Task 3 – Movie Database
Final Report

Federico Fregosi, Mirko Laruina,

Riccardo Mancini, Gianmarco Petrelli

June 8, 2020

Contents

1 Specifications

1.1 Existing Application Summary

The application is an aggregator of movies and movie ratings with the purpose of providing logged users statistics and informations about a large set of movies. Logged-in users can also rate movies they have watched while not logged-in users may still use the service to browse movie rankings and statistics but they are not able to give their rate. Only movies released in Italy are considered.

All users can search a movie and view its details (e.g., title, original title, duration, cast, ...) along with its average rating from users and from external sources.

In addition, all users can browse the list of movies sorting and filtering it by many parameters (e.g. year, genre, country, actors, ...).

System administrators can view all user profile pages and ban users. In order to do that, he can check the full history of ratings. Once a user is banned, he can no longer log in and his username and email cannot be used by new users.

The movie database will be built upon the publicly available IMDb dataset.

The ratings will be gathered by periodically scraping external websites (e.g., Rotten Tomatoes, Coming Soon, MyMovies).

1.2 Task 3 additions overview

Upon the application described above, which we have already designed and implemented, we will add the management of followed/following relationships between users and the suggestion of movies to the user based on his ratings.

In particular, every logged-in user can follow, or un-follow if previously followed, any other user. Following another user gives the ability to see which movies he has been rating. A dedicated section will allow to see all the ratings coming from followed users in a chronological order. A new section will be built and will be available to logged-in users in their profile page. It will show the list of followers and followings of the user, but will also provide suggestions for new users to follow, based on the current relationship with the other users. User search functionality will be extended to all registered users in order to more quickly find another user given his username.

Furthermore, in his homepage, a logged user will receive suggestions about movies that he might like based on both his ratings and the most recent ratings on the platform. By doing so, suggestions will reflect the current trends in the movie industry.

1.3 Actors

Anonymous user, registered user, administrator and updater “bot”.

1.4 Requirement Analysis

1.4.1 Functional Requirements

In addition to what has already been defined in Task2, a **registered user** can:

- follow another registered user

- un-follow a followed user
- browse the users he is following
- browse the users that are following him
- browse suggestions for new users to follow
- browse a list of suggested movies
- browse the latest ratings of followed users
- view the profile page of any other user
- search a user through a query by username

1.4.2 Non-Functional Requirements

- **Availability**: the Database must be replicated in order to be always available. Write operations on the Database can be eventually consistent.
- **Scalability**: the application must be able to scale to an arbitrary number of servers.
- **Security**: passwords must be stored in a secure way.
- **Responsive UI**: Client-side application must provide a responsive view both for pc, laptops and mobile devices.

2 Design

2.1 Use-case diagram

The use-case diagram is shown in Figure 1. Different colors are used to highlight cases that are exclusive of some actors: white cases are referred to all users; blue and green cases are referred to registered user and admin; yellow cases are exclusive of the admin. Green cases highlight the ones that were introduced in task 3 and all refer to actions that a registered user can do.

2.2 Class diagram

The class diagram is shown in Figure 2. It was decided to show *Country*, *Year* and *Genre* as separate entities as they are some of the fields over which aggregate statistics are calculated.

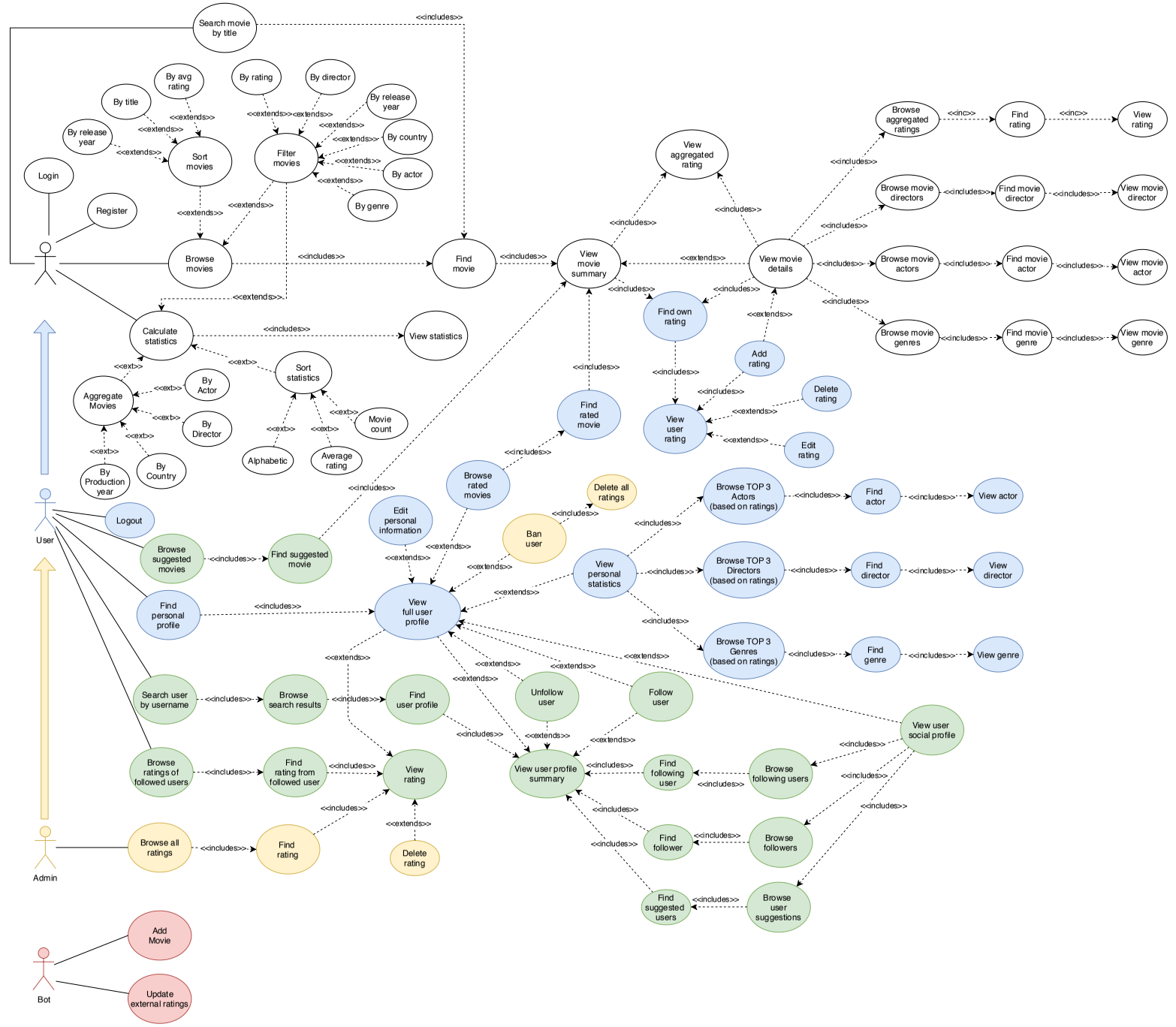

Figure 1: Use-case diagram

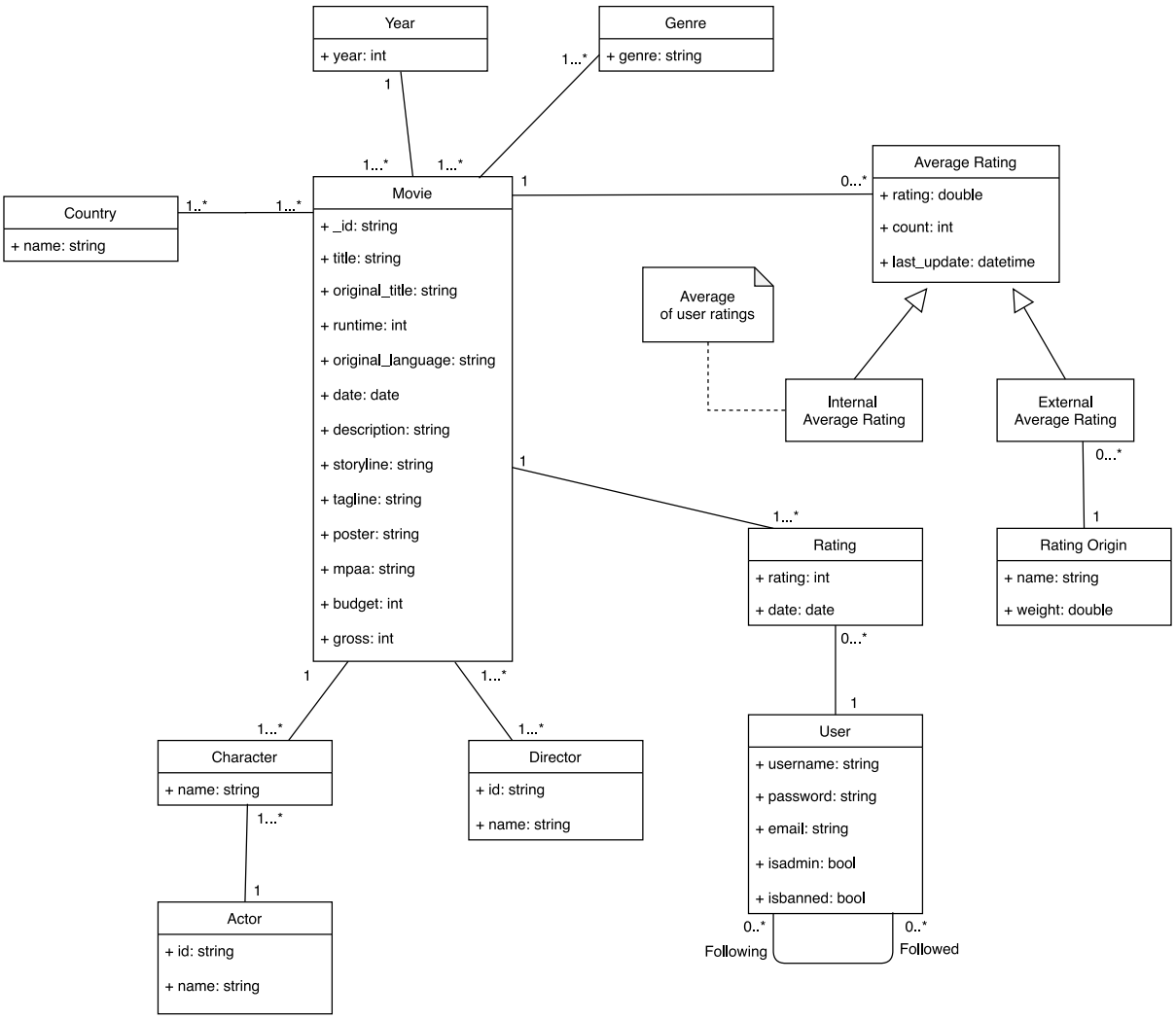

Figure 2: Class diagram for the identified entities

2.3 Data model

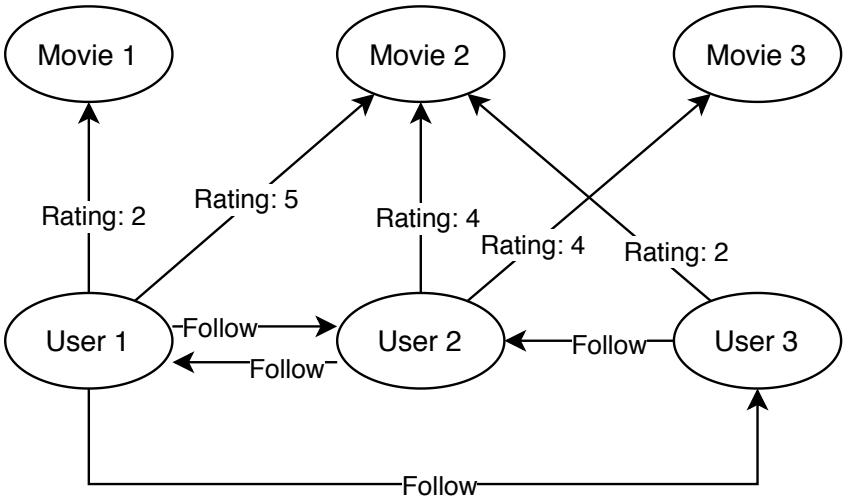

Figure 3: Example graph

The graph database will have 2 different types of nodes and 2 types of edges:

Node types

- **Movie**: it will contain all information needed to show a list of suggested movies: *id*, *title*, *year*, *poster*. In order to efficiently find a movie, an index is added to the *id* attribute.
- **User**: it will contain only the *username* and the *_id* since no other information is required. In order to efficiently find a user, an index is added to both the *id* and the *username* attributes (depending on the API, a user can be fetched either by one or the other).

Edge types

- **Rating** (User->Movie): it will contain the *rating* as attribute.
- **Follow** (User->User): no attribute is required.

In each following subsection, an example document is shown for every collection.

2.4 Graph-centric operations

This section will explain how we intend to execute the queries in the database in terms of graph operations. Simple CRUD operations will not be reported for brevity's sake.

Movie suggestion Movie will be suggested based on the ratings of the users that liked similar movies as the user we are suggesting to. We consider as liked movies, the ones the user has given a rating greater or equal to 3. For example, in figure 3, *Movie 3* may be suggested to *User 1* since *User 2* liked *Movie 2* as *User 1* and he also liked *Movie 3*. In practice, starting from a user we will go to each movie he liked, then to the users who recently liked that movie, then to other movies they liked. We will then count the number of paths like the one described for each movie (the last one in the path) and show the user the top-N movies given this metric.

User suggestion Users to follow will be suggested based on the number of common follow relationships. For example, in figure 3, *User 3* may be suggested to *User 2* since *User 2* follows *User 1* that follows *User 3*. In practice, starting from the user, we will go to each user he follows and from here to each user that the latter user follows. We will then count the number of paths from the user to the users with common followers and show the user the top-N users given this metric.

2.5 Software Architecture

The application will be made of the following 4 components:

- **Mongo DB**: a MongoDB cluster will be deployed with replication.
- **Neo4j**: an auxiliary graph database will be used to calculate more efficiently the movie suggestions and to store information about follow relationships since this kind of operations are more suited to this kind of DB.
- **React Front-end**: web-based UI.
- **Java Back-end**: using *Spring*, the *Java back-end* will provide REST APIs to the *React front-end*.
- **Updater "bot"**: three Python scripts are needed in order to nightly update the DB with the latest movies and ratings:
  1. the **IMDB parser** periodically parses the IMDb dataset to add the latest movies.
  2. the **scraper** continuously parses the rating sources to update the ratings.
  3. the **synchronizer** periodically synchronizes movies from MongoDB to Neo4j.

  These scripts will executes asynchronously from the *Java back-end*.

3 Implementation

3.1 Neo4j database

3.1.1 Coexistence of two databases

Providing strict consistency between the two databases is both very complicated and unnecessary given the application non-functional requirements. Therefore we took the path of eventual consistency. Writes that involve replicating data on both databases are first stored to MongoDB and then asynchronously to Neo4j to reduce perceived latency by the user. In case of Neo4j failure, it will be necessary to recover the synchronization of the two databases using the script described below. This script will be also be executed periodically to update movie details (which are updated asynchronously by the scraper scripts and for which we can afford a non perfect consistency) and after the IMDB parser script to add possible new movies.

Furthermore, Neo4j contains enough data to prevent a double access to both MongoDB and Neo4j in the same user query. However, due to the stricter consistency model, authentication is always performed on MongoDB. Therefore some queries will first need to authenticate using the session id with MongoDB before accessing the data stored in Neo4j.

3.1.2 Database Synchronization

The Neo4j database will be periodically synchronized by a python script that takes data directly from the document database: useful data is first withdrawn from MongoDB, then they it is post-processed in order to filter out unnecessary attributes before merging into Neo4j. This is done in a Python script using the *pymongo* and *py2neo* modules.

The script manage the synchronization of 3 out of the 4 elements present in our graph database:

- **Movie** nodes;
- **User** nodes;
- **Rated** edge.

Follows edges are not managed by the script because they only exists in the graph database.

The work is done by 3 functions (one for each managed element) that simply scan the array of documents that they receives as input in order to keep only the attributes we are interested in. Then they create the relative node and finally they merge it with the current Neo4j database.

The sync function for the **Movie** vertex in shown below:

```
def sync_movie(m):
    global graph

    movie = {k:v for k,v in m.items() if k in keep_movie_attrs}
    n = Node("Movie", **movie)
    graph.merge(n, "Movie", "_id")
```

The sync function for the **User** vertex in shown below:

```
def sync_user(u):
    global graph

    u['_id'] = u['_id'].__str__()
    user = {k:v for k,v in u.items() if k in keep_user_attrs}
    m = Node("User", **user)
    graph.merge(m, "User", "_id")
```

The sync function for the **Rating** edge in shown below:

```
def sync_rating(r):
    global graph

    RATED = Relationship.type("RATED")

    u = graph.nodes.match("User", _id=r["_id"]["user_id"].__str__()).first()
    m = graph.nodes.match("Movie", _id=r["_id"]["movie_id"]).first()
    graph.merge(RATED(u,m, rating=r['rating'], date=r['date']))
```

3.2 Java backend

In figures 4 and 5 you can see the UML class diagrams of the Java back-end. The structure is the same as the one presented in task 2 with the addition of the *Neo4jAdapter* class

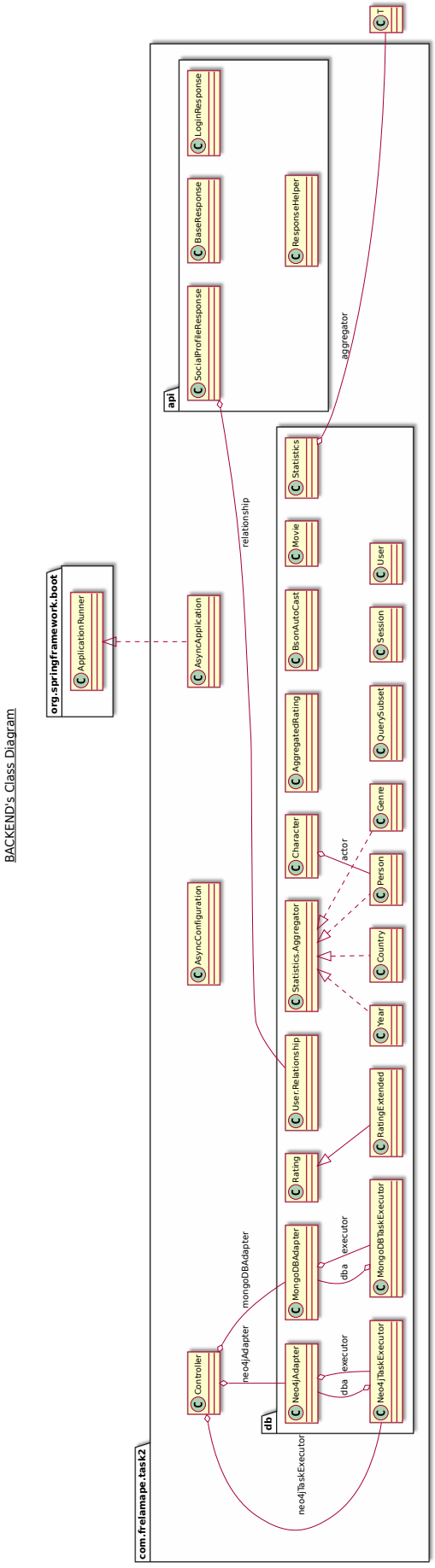

Figure 4: Bird's eye view over the main components of the Java backend.

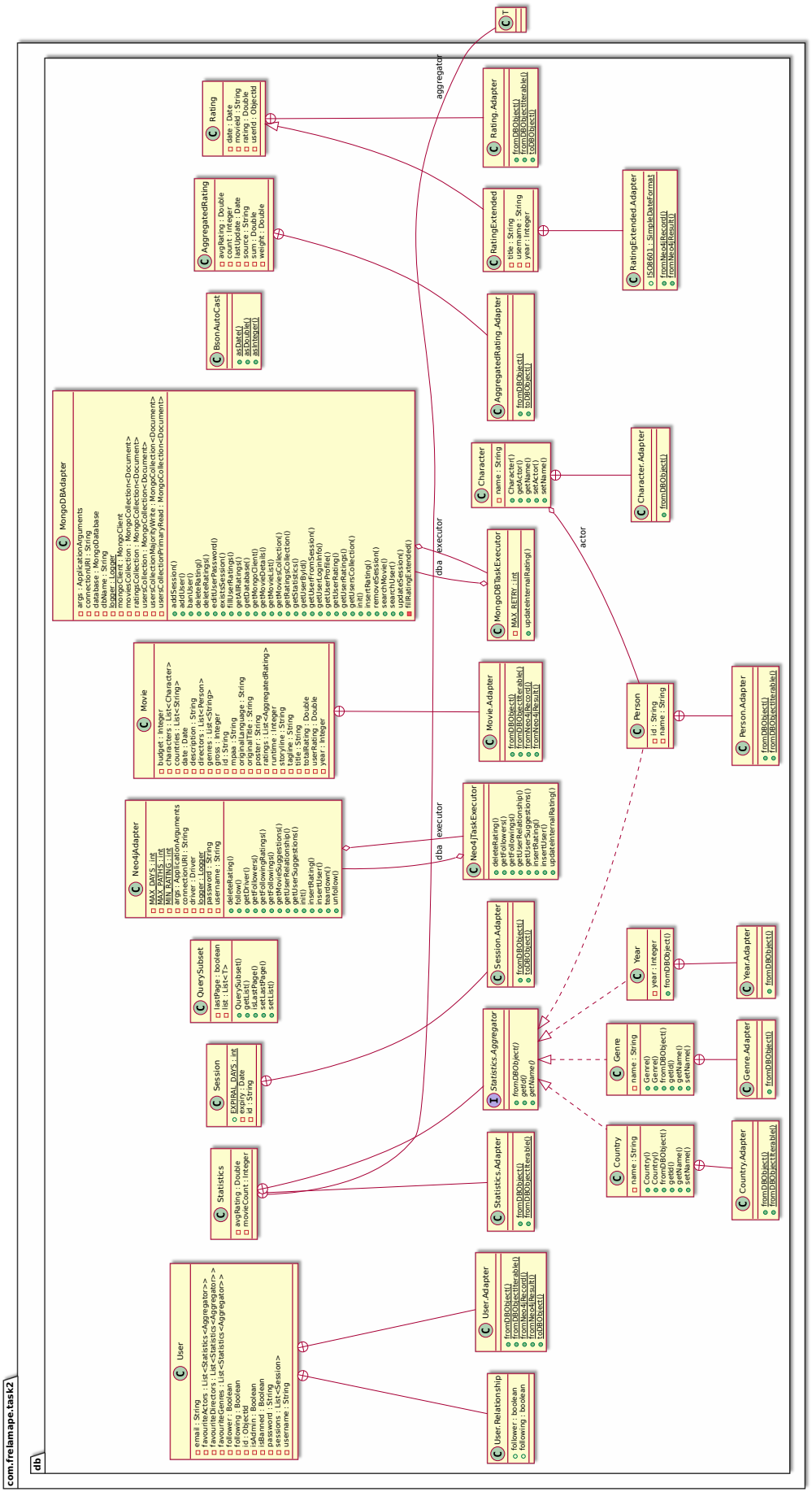

Figure 5: Detailed view over the db component of the Java backend.

which implements the Cypher queries to the Neo4j database (explained in the following sub-section) and of the *Neo4jTaskExecutor* which provides a simple way to asynchronously executing the queries when needed.

3.3 Neo4j Queries

In this section we'll go into details of the queries defined for our Neo4j graph database, according to Cypher query language. Cypher allows users to store and retrieve data from the graph database; its syntax provides a "visual" and logical way to match patterns of nodes and relationships in the graph. It is a declarative, SQL-inspired language for describing visual patterns in graphs using ASCII-Art syntax.

3.3.1 Movie suggestion

Aim of this query is to return a list of n suggested movies (exploiting *result* object), given as input:

- *User u*, the recipient of the suggestions, represented by *$username* in the query string (according to Cypher syntax);
- *int n*, the number of suggestions, represented by *$limit* in the query string.

Earlier we defined what we meant with movie suggestion. However, it is clear that the algorithm provided is inefficient in the sense that it needs to explore a large number of paths. Assuming each user rates in average 50 movies and that a movie has in average 1000 ratings, there are 50*1000*50 = 250000 paths! This is not affordable in such a frequent query. Therefore we need a way to prevent this explosion in the number of paths. The solution we used is 1) to filter out old ratings and – 2) if this is not sufficient, put a hard limit in the total amount of paths that will be considered in the final aggregation. The first limitation has also a positive side-effect: by keeping only most recent ratings, the most trending movies will be returned to the user. Furthermore, to avoid suggesting always the same movies to the user, for each movie we multiply the score by a random quantity so that the final value is randomly picked from a uniform distribution [score/2; score].

In order to do this filtering, we needed to introduce a Cypher sub-query since *LIMIT* statements can only be placed after a RETURN statement. Please note that the *LIMIT* statement effectively prevents the graph traversal to visit more nodes than necessary.

```
76 public QuerySubset<Movie> getMovieSuggestions(User u, int n) {
77   try (org.neo4j.driver.Session session = driver.session()) {
78     Result result = session.run(
79       "CALL{ " +
80         "MATCH (u:User {username:$username}) " +
81         "MATCH (u)-[rum1:RATED]->(m1:Movie)<-[ru2m1:RATED]-(u2:User)-[ru2m2:RATED
    ]->(m2:Movie) " +
82         "WHERE rum1.rating>="+MIN_RATING+" AND ru2m1.rating>="+MIN_RATING+" AND
    ru2m2.rating >=" + MIN_RATING + " " +
83           "AND duration.between(datetime(ru2m1.date), date()).days<=" + MAX_DAYS +
    " " +
84           "AND duration.between(datetime(ru2m2.date), date()).days<=" + MAX_DAYS +
    " " +
85           "AND u<>u2 AND m1<>m2 " +
86         "RETURN m2 " +
87         "LIMIT " + MAX_PATHS +
```

```
88        "} " +
89        "RETURN m2._id AS _id, m2.title AS title, m2.poster AS poster, 0.5*count(*)
      *(1+rand()) as score " +
90        "ORDER BY score DESC " +
91        "LIMIT $limit",
92        parameters(
93          "username", u.getUsername(),
94          "limit", n));
95
96      List<Movie> movies = Movie.Adapter.fromNeo4jResult(result);
97      return new QuerySubset<>(
98          movies,
99          true
100     );
101   }
102 }
```

At line 81, starting from the user passed as argument (u), we match the movies *m2* as the movies that have been rated by *u2* users, namely the users who share interest (i.e. at least a common liked movie) with u. From line 82 to line 85 are expressed some meaningful conditions: we only consider *RATED* edges with a rating value higher than *MIN_RATING*, both for *rum1* and *ru2m1*. This condition ensures that the movie is indeed a common interests among the two users. We also have the condition *ru2m2.rating >= MIN_RATING* to match only appreciated movies. Another relevant condition is expressed at lines 82-83 using the temporal function *duration.between(..).days < MAX_DAYS*, whose scope is to filter rating edges, considering only movies rated during the last week.

3.3.2 User suggestion

The aim of this query is to return n user suggestions to user *user*. The *getUserSuggestions* method takes 2 arguments:

- *user*: the user to get suggestions for.
- n: the number of user suggestions to return.

The same problem of "number of paths explosion" that we seen could happen also in this query. Therefore the same approach as before (without the temporal limitation). We also applied the same randomization technique as before.

```
229 public QuerySubset<User> getUserSuggestions(User user, int n) {
230   try (org.neo4j.driver.Session session = driver.session()) {
231     Result result = session.run(
232             "CALL{ " +
233                 "MATCH (u:User {username: $username}) " +
234                 "MATCH (u)-[:FOLLOWS]->(:User)-[:FOLLOWS]->(u2:User) " +
235                 "WHERE u<>u2 " +
236                 "RETURN u2 " +
237                 "LIMIT " + MAX_PATHS + " " +
238             "} " +
239             "MATCH (u:User {username: $username}) " +
240             "WHERE NOT EXISTS((u)-[:FOLLOWS]->(u2)) " +
241             "RETURN u2.username AS username, u2._id AS _id, " +
242                 "0.5*count(*)*(1+rand()) AS score, " +
243                 "false as following, " +
```

```
244                 "EXISTS((u2)-[:FOLLOWS]->(u)) as follower " +
245             "ORDER BY follower DESC, score DESC " +
246             "LIMIT $limit",
247         parameters(
248             "username", user.getUsername(),
249             "limit", n));
250
251     List<User> users = User.Adapter.fromNeo4jResult(result);
252
253     return new QuerySubset<>(
254         users,
255         true
256     );
257   }
258 }
```

In the first sub-query (lines 232-238), all users *u2* that are followed by a user who the user *u* follows are returned. In order to prevent that the user *u* himself is returned, it is checked that $u \neq u2$. Furthermore, to prevent the return of too many users, the result is limited to `MAX_PATHS`. The query then proceeds in the aggregation by username, filtering out users that are already followed by *u*. Note that this filter requires a sub-query and is therefore quite heavy, for this reason these users are not filtered out in the previous sub-query. Finally, the resulting users are sorted by *score* and the first `$limit` are returned.

3.3.3 Other queries

User creation Creates a new user.

```
1 CREATE (:User {_id: $id, username: $username})
```

Rating insertion/update This operations creates a new rating, if none already exists, otherwise updates the rating already in the DB.

```
1 MATCH (m:Movie {_id: $movie_id})
2 MATCH (u:User {_id: $user_id})
3 MERGE (u)-[r:RATED]->(m)
4 SET r.rating=$rating, r.date=$date
```

Rating deletion Removes a rating.

```
1 MATCH (m:Movie {_id: $movie_id})
2 MATCH (u:User {_id: $user_id})
3 MATCH (u)-[r:RATED]->(m)
4 DELETE r
```

Follow user A started following B.

```
1 MATCH (a:User {username: $username_a})
2 MATCH (b:User {username: $username_b})
3 MERGE (a)-[:FOLLOWS]->(b)
```

Un-follow user A stopped following B.

```
MATCH (a:User {username: $username_a})
MATCH (b:User {username: $username_b})
MATCH (a)-[r:FOLLOWS]->(b)
DELETE r
```

Ratings of followed users Returns a (paged) list of all ratings by followed users in chronological order (most recent first). *skip* and *limit* are set so that only the requested page of n elements is returned.

```
MATCH (:User {username: $username})-[:FOLLOWS]->(u:User)-[r:RATED]->(m:Movie)RETURN
     r.date as date, r.rating as rating,
    m._id as movie_id, m.title as title, m.year as year, m.poster as poster,
    u._id as user_id, u.username as username
ORDER BY r.date DESC
SKIP $skip
LIMIT $limit
```

List of Followed Users Returns the list of the users that a given user (*me* in the query) is following. Furthermore, the query returns the relationship between the returned *user* and a third user (called *pov* in the query), i.e. whether *pow* is *following user* and whether *user* is a follower of *pov*.

```
MATCH (me:User {username: $username})
MATCH (me)-[:FOLLOWS]->(user:User)
MATCH (pov:User {username: $username_pov})
RETURN user.username as username, user._id as _id,
    EXISTS((pov)-[:FOLLOWS]->(user)) as following,
    EXISTS((user)-[:FOLLOWS]->(pov)) as follower
ORDER BY user.username "
SKIP $skip
LIMIT $limit
```

List of Followers Returns the list of the users that follow a given user (*me* in the query). Furthermore, the query returns the relationship between the returned *user* and a third user (called *pov* in the query), i.e. whether *pow* is *following user* and whether *user* is a follower of *pov*.

```
MATCH (me:User {username: $username})
MATCH (user)-[:FOLLOWS]->(me:User)
MATCH (pov:User {username: $username_pov})
RETURN user.username as username, user._id as _id,
    EXISTS((pov)-[:FOLLOWS]->(user)) as following,
    EXISTS((user)-[:FOLLOWS]->(pov)) as follower
ORDER BY user.username "
SKIP $skip
LIMIT $limit
```

Users relationship Returns the relationship between user 1 and user 2, i.e. whether *u1* is *following* B and whether A is a follower of *u2*.

```
MATCH (u1:User {username: $username1})
MATCH (u2:User {username: $username2})
RETURN EXISTS((u1)-[:FOLLOWS]->(u2)),
    EXISTS((u2)-[:FOLLOWS]->(u1))
```

4 Tests

This section reports some tests that have been performed in order to evaluate the performance of the Neo4j database.

Before running the tests, random users, ratings and follow relationships have been created. In particular, 100 test users have been created and, for each user, 50 ratings and 10 follow relationships to another user. In order to better test the movie suggestion functionality, the ratings have been restricted to a subset of 1000 movies (instead of the full $\sim 50k$ movies).

In each of the following experiments, each data point has been obtained running the *siege* application with 100 concurrent users and no delay between requests for 1 minute on a random subset of the possible queries. Experiments have been executed on one of the virtual machines provided by the University of Pisa (1 virtual CPU core, 8GB of RAM).

4.1 Movie suggestion

As already discussed before, the movie suggestion query cannot afford to look up all possible paths and for this reason a limit in the maximum number of paths along which the aggregation is computed has been reduced. This subsection shows how we decided to use 100 as limit.

In figure 6, we reported the number of handled requests every second (i.e. transactions-per-second, tps, in *siege* jargon) for the movie suggestion query with different limits on the maximum number of paths to take into consideration. From the plot, it can be seen that the throughput drastically decreases already in the first step from 100 to 200. For this reason, we decided to set it to set it to 100, which provides an acceptable throughput (around 450). Of course this reasoning holds only for the current configuration on the VM provided by the University of Pisa.

In order to further increase the throughput of this very frequent query, the introduction of an in-memory database could be taken into consideration. This database could store the suggested movies to the user for as long as the user is active in the website (few minutes) in case the user goes to the homepage again (which is very likely). It is not advisable to cache this value longer since this would make suggestions too static, which we do not want for a better user experience.

4.2 Comparison of other common queries

In this subsection, the throughput of the most common queries on the graph database is reported. The queries we took into consideration are:

- Ratings of followed users
- List of followers/following users
- Suggestion of movies

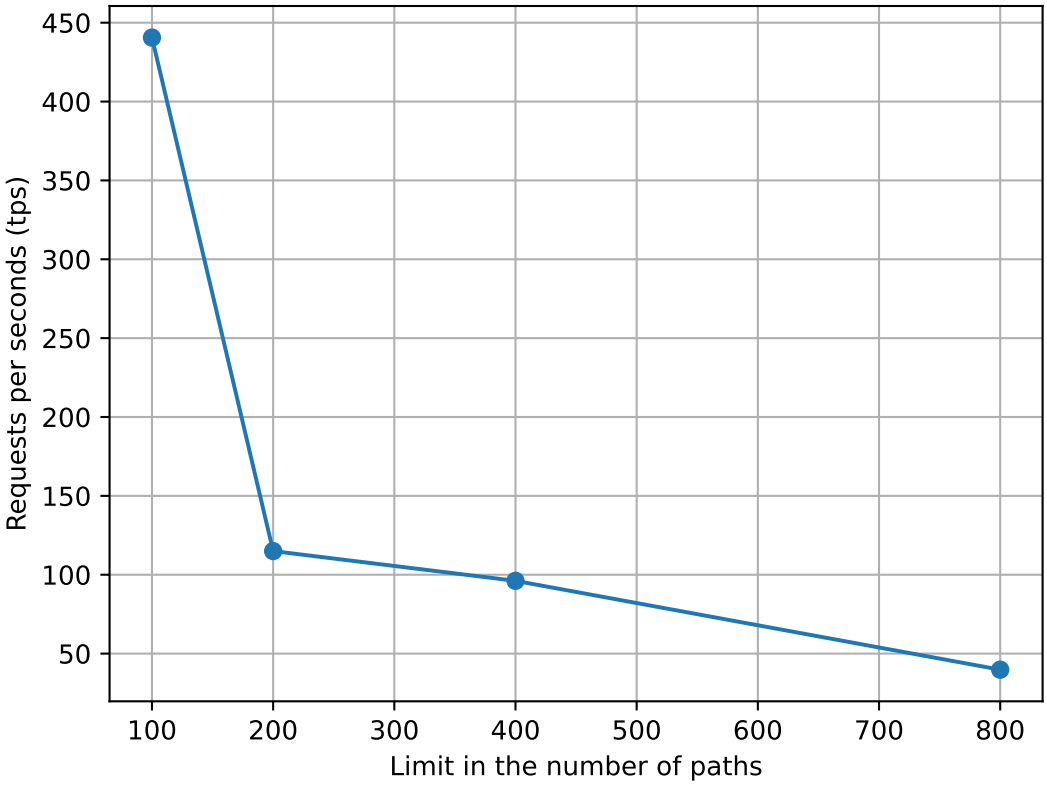

Figure 6: Plot of the handled requests per second for the movie suggestion query at different limits on the maximum number of paths.

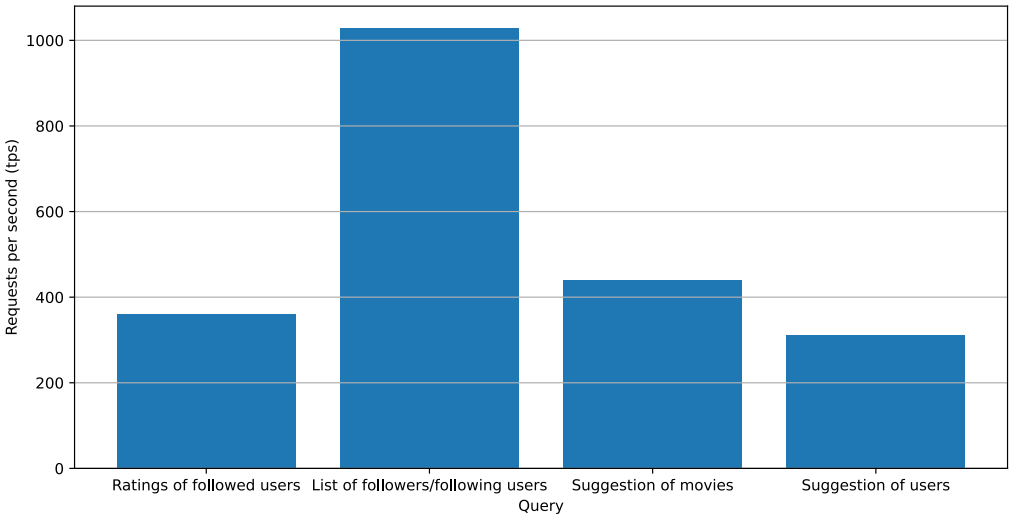

Figure 7: Bar plot of the throughput of different queries to the graph database

- Suggestion of users

Other queries, like ratings, users and follow relationships insertion and deletion have been ignored in the analysis since they represent just a very small amount of the total queries to the back-end.

In figure 7 we can see the throughput of queries to Neo4j. The results obtained for the other queries to MongoDB is also shown as a reference in figure 8. From the figures we can see how Neo4j is significantly slower than MongoDB, if we consider the highest performing queries of both. However, it must be noted that the aforementioned queries on MongoDB are much “simpler” than the ones on Neo4j.

Focusing on Neo4j queries, we can note that the fastest is the list of followers/following users and this was expected since it’s the most simple query that just requires to show all adjacent nodes. Next we have the other three queries that are all quite close to each other. It may look quite surprising to see that the suggestion of movies is faster that the list of ratings of followed users but there are two things to be taken into consideration. First, the suggestion of movies just looks 100 paths (as for the previous set of experiments). Second, the ratings of followed users has to perform an expensive order by operation before returning the requested page. In the last place, we can find the suggestion of users, which is slower due to the fact that it has to check that returned users are not already followed by the user performing the query in a sub-query.

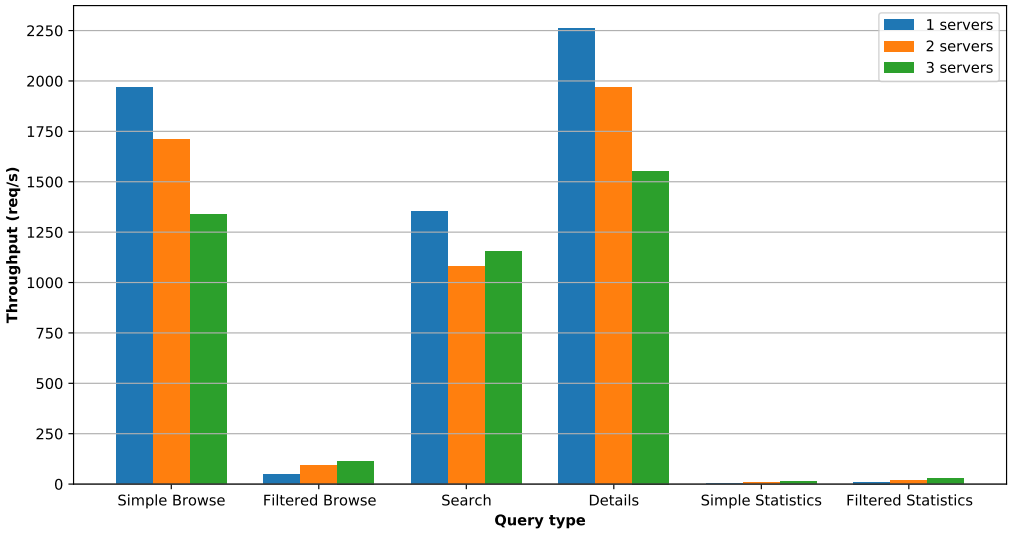

Figure 8: Bar plot of the throughput of different queries to the document database