Expertise
and insight
for the future

Hannu Junno

Recommendation engine with Neo4j graph database

Metropolia University of Applied Sciences

Bachelor of Engineering

Software Engineering

Bachelor's Thesis

8 January 2019

Author	Hannu Junno
Title	Recommendation engine with Neo4j graph database
Number of Pages	34 pages
Date	8 January 2019
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of Department

This thesis presents a proof of concept project for a graph database-based recommendation engine for a streaming service, trying to offer a viable alternative for an existing solution. The main idea of the new recommendation engine is to provide real time recommendations based on viewing time. This is done by processing the metadata of movies and series the user watched within certain timeframes, and then searching for movies and series with similar metadata to recommend.

The study covers development of the recommendation engine including graph data modeling, saving and querying data from the database, and the implementations of the recommendation logics. Theory of the existing solution and graph databases are also included.

The proof of concept project verifies that the solution is applicable and shows great potential. The testing was carried out with simulated test scenarios, and the results prove that no complex algorithms are needed for satisfying recommendations. A viable alternative for the existing solution was found.

Keywords	Neo4j, graph database, recommendation engine

Tekijä Otsikko Sivumäärä Aika	Hannu Junno Suosittelumoottori Neo4j-graafitietokannalla 34 sivua 8.1.2019
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmisotekniikka
Ohjaajat	Janne Salonen, yliopettaja

Opinnäytetyö esittelee suosittelumoottorin rakentamista suoratoistopalvelulle Neo4j-graafitietokannalla. Suosittelumoottorin pääideana on tarjota reaaliaikaisia suosituksia, joissa keskitytään katseluaikoihin. Suositukset toimisivat prosessoimalla elokuvien ja sarjojen metatietoja, joita käyttäjä katsoi tiettyinä ajanhetkinä, ja suosittelemalla vain elokuvia ja sarjoja vastaavilla metatiedoilla. Tavoitteena on todeta katseluaikoihin keskittyvän mallin toimivuus ja tarjota vaihtoehtoinen ratkaisu olemassa olevalle vanhalle suosittelumoottorille.

Opinnäytetyö kattaa koko suosittelumoottorin kehityksen sisältäen graafitiedon mallinnuksen, tiedon tallentamisen ja kyselyn tietokannasta sekä itse suosittelulogiikan toteuttamisen. Työ sisältää myös teoriaa olemassa olevasta suosittelumoottorista sekä graafitietokannoista yleisesti.

Projekti varmisti mallin käyttökelpoiseksi ja toimivaksi. Testaus toteutettiin testidatalla simuloimalla erilaisia käyttäjäskenaarioita. Tulokset osoittivat, että monimutkaisia algoritmeja ei tarvita tyydyttävien suosittelujen saamiseksi. Tavoitteeseen päästiin, ja vaihtoehtoinen ratkaisu löytyi olemassa olevalle vanhalle ratkaisulle.

Avainsanat	Neo4j, graafitietokanta, suosittelumoottori

Contents

List of Abbreviations

ASCII	American standard code for information change.
ACID	Atomicity, consistency, isolation, durability. The standardized transaction attributes often seen in relational databases and graph databases.
SQL	Structured query language. Standardized query language for relational databases.
CRUD	Create, retrieve, update, delete. The basic operations for a data store.
OGM	Object graph mapper.
JSON	JavaScript object notation.
API	Application programming interface.
IMDb	Internet movie database. The largest and most popular online database for movies and tv-series.

1 Introduction

The case company has a streaming service offering movies and tv-series for viewing by rental or a subscription-based model. The recommendation engine of the system could possibly be improved. The recommendation engine only processes data of what has been viewed by customers and based on that it calculates similarities of titles. Title is a generalized term that will be used for covering movies and single episodes of tv-series. One problem with the current solution is that each household has only one set of credentials that is used, and therefore the similarity calculations are made based on the whole household's views, rather than for each individual person, resulting in inaccurate recommendations. Whereas there are no plans to implement a feature for having multiple user profiles within one credential, this problem could be partly solved by utilizing more data that is available. By focusing on viewing times and recommending specific types of titles depending on when the user is watching could result in better recommendations. For example, if a household only watches cartoons every morning, only cartoons should be recommended in the mornings for that household. This is basically tracking the viewing habits of a household by digesting the metadata of titles watched within certain timeframes, and then recommending titles with similar metadata within the same timeframes. The timeframe could be narrowed down to hours on a certain day of the week. This should result in more personalized recommendations and partly solve the problem of multiple persons using the same credentials.

The existing solution is not capable for this sort of expansion and therefore, a totally new system needs to be created. As there are no guarantees that a new system focusing on viewing times will work, the project is labeled as a proof of concept. The target of this project is to build a system capable of adequate personalized recommendations proving concentration on viewing times a viable solution, and therefore offering an alternative recommendation system to replace the old existing one.

2 Theory

2.1 Current solution

2.1.1 Cosine similarity measure

The current solution uses cosine similarity measure to calculate similarities between titles. Cosine similarity is calculated by dividing the dot product of two vectors with multiplication of the vector's lengths. The result value ranges from -1 to 1, and the closer the value is to 1 the more similar they are. Formula (1) for calculating cosine similarity below.

$$similarity = \cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|} = \frac{\sum_{i=1}^{n} A_i \, B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}} \quad (1)$$

In this solution all titles are compared against each other resulting in a huge single relational database table. This means that each title has a similarity value calculated between every other title. The similarity value between two titles is calculated by having the vectors A and B represent users who rented the title, and the dot product being users who rented both titles, an intersection. In this scenario the result values range only from 0 to 1. The formula (2) and visualization of the intersection below in image 1.

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|} = \frac{\mathrm{C.size()}}{\sqrt{A.size()} \sqrt{B.size()}} \quad (2)$$

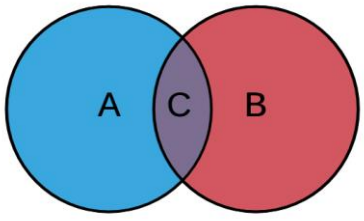

Image 1. Intersection of users who rented the titles A and B.

A simple example scenario: Title A has been rented by 15 users, title B by 20 users, and 10 of the users rented both titles. Calculating the similarity between these two titles results roughly a value of 0,58. Calculations are shown below.

$$\cos(\theta) = \frac{\mathrm{C.size()}}{\sqrt{A.size()}\sqrt{B.size()}} = \frac{10}{\sqrt{15}\sqrt{20}} \approx 0{,}58 \quad (3)$$

The cosine similarities between titles are periodically pre-calculated and stored in a single relational database table. When a user opens the service, the table is queried for similarity values for all the titles rented by the user, and the highest values are chosen for recommendations.

2.1.2 Problems with current solution

The current solution has a couple of problems. The first problem is that it only uses data of what titles have been watched by the users, and it is not flexible for expanding to use more data. Since every household only has one set of credentials, the calculations are made based on the whole household's views, rather than from individual persons. This large variety of data leads to inaccurate results. In addition, inaccurate results are also received from the cosine formula when comparing a title with only few rentals against a title with high number of rentals. For example, if in the example calculation above title B had been rented by 200 users instead of 20, the result value would be about 0,18 instead of 0,58.

The second problem is the scalability and performance for pre-calculations. The amount of comparisons required with the current solution is enormous, and it only keeps growing as number of titles increase. Currently with a little less than 8000 titles over 31 million comparisons are required.

In this use case and scenario, the cosine similarity measure is not the best solution, as it lacks accuracy for the reasons stated above. The performance and scalability are also a severe problem, and therefore an alternative solution is required.

2.2 New solution

2.2.1 What is a graph database?

Graphs build up from vertices and edges, which are often called as nodes and relationships. They allow to model data in an expressive structure making it easily understandable even in complex scenarios. The data model is extremely flexible making it suitable for modern agile development with constantly changing business requirements. [1, p. 8-10]

When planning a data model, it is common to first draw the entities on a whiteboard and see how the data links together. While in relational databases the whiteboard sketch would need to be restructured to fit a relational model, in graph databases the whiteboard sketch is exactly the wanted data model. Therefore, graph data models are considered whiteboard-friendly. [1, p. 26]

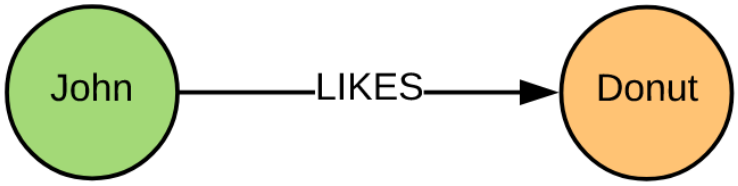

Image 2. Simple whiteboard sketch model example.

In the image 2 above, there are ‘John’ and ‘Donut’ as nodes, and ‘likes’ relationship between them. Nodes can be considered as relational database table rows, and relationships as joins.

Graph databases are designed for quick traversing between nodes. The relationships are considered as ‘first-class citizens’, meaning that they are treated equally important as the nodes [1, p. 6]. They may store key-value pairs, and they are pre-materialized into the database structures adjacent to the nodes. This makes traversing between nodes extremely fast, compared to the costly exponentially growing join operation of a relational database.

Graph databases are good for use cases that rely on data relationships. They are commonly used for social networks, recommendations, fraud detection and master data management amongst other things. A good rule of thumb is that if the relationships are as important as the nodes, a graph database should be considered. [2] [3]

2.2.2 Performance

Graph databases can be categorized as native and non-native, having two main features to differentiate these: processing and storage. Naturally the native is optimized for processing and storing graphs, whereas non-native might be for example a relational database-based solution converted to a graph data model. [4]

Native graph processing offers excellent performance for graph traversal regardless of database size. Global indexing is used to find the starting nodes, and index-free adjacency for quick traversing. Index-free adjacency means that each node has stored direct references to the nodes next to it. With this it is possible to acquire relatively constant speed traversing, because it is directly proportional on how big part of the graph is searched. Because of the excellent performance it is suitable for complex real time queries remaining scalable, whereas a relational database would severely struggle when joining dozens of tables together. Non-native graph processing uses global indexing for traversing, which is far slower and not as scalable. Traversing is slowed down because an extra layer needs to be added on each traversal. It also constantly keeps slowing down the queries as more data is added to the database. [4]

Native graph storages are specifically designed for storing and managing graphs, whereas non-native means that the storage is provided elsewhere. It could be provided from a relational or NoSQL database, and therefore the algorithms being not optimized for storing graphs, resulting in a lack of efficiency. [4]

2.3 Neo4j

2.3.1 Overview

Neo4j is an open-source native graph database. It was released as open-source in 2007 by a Swedish based company, Neo4j incorporated [5]. It was the first graph database with native storage and processing. The source code is written in Scala and Java, and it is available in GitHub [7]. [6]

2.3.2 Property graph model

Neo4j uses property graph model, which stores data into two different types; nodes and relationships. They both may contain unlimited amount of properties, which are basic key-value pairs. It is not a requirement for a node or a relationship to contain properties, but they are often found at least on the nodes. Labels are used for grouping the nodes, which are basically the equivalent of table names in relational databases. If a relationship contains properties, it is called a rich relationship. A simple example of a graph model can be seen in image 3 below. [1, p. 26-27] [7]

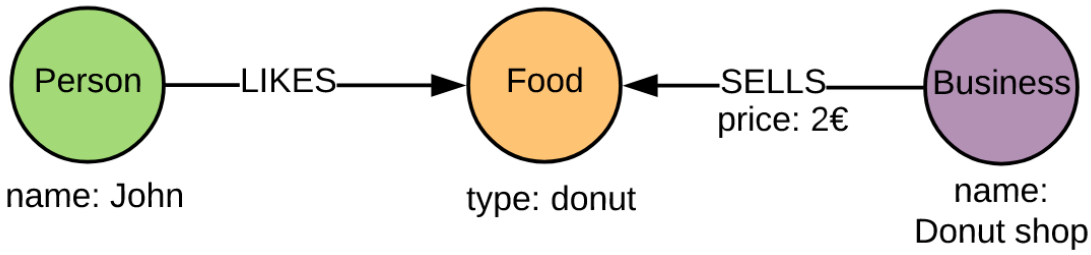

Image 3. Property graph model example

In image 3 above starting from the left there is a node labeled person with a property ‘name: John’, and in the middle there is another node labeled food with a property ‘type: donut’. From John to donut there is a relationship ‘likes’, so John likes donuts. On the right there is a third node labeled business with a property ‘name: Donut shop’. From Donut shop to donut there is a relationship SELLS with a property ‘price: 2€’, meaning that Donut shop sells donuts for 2 euros.

2.3.3 Cypher

Neo4j uses graph query language Cypher, which was invented by the same company, Neo4j incorporated. In 2015 they decided to open source the Cypher language aiming it to become the “SQL for graphs”. Cypher aims to be as human-readable as possible. [8]

Creating nodes and relationships in Cypher is done with the ‘create’ clause. The nodes are wrapped in parentheses and relationships in square brackets. First inside the wrappings a variable may be assigned for the node or the relationship followed with a colon and the node label or relationship type. Assigning a variable is optional and the variable is only accessible within the statement to be executed. Properties may be given inside curly braces. Relationship directions are indicated with ASCII style arrows. Multiple nodes or patterns may be given for clauses by separating them with commas. Below in listing 1 an example how to create nodes and relationships in Cypher. [9]

```
CREATE (john:Person { name: 'John' }),
       (donut:Food { name: 'donut' }),
       (shop:Business { name: 'Donut shop' }),
       (john)-[:LIKES]->(donut),
       (shop)-[:SELLS { price: 2 }]->(donut)
```

Listing 1. A Cypher statement that creates nodes for John, donut and Donut shop, and relationships between John and donut, and Donut shop and donut.

Searching for patterns in Cypher is done with the ‘match’ clause. The match clause requires a pattern to search for and optionally some restrictions or predicates. In the given pattern it is not needed to specify node labels or relationship types, but it may be done to optimize the query. Without a colon in the node or the relationship wrapping it is interpreted as a variable and it will match everything. A commonly used predicate when searching patterns is ‘where’. For example, it may be used to only match nodes with a specific property value. Node properties can be accessed with a dot following with property key. Below in listing 2 a simple query as an example. [10]

```
MATCH (n)-[:LIKES]->(food:Food)<-[sells:SELLS]-(business:Business)
WHERE food.name = 'donut'
RETURN n, business, sells.price
```

Listing 2. A Cypher query that returns everything that likes donuts, businesses that sell donuts, and the prices they are sold for.

Note that node labels, relationship types, property names and assigned variables are case sensitive. Cypher clauses are not case sensitive but should be cased as seen above in Listing 1 and 2.

2.3.4 Data integrity and clustering

Data integrity is crucial in graph databases because everything is linked together. Having a single faulty node or a missing relationship corrupts the whole database. For data integrity and safety there are two essential things: appropriate transactions and clustering.

Neo4j provides ACID compliant transactions. ACID is an acronym for atomicity, consistency, isolation and durability. These four are the standardized transaction attributes often seen in relational databases and graph databases. [11] [12] [13]

- Atomicity ensures that a single transaction is executed as whole or not at all. Transactions usually consist of multiple operations, and if any of the operations fail, everything is rolled back to the original state.
- Consistency guarantees that transactions keep the database in a valid state. If a transaction violates any rules of the database, it is reverted.
- Isolation ensures that simultaneous transactions are handled independently and do not affect others. If multiple transactions concurrently modify the same data, the result is the same as if they were processed sequentially.
- Durability guarantees that a successful transaction is permanently stored in the database and it is available even after a failure.

For fault-tolerant systems clustering is essential. Neo4j Enterprise Edition offers causal clustering that provides safety, scalability and causal consistency. A cluster builds up from core servers and read replicas, offering fault tolerance and load balancing. Visualization of a cluster is described in image 4 below.

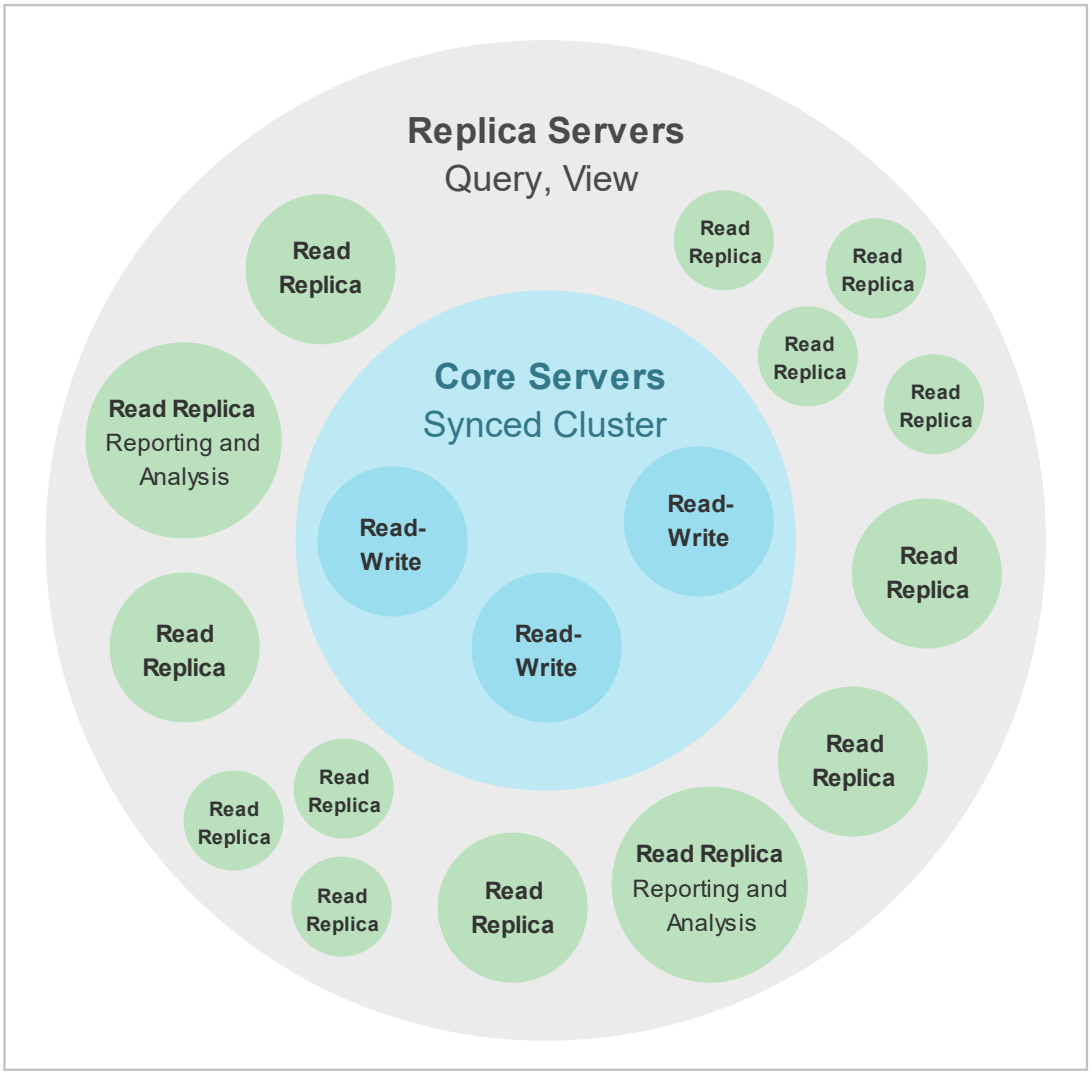

Image 4. Cluster with core servers and read replicas. [15]

Core servers are responsible for ensuring that the data is safe in the cluster. The transactions are replicated throughout the core servers using the raft consensus algorithm. Once a transaction is accepted on majority of the servers, thus a consensus has been reached, the transaction is committed as successful. The majority of the core servers in the cluster need to be running and available in order to do write operations to the database. Therefore, the total number of core servers determine the fault tolerance of the cluster. Having three core servers means that one can go down and the cluster remains functioning, or with five core servers there is fault tolerance for two servers, and so on. If majority of the core server cluster is not available, it will become a read-only to maintain integrity. [14] [15]

The main purpose of read replicas is for scaling the workload for read only graph queries. They are replicated from Core servers and are fully capable Neo4j databases being only restricted to process read-only queries. Read replicas constantly poll the Core server's transaction logs, and if any new writes have been made to the database, they are transferred to the replica. Replicas are considered as disposable and losing one will only affect the query processing times, depending on the total amount of replicas. [15]

Causal consistency ensures that even though the database is distributed, it still acts like a single database for clients interacting with it. When executing a transaction, a bookmark of the transaction may be provided for the client. The bookmark can be passed on as a parameter for a subsequent transaction to ensure that the server has processed the previous transaction before processing the following one. This is called causal chaining. [15]

3 Project work

3.1 Planning graph data model

The first step of starting this project is to plan the graph data model. It will most likely have some changes when the project progresses, but an initial model is needed. Graph data models can be constructed from simple scenarios. An example scenario in the case company: “Bob and Sandy watched the episode Blackwater that belongs to the series Game of Thrones.” The nodes may be identified first by searching for all the proper nouns in the sentence, which are: Bob, Sandy, Blackwater and Game of Thrones. After identifying the nodes relationships can be defined by searching for verbs in the sentence, which are: watched and belongs to. These allow to establish relationships as follows: [16]

- Bob watched Blackwater.
- Sandy watched Blackwater.
- Blackwater episode belongs to Game of Thrones series.

Now that the nodes and relationships are defined the nodes should be labeled. This may be done by searching for common nouns in the sentence, which are: episode and series. For Bob and Sandy no common nouns were found, but it can be easily figured out to categorize them as users. The only thing missing from the graph model are the properties. For each node there already is a property ‘name’, and other properties may be added as seen useful. For the watched relationship at least viewing time is required, and for the episode node a property for total runtime is required to be able to calculate how big portion of an episode a user watched. A simple graph model of the scenario is now done and seen below in image 5. [16]

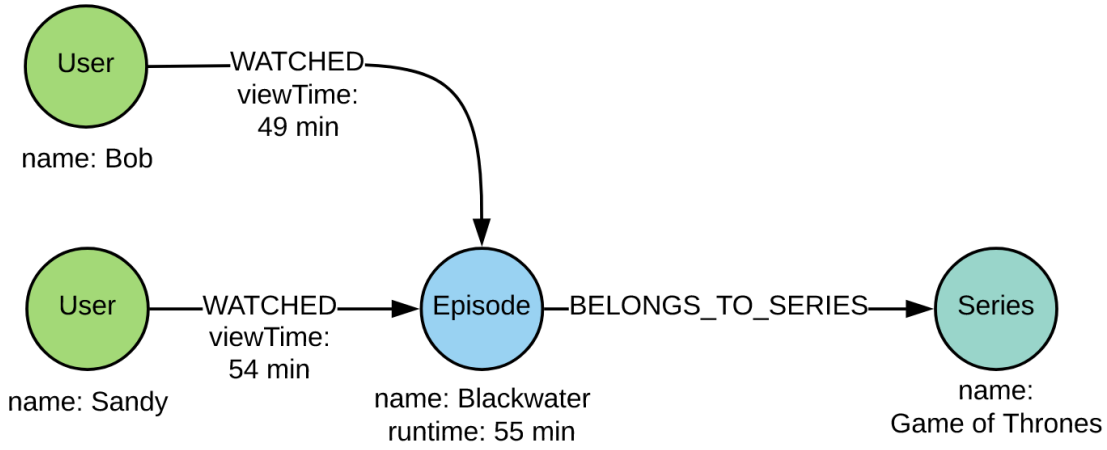

Image 5. Graph model constructed from a simple example scenario in the case company.

The same design process and guidelines allows to create a complete graph data model with the rest of the data. Some minor tweaking is required for the previous model to make it more suitable with the rest of the data. Complete graph data model is illustrated in image 6 below.

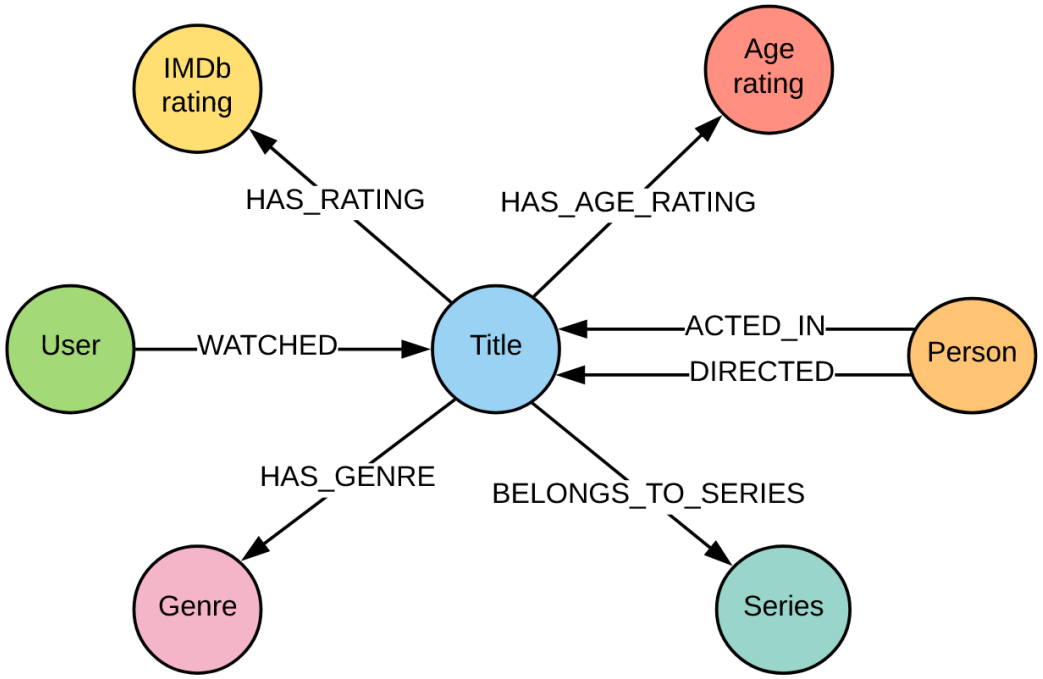

Image 6. Property graph data model for the project.

Title is a generalized term that covers movies and single episodes of series. Not all properties need to be used, so the episode and season numbers may be stored in the title node only when dealing with episodes. IMDb rating refers to the Internet Movie Database ratings.

3.2 Architecture and technology stack

The plan is to have two separate API's: one for feeding data and one for handling the graph database and recommendations. This way the logics are neatly separated, and a layer of security is added, as the database is not directly accessible from the feeder. As both API's will be using the same data models, the models are therefore packaged separately and distributed from an artifactory.

The feeder API is responsible of fetching and receiving data, parsing the data, and sending it forward. Having multiple different data sources requires implementation for both fetching data with get requests and receiving it via post requests. The data is then parsed to the Neo4j domain models and sent via post requests to the recommendation API. The recommendation API is responsible for receiving and saving data, querying data, and handling all the recommendation logic. The API has endpoints to receive post requests that include the data in Neo4j models, which are then saved to the database. For the recommendations the API has a single endpoint for get requests taking user ID as a parameter. The API takes care of querying the data and all the recommendation logics returning only a list of movie IDs to recommend. Architecture layout illustrated in image 7 below.

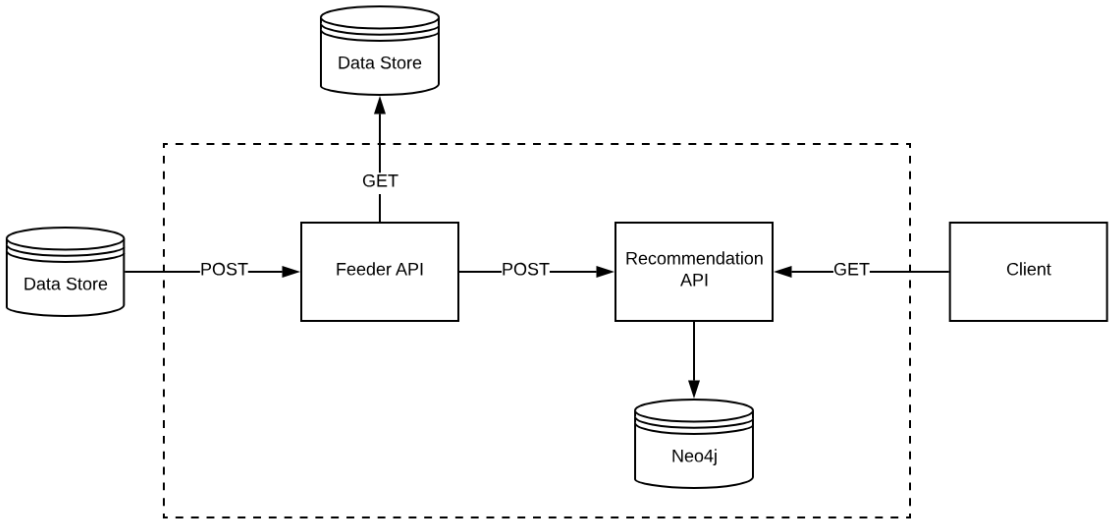

Image 7. Architecture layout.

As this project is initially in a proof of concept stage, the goal is to get the system up and running as fast as possible for testing and see if the results are as desired. Separating the logics would hinder the agility and as so postpone the testing whether this solution is applicable to begin with. Therefore, the feeder API, the recommendation API, and the Neo4j domain models will initially be implemented in the same package to speed up development, as more or less everything is prone to change in the early stages.

Now that the graph data model and the architecture layout is defined, it is time to define the technologies used for this project. The technology choices have been deliberately postponed to the very last possible moment, as is advised in the general guidelines of Clean Code. It is good to keep planning as long as possible without committing to any technology choices, as committing too soon without optimal knowledge might hinder fulfilling the upcoming task to its fullest potential. [17, p. 168]

First requirement for the graph database is excellent performance, so it needs to be native. Secondly, long term support is required. Case company's head architect is familiar with Neo4j, which makes it a potential candidate. The property graph model of Neo4j seems to be a good fit for the project. Long term support is not guaranteed, but it looks promising that the development of Neo4j will continue for a long time, as the company

only keeps growing. It is also the most popular native graph database and has an active community for support. With these considerations Neo4j is chosen to fulfill the task. [18]

For running Neo4j database Docker seems like a good choice, which is officially supported by Neo4j, and therefore official docker images are provided. A docker image includes all the necessary dependencies for an application, which ensures that it will work the same way regardless of environment. The image is used for creating a container that stores configurations and is used for running the application itself. Running an application in a container guarantees isolation and makes disposal and recreation easy. [19] [20]

For the back-end side Java with Spring Boot framework is a clear choice for a few reasons. Firstly, being familiar with it and it being used elsewhere in the same environment. Secondly, Spring Data project provides direct support for Neo4j, and finally, it has a large community for support and comprehensive documentation. Spring Data is a project for providing unified and easy data access for different data stores, having sub-projects for specific databases, including Neo4j. Spring Data Neo4j is based on the Object Graph Mapping (OGM) library provided by Neo4j. The library is for simple management of domain objects, mapping plain old java objects to Neo4j. [21] [22]

The chosen technology stack being:

- Neo4j version 3.5
- Java version 8
- Spring Boot 2.1.0
- Spring Data Neo4j version 5.1.0
- Docker version 18.03

3.3 Initial setup

Setting up Neo4j database can be done easily with Docker Compose tool, which is basically a wrapper for Docker. For a minimalistic setup it is needed to define the base image version, ports that the database is listening to, and the credentials for the database. Running ‘docker-compose up’ from command line will fetch the base image from

docker storage, build a docker image with the set configurations, create a container of the image, and run it. Re-running the command using the same configurations will run the existing container, as the configurations are cached. Below in listing 3 docker compose configurations are presented. [23]

```
version: '3'

services:
     neo4j:
          image: neo4j:3.5
          ports:
           -  "7474:7474"
           -  "7687:7687"
          environment:
           -  NEO4J_AUTH=user123/pass123
          volumes:
           -  ./neo4j/:/data
           -  ./neo4j/:/logs
```

Listing 3. Docker Compose file configurations.

In listing 3 above the Compose file format version is set to 3, and Neo4j base image version is set to 3.4. Default port number in Neo4j for a HTTP connector is 7474 and for a Bolt connector 7687. The HTTP connector will be used for accessing the inbuilt Neo4j Browser and the Bolt connector for database access. The data and logs are mapped to be stored locally on the computer, rather than inside the disposable container. In the early stages of the project the container will be run locally on computer. Later in the development phase the image will be deployed to OpenShift Container Platform. [24]

The Spring Boot application can initially be set up with only two maven artifacts that compile all the dependencies needed. These dependencies allow to configure the connection to the Neo4j database in the application properties file. Code samples of these in listing 4 below.

```
pom.xml file:

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.1.0.RELEASE</version>
</dependency>

<dependency>
     <groupId>org.spring.framework.boot</groupId>
     <artifactId>spring-boot-starter-data-neo4j</artifactId>
     <version>2.1.0.RELEASE</version>
</dependency>
```

```
application.properties file:

spring.data.neo4j.uri=bolt://localhost
spring.data.neo4j.username=user123
spring.data.neo4j.password=pass123
```

Listing 4. Maven dependencies and application properties configurations.

Finally, the application's main class needs to have @EnableNeo4jRepositories annotation to enable scanning for repositories in the package. The application and database are now set up to start the project development.

3.4 Creating data models

Spring Data allows to create plain old java objects with annotations that are directly usable for saving and retrieving data from Neo4j database. Below in listing 5 code examples of title node and watched relationship classes according to the projects graph data model. Getters, setters and empty constructors are required in classes by the OGM, but in code example they are omitted for the sake of brevity.

```
@NodeEntity
public class Title {

      @Id
      @GeneratedValue
      private Long id;

      @Index(unique = true)
      private Long titleId;

      private String name;
      private Integer episodeNumber;
      private Integer seasonNumber;

      @Relationship(type = "HAS_GENRE")
      private List<Genre> genres;

      @JsonIgnoreProperties("title")
      @Relationship(type = "HAS_RATING")
      private HasRating hasRating;

      @Relationship(type = "HAS_AGE_RATING")
      private AgeRating ageRating;

      @Relationship(type = "BELONGS_TO_SERIES")
      private Series series;

}
```

```
@RelationshipEntity
public class Watched {

      @Id
      @GeneratedValue
      private Long id;

      private LocalDateTime startTime;
      private LocalDateTime endTime;
      private Long viewTimeInSeconds;

      @StartNode
      private User user;

      @EndNode
      private Title title;

}
```

Listing 5. Java classes for title node and watched relationship.

On class level there are two different annotations to define whether the class is a node or a rich relationship: @NodeEntity and @RelationshipEntity. The class name is used as the node label or relationship type by default. Every entity saved to Neo4j database requires an internal ID, being marked with @Id annotation. @GeneratedValue annotation let's Neo4j's OGM take care of generating the ID when creating a new entity in the database. It is not advisable to use the internal ID for anything else or to rely on it, as it might change when the database is modified.

Plain relationships without properties do not need their own class, hence they need to be defined by using the @Relationship annotation on a field referencing to the other node entity. By default, the relationship type is the annotated fields name and direction is outgoing. Rich relationships with properties require their own class with @RelationshipEntity annotation and both start, and end nodes defined with @StartNode and @EndNode annotations. In addition, one of the nodes needs to have a reference to the relationship, as relationships always need to be accessible from nodes. This is because writing the graph path starts from nodes and follows with creating relationships between them.

Rich relationships end up having two entities referencing to each other, causing an infinite recursion. Neo4j's OGM can handle this, but Jackson serializer provided by Spring Boot cannot. When serializing from a Java object to JSON format it loops infinitely between the references and causes a stack overflow. One way to avoid this is by using

@JsonIgnoreProperties annotation to tell Jackson to ignore certain properties when serializing.

Indexes are created with @Index annotation and unique constraints with the same annotation and by setting the unique value to true. In addition, the @Id annotation also creates a unique constraint. There are several ways to configure index creations, and one easy way to configure it is in the application properties file by defining the auto-index property. By setting the auto-index value to ‘assert’, all indexes and constraints are removed on application startup and recreated according to the current metadata. Note that this is only intended to be used in development environment. For production environment auto-index property value can be set to ‘validate’, which verifies on startup that all indexes and constraints exists accordingly.

3.5 Saving data

The Spring Data provides Neo4jRepository interface that can be extended and used for saving data to the database. An implementation of the interface can be created and injected with Spring Boots @Autowired annotation. Below in listing 6 a simple example how to save data.

```
public interface TitleRepository extends Neo4jRepository<Title, Long> {}

public class TitleSavingService {

      private final TitleRepository titleRepository;

      @Autowired
      public TitleSavingService(TitleRepository titleRepository) {
            this.titleRepository = titleRepository;
      }

      public void createAndSaveTitle() {
            Title title = new Title();
            title.setName(“Blackwater”);
            titleRepository.save(title);
      }

}
```

Listing 6. Saving data to database using interface provided by Spring Data.

The save method in listing 6 above creates a cypher statement with create clause handing the title entity as a parameter and generating an ID for it. If the save method is run again it creates a new node with the same name but a new ID. If a unique constraint were set on the name it would throw an error when trying to create the second node and cancel the transaction. To update an existing node the ID field needs to be set. This changes the create clause to match, and thus only tries to find an existing node with given id to set the new properties. To access the id of an existing node the node needs to be searched first with a known property, which would be the name in this scenario.

The save method defined by Neo4jRepository interface actually takes 2 arguments: entity and depth. The implementation class created by Spring Boot allows to only provide the entity for the save method, which is then overloaded with depth set to -1. It means that everything reachable from the given entity is saved to the graph. Below in listing 7 a simple code example of this scenario.

```
Series series = new Series();
series.setName("Game of Thrones");

Title title = new Title();
title.setName("Blackwater");
title.setSeries(series);

titleRepository.save(title);
```

Listing 7. Saving data with infinite depth example.

On the code example in listing 7 above, three create statements are called: a series node is created, a title node is created, and a relationship between these two nodes is created. If the series was saved instead of the title, only a single series node would be created. This is because the series object does not have a reference to the title object, but the title has a reference to the series. The reference could be stored the other way around, and that is just a matter of design. Storing it on both nodes would be redundant, as the graph can traverse relationships in both directions.

The saved data can be viewed with Neo4j Browser provided with the Neo4j database. Below in image 8 a screenshot of Neo4j Browser view.

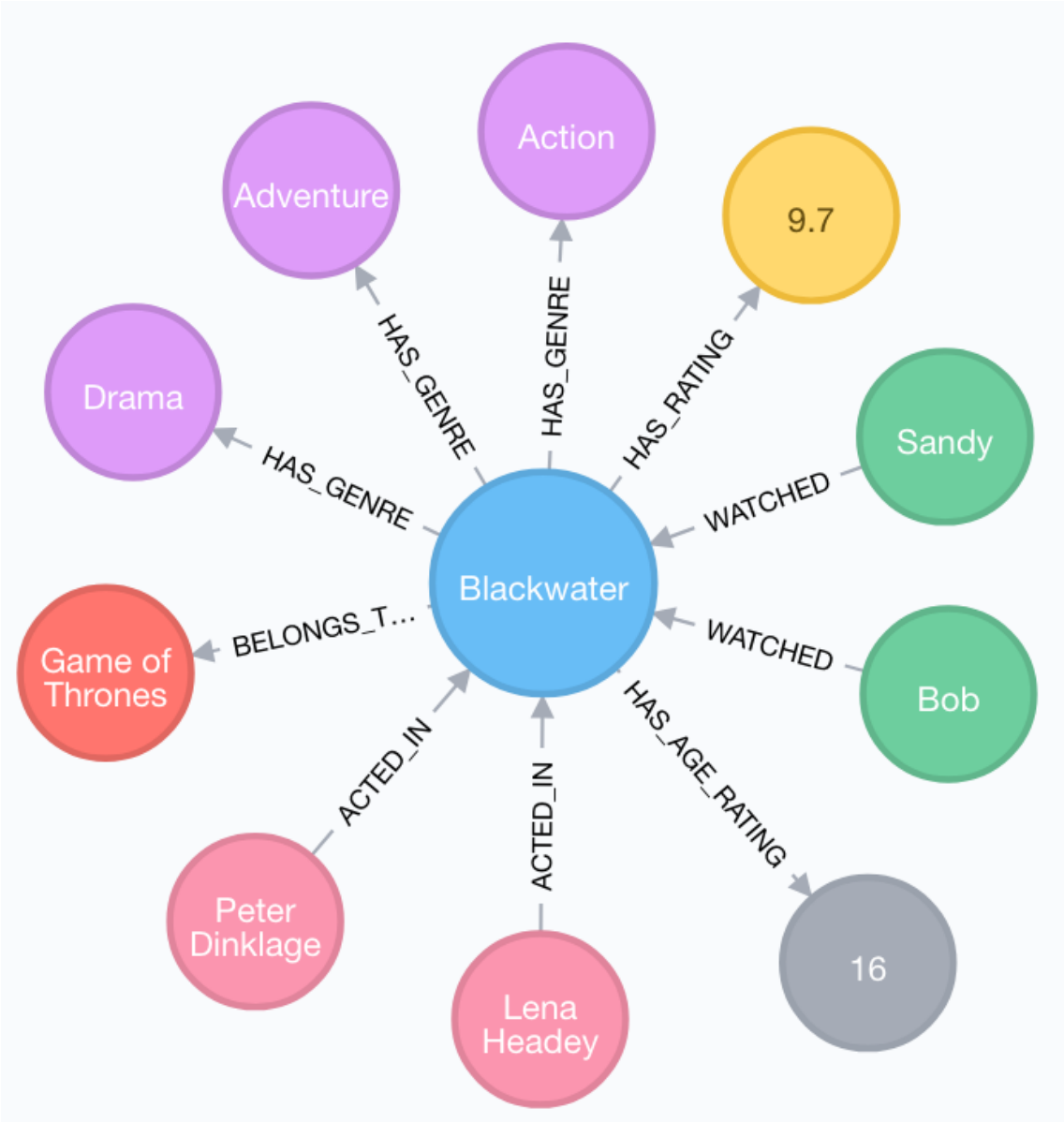

Image 8. Screenshot of Neo4j Browser view.

The Neo4j Browser included with the database is an independent client that may be used for viewing a database. The client is accessed through the HTTP connector, but Bolt connector needs to be used for connecting to a database.

3.6 Accessing data

Using layered architecture there are three different layers to go through for accessing the database from outside the application. The layers are controller, service and repository, each layer being able to access only the one below. Visualization of the layers in image 9 below.

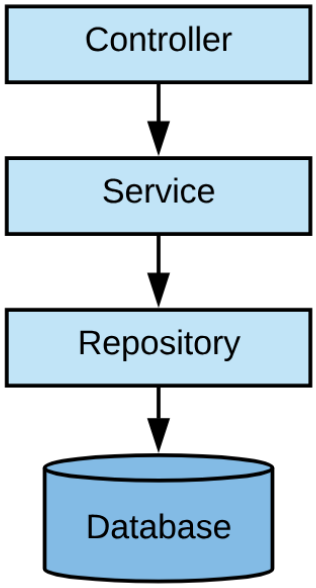

Image 9. Layered architecture.

Controller layer is the access point to the application, being the only thing exposed outside. It is only able to call methods from the service layer. Service layer is responsible for all the business logics and is only able to call methods from the repository layer. Repository layer is solely for handling communication with the database. Below in listing 8 example codes of each layer.

```
@Controller
public class TitleController {

      private final TitleService titleService;

      @Autowired
      public TitleController(TitleService titleService) {
            this.titleService = titleService;
      }

      @GetMapping("/title")
      public Title findByTitleId(@RequestParam Long titleId) {
            return titleService.findByTitleId(titleId);
      }

}

@Service
public class TitleService {

      private final TitleRepository titleRepository;

      @Autowired
      public TitleService(TitleRepository titleRepository) {
            this.titleRepository = titleRepository;
      }

      @Transactional(readOnly = true)
      public Title findByTitleId(Long titleId) {
            return titleRepository.findByTitleId(titleId);
      }

}

public interface TitleRepository extends Neo4jRepository<Title, Long> {

      Title findByTitleId(@Param("titleId") Long titleId);

      @Query("MATCH (title:Title) " +
             "WHERE title.titleId = $titleId " +
             "RETURN title")
      Title customQueryToFindByTitleId(@Param("titleId") Long titleId);

}
```

Listing 8. Code examples of controller, service, and repository layers.

@Transactional annotation is provided by Spring Data and is used to mark the method as a single transaction. Furthermore, the transaction may be specified as read only, which allows Neo4j to allocate the request for read only replicas.

Neo4jRepository interface extends CrudRepository interface, having the basic CRUD operations inbuilt: Create, Retrieve, Update and Delete. For writing custom Cypher queries @Query annotation needs to be used. Example of using an inbuilt find operation and

a custom query resulting in similar Cypher statement seen in listing 8 in the TitleRepository interface.

3.7 Recommendation logic

The main recommendation logic constructs of two components: fetching metadata of titles user watched, and finding titles based on the metadata. The idea is to execute these queries in real time when a user enters the service and immediately recommend the titles found.

The query for fetching metadata of titles user watched takes day(s) of week and a scope of view starting time as parameters. This allows to get precise data on what kind of titles user watches on certain days of week and within a given time period. For example, if a user enters the service on Monday at 17:00, the query could fetch data of everything the user started watching on Monday between 16:00-18:00. In addition, it also takes a date as parameter for excluding older views. This allows to consider only for example views from the last year, as people might have tendency to change their viewing habits. Query for finding the titles and the metadata is demonstrated below in listing 9.

```
@Query(
    "MATCH (user:User)-[watched:WATCHED]->(title:Title) " +
    "WHERE user.userId = $userId " +
    "WITH title, localdatetime(watched.startTime) AS date " +
    "WITH title, localtime(date.hour + ':' + date.minute) AS startTime " +
    "WHERE date > localdatetime($startingFromDate) " +
    "AND date.dayOfWeek IN $daysOfWeek " +
    "AND startTime >= localtime($fromStartTime) " +
    "AND startTime <= localtime($toStartTime) " +
    "RETURN title, " +
    "[ [ (title)-[r_h1:`HAS_RATING`]->(i1:`ImdbRating`) | [ r_h1, i1 ] ], " +
    "[ (title)-[r_b1:`BELONGS_TO_SERIES`]->(s1:`Series`) | [ r_b1, s1 ] ], " +
    "[ (title)-[r_h1:`HAS_GENRE`]->(g1:`Genre`) | [ r_h1, g1 ] ], " +
    "[ (title)-[r_h1:`HAS_AGE_RATING`]->(a1:`AgeRating`) | [ r_h1, a1 ] ] ] ")
Collection<Title> findTitlesUserViewedOnDayAndBetweenTime(
@Param("userId") Long userId,
@Param("startingFromDate") String startingFromDate,
@Param("daysOfWeek") List<Integer> daysOfWeek,
@Param("fromStartTime") String fromStartTime,
@Param("toStartTime") String toStartTime);
```

Listing 9. Query for finding data of titles user has watched.

Dates are parsed from given string parameters and compared against the node properties in Cypher as seen in listing 9 above. The final part of the return statement is used for returning the relationships and nodes the titles have references to. Separate queries are implemented to get data of actors and directors related to the titles. It is simply done by collecting all the title ID's and doing queries to find all the persons appeared in the titles. In addition, appearance counts can be returned directly from the queries. Listing 10 illustrates a query for finding actors and the custom query result class.

```
public interface PersonRepository extends Neo4jRepository<Person, Long> {

      @Query(
            "MATCH (person:Person)-[:ACTED_IN]->(title:Title) " +
            "WHERE title.titleId IN $titleIdList " +
            "RETURN person, count(person) as count " +
            "ORDER BY count DESC")
      Collection<PersonCountQueryResult> findActorsWithCount(
      @Param("vodIdList") List<Long> vodIdList);

}

@QueryResult
public class PersonCountQueryResult {

      private Person person;
      private int count;

}
```

Listing 10. Query for finding actors appeared in titles user watched and the class of the object the result is mapped to.

From the data it is important to exclude titles user did not completely watch. For example, if a user watched only 10% of a title, it is relatively safe to assume that the user did not like it, and the data should not be considered. The view percentage calculation could be done in listing 9 Cypher query above, but to keep the query clean and easily readable it is implemented separately in the application. For a starting point the percentage threshold of excluding titles not watched completely will be set to 80%, and then adjusted accordingly in testing phase.

There are numerous ways of utilizing the data and executing particularly specific queries with it. Data experimenting will start by calculating distribution percentages for different data categories and by having percentage thresholds for determining to apply the data or not. In some data categories average values are calculated and applied with certain

tolerances to the queries. A query with most of the data categories applied as parameters is presented below in listing 11.

```
@Query(
      "MATCH (user:User)-[watched:WATCHED]->(title:Title), " +
            "(title)-[rated:HAS_RATING]->(imdbRating:ImdbRating), " +
            "(title)-[:HAS_GENRE]->(genre:Genre), " +
            "(title)-[:HAS_AGE_RATING]->(ageRating:AgeRating) " +
      "WHERE NOT user.userId = $userId " +
      "AND title.runTimeInMs >= $minRuntime " +
      "AND title.runTimeInMs <= $maxRuntime " +
      "AND title.releaseYear >= $minReleaseYear " +
      "AND title.releaseYear <= $maxReleaseYear " +
      "AND rated.count >= $minImdbRatingCount " +
      "AND imdbRating.ratingValue >= $minImdbRatingValue " +
      "AND imdbRating.ratingValue <= $maxImdbRatingValue " +
      "AND genre.name IN $genreNames " +
      "AND ageRating.value IN $ageRatings " +
      "RETURN title")
Collection<Title> findTitles(
@Param("userId") Long userId,
@Param("minRuntime") Long minRuntime,
@Param("maxRuntime") Long maxRuntime,
@Param("minReleaseYear") Integer minReleaseYear,
@Param("maxReleaseYear") Integer maxReleaseYear,
@Param("minImdbRatingCount") Integer minImdbRatingCount,
@Param("minImdbRatingValue") Double minImdbRatingValue,
@Param("maxImdbRatingValue") Double maxImdbRatingValue,
@Param("genreNames") List<String> genreNames,
@Param("ageRatings") List<String> ageRatings);
```

Listing 11. Query for finding titles with parameters for most of the data categories and excluding the user recommendations are for.

Age ratings will have percentage distributions calculated and a relatively low percentage threshold, such as 20%, for adding it to the query. Another solution could be to check the highest age rating exceeding the percentage threshold and unlock all age ratings below it.

Genres will simply be counted and have percentage distributions calculated. Exceeding the percentage threshold will add the genre to the list given as parameter for the query. The threshold might need to be a little higher for this, such as 30%, because the user might have tendency to watch genres of certain moods during a specific time and day of week.

IMDb rating values will be rounded by one, counted, and have percentage distributions calculated. An average is calculated from values exceeding the percentage threshold, and the minimum and maximum IMDb rating values are defined by summing and

subtracting given tolerances from the average value. For example, having IMDb rating average value of 7,5 and a tolerance of ±1 leads to a query with minimum of 6.5 and maximum of 8.5 IMDb rating value. The minimum and maximum tolerances will be defined separately though, as ignoring the lower rating values is more crucial than ignoring the higher.

Runtimes will be divided into different scopes. For each scope is counted how many titles fit in it and then have the percentage distributions calculated. Every scope that exceeds the percentage threshold will create an individual query with the scope's minimum and maximum runtimes. In tv-series episodes are often roughly 20 minutes or 45 minutes long, so at least those two need to be differentiated. For movies there is no need for very strict divisions. Initially they will be divided into two scopes: shorter movies less than 95 minutes, and everything longer than that. Initial runtime scopes listed below.

- 0-15 min
- 16-35 min
- 36-55 min
- 56-75 min
- 76-95 min
- 96-300 min

Release years will be divided into decades, counted, and have percentage distributions calculated. The lowest and highest decades exceeding the percentage thresholds will define the minimum and maximum release years, with given tolerances.

Actor and director appearances will initially just be counted and exceeding the count threshold will add separate queries for them. By setting the appearance threshold relatively low gives a high probability for finding movies in the same movie series a user is watching. It also returns other titles the persons have appeared in, as the user might be interested in those as well.

One notable thing with the listing 11 query above is if nobody has watched the title it is not found with the query. However, new titles do have separate promotions, so this should not be an issue, and they will be found as soon as one person has watched it. Days of week and starting time scopes could also be added for the query similarly as for the user statistic query, but that requires further testing is it beneficial. For finding titles

with specific actors or directors there are similar queries as the one above with just added parameters. If the query returns an episode of a series there is additional logic implemented in the application so that only one entity of the series is recommended, which is either the first or the latest episode user has not seen.

Besides finding new movies or series to recommend, there is also separate logic implemented for suggesting the next episodes in series user watched. This is a very common feature in similar services providing easy access to following episodes. The logic for these suggestions is as follow: First all the series user has watched are searched. For each series all episodes the user has seen are searched, and by sorting them descending with season and episode number, the first match will be the latest seen episode. If user watched less than 90% of the latest seen episode it is suggested, otherwise next episode is searched and suggested if found. Listing 12 illustrates the Cypher queries for this implementation.

```
@Query(
   "MATCH (u:User)-[:WATCHED]->(:Title)-[:BELONGS_TO_SERIES]->(s:Series) " +
   "WHERE u.userId = $userId " +
   "RETURN s")
Collection<Series> findAllSeriesUserWatched(@Param("userId") Long userId);


@Query(
   "MATCH (u:User)-[w:WATCHED]->(t:Title)-[:BELONGS_TO_SERIES]->(s:Series) " +
   "WHERE u.userId = $userId " +
   "AND s.name = $seriesName " +
   "RETURN u AS user, t AS title, w AS watched, s AS series " +
   "ORDER by t.seasonNumber DESC, t.episodeNumber DESC " +
   "LIMIT 1")
UserWatchedTitleData findLatestWatchedEpisodeInSeries(
@Param("userId") Long userId,
@Param("seriesName") String seriesName);


@Query(
   "MATCH (t:Title)-[:BELONGS_TO_SERIES]->(s:Series) " +
   "WHERE s.name = $seriesName " +
   "AND t.seasonNumber = $seasonNumber " +
   "AND t.episodeNumber = $episodeNumber " +
   "RETURN t")
Title findEpisodeInSeason(
@Param("seriesName") String seriesName,
@Param("seasonNumber") Integer seasonNumber,
@Param("episodeNumber") Integer episodeNumber);
```

Listing 12. Cypher queries for suggesting next episodes in series user watched.

If a user starts watching the same series from the start, this implementation does not quite work as it is. One simple solution could be to search the latest seen episode only from the last month for example. A more complex problem is how to implement suggestions for multiple persons using the same device to watch the same series at a different pace. However, this is not a common scenario, but the issue was brought up by the user experience team in the case company. This could potentially be solved for at least two persons by checking if there are subsequent duplicate watches of the same episodes and by subtracting the duplicates the lowest season and episode number left would be the other persons bookmark.

3.8 Testing

The testing for this project was conducted with real title data and test user data. The advantage of test user data is that it allows to accurately simulate different user scenarios, allowing to easily verify that the recommendation logic works. Real user data being much more scattered and therefore harder to interpret without comprehensive metrics of old and new system, which were not available for the timeframe of this project.

The testing showed promising results, as were expected. Initial thresholds defined in the previous chapter were quite close for obtaining satisfying results, but some of them needed to be narrowed down a little more to shrink down the amount of results. However, the thresholds and tolerances need more tuning once testing with real user data, as real user's behavior is probably not as consistent as simulated with the testing data. But a baseline is now set, which can be used for reference when adjusting to real user data.

One problem noticed during testing is the source data integrity. For example, some of the titles were missing IMDb ratings and therefore being ignored by the query for finding titles that expects the rating to exist. Another example is having an actor's name misspelled in a title's metadata, affecting the count of actor appearances. All persons do have a unique ID for identifying them, but in some cases, it is missing and therefore the name is used as a secondary identifying method.

Performance testing for the graph database was conducted with a 2015 model of MacBook Pro with Intel Core i7 2,5GHz processor and 16GB of memory. Saving data

seemed to get significantly slower as the database size grew, which was most likely because of the high number of indexes, that was 18 during the testing. A little less than 8000 titles including metadata resulted in roughly 54 000 nodes and 109 000 relationships, which took approximately two and half hours to save to the database. Note that this also included the time for fetching and processing the data in the application before it could be saved. The query execution times were fast, as promised by Neo4j. A single query for finding titles as seen in Listing 11 took about 60 milliseconds to execute.

4 Summary

4.1 Results

The results of this study show that the performance of the graph database is good, but optimization is needed for the current implementation. For faster saving capabilities the number of indexes should be reduced and used only for those that are often targeted by the queries. Having a high number of indexes is a trade-off for faster queries but slower saving. However, it is not often required to wipe out and resave the entire title database, so query performance should be the priority. To maintain good performance for real time recommendations the number of queries should be kept as low as possible. With the current implementation there are some redundant separate queries that should be combined for better performance. Overall each query and piece of logic in the application should be inspected thoroughly for potential performance degraders.

Issues with source data integrity should be looked into, as the current implementation is tightly dependent on metadata of titles. Queries relying on the existence of the metadata could possibly be improved to ignore a given parameter if the data is not present. Data inaccuracy is a tougher problem to solve, as the data provider can only guarantee accuracy to a certain degree. However, having incomplete or broken data is a problem that occurs for every system handling data. As long as it is marginal it should not be a big problem but being aware of it is still important to try and minimize the impact of it.

Overall, as a result of this thesis project a solid structure was built for a recommendation engine concentrating on viewing times and giving satisfying results. The system utilizes a good amount of data and is easily expandable for further development allowing new implementations or even integrations of other solutions. The testing was brief, but enough to prove the concept as a viable solution.

4.2 Future

The future holds many opportunities for this system, as this project was just the peak of an iceberg. The first thing that should be implemented next is analytics for gathering data

of how much are the old recommendations clicked in the service. This would give clear metrics for the future when wanting to compare the systems side by side. The analytics are also highly valuable for future developments as they provide an easy comparison for potential improvements.

As the system is easily expandable, more data should be taken into use. Available data omitted from the scope of this project were devices used to watch the titles and age rating reasons for the title's metadata. Used devices is especially an important piece of data, as users most likely watch different type of content on their mobile phones than on their TV. Adding the type of device used as a parameter for finding titles when user enters the service could significantly improve the end results.

There has been discussion of potentially combining the results with another type of recommendation solution that is currently in development in the case company. A single type of recommendation system is rarely the best solution and therefore hybrid models are often being used.

4.3 Conclusion

The results of this thesis project proved that a recommendation engine does not necessarily need complex algorithms as is often thought. By utilizing the available data in smart and simple ways might lead to good results as well. In this case precise filtering with data was the key. Simple, yet effective. The solution staying simple and easily perceivable is an important matter for being able to maintain and develop the system. Future developers should quite easily get an idea of how the system works just by looking at the Cypher queries.

The target of this proof of concept project was reached with satisfying results. A viable alternative was found to offer as a replacement for the existing solution. Comprehensive testing with real users and deep diving into the data are now needed to move this project forward. A million and one things could be done for further developing the system. The future looks promising.

References

1 Ian Robinson, Jim Webber, Emil Eifrem. Graph databases, 2nd edition, 2015. O'Reilly Media.

2 Guest View: Relational vs. graph databases: Which to use and when? https://sdtimes.com/databases/guest-view-relational-vs-graph-databases-use/. Accessed: 28th October 2018.

3 Why you should use a graph database. https://www.infoworld.com/article/3251829/nosql/why-you-should-use-a-graph-database.html. Accessed: 28th October 2018.

4 Graph Databases for Beginners: Native vs. Non-Native Graph Technology. https://neo4j.com/blog/native-vs-non-native-graph-technology/. Accessed: 28th October 2018.

5 https://neo4j.com/company/. Accessed: 28th October 2018.

6 The history of Neo4j – Open Source, Big Community. https://neo4j.com/open-source-project/. Accessed: 28th October 2018.

7 What is a Graph Database? https://neo4j.com/developer/graph-database/. Accessed: 28th October 2018.

8 Cypher, The Graph Query Language. https://neo4j.com/cypher-graph-query-language/. Accessed: 29th October 2018.

9 https://neo4j.com/docs/developer-manual/3.4/cypher/clauses/create/. Accessed 31st October 2018.

10 https://neo4j.com/docs/developer-manual/3.4/cypher/clauses/match/. Accessed 31st October 2018.

11 Graph Databases for Beginners: ACID vs. BASE explained. https://neo4j.com/blog/acid-vs-base-consistency-models-explained/. Accessed: 28th October 2018.

12 ACID (atomicity, consistency, isolation, and durability) https://searchsqlserver.techtarget.com/definition/ACID. Accessed: 28th October 2018.

13 Atomicity Consistency Isolation Durability (ACID). https://www.techopedia.com/definition/23949/atomicity-consistency-isolation-durability-acid. Accessed: 28th October 2018.

14 The Raft Consensus Algorithm. https://raft.github.io/. Accessed 11th November 2018.

15 https://neo4j.com/docs/operations-manual/current/clustering/introduction. Accessed 11th November 2018.

16 Graph Modeling Guidelines. https://neo4j.com/developer/guide-data-modeling/. Accessed 4th November 2018.

17 Robert C. Martin. Clean code: A handbook of Agile Software Craftmanship, 2008. Prentice Hall.

18 Why Neo4j is the most popular graph database. https://hub.packtpub.com/neo4j-most-popular-graph-database/. Accessed 25th November 2018.

19 What is Docker? https://opensource.com/resources/what-docker. Accessed 2nd December 2018.

20 Neo4j with Docker. https://neo4j.com/developer/docker/. Accessed 2nd December 2018.

21 Spring Data Neo4j. https://spring.io/projects/spring-data-neo4j. Accessed 9th December 2018.

22 Neo4j - OGM Object Graph Mapper. https://neo4j.com/developer/neo4j-ogm/. Accessed 9th December 2018.

23 Overview of Docker Compose https://docs.docker.com/compose/overview/. Accessed 2nd December 2018.

24 Configure Neo4j connectors. https://neo4j.com/docs/operations-manual/current/configuration/connectors/. Accessed 6th December 2018.