UNIVERSITÀ DI PISA

Computer Engineering

Large-Scale and Multi-Structured Database

BoardGameNet

Project Documentation

Gaia Anastasi
Clarissa Polidori
Leonardo Poggiani

Academic Year: 2020/2021

Table of Contents

Chapter 1

Introduction

1.1 Description

BoardGameNet is designed to create a community of people who are passionate about board games. It is organized as a social network where you can learn about new games or discuss your favorite ones with your friends. The application offers a list of many board games, which you can filter by name, category, or sort by some parameters such as average **rating**, number of **comments** or publication's year. The **Game**'s main page includes the most important features: description, rules and the link of URL site, where you can see more details about games or the game's average price. Moreover you can leave a **review** or a **rating** from 1 to 10 and you can also view the average rating obtained by integrating the external rating of the site with the ratings of the application's users.
Another important feature is the opportunity to read **articles** published by the most popular users, named *influencers*. You can also create **groups** with your friends about a game, where you can discuss by posting.
To sum up, in this platform you can read detailed information on board games, like or review them, but you can also take advantage of the social part of this application: you can make friends, **follow** *influencers*, make **groups** with friends, and **post** on them. In your home page you will see a list of **articles** of the *influencers* that you are following. If you click on one of them you can read it and leave a **like**, a **dislike** or a **comment** on it.

Link to the GitHub repository:
`https://github.com/leonardopoggiani/Large-Scale-Project`

Chapter 2

Design

2.1 Requirements

2.1.1 Functional Requirements

The application's users are divided into four categories that we will analyze specifically in the next chapter. The following list summarizes the functional requirements for each of the actors of our application: *admin*, who has the highest level of privilege, he/she keeps the databases updated and acting also as an analyst; *moderator*, who has a lower privilege level, he/she has a controlling role in the social part as well as manages the user behaviour; *influencer*,who is just a more popular standard user, and finally there is *standard user*.
In any case, to access the application it is necessary to sign up. The requirements for each actor are the following:

- *Standard user*'s functional requirements:
  - Login and Logout
  - Modify profile's information
  - Browse **articles** written by followed *influencers*. If one still doesn't follow any *influencer*, then can browse the suggested **articles** based on his/her favorite categories.
  - Filter **articles** by **author**, game or day of publication. Order articles by number of **comments**, number of **likes** or numbers of **dislike**
  - Browse suggested **games** based on one's favourite categories.
  - Filter **games** by name, category, year of release. Order games by number of votes, average **rating** or number of **reviews**.
  - View the detailed information of a **game** or read an **article**.
  - Browse *friends*, suggested *friends*, followed *influencers* or suggested *influencers*.
  - Browse the **groups** he/she is *admin* of, delete them and browse the groups he/she is a member of without being admin.

  - **Post** in a **group**.
  - Add members to a **group** or create a new one.
  - Start to follow *standard users* or *influencers*.
  - Leave or remove a **like**, a **dislike** or a **comment** on an article
  - Leave a **rating** and leave or remove a **review** on an game

- *Influencer*'s functional requirements:
  - Publish **article** and delete them.
  - All the *Standard User*'s functionalities

- *Moderator*'s functional requirements:
  - Show statistics on **users** to promote them as an *influencers* or downgrade them to standard users.
  - Delete **comments**, **reviews** or **articles** from all **users** if deemed inappropriate.
  - All the *Standard User*'s functionalities.

- *Admin*'s functional requirements:
  - Find **users** or **games**.
  - Delete **users** who have not logged in for too long.
  - See detailed statistics on **games** and **users**.
  - Promote **users** as a *moderators*.
  - Manually add or delete **games**.

2.1.2 Non-Functional Requirements

- *Usability*: Perform the operations provided by the application must be easy for any user, so a simple and intuitive Graphical User Interface is required.
- *Maintainability*: The code must be readable and modulated to allow the addition of new features and the identification of any errors in an easy way.
- *Posts*, *Comments* and *Reviews* must follow a *causal-consistency*.
- *High Availability*: Considering that the application is mainly a social network, it is important to offer a service that is as available as possible.
- *Security*: Users passwords are encrypted before being stored in the database.
- *Read-your-own-write consistency*: To allow the user to immediately check if his operations are carried out successfully.
- *Low latency*: A low response time is necessary in order to avoid too long waits for the user.

2.1.3 Variety, Velocity and Volume of data

Variety is guaranteed by the fact that users and games in our database were mainly downloaded from three websites:

- https://boardgamegeek.com
- https://luding.org
- https://www.wikidata.org

To download information from these sites we used a scraper obtained from

- https://github.com/recommend-games/board-game-scraper

Through python scripts we adapted the datasets obtained from scraping according to our needs, eliminating fields, adding new ones or modifying their value or format. Then, once we got the complete game and user datasets, we uploaded them to the **Document Database** and **Graph Database**. All the other data were created randomly from these two datasets through a java script.

Regarding the **volume of data**, we have about 60 000 games, about 32 000 users and about 300 articles reaching about 70MB, without considering the groups and all the relationships between these main entities, so we estimated a volume of no less than 100 MB. We will describe the estimated data load in the next section.
The **velocity** is guaranteed by the fact that specific *analytics* are designed to allow the admin to detect obsolete user profiles, users who have not logged in for a long time, and delete them. The same for the less rated games, with few reviews or articles about them, they are deleted by the admin who will update the list of games periodically, making decisions based on statistics of the categories preferred by users, on the average age of them and on the countries in which the app finds more accesses.

2.2 Load estimation

We estimated the load of the application with approximate computations to be able to design it in the best possible way from the performance point of view.
Starting from the data obtained through scraping, we have **60 000** games and **30 000** users, in which information not present was added via a Python script.
Starting from this data and assuming that between the registration of new users and the cancellation of inactive users remain stable, we can assume (optimistic estimate) about 70% of active users, where by active we mean the user who makes at least one login per day, thus getting **21 000** active users.
Among the users we can estimate that 85/90% is *normalUser*, that is a simple user of the application, while the remaining part of the users will be *influencer*, users who can publish

articles on games, arriving then to estimate **5 000** influencers.
It might be possible to arrive at this number by introducing (at least in the application launch phase) more facilitated user promotion policies. If a certain threshold is reached, requirements may become more stringent.
Assuming an average rate of publication of articles equal to 1/2 a week, in a year we could arrive to count about **520 000** articles.
Also, assuming that active **21 000** normalUsers read at least one of the six suggested articles, we could have approximately **20 000** articles read per day once fully operational.
On these articles, users can leave likes, dislikes or comments. Assuming that each user who reads the article leaves a like or a dislike but only 10% leaves comments, we estimate a number of likes equal to **14 000** per day, a number of dislikes equal to approximately **6,000** per day and approximately **2 000** comments per day.
Assuming that users are more likely to follow other users than to remove the follow once they are followed, we can estimate an average of 2 followers per day, for a total of approximately **40 000** total follow and a 10% of unfollow operations, so about **4 000**.
Given these numbers we then estimated that each active group could have 1/2 groups for a total of about **40 000** groups. If each group has about 7/8 members, who exchange an average of 5 posts per day, we could have about **1 600 000** posts.
As for games, the number could continue to grow by scraping new releases or by finding new sites to scrap on. The number of games at full capacity could stabilize around **150 000** table games.
Being a social network oriented towards the discovery of new board games, it is assumed that an active user checks the reviews and the characteristics of at least a couple of games among those suggested for the user, therefore about **40 000** games viewed per day .
These calculations were made starting from the data obtained from scraping and assuming a modest success of the application, that is a constant number of users over time, a fairly typical trend for a blog.

2.3 Main Actors and Use Cases Diagram

The application was developed to be used by 4 different actors:

- ***Standard User***, the main actor who has access to all the features of the application. After logging in, can view the list of articles written by the following *influencers* on homepage, otherwise can display a list of articles suggested based on its interests and then filter the entire list. By choosing from a menu, the user can see the list of games present in the application. This list is suggested according to friends or favourite categories. He/she can also view the list of users he/she follows, influencers and non-influencers, and view suggestions for new people to meet. When the user has friends, he/she can create its own personal groups in which to discuss a specific game by

posting.

- ***Influencer***,is a standard user who has been promoted by the *moderator* as *influencer* based on his/her activity in the application, displaying users who have many followers or those who review more games. Once a user become an *influencer* he/she can also publish articles about 1 or 2 games.

- ***Moderator***, unlike the standard user and the *influencer* he/she has a higher level of privilege, he/she has access to all the features of the app and also manages the part of the social network, can delete all comments on articles, reviews on games and all articles deemed inappropriate. It can also display a statistics page, where useful information is shown to promote a standard user to an *influencer* or to downgrade him to a standard user.

- ***Admin***, it is not a standard user, it does not have access to all the functions of simple users, it also acts as an analyst, in the dedicated application page it can display many types of statistics useful for making decisions, such as deleting games, adding new ones , promote a user as *moderator*. It can also delete users from the application, obsolete profiles that have not been logged in for a long time.

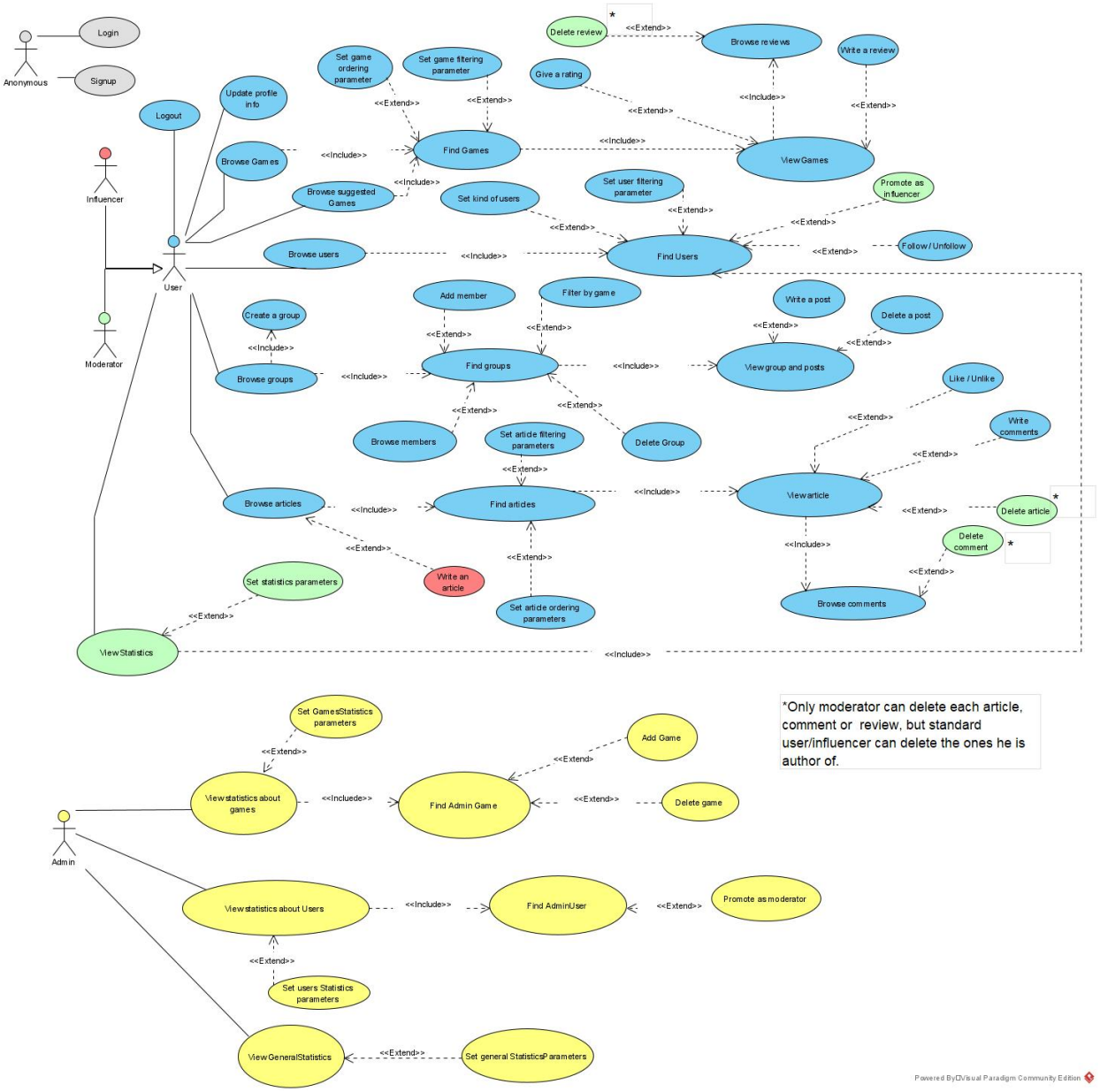


Figure 2.1: Use Cases Diagram

This use case specifically describes the functional requirements of chapter 1, but needs some clarification:

- In blue there are the functions available for the *standard user* and therefore also for the actors who represent a specialization of them, *influencer* and *moderator*
- In red there are actions available only for the *influencer*, which are added to the blue ones.
- Green shows the specific functions for a standard user who is also moderator, which are also added to those in blue. A moderator has a page in which he/she can navigate, which contains useful statistics for his/her moderator function, from it you can directly search for a user and promote or demote him, that's why the "include" relationship between the moderator's statistics and the find user.

- Finally we have a separate Use Cases for the admin, which accesses a completely different part of the application and can perform different actions, all based on the statistics provided. The admin searches for a user or a game directly without viewing the entire lists.

2.4 UML Class Diagram

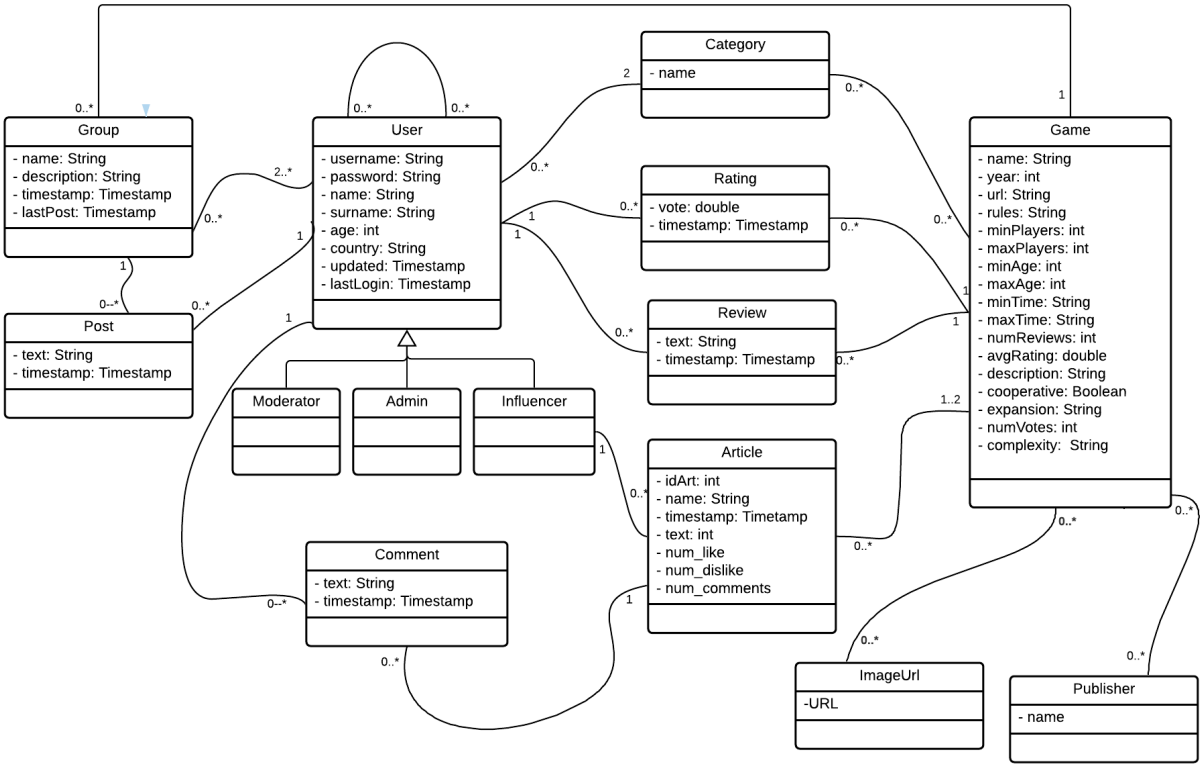


Figure 2.2: Class Diagram

User	User that can be *Influencer, Moderator or Admin*
Game	Game provided by the application
Article	Article wrote by an influencer
Rating	Rating left by a user about a *Game*
Comment	Comment left by a user about an *Article*
Review	Review left by a user about a *Game*
Category	Game's category
Publiher	Game's publisher
ImageUrl	Game's image
Group	Group created by a user
Post	Post created by a user within a *Group*

Table 2.1: Classes definition

- A **User** must choose two categories of games that he/she prefers when registering. After logging in he/she can write 0 or more ***Comments*** to an article, write 0 or more ***Reviews*** to a game, and leave 0 or more ***Ratings***, a maximum of one for each game. You can create or join 0 or more groups and publish 0 or more ***Posts*** in them. He can also follow others ***Users***.
- A ***Review*** such as a ***Rating*** or a ***Comment*** are obviously made by one and only one ***User*** , and are related to one and only one ***Game*** or ***Article***
- A ***Post*** is related to one and only one group
- A ***Group*** can be created to discuss only one ***Game***
- A ***Influencer*** can write 0 or more articles
- A ***Article*** must cover at least one game or at most 2.
- A ***Game*** can have 0 or more ***Publisher***, 0 or more ***ImageUrl*** and 0 or more ***Categories***

Chapter 3

Project

3.1 Adopted databases

For our application, we have decided to use two different types of databases, **DocumentDB** and **GraphDB** for the reasons that we will explain below.
First of all, we considered the main datasets of our application obtained through the scraping mechanism: **Games** and **Users**. Given the large amount of data of this type and given the fact that we have to ensure flexibility, we considered to use a document DB. For example we need to store *heterogeneous* data such as URLs, arrays, data of different types, often empty or missing fields about **Games**. Plus, we need to filter often to guarantee the *usability* requirements, and the *high availability* and the speed in finding information is very important. Information about **Users** and **Games** must always be available and the document DB lends itself better than others to the partitioning mechanism plus, it ensure us to obtain such information after a short period of time.
We decided to store **Games** and **Users** in two collections and we decided to add a third one representing the entity **Article**. Initially, to represent a 1 to many relationship with users we thought of inserting the articles as users' embedded documents, but almost immediately we realized that it was not the best solution for the following reasons. The articles have a text field that can grow indefinitely, plus each **Influencer**, can write an indefinite number of **Articles** and this could lead to saturate the space allocated for a single document in a document DB. Furthermore, considering how we designed our application, the need to find information on all the articles written by a particular user never occur. Each article stores information about the username of its author and this is enough for our purposes. Each of the other entities that characterize our application i.e. **Groups**, **Like**, **Comment**, **Review**, **Post**, **Follow** and **Rating** represents the social network part of our application so clearly a GraphDB is more suitable for storing their information. The social network part also revolves around **Games**, **Users**, and **Articles**, so we decided to replicate the information useful for this purpose in the two databases, the choices will be described in the next paragraphs.

3.2 Document Database

As a Document Database we decided to use **MongoDB** because it supports indexing, clustering and replication as well as because of our need for a non-relational **DBMS**, in which to store large amounts of data in a *flexible* way and which was optimized for *analytics*.

3.2.1 Entities Handled

Document database is used to manage the following entities

- **Game**, game's characteristics and rules
- **User**, user's anagraphical information and login credentials.
- **Article**, article's informations.
- **ImageUrl**, game's image URL
- **Publisher**, game's publisher
- **Category**, game's category.

We organized this information into 3 collections that we will describe in details in the next paragraph.

3.2.2 Collection Structure


```
{
  "_id": {
    "$oid": "600706c9b17559309la7caf2"
  },
  "first_name": "Jeremy",
  "last_name": "Chakado",
  "registered": 2014,
  "last_login": "2020-11-30T00:00:00+0000",
  "country": "France",
  "username": "chakado",
  "updated": "2020-12-01T15:21:12+0000",
  "age": 46,
  "password": "eae453819442937c9a7e02d0e8e6265d9659950f38adf026ada8flac13ba65f2",
  "role": "normalUser"
}
```


Figure 3.1: User's collection

This collection contains only information about the **User**: personal information, access credentials and users' roles.
Attributes list:

- *first_name*, user's first name.
- *last_name*, user's last name,
- *registered*, user registration's year
- *last_login*, last user's access to the application.
- *country*, user's origin country.
- *username*, (Unique) User's username and also user's ID.
- *updated*, present only if the data has been scraped
- *password*, user's encrypted password
- *role*, user's role which can assume the following values: *normalUser*, *influencer*, *moderator*, *admin*.


```
{ "_id": {
    "$oid": "5ffec15e79012219f8c40195"
  },
  "name": "Muse: Awakenings",
  "year": 2018,
  "description": "Muse: Awakenings is a standalone game that can be played as its own complete experience",
  "publisher": [
    "Quick Simple Fun Games:32523",
    "Fractal Juegos:33618",
    "funbot:42203",
    "Gém Klub Kft.:8820"
  ],
  "url": "https://boardgamegeek.com/boardgame/252776",
  "image_url": [
    "https://cf.geekdo-images.com/HAl2mFKP0mujF49-X0mMiQ__",
    "original/img/mbdrLObYg202IpTEgBHY22LvhAI=/0x0/pic4133038.png",
    "https://cf.geekdo-images.com/HAl2mFKP0mujF49-X0mMiQ__",
    "thumb/img/ENOc9pIVCgUX02pXZ2eU3LuN5-Y=/fit-in/200x150/filters:strip_icc()/pic4133038.png"
  ],
  "min_players": 2,
  "max_players": 12,
  "min_age": 10,
  "max_age": null,
  "min_time": 30,
  "max_time": 30,
  "category": [
    "Card Game:1002",
    "Humor:1079",
    "Party Game:1030"
  ],
  "cooperative": true,
  "num_votes": 183,
  "avg_rating": 6.9167717528373265,
  "complexity": 1,
  "expansion": null,
  "num_reviews": 0,
  "rules_url": ""
}
```


Figure 3.2: Game's collection

This collection contains information about three entities, **Game**, **Publisher**, **Category** and **ImageURL**. Main attribute list:

- *name*,(unique) game's name and also game's ID.
- *year*, game's year of release.
- *description*, game's description.
- *publisher*, list of game's publishers.
- *image_url*, list of game's images' URL.
- *category*, list of game's categories.
- *num_votes*, number of game's ratings.
- *avg_rating*, game's average rating.
- *num_reviews*, number of game's reviews.


```
{
    "_id": {
      "$oid": "6007663ab73e0f1aa1cc8ee7"
    },
    "id": 1,
    "author": "camulonus",
    "title": "New Article",
    "body": "Article's text",
    "timestamp": "2021-01-20 00:07:38.654",
    "num_likes": 90,
    "num_dislikes": 15,
    "num_comments": 100,
    "games": [
      "Stoff-Memo Baustelle",
      "Remember Limerick! The War of the Two Kings: Ireland, 1689-1691"
    ]
  }
```


Figure 3.3: Article's collection

This collection contains information about **Article** and its author. Main attribute list:

- *id*,(unique) article's ID
- *author*, article's author (user's username)
- *title*, article's title.
- *body*, article's body.
- *timestamp*, timestamp of article's publication
- *num_likes*, number of article's likes.

- *num_dislikes*, number of article's dislikes.
- *num_comments*, number of article's comments.
- *games*, list of games mentioned in the article (at most two).

3.2.3 Queries Handled

List of queries handled by MongoDB:

- Insert a new **Game** (admin only)
- Retrieve a **Game** by name
- Delete a **game** (admin only)
- Retrieve **Game** list using filtering parameters
- Retrieve **Game** list using ordering parameters
- Retrieve **Game**'s details
- Update attribute *num_reviews* of a **Game**
- Update attribute *num_votes* of a **Game**
- Update attribute *avg_rating* of a **Game**
- Insert a new **User** at registration time
- Retrieve **User** information at login time
- Modify **User**'s information
- Retrieve **User** list
- Retrieve **User** by username
- Insert a new **Article** (*Influencer* only)
- Delete an article (*Influencer* and *Moderator* only)
- Update attribute *num_comments* of an **Article**
- Update attribute *num_likes* of an **Article**
- Update attribute *num_dislike* of an **Article**
- Retrieve **Article** list using filtering parameters
- Retrieve **Article** list using ordering parameters

- Retrieve **Article**'s details
- **Analytic**: Show the least rated games
- **Analytic**: Distribution of users by country
- **Analytic**: Get Information about a specific category
- **Analytic**: Show game's distribution by *category*
- **Analytic**: Show less recent logged **Users**
- **Analytic**: Show **Users** distribution by age
- **Analytic**: Show daily average of login by *country*
- **Analytic**: Show the number of articles published in a specific period
- **Analytic**: Distribution of login by day
- **Analytic**: Distribution of games covered in articles in a specific period.
- **Analytic**: Number of likes/dislike for each influencer

The implementation of these queries and analytics will be illustrated in Chapter 4

3.2.4 Indexes

We designed and implemented different indices for the collections managed by MongoDB based on the assumptions about the system's load and performance.

Analyzing the application we noticed how it fell into the *read-heavy applications* category.
In fact, given the typical structure of a **Social network**, reading operations are preferred over writing operations, with particular attention to computations which can also become expensive in the case of analytics. The writes are limited to simple operations (add an article, add a post, remove user, add a group ..) and not very expensive from the performance point of view and, although they can also be performed with a high frequency, the focus will be certainly on reading operations, as regards reading games, articles and user suggestions to follow.
We therefore decided to prefer the creation of multiple indexes that would allow the information to be quickly retrieved from the database, above all for what concerns users and games, which represent the main load of the application.
In this way, access to the data will be faster and the cost necessary to update the indexes is not very high as can be seen from the low number of operations that impact reading on the indexes, indicated below. Even the reported writing operations are usually carried out much more infrequently than the reading ones.

Minimum number of players

```
> db.Games.find( {"min_players" : {$gte: 4} }).explain("executionStats")
< { queryPlanner:
    { plannerVersion: 1,
      namespace: 'Project.Games',
      indexFilterSet: false,
      parsedQuery: { min_players: { '$gte': 4 } },
      winningPlan:
       { stage: 'COLLSCAN',
         filter: { min_players: { '$gte': 4 } },
         direction: 'forward' },
      rejectedPlans: [] },
   executionStats:
    { executionSuccess: true,
      nReturned: 1757,
      executionTimeMillis: 43,
      totalKeysExamined: 0,
      totalDocsExamined: 60021,
```

Figure 3.4: Without the index on min_players

Users can perform a search for games based on the minimum number of players required to play. So it's useful to define an index for that field as shown in the picture below.

query	type (W/R)
Retrieve Game list using filtering parameter	R

Table 3.1: Queries affected by index on min_players.

```
> db.Games.find( {"min_players" : {$gte: 4} }).explain("executionStats")
< { queryPlanner:
    { plannerVersion: 1,
      namespace: 'Project.Games',
      indexFilterSet: false,
      parsedQuery: { min_players: { '$gte': 4 } },
      winningPlan:
       { stage: 'FETCH',
         inputStage:
          { stage: 'IXSCAN',
            keyPattern: { min_players: 1 },
            indexName: 'num_players',
            isMultiKey: false,
            multiKeyPaths: { min_players: [] },
            isUnique: false,
            isSparse: false,
            isPartial: false,
            indexVersion: 2,
            direction: 'forward',
            indexBounds: { min_players: [ '[4, inf.0]' ] } } },
      rejectedPlans: [] },
   executionStats:
    { executionSuccess: true,
      nReturned: 1757,
      executionTimeMillis: 3,
      totalKeysExamined: 1757,
      totalDocsExamined: 1757,
```

Figure 3.5: With the index on min_players

Game name

The index on the game's name is fundamental for the searches carried out on the basis of the latter and is extremely efficient from the performance point of view.

```
> db.Games.find({name:"Streams"}).explain("executionStats")
< { queryPlanner:
    { plannerVersion: 1,
      namespace: 'Project.Games',
      indexFilterSet: false,
      parsedQuery: { name: { '$eq': 'Streams' } },
      winningPlan:
       { stage: 'COLLSCAN',
         filter: { name: { '$eq': 'Streams' } },
         direction: 'forward' },
      rejectedPlans: [] },
   executionStats:
    { executionSuccess: true,
      nReturned: 1,
      executionTimeMillis: 32,
      totalKeysExamined: 0,
      totalDocsExamined: 60022,
```

Figure 3.6: Without the index on name

```
> db.Games.find({name:"Streams"}).explain("executionStats")
< { queryPlanner:
    { plannerVersion: 1,
      namespace: 'Project.Games',
      indexFilterSet: false,
      parsedQuery: { name: { '$eq': 'Streams' } },
      winningPlan:
       { stage: 'FETCH',
         inputStage:
          { stage: 'IXSCAN',
            keyPattern: { name: -1 },
            indexName: 'name_-1',
            isMultiKey: false,
            multiKeyPaths: { name: [] },
            isUnique: false,
            isSparse: false,
            isPartial: false,
            indexVersion: 2,
            direction: 'forward',
            indexBounds: { name: [ '["Streams", "Streams"]' ] } } },
      rejectedPlans: [] },
   executionStats:
    { executionSuccess: true,
      nReturned: 1,
      executionTimeMillis: 0,
      totalKeysExamined: 1,
      totalDocsExamined: 1,
```

Figure 3.7: With the index on name

Year

The index on the year of publication of a game can be useful in the filtered searches carried out by the user and like those previously indicated it does not particularly impact on performance but is still effective.

query	type (W/R)
Retrieve a Game by name	R
Retrieve Game list using filtering parameters	R
Retrieve Game's details	R
Update attribute numreviews of a Game	W
Update attribute numvotes of a Game	W
Update attribute avgrating of a Game	W
Delete a Game	W

Table 3.2: Queries affected by index on game name.

```
> db.Games.find({"year" : 2018}).explain("executionStats")
< { queryPlanner:
    { plannerVersion: 1,
      namespace: 'Project.Games',
      indexFilterSet: false,
      parsedQuery: { year: { '$eq': 2018 } },
      winningPlan:
       { stage: 'COLLSCAN',
         filter: { year: { '$eq': 2018 } },
         direction: 'forward' },
      rejectedPlans: [] },
   executionStats:
    { executionSuccess: true,
      nReturned: 2736,
      executionTimeMillis: 52,
      totalKeysExamined: 0,
      totalDocsExamined: 60021,
```

Figure 3.8: Without the index on year

```
> db.Games.find({"year" : 2018}).explain("executionStats")
< { queryPlanner:
    { plannerVersion: 1,
      namespace: 'Project.Games',
      indexFilterSet: false,
      parsedQuery: { year: { '$eq': 2018 } },
      winningPlan:
       { stage: 'FETCH',
         inputStage:
          { stage: 'IXSCAN',
            keyPattern: { year: -1 },
            indexName: 'year_-1',
            isMultiKey: false,
            multiKeyPaths: { year: [] },
            isUnique: false,
            isSparse: false,
            isPartial: false,
            indexVersion: 2,
            direction: 'forward',
            indexBounds: { year: [ '[2018, 2018]' ] } } },
      rejectedPlans: [] },
   executionStats:
    { executionSuccess: true,
      nReturned: 2736,
      executionTimeMillis: 12,
      totalKeysExamined: 2736,
      totalDocsExamined: 2736,
```

Figure 3.9: With the index on year

query	type (W/R)
Retrieve Game list using filtering parameter	R

Table 3.3: Queries affected by index on year.

username

The index on the user's username has a significant impact on a large number of operations and is therefore fundamental from a performance point of view.

```
> db.Users.find({username:"chakado"}).explain("executionStats")
< { queryPlanner:
    { plannerVersion: 1,
      namespace: 'Project.Users',
      indexFilterSet: false,
      parsedQuery: { username: { '$eq': 'chakado' } },
      winningPlan:
       { stage: 'COLLSCAN',
         filter: { username: { '$eq': 'chakado' } },
         direction: 'forward' },
      rejectedPlans: [] },
   executionStats:
    { executionSuccess: true,
      nReturned: 1,
      executionTimeMillis: 23,
      totalKeysExamined: 0,
      totalDocsExamined: 31836,
```

Figure 3.10: Without the index on username

query	type (W/R)
Retrieve User information at login time	R
Modify User's information	W
Retrieve User list	R
Retrieve User by username	R

Table 3.4: Queries affected by index on username.

```
> db.Users.find({username:"chakado"}).explain("executionStats")
< { queryPlanner:
    { plannerVersion: 1,
      namespace: 'Project.Users',
      indexFilterSet: false,
      parsedQuery: { username: { '$eq': 'chakado' } },
      winningPlan:
       { stage: 'FETCH',
         inputStage:
          { stage: 'IXSCAN',
            keyPattern: { username: -1 },
            indexName: 'username_-1',
            isMultiKey: false,
            multiKeyPaths: { username: [] },
            isUnique: false,
            isSparse: false,
            isPartial: false,
            indexVersion: 2,
            direction: 'forward',
            indexBounds: { username: [ '["chakado", "chakado"]' ] } } },
      rejectedPlans: [] },
   executionStats:
    { executionSuccess: true,
      nReturned: 1,
      executionTimeMillis: 0,
      totalKeysExamined: 1,
      totalDocsExamined: 1,
```

Figure 3.11: With the index on username

3.3 Graph Database

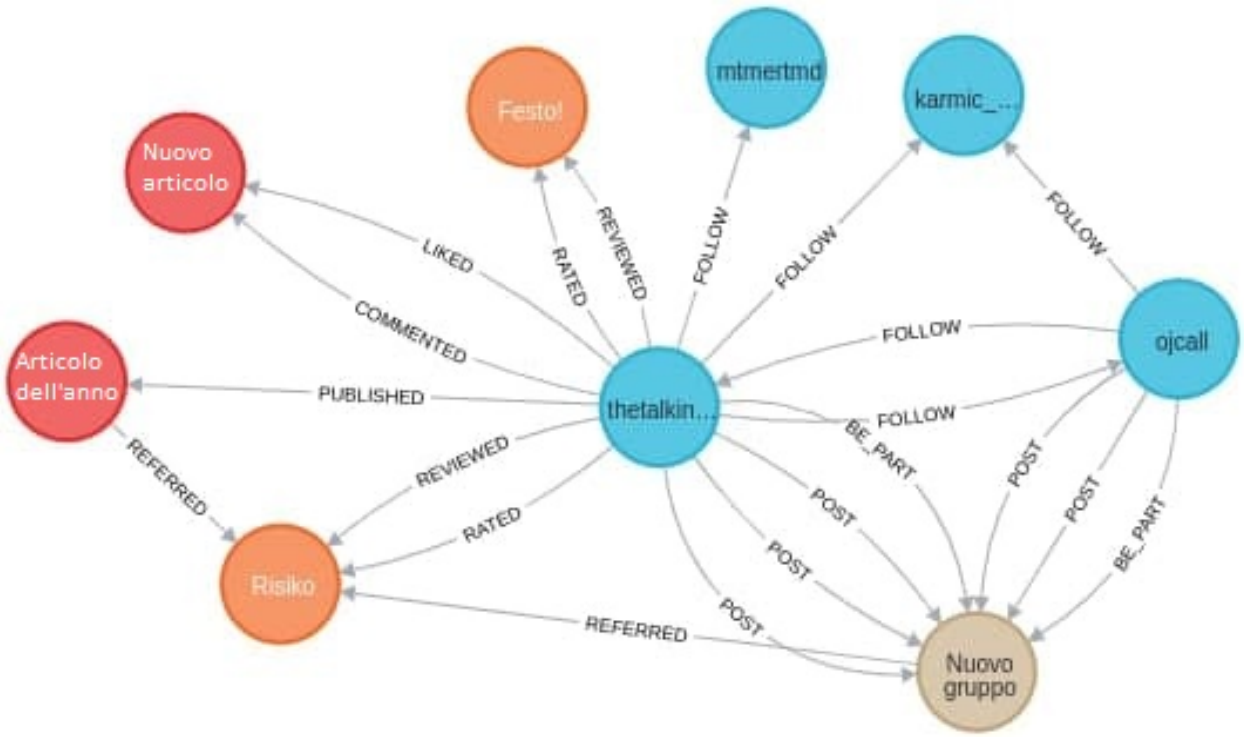
The Graph Database selected is **Neo4j** because it offers *indexing*, *clustering* and *replication* mechanisms and for properties that characterize it like Safety, Scaling and **Causal Consistency**.

3.3.1 Entities Handled

Graph database is used to manage the following entities

- **Game**, only name and two of categories
- **User**, only username and two favorite categories
- **Article**, only title
- **Group**, group created by a **User**
- **Post**, post published by a **User** inside a **Group**
- **Rating**, rating left by a **User** about a **Game**
- **Review**, review left by a **User** about a **Game**
- **Comment**, comment left by a **User** about an **Article**

3.3.2 Example of the graph in Neo4J



The previous image is a representative screenshot of our graph database and in the next section the **Graph Model** is presented to show in detail how we translated the entities described above in terms of vertices and edges with all their properties.

3.3.3 Graph Structure

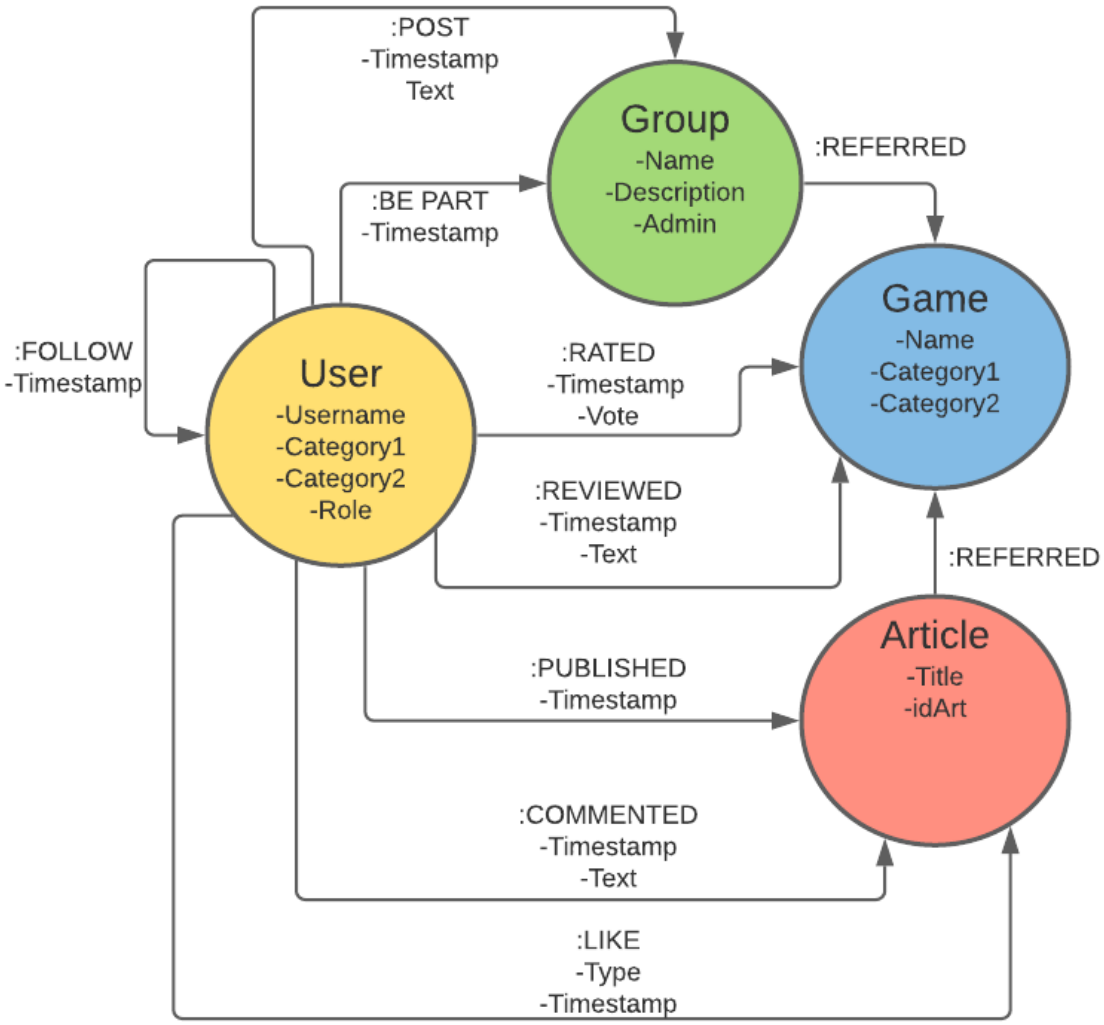


Figure 3.12: Graph Model

3.3.4 Queries Handled

Queries about User

Application Queries	Graph Queries
Insert a new **User** into the system at registration time	Insert a new **User** node into the graph
Delete a **User** (*admin* only)	Delete a **User** node
Create a follow relationship between the current User and the selected User	Add a **FOLLOW** edge between two **User** nodes
Delete a **FOLLOW** relationship	Delete a **FOLLOW** edge between two **Users**
Retrieve friend list of a **User** U	Match all the **User** nodes linked to U by two edges, one incoming and the other outgoing
Retrieve follower/followed list of a **User** U	Match all the **User** nodes linked to U by incoming edges (followers) or outgoing edges (followed)
Modify **User** information	Modify **User** node by changing some properties
Retrieve suggested **User**(*influencers followed by friends)* for an **User** U	Match all the **User** nodes that have role property equals to "influencer" and outgoing **FOLLOW** edges to **User** node A if the latter is linked to U by two **FOLLOW** edges, one incoming and one outgoing
Retrieve suggested **User** *(influencers with more followers)* of an **User** U	Match some **User** nodes with *role* property equals to *influencer* that have the most number of incoming **FOLLOW** edges from others **User** nodes
Retrieve suggested **Users**(*normalUser friends of friends)* of an **User** U	Match all the **User** node that have role property equals to *normalUser* and that are linked to 2 **FOLLOW** edges(1 outgoing and 1 incoming) to **User** node A if the latter is linked to U by two **FOLLOW** edges (1 incoming and 1 outgoing)
Retrieve suggested **Users** (*normalUser*) with the same favourite categories of an **User** U	Match **User** nodes with *role* property equals to *influencer* that have the same *category* properties of **U**
Count **Users** (*friends or followed influencers*) of a **User** U	Count all **FOLLOW** outgoing edges from U / Count all **User** nodes that linked to U with 2 edges (1 outgoing and 1 incoming)
Add/Delete **Follow** between **Users**	Add or Delete an **FOLLOW** edge between two **User** nodes
Change **User**'s role	Set **Game**'s *role* property with a different value

Analytic: *normalUser* who has reviewed the most distinct **Game**'s categories	Match the **User** nodes with *role* property equals to **normalUser** that has the largest number of outgoing **REVIEWED** edges to **Game** nodes of different categories
Analytic: *Influencer* who wrote about the largest number of distinct **Game**'s categories	Match the **User** nodes with *role* property equals to *Influencer* that has the largest number of outgoing **PUBLISHED** edges to **Game** nodes of different categories

Queries about Games

Application Queries	**Graph Queries**
Insert a new **Game**	Insert a new **Game** node into the graph
Delete a **Game**	Delete a **Game** node from the graph
Show list of **Game**'s **Reviews**	Match all the **Game**'s incoming **REVIEWED** edges
Show list of **Game**'s **Ratings**	Match all the **Game**'s incoming **RATED** edges
Count **Game**'s **Reviews**	Count **Game**'s **REVIEWED** incoming edges
Count **Game**'s **Ratings**	Count **Game**'s **RATED** incoming edges
Compute **Game**'s average **Rating**	Compute the average of the *vote* properties of **Game**'s **RATED** incoming edges
Add **Review** to a **Game**	Insert a **REVIEWED** edge from an **User** node to a **Game** node
Delete **Review** from a **Game** (only *moderator* and **Review** creator	Delete a **REVIEWED** edge between an **User** and a **Game**
Add **Rating** to a **Game**	Insert a **RATED** edge from an **User** node to a **Game** node
Retrieve *suggested* **Games** *by favourite* ***Game**'s categories*	Match all the **Game** nodes that have the main *Category* property equals to one of the two *Category* properties of a **Game** node

Queries about Group

Application Queries	**Graph Queries**
Retrieve **User**'s **Groups** (*Admin or only member*)	Match all **Group** nodes that have the *admin* property equals to this **User**'s username
Add a new **Group**	Insert a new **Group** node and a new **BE_PART** edge from **User** group's author and the **Group**. Insert also a **REFERRED** edge from the **Group** to the **Game** which it is about.
Delete a **Group** (only if *group's admin*)	Delete a **Group** node and all its outgoing and incoming edges
Retrieve **Group**'s members	Match all **User** nodes that have incoming edges from that **Group**
Count **Group**'s members	Count all **BE_PART** edges from **User** nodes to that **Group**
Retrieve list of **Posts** inside a **Group**	Match all **POST** edges from **User** nodes to that **Group**
Add a **Post** inside a **Group**	Add a **POST** edge from a **User** node to that **Group**
Delete a **Post** (*only post's author*)	Delete a **POST** edge between an **User** node and the post's group.
Add/Delete a **Group**'s member	Add or Delete a **BE_PART** edge from an **User** to a **Group**
Search *timestamp* of last **Post** published inside a **Group**	Match all **POST** edges between **User** nodes and that **Group**, to order by the *timestamp* property and limit to 1

Queries about Article

Add an **Article** (only if *influencer*)	Create a **Article** node with an **PUBLISHED** incoming edge from an **User** node and with one or two outgoing **REFERRED** edges to the **Game** nodes it talks about
Delete an **Article** (only if *article's author or moderator*)	Delete an **Article** node and all its outgoing and incoming edges
Add **Comment** to an **Article**	Create a **COMMENT** edge from an **User** node to a **Article** node
Delete **Comment** from an **Article**(only if *comment's author or moderator*)	Delete a **COMMENT** edge between a **User** and an **Article**
Retrieve list of **Article**'s **Comments**	Match all **COMMENT** edges between **User** nodes and that **Article** node
Count **Article**'s **Comments**	Count all **COMMENT** outgoing edges of that **Article**
Add **Like/Dislike** to an **Article**	Create a **LIKE** edge from an **User** node to a **Article** node setting *type* property to "like" or "dislike"
Delete **Like/Dislike** to from **Article**(only if *like's*)	Delete a **LIKE** edge between a **User** and an **Article**
Count **Article**'s likes/dislikes	Count all **LIKE** outgoing edges of that **Article**
Retrieve suggested **Articles** for an **User** U by categories of games that the articles talk about	Match all **Article** nodes that have at least an outgoing edge to a **Game** node that has *catgory* property equals one of the **User** node U *Category* properties.
Retrieve suggested **Articles** for an **User** U written by influencers followed by friends of U	Match all **Article** nodes that have an **PUBLISHED** incoming edge from and **User** that has an incoming **FOLLOW** edge from another **User** A that linked to U by 2 edges (1 incoming and 1 outgoing)

3.3.5 Indexes

The indexes adopted for Neo4j proved to be particularly useful in the data import phase, when it was necessary to carry out intensive merging and creation of relationships starting from nodes. However, the maintenance of the indexes in the phase following the creation of the graph can lead to various problems as the writes become slower and the space occupied increases.

For the implementation of our social network, the insertions within the graph are fundamental as users have the possibility to insert comments, likes, reviews, but they are all insertions of relations starting from existing nodes, since the insertions of nodes (article, group, user and game) do not occur with a high frequency.

Game name

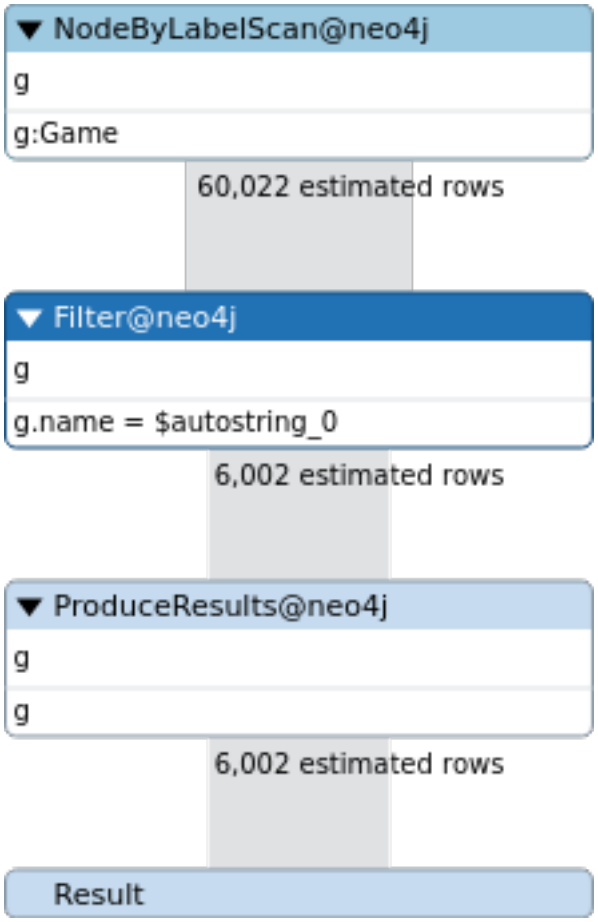


Figure 3.13: Without the single-property index on game name

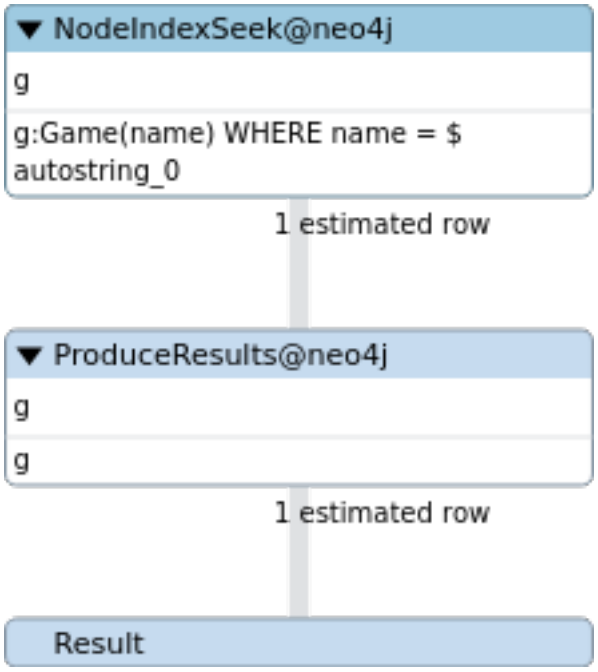


Figure 3.14: With the single-property index on game name

query	type (W/R)
Delete a Game	W
Show list of Game's Reviews	R
Show list of Game's Ratings	R
Count Game's Ratings	R
Count Game's Reviews	R
Compute Game's average Rating	R
Add Review to a Game	W
Delete Review from a Game	W
Add Rating to a Game	W
Add Review to a Game	W
Add Post to a Game	W

Table 3.5: Queries affected.

Games categories

Composite indexes are particularly delicate to treat because they have strong limitations from the point of view of performance. In particular they should be used only when there are predicates (searches) on all indexed properties. If there are predicates on only a subset of the indexed properties, it will not be possible to use the composite index.
In our case, this index could be particularly useful during queries made to suggest games or articles of interest to the user, based on the categories specified at the time of registration. Therefore the search is always done through both properties as the user cannot specify only one.
It can happen that a game falls into only one category, but we observed that it is a rather rare phenomenon and this justifies the use of the index and the same stand for the indices below.

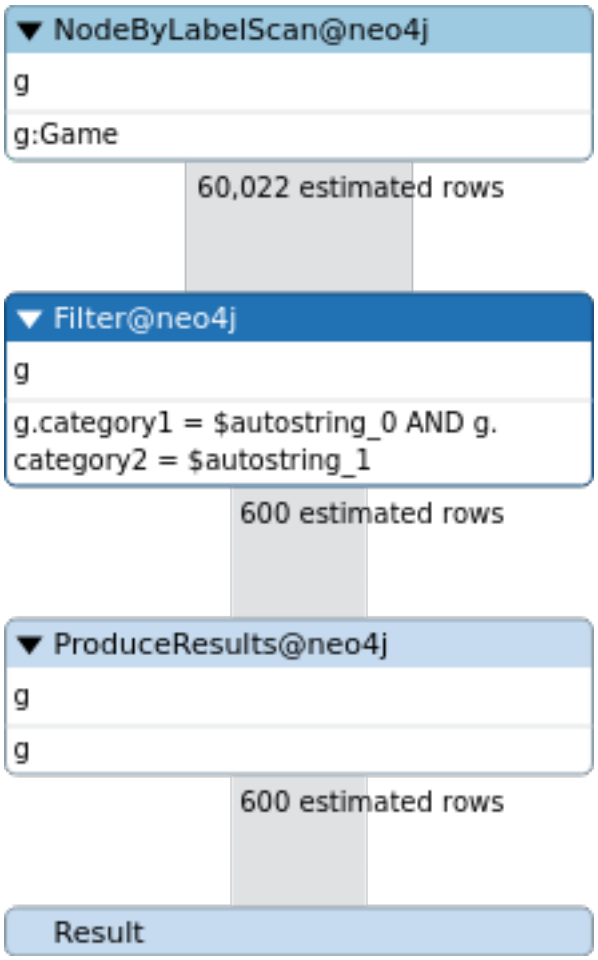


Figure 3.15: Without the composite index on games category

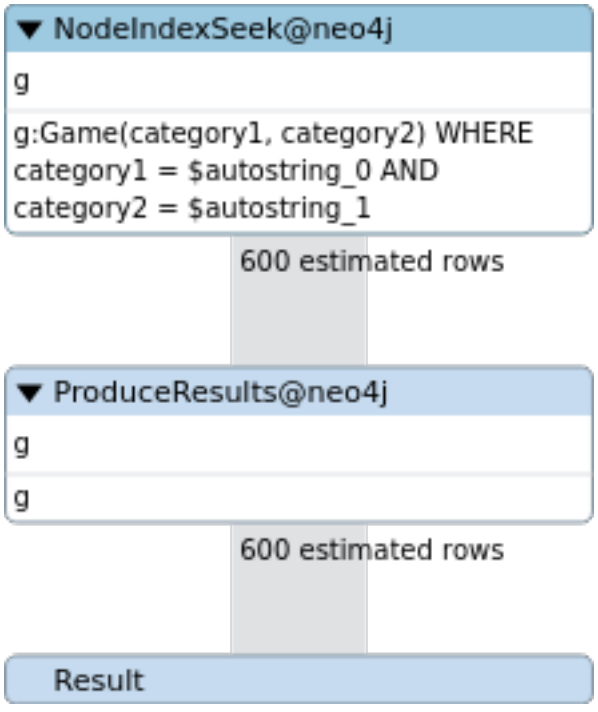


Figure 3.16: With the composite index on games category

query	type (W/R)
Retrieve suggested Games by favourite Game's categories	R

Table 3.6: Queries affected.

Username

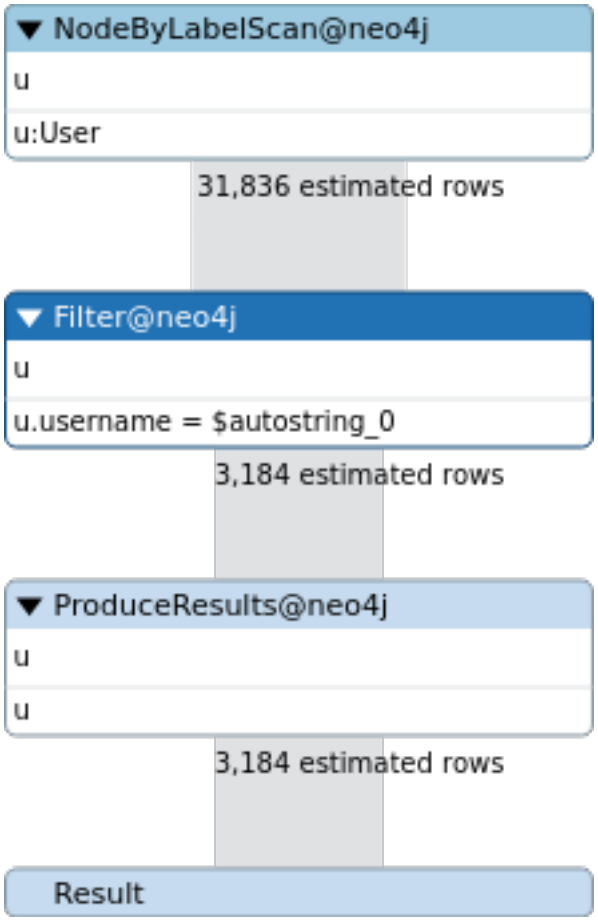


Figure 3.17: Without the single-property on username

query	type (W/R)
Delete a User	W
Show list of User's friends	R
Create a follow relationship between the current User and the selected User	W
Retrieve follower/followed list of a User	R
Modify User information	W
Retrieve suggested User	R
Count Users	R
Normal User who has reviewed the most distinct Game's categories	R

Table 3.7: Queries affected.

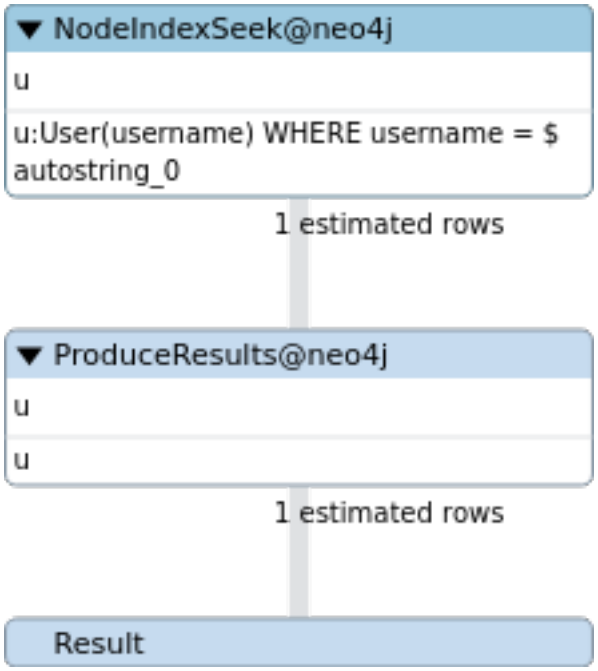


Figure 3.18: With the single-property on username

User cateogories

Figure 3.19: Without the composite on user categories

query	type (W/R)
Retrieve suggested Users (normalUser) with the same favourite categories	R

Table 3.8: Queries affected.

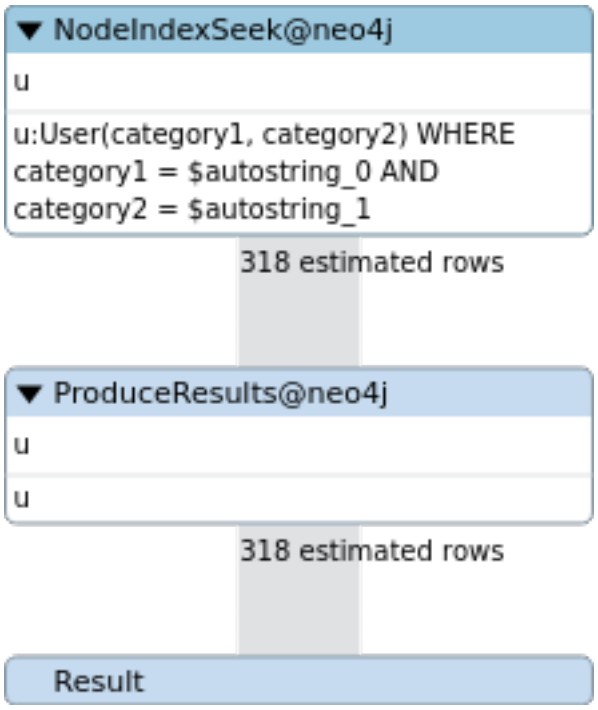


Figure 3.20: With the composite on user categories

3.4 Redundancies between Databases

Some of the information is replicated in both databases. The choices we made are due to the fact that we want to improve the performance of our application. Redundancies leads to an increase in memory consumption, and presupposes having to ensure a given consistency of data between the two databases. We are going to show in detail the information we decided to replicate and how we decided to act in case of inconsistency.

User's redundancy

As we already described, in **MongoDB** we store all the user's personal information and the credentials necessary for login. Also in **Neo4j** there is the User vertex and we decided to replicate only his/her ***Username***, which is also the identifier with which we can locate a particular user in the two databases. This information is sufficient because the login is performed in **MongoDB**, like all *Analytics* based on the age of users, on the country of origin and on logins. In **Neo4j** only queries for suggestions or user activity in the social network area are performed (how many reviews, how many likes ..) by using the two properties *category1* and *category2* that the **User** has only in **Neo4j**.

Game's redundancy

The entity **Game**, like the **User**, is present in both databases for approximately the same reasons. The details relating to the **Game** are always retrieved from MongoDB, and in the same way the filtering based on parameters.
In **Neo4j** we need to report the **Game**'s ***name*** in order to uniquely identify it and the two main ***Categories*** that characterize it, in order to get suggestions for users and to be able to implement all the features that revolve around the **game** entity such as groups, posts, reviews and ratings
In **MongoDB**, in the **Games** collection it has been decided to replicate redundant information, data that can be obtained by querying **Neo4j** such as: number of **Reviews**, number of **Ratings** and average **Rating**. This decision was made to sort the games according to these values. By adopting this solution every time the user wants to sort and then view the information about the selected *Game*, it is not necessary to access both databases, but only to query **MongoDB**, with the result of a more fluid and faster application.

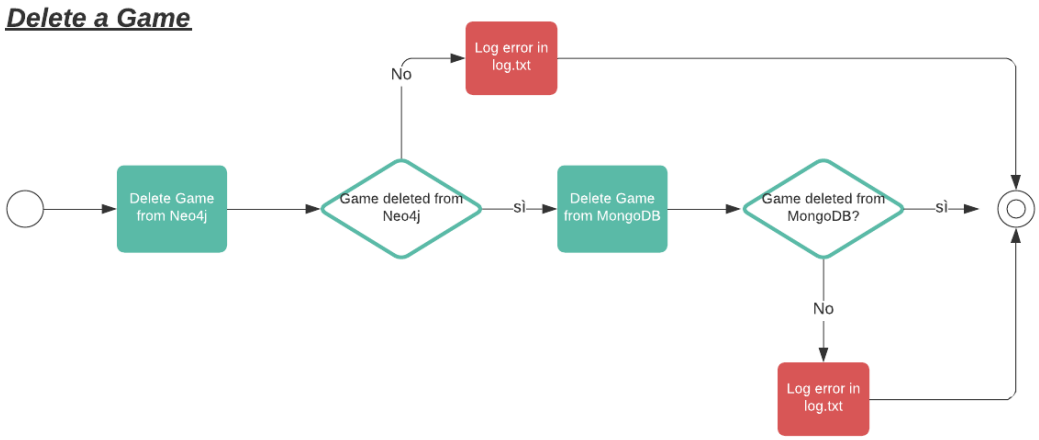
Article's redundancy

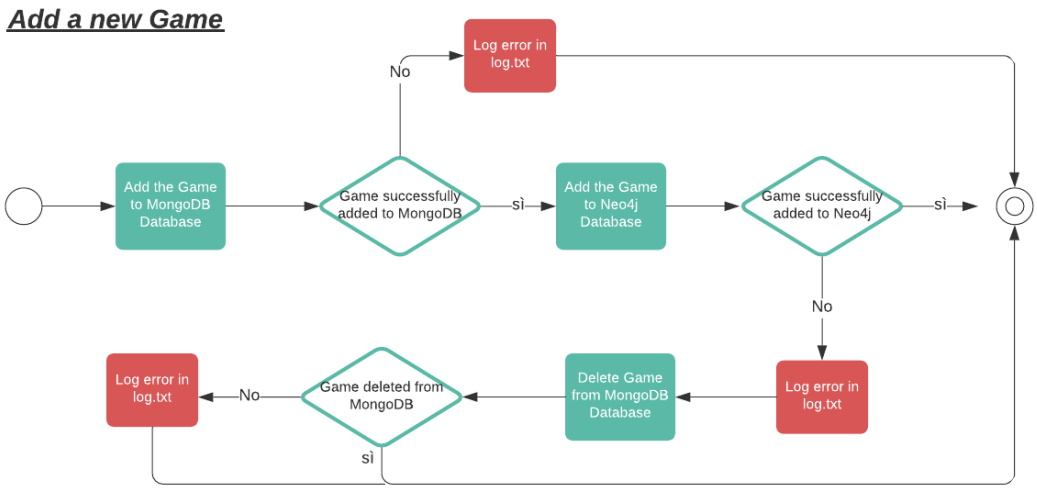
The **Article** entity is also present in both databases, the reasons are once again the same. The details of an **Article** are stored in **MongoDB**, for the reasons described above, but it is necessary to store the Article vertex also in Neo4j in order to handle the

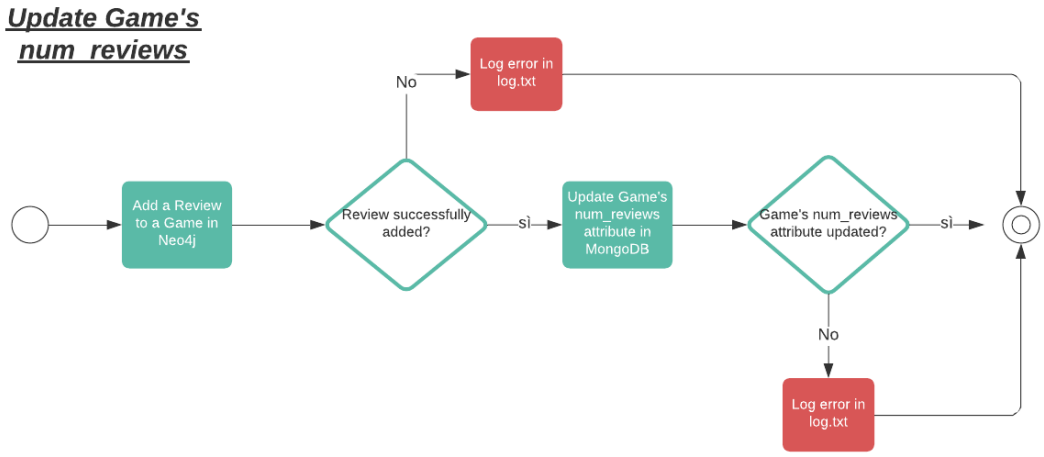
functionality of leaving a **Like** or a **Comment** on an **Article** by the **User**. The ***id*** and ***title*** properties are sufficient for this purpose.
Through **Neo4j** the application suggests articles to users based on their favorite categories and the **influencers** they **follow**, in fact an article is connected to the **Games** which it talks about and to its author. In **MongoDB** we need to report in the **Article** collection, 3 redundant attributes, ***num_comments***, ***num_likes*** and ***num_dilikes***, in order to efficiently execute the *Analytics* we designed, and to be able to improve the sorting and research processes. In **MongoDB** it is necessary to specify the author's ***Username***.

3.4.1 Cross-Database Consistency Management

As described in the previous paragraph we have 3 datasets distributed between the two databases which are **Games**, **Articles** and **Users**. The consistencies must be managed when we execute an *add*, a *delete* or an *update* operation about one of these entities. The latter is due to the presence of redundancies within the collection of **Articles** and **Games**, that is: *num_comments, num_likes, num_dislike, num_votes and avg_rating*, every time a **Review**, a **Comment**, a **Like** or a **Rating** is added or deleted these values must be updated. We will report below an example for adding, removing or updating of a **Game** following a like operation. What is described in the diagram is also true for **Articles** and **Users**.







3.5 Database Properties

3.5.1 Availability

Availability is a fundamental requirement for newer application including ours. In our case we would like to have a high availability to let users use our application smoothly. To increase

our application's availability we introduce a cache system to speed up the response time and let us shown articles and games homepage to users as soon as possible. Plus we used replicas to ensure a higher level of availability and we will discuss it in a more detailed way in next paragraphs.

3.5.2 Consistency

In our application we have to deal with a large amount of data and we would like to have data consistency in our application, this is contrast with availability that we preferred over consistency but we tried to make our data as consistent as possible adopting a moderate level of consistency between replicas. In particular we opted for a **read-your-writes consistency** for our information about games, users and articles since users expect that their new activities on the application will be updated immediately. Regarding read operations we chose a primary preferred approach, which allows us to label our system as only *eventually consistent*. Instead we chose a **casual-consistency** for social media management because a user usually sees immediately his own posts, but in any way it's ensured that posts in the same groups or reviews under a game will be always in order.
On write operation the control is returned to the application if the majority of the servers in the cluster have acknowledged it. Other servers' data will be eventually consistent.

3.5.3 Replicas

As anticipated in the previous paragraph we provided a replicas support for our application. Replicas are updated in a deferred way and are used for backup purpose but also for speeding up the response time of our application and to computed read operation faster. Replica allows use to overcome possible servers' breakdown and to guarantee fault tollerance and scalability, plus it allows us to guarantee a better availability. In our system there are 3 replicas for **MongoDB**, one in each machine of the cluster providing to us. For what concern **Neo4j**, it is present in a single instance on server 172.16.3.144. However, we provided a local **Neo4j** cluster so that it can be possible to test the application with the replication mechanism in both **MongoDB** and **Neo4j**.

3.5.4 Sharding

Sharding is a method for distributing data across multiple machines, which combined with **replication** ensures an **high availability** and **fast responses**. It is the method by which **Mongodb** supports *horizontal scaling*, it uses Sharding to support deployments with very large data sets and high throughput operations. We thought to some possible solution for the mechanism of **Sharding** which is not actually implemented in our database cluster.
A database can have a mixture of sharded and unsharded collections. Sharded collections are partitioned and distributed across the shards in the cluster, Unsharded collections are stored

on a primary shard. Each database has its own primary shard. So we decided to apply it only to **Articles** and **Users**. Careful consideration when choosing the shard key is necessary for ensuring cluster performance and efficiency. As **sharding strategy** we thought of the **Zone strategy**, because one common deployment pattern where zones can be applied is the following

- Ensure that the most relevant data reside on shards that are geographically closest to the application servers.

And this is exactly what we wanted to achieve. In sharded clusters, you can create zones of sharded data based on the shard key. You can associate each zone with one or more shards in the cluster. A shard can associate with any number of zones and each of these covers one or more ranges of shard key values for a collection.

- We want to use a Geographic sharding, so use the **User's country** of origin as the **Sharding key**.

Obviously not all the countries of origin of users will have the same turnout of users. For this reason, we have specific statistics to display the percentage of users who come from the 6 countries with most turnout and one that allows us to view the average access to the application for each of these countries. With this information, the groups of countries that will represent our zones can be organized.

Chapter 4

Implementation

4.1 Programming languages and frameworks adopted

For the development of the application various open-source frameworks and libraries have been used in order to make the development process faster and less error-prone.
For the dependency management we used the project management software **Maven**, which facilitated the installation of the packages needed by the other frameworks.
For the design of the graphical interface we used the Java extension, **JavaFX**, which allowed us to create the interface interactively through the *Scene Builder* program just injecting the dependencies from the GUI to the JavaFX application.
The simple caching mechanism implemented was created through Google's collections and methods, **Guava**, which is also used which is also used for the definition of the collections needed.
The cache, as well as another critical aspect of our application, the *password encrypter*, has been tested through **JUnit5**.
The application was entirely written in Java as regards the application logic, while Python was used for the scraping and subsequently formatting of data.

4.2 Package structure

As we already said, we wanted to ensure an high level of readability and maintainability so it has been decided to use a package structure. All the packages are structured by *layers* and are named according to their function architecturally trying to maintain them as simple and understandable as possible. We followed the convention of having the first character of the package name in the lower case to avoid conflicts with classes.

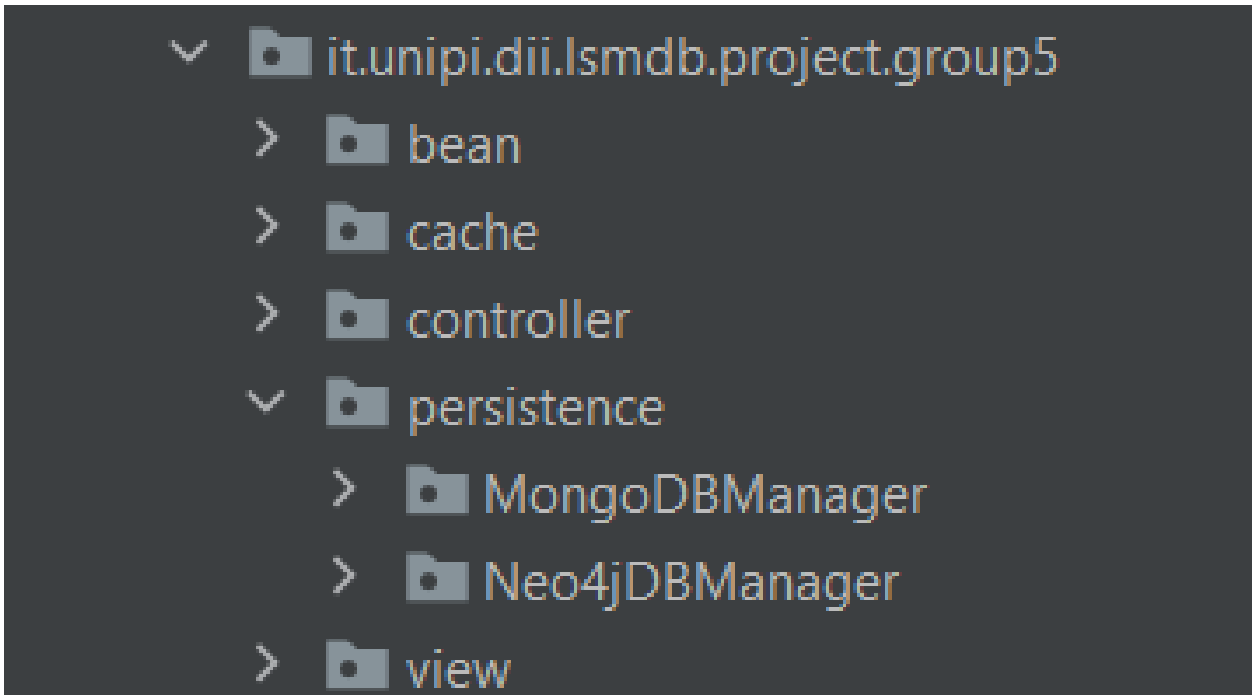


Figure 4.1: Package structure

4.2.1 Bean

The *bean* package contains few classes that we used as beans classes.

4.2.2 Cache

The *cache* package contains classes that are helpful to cache information.

4.2.3 Controller

The controller package containes the middleware classes that interact with both the frontend classes and the backend ones.

4.2.4 Logger

We provided also a logger because we thought would be useful to help the admin in recovering and managing any errors in the code. The logger records the errors that happen during runtime and provide also some informations that could be useful for error recovery.

```
1 public class Logger {
2
3
4     public static void warning(String text){
5         if (nullMsg(text)) {
6             return;
```

Class Name	Short Description
ActivityBean	Used to store useful information for computing analytics
AgeBean	Used to store information about users and their age useful during the analytics computation
ArticleBean	Used to store information about articles
CategoryBean	Used to store information to compute analytics about game's category such as *category's name, number of games based on that category and so on*
CommentBean	Used to store information about comments under an article
CountryBean	Used to store information about country used to compute analytics
GameBean	Used to store information about games
GroupBean	Used to store information 7 about groups
InfluencerInfoBean	Used to store information about influencer used to compute analytics
LikeBean	Used to store information about likes left on articles
PostBean	Used to store information about post published in groups
RatingBean	Used to store information about rates left to games
ReviewBean	Used to store information about reviews left under games
UserBean	Used to store information about users
VersatileUser	Used to store information about the most versatile influencer or normal user and it is used to compute analytics

Class Name	Short Description
ArticlesCache	Used to cache information about articles shown in the user's homepage
GamesCache	Used to cache information about games shown in user's games page

```
7         }
8
9         LoggerThread lt = new LoggerThread("[WARNING] " + text);
10        lt.start();
11
12    }
13
14    public static void error(String text){
15        if (nullMsg(text)) {
16            return;
```

Class Name	Short Description
AnalyticsDbController	Used to call the methods of backend classes to compute analytics
ArticlesPagesDBController	Used to call backend methods to recollect all the information related to articles that have to be shown
GamesPagesDBController	Used to call backend methods to recollect all the information related to games that have to be shown
GroupsPagesDBController	Used to call backend methods to recollect all the information related to groups that have to be shown
LoginSignupDBController	Used to call backend methods to handle the signup and login operations
UsersPagesDBController	Used to call backend methods to recollect all the information related to users that have to be shown

```
17          }
18
19          LoggerThread lt = new LoggerThread("[ERROR] " + text);
20          lt.start();
21
22      }
23
24      public static void log(String text){
25          if (nullMsg(text)) {
26              return;
27          }
28
29          LoggerThread lt = new LoggerThread("[LOG] " + text);
30          lt.start();
31
32      }
33
34      private static boolean nullMsg(String text) {
35          if (text == null) {
36              warning("NullPointerException avoided");
37              return true;
38          }
39          return false;
40      }
41
42  }
```

Every function create and start a new LoggerThread:

```
1  public class LoggerThread extends Thread{
2
3      private static String logFilePath="logs/log.txt";
4      private String msg;
5
6      LoggerThread(String msg){
7          this.msg=msg;
8      }
9
10     @Override
11     public void run(){
12         String newMsg = Instant.now().toString() + " " + msg + "\n
       ";
13         writeOnFile(newMsg);
14     }
15
16     private static synchronized void writeOnFile(String msg){
17         try {
18             Files.write(Paths.get(logFilePath), msg.getBytes(),
       StandardOpenOption.APPEND);
19         }
20         catch (IOException i){
21             i.printStackTrace();
22         }
23     }
24 }
```

4.2.5 Persistence

This package handle the interaction with our two databases, Neo4j and MongoDB. It is divided in two subpackages called *MongoDBManager* and *Neo4jDBManager*

MongoDBManager

The *MongoDBManager* package is used to handle the document DB MongoDB. It contains all the classes that interact with MongoDB.

Neo4jDBManager

The *Neo4jDBManager* package is used to handle the graphDB Neo4j. It contains all the classes that interact with Neo4j.

Class Name	Short Description
AnalyticsDBManager	Used to compute analytics using data stored on MongoDB
ArticleDBManager	Used to get, update, insert or delete information about articles stored on MongoDB
GameDBManager	Used to get, update, insert or delete information about games stored on MongoDB
LoginSignupDBManager	Used to get, update, insert or delete information stored on MongoDB about user to handle login and signup operations
MongoDBManager	Used to create or close a MongoDB connection and to get a collection
UserDBManager	Used Used to get, update, insert or delete information about users stored on MongoDB

Class Name	Short Description
AnalyticsDBManager	Used to compute analytics using data stored on Neo4j
ArticlesDBManager	Used to get, update, insert or delete information about articles stored on Neo4j
CommentsDBManager	Used to get, update, insert or delete information about comments stored on Neo4j
GamesDBManager	Used to get, update, insert or delete information about games stored on MongoDB
GroupsDBManager	Used to get, update, insert or delete information about groups stored on Neo4j
LikesDBManager	Used to get, update, insert or delete information about likes and dislikes stored on Neo4j
Neo4jDBManager	Used to create or close a Neo4j connection
RatingsDBManager	Used to get or insert information about ratings stored on Neo4j
ReviewsDBManager	Used to get, update, insert or delete information about reviews stored on Neo4j
UserDBManager	Used Used to get, update, insert or delete information about users stored on MongoDB

4.2.6 View

The *View* package contains all the classes used to implement the graphic interface. In particular, all the classes in this package are FXML controller where FXML is the framework used to implement our graphic interface. All the FXML files produced by FXML framework are collected in *resources* directory.

Class Name	Short Description
AddArticlePageView	FXML controller used to implement the page to insert new articles.
AddMember	FXML controller used to add a new member to a group
AdminGames	FXML controller used to implement the games page if the user is an **admin**.
AdminHomepage	FXML controller used to implement the homepage if the user is an **admin**
AdminUsers	FXML controller used to implement the users page if the user is an **admin**
ArticlePageView	FXML controller used to implement the articles page where are shown all the information about a specific article
GamePageView	FXML controller used to implement the games page where are shown all the information about a specific game
HomepageArticles	FXML controller used to implement the articles homepage where are shown the suggested articles or the filtered ones
HomepageGames	FXML controller used to implement the games homepage where are shown the suggested articles or the filtered ones
HomepageGroups	FXML controller used to implement the groups homepage where are shown the groups you have created or the ones in which you are just a member, but not the admin
HomepageUsers	FXML controller used to implement the users homepage where are shown people you are following, some suggested users you could follow. You can also filter your research
LoginPageView	FXML controller used to implement the login page
PostView	FXML controller used to show groups post. It open a new window where the post of a specific group are shown and where a member can write a new post
ProfileSettingsPageView	FXML controller used to implement the page where users can see and update their personale information
RemoveMember	FXML controller used to implement the window that is shown whenever the group admin decided to delete a member
SignupPageView	FXML controller used to implement the signup page

StaticsModeratorPageView	FXML controller used to implement the statics page for the moderator
UserFilterPageView	FXML controller used to implement the window that is shown whenever a user wants to filter users. In this window the filtered users are shown as a list of users

In this package there are also two sub-packages called: *javafxutils* and *tablebean*

Javafxutils

The *javafxutils* package contains only a class called ***PostPane*** that extends the javafx class ***Pane*** and it is used to implement a panel to show the posts of a group.

Tablebean

The *tablebean* is used as a bean to store information that have to been shown in a table later.

Class Name	Short Description
GroupMemberBean	use to store information about the members of a group
TableGroupBean	use to store information about a group

4.3 Analytic queries on MongoDB

Number of users from country

This query considers all the users in our application, gathering them according to their country and returning the number of users from each country

```
db.Users.aggregate([
  {$match:
    {"country": { "$exists": true, "$ne": "" , "$ne":null}}
  },
  {$group:
    {_id: "$country", count: {$sum: 1}}
  },
  {$project:{'country':'$_id',count:1, _id:0}}
  {$sort: {count:-1}},
  {$limit:6}
])
```

Get general information about a specific category

Given a specific category this query returns the number of ratings and the average rating considering all the games related to this category. It also returns the number of games related to the specified category.

```
db.Games.aggregate(
    [
        {$match: {category: {$ne: '', $ne:null} } },
        {$unwind: "$category"},
        {$group:
            {
                _id: "$category",
                totalGames: {$sum: 1 },
                avgRating: {$avg: "$avg_rating"},
                numRatings: {$sum: "$num_votes"}
            }
        }
        {$project:{'category':'$_id', totalGame:1, avgRating:1,
   numRatings:1, _id:0}}
    ]
)
```

Game distribution

For each category this query computes the total number of games based on the currently considered category and then will return the top 6 category for number of games based on them.

```
db.Games.aggregate(
    [
        {$match: {category: {$ne: '', $ne:null} } },
        {$unwind: "$category"},
        {$group:
            {
                _id: "$category",
                totalGames: {$sum: 1 },
            }
        },
    {$project:{'category':'$_id',totalGames:1, _id:0}}
   {$sort: {totalGames:-1}},
   {$limit:6}
   ]
```

```
)
```

Show Less Recent Logged Users

This query shows the users who has logged less recently in the application.

```
db.Users.aggregate(
    [
        {$match: {last_login: {$ne: '', $ne:null} } },
        {$sort: {last_login:1}},
        {$project:{username:1,last_login:1, _id:0}}
    ]
)
```

Get users for age

For each age this query returns the total number of games based on the currently considered age.

```
db.Users.aggregate(
    [
        {$match: {age: {$ne: '', $ne:null} } },
        {$group: {
            _id:"$age",
            count: {$sum: 1}
            }
        },
        {$project:{'age':'$_id',count:1, _id:0}}
    ]
)
```

Get activities statistics

This query groups all the users on the day of their last login and for each day count the number of people who have logged in the application.

```
db.Users.aggregate(
    [
        {$project:{ _id:0, date:{$toDate: '$last_login'}, username
    :1}},
        {$match: {date: {$ne:null} } },
        {$group: {
            _id:{month:{$month:'$date'},
```

```
                day:{$dayOfMonth:'$date'},
                year:{$year:'$date'}
            },
            count: {$sum: 1}
            }
        },
        {$project:{date:'$_id',count:1, _id:0}},
        {$sort:{'date.day':1}},
        {$sort:{'date.month':1}},
        {$sort:{'date.year':1}}

    ]
)
```

Daily average login for country

This query computes the number of login made in a single day and then considers users' country to compute the average number of login made by users from each country in a single day.

```
db.Users.aggregate(
    [
        {$project:{ _id:0, date:{$toDate: '$last_login'}, username
   :1, country:1}},
        {$match: {date: {$ne:null} } },
        {$match: {country: {$ne: '', $ne:null} } }
        {$group: {
            _id:{month:{$month:'$date'},
                day:{$dayOfMonth:'$date'},
                year:{$year:'$date'},
                country: '$country',
            },
            count: {$sum: 1}
            }
        },
        {$group: {
            _id: '$country',

            avg: {$avg: '$count'}
        }
        },
```

```
        {$project:{country:'$_id',avg:1, _id:0}},
        {$sort:{'avg':-1}}

    ]
)
```

Number of articles published in a specific period

Considering a specific period this query returns the top 10 influencers with the least number of articles published during the period previously specified

```
db.Articles.aggregate(
    [
        {$match: {
            timestamp:
                {$gte:<start>,
                $lte:<end>
                }
            }
        },
        {$group: {
            _id:'$author',
            count: {$sum: 1}
            }
        },
        {$project:{author:'$_id',count:1, _id:0}},
        {$sort:{'count':1}},
        {$limit:10}

    ]
)
```

Number of distinct games in articles published by an influencer in a specific period

This query returns the top 10 influencers with the least number of articles about distinct games published in the period previously specified. It considers only the influencers that have written about less than 10 different games.

```
db.Articles.aggregate(
    [
        {$unwind:'$games'},
```

```
        {$match: {
            timestamp:
                {$gte:'2020-12-30',
                $lte:'2021-01-30'
                }
            }
        },
        {$group: {
            _id:{
            author: '$author',
            game: '$games'
            }
            }
        },
        {$group: {
            _id:{
            author: '$_id.author',
            },
            count: {$sum: 1}
            }
        },
        {$match: {
           count:
                {
                $lte:10
                }
            }
        },
        {$project:{author:'$_id',count:1, _id:0}},
        {$sort:{'count':1}},
        {$limit:10}

    ]
)
```

Get number of likes for influencer

This query returns the top 3 influencers that have received the most number of likes considering all their articles

```
db.Articles.aggregate(
```

```
    [
        {$group: {
            _id: '$author',
            numLikes: {$sum: '$num_likes'}
            }
        },
        {$project:{author:'$_id',numLikes:1, _id:0}},
        {$sort:{'numLikes':-1}},
        {$limit:3}

    ]
)
```

Get number of dislikes for influencer

This query returns the top 3 influencers that have received the most number of dislikes considering all their articles

```
db.Articles.aggregate(
    [
        {$group: {
            _id: '$author',
            numDislikes: {$sum: '$num_dislikes'}
            }
        },
        {$project:{author:'$_id',numDislikes:1, _id:0}},
        {$sort:{'numDislikes':-1}},
        {$limit:3}

    ]
)
```

4.4 Most relevant queries and Suggestion on Neo4j

Influencers who wrote articles about the largest number of different game categories

```
MATCH   (u:User {role:"influencer"})-[:PUBLISHED]->(a:Article)-
        [:REFERRED]->(g: Game)
RETURN  u.username AS influencer,
        COUNT (DISTINCT g.category1) AS numeroCategorie
```

```
ORDER BY numeroCategorie DESC
LIMIT 3
```

Standard users who wrote reviews on the largest number of different game categories

```
MATCH (u:User {role:"normalUser"})-[:REVIEWED]->(g: Game)
RETURN u.username AS username,
COUNT(DISTINCT g.category1) AS numeroCategorie
ORDER BY numeroCategorie DESC
LIMIT 3
```

List of games suggested in the games section of a user based on his favorite categories

```
MATCH (g:Game),(u:User)
WHERE u.username=$username AND ((g.category1 = u.category1
    OR g.category1 = u.category2)
    OR (g.category2 = u.category1 OR
    g.category2 = u.category2))
RETURN g,u LIMIT $limit
```

Search suggested users to follow based on friends of friends

This query returns the list of suggested users to follow based on friends of friends, with param *role* that specify if query have to search influencers or standard users.

```
MATCH (me:User{username:$username})-[:FOLLOW]->(friend:User),
                    (friend)-[:FOLLOW]->(me), (tizio:User{role:$role
    })
WHERE NOT((me)-[:FOLLOW]->(tizio))
        AND (tizio)-[:FOLLOW]->(friend)
        AND (friend)-[:FOLLOW]->(tizio)
        AND NOT tizio.username=$username
RETURN tizio.username AS suggestion
LIMIT 6
```

Search suggested users to follow based on favorite categories of the user

This query returns the list of suggested users to follow based on favorite categories of the user with param *role* that specify if query have to search influencers or standard users.

```
MATCH (ub:User{username:$username}),(ua:User{role:$role})
WHERE NOT(ua.username=$username)
      AND NOT (ub)-[:FOLLOW]->(ua)
      ((ub.category1=ua.category1 or ub.category1=ua.category2)
      OR (ub.category2=ua.category1 or ub.category2=ua.category2))
RETURN ua.username AS suggestion
LIMIT 6;
```

Retrieve the 4 influencers with the highest number of followers

This query returns the 4 influencers with the highest number of followers order by followers descending.

```
MATCH (u:User)-[f:FOLLOW]->(u2:User{role:"influencer"}),
(me:User{username:$username})
WHERE NOT (me)-[:FOLLOW]->(u2)
RETURN u2.username AS suggestion ,COUNT(f) AS quantiFollowers
ORDER BY quantiFollowers DESC
LIMIT 4
```

List of articles suggested in a user's home if a user follows at least one influencer

This query returns the articles suggested for a user who follows at least one influencer, returns a certain number of articles published by the influencers that it follows sorted by publication timestamp.

```
MATCH (u:User{username:$username})-[f:FOLLOW]->
      (i:User{role:"influencer"})-[p:PUBLISHED]-(a:Article)
RETURN a, i, p
ORDER BY p.timestamp
LIMIT $limit
```

List of articles suggested in a user's home if a user doesn't follow any influencer

This query returns the suggested articles for a user who does not yet follow any influencer, the suggestion is based on its favorite categories.

```
MATCH (i:User)-[p:PUBLISHED]->(a:Article)-[r:REFERRED]->(g:Game),
    (u:User)
WHERE NOT(i.username = u.username)
        AND u.username=$username
        AND ((g.category1 = u.category1
        OR g.category1 = u.category2)
        OR (g.category2 = u.category1 OR g.category2 = u.category2
    ))
RETURN a,i,p
ORDER BY p.timestamp
LIMIT $limit
```

Chapter 5

User guide

When the application is launched, it will be possible to login using a previously registered account, or register by entering the requested information.

After logging in, you are directed to the main screen, the home page containing the 6 most suggested articles for the user.
If it is the user's first access, the articles suggested will belong to the categories indicated by the user when registering.
Later, if the user starts following influencers, on the homepage the articles published by those influencers will appear, otherwise the articles suggested by categories will continue to be shown.

Home screen changes depending on the type of user logging in.

The games homepage shows the 6 games most suggested for the user based on the categories he/she choose at the time of registration and it is possible, as for the articles, to filter or sort the games according to various criteria.

The groups homepage shows the groups that have been created by the user and those of which is member, allowing to perform various actions on the groups such as leaving the group, writing a post or adding a member.

The users homepage shows the the followed users, the suggested one and also the suggested influencer. The user can also filter other users by their username or by type.

The article page allow the user to read the information written by an influencer about

username	password	ruolo
eifelmartin	cane	admin
yanawhy	cane	moderator
thetalkingstove	cane	influencer
ubik	cane	normalUser

Table 5.1: Some users available on the database.

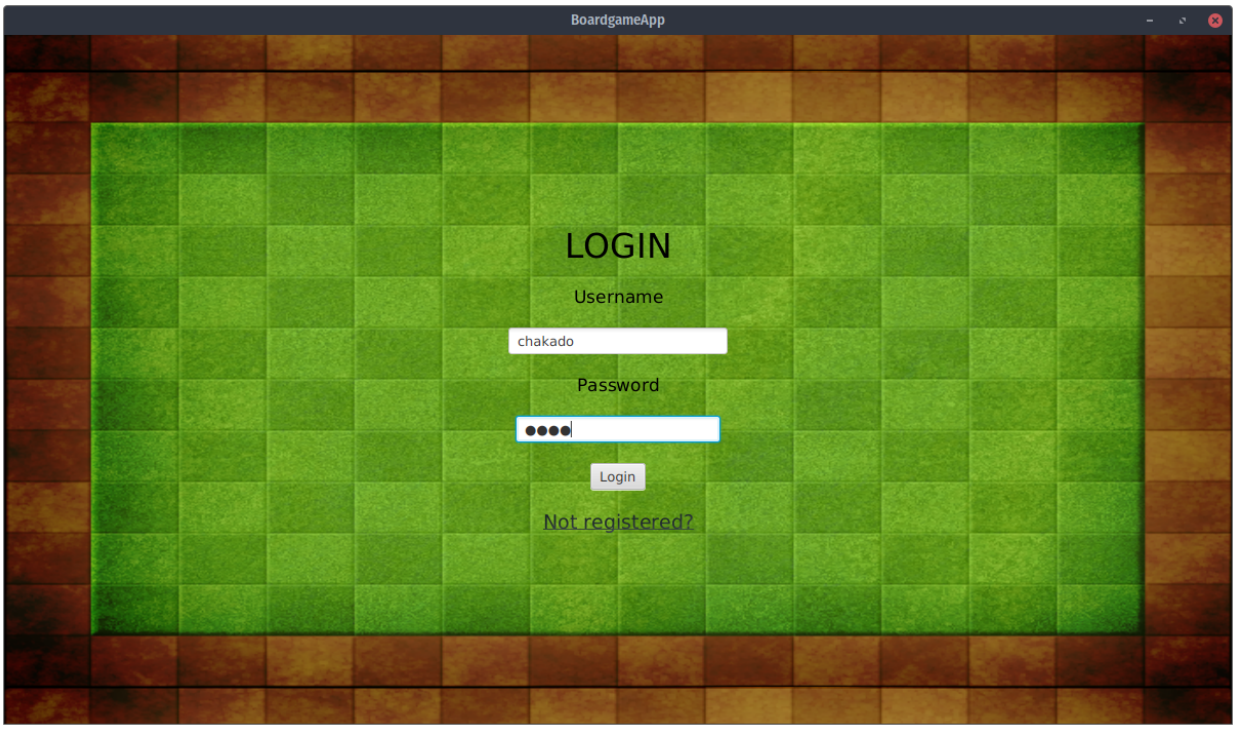


Figure 5.1: Login page.

Figure 5.2: Articles homepage with some suggested articles for the user.

a game and allows him also to write a comment, add a like or a dislike and can be entered just by clicking on the rectangle of the suggested article.

The game page allows the user to read the description and other informations about a game. The user can also write a review about a game or give a rating.

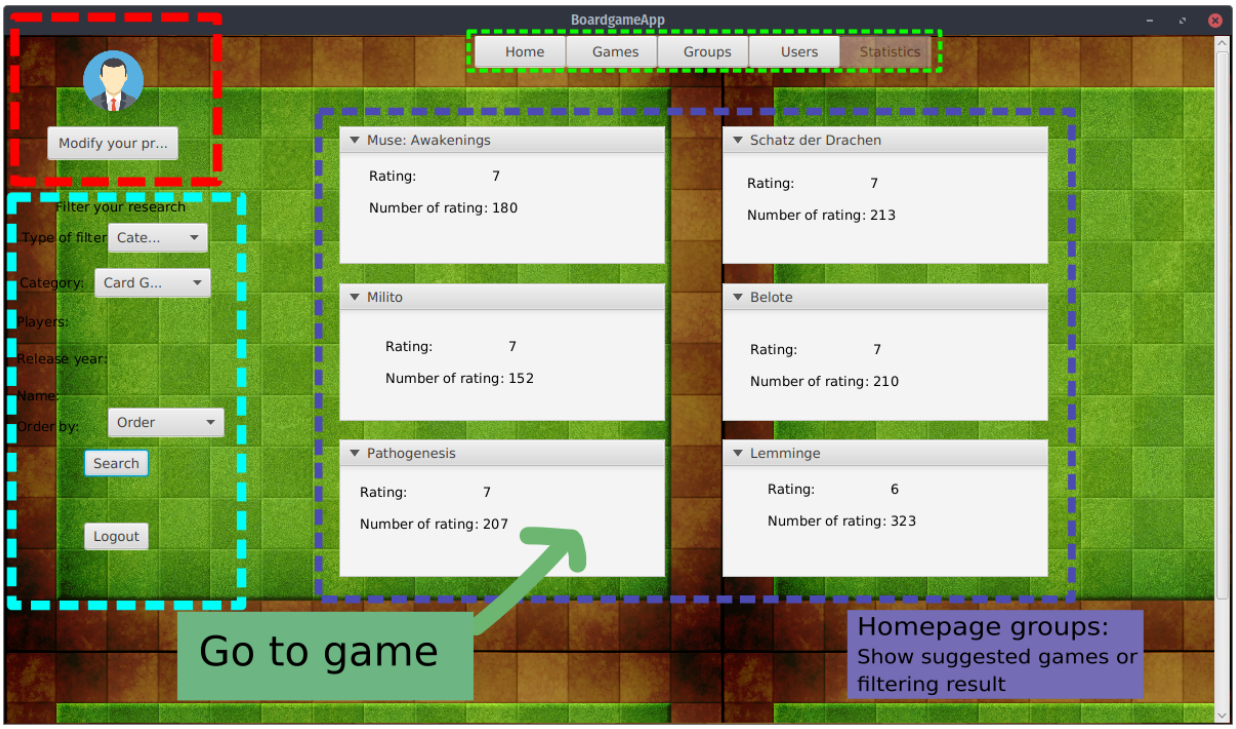


Figure 5.3: Games homepage with some suggested games for the user.

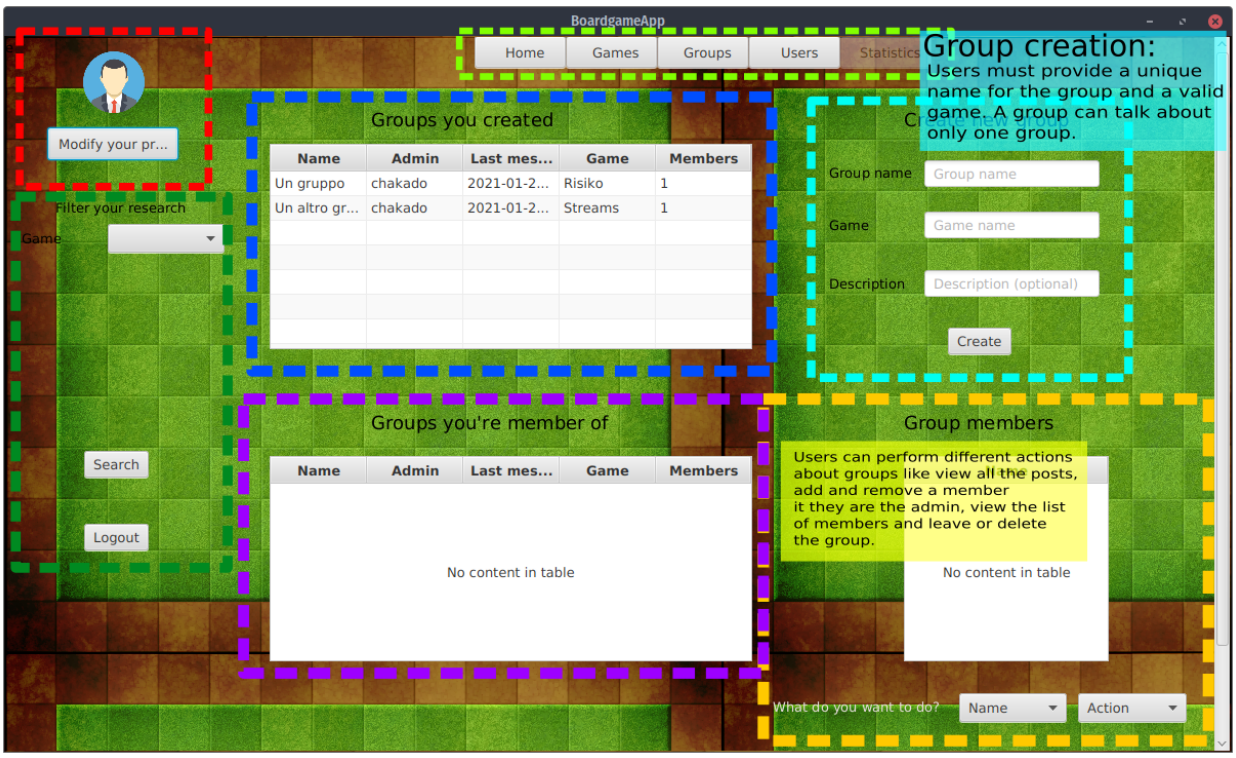


Figure 5.4: Groups page, users can perform different actions if they are admins or only members.

5.1 Admin guide

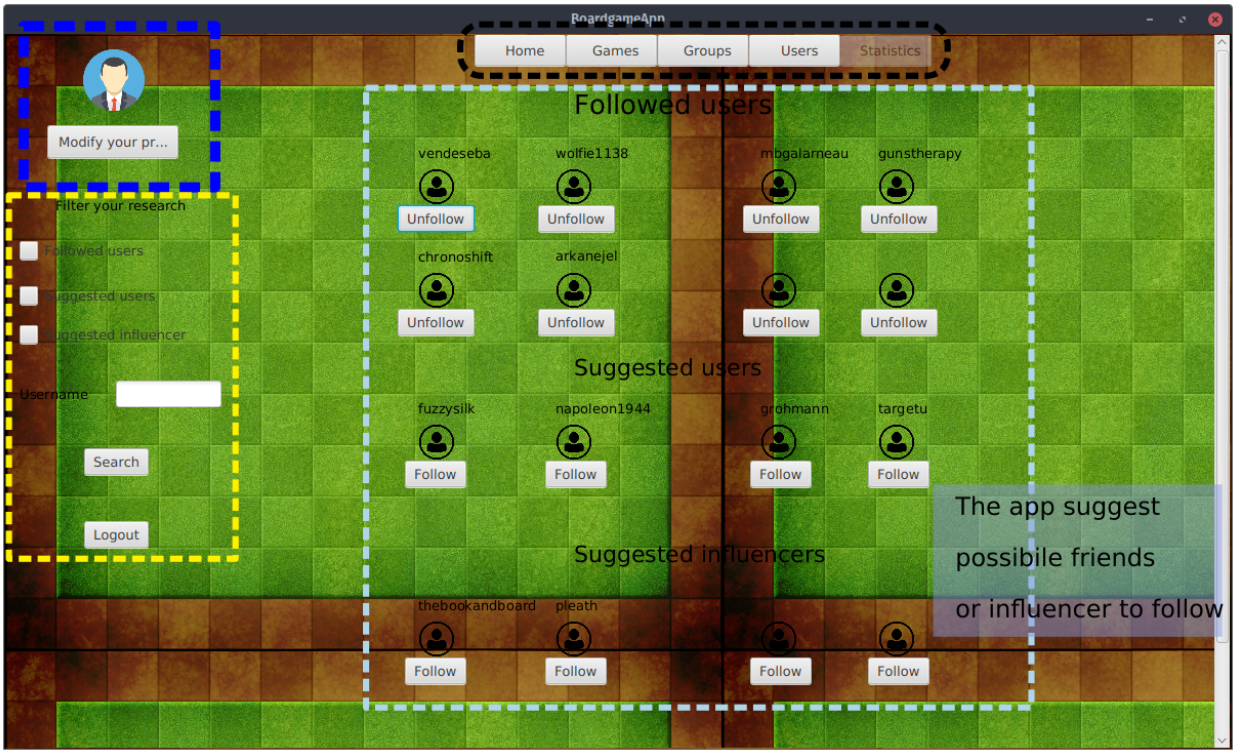


Figure 5.5: Friends page, users can follow or unfollow users.

Figure 5.6: Article page, users can leave comments, like or dislike.

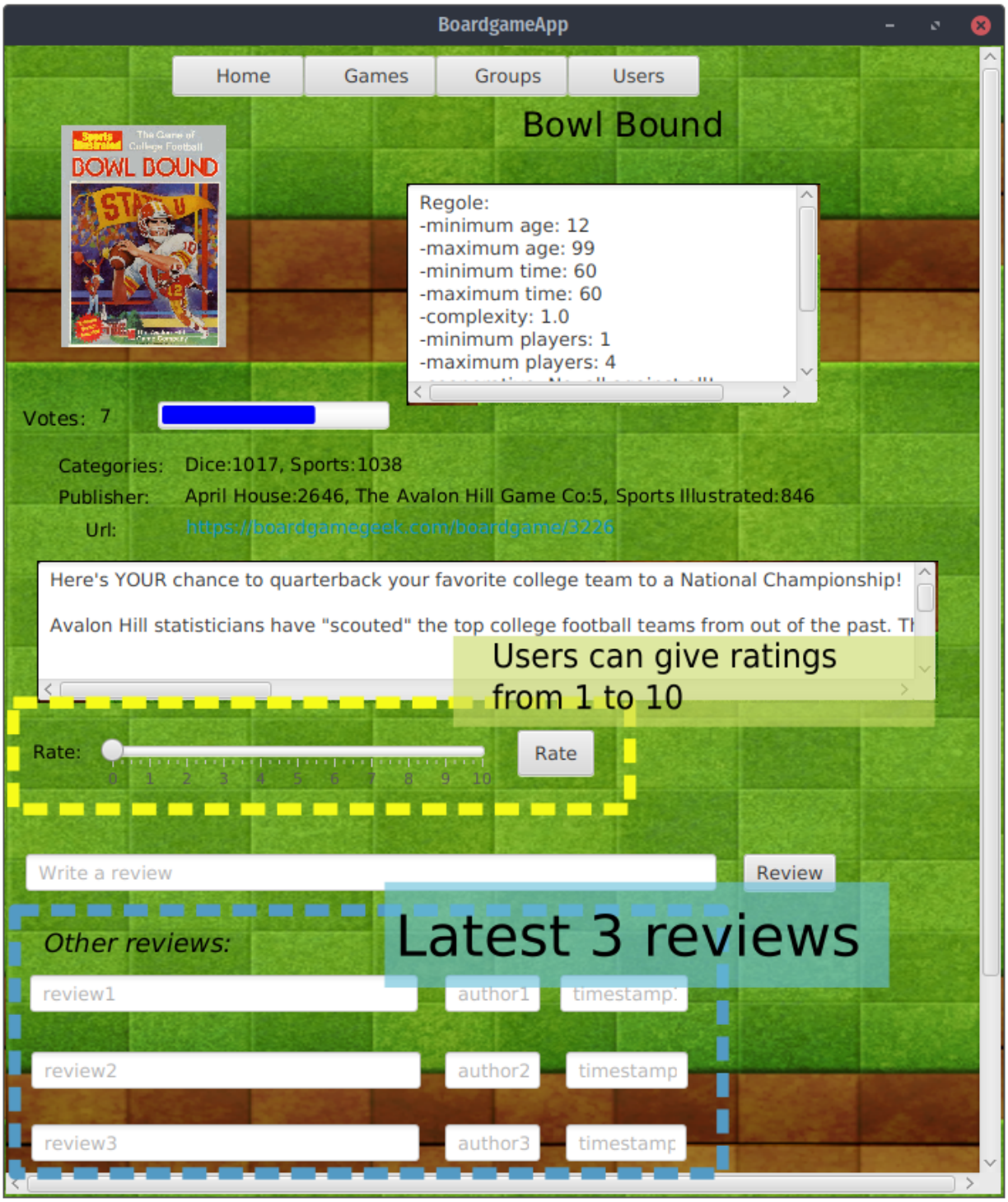


Figure 5.7: Game page with details about a game. Users can leave ratings and reviews.

Figure 5.8: Homepage of an admin.

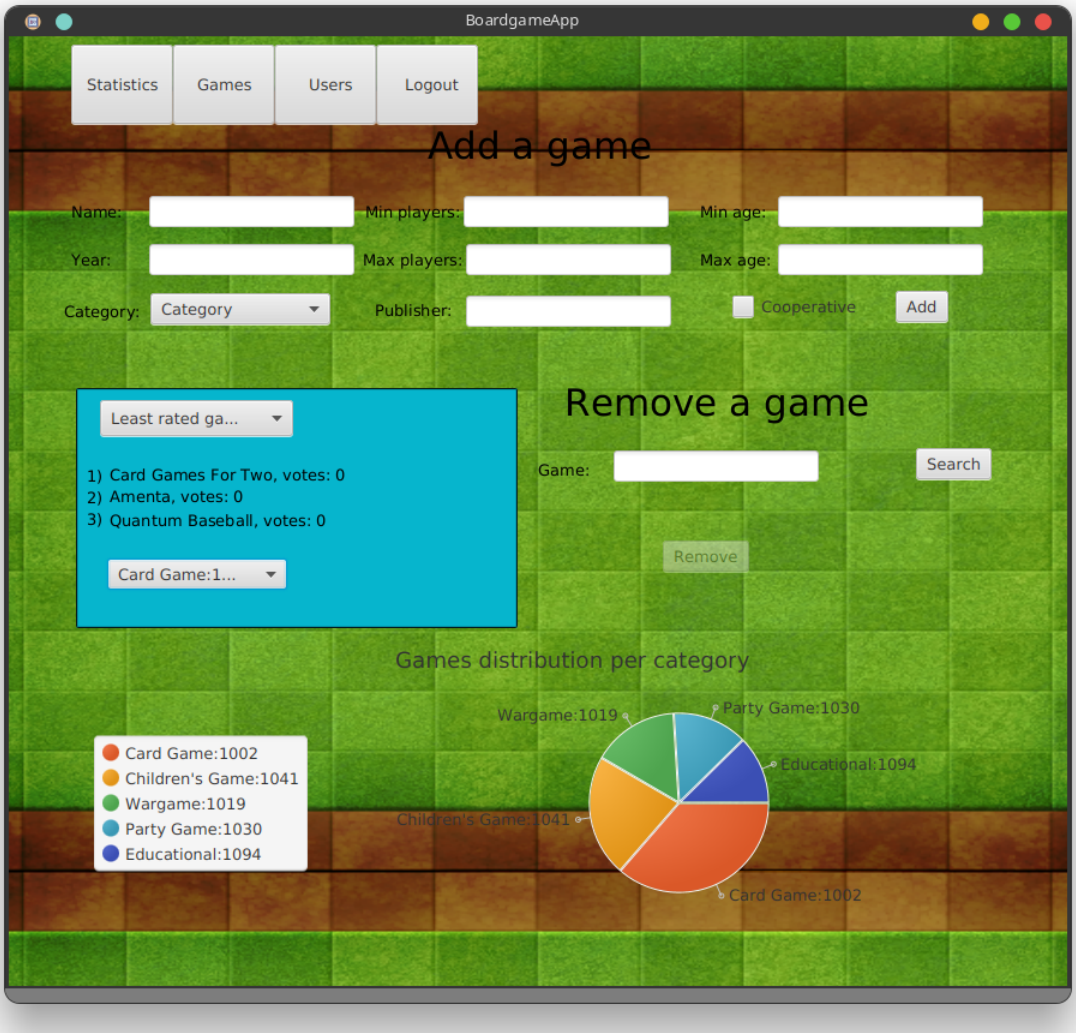


Figure 5.9: Game page of an admin.

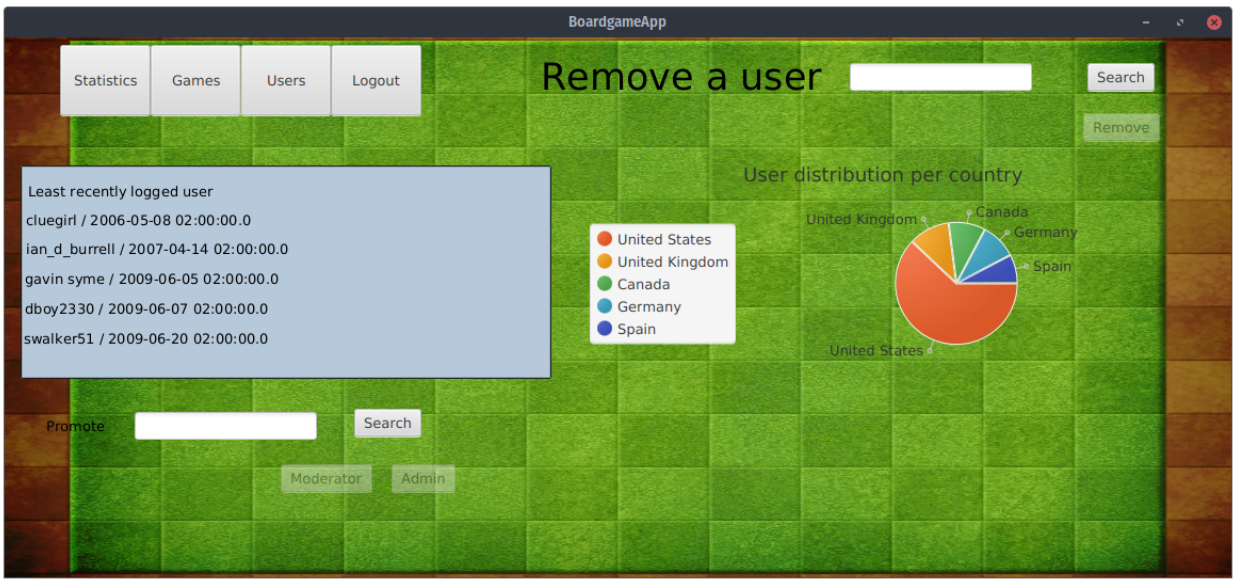


Figure 5.10: User page of an admin.

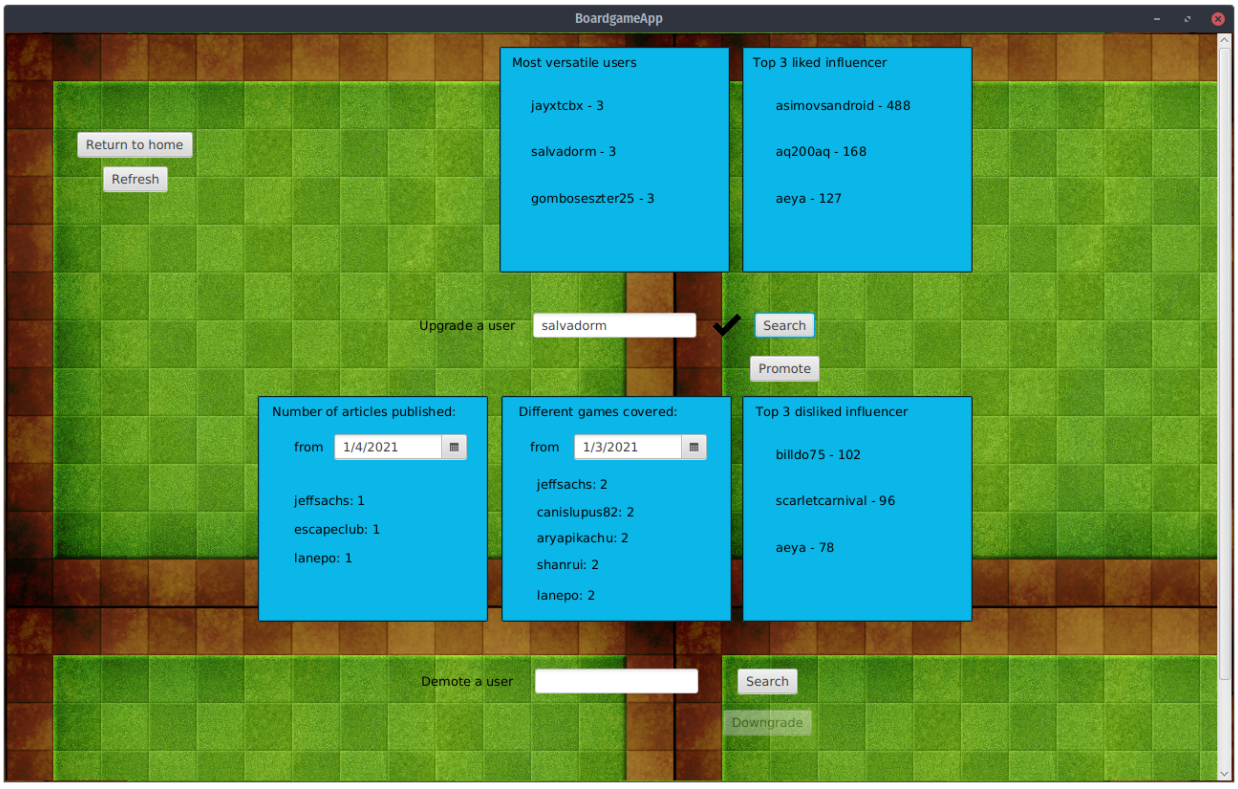


Figure 5.11: Moderator Analytics Page.