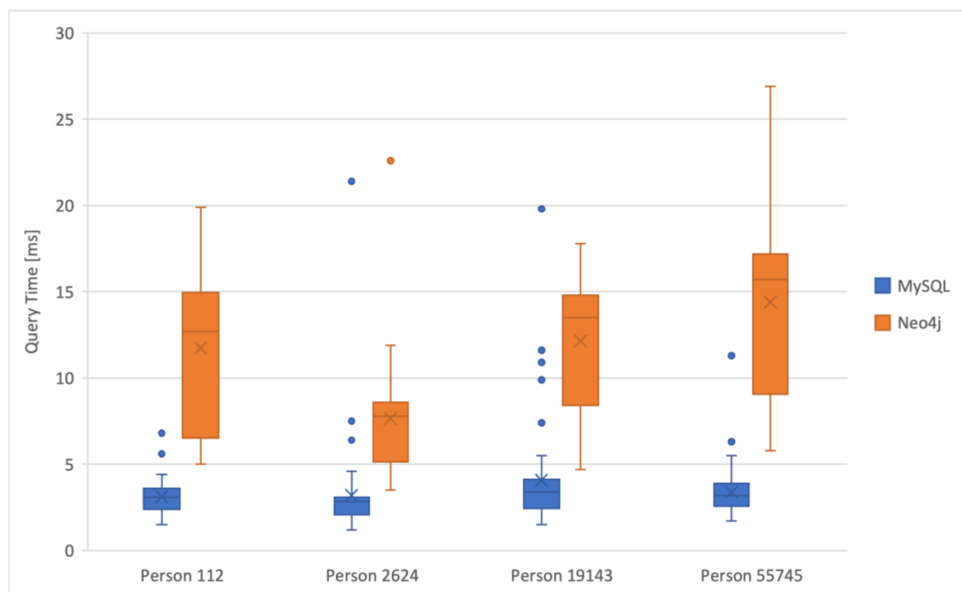

Figure 4.1: *Query performance comparison for list of friends*

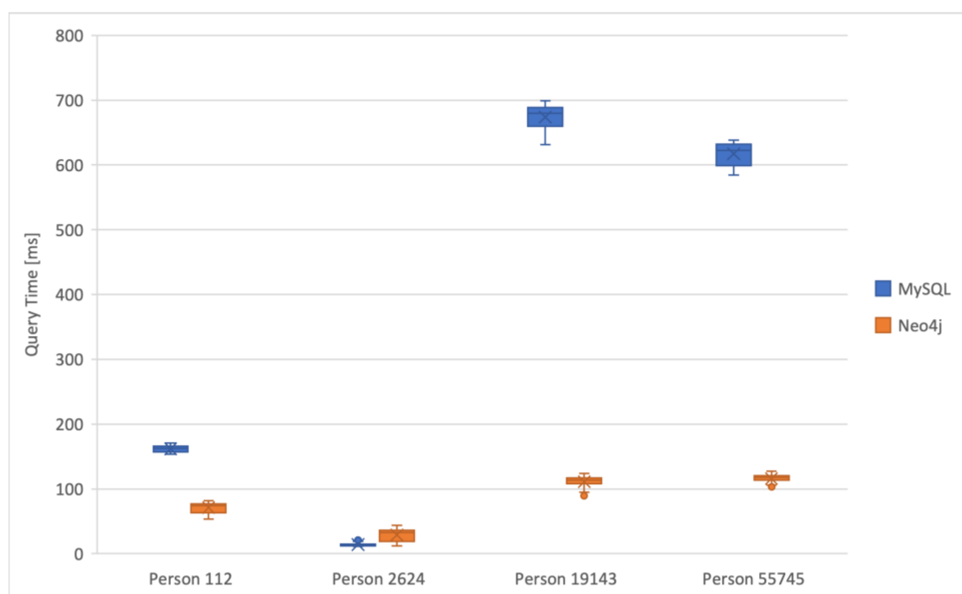

Figure 4.2: *Query performance comparison for friends recommendation #1*

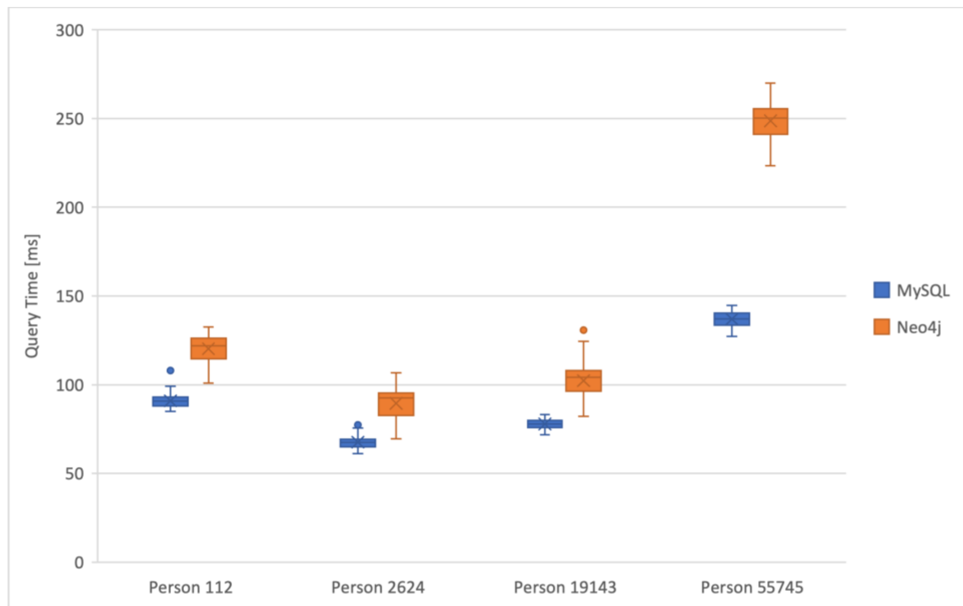

Figure 4.3: ***Query performance comparison for friends recommendation #2***

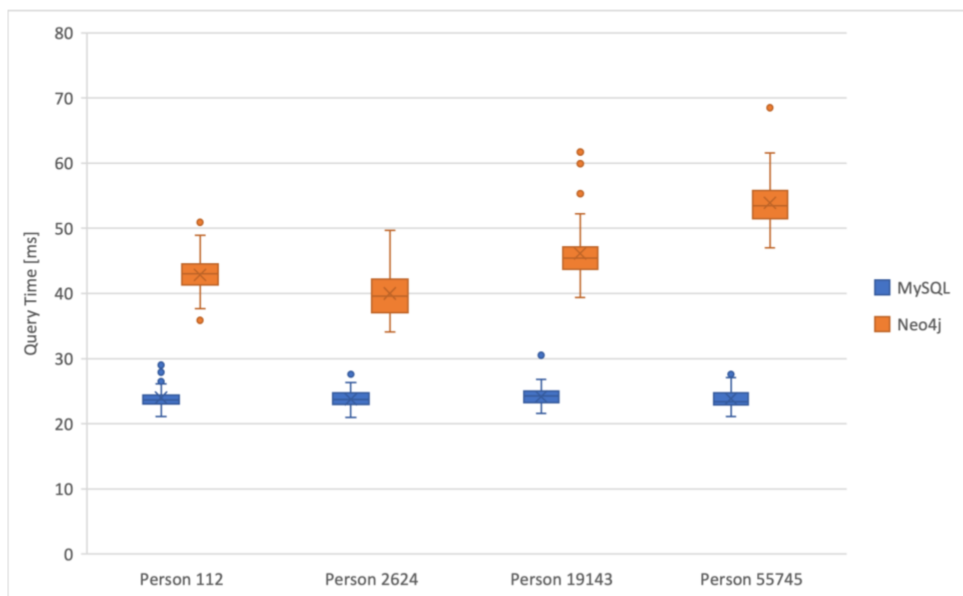

Figure 4.4: ***Query performance comparison for auto-suggest***

4.3.2 Result of Number of Clauses Evaluation

Figure 4.5 presents the result of the number of clauses evaluation as a bar chart.

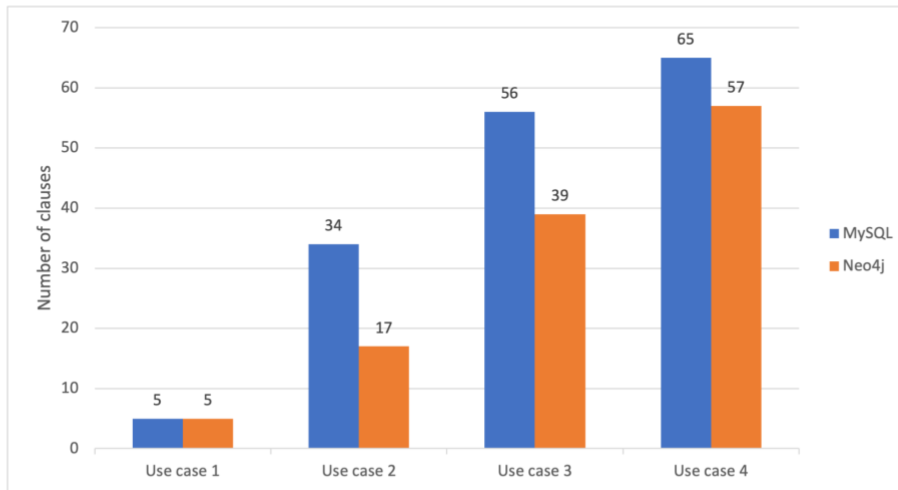

Figure 4.5: ***Result of number of clauses evaluation***

4.3.3 Result of Developer Convenience Evaluation

Table 4.3 presents the result of the developer convenience evaluation.

Use case	MySQL	Neo4j
List of friends	Easy	Easy
Friends recommendation	Ultra hard	Hard
Friends recommendation #2	Hard	Intermediate
Auto-suggest	Hard	Hard

Table 4.3: ***Result of developer convenience evaluation***

4.4 Discussion

The results of the evaluation presented in section 4.3 paint a mixed picture. The trend is not the same for all criteria.

The performance evaluation shows that the performance is strongly dependent on the use case and the data set.

For use case 1, MySQL delivers the better performance. This is interesting in the sense that MySQL needs to execute a join of the two tables *person* and *friendship* at runtime, while Neo4j needs to fetch persisted relationships for a node in a graph. The obvious assumption would be

that Neo4j could serve this use case more performantly. It is not possible to say conclusively, why this is the case.

For use case 2, the effect that was surprising for use case 1 emerges. MySQL needs to execute a number of joins at runtime, which depends on the number of people that can be reached by traversing all the friendships with a distance of 3 starting at the original person. For the person with ID *2624*, the total number of persons that can be reached with said distance is 1451. For the person with ID *19143*, the total number is 19064. For each of these persons, the number of common friends must then be determined by using another join. This is a costly operation, and it has more impact, the larger the set of persons gets. Neo4j does not have to calculate any joins at runtime, it needs to traverse the persisted relationships of the graph.

Use case 3 shows the same trend for MySQL and Neo4j: The larger the number of involved records or nodes and relationships respectively, the longer it takes to execute the query, despite the limited result set. For the person with ID *2624*, there are 0 people involved with the same interests as the original person, 19'996 people living in the same country and 66 people with common friends. For the person with ID *55745*, there are 16,428 people involved with the same interests as the original person, 19’943 people living in the same country, and 2,302 people with mutual friends.

Here, the optimization of friendships as a bidirectional relationship in MySQL comes into play particularly well. In Neo4j, no such optimization was done, because in Neo4j, a relationship is automatically valid in both directions. In case of a bidirectional relationship, the direction should be omitted in the queries [Ian13, p. 65]. A quick test to measure the performance impact of unidirectional vs. bidirectional traversing of friendship relationships showed that for use case 3, directed traversing of the relationships is about 30% more efficient on average. This is an interesting topic for future work.

Use Case 4 depends heavily on matching a search string in one or multiple columns or properties at different positions. Comparing the other results, the performance for this use case is more consistent. It seems that MySQL is somewhat more efficient for this type of query.

Considering the number of clauses required for writing the queries, there is a clear trend. Neo4j requires up to 50% less clauses to write the queries for the same results as with MySQL. It implies, that writing Cypher queries is easier, less time consuming and less prone to errors.

This implication is backed by the results of the evaluation of developer convenience. On average, Cypher queries are rated to be easier to understand while SQL queries produce more cognitive load. Thanks to the ASCII-art type of syntax, reasoning about Cypher queries is easier, even for me, who has known SQL for more than 15 years but seen Cypher for the first time.

Looking at all the results combined, one can observe a slight trend towards Neo4j. However, determining which system performs better over all depends on the prioritization of the criteria and the use cases.

5 Prototype

5.1 Architecture **35**

5.2 Technology Stack **36**

5.3 Implementation **36**

5.3.1 REST API ... 36

5.3.2 Frontend Application ... 38

5.4 User Interface **38**

To demonstrate the use cases defined for this thesis in a live example, a prototype was planned, designed, and implemented. The following sections explain its architecture, describe the technology stack that was used and present the user interface.

5.1 Architecture

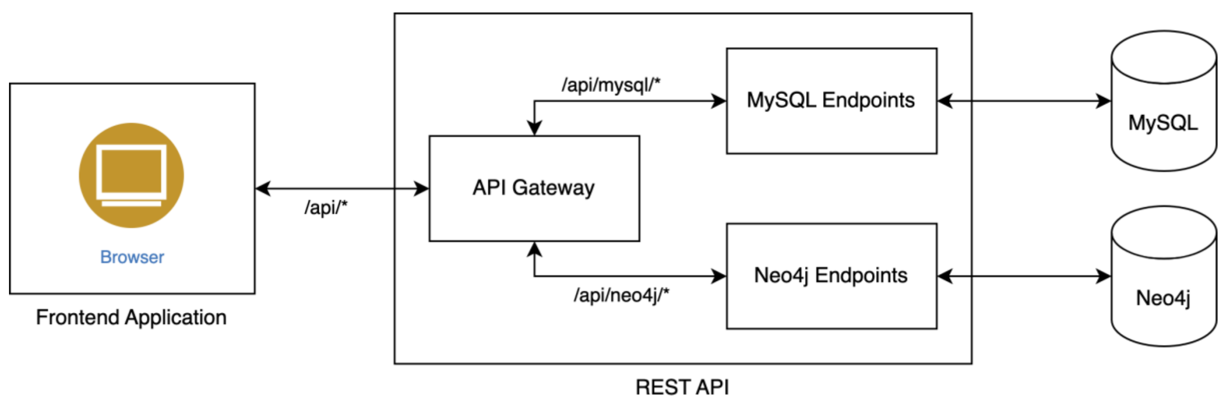

Figure 5.1: ***The architecture of the prototype***

The prototype is implemented as a single-page frontend application that communicates with a backend API based on RESTful principles. One of the big advantages of RESTful web services is the separation of concerns. The user interface is separated from the data storage, allowing the two components to evolve independently [FIE00].

Like many new software tools these days, the frontend is implemented as a web application. It does not require any software to be installed on a client, a simple web browser is enough to access the application.

The REST API provides two sets of endpoints, one for requests to be processed using the MySQL database and one for requests to be processed using the Neo4j database. The distinction is made based on the URI of the request. While request URIs prefixed with `/api/mysql/` are processed using MySQL, request URIs prefixed with `/api/neo4j/` are processed using Neo4j.

5.2 Technology Stack

The frontend application is implemented based on Angular [16]. Angular is a component-based framework for building scalable web applications. The framework built on TypeScript [17] is developed by Google and has first been released in 2010. Angular projects can scale from single-developer projects to enterprise-level applications. The framework provides a suite of developer tools that help developing, building, testing, and updating code [18].

The REST API is implemented based on PHP [19] and the web application framework Symfony [20]. Symfony was first released in 2005 under MIT license and has since been under active development. It aims to speed up the development of web application by offering a wide range of tools and presets replacing repetitive coding tasks. By implementing a lot of important design patterns such as MVC, factories and singletons and by sticking to the approach of domain-driven design, it helps writing good quality code.

To connect to the MySQL database, the Doctrine database abstraction layer (DBAL) [21] is used. While it would provide a large set of features to interact with several types of databases, it is only used to connect to the database and execute raw queries. This ensures the least amount of overhead.

To connect to the Neo4j database, the Neo4j PHP Client and Driver package [22] is used. The package provides the required drivers to access a Neo4j database using the Bolt protocol.

5.3 Implementation

5.3.1 REST API

Each endpoint is implemented as a controller method. A controller method typically follows this sequence:

1. Validate the request: Are all the required arguments provided as part of the request? If not, return a validation error.
2. Process the request: Prepare the query, hand over the query to the desired database server, fetch the result.
3. Return a response: Return the requested data as JSON [23].

```
1     #[Route('/api/mysql/friends', name: 'api_mysql_friends')]
2     public function friends(Request $request): Response
3     {
4         $userId = (int)$request->get('userId');
5         if ($userId === 0) {
6             return new JsonResponse('Invalid request', 400);
7         }
8
9         $query = MysqlQueryCollection::getFriendsListQuery($userId);
10        $results = $this->connection->fetchAllAssociative($query);
11
12        return new JsonResponse([
13            'data' => $results,
14            'count' => count($results),
15        ]);
16    }
```

Code 5.1: ***Controller method for use case 1***

Line 1 is the route definition, meaning that this method is called when a request comes in with the uri `/api/mysql/friends`.

Lines 4 to 6 take care about the validation of the request. First, the query parameter `userId` is casted to an integer and then it is verified, that it isn't equal to 0. If it is, a JSON response is returned with the HTTP status code 400, meaning *Bad Request* [FR14].

Lines 9 and 10 prepare the query, send it to the MySQL database and fetch the results.

Lines 12 to 15 return the fetched data as a JSON response to the client.

All endpoints are implemented following this schema. The available endpoints are listed in Table 5.1. It's important to note that *{db}* is a placeholder for either *mysql* or *neo4j*. Additionally, each endpoint expects a query parameter *userId* to be present defining in the context of which user the request should be executed.

URI	Method	Description
/api/{db}/friends	GET	Run query for use case 1
/api/{db}/friends-recommendation	GET	Run query for use case 2
/api/{db}/friends-recommendation-2	GET	Run query for use case 3
/api/{db}/auto-suggest?s=X	GET	Run query for use case 4 with search term X
/api/{db}/profile?profileId=X	GET	Return profile for person with ID X
/api/{db}/topic?topicId=X	GET	Return topic with ID X
/api/{db}/friend?profileId=X	POST	Add person with ID X as friend
/api/{db}/friend?profileId=X	DELETE	Remove person with ID X as friend
/api/{db}/like?topic=X	POST	Like topic with ID X
/api/{db}/like?topic=X	DELETE	Unlike topic with ID X

Table 5.1: ***Overview of REST API endpoints***

5.3.2 Frontend Application

The frontend application of the prototype is straight forward. It provides a view for each use case. Next to the result it gets from the REST API, it presents a performance statistic for each request and calculates the average value for each use case and database.

The statistics component is implemented based on an HTTP interceptor. It intercepts all requests and responses to and from the backend. If it recognizes a response for a use case, it stores the duration provided by the backend in a global statistics service.

```
1     export class HttpInterceptorService implements HttpInterceptor {
2       constructor(private statistics: StatisticsService) {}
3
4       public intercept(
5         req: HttpRequest<any>, next: HttpHandler
6       ): Observable<HttpEvent<any>> {
7         return next.handle(req).pipe(
8           map(event => this.logResponse(event)),
9         );
10      }
11
12      private logResponse(event: HttpResponse<any> | any) {
13        if (event instanceof HttpResponse) {
14          this.statistics.save(event)
15        }
16        return event;
17      }
18    }
```

Code 5.2: ***HTTP interceptor to keep track of performance***

The visualization of the statistics is presented in the next section.

5.4 User Interface

The goal of the prototype is to demonstrate the use cases covered in this thesis. For each use case, the prototype provides an interactive view to explore the behavior in action. A toolbar allows to adjust some parameters.

Figure 5.2: ***The left side of the toolbar***

The controls on the left side of the toolbar allow to switch between the use cases.

Figure 5.3: ***The right side of the toolbar***

The control in the middle allows to choose a person. By default, all requests are executed from the perspective of the person *John Doe* (ID 112). This is the person that was also used for the queries executed as part of the evaluation. To be able to execute the requests from the perspective of another person, the control provides an option to switch to the person called *Bella Rempel* (ID 41127) that was chosen randomly out of the dataset.

The control on the right side allows to choose the database engine to use for the requests. The endpoint used to fetch the data depends on this setting. Switching the control updates the current view of the application automatically with data from the newly selected database.

View of a Use Case

In figure 5.4, the view of use case 3 is presented. On the left side, the results of the query are listed as a table. On the right side, the performance statistics for the queries of that use case are listed and the average is calculated for both database systems. The button «Run again» allows to run the most recent query again.

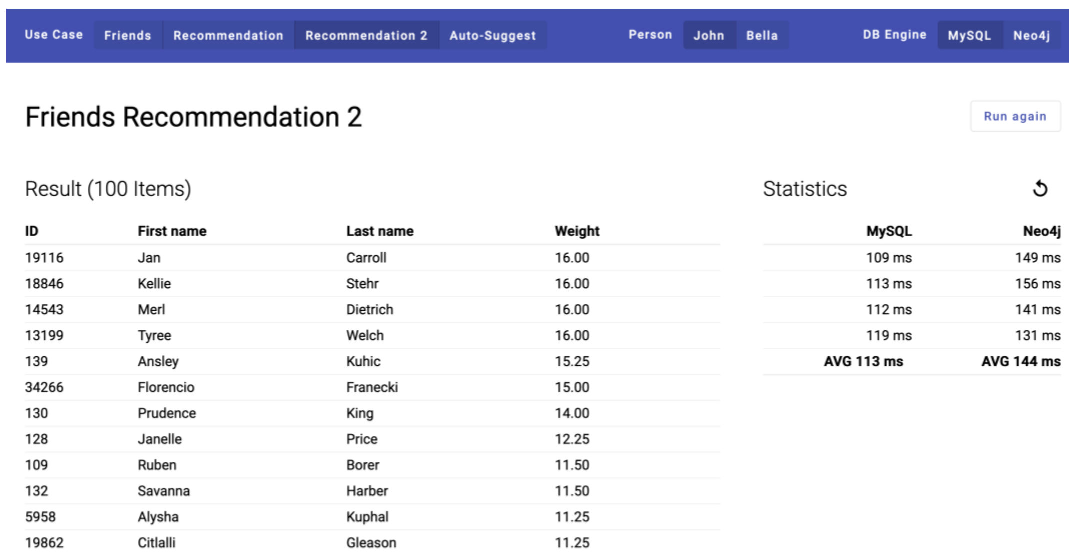

Figure 5.4: ***The view of use case 3***

View of a Person

The view of a person presents the topics the person likes and the friends of that person. Likes and friendships that are in common with the logged in user are presented first in the lists and get highlighted with a green checkmark to visualize it's a common interest or friend.

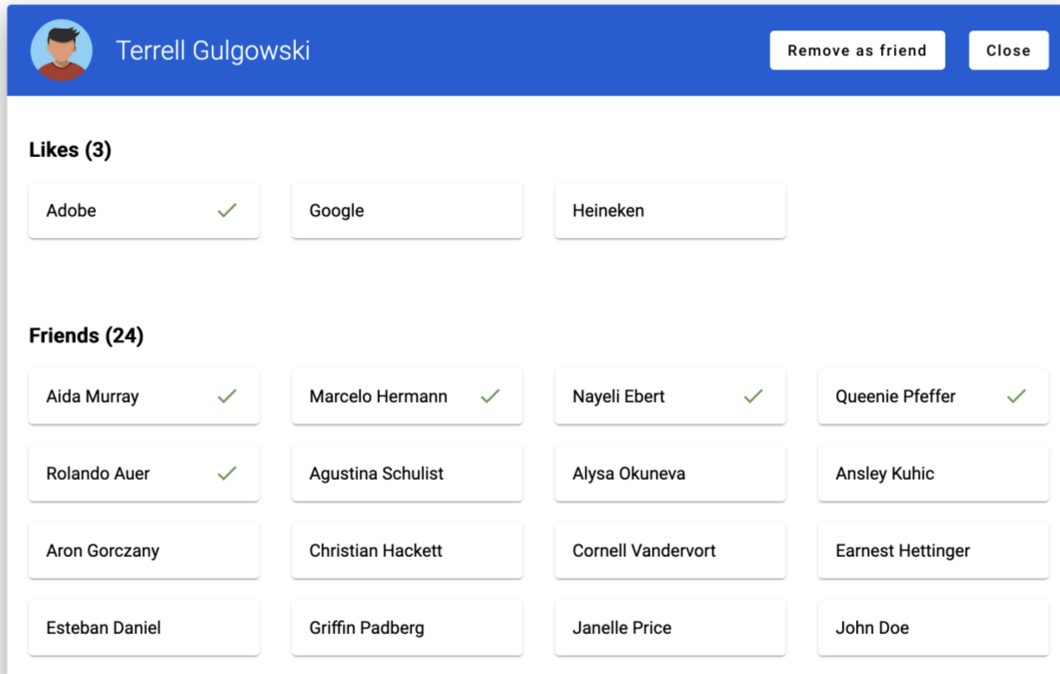

Figure 5.5: *The view of a person*

On the top right, two buttons provide the options to add or remove a person as friend and to close the view.

View of a Topic

The view of a topic presents the persons, that like that topic, up to a maximum of 200 persons. Persons, that are friends with the logged in user are presented first in the list and get highlighted with a green checkmark to visualize the common friendship.

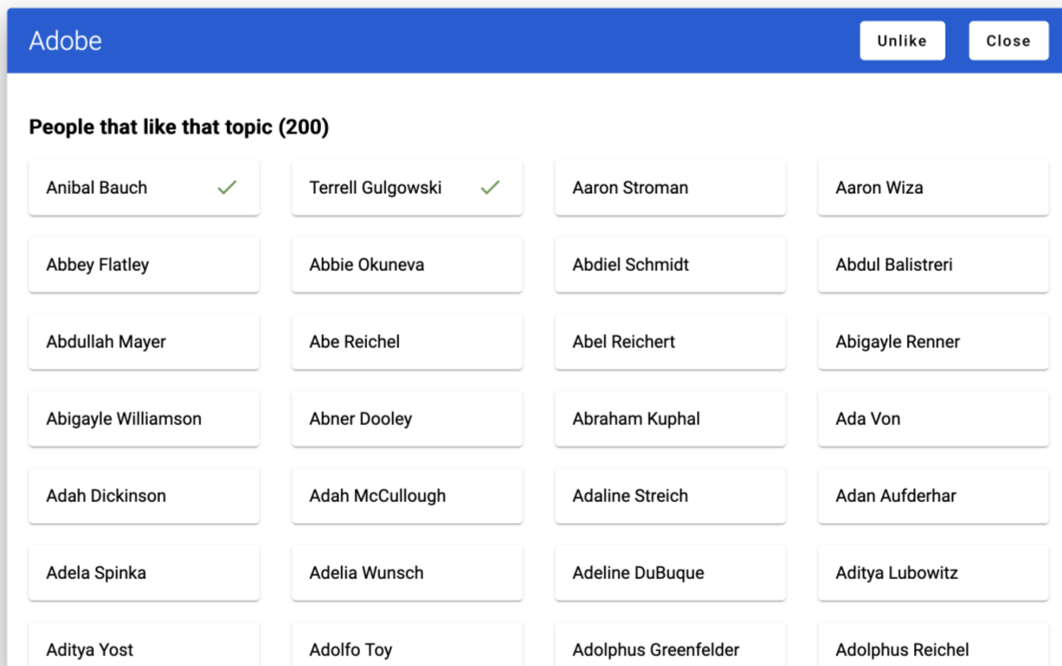

Figure 5.6: ***The view of a topic***

On the top right, two buttons provide the options to «Like» respectively «Unlike» the topic and to close the view.

6
Conclusion and Future Work

6.1 Conclusion 42
6.2 Future Work 42

6.1 Conclusion

The systematic evaluation of the two database management systems MySQL and Neo4j has shown several interesting and surprising aspects. There is no clear conclusion as to which system is universally better suited in the context of social web applications. Both systems have proven to be powerful, stable, and capable of managing and serving a large dataset in that context.

Depending on the prioritization of the criteria, one or the other system is more convincing. If only the query performance is considered, it depends strongly on the use case which database system is more efficient. If other factors such as the complexity of the queries, the cognitive load produced or the developer convenience are taken into account as well, there is a slight trend towards Neo4j.

Writing and maintaining database queries has shown to be a tedious and time-consuming task. It usually requires several iterations of testing and improvement to get a query that delivers the proper result in an efficient way. Having a small dataset at hand to test and validate the queries during development is a big help.

For large and complex platforms, it is worth evaluating a combination of the best of both worlds and choosing a hybrid approach as described in the next section about future work.

6.2 Future Work

In the course of this thesis, a few questions and ideas have come up, which would be very interesting to explore in a future work:

Unidirectional vs. Bidirectional Relationships

In Neo4j, a relationship is directed but it can be traversed in both directions depending on the query. A quick test showed that it has an impact on performance. It would be interesting to explore in-depth how big the impact is and what the alternatives are.

Caching

Instead of submitting complex queries to a normalized database at runtime, some queries could be processed in the background and stored in a cache. This would allow runtime queries to be handled more efficiently and with simpler queries. Use cases 2 and 3 would be good candidates for such a solution as they don't need to rely on up-to-date data.

Hybrid Solution

The performance evaluation showed that depending on the use case, both systems can be better performing. Mixed with the knowledge that Cypher queries are simpler to understand and write, it would be exciting to evaluate using Neo4j as the main database system but to fall back on a MySQL server for some use cases. The problem will then be to keep the data up to date in both database management systems. One possible approach would be to use the MySQL database only for cached results as mentioned before.

Clustering

Both MySQL and Neo4j offer the ability to combine multiple database servers into a cluster. It would be exciting to evaluate how this affects performance.

A

Source Code

The source code of this project is available as ZIP file and can be downloaded here: https://gitlab.com/ch-fries/master-thesis-prototype

The structure of the source code

The source code is organized in this structure:

```
/backend
/dataset
/evaluation
/frontend
/README.md
```

Explanation

The folder *backend* contains the source code for the REST API.

The folder *dataset* contains the generated dataset.

The folder *evaluation* contains the performance measurement data used for the evaluation.

The folder *frontend* contains the source code for the web application.

The file *README.md* explains how to run the project locally.

References

[Don81]

Donald D. Chamberlin, et al. A history and evaluation of System R. *Communications of the ACM*, 24.10, 1981, pages 632-646.

[Edg70|

Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13.6, 1970, pages 377-387.

[Est23]

Esteban Ortiz-Ospina and Max Roser. The rise of social media. *Our world in data*, 2023.

[Fie00]

Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine, 2000.

[FR14]

Roy Thomas Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, IETF, 2014.

[Fri29]

Frigyes Karinthy. Chain-links. *Everything is different*, 1929, pages 21-26.

[Ian13]

Ian Robinson, Jim Webber, and Emil Eifrem. Graph Databases, O'Reilly Media, 2013, ISBN 978-1-449-35626-2.

[Ken89]

Kent Layton and Martha E. Irwin. Enriching your reading program with databases. *The Reading Teacher*, 42.9, 1989, page 724.

[Ren18]

Renzo Angles. The Property Graph Database Model. *AMW*, 2018.

[Tho88]

Thompson, Richard Bryan. A study of the use of domain key normal form criteria in database design, page 2-6, 1988.

Referenced Web Resources

[1] DB-Engines: Knowledge Base of Relational and NoSQL Database Management Systems. https://db-engines.com/en/ranking (accessed on February 24, 2023)

[2] Neo4j Docs: Graph database concepts. https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/ (accessed March 18, 2023)

[3] Cypher Query Language. https://neo4j.com/developer/cypher/ (accessed March 18, 2023)

[4] Kaggle. https://www.kaggle.com/datasets (accessed April 16, 2023)

[5] Visual Capitalist: The Top 100 Most Valuable Brands in 2022. https://www.visualcapitalist.com/top-100-most-valuable-brands-in-2022/ (accessed April 28, 2023)

[6] Proxmox Virtual Environment. https://www.proxmox.com/en/proxmox-virtual-environment/overview (accessed April 14, 2023)

[7] MySQL: System Requirements. https://dev.mysql.com/doc/mysql-monitor/8.0/en/system-prereqs-reference.html (accessed April 14, 2023)

[8] Neo4j Docs: System Requirements. https://neo4j.com/docs/operations-manual/current/installation/requirements/ (accessed April 14, 2023)

[9] MySQL Community Downloads. https://dev.mysql.com/downloads/ (accessed July 2, 2023)

[10] Neo4j Download Center. https://neo4j.com/download-center/ (accessed April 14, 2023)

[11] Neo4j Docs: Installation. https://neo4j.com/docs/operations-manual/current/installation/linux/debian/ (accessed April 14, 2023)

[12] MySQL: WITH (Common Table Expressions) https://dev.mysql.com/doc/refman/8.0/en/with.html (accessed July 3, 2023)

[13] MySQL: Optimize table command. https://dev.mysql.com/doc/refman/8.0/en/optimize-table.html (accessed July 2, 2023)

[14] MySQL Workbench. https://www.mysql.com/products/workbench/ (accessed August 4, 2023)

[15] Neo4j Desktop. https://neo4j.com/docs/desktop-manual/current/ (accessed March 18, 2023)

[16] Angular. https://angular.io/ (accessed July 21, 2023)

[17] TypeScript: JavaScript with Syntax for Types. https://www.typescriptlang.org/ (accessed July 22, 2023)

[18] What is Angular? https://angular.io/guide/what-is-angular (cccessed July 22, 2023)

[19] PHP: Hypertext Preprocessor. https://www.php.net/ (accessed July 21, 2023)

[20] Symfony, High Performance PHP Framework for Web Development. https://symfony.com/ (accessed July 21, 2023)

[21] Doctrine DBAL. https://www.doctrine-project.org/projects/doctrine-dbal/en/current/reference/introduction.html (accessed July 23, 2023)

[22] Neo4j PHP Client and Driver. https://github.com/neo4j-php/neo4j-php-client (accessed July 23, 2023)

[23] JavaScript Object Notation (JSON). https://www.json.org/ (accessed July 23, 2023)