Department of Information Engineering(DII)

UMDb - Movie Review

Tesfaye Yimam Mohammad

Arsalen Bidani

Aida Himmiche

Semester Final Project of Large Scale and Multistructure Database course.

Winter Session
Academic Year 2021/22

Contents

List of Figures

List of Tables

Listings

1 Introduction

UMDB is a web based application that aims to provide a platform for UniPi students where they can find information about a large collection of movies, give some reviews on them, and interact with other users as well. UMDB stands for "UNiPi Movie Database", therefore it stores movie-related information including ratings and reviews. Users can view movies, rate them, write reviews on them, and/or add them to their watch-lists. Users can also follow other users and view other users' profiles in addition to their own, moreover, movies and user profiles will be suggested based on their interactions on the platform. While User records may be manipulated by the users themselves, movie data manipulation will be controlled solely by the administrators.

2 Design

2.1 The Application Requirements

2.1.1 Actors

The major actors in the application are grouped in three categories: Anonymous users, Registered users and Administrators.Every registered user has their own privilege in the system based on their credentials given to them during a registration time.

- **Anonymous User**: A user with no account in the system has the most restricted access in the application. As a result, this user type is invited to create an account to exploit the extensive functionalities of our application.
- **Registered User**: This is the type of user which has login credentials to access their own session in the application. Once logged in he/she

can post reviews, create watch-lists, follow watch-lists, follow other users, and more.

- **Administrator**: it is the third type of users which has access to all previously mentioned functionalities, in addition to adding/removing movies and/or users from the database(s).

2.1.2 Functional Requirements

Here is a list organized by actors, describing the functionalities available for each:

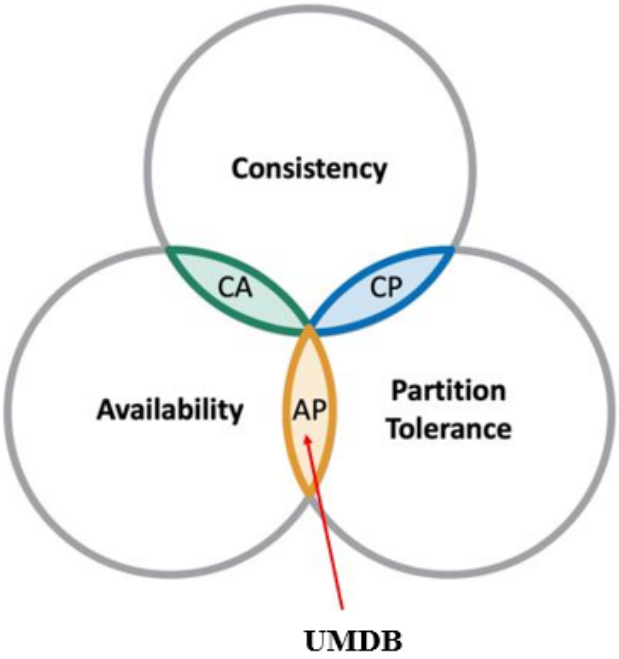
- **Anonymous User**:
  - Create a personal account in the application.
  - Log into a personal account using credentials.
  - Search for movies and read user reviews.
  - Search for registered users on the platform.
  - Find general analytics about movies, users, and reviews on the platform.
- **Registered User**: All previously mentioned in addition to:
  - View/edit personal information on the user-dedicated page.
  - Post a review about a movie.
  - Edit/delete personal previously-posted reviews.
  - Create a watch-list.
  - Edit/delete personal watch-lists.
  - Add/remove a movie from a personal watch-list.
  - View other users' watch-lists.
  - Follow/unfollow other users' watch-lists.
  - View list of followed watch-lists.
  - Follow/unfollow other users.

  - View list of followed users.
  - Receive user profile suggestions.
  - Receive watch-list suggestions.
  - Receive movie suggestions.
  - Logout of personal account.
  - Delete personal account.

- **Administrator**: All previously mentioned in addition to:
  - Add/delete a movie.
  - Delete any user account of choice.
  - Delete any user review of choice.
  - Give administrator privileges to users of choice.

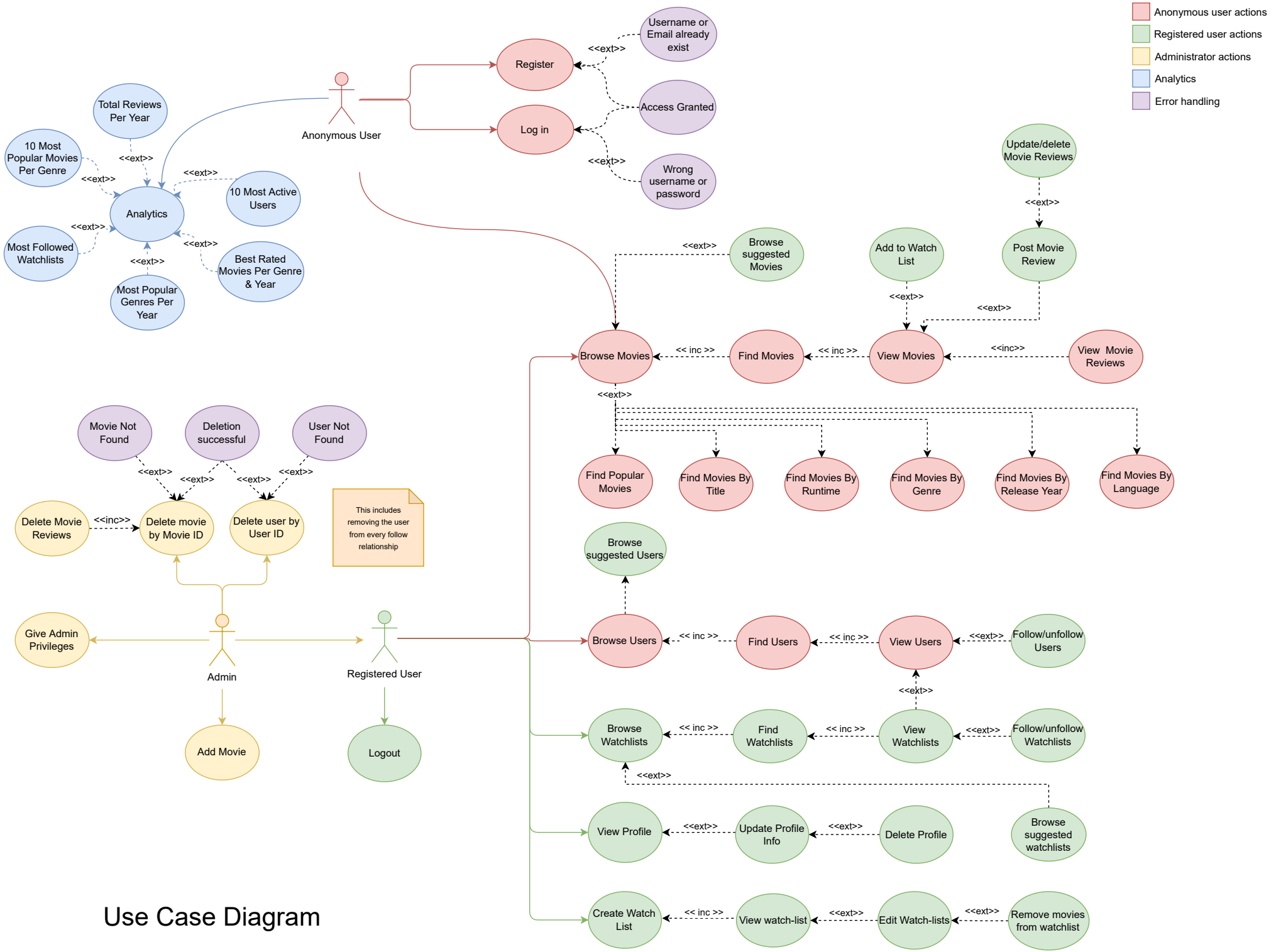
2.1.3 Non-Functional Requirements

- Usability: The user interface should be as comfortable as possible to the end user.
- Users must be able to exploit all the functionalities of the application being supported by high availability, a low latency and a good level of tolerance with respect to the possibility that there might be network partitions. Therefore, the application has to be designed in order to cover the AP (Availability / Partition protection) edge of the CAP triangle. More detail is found at fig.2.1.3.



- Maintainability: The application's code must be easy to read and to maintain.
- Security: Users' passwords must be stored in an encrypted form within the database, to guarantee an high level of security the SHA-256 function will be adopted.

2.2 Use case Diagrams


Anonymous user actions
Registered user actions
Administrator actions
Analytics
Error handling
Anonymous User
Register
Log in
Username or Email already exist
Access Granted
Wrong username or password
<<ext>>
Total Reviews Per Year
10 Most Popular Movies Per Genre
Analytics
10 Most Active Users
Most Followed Watchlists
Best Rated Movies Per Genre & Year
Most Popular Genres Per Year
Update/delete Movie Reviews
Post Movie Review
Add to Watch List
Browse suggested Movies
Browse Movies
<< inc >>
Find Movies
View Movies
<<inc>>
View Movie Reviews
Find Popular Movies
Find Movies By Title
Find Movies By Runtime
Find Movies By Genre
Find Movies By Release Year
Find Movies By Language
Movie Not Found
Deletion successful
User Not Found
Delete Movie Reviews
<<inc>>
Delete movie by Movie ID
Delete user by User ID
This includes removing the user from every follow relationship
Browse suggested Users
Browse Users
Find Users
View Users
Follow/unfollow Users
Give Admin Privileges
Admin
Registered User
Browse Watchlists
Find Watchlists
View Watchlists
Follow/unfollow Watchlists
Add Movie
Logout
View Profile
Update Profile Info
Delete Profile
Browse suggested watchlists
Create Watch List
View watch-list
Edit Watch-lists
Remove movies from watchlist
Use Case Diagram

2.3 Class Analysis Diagram

Fig.1 shows the Class diagram of the application.

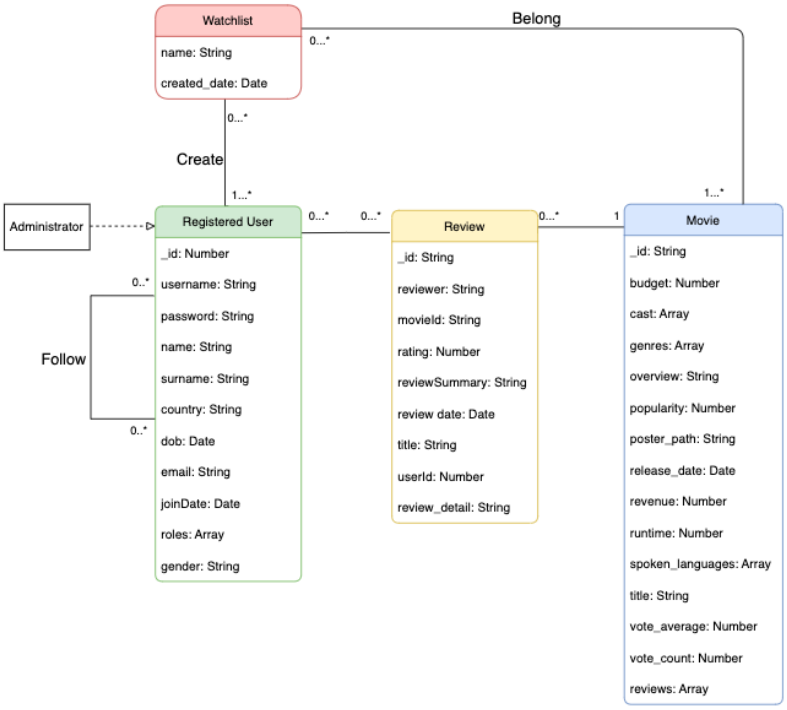


Figure 1: Class Diagram of UMDB

2.4 Dataset

Description To simulate a large-scale database for this application, we consulted different sources to amass data.

We chose a movie dataset provided on the Kaggle platform under the name *The Movies Dataset*. It contains a total of 45k movies, with information including the title, budget, cast, release date, genres, runtime, overview, languages spoken, revenue, average rating, and number of votes.

The user reviews were acquired from a different dataset on the same platform with the name *IMDb Review Dataset*, it grouped a total of over 190k reviews with information on the reviewer, movie title, rating, review summary, review detail, and review date. The way we matched the reviews with the movies was using a common field referring to the IMDb ID of the movie.

User information was not provided by any dataset, instead we used the open-source API *Random User Generator* to randomly generate 78k profiles with personal details such as the E-mail, password, name, surname, gender, country, date of birth, and date of account creation. To connect this data to the reviews, we randomly assigned each user profile one of the unique usernames in the reviews.

Volume and Format Here are the details about the volume and format of the datasets after complete manipulation:

- **The Movies Dataset**: Size : 323.6MB ; Format : CSV.
- **IMDb Review Dataset**: Size : 258.2MB, Format: JSON.
- **Users Dataset**: 13.3MB; Format : CSV.

Variety The data we used is multi-sourced and multi-formatted in order to achieve the aspect of variety.
Velocity/Variability The administrator can populate the database with new movies to achieve velocity.

2.5 Database Design

To store all the data of the application we have decided to use a Document Database (MongoDb) and a Graph database (Neo4j). The designed data model ensures that each read operation requires a single call either to the Graph Database or to the Document database.

2.5.1 Data Models

The **MongoDb** has been used in order to maintain data for the following collections: *users*, *movies*, *reviews*. Here we present the detailed information about the attributes found in the documents of each.

- Users
  - _id: Number
  - username: String
  - email: String
  - password: String
  - gender: String
  - name: String
  - surname: String
  - country: String
  - dob: Date
  - roles: Array[String]
  - joinDate: Date
- Reviews
  - _id: String
  - userId: Number
  - reviewer: String

    - movieId: String
    - rating: Number
    - title: String
    - review_summary: String
    - review_detail: String
    - review_date: Date

- Movies
    - _id: String
    - title: String
    - budget: Number
    - cast: Array[Object]
    - genres: Array[String]
    - overview: String
    - popularity: Number
    - poster_path: String
    - release_date: Date
    - revenue: Number
    - runtime: Number
    - spoken_languages: Array[String]
    - vote_average: Number
    - vote_count: Number
    - reviews: Array[Object]

NB: The 'roles' entry in the users collection differentiates the normal registered users from those who have administrative privileges in the system.

In the upcoming paragraphs we are going to present the skeleton of the document database and the reasoning behind it.

Movie Collection: As shown in fig.2 the "movies" collection comprises data about the movies imported from the appropriate aforementioned CSV file.

Our first design of this collection stated that the movie document included its reviews as embedded documents in the form of an array, since the reviews would only be retrieved when a Read operation on the movie is executed. However, considering the nature and context of our application (being Read-heavy and prone to accumulating large numbers of reviews per movie) we thought about the size of the documents and the speed of the Read operations on the collection.

The maximum size a document can have in a MongoDb collection is 16Mb. In our "movies" collection, the average size of a movie document excluding the reviews is 4.5kB, and the average size of a review document is 1.5kB. The maximum number of reviews per movie in our dataset is more or less 2880. At first glance, it seems as though the final size of the document with embedded reviews won't be a problem because of the small size of the review dataset at hand. However, looking at a case where a low number of 15000 registered users wanted to post one review each on a popular movie, the size would easily exceed 16Mb, not only making the Read operations very slow, but also making the Write operations impossible.

Having ruled out the possibility of embedding the reviews into the "movies" collection, we thought of keeping them in a collection of their own, with a reference to the movie they are posted for. However this would mean that whenever a movie is to be displayed, two collections need to be accessed to get the entire data, and that beats the purpose of using NoSQL.

Since neither extremes are suitable for our application, we decided to opt for the middle ground. We created our "movies" collection with an embedded array of only the 20 most recent reviews for each movie, and also kept the reviews in another collection in the case of the user wanting to load more. This allows us to avoid a join operation right from the start, but only resort to it when required.

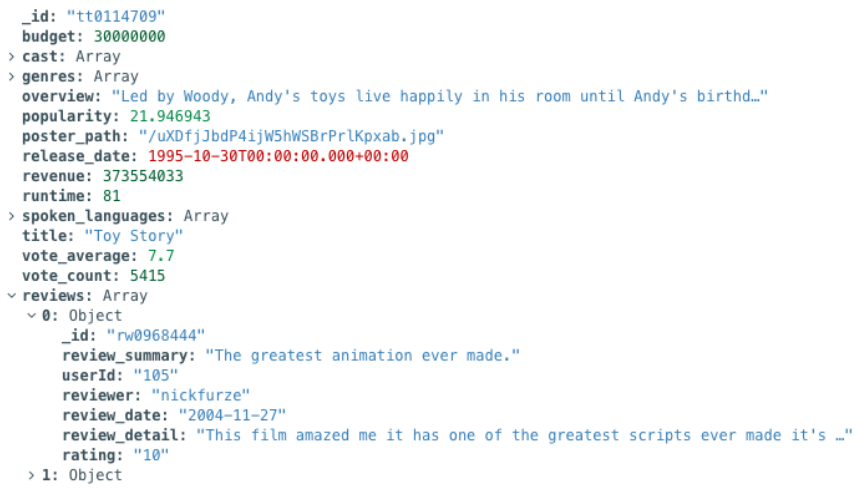


Figure 2: A document inside the Movie collection

User Collection: As depicted in fig:3 The documents in this collection provide the all the necessary information about the Users. The queries on this collection are independent from the other two and comply with the NoSQL requirements.

Figure 3: A document inside the User collection

Review Collection: This collection groups all the reviews imported from the appropriate dataset, including the ones embedded in the movie documents. Our reason behind allowing repetition in this case, is because without

it, it would be hard to keep track of the reviews that are being added as they would be pushed directly in the embedded array, moreover, aggregations on the reviews would require a lookup operation with the access of both the "reviews" and "movies" collection.

Since we would prefer to avoid join operations as much as possible, we made our decision keeping in mind that no inconsistencies between the duplicated reviews shall arise. For that, we made sure of the atomicity of Create-Update-Delete actions on said duplicates.

NB: The "userId" attribute stored in the review documents in not used in any join/lookup operations. It is stored as such for purposes related to front-end navigation and authentication. The fact that the field never undergoes any sort of modification makes it so that no consistency management is needed.


```
_id: "rw1986516"
movieId: "tt0114709"
rating: 10
review_date: 2008-07-12T00:00:00.000+00:00
review_detail: "The movie was totally fun and cool. The toys in Andy's Room are superb…"
review_summary: "It was totally Fun and cool"
reviewer: "manikhero"
title: "Toy Story"
userId: 1
```


Figure 4: A document inside the Review collection

2.5.2 Graph Database

The graph database has been used in order to move in the most efficient way through the FOLLOW,CREATE and BELONGS relationships between different users, watch-lists and the movies nodes. By exploiting this NoSQL architecture, some queries can be efficiently performed to find, having a certain user as a starting point, a list of suggested users to follow or a list of recommended watch-lists based on the the number of followers they have and the movie recommendations based on their occurrence in a number of watch-lists to mention a few. **GraphDB Nodes**: As mentioned above there are three main nodes in the GraphDB and below we have presented the attributes of each node.

Taking into account these objectives, it was not necessary to store in the Graph Database too much information concerning the instances of each and every entity. In particular, for each of these ones, only the following information are present in the graph database:

- User
  - username :String
  - id: Int
- Watchlist
  - created_date :Date
  - name :String
- Movie
  - title: String
  - movie_id: String
  - overview :String
  - poster_path :String
  - vote_average :Int

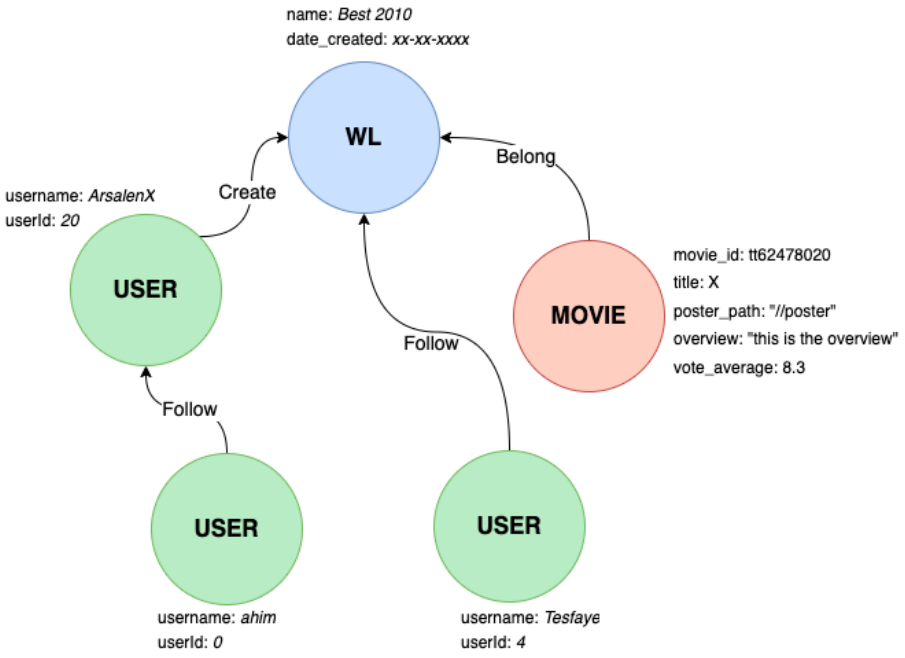


Figure 5: The GraphDB Model of UMDB

Relation types in the Neo4j As mentioned above the three nodes are connected in the neo4j database as follows.

- User-User : (u1:User)-[:FOLLOW]→(u2:User) It connects a user to an other user. It has no attributes
- User-Watchlist : (u:User)-[:FOLLOW]→(w:Watchlist). This relation connects user to watchlist nodes he/she is following. It has no attributes.
- Movie-Watchlist : (m:Movie)-[:BELONGS]→(w:Watchlist). This relation connects a movie node with a watchlist node. It has no attributes
- User-Watchlist : (u1:User)-[:CREATE]→(u1:User). It connect the user with the watchlists he/she has created. It has no attributes

2.5.3 Indexes

One of the components of the database design is indexes, as it makes it easier to query the large documents. We stated that one of the requirements is the speed handling of the web application.

We opted for 4 indexes on **MongoDB** as well as 4 indexes in **Neo4j**.

- **MongoDB**
  - **Popularity** Simple Index Type
  - **Year,Genre** Compound Index Type
  - **_id** Hashed Index Type (Default)
  - **Username** Unique Index Type
- **Neo4j**
  - **User_id** BTREE Index Type
  - **username** Unique Index Type
  - **watchlist_id** BTREE Index (Default)
  - **Username** FULLTEXT Index Type

Indexes in the performance as they optimise multi-key lookups but we need to be careful of the costs of indexes on update,inserts and deletes.

Popularity Simple Indexing type

```
db.movies.aggregate([ { $sort: { popularity: -1 } }, { $project: { title:
    1, poster_path: 1, overview: 1, vote_average: 1, release_date: 1, },
    }, ])
```

Listing 1: Popularity Simple Indexing type

Year Genre Component Index type

```
db.movies.aggregate([ { $match: { $and: [ { release_date: { $gte: new
    ISODate("2012-01-12T20:15:31Z"), $lt: new ISODate("2020-01-12T20:15:31
    Z") } }, { genres: { $regex: "Comedy", $options: "i" } }, ], }, }, {
    $sort: { vote_average: -1 } }, { $project: { title: 1, poster_path: 1,
     vote_average: 1, release_date: 1, }, }, ]).explain("executionStats")
```

Listing 2: MongoDB: Year Genre Component Index type

The indexing for the Neo4j part
For the Neo4j we have done the User Id and the Full text indexing. Below are the listings for both of them, according to their order.
Indexing for User Id

```
match(u:User)
            where u.user_id=${userId}
            with *
            optional match(ux:User)-[followers:FOLLOW]->(u)
            optional match(u)-[following:FOLLOW]->(Uy:User)
            return {user_id:u.user_id,username:u.username,following:count(
    distinct following), followers:count(distinct followers) }
```

Listing 3: Neo4j Indexing: User Id

FULLTEXT with index:

```
CALL db.index.fulltext.queryNodes("userSearch", "a")
        YIELD node,score
        return {user_name:node.username,user_id:node.user_id}
        ORDER BY score
        SKIP ${skip}
        LIMIT 7
```

Listing 4: Neo4j Indexing:Full text: One

FullTEXT with out index:

```
MATCH (u:User)
    WHERE toUpper(u.username) CONTAINS  toUpper("${searchString}")
    with *
```

```
    optional match(ux:User)-[follower:FOLLOW]->(u)
    with u,count(follower) as num
    where num in range(${min},${max})
    return {user_name:u.username,user_id:u.user_id}
    ORDER BY u.username
    SKIP ${skip}
    LIMIT 7
```

Listing 5: Neo4j Indexing:Full text:Two

With Out indexing for Compound:

```
    nReturned: 568,
     executionTimeMillis: 92,
     totalKeysExamined: 0,
     totalDocsExamined: 45145,
```

With Indexing for Compound:

```
nReturned: 568,
     executionTimeMillis: 8,
     totalKeysExamined: 3417,
     totalDocsExamined: 568,
     executionStages:
```

Without index for the Genre:

```
     nReturned: 568,
     executionTimeMillis: 92,
     totalKeysExamined: 0,
     totalDocsExamined: 45145,
```

With indexing for the Genre:

```
    nReturned: 568,
     executionTimeMillis: 8,
     totalKeysExamined: 3417,
     totalDocsExamined: 568,
     executionStages:
```

Neo4j User Id: without index

```
without index
| Time | DbHits | Rows | Memory (Bytes) |
 105  | 157306 | 1    | 1512
```

Neo4j User Id: with index

```
| Time | DbHits | Rows | Memory (Bytes) |
+----------------------------------------
| 10   | 57     | 1    | 1432           |
+----------------------------------------
```

Neo4j FULLTEXT: without indexing

```
| Time | DbHits | Rows | Memory (Bytes) |
 105  | 157306 | 1    | 1512
```

Neo4j FULLTEXT: with indexing

```
time | DbHits | Rows | Memory (Bytes) |
| 10   | 226    | 113  | 0
```

2.6 Software Architecture

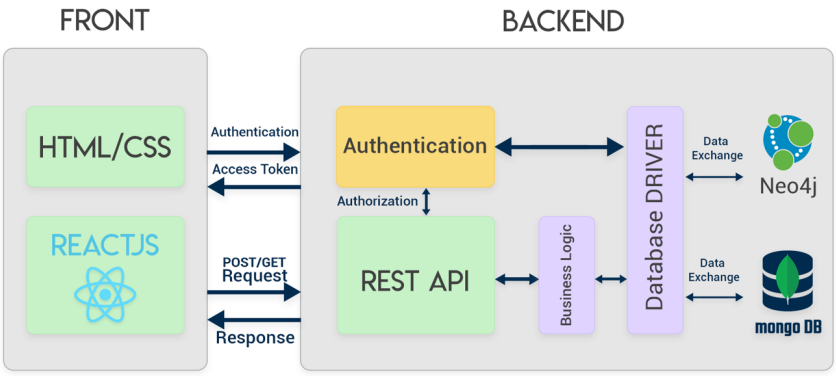


Figure 6: The Software architecture of UMDB

We used the well known Model-View-Controller (MVC) software development pattern to fdesign our project.The model contains the data logic of the movies,reviews and users collections.The controller contains all the control logic for all the documents and graph DB nodes.

2.6.1 Modules

As depicted in fig.6 the UMDb application has the following organization, according to the MVC architecture.

- **Backend**:- This is a folder that contains all the back end related files. All the controllers, models, routes and middle-ware of the application are grouped there.
- **Public**: This folder includes the index.html file and images referenced throughout the application.

- src: it comprises the API, hooks and all the JavaScript files that play a role in the rendering of the application.

We used the MERN stack plus the Neo4j graph database. Below are the list of the software tools used to accomplish this work.

- HTML: to design the basic structure of the web application.
- CSS: for manipulating the aesthetic of the front-end pages.
- React: to design the UI of our application.
- POSTMAN: for testing the routes to our business logic.
- Neo4j: The nodes of the graph DB are stored and queried from the Neo4j database.
- MongoDB: The collections of the document database are stored and queried from the MongoDB database.

2.6.2 User Interface

For a better interaction between the end user and application under the hood of the databases, it has been decided to use the the React framework for the front end UI. As said previously from the MERN stack React is used to build all the UI part so that the end user can have one of the finest experiences to the state-of-the-art web technology. The Home page of the application is shown in fig.7. It is very easy and user friendly application so that the end user can easily click on the provided navigation buttons to surf our website.Moreover,in this page the most popular movies are depicted.

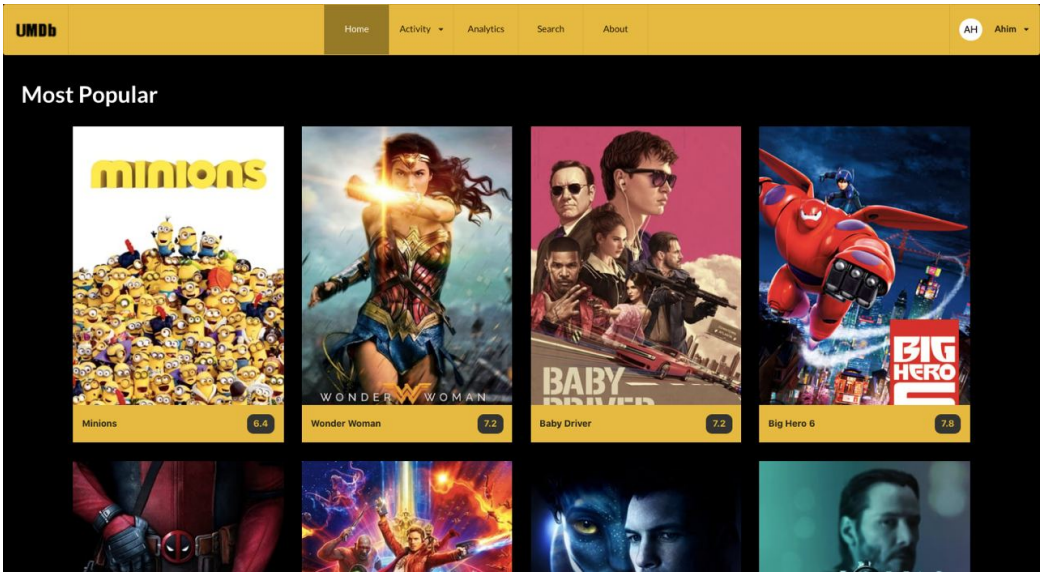


Figure 7: Home Page

As seen in fig.7 all the buttons in the navigation bar helps the user to surf the site. In the right top corner there is a Sing in and Login option. As can be seen the home page has got a suggestion of the most popular movies based on the user activities.

2.7 Database Deployment

2.7.1 Cluster replicas Topology and Consistency management

The document database has been deployed on a cluster of three virtual machines by which their IP address as depicted in fig.8

- The primary replica is at the *172.16.4.60*
- The Secondary replica is at the*172.16.4.64*
- The last replica is at the *172.16.4.65*

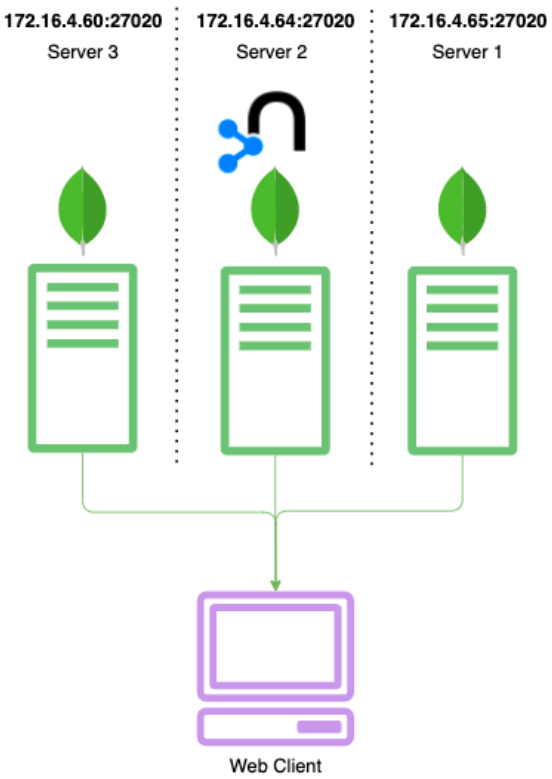


Figure 8: Replica topology for the document database

The data consistency between the different collections of the document database is completely handled by the application code.
A higher availability level is going to be provided, since the databases won't get stuck communicating in order to synchronize every modification occurred; at the same time, plus to this the usage of partition replica, removes the possibility of a denial of service situation, since the eventual failure of the primary server won't stop the application's runtime. Hence, the combination of the two properties from the CAP triangle locates our UMDB application into the (A-P) edge of the triangle as seen from fig.2.1.3.

2.7.2 Cross Database consistency management

Consistency between MongoDB and GraphDB

So as to maintain the consistency between the two databases, for each write operation that took place in the DocumentDB, the GraphDB is updated accordingly before the control is returned to the application.In fig.9 the logic to maintain the consistency between the databases for adding a user data. This same logic is applied to add a movie to the database. Generally, speaking the deletion also follows a similar reasoning except the fact that the two operations are opposite.

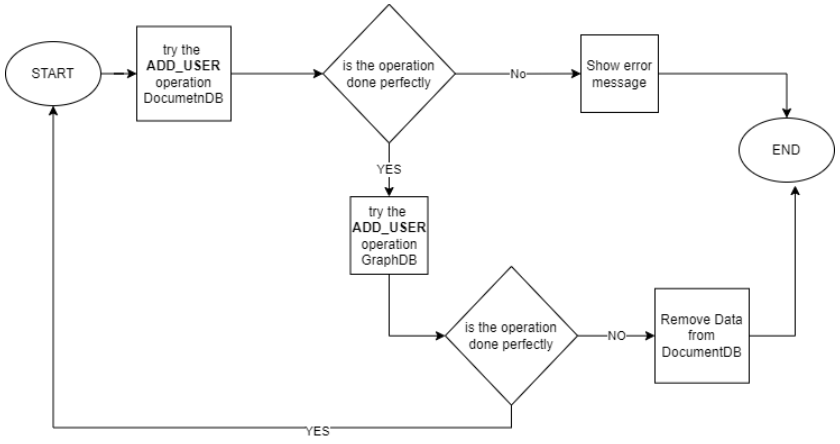


Figure 9: Cross Database Coherence for ADD USER operation

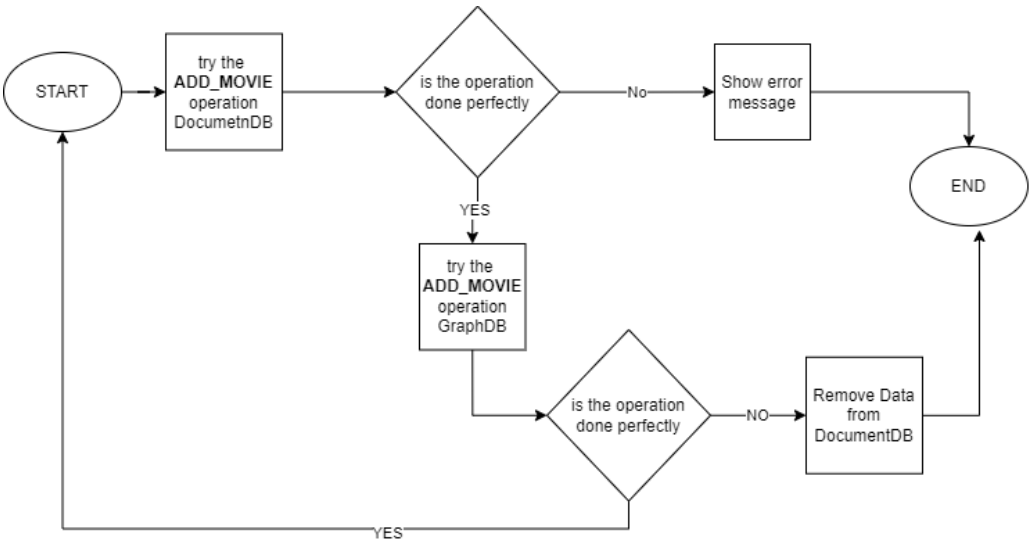


Figure 10: Cross Database Coherence for ADD MOVIE operation

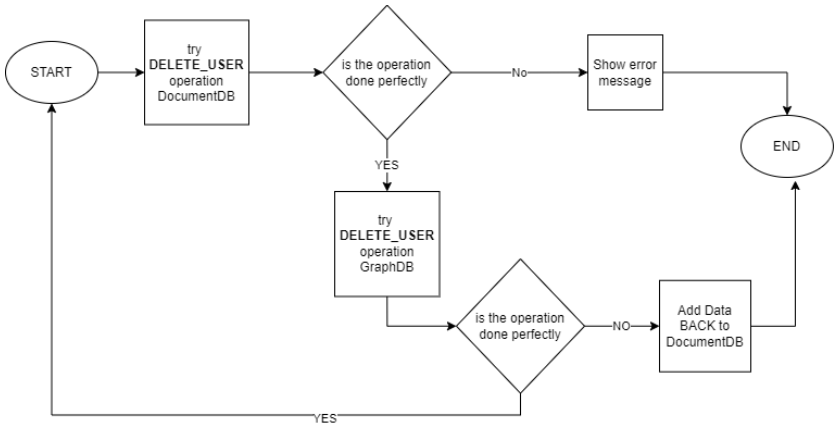


Figure 11: Cross Database Coherence for DELETE USER operation

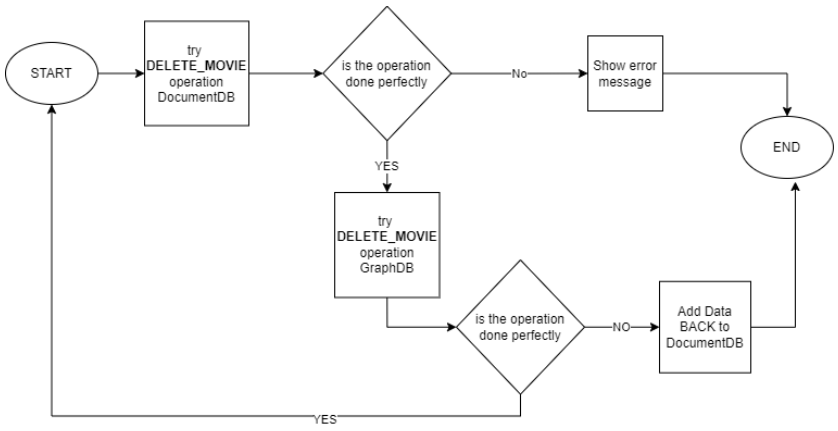


Figure 12: Cross Database Coherence for DELETE MOVIE operation

3 Sharding

A further improvement on the availability could be achieved by deploying the document database on a sharded architecture. A possible sharding key capable of distribute in a relatively could be the _id field inserted by the DMBS itself. This field, in fact, is computed by means of an hash algorithm on the document itself. Hash algorithms are able per design to produce random values uniformly distributed on a quite wide spectrum of values(given by the digest length itself).

4 Implementation

4.0.1 MongoDB Query Implementations And Analysis

The document part of our application is implemented using MongoDB, a source-available cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemas. [1].

Read Operations		
Operation	**Expected Frequency**	**Cost**
View movie information with reviews	Very High	Low(One Read)
Get popular movies	High	High (Aggregation)
Get popular movies by genre	Average	High (Aggregation)
Get top movies by year	Average	High (Aggregation)
Get top movies by year and genre	Average	Very high (Complex Aggregation)
Get top movies by year	Average	High (Aggregation)
Get the most popular genre per year	Average	Very High (Complex Aggregation)
Movie search with dynamic filtering	High	Average (multiple Reads)
Get the reviews of a movie	Very High	Low (One Read)
Get "more" reviews of a movie	Average	High (Aggregation)
Get the total reviews added per year	Average	Very High(Complex Aggregation)

Table 1: Read Operations on MongoDB

Write Operations		
Operation	**Expected Frequency**	**Cost**
Add user	High	Average (Document add)
Edit user	Average	Average (Document update)
Delete user	Average	Average (Document remove)
Add movie	Low	Average (Document add)
Delete movie	Low	Document remove + Documents (reviews) remove
Add review	High	Document add + Document update
Edit review	Average	Document update + Document update
Delete review	Average	Document Delete + Document update
Update a movie's vote information	High	Low(1 or 2 attribute writes)

Table 2: Write Operations on MongoDB

From this Query analysis we can say that the application is equally Read-Write heavy, because of the complex aggregations in the read operations, and also the 2-part atomic write operations related to reviews, as they are duplicated in two collections. **Read Operations:** Here is a selection of Read operation implementations in MongoDB:

Get popular movies

```
db.collection("movies").aggregate([
     { $sort: { popularity: -1 }},
     { $project: {
         title: 1, poster_path: 1, overview: 1, vote_average: 1,
 release_date: 1}}
   ]).skip(skipped).limit(28).toArray();
```

With this aggregation we get the popular movies by sorting based on the "popularity" attribute and projecting only the fields used in the front-end.

Get top movies by year and genre

```
db.collection("movies").aggregate([
     { $match: {
         $and: [
           { release_date: { $gte: reqYear, $lt: yearLimit }},
           { genres: { $regex: genre, $options: "i" }}
         ]}},
     { $sort: { vote_average: -1 } },
     { $project: {
         title: 1, poster_path: 1, vote_average: 1, release_date: 1}},
   ]).skip(skipped).limit(30).toArray();
```

This aggregation takes a genre and year as inputs and finds the movies that match, then gets the top movies by sorting based on the "average vote" attribute and projecting only the fields used in the front-end.

Get the most popular genre per year

```
db.collection("movies").aggregate([
     { $unwind: { path: "$genres" }},
     { $project: {
         year: { $year: { $toDate: "$release_date" }},genres: 1,
 popularity: 1}},
     { $group: {
         _id: { genre: "$genres", year: "$year" },
         average_popularity: {$avg: "$popularity"}}},
     { $group: {
         _id: "$_id.year",
         most_popular_genre: {
```

```
            $max: "$average_popularity",
          },
          data: { $push: "$$ROOT" },}},
      { $sort: { "data._id": -1 } },
    ]).toArray();
```

To find the most popular genres in each year, there needs to be a grouping based on the genres and year, but given as our data for the movies contained a array of "genres" and not individual genre Strings, we need to unwind the array and create a document with the same movie but for each genre in the array. Then we project the release data to only retrieve the year, and genre and popularity. After that we start grouping by year and genre so that we can calculate the average popularity of that genre in that year. Then we need to do a comparison between the genres of each year to find the most popular one.

Since "group" aggregations don't allow to keep data other than the accumulator, we needed to store the members of the new year-based groups, so we use "push: ROOT" and end up with an array of objects containing the previously generated documents.

Since the final object structure is complex, it undergoes manipulation in the function before returning it.

Get "more" reviews of a movie

```
db.collection("reviews").aggregate([
    { $match: { movieId: movieId } },
    {$project: {
        review_date: { $toDate: "$review_date" },movieId: 1,title: 1,
 reviewer: 1,rating: 1,review_detail: 1,review_summary: 1,userId: 1,
      }},
    { $sort: { date: -1 } },
  ]).skip(20).toArray();
```

Normally, retrieving a movie's review is done in a simple find(), since we keep 20 most recent review embedded in the movie document. However, a user may request to load more reviews and therefore, we need to fetch the "next"

20 most recent reviews from the reviews collection.

We match by movie ID in the reviews collection and project the fields we need. Then we sort the documents by date and skip the first 20 as they are already embedded.

Dynamic filtering for movies

```
    var query = { $and: [] };

    if (title) { query.$and.push({ title: { $regex: title, $options: "i" } });}
    if (genre) { query.$and.push({ genres: { $regex: genre, $options: "i" } });}
    if (minRuntime) { query.$and.push({ runtime: { $gte: minRuntime } });}
    if (maxRuntime) { query.$and.push({ runtime: { $lte: maxRuntime } });}
    if (minDate) { query.$and.push({ release_date: { $gte: new Date(minDate) } });}
    if (maxDate) { query.$and.push({ release_date: { $lte: new Date(maxDate) } });}
    if (language) { query.$and.push({spoken_languages: { $regex: language, $options: "i" },});}

    let db = await mongoDriver.mongo();
    const movies = await db.collection("movies").find(query, {
        projection: { title: 1, poster_path: 1, vote_average: 1, overview: 1 },
    }).toArray();
```

This query allows us to filter the search for movies based on the content of a form input sent by the front end, where the user can use many combinations of filters. We check if each attribute is not empty, then push it into the find() query, then send back an array of matching movies.

Write Operations: Here is a selection of Write operation implementations in MongoDB. In this section we will only include Creating a user, Deleting a movie, and Creating a review for the sake of conciseness.

Create User/Register: Anonymous users

We start the query by checking if the user exists in the users collection, with

the assumption that if it is found in MongoDB it will also be in Neo4j.

```
    // Check if user exists already by username and email
    if (exists) {
      return res.status(409).json({ message: "User already exist." });
    } else {
      // Hash password
      // Create user document on Mongo
      // Calculate a new _id for the user
      // Create user object
      try {
        var newUser = {
            ...set the attributes
        };

        //Consistency management
        try {
          await db.collection("users").insertOne(newUser);
        } catch (e) {
          return res.status(500).json({ message: "User Document not
    Created" });
        }
        //Create the user node in Neo4j
        const node = await userController.createUser(username, parseInt(
    newId));

        //No error was thrown
        return res
          .status(201)
          .json({ User_Document: newUser, User_Node: node });
      } catch (err) {
        //Delete the user that was only inserted in MongoDB and not Neo4j
        await userController.backtrackDelete(newId);
        return res.status(500).json({ message: err.message });
     }}}
```

Delete a movie: Admin privileges

```
    try {
        let db = await mongoDriver.mongo();
        //Store the movie to be deleted for backtracking
        movie_to_be_deleted = await db.collection("movies").findOne({ _id:
     movieId });

```

```
        try {
          await db.collection("movies").deleteOne({ _id: movieId });
        } catch (err) {
          return res.status(400).json({ message: "Mongo: Movie Deletion Failed" });
        }
        //Delete the movie node in Neo4j
        await deleteMovie(movieId);
        //No errors were caught => delete the movie reviews from the review collection
        await reviewController.deleteAllMovieReviews(movieId);
        res.status(200).json({ message: "Task executed successfully" });
    } catch (err) {
        try {
        //If any of the previous steps fail, reinsert the movie document in Mongo DB
          await backtrackInsert(movie_to_be_deleted);
        } catch (err) {
          res.status(400).json({ message: err.message });}}
```

Create a review: Registered users

```
    // Calculate new ID for the review
    newReview = {
      ... set attributes of the review object};

    let movie = await db.collection("movies").findOne({ _id: movieId });

    // If the embedded array is at max length => push review at the top and shift(delete from array copy) the least recent review
    if (movie["reviews"].length >= 20) {
      new_reviews = movie["reviews"]
      new_reviews.push(newReview)
      new_reviews.shift();
      console.log(new_reviews)
    } else {
      new_reviews = movie["reviews"]
      new_reviews.push(newReview);
    }

    // Insert the review in MongoDB
    try {
      await db.collection("reviews").insertOne(newReview);
    }catch (e){
```

```
22      return res.status(400).json({ message: "Review Insertion
        unsuccessful" });
23    }
24
25    // Calculate the new vote_average and vote_count of the movie
26    const new_vote = await updateMovieRatingCreation(movie, newReview["
        rating"]);
27
28    try {
29    //update the movie node on Neo4j with the new average_vote
30      await updateVoteMovieNode(movieId, (new_vote.vote_average).toFixed
        (1));
31      //No errors caught => update the movie document with the new
        embedded array, the vote_count and the vote_average
32        await db.collection("movies").updateOne({_id: movieId}, {
33          $set: {
34            reviews: new_reviews,
35            vote_average: new_vote.vote_average,
36            vote_count: new_vote.vote_count}});
37    }catch(e){
38      throw new Error ("Update movie info unsuccessful")
39    }
40    res.status(201).json({ review: newReview, message: "Review Added
        successfully" });
41  } catch (err) {
42  //Delete the new review from the reviews collection and log error
43    let db = await mongoDriver.mongo();
44    await db.collection("reviews").deleteOne({_id: newReview._id});
45    res.status(400).json({ message: err.message });
46  }
```

The grapDB part is implemented using the neo4j grapDB. It is well known for its index-free adjacency technique which is cheaper and more efficient than doing the same task with foreign keys as in RDBMS or indexes, because query times are proportional to the amount of the graph searched, rather than increasing with the overall size of the data [2]. *Cypher*, which is a declarative language, is the the de facto quering language for the neo4j GrapDB.

4.1 Main Queries in the GraphDB

In the neo4j database there are three entities/nodes : **User**, **Watchlist** and **Movie**

4.1.1 User nodes

Creating a user node.Upon the the time the user registers to the the system a user node is added to the graph database. Here is a code snippet from the neo4j Desktop.

```
    match(u:User)
    with u ORDER BY u.user_id DESC LIMIT 1
    create(ux:User {user_id:u.user_id+1,username:"${username}"})
    return {user_id:ux.user_id,username:ux.username}‘
```

Delete a user node: When a user node is removed from a database all his/her watchlists will also be removed from the Graph database.

```
MATCH(u:User{user_id:${user_id}})-[:CREATE]->(w:Watchlist)
DETACH DELETE u,w
```

Listing 6: Deleting a user node

Adding a follow relation: As we are going to build a movie review social application the user has the all the capabilities to follow other users in the social network. When the user wants to do so the FOLLOW relation is created as below. In our application a user can not follow him/her self. In the following listing a sample code that depicts the user-follow-user relation in the GraphDB.

```
match(u:User{user_id:${userId}})
match(ux:User{user_id:${followedUserId}})
where u.user_id<>ux.user_id and NOT (u)-[:FOLLOW]-(ux)
create (u)-[r:FOLLOW]->(ux)
return 'relationship created'
```

Listing 7: Creating User FOLLOW User relation

Deleting the follow relation: from above we have seen that a user can follow an other user,on the other hand a user can decide to stop following an other user.In line number 3 of the listing 7 before a relation is created the, a boolean check is done if there is a follows relation between those users.

```
1 MATCH(u:User{user_id:${user_id}-[f:FOLLOW]->(ux:User{user_id:${user_id}})
2 WHERE u.user_id<>ux.user_id
3 DELETE f
4 RETURN 'follows relationship deleted'
```

Listing 8: Deleting a user FOLLOW user relation

4.1.2 Movies Nodes

Create movie node: The administrator of the system can add a movie node to the database.The below snippet of code creates a movie node in the noe4j database.

```
1 MERGE (m:Movie{
2         movie_id:"${movie_id}",
3         title:"${title}",
4         poster_path:"${poster_path}",
5         vote_average:${vote_average},
6         overview:"${overview}"})
7         return m
```

Listing 9: Creating a movie node

Delete Movie Node: When a movie is deleted from the watchlist it belongs the relation that connect the movie with the watchlist and the node itself will be deleted from the database.Here is a snippet of a code that depicts the deletion of a movie node from the GraphDB.

```
1 MATCH(m:Movie{movie_id:"${movie_id}"})
2    DETACH  DELETE m
```

Listing 10: Delete Movie node

4.1.3 Watchlist nodes

Create Watchlist

```
MATCH (u:User{user_id:${user_id}})
CREATE(w:Watchlist{name:"${name}",created_date:date()})
CREATE (u)-[:CREATE]->(w)
return {created_by:u.username,user_id:u.user_id,watchlist_name:w.name,id:
    ID(w),created_date:w.created_date}
```

Listing 11: Creating a watchlist node

Update Watchlist

```
match(w:Watchlist)<-[:CREATE]-(u:User)
    where ID(w)=${watchlist_ID} AND u.user_id=${user_id}
    SET w.name="${name}"
    return {name:w.name,id:ID(w),created_date:w.created_date}
```

Listing 12: Updating a watchlist node

Delete watchlist

```
    Match(w:Watchlist)<-[c:CREATE]-(u:User)
    where ID(w)=${watchlistl_ID} and (u.user_id=${user_id} or true)
    detach delete  w
```

4.1.4 Recommendations/Suggestions in the GraphDB

Recommended Users

```
MATCH (u1:User{user_id:${userId}})-[f1:FOLLOW]->(wl:Watchlist)<-[f2:
   FOLLOW]-(ux:User)
    where  NOT (u1)-[:FOLLOW]->(ux)
    return {user_name:ux.username,user_id:ux.user_id} LIMIT 7
```

Listing 13: Recommended Users

```
  try {
```

```
    let session = neo4jdbconnection.session();
    const suggestedMovies = await session.run(
      `MATCH(u:User{user_id:${userId}})-[:FOLLOW]->(ux:User)
      with *
      optional match(ux)-[:CREATE]->(w:Watchlist)<-[:BELONGS]-(m:Movie)
      with m,count(m) as num
      where num>=2
      return {title:m.title,poster_path:m.poster_path,overview:m.overview,
    vote_average:m.vote_average,id:m.movie_id}
      limit 7`
    );
    session.close();
    if (!suggestedMovies.records)
      return res.status(400).json({ message: `an error occured` });
    const result = suggestedMovies.records.map((e) => {
      return {
        ...e["_fields"][0],
        vote_average: e["_fields"][0]["vote_average"],
      };
    });

    res.status(200).json(result);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
```

Listing 14: Recommended Users Express JS Method

Recommended Movies

```
MATCH(u:User{user_id:${userId}})-[:FOLLOW]->(ux:User)
      with *
      optional match(ux)-[:CREATE]->(w:Watchlist)<-[:BELONGS]-(m:Movie)
      with m,count(m) as num
      where num>=2
      return {title:m.title,poster_path:m.poster_path,overview:m.overview,
    vote_average:m.vote_average,id:m.movie_id}
      limit 7
```

Listing 15: Recommended Movies

```
try {
    let session = neo4jdbconnection.session();
    const suggestedMovies = await session.run(
```

```
      `MATCH(u:User{user_id:${userId}})-[:FOLLOW]->(ux:User)
      with *
      optional match(ux)-[:CREATE]->(w:Watchlist)<-[:BELONGS]-(m:Movie)
      with m,count(m) as num
      where num>=2
      return {title:m.title,poster_path:m.poster_path,overview:m.overview,
    vote_average:m.vote_average,id:m.movie_id}
      limit 7`
    );
    session.close();
    if (!suggestedMovies.records)
      return res.status(400).json({ message: `an error occured` });
    const result = suggestedMovies.records.map((e) => {
      return {
        ...e["_fields"][0],
        vote_average: e["_fields"][0]["vote_average"],
      };
    });

    res.status(200).json(result);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
```

Listing 16: Recommended Movies Express JS Method

View suggested Watchlists that are from the users you follow.

```
\begin{lstlisting}[caption=Recommended Watchlists,]
 MATCH (u1:User{user_id:${userId}})-[f1:FOLLOW]->(ux:User)-[fl:FOLLOW]->(
    wl:Watchlist)
    where NOT (u1)-[:FOLLOW]->(wl) AND NOT (u1)-[:CREATE]->(wl)
    with wl
    match (ux:User)-[:CREATE]->(wl)
    return {name:wl.name,id:ID(wl),owner:ux.username,owner_id:ux.user_id,
    created_date:wl.created_date}
```

```
 try {
    let session = neo4jdbconnection.session();
    const suggestedWatchlists = await session.run(`
    MATCH (u1:User{user_id:${userId}})-[f1:FOLLOW]->(ux:User)-[fl:FOLLOW
    ]->(wl:Watchlist)
    where NOT (u1)-[:FOLLOW]->(wl) AND NOT (u1)-[:CREATE]->(wl)
```

```
    with wl
    match (ux:User)-[:CREATE]->(wl)
    return {name:wl.name,id:ID(wl),owner:ux.username,owner_id:ux.user_id,
    created_date:wl.created_date}`);
    session.close();
    if (!suggestedWatchlists.records)
      return res.status(400).json({ message: `an error occured` });
    const result = suggestedWatchlists.records.map((e) => {
      return {
        ...e["_fields"][0],
        id: e["_fields"][0].id.low,
        owner_id: e["_fields"][0].owner_id.low,
      };
    });

    res.status(200).json(result);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
```

Listing 17: Recommended Watchlists Express JS Method

View suggested user profiles that follow the same Watchlists you do.

```
//View suggested user profiles that follow the same Watch Lists you do.
//paramemters:user_id
MATCH (u1:User{user_id:${user_id}})-[f1:FOLLOW]->(wl:Watchlist)<-[f2:
    FOLLOW]-(ux:User)
where  NOT (u1)-[:FOLLOW]->(ux)
return {user_name:ux.username,user_id:ux.user_id} LIMIT 7

```

View suggested movies based on their occurrence on the watchlists of users you follow.

```
    //View suggested movies based on their occurrence on the watchlists of
     users you follow.
// parameters : user_id
MATCH(u:User{user_id:${user_id}})-[:FOLLOW]->(ux:User)
with *
optional match(ux)-[:CREATE]->(w:Watchlist)<-[:BELONGS]-(m:Movie)
with m,count(m) as num
where num>=2
```

```
return {title:m.title,poster_path:m.poster_path,overview:m.overview,
    vote_average:m.vote_average,id:m.movie_id}
limit 7

```

Retrieve a user's own watchlist.

```
   try {
    let session = neo4jdbconnection.session();
    const suggestedUsers = await session.run(
      `MATCH (u1:User{user_id:${userId}})-[f1:FOLLOW]->(wl:Watchlist)<-[f2
    :FOLLOW]-(ux:User)
    where  NOT (u1)-[:FOLLOW]->(ux)
    return {user_name:ux.username,user_id:ux.user_id} LIMIT 7`
    );
    if (!suggestedUsers.records)
      return res.status(400).json({ message: `an error occured` });
    const result = suggestedUsers.records.map((e) => {
      return {
        ...e["_fields"][0],
        user_id: e["_fields"][0].user_id.low,
      };
    });

    res.status(200).json(result);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
```

Find the most active users based on the number of followers of their watchlists.

```
 MATCH (u1:User)-[f1]->(w:Watchlist)<-[f2]-(u2:User)
with *
WHERE
type (f1) in ["CREATE"]AND
type (f2) in ["FOLLOW"]AND
NOT (u1)-[:FOLLOW]-(w) AND
NOT (u2)-[:CREATE]-(w)
RETURN u1.user_name AS YOU, w.watchlist_name AS TitleOfYourWatchlist,
COLLECT(u2.user_name)AS WatchlistFollowers, count(f2) AS NumFollowers
ORDER BY NumFollowers
```

```
DESC
LIMIT 10
```

4.1.5 Summary of Queries analysis for the GraphDB

Read Operations		
Operation	Expected Frequency	Cost
Most followed Watchlists	Medium	Medium (Multiple reads)
Most Active Users based on their number of followers of their watchlist	Low	High
Suggested Movies based on their occurrence on thewatchlists of users you follow	Low	High
Suggested Watchlists from the users that you follow	Average	Medium
Suggested Users	Average	Medium (Multiple reads)
Retrive a user's own watchlists	High	Low

Table 3: Read Operations in the GraphDB

Write Operations		
Operation	Expected Frequency	Cost
Insert/update/Remove a Watchlist	Low	Low
Update User information	Low	Low

Table 4: Write Operations in the GraphDB

References

[1] https://www.mongodb.com/docs/manual/introduction/, "Mongo db documentation," 2022.

[2] N. I. Joy Chao, “Getting started guide for neo4j,” April, 2022.