UNIVERSITÀ DI PISA

Large Scale and Multi-Structured Database

Anime Advisor

Aniello Giuseppe

De Filomeno Elisa

Secondulfo Valerio

Index

About the project

Introduction

Anime Advisor is an application that lets users to find an high amount of Animes (which is the name given for japan's animations products) of all kind and generes.

The animes are showed with numerous attributes, which can be seen in their personal page, such as the name of the anime, the type (if it's an anime made for Television, Cinema, a Music video, an OVA which is an original anime video made for the home video market, and many others), the source (if it's an Original product or it's a transposition of a manga comic or a book and so on), the year of premier, it's generes, the studio who made it along with the list of producers and licensors.

The user has the possibility to find an anime by name or to search by a list of the most rilevant generes for this kind of products.

An advanced search section is also offered to all the users. This section offers different statistical research on the anime's dataset like different top 10 of animes studio and producers based on their score or average score, number of elements producted by studio or producers and the most followed and reviewed animes.

A person who approach the application for the first time will have the possibility to create a personal account or to browse the various animes present without creating a profile.

A user which creates an account will have also access to numerous functionality that an unregistered user won't have.

A register user will be able, in fact, to give his personal opinion of an anime product by voting it,leaving a review and follow an anime showing the other users of the community his preferences and opinions. Also the user will have the possibility to follow other users, check their favourite animes and reviews made in order to make new friends and discover new animes.

Besides normal users there are a special class of users called Admin which have the duty to manage the animes data stored in the database and he social side too.

Functional requirements

An **unregistered** user can

- Log in
- Browse an anime by name or keywords
- Browse an anime by genre
- Access to the advanced search menu

A **registered** user can

- Do all the actions of an unregistered user
- Log out
- Follow or unfollow an anime
- Review and edit his own review of an anime
- Search other users
- Follow or unfollow other users
- See the most active users (the users who made the highest number of reviews)
- See the profile of a followed user, his reviews and the animes that he or she's following

Non-functional requirements

- **Data Consistency**: All the users must see the last version of the data and the update operations must be performed in the same order in which they are issued.
- **Performance**: A low response time is necessary in order to give to the users the informations requested in the lowest time possible.
- **Reliability** : The application must never crash, all errors and critical points must be managed with exceptions.
- **Usability**: The application's interface has to be user-friendly and intuitive for the user.

Working hypothesis & assumptions

During the designing phase of the project we came up to the following decisions:

- A user can't change his nikcname.
- Two reviews on the same anime can't have the same title.
- A user can't change his vote for an anime.

Those decisions are based on the fact that we wanted to focus on the main functionalities of the application and discard the options that would affect both the performance of the application itself and it's simplicity.

Changing a user's nickname infact, would have required to change the value in all the anime in which the user inserted a vote and that would have affected the performance of the application.

We decided to have a unique title for each review both for having an easy way to retrieve the single reviews and also to make each review a unique representation of the author's thoughts about the anime.

The decision of not allowing the user to change his vote was taken because we wanted to represent a vision of the user about the anime which reflected his perception of it in that specific moment. Allowing the changing of the vote along the time could have affected the significance of the score of the anime during the years.

Suggestions

Regarding the social aspect of our application we though about different levels of suggestion for Animes and Users to Like/follow:

- **SUGGESTED ANIME**

Users can see a list of max 10 suggested animes.

- An anime is Very Suggested if it's both liked and reviewed by a followed user.

- If it's not reached a number of 10, then in the suggestion list also appears animes which are only liked by followed users even if they didn't post any review on them.

- if at this time it's still not reached the number of 10 then in the suggestion list appears animes which are only reviewed by followed users.

- **SUGGESTED USERS**

Users can see a list of max 10 suggested users to follow.

- We decided to give priority to users who are followed by a followed user.

- If it's not reached a number of 10, then in the suggestion list also appears users who are followed by the users of the first priority.

Actors and use case

Actors

The application authors are three:

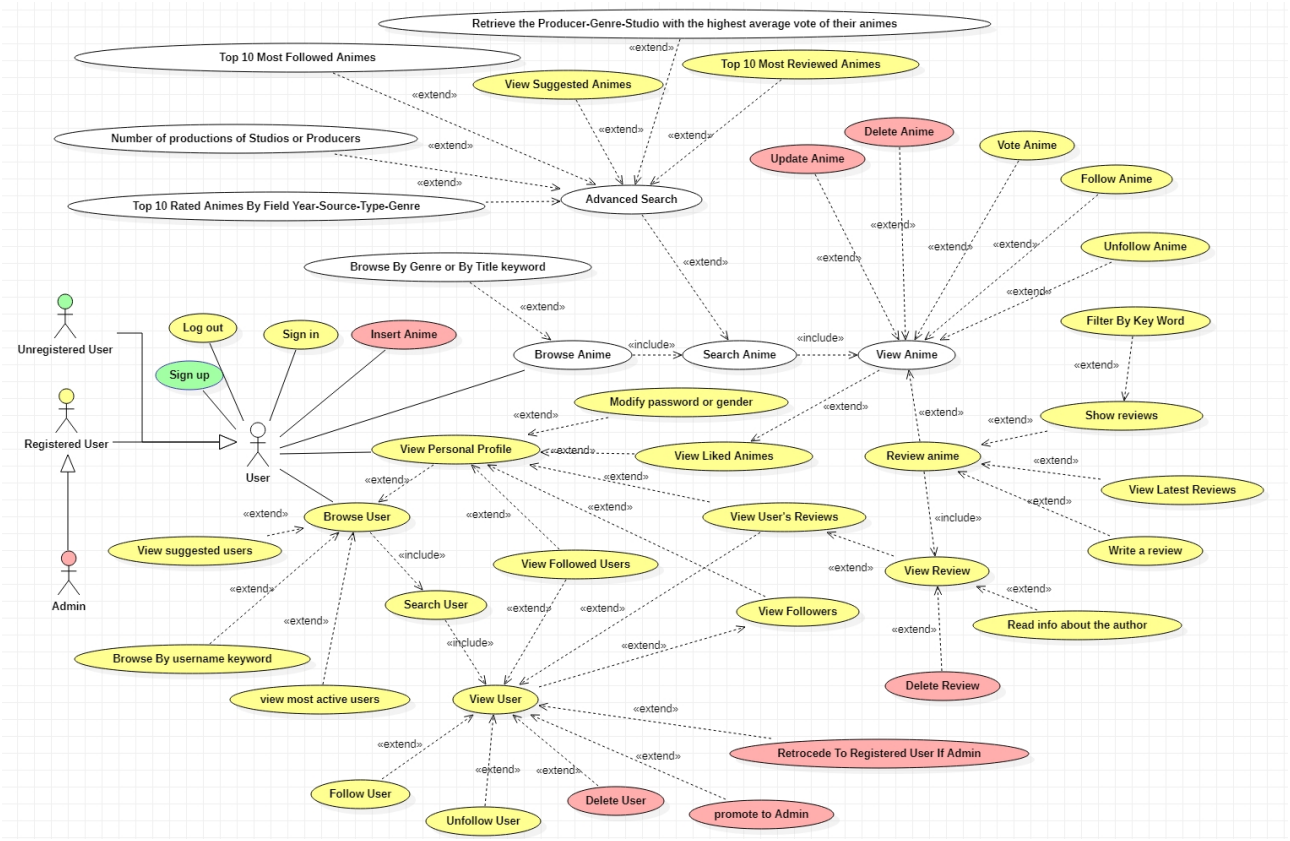
Unregistered user: this type of actor has limited functions. He can only use the browse menu by searching animes by name or genre or do an advance search. He can create a new account or log in to have access to all the other functionalities.

Registered User: A registered user can do all the actions of an unregistered user and also vote an anime and leave a review. He also have access to the social functionalities of the application in which he can follow an anime, search and follow a user. He also can see the animes followed and the reviews made by a user which the user is following.

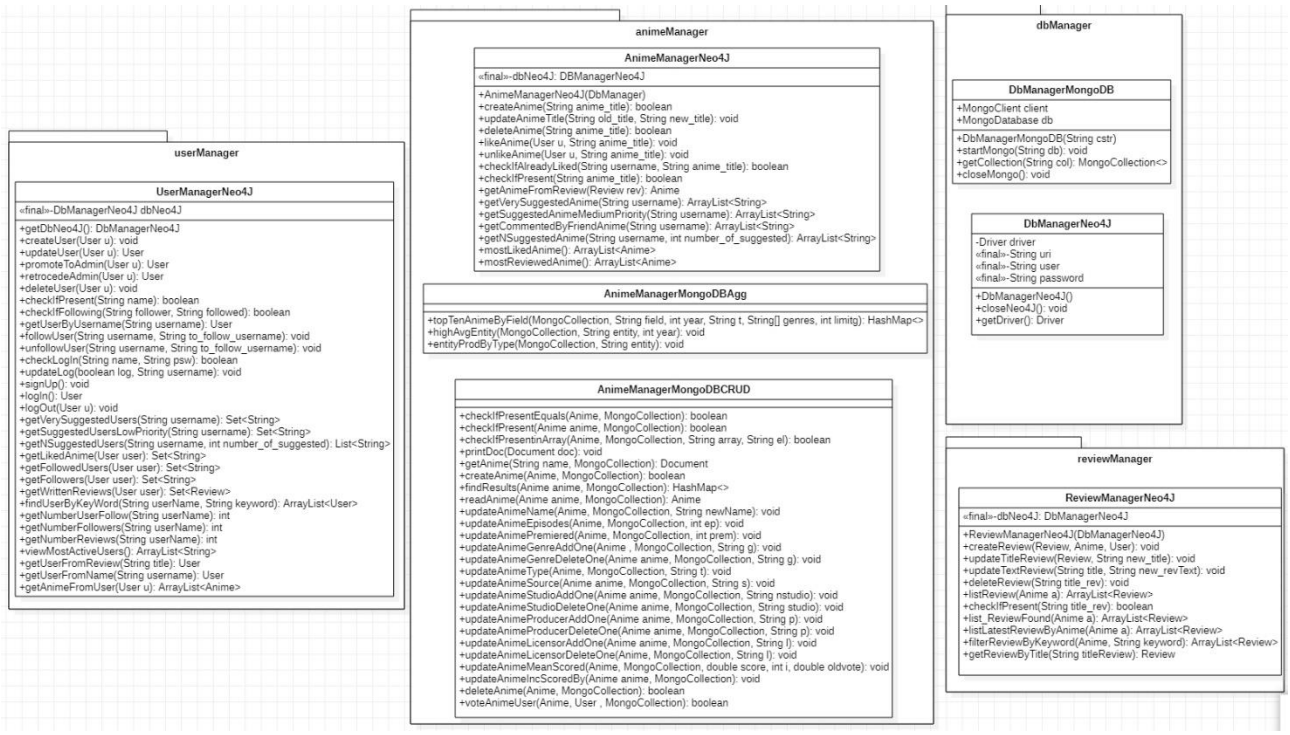
Admin: He can perform all the actions of a Registerer user. An admin also have the responsibility of the management of the anime and social sections.

An admin can insert, update or delete an anime and, regarding the social section, he can delete a user, promote him to admin or demote an admin to a normal user and delete reviews.

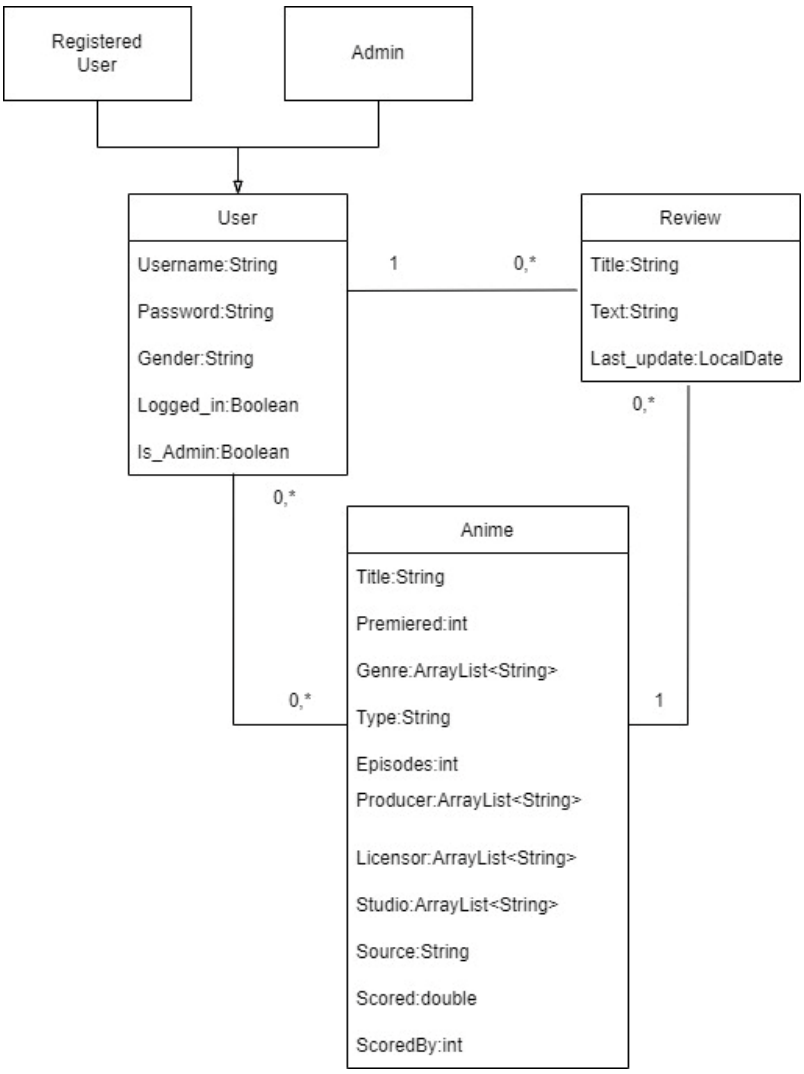
Use case



Class diagram



UML class diagram



Architectural design

The main language used for the development of the Anime Advisor application is JAVA.

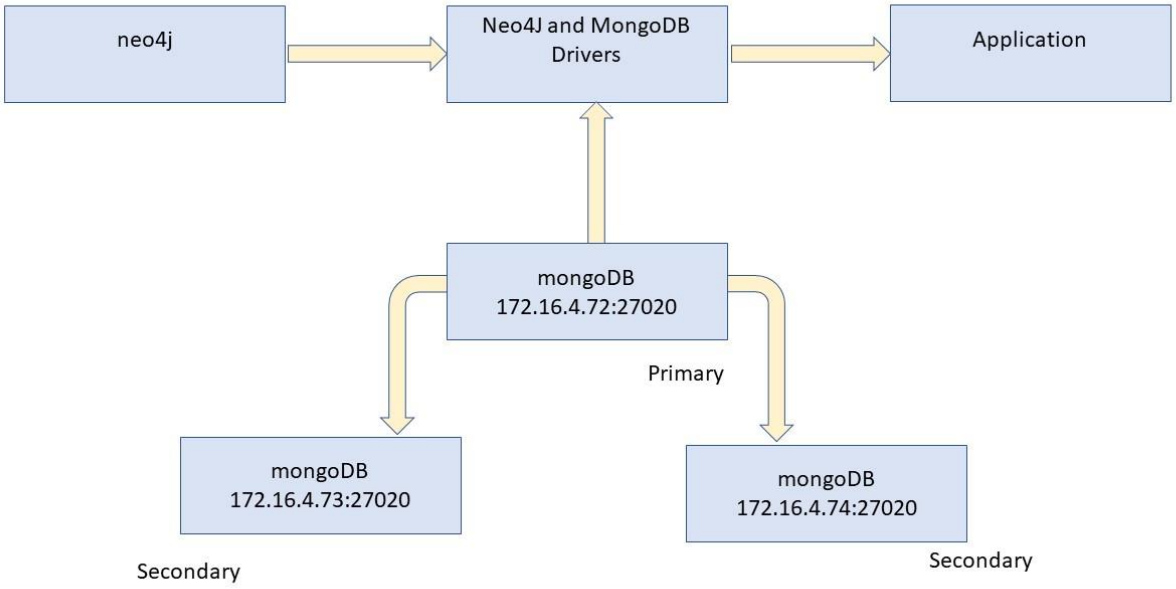
- **Client side**

  The front end module consist in a simple command line interface (CLI) which prints the results on screen and take inputs from keyboard. The interface is designed to be the simplest and clearest possible for the user. The inputs are numerical values for choosing answers like “yes”, “no”, “go back” and different options by picking the associated value.

  To connect the client with the servers, we implemented a driver to interface with MongoDB and one to interface with Neo4j.

- **Server side**

The server side consists in a cluster of three virtual machines provided to us by the University of Pisa. Those virtual machines have been used to host a sharded MongoDB cluster.



Data modelling

Datasets used

We used several datasets in order to populate the databases.

Anime

For the Anime data we took a collection of csv files from Kaggle scraped from two different websites which contain lots of animation products with many informations.

Dataset 1:

Link: https://www.kaggle.com/vishalmane10/anime-dataset-2022

Source: https://www.anime-planet.com/

Dimension: 13,6 MB

Dataset 2:

Link: https://www.kaggle.com/oksanakalytenko/anime-recommendations/data

Source: https://myanimelist.net/

Dimension: 1,95MB

Both files contains the following informations about the anime:

- **Name**: the name of the anime. Type: String
- **Genre**: an Array list of the genres of the anime
- **Premiered**: the year in which the anime was premiered
- **Episodes**: the number of episodes of the anime
- **Source**: the source of the anime. It specifies if it's an original product,it comes from a Manga book, it's based on a novel and so on.
- **Type**: the type of the anime. It specifies if it's a movie, a series, a music video, an OVA (a product made for the home video) and so on.
- **Studio**: an Array list of the studios that worked on the realization of the anime.
- **Producer**: an Array list of the anime's producers.
- **Licensor**: an Array list of the anime's licensors.
- **Scored**: the score totalized by summing the scores of the various users.
- **ScoredBy**: the number of users who scored that anime.

- **Members**: the number of followers of that anime.

For our project we left the stock values of Score and ScoredBy in order to have already some values for testing the calculation of the scores and the avereges.

User

Link: https://www.kaggle.com/marlesson/myanimelist-dataset-animes-profiles-reviews?select=profiles.csv
Source: https://myanimelist.net/
Dimension: 8.66 MB

The file contains the following informations about the user:

- **Profile**: the nickname of the user.
- **Gender**: the gender of the user.
- **Birthday**: the birthday of the user.
- **Favourite_animes**: an array list containing the ID of the animes followed by the user.
- **Links**: the link to the user profile on MyAnimeList.net .

For this dataset we chose to keep only the Profile name of the users and the gender data.

The missing values of Gender were filled randomly choosing between Male and Female.
In the final version of the dataset we opted for a third option: “Not specified”.

Review

Link: https://www.kaggle.com/marlesson/myanimelist-dataset-animes-profiles-reviews?select=reviews.csv

Source: https://myanimelist.net/

Dimension: 654MB

The file contains the following informations about the user:

- **uid**: the user’s id.
- **Profile**: the nickname of the user.
- **Anime_uid**: the nickname of the anime.
- **Text**: the review text.
- **Score**: the score given to the anime by the user.
- **Scores**: several scores given by the users based on some values.
- **Link**: the link to the anime’s page.

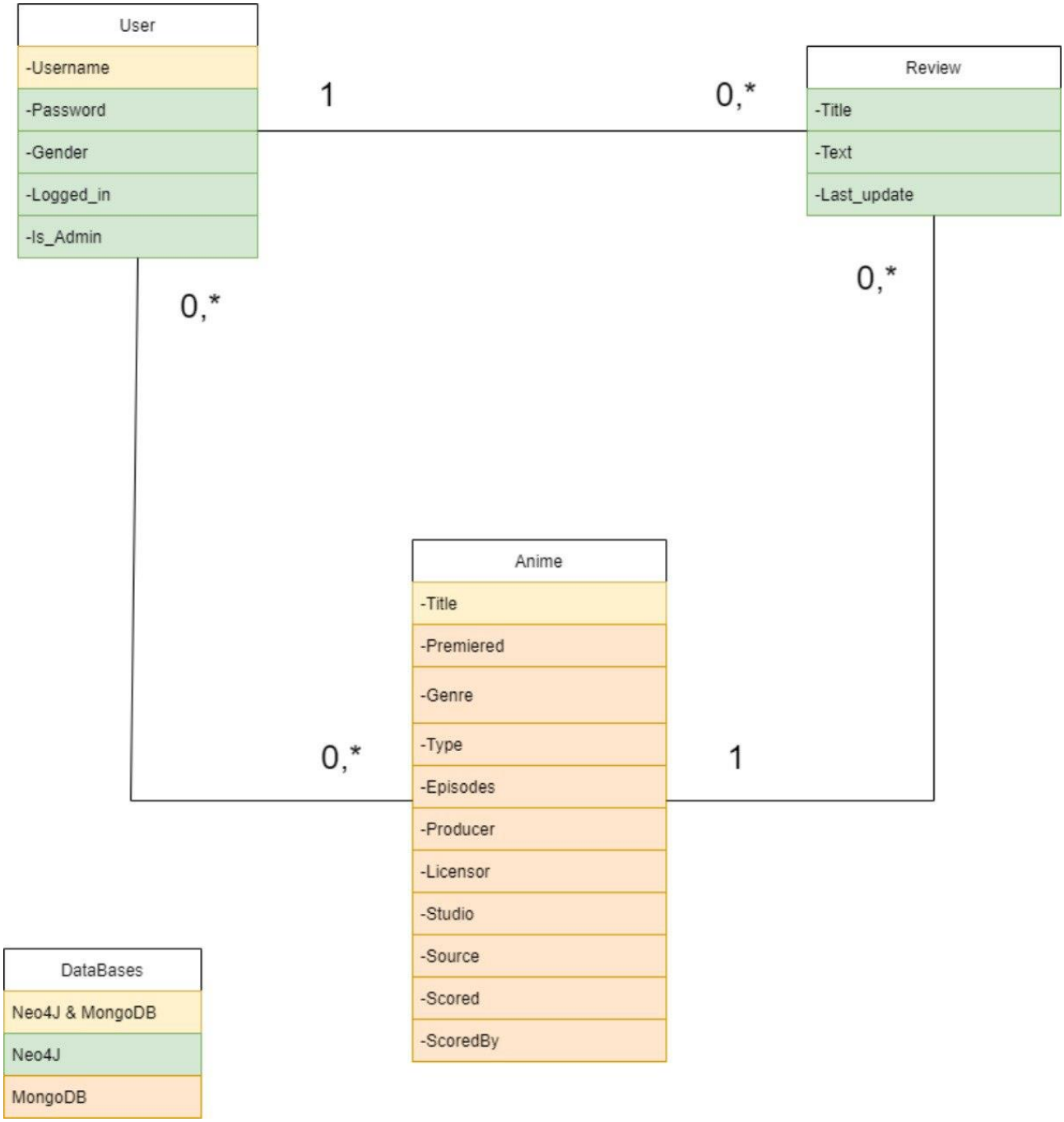
For this dataset we choosed only to keep Profile, Text. We modified the field Anime_uid by changing it’s values with the anime title which every id was referring to and inserted a new field Title where the users will put a unique title for the review.

Data among databases

The anime dataset has been stored and managed using MongoDB.

Due to it's fastness in retriving data informations from documents, we stored the anime's dataset in it.

Users, reviews and a portion (only the name) of anime datasets have been stored and managed using Neo4J.



Databases used

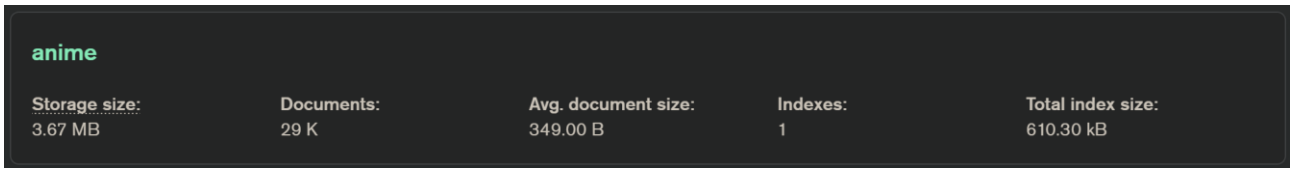
MongoDB

Introduction

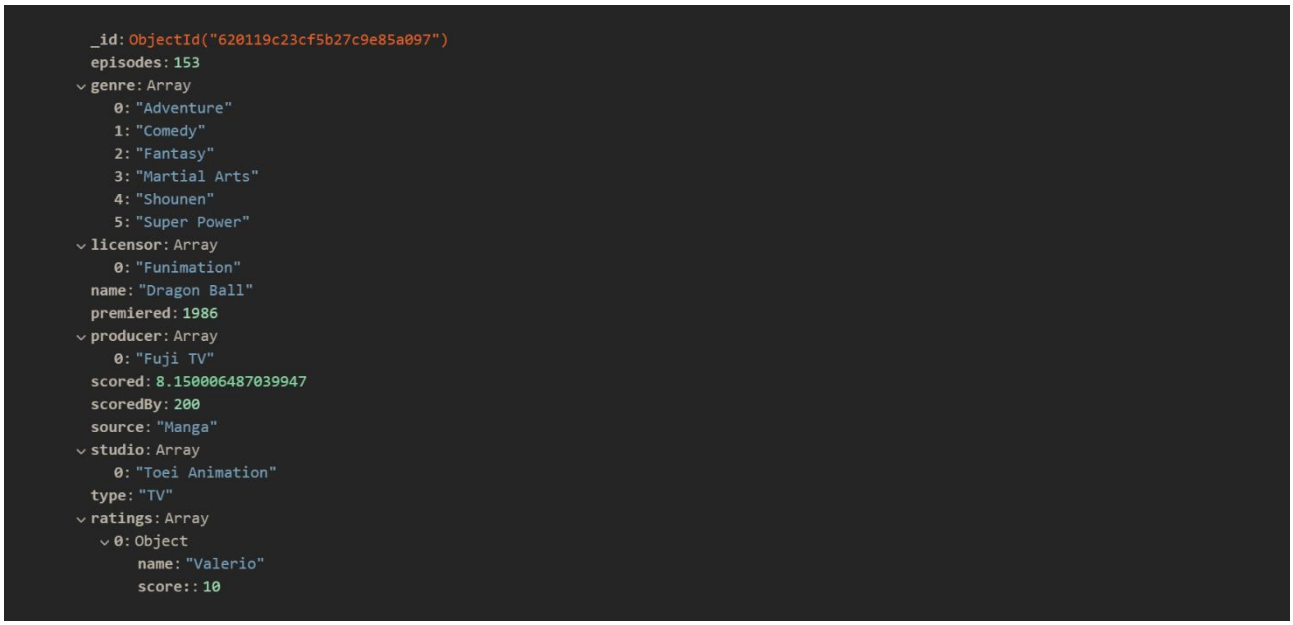
MongoDB is the databased chosen to store and manage the anime dataset.

The decision was taken due to it's high performance in reading documents, in order to let the user to retrieve more specific informations about the animes based on statistical researches using Mongo's aggregation queries. Also we exploited the ability of the JSON file to contain embedded documents to store the name of the users who have voted an anime (along with the vote) in order to keep track of who had already voted and who not.

Document organization and structure



Besides the _id generated by MongoDB the JSON file is structured in the following way:



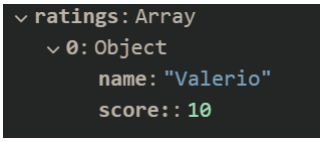
Each document contains a distinct anime. The fields' type are specified used are descripted in the Dataset-Anime section.

The fields contained in the JSON file are the same specified in the Dataset-Anime section except

- **Members**
- **Ratings**

We removed the members count for each anime because the “Followers” section has been assigned to Neo4J so there was no point to keep track of it also in Mongo.

Ratings instead contains a small portion of the User’s dataset (the name of the user) and the vote assigned to the anime in order to keep track of the users who have already voted the product.



The name of every document in unique and can be update only by an admin if he must.

Queries analysis and implementation

We implemented several queries for the animes in MongoDB, divided in CRUD (Create,Read,Update,Delete) operations and aggregations.

The aggregations have been used in order to retrieve some interesting informations and statistic of animes and some of theire fields.

Aggregations queries:

Top 10 animes by field:

With this query a user can find the top 10 animes by the specified field based on the ratings of the Animes presents in the field.

The field can be:

- **Year:** the user can choose the year in which he wants to calculate the top 10

```
db.anime.aggregate([
{$match:{premiered:1999}},
{$sort:{scored:-1}},
{$limit:10}
  ])
```

- **Source:** the user can choose the type of source for which to calculate the top 10
  The sources chosen for the user to select are:
- **Type:** the user can choose the type of the anime for which to calculate the top 10
- **Genre:** the user can choose the type of genre (one or more genres) for which to calculate the top 10

Top 10 Studio, Genre or Producer with the highest average on a specified year

With this query the user can search which is the studio, the genre or the producer presents having the highest average calculated by the sum of the anime belonging to the field chosen on a specified year.

```
> db.anime.aggregate([
    {$match:{"premiered":1985}},
    {$unwind:"$genre"},
    {$group:{_id:"$genre",averageScore:{$avg:{$sum:"$scored"}}}},
    {$sort:{averageScore:-1}},
    {$limit:10}

    ])
```

In this case we can change the year of premier and the field to search for in order to have the various results.

Number of productions of an entity by type

This query retrieves all the entities, which can be Studios or Producers, and the number of their productions by type.

We chose to don't consider the Licensors for this query because it could non be so interesting for most of the users doing researches by Licensors.

```
Anime_Advisor > db.anime.aggregate([
                  {$match:{"studio":{$ne:"Unknown"}}},
                  {$unwind:"$studio"},
                  {$group:{"_id":{studio:"$studio",type:"$type"},count:{$sum:1}}},
                  {$sort:{count:-1}}


                  ])
```

Use of indexes

In order to choose if and where put one or more indexes in the Anime database we execute different tests using the three main aggregation's queries.

For this kind of tests we used the same queries reported in the previous images.

Top 10 animes by field:

Index on: **Score**

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	10	28906	18
WITHOUT	10	28906	19

Top 10 Studio, Genre or Producer with the highest average on a specified year

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	641	28906	21
WITHOUT	641	28906	21

Number of productions of an entity by type

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	21915	28906	71
WITHOUT	21915	28906	72

Index on: **Name**

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	10	28906	19
WITHOUT	10	28906	18

Top 10 Studio, Genre or Producer with the highest average on a specified year

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	641	28906	20
WITHOUT	641	28906	20

Number of productions of an entity by type

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	21915	28906	72
WITHOUT	21915	28906	73

Index on: **Genre**

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	10	28906	19
WITHOUT	10	28906	18

Top 10 Studio, Genre or Producer with the highest average on a specified year

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	641	28906	26
WITHOUT	641	28906	27

Number of productions of an entity by type

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	21915	28906	79
WITHOUT	21915	28906	71

Index on: **Year**

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	10	28906	13
WITHOUT	10	28906	18

Top 10 Studio, Genre or Producer with the highest average on a specified year

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	641	28906	4
WITHOUT	641	28906	21

Number of productions of an entity by type

INDEX	DOCUMENT RETURNED	DOCUMENTS EXAMINED	EXECUTION TIME(ms)
WITH	21915	28906	84
WITHOUT	21915	28906	78

Due to the results obtained we choose to leave only 1 index on the **Year** attribute because from the results taken in the tables we can see that it's the field where we have the best performances with the index on.

MongoDB replicas

Replica copies of the same data in different servers were used in order to ensure Availability and Consistency. With this cluster of virtual machines we ensured that the user could always have access to the application.

We used three different virtual machines, provided by the University of Pisa, composed by a primary server and two secondary ones. The secondary servers keep replicas of the primary server's data and they can communicate with each other.

VM	IP address	Port	OS
Replica-0	172.16.4.72	27020	Ubuntu 18.04
Replica-1	172.16.4.73	27020	Ubuntu 18.04
Replica-2	172.16.4.74	27020	Ubuntu 18.04

The virtual machine with IP address 172.16.4.72 has the highest priority so it has been elected as primary server, unless a problem arises.

In our application we preferred guaranteeing strict consistency of data so that we ensure that users and admins will always read updated data. We can consider our application as read oriented, with not many writings. In fact only an admin can add or delete a document in MongoDB and this operation is not much frequent. Also updates of documents are not as frequent as reading operations.

We have decided to set the configuration as follows:

```
"mongodb://172.16.4.72:27020,172.16.4.73:27020,172.16.4.74:27020/?r
etryWrites=true&w=3&wtimeoutMS=5000&readPreference=nearest"
```

- Write concern (w=3): wait for all the replicas to be updated
- Read preference (readPreference=nearest): read from the member of the replica set with the least network latency to have the fastest response.

MondoDB sharding

It has been decided to split the dataset into three shards with three replica sets for each shard in order to have both persistency against failures and high availability.

To do this we decided to use a partitional algorithm based on hash strategy.

As possible candidates for the sharding key we took in consideration three attributes of the Anime dataset:

Genre: We first though about using genre as a shard key, but in this case there could have been animes that had many genres in their informations and that could have create redundancy in case we replicated that anime for all the servers that contained part of the genres inside it. This option was then discarded

Year: We though about using the year of premierer of the anime but even in this case there would be probably less researches for old animes and that could have generate an high amount of interactions with the server containing the newest ones and many less interactions with the servers containing the less recent ones.

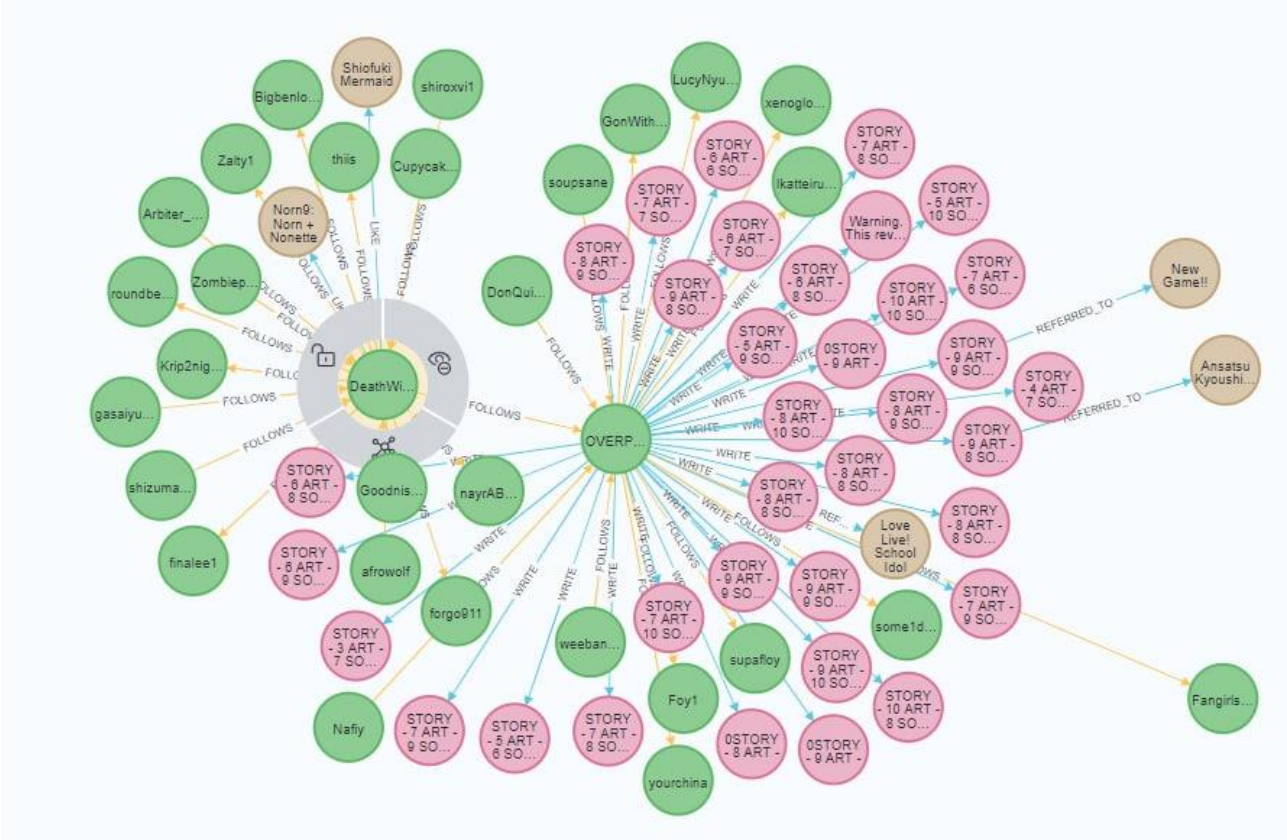
Name: At the end we decided to use the Name of the animes as shard key. Each shard will contain a range of anime's initial letters divided into three main chunks. This ensures a good cardinality

Neo4J

Introduction

Graph Database stores all the information needed to build the network infrastructure of the application. Anime, Friends, Reviews form a real Social Network, in which a Graph DB is optimized for, increasing the performance of our application. In fact we needed to find the best way to perform quickly the retrieving of a user's followers, their numbers, the ranking of the top 10 reviewed and liked Anime. Users can also view a list of suggested anime and users according to the animes liked and reviewed by their friends. All these queries are very fast in a Graph DB where is just needed to count the number of edges without wasting memory with join operations like in a relational model. We also decided to prioritize the performance issue and introduced a redundancy in the attribute title of Anime, which is already contained in the Document database.

Here's an example of a portion of our graph:



Graph design

NODES:

Three main types of nodes are used within the database:

- **User nodes:**
  these nodes represent the users registered within the application, having as attributes the ones specified in their account information (username, password, gender) and the ones useful to know if the user is and admin and has logged in the application.
    - logged_in = true -> the user has logged in the application
        - = false -> the user isn't logged in the application
    - is_admin = true -> the user ha admin privileges,
        - = false -> the user doesn't have admin privileges
- **Anime nodes:**
  These nodes represent all the anime stored within the application with only their title information, the rest of their attributes are stored in the relative document in MongoDB. The title is the portion of the anime dataset that has been replicated into the MongoDB database to fasten up queries.

- **Review nodes:**
  these nodes represent all the reviews stored within the application, having as attributes the title, the text of the review and the date in which the review has been written.

  Title: must be chosen at the creation of the review and must be unique among all reviews. During the population the title consisted of the first 28 characters of the text of the reviews.

  last_update: is the date in which the review has been last modified. If the review has not been modified then is the day in which it has been created. During the population this attribute is created randomly.

  RELATIONSHIPS:

  Four different kinds of relationships between nodes are used within the database:

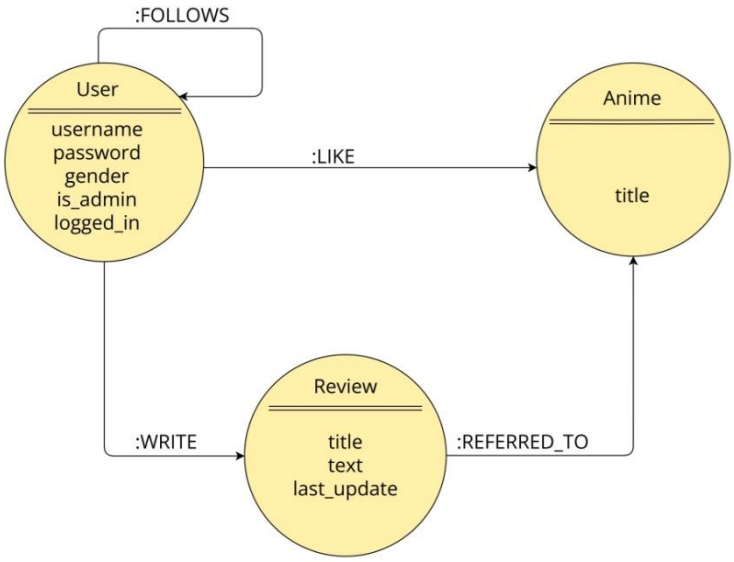
  • **User → FOLLOWS → User**, which represents a user following another in the application. This relationship has no attributes.

  • **User → LIKE → Anime**, which represents a user liking an anime of the application. This relationship has no attributes.

  • **User → WRITE → Review**, which represents a user writing a review in the application. This relationship has no attributes.

  • **Review → REFERRED_TO → Anime**, which represents which anime in the application is a review referred to. This relationship has no attributes.

The GraphDB's architecture is shown below:



Queries implementation

We report only the relevant queries of our application, CRUD operations and other basic operations have

been omitted.

COUNT FOLLOWING/LIKING

We implemented some simple statistics in order to let users see how many reviews they have written, how many followers they have and how many users they are following.

Admins can also see the top 10 most reviewed anime and a list of the top 10 anime most liked.

Queries with the same structure:

- How many followers does a specific user have?

- How many users is a specific user following?

- How many reviews has a specific user written?

Below is the implementation of the first query:

Domain-specific	**Graph-centric**
How many followers does a specific user have?	How many ingoing (:FOLLOWS) edges does the specific (u:User) node have?

CYPHER

```
1. MATCH p=(u:User{username:$user})<-[:FOLLOWS]-(u1:User)
2. RETURN COUNT(p) as n_follow
```

Other queries with the same structure:

- Which are the top 10 anime most liked?

- Which are the top 10 anime most reviewed?

We report the implementation of the second query:

Domain-specific	**Graph-centric**
Which are the top 10 anime most reviewed?	Which are the first 10 Anime nodes ordered by a descending order of the number of REFERRED_TO edges for each anime node?

CYPHER

```
1. MATCH (a:Anime)<-[t:REFERRED_TO]-(r:Review)
2. RETURN Count(t), a.title
3. ORDER BY COUNT(t) DESC LIMIT 10
```

SUGGESTED USERS AND ANIMES

These are the queries to get the suggested users and anime.

Users suggestions:

Domain-specific	**Graph-centric**
What are the suggested users for a specific user? (The users that are followed by a followed user of a followed user of the user token into consideration)	Which are the nodes that have exactly a distance of three (:FOLLOWS) ingoing edges from the node token into consideration and are of the same type (User)?

CYPHER

```
1. MATCH (u1:User)-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(u3:User)-[:FOLLOWS]->(u4:User)
2. WHERE NOT (u1)-[:FOLLOWS]->(u4) and u1<>u4 AND u1.username = $username
3. RETURN u4.username
```

Domain-specific	**Graph-centric**
What are the very suggested users for a specific user? (The users that are followed by a followed user of the user take into consideration)	What are the nodes that have an exact distance of two (Follow) ingoing hopes from the node take into consideration and are of the same type (User)?

CYPHER

```
1. MATCH (u1:User)-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(u3:User)
2. WHERE NOT (u1)-[:FOLLOWS]->(u3) and u1<>u3 AND u1.username = $username
3. RETURN u3.username
```

Anime suggestions:

Low Priority suggested anime:

Domain-specific	**Graph-centric**
What anime have been commented by a friend of a specific user? (The anime that have at least a review referred to them, written by a followed user of the user taken into consideration)	What are the Anime nodes that have an ingoing (:REFERRED_TO) edge from a Review node that has an ingoing (:WRITE) edge from a User node that has an ingoing (:FOLLOWS) edge from the node taken into consideration and don't have an ingoing (Follow) edge from the (User) nodes?

CYPHER

```
1. MATCH (u1:User)-[:FOLLOWS]->(u2:User)-[:WRITE]->(r:Review)-[:REFERRED_TO]->(a:Anime)
2. WHERE NOT (u1)-[:LIKE]->(a) AND u1.username = $username
3. RETURN a.title
```

Medium Priority suggested anime:

Domain-specific	**Graph-centric**
What are the suggested anime for a specific user? (The anime that are liked by a followed user of the user taken into consideration)	What are the Anime nodes that have exactly a distance of two (:FOLLOWS) ingoing edges from the User node taken into consideration?

CYPHER

```
1. MATCH (u1:User)-[:FOLLOWS]->(u2:User)-[:LIKE]->(a:Anime)
2. WHERE NOT (u1)-[:LIKE]->(a) AND u1.username = $username
3. RETURN a.title
```

High Priority suggested anime:

Domain-specific	**Graph-centric**
What are the very suggested anime for a specific user? (The anime that are liked and have a review referred to them by a followed user of the user taken into consideration)	What are the Anime nodes that have two ingoing (:FOLLOWS and :REFERRED_TO) edges from the same (u2:User) node that has an ingoing (:FOLLOWS) edge from the User node taken into consideration and no ingoing (:FOLLOWS) edges from that node?

CYPHER

```
1. MATCH (u1:User)-[:FOLLOWS]->(u2:User)-[:LIKE]->(a:Anime)<-[:REFERRED_TO]-(r:Review)<-[:WRITE]-(u2)
2. WHERE NOT (u1)-[:LIKE]->(a) AND u1.username = $username
3. RETURN a.title
```

TOP 10 MOST ACTIVE USERS

Domain-specific	**Graph-centric**
Who are the most active users? (Users who have written the greatest number of reviews)	Which are the user nodes with the greatest number of edges WRITE?

CYPHER

```
1. MATCH p=(u:User)-[t:WRITE]->(r:Review)
2. RETURN COUNT(t) AS n_rev, u.username
3. ORDER BY n_rev DESC LIMIT 10
```

FOLLOW/UNFOLLOW AND LIKE/UNLIKE

Queries with this structure:

- A user decides to follow/unfollow another user

- A user decides to like/unlike a specific anime

User likes Anime:

Domain-specific	**Graph-centric**
A specific user likes an anime.	Given two nodes (User and Anime), creates a (FOLLOWS) edge outgoing from the first (User) node and ingoing in the second (Anime) node.

CYPHER

```
1. MATCH (u:User)
2. WHERE u.username= $username
3. MATCH (a:Anime)
4. WHERE a.title=$title
5. MERGE (u)-[:LIKE]->(a)
```

User unlike Anime:

Domain-specific	**Graph-centric**
A specific user ulikes a specific anime.	Given two nodes (User and Anime), delete, if existing, the LIKE edge outgoing from the first (User) node and ingoing in the second (Anime) node.

CYPHER

```
1. MATCH p=(u:User{username:$username})-[r:LIKE]->(a:Anime{title:$title})
2. DELETE r
```

Implementation details

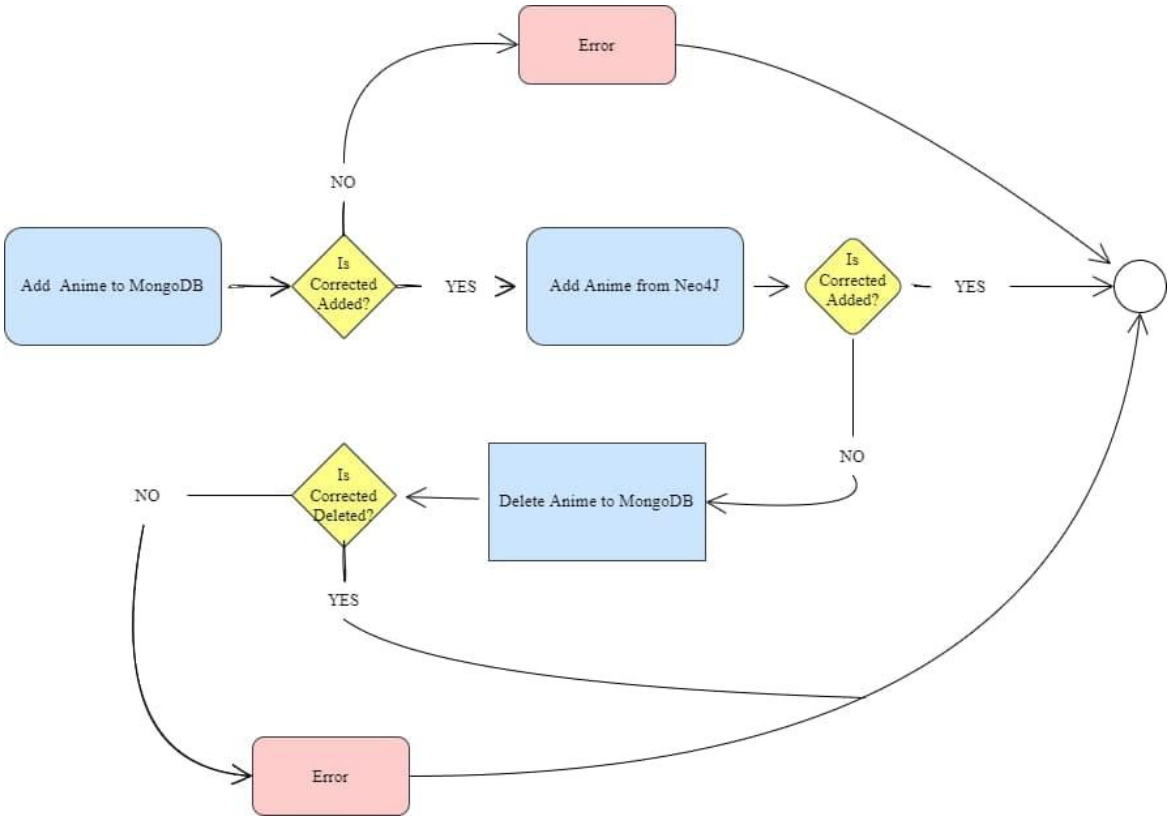
- During the population all the users in the dataset were considered normal registered users, only 3 users with Admin privileges were manually added.
- Before attempting any creation of a node or relation, it's always checked if it is already present in the graph to avoid duplicates.
- A view of the final dataset stored in the Neo4j database is depicted below:

Data consistency

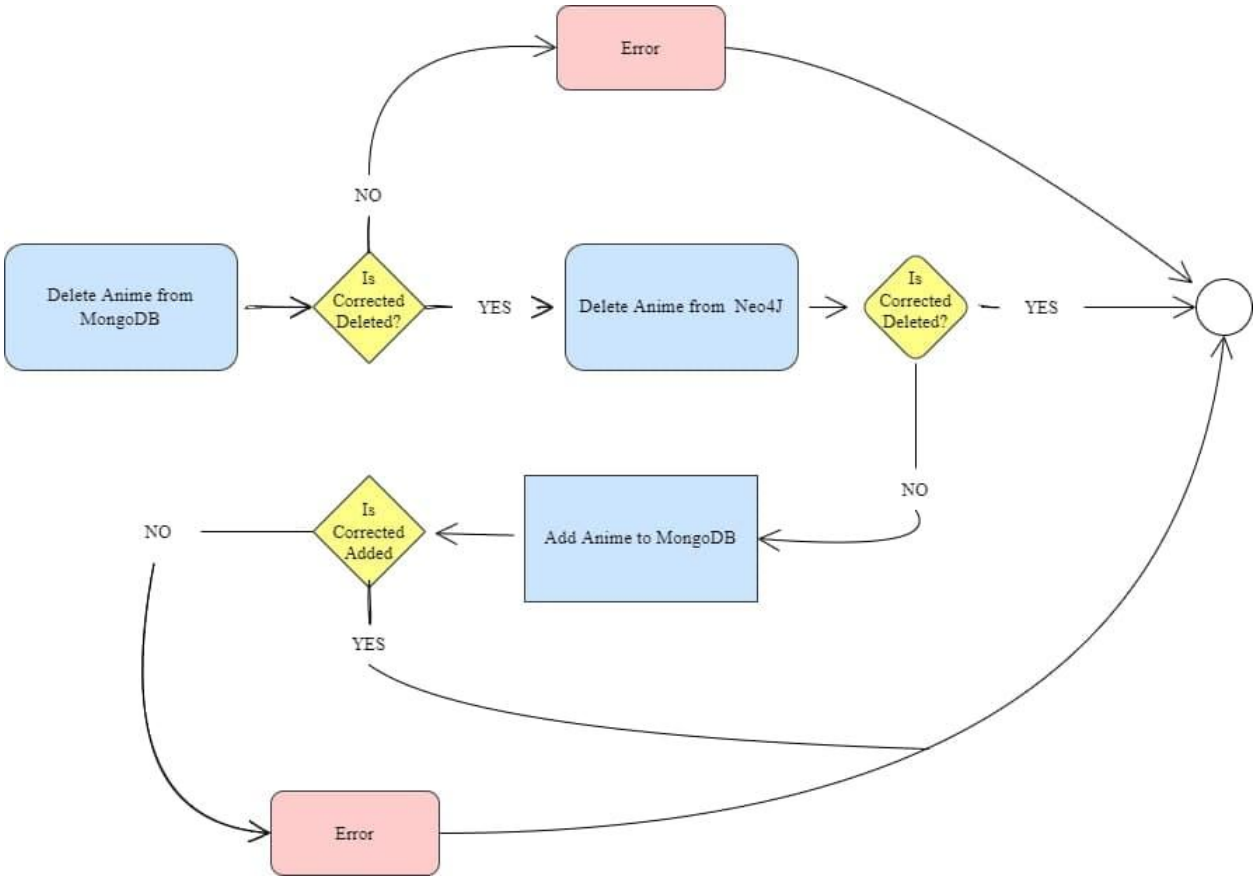
Data among the databases

Having the names of the animes stored as nodes of Neo4J and the name of the users as embedded documents in MongoDB we had to manage the consistency between the two databases when an Anime is being created, deleted, updated, or voted.

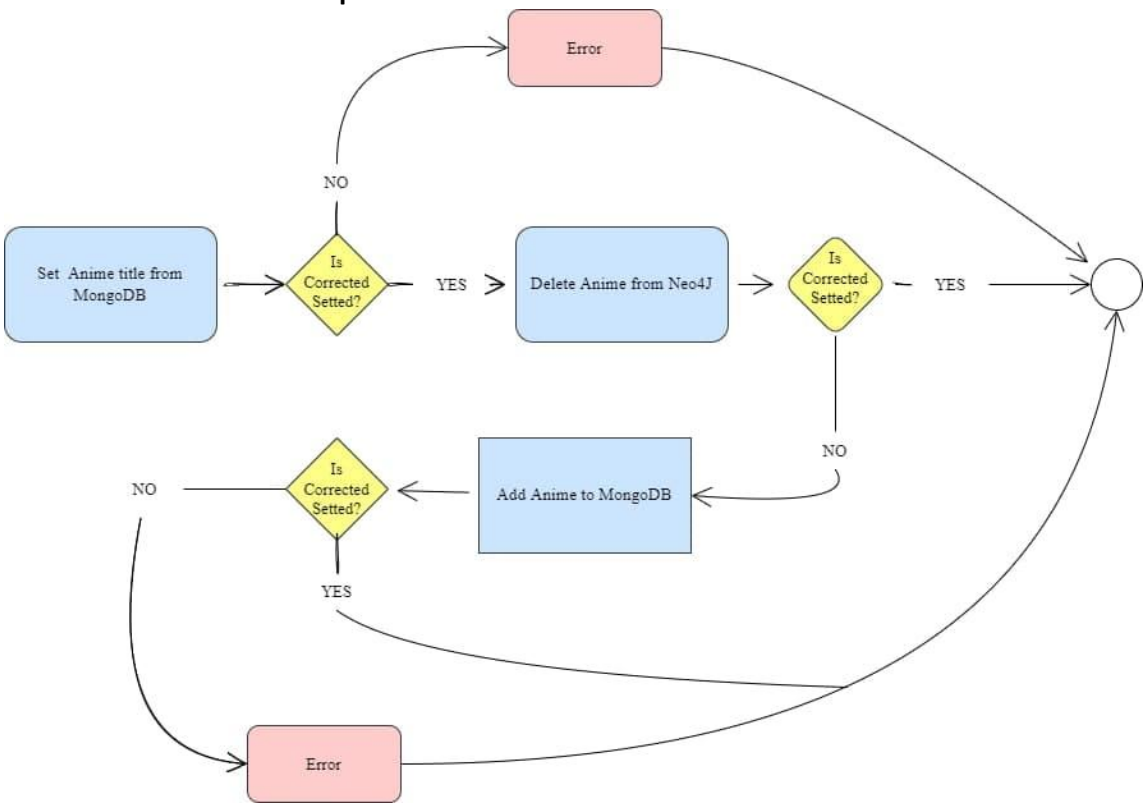
When an anime is created:



When an anime is deleted:



When an anime's name is updated:



The CAP theorem

For our application we decided to ensure Consistency and Availability:

- **Consistency:** All users will have the same updated version of the data
- **Availability:** We ensured the availability of the data by having the data saved on three different virtual machines provided to us by the University of Pisa.

Thanks to MongoDB replicas we can ensure data availability and fault tolerance against the loss of data or a database server.

In some cases replication of data can provide a lower reading time of the data for users who send read operations to different servers.

Also, replicas ensures a service always available because even if a the primary server goes down there will be another one available for the user.