UNIVERSITÀ DI PISA

MSc Computer Engineering
Large-Scale and Multi-Structured Databases

BoardGame-Café Application

(Social Network)

Group Members:
Francesco Bruno
Gaetano Sferrazza
Marco Imbelli Cai

Academic Year: 2024/2025

Table Of Contents

Introduction

Board-Game Café is a social networking application designed for board games enthusiasts that provides several functionalities for discovering new boardgames, staying up-to-date on people's opinions about their favorites, make new friends based on common boardgames interests, and much more.

The application can be used by everybody: a ***non-registered*** *user* will be able to browse in read-only mode the contents of the social network, and to fully experience the app's functionalities, he'll need an account.

A ***registered user*** can browse in a large number of boardgames with the possibility to learn about their details, review and rate them, and write posts about them. Being the application structured as a social network, users are able to interact with each other: a user can follow other users, and comment or likes other users' posts.

Admins can manage Users, Boardgames, Posts, Comments and Reviews. They also have access to the usage analytics of the application. Moreover, Admins can decide to take corrective actions towards users that do not respect the guidelines of the application, and thus decide to delete their content and/or ban them.

Feasibility Analysis

The development of any data-driven application begins with a comprehensive feasibility study to evaluate the strengths, weaknesses, limitations, and potential opportunities of the project. This study not only clarifies the scope of the application but also helps identify challenges and future directions.

In this case, the initial stage involved assessing the viability of obtaining, cleaning, and linking datasets from different sources to populate multiple database management systems (DBMS): MongoDB and Neo4J. This chapter provides a detailed account of the data preprocessing pipeline and the methodologies used to achieve a consistent, integrated dataset that satisfies the application's requirements.

The first task that has been carried out has been a feasibility analysis to determine the practicality of the idea and its implementation. Key considerations included:

- **Data availability**: identifying reliable sources for datasets containing relevant information.
- **Integration challenges**: anticipating difficulties in linking datasets from different origins.
- **Consistency and quality**: ensuring the datasets were clean, complete, and suitable for the intended purpose.

- **Future scalability**: evaluating how the preprocessing pipeline could adapt to accommodate new data or additional features in the application.

This analysis highlighted the challenges of working with heterogeneous data sources and guided the development of a robust preprocessing pipeline to address these challenges effectively.

Web Scraping for Dataset

The datasets used for this project were obtained from three distinct platforms:

1. **BoardGameGeek**: contained user reviews and metadata for board games (source: Kaggle - BoardGameGeek Reviews Dataset)
2. **Reddit subreddit (r/boardgames)**: included posts and comments from board game enthusiasts (source: Reddit r/boardgames)
3. **RandomUser.me**: provided randomly generated user profiles (source: RandomUser API)

These sources were chosen for their relevance to the application's domain and the broad volume of data they offered. However, at this stage, the raw datasets still required significant preprocessing to meet the application's requirements.

Data Pre-Processing

The preprocessing phase was crucial in transforming the raw datasets into a consistent and usable form. The steps that were taken to carry out this task are described in this section.

Each dataset used different formats for user identifiers. To establish connections between the datasets, the following steps have been completed:

- Mapped usernames from the *RandomUser* dataset to match those of post authors, comment authors, and review creators in the other datasets.
- Replaced existing IDs with MongoDB-compatible IDs (e.g., *ObjectId* format), using ad-hoc Python scripts.

The datasets were cleaned to improve their quality and usability:

- **Field removal**: eliminated attributes considered irrelevant to the application's scope.
- **Field addition**: introduced new attributes, essential for linking data or enhancing functionality.
- **Duplicate removal**: identified and eliminated redundant entries.

- **Attribute renaming**: renamed fields to ensure uniformity and readability across datasets.

One of the major challenges was ensuring that data from different sources could be integrated seamlessly. This required:

- Writing Python scripts to manipulate JSON files, replacing usernames and IDs to create meaningful relationships.
- Verifying the integrity of links between datasets to ensure no orphaned records remained.

While MongoDB uses JSON files for data storage, Neo4J required CSV files for bulk import. To maintain consistency, the following tasks were carried out:

- Extracted relevant properties from the cleaned JSON datasets and converted them into CSV format.
- Selected only the attributes necessary for optimal performance in Neo4J, such as node labels and relationship types.
- Ensured that the data in Neo4J mirrored the structure and consistency of MongoDB data, enabling cross-database queries and analysis.

The preprocessing pipeline resulted in:

- Cleaned and integrated datasets ready for MongoDB and Neo4J.
- Seamless relationships between user data, reviews, posts, and comments.
- Enhanced dataset usability, ensuring alignment with application goals.

Some difficulties arose during preprocessing, including:

- **Handling inconsistent data formats**: the problem was solved by writing modular Python scripts for flexible manipulation.
- **Large dataset size**: addressed using efficient data structures and processing techniques.
- **Maintaining data integrity**: achieved through iterative validation steps and testing.

By addressing these challenges, the final datasets met the desired specifications, enabling successful application development.

Development

This chapter provides an overview of the activities carried out during the project *development*, detailing the methodologies, tools, and technical choices adopted. We followed a structured approach, starting from the *requirements analysis* and progressing to the final *implementation* of the project. After defining the functional and non-functional requirements during the analysis phase, the focus shifted to the design of the utilized DBMSs, the software components, their integration and testing. Subsequently, the implementation phase translated the design solutions into code, leveraging specific frameworks and tools to ensure quality and maintainability.

Functional Requirements

The functional requirements provided by the application, subdivided in the three main actors sets, are the following:

1. Unregistered user
    - Sign-up
    - Login
    - Browse posts (a limited subset)
    - Browse boardgames (a limited subset)
2. Registered user
    - Login and Logout
    - View his/her personal profile
    - View his/her account details
    - Edit his/her account details
    - Delete his/her account
    - View the posts he/she wrote
    - View the reviews he/she wrote
    - Browse other users' profiles
    - Search for other users
    - Visualize another user's profile
    - Visualize another user's posts
    - Visualize another user's reviews
    - Follow/Unfollow another user
    - Show users that posted about boardgames he/she posted about too
    - Show users that enjoy the same posts as he/she does
    - Show influencers Users in the boardgame community
    - Browse other users' posts
    - Search for posts about a boardgame
    - Show posts by followed users

- Show posts liked by followed users
- Create a new post
- Open a post's details
- Comment an existing post
- Add/remove Like in existing post
- Delete a previously written comment
- Browse boardgames
- Search for a boardgame
- Show boardgames posted by followed users
- Show the top rated boardgames in a given year
- Show boardgames belonging to a given category
- Show a boardgame's details
- Create review into a boardgame
- Edit a previously written review
- Delete a previously written review

3. Admin
    - Login and Logout
    - Edit his/her account details
    - Show the countries with the highest number of users
    - Show the average age of users per country
    - Show the most active users in the application
    - Show the most posted and commented boardgames
    - Show the top commented post about a boardgame
    - Show the list of currently banned users
    - Browse boardgames
    - Search for a boardgame
    - Show the top rated boardgames in a given year
    - Show boardgames belonging to a given category
    - Add a new boardgame
    - Show a boardgame's details
    - Delete a boardgame
    - Edit a boardgame's details
    - Delete another user's review
    - Browse other users' posts
    - Search for posts about a boardgame
    - Open a post's details
    - Delete anther user's comment
    - Delete another user's post
    - Browse other users' profiles
    - Search for other users
    - Visualize another user's profile
    - Visualize another user's posts

- Visualize another user's reviews
- Show influencers in the boardgame community
- Delete another user's profile
- Ban another user
- Unban a previously banned user

Non-Functional Requirements

The non-functional requirements of the application are outlined in the following section:

- **Usability**: the application interface must be intuitive, with clearly labeled elements arranged in a logical layout.
- **Low latency**: the application must be responsive, ensuring quick loading times accessing the database and smooth interactions to provide a seamless user experience.
- **High availability**: the application should remain accessible and operational, even if some data is slightly outdated, ensuring continuous usability.
- **Tolerance to the loss of data**: To avoiding a single point of failure

CAP Theorem

In line with the non-functional requirements, the application is designed within the *AP* set of the CAP theorem, where *Availability* and *Partition Tolerance* were prioritized, sacrificing Consistency.

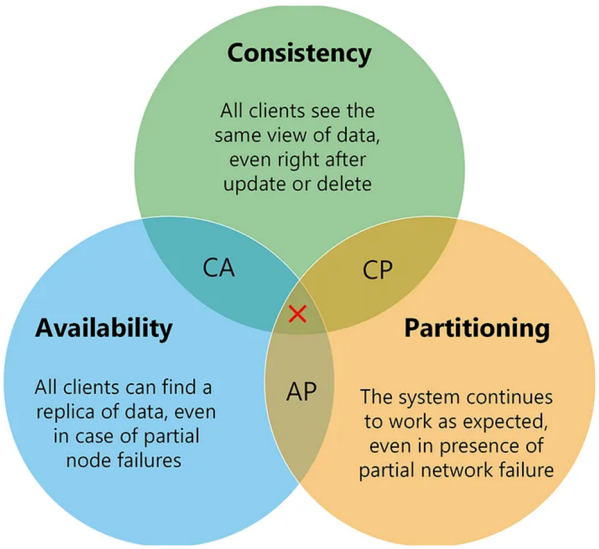

Figure 1: CAP theorem overview

Project Design

This chapter will go through the project's design choices, use cases, class diagrams and the chosen database architectures

Use Case Diagrams

Use case diagrams have been developed for the three main actors of the application, and they are reported in this paragraph. Note that, in *Figure 2* the *blue ovals* describe actions related to some kind of content (*posts*, *reviews*, *comments*) that can be carried out only by the *author* of said content (the *author* of some kind of content is simply the user that created such content).

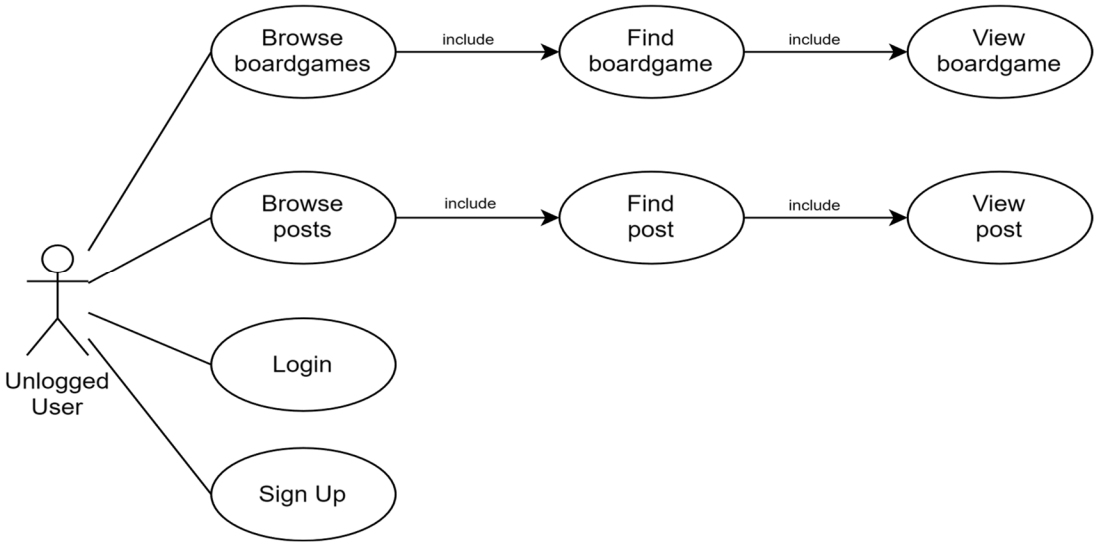

Figure 2: Unregistered (Guest) user use case

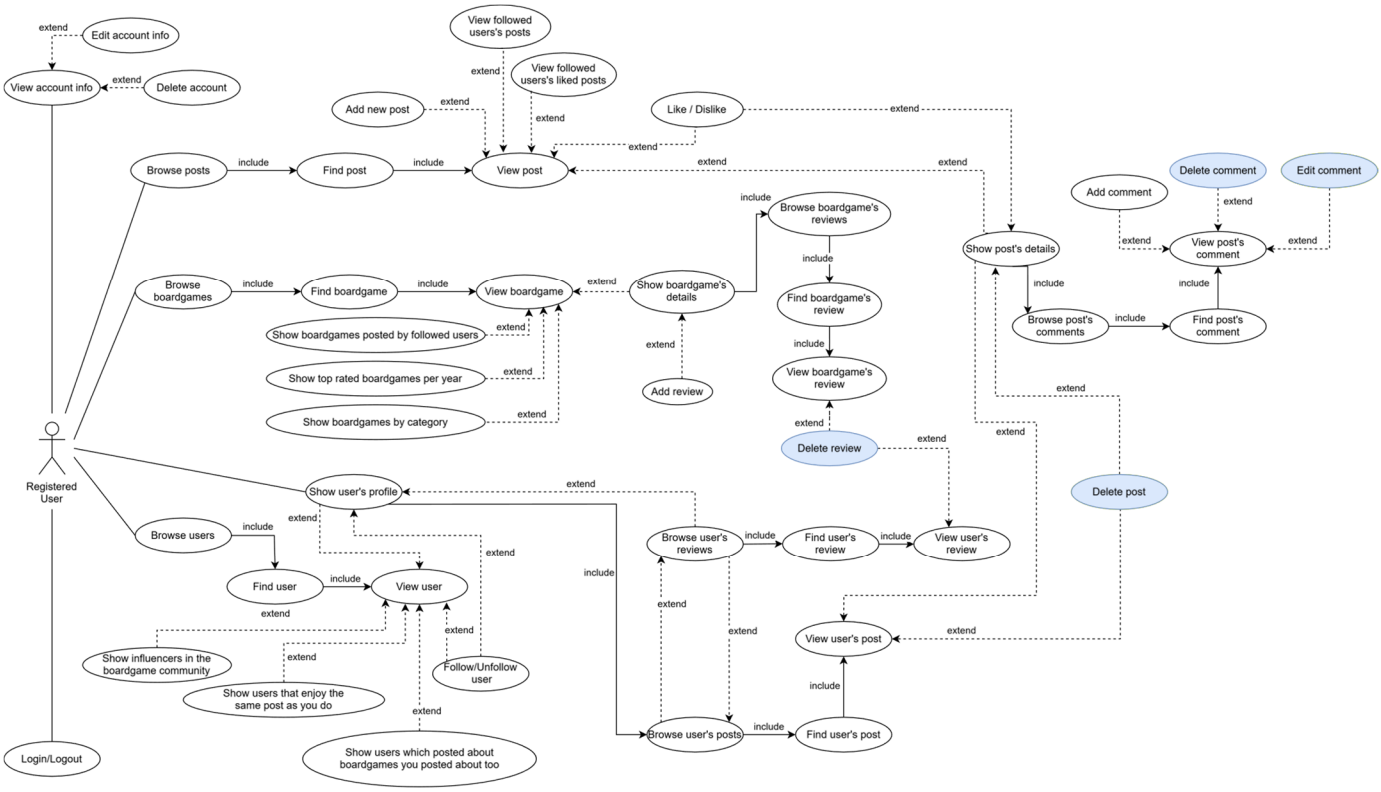

Figure 3: Registered user use case

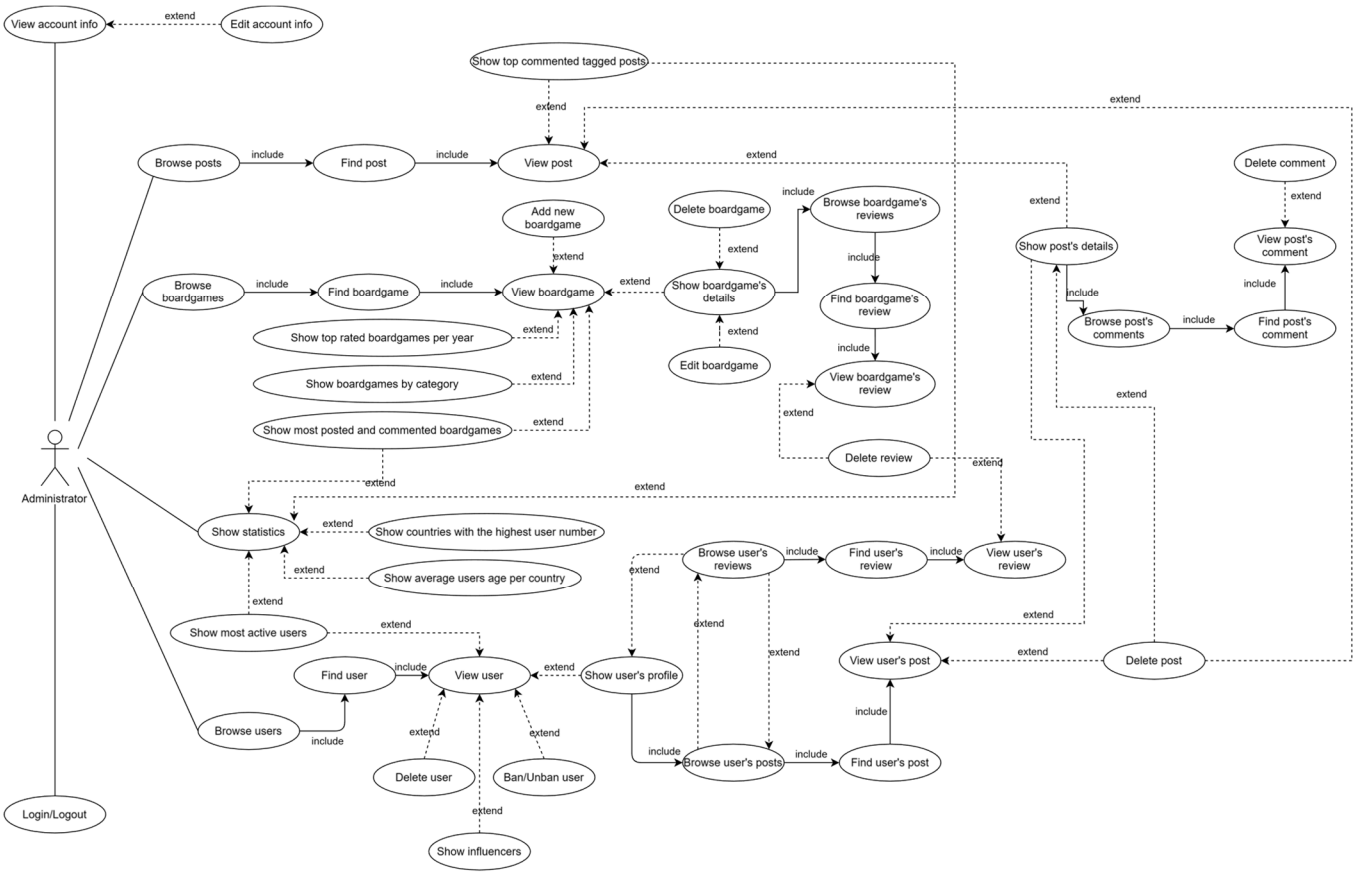

Figure 4: Administrator use case

Class Diagram

The UML class diagram is illustrated in this paragraph.

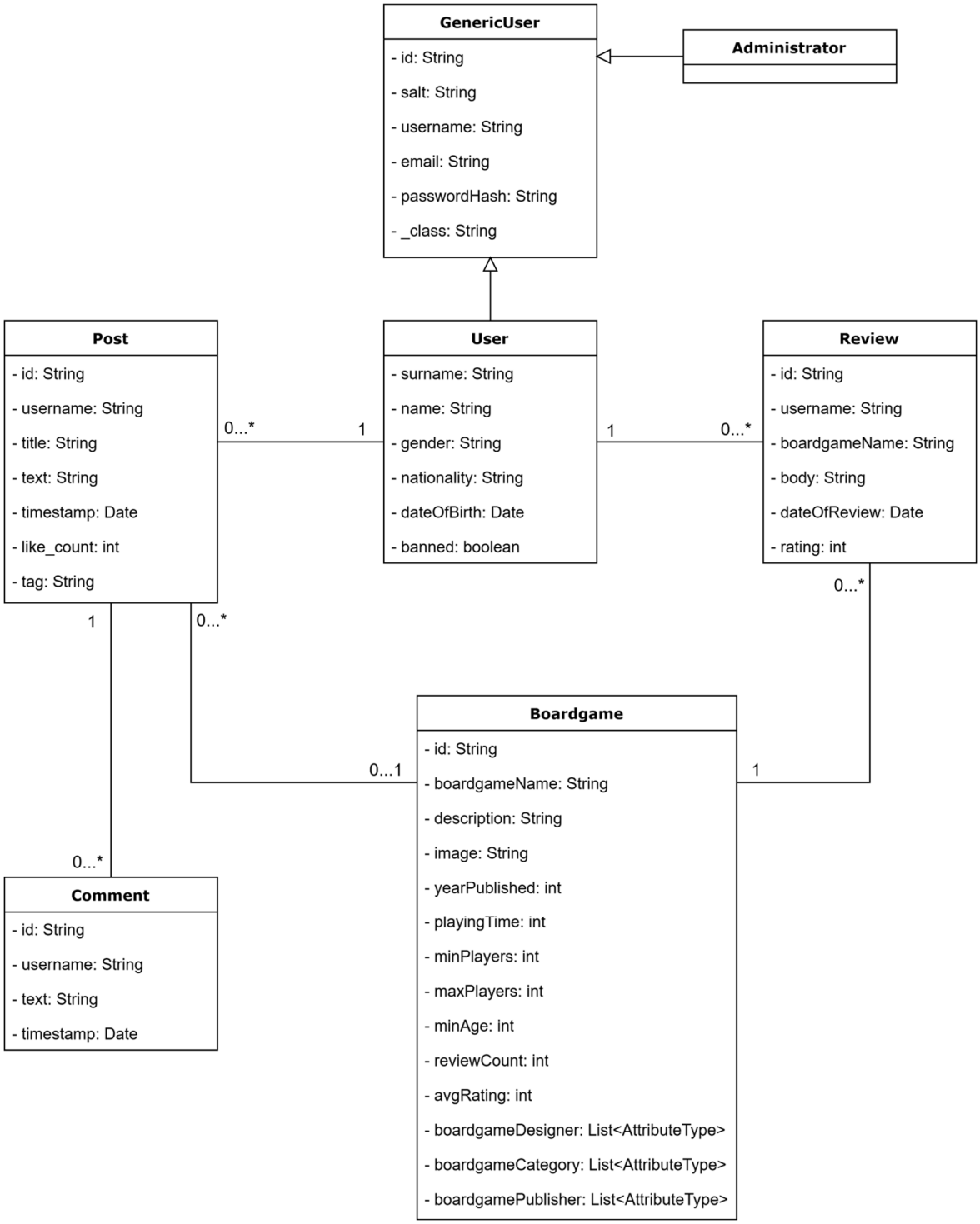

Figure 5: UML Class Diagram

The UML class diagram can be described as follows:

- a User can write from zero to many reviews
- a User can write from zero to many posts
- a User can write from zero to many comments
- a Post can be referred to one boardgame or none
- a Post can have from zero to many comments
- a Review must be referred to one boardgame
- a Boardgame can have from zero to many reviews

Databases

Two types of databases have been used:

- **MongoDB:** was chosen as a document-oriented database to ensure both low latency and flexibility in data storage. It allows for the seamless management of complex data structures, accommodating all relevant details for each entity. MongoDB is also optimized for handling intricate queries on these structures efficiently, making it ideal for the application's data requirements.
- **Neo4J:** on the other hand, was selected for its graph-based architecture, which excels in capturing and navigating relationships between entities, such as user interactions, likes, and follows. This approach enables the application to provide dynamic and context-aware suggestions, such as recommending relevant authors or podcasts based on these relationships.

Databases Design

Mongo DB: Collections and Documents

The following section describes the structure of the documents contained in the different MongoDB collections that have been used in the application.

Users Collection

The following is an example of a document retrieved from the *Users* collection, which consists of over 110 thousand documents.

```
{
  "_id": ObjectId('65a92ef56448dd9015694901'),
  "username": "whitekoala768",
  "email": "noah.lavoie@example.com",
  "name": "Noah"
  "surname": "Lavoie",
  "gender": "male",
  "dateOfBirth": { "$date": "1970-08-29T00:00:00.000Z" },
  "nationality": "Italy",
  "banned": false,
  "salt": "5zBLQf2O",
  "passwordHash":
"001bcffe78d4f36e17dcd7f3e5e340d18fcd4d46a7f15eeefcb1c7f03dd892c7",
  "_class": "user"
}
```

Since the application is primarily focused on boardgames and user interactions with them (e.g., writing reviews, voting, and participating in discussions), redundant fields like *reviewCount* (i.e., the number of reviews the user created) or *postCount* (i.e., the number of posts the user created) have not been included in this collection. This is because the database design reflects the centrality of the boardgames, where the main information pertains to the games themselves and their associated reviews, rather than user-specific details. This approach has helped in avoiding unnecessary complexity and in keeping the database optimized for the application's core features.

Reviews Collection

The following is a sample document extracted from the *Reviews* collection, which contains approximately 900 thousand documents.

```
{
  "_id": ObjectId('65a8f4d932bd16dc438261b6'),
  "boardgameName": "Carcassonne",
  "username": "redpanda520",
  "body": "Strategy, fun, beautiful components!  Just enough randomness to keep it interesting.",
  "rating": 8,
  "dateOfReview": {"$date": "2020-12-12T00:00:00.000+00:00"}
}
```

The *boardgameName* field is included in each review document in order to facilitate efficient querying when displaying the list of all the reviews for a given board game. This redundancy eliminates the need for multiple queries when fetching reviews for a specific game. In this way, all the reviews for a given boardgame can be quickly retrieved, which is a common use case in this application. For the same reason, the *username* redundance has been included too. These choices are guided by the application's common use cases and the need for fast, efficient queries.

Boardgames Collection

The following is an example of a document extracted from the *Boardgames* collection, which comprises more than 20,000 documents.

```
{
  "_id": ObjectId('65a8f90632bd16dc4390d9c2'),
  "boardgameName": "Pandemic",
  "image": "https://cf.geekdo-images.com/S3ybV1LAp-8SnHIXLLjVqA__original/img/IsrvRLpUV1TEyZsO5rC-btXaPz0=/0x0/filters:format(jpeg)/pic1534148.jpg",
  "description": "In Pandemic, several diseases have broken out simultaneously all over the world! The players are disease-fighting specialists [...],
  "yearPublished": 2008,
  "minPlayers": 2,
  "maxPlayers": 4,
  "playingTime": 45,
  "minAge": 8,
  "boardgameCategoryList": [...],
  "boardgameDesignerList": [...],
  "boardgamePublisherList": [...],
  "reviewCount": 2
  "avgRating": 7.5
}
```

In the design of this collection, an embedded document for the board game's reviews has not been included. This decision has been made because reviews are the most frequently used feature by users and embedding them could lead to document saturation. However, based on the functionalities of the application (and in particular, when displaying the details of a boardgame), two redundancies have been introduced in the *boardgames* collection's documents: *reviewCount* and *avgRating*. The first represents a counter of reviews that users have created for the given boardgame, while the second is an indicator of the average rating of such reviews. Such redundancies are small in size and take up minimal space in the database, ensuring that the overall memory usage remains low regardless of the number of reviews.
Of course, this additional data will require additional steps to be carried out to remain updated, but this is a good trade-off between memory usage and low latency.

Posts Collection

The following is an example of a document extracted from the *Posts* collection, which consists approximatively of 14.000 documents.

```
{
  "_id": ObjectId('65a930a56448dd90156b31ff')
  "username": "whitelion758",
  "title": "Carcassonne with my girlfriend turned into a fight",
  "text": "Hey guy's I was playing Carcassonne with my girlfriend 10 mins ago,
and in the middle of the game she did something really pissed me off [...],
  "tag": "Carcassonne",
  "timestamp": { "$date": "2022-03-24T09:42:41.000Z" },
  "like_count": 41,
  "comments": [...]
}
```

In the design of the *Posts* entity, the comments written by users under the specific post have been embedded directly within the post document. This approach allows the retrieval of the complete document in a single read operation, ensuring that the post and its associated comments are displayed together seamlessly, without any additional querying action. Embedding the comments ensures immediate availability whenever a post is retrieved, thereby improving the application's responsiveness and query efficiency.

Additionally, new comments are inserted at the head of the array. This allows the comment list to be pre-sorted, so that it can be displayed as-is in the application (i.e., in descending order, from the most recent to the oldest one).

An example of a *Comment* embedded document is shown below:

```
{
  "_id": ObjectId('67210ab54dede67f933f3980'),
  "username": "whitelion538",
  "text": "Nice game, really boring.",
  "timestamp": {"$date": "2024-10-29T16:17:57.621Z"}
}
```

Neo4J DB: Nodes and Relationships

Neo4J has been leveraged in the application to model and manage highly interconnected data. The database's graph-based structure allows the definition of entities as nodes and their connections as relationships, aligning seamlessly with the social and content-driven nature of the system. This approach enabled efficient traversal of relationships and supported advanced queries for personalized recommendations, community analysis, and user interactions.

Nodes

The nodes of the Neo4J graph are of the following types:

- ***Boardgame:*** represents an individual boardgame. It contains its *boardgameId* (aligned with the MongoDB *boardgameId* field of the documents in the *Boardgames* collection), *boardgameName*, *image* (URL to the boardgame's image), a truncated version of the boardgame description (mirroring the MongoDB description), and the *yearPublished*. These details are essential for displaying a well-organized view of all boardgames on the main page, enabling users to browse them efficiently.
- ***Post:*** represents an individual post made by a user within the application. It contains only its *postId* attribute, which is the same as the *postId* field in the (more detailed) MongoDB document.
- ***User:*** represents an individual user within the application. It contains the *userId* and the *username* attributes, both of which are used to identify the user across different sections of the application.

Relationships

In the employed graph, four key relationships are defined to interconnect the different entities:

- **User → FOLLOWS → User:** represents the connection between users, in which one user follows another user. This helps tracking the social interactions among users within the application.

- **User → WRITES_POST → Post**: represents the action of a user creating a post, allowing the association of posts with their respective author within the application.
- **Post → REFERS_TO → Boardgame:** links a post to the specific boardgame it discusses or references. This allows the association of user-generated content with the referred board games in the application, and the identification of which games are most frequently discussed within the social network.
- **User → LIKES → Post:** represents the action of a user liking a post. This helps in tracking user engagement and understand which posts are the most appreciated within the application.

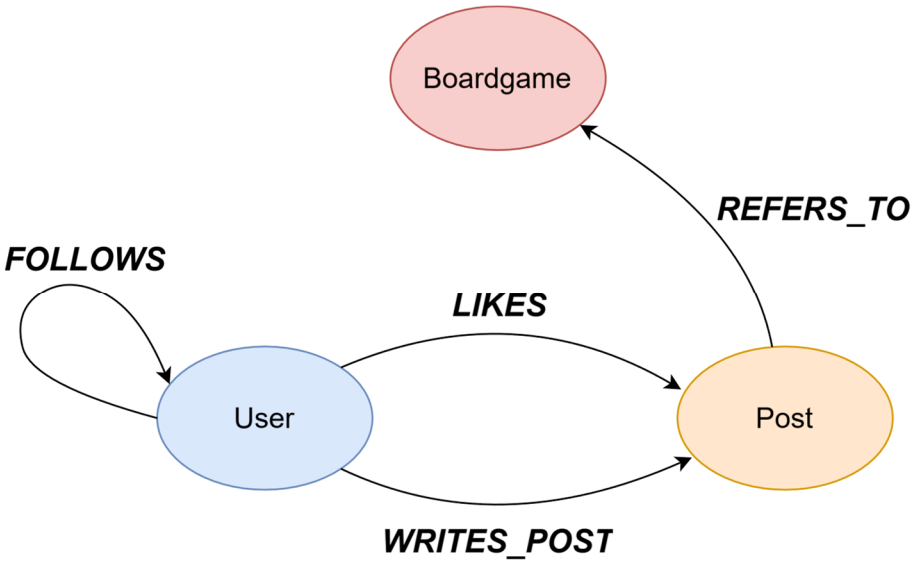

Figure 6: GraphDB relationships model

Below, an example of a Neo4J graph is shown, in which the different relationships within the application are shown:

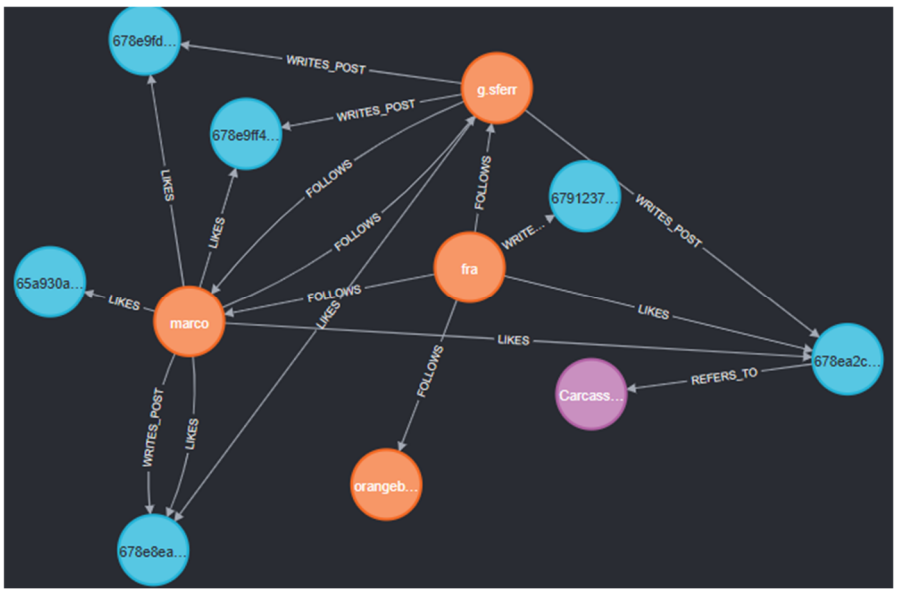

Figure 7: Neo4J snapshot showing the used relationships

Application Queries

Admin Queries

The use of MongoDB for the application required the definition of a set of queries tailored to address the operational and administrative needs of the system. Below is a list of the main queries organized by purpose and audience, along with a brief description of their functionality.

These queries provide analytical data useful for monitoring user activity and application behavior:

- **Show the countries with the highest number of registered users**: Identifies the countries with the largest number of registered users, useful for understanding the geographical distribution of users.
- **Show the average age of users by nationality**: Calculates the average age of users for each nationality, providing a detailed demographic overview.
- **Show Most Active Users**: Displays the most active users based on the number of interactions (reviews and their average frequency).
- **Suggest the most posted and commented boardgame**: Identifies the boardgame with the highest number of posts that have received the highest number of comments, highlighting the most actively discussed game in the application.
- **Suggest to top commented tagged post**: Retrieves the top posts associated with a specific boardgame (tagged) that have received the highest number of comments, showcasing the most actively discussed posts for a given boardgame.
- **Show all boardgames:** Displays a list of all boardgames, showing 12 at a time, sorted from the most recently added to the oldest.
- **Show the top 4 boardgames with the highest average review scores per year**: Displays the four boardgames with the highest average review scores for each year, offering insights into the most appreciated games for that selected year.
- **Show boardgames that belong to a specific category**: Allows filtering boardgames based on a specific category, useful for both users and administrators.
- **Show all posts:** Displays the most recent posts, presented in a paginated pattern with 10 posts per page. In addition, by inserting a specific tag (boardgame reference), all posts related to that specific boardgame will be listed.

- **Show all users:** Visualizes a list of all users in the system, presented in a paginated format for efficient browsing (10 at a time). Additionally, users can search for a specific user by entering their name.
- **Suggest Influencers in the boardgame community**: Identifies users with significant influence in the boardgame community, based on metrics such as the number of followers, posts, and average like count received on their own posts, highlighting key contributors to discussions.
- **Show banned users:** Displays a paginated list of users who have been banned, showing 10 users at a time. In this way, the administrator can efficiently monitor and manage banned accounts, maintaining a safe and respectful community environment.

User Queries

- **Show all boardgames:** Displays a list of all boardgames, showing 12 at a time, sorted from the most recently added to the oldest.
- **Suggest boardgames posted by followed users:** displays the boardgames that have been posted about by users the current user follows, providing insights into the games discussed within their social circle.
- **Top rated boardgames per year:** Retrieves the top-rated boardgames for a specified year, based on their average rating, allowing users to discover the most acclaimed games released or reviewed during that period.
- **Show boardgames that belong to a specific category**: Allows filtering boardgames based on a specific category, useful for both users and administrators.
- **Show the top 4 boardgames with the highest average review scores per year**: Displays the four boardgames with the highest average review scores for each year, offering insights into the most appreciated games for that selected year.
- **Suggest posts by followed user**: Displays posts created by users that the current user follows, enabling the user to stay updated with content shared by their network.
- **Suggest posts liked by followed users:** Retrieves posts that have been liked by users the current user follows, providing insights into the preferences and interactions of their network.
- **Show all posts:** Displays the most recent posts, presented in a paginated pattern with 10 posts per page. In addition, by inserting a specific tag

(boardgame reference), all posts related to that specific boardgame will be listed.

- **Show all users:** Visualizes a list of all users in the system, presented in a paginated format for efficient browsing (10 at a time). Additionally, users can search for a specific user by entering their name.
- **Suggest Users which posted about boardgames you posted about too**: Identifies and displays users who have created posts related to the same boardgames as you, highlighting connections and shared interests within the community.
- **Suggest Users that enjoy the same posts as you do**: Shows users who have liked the same posts as you, emphasizing shared interests and encouraging interaction within the community.
- **Suggest Influencers in the boardgame community**: Identifies users with significant influence in the boardgame community, based on metrics such as the number of followers, posts, and average like count received on their own posts, highlighting key contributors to discussions.

Distributed Database

To meet the demands of the application, a distributed database system has been exploited to efficiently handle large volumes of data and high traffic loads while maintaining optimal performance. This architecture allows the database to be distributed across multiple servers or nodes, ensuring both redundancy and quick data access. Finally, the risk of bottlenecks is mitigated and the system's resilience is enhanced.

Replica Set

To fulfill the non-functional requirements of our application, we implemented a replication strategy specifically for MongoDB. In detail, three replicas of the MongoDB database were set up to ensure high availability and fault tolerance, while only one replica for the Neo4J part, as it is shown in the below architecture's image

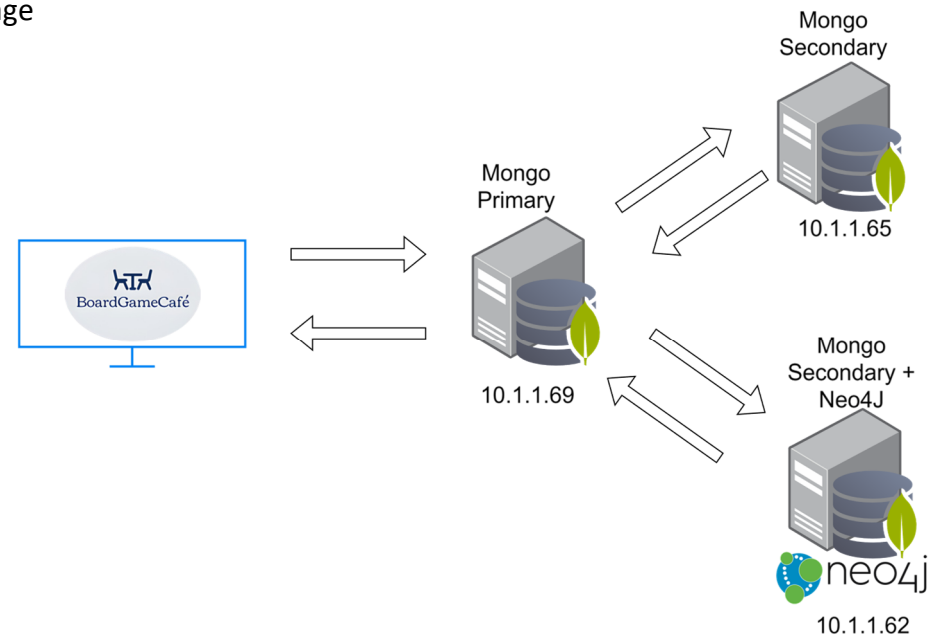

Figure 8: Overview of the distributed DBs architecture

Replica Configuration

The replica set, named *boardgamecafe*, consists of three replicas distributed across three different virtual machines.

Below, a table displaying the key characteristics of each replica is reported. As shown, the lowest *Election Priority* has been assigned to the *Secondary 2* replica, which is the one also hosting the Neo4J database. In this way, thanks to MongoDB's heartbeat mechanism, if the primary node fails, the other replicas become aware of that: they will start a new primary election

round, in which it's highly likely that the *Secondary 1* replica will be elected, while also managing the load on the machine where the Neo4J database is already running.

Virtual Machine	IP	Election Priority	Mongo Port	O.S.
Primary	10.1.1.69	3	27018	Ubuntu
Secondary 1	10.1.1.65	2	27019	Ubuntu
Secondary 2 (With Neo)	10.1.1.62	0.5	27020	Ubuntu

Table 1: VM configuration details

The previous table is translated into the following configuration, which defines the structure and roles of each member in the MongoDB replica set.

```
rsconf = {
  _id: "boardgamecafe",
  members: [
    {
      _id: 0,
      host: "10.1.1.69:27018",
      priority: 3,
      tags: { role: "primary" }
    },
    {
      _id: 1,
      host: "10.1.1.66:27019",
      priority: 2,
      tags: { role: "replica 1 (without Neo4J)" }
    },
    {
      _id: 2,
      host: "10.1.1.62:27020",
      priority: 0.5,
      tags: { role: "replica 2 (with Neo4J)" }
    }
  ],
  settings: {
    setDefaultRWConcern: { w: 2, wtimeout: 5000 }
  }
};
```

Write Concern: Majority

By analyzing the above configuration, in the *settings* field, the following can be observed:

- "*w: 2"*: this setting ensures that the write operation will only be considered successful if it is acknowledged by at least two nodes in the replica set. Specifically, the write operation must be confirmed by the primary node as well as at least one secondary node. This provides an additional level of fault tolerance and data durability, ensuring that even if the primary node fails shortly after performing the write operation, the data will still be available on the secondary node.
- "*wtimeout: 5000*": this setting defines the maximum amount of time (in milliseconds) for which the system will wait for the write acknowledgment to be received. In this case, it is set to 5000 milliseconds (5 seconds). If the required acknowledgment from at least two nodes (primary and one secondary) is not received within the time frame, the write operation will fail with a timeout error. This prevents the system from indefinitely waiting for acknowledgment in situations such as network issues or node unavailability, ensuring that the application can respond in a reasonable time frame.

Read Concern: Nearest

Given the nature of the application (i.e., a boardgame social network), its non-functional requirements emphasize high availability and partition tolerance. For this reason, the *Nearest* read concern was chosen to prioritize low latency. By leveraging this read policy, data is retrieved from the closest replica node in terms of latency, minimizing response times and improving scalability by distributing the read load across the replica set. It is clear that, however, the *Nearest* node might not yet have received the latest update, due to the eventual consistency process. This is tolerable in this case because the application is designed to handle scenarios where slightly stale data does not critically impact the user experience. This trade-off allows to maintain a high degree of availability and performance, ensuring that users can access the system reliably even under heavy traffic or in the event of network partitions.

Sharding Proposal

Sharding is a key strategy for managing large volumes of data and high traffic loads by distributing the workload across multiple servers. This approach helps to overcome the storage and processing limitations of a single node, ensuring efficient data access and storage, as well as balancing the load among nodes, improving performance and reducing query response times. Combined with replication, sharding enhances fault tolerance, ensuring the system remains available and operational even in case of node failures. Sharding enables horizontal scaling, supporting the growth of the application without compromising speed or reliability.

Boardgames collection

Sharding key	boardgameName
Partition algorithm	Hashing

Table 2: sharding proposal for the 'Boardgames' collection

For the *Boardgames* collection, sharding could be implemented on the *boardgameName* field. This would allow an uniform distribution of the documents of this collection across the different replicas.

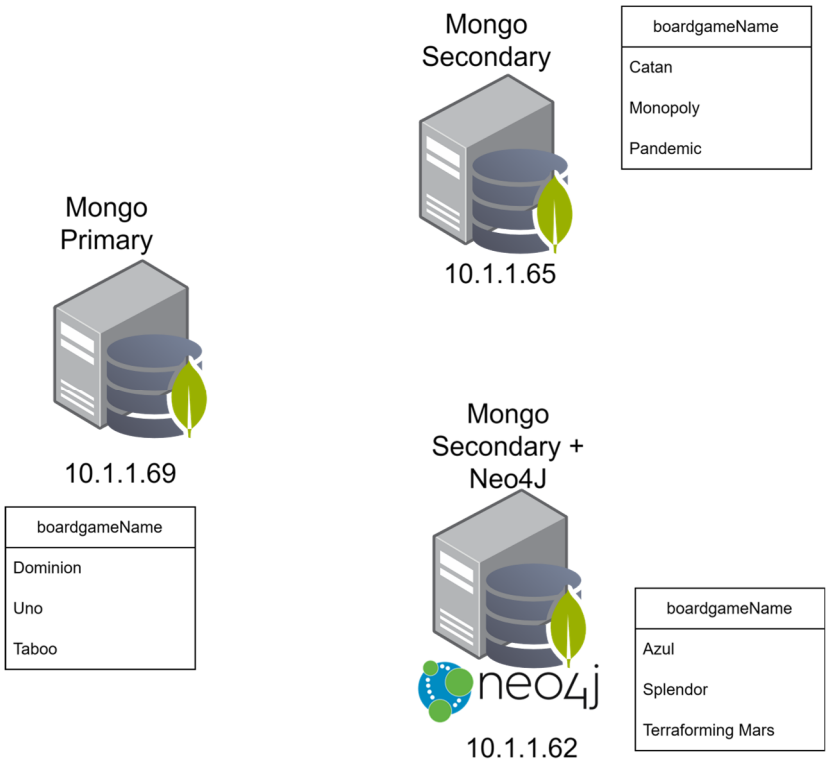

Figure 9: overview for the sharding architecture

Reviews collection

Sharding key	boardgameName
Partition algorithm	Hashing

Table 3: sharding proposal for the 'Reviews' collection

Choosing as sharding key the *boardgameName* field of the *Reviews* collection's documents, along with a partitioning algorithm based on hashing, would allow such documents to be memorized in the same replica that hosts the board game document they are referred to. Since whenever a board game details page is opened, the first content that is loaded is its reviews list, this strategy would allow an easier and quicker data retrieval step.

Users collection

Sharding key	username
Partition algorithm	Hashing

Table 4: sharding proposal for the 'Users' collection

For the *Users* collection, sharding could be implemented on the *username* field. This would allow an uniform distribution of the documents of this collection across the different replicas.

Posts collection

Sharding key	username
Partition algorithm	Hashing

Table 5: sharding proposal for the 'Posts' collection

The sharding technique on the *Posts* collection could follow the same idea that guided the choice behind the *Boardgames* and *Reviews* collections' sharding. Choosing as sharding key the *username* field of the *Posts* collection's documents, along with a partitioning algorithm based on hashing, would allow such documents to be memorized in the same replica that hosts the *Users* collection's document of their author. Since whenever a user's profile is opened, the first content that is loaded is his/her posts list, this strategy would allow an easier and quicker data retrieval step.

Implementation

Package Architecture

The application has been structured through the extensive usage of the *MVC (Model-View-Controller)* design pattern, with an additional *service* layer that is used to abstract the communication with the two databases. Moreover, a *ModelBean* class has been developed to correctly maintain the state of some needed objects throughout the use of the application, and to facilitate the communication of such among different sections of the latter:

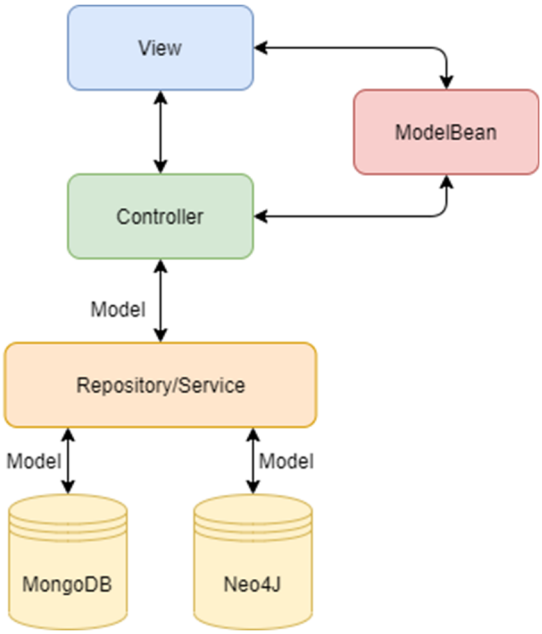

Figure 10: MVC architectural pattern

More in depth, the packages that make up the application code are the following:

- **mvc:** outer container that packages inner classes that make up the MVC structure of the application.
    - **model:** model classes, employed to represent databases entities and relationships. The package also includes the *ModelBean* class, whose usage is crucial for the correct functioning of the application.
        - **mongo:** MongoDB model classes.
        - **neo4j:** Neo4J model classes.
    - **view:** classes used to implement the *View* part of the MVC pattern. These classes are used to make the user able to navigate through the different sections of the application.
    - **controller:** classes in which the behavior and logic of the different parts of the application are specified.
- **repository:** package in which classes that allow the interaction between application and databases are kept.

    - **mongodbms:** classes and interfaces to realize a layer that abstracts away the communication from the application to MongoDB and vice-versa.
    - **neo4jdbms:** classes and interfaces to realize a layer that abstracts away the communication from the application to Neo4J and vice-versa.
- **services:** service classes used to implement an additional layer of communication between application and databases. The most complicated data accessing methods have been implemented in such classes.
- **utils:** collection of utility classes.

MongoDB Model Classes

The model classes that have been developed and used in the application to mirror the structure of the data kept in the employed MongoDB collections are described in this section. The classes present all the needed *getter, setter* and *constructor* methods, which have not been included here for the sake of brevity.

GenericUserModelMongo Class

```
@Document(collection = "users")
public abstract class GenericUserModelMongo {
    @Id
    protected String id;
    protected String username;
    protected String email;
    protected String salt;
    protected String passwordHash;
    protected String _class;
    [. . .]
}
```

UserModelMongo Class

```
@Document(collection = "users")
@TypeAlias("user")
public class UserModelMongo extends GenericUserModelMongo {
    private String name;
    private String surname;
    private String gender;
    private Date dateOfBirth;
    private String nationality;
    private boolean banned;
    [. . .]
}
```

AdminModelMongo Class

```
@Document(collection = "users")
@TypeAlias("admin")
public class AdminModelMongo extends GenericUserModelMongo {
    [. . .]
}
```

BoardgameModelMongo Class

```
@Document(collection = "boardgames")
public class BoardgameModelMongo {
    @Id
    private String id;
    private String boardgameName;
    private String image;
    private String description;
    private int yearPublished;
    private int minPlayers;
    private int maxPlayers;
    private int playingTime;
    private int minAge;
    private Double avgRating;
    private int reviewCount;
    private List<String> boardgameCategory = new ArrayList<>();
    private List<String> boardgameDesigner = new ArrayList<>();
    private List<String> boardgamePublisher = new ArrayList<>();
    [. . .]
}
```

ReviewModelMongo Class

```
@Document(collection = "reviews")
public class ReviewModelMongo {
    @Id
    private String id;
    private String boardgameName;
    private String username;
    private int rating;
    private String body;
    private Date dateOfReview;
    [. . .]
}
```

PostModelMongo Class

```
@Document(collection = "posts")
public class PostModelMongo {
    @Id
    private String id;
    private String username;
    private String title;
    private String text;
    private String tag;
    private Date timestamp;
    private int like_count;
    private List<CommentModel> comments = new ArrayList<>();
    @JsonIgnore
    private String _class;
    [. . .]
}
```

Neo4J Model Classes

The model classes that have been developed and used in the application to mirror the structure of the data kept in the employed Neo4J graph are described in this section. The classes present all the needed *getter, setter* and *constructor* methods, which have not been included here for the sake of brevity.

UserModelNeo4j Class

```
@Node("User")
public class UserModelNeo4j {
    @Id
    private String id;
    private String username;
    @Relationship(type = "FOLLOWS", direction =
Relationship.Direction.OUTGOING)
    private List<UserModelNeo4j> followedUsers;
    @Relationship(type = "FOLLOWS", direction =
Relationship.Direction.INCOMING)
    private List<UserModelNeo4j> followerUsers;
    @Relationship(type = "WRITES_POST", direction =
Relationship.Direction.OUTGOING)
    private List<PostModelNeo4j> writtenPosts;
    @Relationship(type = "LIKES", direction =
Relationship.Direction.OUTGOING)
```

PostModelNeo4j Class

```
@Node("Post")
public class PostModelNeo4j {
    @Id
    private String id;
    @Relationship(type = "REFERS_TO", direction =
Relationship.Direction.OUTGOING)
    private BoardgameModelNeo4j taggedGame;
    @Relationship(type = "WRITES_POST", direction =
Relationship.Direction.INCOMING)
    private UserModelNeo4j author;
    @Relationship(type = "LIKES", direction =
Relationship.Direction.INCOMING)
    private UserModelNeo4j likes;
```

BoardgameModelNeo4j Class

```
@Node("Boardgame")
public class BoardgameModelNeo4j {
    @Id
    public String id;
    public String boardgameName;
    public String image;
    public String description;
    public int yearPublished;
    @Relationship(type = "REFERS_TO", direction =
Relationship.Direction.INCOMING)
    private List<PostModelNeo4j> posts;
    [. . .]
}
```

Mongo DB Management

CRUD Operations

CRUD operations have been developed, on the different MongoDB collections, by leveraging the built-in methods provided by the *Spring* Java framework. For the sake of brevity, only a single example of a *Create*, *Read*, *Update* and *Delete* operation is shown in this paragraph, as the analogous methods implemented for the remaining collections follow the same structure. The shown examples belong to the *Boardgame* collection.

CREATE

```
public BoardgameModelMongo addBoardgame(BoardgameModelMongo boardgame) {
    try {
        return boardgameRepoMongoOp.save(boardgame);
    } catch (Exception e) {...}
}
```

READ

```
public Optional<BoardgameModelMongo> findBoardgameByName(String boardgameName) {
    Optional<BoardgameModelMongo> boardgame = Optional.empty();
    try {
        boardgame = boardgameRepoMongoOp.findByBoardgameName(boardgameName);
    } catch (Exception e) {...}
    return boardgame;
}
```

UPDATE

```
public boolean deleteBoardgame(BoardgameModelMongo boardgame) {
    try {
        boardgameRepoMongoOp.delete(boardgame);
        return true;
    } catch (Exception e) {...}
}
```

DELETE

```
public boolean updateBoardgameMongo(String id, BoardgameModelMongo newBoardgame) {
    try {
        Optional<BoardgameModelMongo> boardgame = boardgameRepoMongoOp.findById(id);
        if (boardgame.isPresent()) {
            BoardgameModelMongo boardgameToBeUpdated = boardgame.get();
            boardgameToBeUpdated.setBoardgameName(newBoardgame.getBoardgameName());
            boardgameToBeUpdated.setImage(newBoardgame.getImage());
            boardgameToBeUpdated.setDescription(newBoardgame.getDescription());
            boardgameToBeUpdated.setYearPublished(newBoardgame.getYearPublished());
            boardgameToBeUpdated.setMinPlayers(newBoardgame.getMinPlayers());
            boardgameToBeUpdated.setMaxPlayers(newBoardgame.getMaxPlayers());
            boardgameToBeUpdated.setPlayingTime(newBoardgame.getPlayingTime());
            boardgameToBeUpdated.setMinAge(newBoardgame.getMinAge());
            boardgameToBeUpdated.setBoardgameCategory(newBoardgame.getBoardgameCategory());
            boardgameToBeUpdated.setBoardgameDesigner(newBoardgame.getBoardgameDesigner());
            boardgameToBeUpdated.setBoardgamePublisher(newBoardgame.getBoardgamePublisher());
            boardgameToBeUpdated.setAvgRating(newBoardgame.getAvgRating());
            boardgameToBeUpdated.setReviewCount(newBoardgame.getReviewCount());

            this.addBoardgame(boardgameToBeUpdated);
        }
        return true;
    } catch (Exception e) {...}
}
```

MongoDB Aggregations

In this section the MongoDB aggregations that have been used in the application will be reported. Both the *Mongo Query Language* implementation and *Java* implementation will be provided.

Find all board games that belong to a given category

```
db.boardgames.aggregate([
    { $match: { boardgameCategory: "CATEGORY_VALUE" } },
    { $skip: SKIP_VALUE },
    { $limit: LIMIT_VALUE }
])
```

```java
public List<BoardgameModelMongo> findBoardgamesByCategory(String category, int limit, int skip) {
    List<BoardgameModelMongo> boardgameOfThisCategory = new ArrayList<>();
    try {
        Aggregation aggregation = Aggregation.newAggregation(
                // Filtering boardgames based on their category
                Aggregation.match(Criteria.where( key: "boardgameCategory").is(category)),
                // Skipping first 'skip' results
                Aggregation.skip(skip),
                // Limiting to 'limit' results
                Aggregation.limit(limit)
        );

        // Aggregation pipeline execution
        boardgameOfThisCategory = mongoOperations.aggregate(
                aggregation,
                collectionName: "boardgames",
                BoardgameModelMongo.class
        ).getMappedResults();
    } catch (Exception e) {...}
    return boardgameOfThisCategory;
}
```

Find the top commented posts about a board game

```
db.posts.aggregate([
    { $match: { tag: "TAG_VALUE" } },
    {
        $project: {
            id: "$_id",
            title: 1,
            username: 1,
            timestamp: 1,
            tag: 1,
            like_count: 1,
            comments: 1,
            numComments: { $size: "$comments" }
        }
    },
    { $sort: { numComments: -1, _id: 1 } },
    { $skip: SKIP_VALUE },
    { $limit: LIMIT_VALUE }
])
```

```java
public List<PostModelMongo> findTopCommentedTaggedPosts(String tag, int limit, int skip) {
    MatchOperation matchOperation = match(Criteria.where( key: "tag").is(tag));
    ProjectionOperation projectionOperation = project()
            .and( name: "_id").as( alias: "id")
            .and( name: "title").as( alias: "title")
            .and( name: "username").as( alias: "username")
            .and( name: "timestamp").as( alias: "timestamp")
            .and( name: "tag").as( alias: "tag")
            .and( name: "like_count").as( alias: "like_count")
            .and( name: "comments").as( alias: "comments")
            .and(ArrayOperators.Size.lengthOfArray( fieldReference: "comments")).as( alias: "numComments");

    SortOperation sortOperation = sort(Sort.by(Sort.Direction.DESC,  ...properties: "numComments")
            .and(Sort.by(Sort.Direction.ASC,  ...properties: "_id")));  // Ordering by numComments and _id

    SkipOperation skipOperation = skip(skip);
    LimitOperation limitOperation = limit(limit);

    Aggregation aggregation = Aggregation.newAggregation(
            matchOperation,
            projectionOperation,
            sortOperation,
            skipOperation,
            limitOperation
    );
    AggregationResults<PostModelMongo> results = mongoOperations.aggregate(
            aggregation,  collectionName: "posts", PostModelMongo.class);

    if (results == null || results.getMappedResults() == null) {...}
    return results.getMappedResults();
}
```

Find the most posted and commented tags

```
db.posts.aggregate([
    { $match: { "tag": { $exists: true } } },
    {
        $group: {
            _id: "$tag",
            postCount: { $sum: 1 },
            commentCount: { $sum: { $size: "$comments" } }
        }
    },
    {
        $project: {
            tag: "$_id",
            postCount: 1,
            commentCount: 1,
            _id: 0
        }
    },
    { $sort: { postCount: -1, commentCount: -1 } },
    { $limit: LIMIT_RESULTS }
])
```

```java
public Document findMostPostedAndCommentedTags(int limitResults) {
    MatchOperation matchOperation = match(new Criteria( key: "tag").exists( value: true));

    GroupOperation groupOperation = group( ...fields: "tag")
            .count().as( alias: "postCount")
            .sum(new AggregationExpression() {
                @Override
                public @NotNull Document toDocument(@NotNull AggregationOperationContext context) {
                    return new Document("$size", "$comments");
                }
            }).as( alias: "commentCount");

    ProjectionOperation projectionOperation = project()
            .and( name: "_id").as( alias: "tag")
            .and( name: "postCount").as( alias: "postCount")
            .and( name: "commentCount").as( alias: "commentCount")
            .andExclude( ...fieldNames: "_id");

    SortOperation sortOperation = sort(Sort.by(Sort.Direction.DESC,  ...properties: "postCount", "commentCount"));

    LimitOperation limitOperation = limit(limitResults);

    Aggregation aggregation = newAggregation(matchOperation,
            groupOperation,
            projectionOperation,
            sortOperation,
            limitOperation);

    AggregationResults<Document> results = mongoOperations.aggregate(aggregation,  collectionName: "posts", Document.class)
    return results.getRawResults();
}
```

Find the top rated boardgame given a year

```
db.reviews.aggregate([
    { $project: {
            name: "$boardgameName",
            rating: 1,
            year: { $year: "$dateOfReview" }
     } },
    { $group: {         // Group by board game name and year
            _id: { name: "$name", year: "$year" },
            avgRating: { $avg: "$rating" },     // Get avg rating
            numReviews: { $sum: 1 }                     // Count reviews
     } },
    { $match: { // Get games of the given year with the min number of reviews
            "numReviews": { $gte: minReviews },
            "_id.year": year
    } },
    { $group: { // Group by year and push the boardgames data
            _id: "$_id.year",
            topGames: {
                $push: {
                    name: "$_id.name",
                    avgRating: "$avgRating",
                    numReviews: "$numReviews"
                }
            }
    } },
    {   // Sort the top games in each year by avg rating, descending
        $addFields: {
            topGames: {
                $sortArray: {
                    input: "$topGames",
                    sortBy: { avgRating: -1 }
                }
            }
        }
    },
    { $project: {       // Limit the number of top games per year
            topGames: { $slice: ["$topGames", limit] }
    } },
    { $sort: { "_id": 1 } }             // Sort by year in ascending order
]);
```

```
public Document getTopRatedBoardgamePerYear(int minReviews, int limit, int year) {
    ProjectionOperation projectYear = project().andExpression( expression: "boardgameName").as( alias: "name")
                                               .andExpression( expression: "rating").as( alias: "rating")
                                               .andExpression( expression: "year(dateOfReview)").as( alias: "year");
    Criteria minReviewsAndYear = new Criteria().andOperator(
            Criteria.where( key: "numReviews").gte(minReviews),
            Criteria.where( key: "_id.year").is(year) // Accessing year via _id, since the year became a key because of the grouping
    );
    GroupOperation groupByYearAndGame = group( ...fields: "name" ,"year")
            .avg( reference: "rating").as( alias: "avgRating")
            .count().as( alias: "numReviews");

    MatchOperation matchMinReviews = match(minReviewsAndYear);

    GroupOperation groupByYear = group( ...fields: "_id.year") GroupOperation
            .push(new BasicDBObject("name", "$_id.name")
                    .append( key: "avgRating",  val: "$avgRating")
                    .append( key: "numReviews",  val: "$numReviews")) GroupOperationBuilder
            .as( alias: "topGames");
    AddFieldsOperation addFieldsSortTopGames = addFields() AddFieldsOperationBuilder
            .addField("topGames") ValueAppender
            .withValue(new BasicDBObject("$let", new BasicDBObject("vars", new BasicDBObject("topGames", "$topGames"))
                    .append( key: "in", new BasicDBObject("$sortArray", new BasicDBObject("input", "$$topGames")
                            .append( key: "sortBy", new BasicDBObject("avgRating", -1)))))).build();

    ProjectionOperation limitTopGamesPerYear = project().and( name: "topGames").slice(limit).as( alias: "topGames");
    SortOperation sortByYear = sort(Sort.by(Sort.Order.asc( property: "_id"), Sort.Order.desc( property: "topGames.avgRating")));

    Aggregation aggregation = newAggregation(projectYear, groupByYearAndGame, matchMinReviews, groupByYear,
                                             addFieldsSortTopGames, limitTopGamesPerYear, sortByYear);
    AggregationResults<Document> results = mongoOperations.aggregate(aggregation,  collectionName: "reviews", Document.class);
    return results.getRawResults();
```

Find the average age of users by their nationality

```
db.users.aggregate([
    { $match: { _class: "user" } },
    {  // Compute the age by subtracting dateOfBirth from the current date
        $project: {
            nationality: 1,
            age: {
                $dateDiff: { startDate: "$dateOfBirth",
                             endDate: "$$NOW",
                             unit: "year" }
            }
        }
    },
    {  // Group by nationality and calculate the average age
        $group: { _id: "$nationality",
                  averageAge: { $avg: "$age" }
                }
    },
    {      // Round the average age to 1 decimal place and project fields
        $project: {
            nationality: "$_id",
            averageAge: { $round: ["$averageAge", 1] },
            _id: 0
        }
    },
    ...(limit > 0 ? [{ $limit: limit }] : [])     // Apply limit if > 0
]);
```

```java
public Optional<Document> showUserAvgAgeByNationality(int limit) {
    MatchOperation matchOperation = match(new Criteria( key: "_class").is( value: "user"));

    ProjectionOperation computeAge = Aggregation.project()      // Projecting age, computed as the difference (birthday - today)
            .andExpression( expression: "{$dateDiff: {startDate: '$dateOfBirth', endDate: '$$NOW', unit: 'year'}}") ExpressionProjectionOpe
            .as( alias: "age") ProjectionOperation
            .andExpression( expression: "nationality").as( alias: "nationality");

    GroupOperation groupByCountry = Aggregation.group( ...fields: "nationality") // Grouping by nationality and maintaining the age
            .avg( reference: "age").as( alias: "averageAge");

    ProjectionOperation projectFields = Aggregation.project( ...fields: "averageAge")
            .andExpression( expression: "_id").as( alias: "nationality")
            .andExpression( expression: "averageAge").as( alias: "averageAge")
            .and(ArithmeticOperators.Round.roundValueOf( fieldReference: "averageAge").place(1)).as( alias: "averageAge");

    List<AggregationOperation> operations = new ArrayList<>();
    operations.add(matchOperation);
    operations.add(computeAge);
    operations.add(groupByCountry);
    if (limit > 0)         // Limit can be -1 (i.e. there is no limit)
        operations.add(Aggregation.limit(limit));
    operations.add(projectFields);

    Aggregation aggregation = Aggregation.newAggregation(operations);

    AggregationResults<UserModelMongo> results = mongoOperations.aggregate(aggregation,  collectionName: "users", UserModelMongo.class);

    return Optional.ofNullable(results != null ? results.getRawResults() : null);
}
```

Find the countries with the highest number of users

```
db.users.aggregate([
    { $match: { _class: "user" } },
    {  // Group by nationality and count the number of users
        $group: {
            _id: "$nationality",
            numUsers: { $sum: 1 }
        }
    },
    { $sort: { numUsers: -1 } },
    {  // Project the fields: rename _id to nationality and include numUsers
        $project: {
            nationality: "$_id",
            numUsers: 1,
            _id: 0
        }
    },
    ...(limit > 0 ? [{ $limit: limit }] : [])     // Limit if greater than 0
]);
```

```java
public Document findCountriesWithMostUsers(int minUserNumber, int limit) {
    // Filtering documents to make sure we're treating users
    MatchOperation matchOperation = match(new Criteria( key: "_class").is( value: "user"));

    // Grouping users country by country and counting how many users we have in each one of them
    GroupOperation groupOperation = group( ...fields: "nationality")
            .count().as( alias: "numUsers");  // Counting users

    // Ordering countries based on the number of users (DESC)
    SortOperation sortOperation = sort(Sort.by(Sort.Direction.DESC,  ...properties: "numUsers"));

    ProjectionOperation projectionOperation = project()
            .andExpression( expression: "_id").as( alias: "nationality")  // Projecting id as nationality
            .andExpression( expression: "numUsers").as( alias: "numUsers");  // Projecting the number of users

    List<AggregationOperation> operations = new ArrayList<>();
    operations.add(matchOperation);
    operations.add(groupOperation);
    operations.add(sortOperation);
    if (limit > 0)
        operations.add(Aggregation.limit(limit));
    operations.add(projectionOperation);

    Aggregation aggregation = Aggregation.newAggregation(operations);

    AggregationResults<UserDBMongo> result = mongoOperations
            .aggregate(aggregation,  collectionName: "users", UserDBMongo.class);

    return result.getRawResults();
}
```

Find the most active users based on their reviews

```
db.reviews.aggregate([
    {  // Match reviews within the specified date range
        $match: {
            dateOfReview: {
                $gte: startDate,
                $lte: endDate
            }
        }
    },
    {  // Group by username, count reviews, and collect review dates
        $group: {
            _id: "$username",
            reviewCount: { $sum: 1 },
            reviewDates: { $push: "$dateOfReview" }
        }
    },
    { $match: {       // Filter users with at least 2 reviews
         reviewCount: { $gte: 2 }
    } },
    {          // Sort review dates in ASC order
        $project: {
            reviewCount: 1,
            orderedReviewDates: {
                $sortArray: { input: "$reviewDates", sortBy: 1 }
            }
        }
    },
    {          // Calculate differences between adjacent review dates
        $project: {
            reviewCount: 1,
            orderedReviewDates: 1,
            dateDifferences: {
                $map: {
                    input: { $range: [0, { $subtract: [{ $size: "$orderedReviewDates" }, 1] }] },
                    as: "index",
                    in: {
                        $dateDiff: {
                            startDate: { $arrayElemAt: ["$orderedReviewDates", "$$index"] },
                            endDate: { $arrayElemAt: ["$orderedReviewDates", { $add: ["$$index", 1] }] },
                            unit: "day"
                        }
                    }
                }
```

```
            }
        }
    },
    {  // Calculate the average time difference between reviews
        $project: {
            reviewCount: 1,
            averageDateDifference: { $avg: "$dateDifferences" }
        }
    },
    {  // Sort by review count (DESC) and average date difference (ASC)
        $sort: {
            reviewCount: -1,
            averageDateDifference: 1
        }
    },
    ...(limitResults > 0 ? [{ $limit: limitResults }] : [])
]);
```

```
public Document findActiveUsersByReviews(Date startDate, Date endDate, int limitResults) {
    try {
        // Filtering reviews to get those in the specified time interval
        MatchOperation matchOperation = Aggregation.match(Criteria.where( key: "dateOfReview")
                .gte(startDate)
                .lte(endDate));

        // Grouping by user, counting the reviews and gathering the dates of the reviews
        GroupOperation groupOperation = Aggregation.group( ...fields: "username")
                .count().as( alias: "reviewCount")
                .push( reference: "dateOfReview").as( alias: "reviewDates");

        // Filtering users that published at least 2 reviews
        MatchOperation matchReviewCount = Aggregation.match(Criteria.where( key: "reviewCount").gte( value: 2));

        // Ordering the review dates
        Document sortReviewDates = new Document("$project",
                new Document("reviewCount", 1)
                        .append("orderedReviewDates", new Document("$sortArray",
                                new Document("input", "$reviewDates").append("sortBy", 1))));

// Computing the difference between adjacent dates (using $map)
Document calculateDateDifferences = new Document("$project",
        new Document("reviewCount", 1)
                .append("orderedReviewDates", 1)
                .append("dateDifferences",
                        new Document("$map",
                                new Document("input",
                                        new Document("$range",
                                                Arrays.asList(0,
                                                        new Document("$subtract",
                                                                Arrays.asList(
                                                                        new Document("$size", "$orderedReviewDates"), 1))))
                        ).append("as", "index")
                                .append("in",
                                        new Document("$dateDiff",
                                                new Document("startDate",
                                                        new Document("$arrayElemAt",
                                                                Arrays.asList("$orderedReviewDates", "$$index")))
                                                .append("endDate",
                                                        new Document("$arrayElemAt",
                                                                Arrays.asList(
                                                                        "$orderedReviewDates",
                                                                        new Document("$add",
                                                                                Arrays.asList("$$index", 1)))))
                                                .append("unit", "day"))))
                ));
        // Compute and project the average of the differences
        Document calculateAverageFrequency = new Document("$project",
                new Document("reviewCount", 1)
                        .append("averageDateDifference", new Document("$avg", "$dateDifferences")));

        SortOperation sortOperation = Aggregation
                .sort(Sort.by(Sort.Direction.DESC, ...properties: "reviewCount")
                        .and(Sort.by(Sort.Direction.ASC, ...properties: "averageDateDifference")));

        LimitOperation limitOperation = Aggregation.limit(limitResults);

        Aggregation aggregation = Aggregation.newAggregation(matchOperation, groupOperation, matchReviewCount,
                new CustomAggregationOperation(sortReviewDates), new CustomAggregationOperation(calculateDateDifferences),
                new CustomAggregationOperation(calculateAverageFrequency), sortOperation, limitOperation);

        AggregationResults<ReviewModelMongo> results = mongoOperations.aggregate(aggregation, collectionName: "reviews", ReviewModelMongo.class);
        return results.getRawResults();
    } catch (Exception ex) {...}
}
```

Find, among the most followed users, the ones that have a given minimum average like count

```
db.posts.aggregate([
    {  // Filter posts by usernames
        $match: {
            username: { $in: mostFollowedUsersUsernames }
        }
    },
    {  // Filter posts within the last 2000 days (pastDate is a correctly
       // formatted date object representing the last 2000 days)
        $match: {
            timestamp: { $gte: pastDate }
        }
    },
    {  // Group posts by username, count posts, and compute avg likes
        $group: {
            _id: "$username",
            postCount: { $sum: 1 },
            avgLikes: { $avg: "$like_count" }
        }
    },
    {  // Filter users who have at least 2 posts
        $match: {
            postCount: { $gte: 2 }
        }
    },
    {  // Filter users by minimum average like count
        $match: {
            avgLikes: { $gte: minAvgLikeCount }
        }
    },
    {  // Sort users by average likes DESC
        $sort: {
            avgLikes: -1
        }
    },
    ...(limit > 0 ? [{ $limit: limit }] : []),    // Limit results
    {  // Project only the username
        $project: {
            username: "$_id",
            _id: 0
        }
    }
]);
```

```java
public List<String> findMostFollowedUsersWithMinAverageLikesCountUsernames(
        List<String> mostFollowedUsersUsernames,
        long minAvgLikeCount, int limit)
{
    LocalDate pastDate = LocalDate.now().minusDays( daysToSubtract: 2000);  // Consider only posts which have been posted in
    Date pastDateFullDate = Date.from(pastDate.atStartOfDay(ZoneId.systemDefault()).toInstant());

    MatchOperation matchOperationUsernames = match(Criteria.where( key: "username").in(mostFollowedUsersUsernames));
    MatchOperation matchOperationDate = match(Criteria.where( key: "timestamp").gte(pastDateFullDate));

    GroupOperation groupByUsername = group( ...fields: "username")
            .count().as( alias: "postCount")
            .avg( reference: "like_count").as( alias: "avgLikes");

    MatchOperation matchPostCount = match(Criteria.where( key: "postCount").gte( value: 2));
    MatchOperation matchByMinAvgLikes = match(Criteria.where( key: "avgLikes").gte(minAvgLikeCount));

    SortOperation sortByAvgLikesDesc = sort(Sort.by(Sort.Direction.DESC,  ...properties: "avgLikes"));
    LimitOperation limitOperation = limit(limit);
    ProjectionOperation projectionOperation = project()
            .and( name: "_id").as( alias: "username")
            .andExclude( ...fieldNames: "_id");

    Aggregation aggregation = newAggregation(matchOperationUsernames, matchOperationDate, groupByUsername,
                                             matchPostCount, matchByMinAvgLikes, sortByAvgLikesDesc,
                                             limitOperation, projectionOperation);
    AggregationResults<Document> results = mongoOperations.aggregate(aggregation,  collectionName: "posts", Document.class);
    return results.getMappedResults().stream() Stream<Document>
            .map(doc -> doc.getString( key: "username")) Stream<String>
            .collect(Collectors.toList());
}
```

Neo4J DB Management

This chapter will describe how the Neo4J database was managed within the Java application. To simplify database interaction and implement standard operations, the *Spring Data for Neo4J* framework was used. This approach effectively integrated the DBMS with the application, providing an abstraction layer that facilitated CRUD operations and improved code maintainability.

CRUD Operations

This section will present the basic operations of *create*, *read*, *update*, and *delete*. The main implemented methods will be highlighted, leveraging the tools provided by *Spring* for Neo4J. For the sake of brevity, only the implementation covering of the CRUD operations of one of the 3 entities managed on Neo4J are shown, as the other entities all follow the same pattern. The reported methods are those implemented for the *User* nodes.

CREATE

```
public boolean addUser(UserModelNeo4j user) {
    try {
        userNeo4jDB.save(user);
        return true;
    } catch (Exception e) {
        System.err.println("[ERROR] addUser()@UserDBNeo4j.java raised an exception: " + e.getMessage());
        return false;
    }
}
```

READ

```
public Optional<UserModelNeo4j> findByUsername(String username) {
    Optional<UserModelNeo4j> user = Optional.empty();
    try {
        user = userNeo4jDB.findByUsername(username);
    }
    catch (Exception e) {
        System.err.println("[ERROR] findByUsername()@UserDBNeo4j.java raised an exception: " + e.getMessage());
    }
    return user;
}
```

UPDATE (Relationships included)

```
public boolean updateUser(String id, UserModelNeo4j updated) {
    try {
        Optional<UserModelNeo4j> old = userNeo4jDB.findById(id);
        if (old.isPresent()) {
            UserModelNeo4j oldUser = old.get();
            oldUser.setUsername(updated.getUsername());
            oldUser.setFollowedUsers(updated.getFollowedUsers());
            oldUser.setWrittenPosts(updated.getWrittenPosts());
            oldUser.setLikedPosts(updated.getLikedPosts());
            userNeo4jDB.save(oldUser);
            return true;
        }
        return false;
    } catch (Exception e) {
        System.err.println("[ERROR] updateUser()@UserDBNeo4j.java raised an exception: " + e.getMessage());
        return false;
```

DELETE (Relationships included)

```
public boolean deleteUserDetach(String username) {
    try {
        userNeo4jDB.deleteAndDetachUserByUsername(username);
        return true;
    } catch (Exception e) {
        System.err.println("[ERROR] deleteUserDetach()@UserDBNeo4j.java raised an exception: " + e.getMessage());
        return false;
    }
}
```

Cypher Queries Implementation

This section will describe the complex *Cypher* queries exploited in the project. It will explain how these queries were implemented through custom methods, fully utilizing the flexibility of the *Cypher* language and its integration with *Spring*.

Suggest users that enjoy (i.e., liked) the same posts as you do:

```
@Query("""
        MATCH(myPost:Post)<-[:LIKES]-(me:User{username: $username})
        WITH myPost, COLLECT(DISTINCT myPost.id) as myPostID, me
        MATCH (notFriend:User)-[:LIKES]->(p:Post)
        WHERE (p.id IN myPostID) AND (notFriend.username <> $username)
        AND NOT (me)-[:FOLLOWS]->(notFriend)
        WITH notFriend, COUNT(p) as sameLikedPosts
        RETURN notFriend.username
        ORDER BY sameLikedPosts DESC
        SKIP $skipCounter LIMIT $limit;
        """)
List<String> findUsersBySameLikedPosts(@Param("username")String username,
                                       @Param("limit") int limit,
                                       @Param("skipCounter") int skipCounter)
```

```
public List<String> getUsersBySameLikedPosts(String username, int limit, int skipCounter) {
    List<String> suggestedUsers = new ArrayList<>();
    try {
        return userNeo4jDB.findUsersBySameLikedPosts(username, limit, skipCounter);
    } catch (Exception e) {
        System.err.println("[ERROR] getUsersBySameLikedPosts()@UserDBNeo4j.java raised an exception: "
                + e.getMessage());
    }
    return suggestedUsers;
}
```

Suggest users which posted about boardgames you posted about too:

```
@Query("""
        MATCH (me:User{username: $username})-[:WRITES_POST]->(myPosts:Post)-[:REFERS_TO]->(myBGames:Boardgame
        WITH me, COLLECT(DISTINCT myBGames.boardgameName) as myBGamesNames
        MATCH (notFriend:User)-[:WRITES_POST]->(post:Post)-[:REFERS_TO]->(notFriendBgames:Boardgame)
        WHERE (notFriendBgames.boardgameName IN myBGamesNames)
        AND (NOT (me)-[:FOLLOWS]->(notFriend)) AND me <> notFriend
        RETURN notFriend.username SKIP $skipCounter LIMIT $limit
        """)
List<String> usersByCommonBoardgamePosted(@Param("username") String username,
                                          @Param("limit") int limit,
                                          @Param("skipCounter") int skipCounter);
```

```
public List<String> getUsersByCommonBoardgamePosted(String username, int limit, int skipCounter) {
    List<String> suggestedUsers = new ArrayList<>();
    try {
        return userNeo4jDB.usersByCommonBoardgamePosted(username, limit, skipCounter);
    } catch (Exception e) {
        System.err.println("[ERROR] getUsersByCommonBoardgamePosted()@UserDBNeo4j.java raised an exception: "
                + e.getMessage());
    }
    return suggestedUsers;
}
```

Suggest influencer users (only one different criteria used)

```
@Query("""
        MATCH (u)<-[followRel:FOLLOWS]-(follower:User)
        WITH u, COUNT(DISTINCT followRel) AS followersCount
        WHERE followersCount >= $minFollowersCount
        RETURN u.username AS username
        ORDER BY followersCount DESC
        LIMIT $limit
        """)
List<String> findMostFollowedUsersUsernames(@Param("minFollowersCount") long minFollowersCount,
                                            @Param("limit") int limit);
```

```
public List<String> getMostFollowedUsersUsernames(long minFollowersCount, int limit) {
    List<String> mostFollowedUsersUsernames = new ArrayList<>();
    try {
        return userNeo4jDB.findMostFollowedUsersUsernames(minFollowersCount, limit);
    } catch (Exception e) {
        System.err.println("[ERROR] getMostFollowedUsersUsernames()@UserDBNeo4j.java raised an exception: "
                + e.getMessage());
    }
    return mostFollowedUsersUsernames;
}
```

Suggest posts liked by followed users:

```
@Query("""
        MATCH (u:User{username: $username})-[:FOLLOWS]->(following:User)-
        [:LIKES]->(p:Post), (p)<-[l:LIKES]-(:User)
        WHERE NOT EXISTS{MATCH (u)-[:LIKES]->(p)}
        WITH p, COUNT(l) AS likes
        RETURN p
        ORDER BY likes desc
        SKIP $skipCounter
        LIMIT $limit
        """)
List<PostModelNeo4j> findPostsLikedByFollowedUsers(@Param("username") String username,
                                                   @Param("limit") int limitResults,
                                                   @Param("skipCounter") int skipCounter);
```

```
public List<PostModelNeo4j> getPostsLikedByFollowedUsers(String username, int limitResults, int skipCounter) {
    List<PostModelNeo4j> posts = new ArrayList<>();
    try {
        posts = postRepoNeo4j.findPostsLikedByFollowedUsers(username, limitResults, skipCounter);
    } catch (Exception ex) {
        System.err.println("[ERROR] getPostsLikedByFollowedUsers()@PostDBNeo4j.java raised an exception: "
                + ex.getMessage());
    }
    return posts;
}
```

Suggest posts by followed users:

```
@Query("""
        MATCH (currentUser:User {username: $username})-[:FOLLOWS]->(followedUser:User)-[:WRITES_POST]->(post:Post)
        OPTIONAL MATCH (post)<-[:LIKES]-(likedBy:User)
        WITH post, COUNT(likedBy) AS likeCount
        RETURN post
        ORDER BY likeCount DESC
        SKIP $skipCounter
        LIMIT $limit
        """)
List<PostModelNeo4j> findPostsCreatedByFollowedUsers(@Param("username") String username,
                                                     @Param("limit") int limitResults,
                                                     @Param("skipCounter") int skipCounter);
```

```
public List<PostModelNeo4j> getPostsByFollowedUsers(String username, int limitResults, int skipCounter) {
    List<PostModelNeo4j> posts = new ArrayList<>();
    try {
        posts = postRepoNeo4j.findPostsCreatedByFollowedUsers(username, limitResults, skipCounter);
    } catch (Exception ex) {
        System.err.println("[ERROR] getPostsByFollowedUsers()@PostDBNeo4j.java raised an exception: "
                + ex.getMessage());
    }
    return posts;
}
```

Suggest boardgames posted by followed users:

```
@Query("MATCH (you:User {username: $username})-[:FOLLOWS]->(otherUser:User)-[:WRITES_POST]->(post:Post)\n" +
        "-[:REFERS_TO]->(game:Boardgame) \n" +
        "RETURN DISTINCT game \n" +
        "SKIP $skipCounter \n"+
        "LIMIT $limit")
List<BoardgameModelNeo4j> getBoardgamesWithPostsByFollowedUsers(@Param("username") String username,
                                                                @Param("limit") int limit,
                                                                @Param("skipCounter")int skipCounter);
```

```
public List<BoardgameModelNeo4j> getBoardgamesWithPostsByFollowedUsers(String username,
                                                                       int limit,
                                                                       int skipCounter) {
    List<BoardgameModelNeo4j> boardgames = new ArrayList<>();
    try {
        boardgames = boardgameRepoNeo4j.getBoardgamesWithPostsByFollowedUsers(username, limit, skipCounter);
    } catch (Exception e) {
        System.err.println("[ERROR] getBoardgamesWithPostsByFollowedUsers()@BoardgameDBNeo4j.java " +
                "raised an exception: " + e.getMessage());
    }
    return boardgames;
}
```

Database Consistency Management

In this section, the necessary operations to maintain consistency between the two chosen Databases (MongoDB and Neo4J) are outlined.

Add User

When a new user subscribes to the application, it is essential to update both databases to ensure consistency:

1. Insert the new *user* into the MongoDB collection.
2. Create a corresponding *user* node in the Neo4J database, setting all the appropriate fields.

Update User

When a user updates his/her information through his/her profile page, updating both databases is not necessary, since the only fields stored in Neo4J (user *id* and *username*), are immutable from the user's profile page.

Delete User

For a user's deletion, the workflow is the following:

1. Delete the *user* node from Neo4J
2. Delete all his/her *posts* from Neo4J
3. Delete all his/her *relationships* on Neo4J *(FOLLOWS, WRITES_POST, LIKES)*
4. Delete the *user* document from MongoDB
5. Delete all his/her *posts* from MongoDB
6. Delete all his/her *comments* from posts MongoDB
7. Delete all his/her *reviews* from MongoDB

Add Boardgame

When a new boardgame is inserted in the application (action that can be carried out only by an administrator), both databases need to be updated:

1. Insert the new *boardgame* in the MongoDB collection, with all its fields.
2. Create a corresponding *boardgame* node in the Neo4J database, with the relevant information to be shown in the main boardgames page.

Update Boardgame

When the administrator updates the boardgame details, the following steps are taken to ensure consistency across both databases:

1. Update the twin *boardgame* node on Neo4J
2. Update the *boardgame* details on Mongo

Delete Boardgame

When the administrator deletes a boardgame, both the databases need to be updated:

1. Delete the *boardgame* node from Neo4J
2. Delete all the *posts* created by tagging that boardgame, and the relationship with the boardgame node from Neo4J.
3. Delete that *boardgame* from Mongo
4. Delete all the *reviews* related to that boardgame from Mongo
5. Delete all the *posts* related to that boardgame from Mongo

Add Post

When a user adds a post to the application, the following steps are carried out to maintain consistency between the databases:

1. Insert the new *post* in MongoDB
2. Insert the new *post* also in Neo4J
3. Creation the *WRITES_POST* relationship between the author and that post

Update Post

When a user edits one of his/her posts, only the MongoDB database needs to be updated, as there are no editable fields in the corresponding Neo4J node.

Delete Post

When a post is deleted, the following steps are taken to ensure consistency between both databases:

1. Delete the *post* on Neo4J
2. Delete the *WRITES_POST* relationship between the author and the post
3. Delete all the *LIKES* relationship between the post and the user who liked that post
4. Delete the *post* on MongoDB (also all its comment will be deleted)

Add Review

When a user inserts a new **review**, it's necessary create that review only on Mongo, because we don't have the relative node in Neo4J graph.

Update Review

When a user updates their **review**, it is not necessary to propagate this change to Neo4J.

Delete Review

When a user delete their **review**, it is not necessary to propagate this change to Neo4J, but obviously it's needed to update the redundances information in the *boardgame* document who received that review.

Indexes Analysis

The following section will provide an in-depth analysis of the indexes that have been chosen for the two databases. Each metric has been computed as the average of the results given by 30 repeated experiments, that have been carried out to obtain statistically valid samples.

MongoDB

Users Collection

Basically any search operation on this collection is carried out by using the *username* field. For this reason, the index that has been chosen for the *Users* MongoDB collection is the following:

Attribute	Index type	Properties
username	Single	1

Table 6: Index chosen for the 'Users' collection

The following query has been run to obtain statistical data:

```
db.users.find({ username:"gino" }).explain("executionStats")["executionStats"];
```

The performance gains that have been reached with the adoption of the index are shown below:

	username
Without index	
avgExecutionTimeMillis	51.2
totalKeysExamined	0
totalDocsExamined	111 590
With index	
avgExecutionTimeMillis	0.6
totalKeysExamined	1
totalDocsExamined	1

Table 7: Results obtained by creating the index

Posts Collection

Posts can be searched, within the application, by filtering on the boardgame they're referred to: this makes the *tag* (boardgameName) field a good first candidate for indexing. When opening the *Posts Feed* page, all the posts created by the current

user's followed users are retrieved from the database (using the *username* field). Moreover, such posts are ordered by their date of creation, which involves sorting by their *timestamp* field. To optimize both filtering and sorting, a compound index on the *username* and *timestamp* fields is created, ensuring faster query performance and efficient retrieval of the most recent posts related to a specific boardgame.

For these reasons, the indexes that have been chosen for the *Posts* MongoDB collection are the following:

Attribute	Index type	Properties
tag	Single	1
username, timestamp	Compound	{1, -1}

Table 8: Indexes chosen for the 'Posts' collection

The following queries have been run, respectively, to obtain statistical data about the three indexes:

```
db.posts.find({ "tag":"Catan" }).explain("executionStats")["executionStats"];
db.posts.find(
    {username:"lazyladybug603"})
    .sort({timestamp:-1}
).explain("executionStats")["executionStats"];
```

The performance gains that have been reached with the adoption of the index are shown below:

	tag	username, timestamp
Without index		
avgExecutionTimeMillis	10.6	45
totalKeysExamined	0	0
totalDocsExamined	14 005	14 005
With index		
avgExecutionTimeMillis	0.8	0.2
totalKeysExamined	420	11
totalDocsExamined	420	11

Table 9: Results obtained by creating the indexes

Reviews Collection

Reviews are, within the application, always shown in descending order based on their creation timestamp. It is reasonable, considering the main concept of the application, to assume that the average user will spend most of the time looking at

boardgames details: when opening such pages, all the reviews of the game will be loaded (10 at a time) from the database. Declaring a compound index on the *boardgameName* and *dateOfReview* fields to improve the efficiency of related queries is a choice that optimized response times.

Moreover, an index on *username* field was declared to improve the efficiency of browsing users' reviews from their profiles.

Attribute	Index type	Properties
username	Single	1
boardgameName, dateOfReview	Compound	{1 -1}

Table 10: Indexes chosen for the 'Reviews' collection

The following queries have been run, respectively, to obtain statistical data about the two indexes:

```
db.reviews.find(
    { boardgameName: "Catan" })
    .sort({ dateOfReview: -1 })
.explain("executionStats")["executionStats"];
db.reviews.find(
    { username: "greengoose169" } )
    .sort({ dateOfReview: -1 })
.explain("executionStats")["executionStats"];
```

The performance gains that have been reached with the adoption of the index are shown below:

	username	boardgameName, dateOfReview
Without index		
avgExecutionTimeMillis	461	706
totalKeysExamined	0	0
totalDocsExamined	899 765	899 765
With index		
avgExecutionTimeMillis	6.2	14.8
totalKeysExamined	60	5 406
totalDocsExamined	60	5 406

Table 11: results obtained by creating the indexes

Boardgames Collection

Being boardgames the main concept of the application, minimizing the time that is needed to open up a boardgames page is a key goal. The *boardgameName* field was chosen as the target for creating an index to optimize the performance of queries that filter documents by boardgame name.

Attribute	Index type	Properties
boardgameName	Single	1

Table 12: Indexes chosen for the 'Boardgames' collection

The following queries have been run, respectively, to obtain statistical data about the two indexes:

```
db.boardgames.find(
        {boardgameName:"Catan"}).sort.explain("exectuionStats")["executionStats"];
```

The performance gains that have been reached with the adoption of the index are shown below:

	boardgameName
With index	
executionTimeMillis	10.2
totalKeysExamined	0
totalDocsExamined	20 889
Without index	
executionTimeMillis	0.2
totalKeysExamined	1
totalDocsExamined	1

Table 13: Results obtained by creating the indexes

Neo4J

User Nodes

An index on the *username* field of *User* type nodes has been created to speedup the queries that suggest new potential friends based, for example, on the same boardgame posted, or on the same liked posts. Moreover, new friends can be suggested by checking out the boardgame community's influencers. Without an index, Neo4J would perform a full scan of the entire graph to locate *User* nodes with a given username: an operation that can be very slow on large datasets. With an index, Neo4J is able to access the relevant nodes directly, significantly reducing search time and improving overall queries performance.

The following cypher query has been run to obtain statistical data:

```
PROFILE MATCH (u:User{username:'goldencat892'}) return u
```

The performance gains that have been reached with the adoption of the index are shown below:

Username	
Execution time without index (ms)	22.22
Execution time with index (ms)	5.09

Table 14: Results obtained by creating the index

Figure 11: Cypher query analysis using the PROFILE tool in Neo4J (left: without index, right: with index)

Post Nodes

An index on the *id* field to optimize queries that suggest new posts has been defined.

The following cypher query has been run to obtain statistical data:

```
PROFILE MATCH (p:Post{id:'65a930a66448dd90156b3d35'}) return p
```

The performance gains that have been reached with the adoption of the index are shown below:

PostId	
Execution time without index (ms)	17.43
Execution time with index (ms)	5.91

Table 15: Results obtained by creating the index

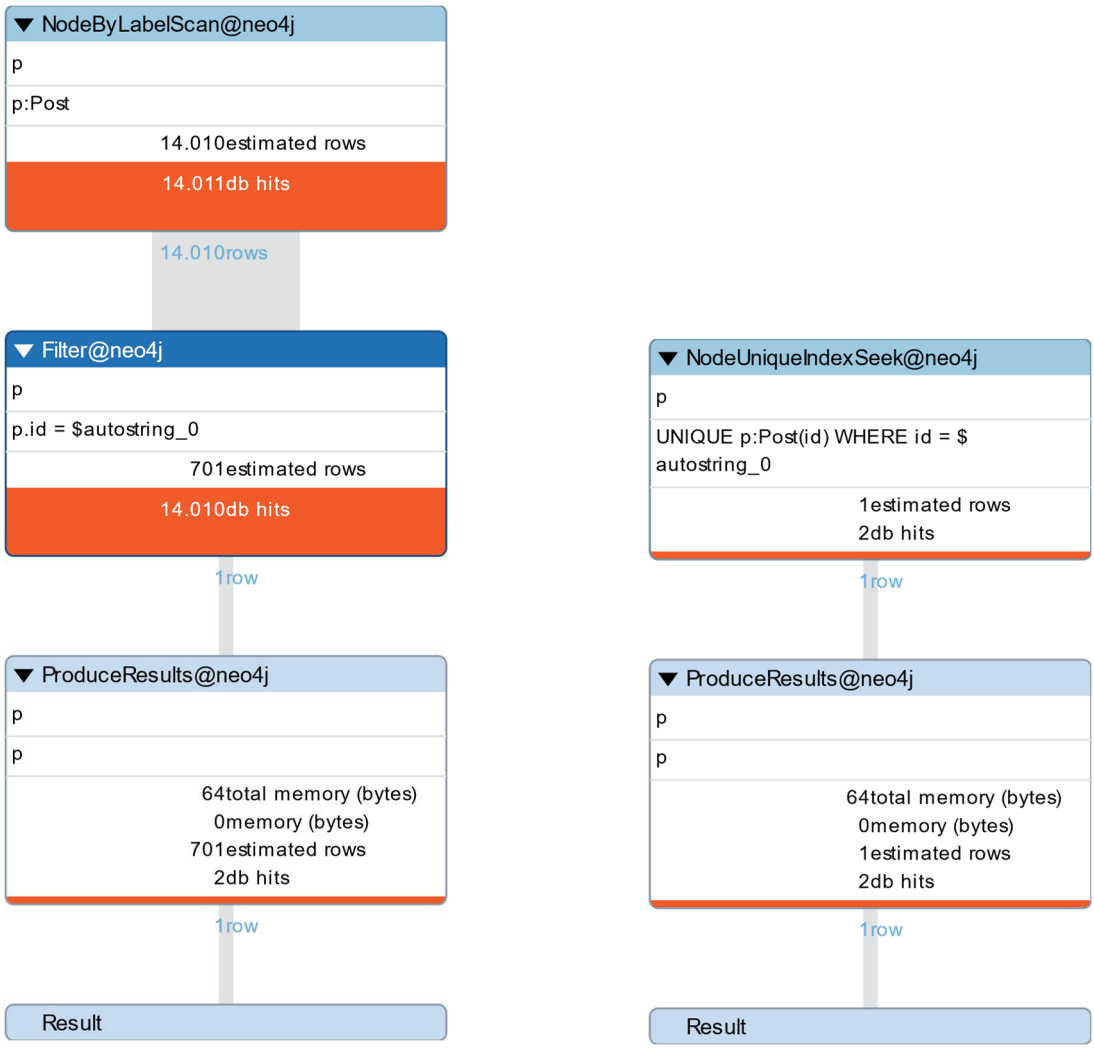

Figure 12: Cypher query analysis using the PROFILE tool in Neo4J (left: without index, right: with index)

Unit Test

A testing phase has been carried out in order to verify the correctness of the operations on the two different databases. Unit tests have been developed exploiting the *JUnit* Java framework, that allows to automatically run the developed methods and receive immediate feedback about the success or failure of the latter.

MongoDB Testing

The following table showcases a list and a short description of the test classes that have been developed to test the operations carried out by the application on MongoDB:

Class name	Description
BoardgameDBMongoTest	Test class for the CRUD operations carried out on the *Boardgames* collection.
PostDBMongoTest	Test class for the CRUD operations carried out on the *Posts* collection.
ReviewDBMongoTest	Test class for the CRUD operations carried out on the *Reviews* collection.
UserDBMongoTest	Test class for the CRUD operations carried out on the *Users* collection.

Table 16: MongoDB testing classes

Neo4J Testing

The following table showcases a list and a short description of the test classes that have been developed to test the operations carried out by the application on Neo4J:

Class name	Description
BoardgameDBNeo4jTest	Test class for the CRUD operations carried out on the *Boardgame* nodes and relationships.
PostDBNeo4jTest	Test class for the CRUD operations carried out on the *Post* node and relationships.
UserDBNeo4jTest	Test class for the CRUD operations carried out on the *Users* nodes and relationships.

Table 17: Neo4J testing classes

Conclusions

The development of this board games-themed social network application provided valuable insights into working with large-scale and multi-structured databases. The *Spring* Java framework, along with that *MVC* pattern, allowed the creation of a greatly scalable application that allows the connection of potentially thousands of boardgames aficionados. Board games enthusiasts can easily register to the application and navigate through its sections using an intuitive user interface, which allows them to discover a vast number of board games, review them, and create posts about their favourites. The combination of document-oriented and graph-based databases allowed an efficient management and querying of diverse types of data, providing scalability and flexibility.

Future Works

The application is, as it, a finite product that could be ready to be deployed on the market. However, several enhancements can be made to further improve its functionalities and performance, some of which are listed in this final paragraph:

- Adoption of ML techniques to implement an advanced recommendation system to provide an even better use experience
- Integration of real-time features, such as live chat and a notification system, could improve user engagement

Development of more admin-related feature, to provide the application adminostrators with an even more functional and complete dashboard.

References

The application's code is freely available at the following *GitHub* link:

- https://github.com/g-sferr/BoardGame-Cafe_App/tree/master