Università di Pisa

Department of Information Engineering

Large Scale and Multi-Structured Databases Project Report

GitHub: `https://github.com/Gabri00s/Progetto_Large_Scale.git`

Members:
Federico Cavedoni
Gabriele Silano
Simone Labella

Academic Year 2023/2024

Contents

Chapter 1

Introduction

1.1 Project Introduction

MONGODBGG is a dynamic web-based platform designed to enhance the board gaming experience for enthusiasts.

By providing a comprehensive database of board games, the platform allows users to discover, explore, and interact with a vast collection of games. It aims to foster a community where users can engage with one another by sharing their thoughts, following their favorite games, and reviewing their gaming experiences. Whether users are casual players or seasoned board game aficionados, MONGODBGG serves as a one-stop destination to deepen their involvement in the world of board games.

The application called MONGODBGG is developed as an application whose main functionalities consist of collecting board games, presenting detailed information about the games to users, allowing them to write reviews, and follow other users. The information about the games includes name, categories, mechanics, and rating, which are the filters that users can use to search for games in the database. Additionally, users can add the games they are interested in to their personal game list.

Users can write reviews about board games, including a comment and a rating, like other users' reviews, and follow other users to explore their gaming activity.

The application also provides game-related statistics accessible to all users, while administrators, with special permissions, can perform advanced queries and analytics to obtain detailed insights on games and user activities.

Chapter 2

Design Overview

2.1 System Overview and Specification

This section provides a general overview of the system and its specifications. The system is designed to meet the following objectives:

- **High reliability and availability.**

- **Scalability** to handle increasing loads.

The system consists of multiple components that interact seamlessly to deliver the intended functionality. The following sections will detail the primary actors, requirements, and the system's design through various diagrams.

2.2 Main Actors

The main actors in the system are:

- **Unregistered User:** An actor who can browse the system but has limited access to its features and cannot perform actions that require authentication.

- **Registered User:** An actor who has created an account and can access and manage personalized features and data within the system.

- **Admin:** The actor responsible for managing the system, including user accounts, content, and system settings, ensuring the system operates smoothly.

Each actor has specific roles and responsibilities, which are critical to the system's operations. Their interactions are described in detail in the use case diagram.

2.3 Requirements

The system requirements are categorized into functional and non-functional requirements, tailored to the different types of users interacting with the system.

2.3.1 Functional Requirements

The functional requirements define the capabilities and behaviors of the system for each type of user:

Unregistered User

- The system shall allow unregistered users to browse the boardgames.

- The system shall allow unregistered users to view boardgame reviews.

- The system shall provide registration functionality for unregistered users.

- The system shall allow unregistered users to log in.

- The system shall allow unregistered users to view other users.

Registered User

- The system shall allow registered users to manage their own matches.

- The system shall allow registered users to manage their own reviews.

- The system shall allow registered users to manage their likes and follows.

- The system shall allow registered users to manage their account informations.

- The system shall allow registered users to view their own statistics.

Admin

- The system shall allow admins to manage all user accounts.

- The system shall allow admins to manage all matches.

- The system shall allow admins to manage all reviews.

- The system shall allow admins to manage all board games.

- The system shall allow admins to view global statistics.

- The system shall allow admins to manage all likes and follows.

2.3.2 Non-functional Requirements

The non-functional requirements ensure the system's performance, reliability, scalability and security, maintaining its efficiency and quality. These requirements include:

- **Web Application Implementation:** The system shall be implemented as a web application, accessible from various devices and platforms.

- **High Availability:** The system shall emphasize high availability to provide uninterrupted service.

- **Object-Oriented Programming:** The system shall be developed using object-oriented programming languages to enhance code maintainability and reusability.

- **Encryption of User Passwords:** The system shall encrypt user passwords to protect against unauthorized access and ensure data security.

- **Scalability:** The system shall be scalable to accommodate increasing loads and support databases growth.

2.4 Use Case Diagram

The following use case diagram illustrates the interactions between the main actors and the system.
In this diagram, the relationships between the unregistered user, the registered user, and the admin with the system are clearly depicted, along with the use cases they interact with.

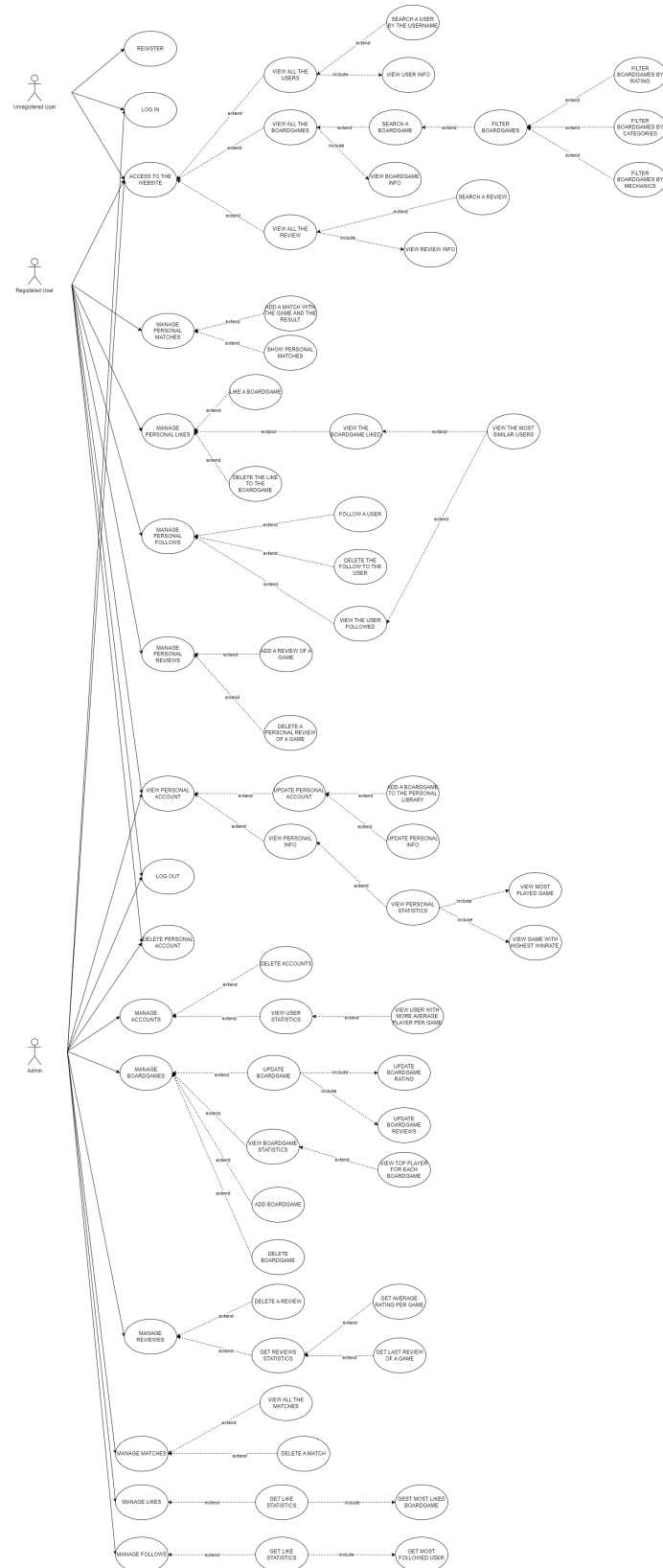Figure 2.1: Use Case Diagram

2.5   UML Class Diagram

The UML Class diagram represents the static structure of the system by showing its classes, attributes, operations, and the relationships between the objects.
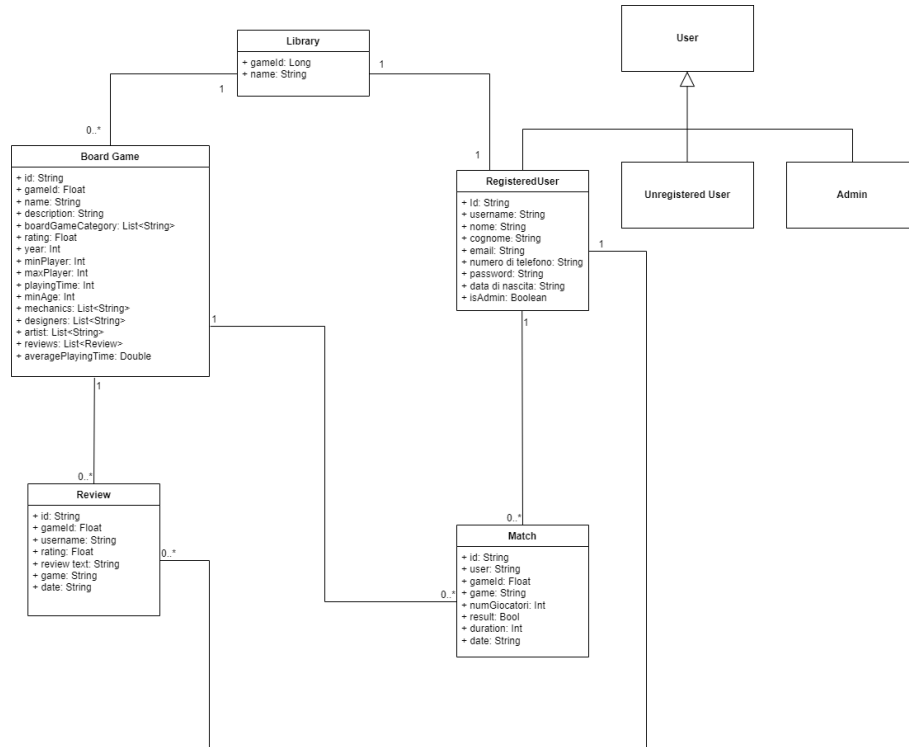
Figure 2.2: UML Class Diagram

Chapter 3

Dataset Creation

Our primary dataset was sourced from Kaggle. The dataset originates from **BoardGameGeek**, the largest online forum dedicated to board games and card games. This provided us with detailed information about board games and reviews related to them.

To meet the requirement for more diverse data, we scraped additional reviews from an important Italian board games forum, **"La Tana del Goblin."**

3.1 Extraction and Creation

While we had access to board game details and reviews, we needed additional information to complete our dataset:

- From the reviews, we extracted usernames, which we then used to generate additional user-related data.

- We created user libraries.

- We generated matches.

- We established follow relationships.

- We created like relationships.

For the user libraries, we opted to use a **normal distribution**. This approach reflects real-world scenarios where a few highly passionate users own many games, while the majority of users have a smaller number of games (occasional players). For simplicity and to better mirror reality, we assigned users all the games they reviewed.

To create matches and relationships, we applied the same consideration and used a normal distribution for both users and games. This ensures a mix of **highly active users** and popular games, alongside a larger number of **less active users** and less popular games.

3.2   Database

For our application, we decided to use both **MongoDB** and **Neo4j**. MongoDB was a project requirement, but it also proved highly useful due to its flexibility in managing unstructured data. It allows us to efficiently execute queries by embedding related data within documents.

Neo4j, on the other hand, is ideal for modeling and representing relationships between entities as a network, which is particularly useful for implementing graph-based queries and recommendations.

3.2.1   MongoDB

The MongoDB database is divided into four collections:

- **Users**

- **Reviews**

- **BoardGames**

- **Matches**

Table 3.1: Dimension

Collection	Documents	Storage Size
User	413k	650M
Reviews	19M	1.5G
BoardGames	22k	30M
Match	300k	20M

```
_id: ObjectId('66cc96d580ba5df27b4fb81e')
username : "shurei"
nome : "Amanda"
cognome : "Nelson"
email : "shurei@gmail.com"
numero di telefono : "3263683403"
password : "59b794907fdb85028ff84cb4d042aed4c7b93a2a014daa0877fd93447e8548ee"
data di nascita : "10-07-2003"
▼ library : Array (3)
  ▼ 0: Object
      id : 209010
      name : "Mechs vs. Minions"
  ▼ 1: Object
      id : 244271
      name : "Dice Throne: Season Two - Battle Chest"
  ▼ 2: Object
      id : 2077
      name : "Hell's Highway: Operation Market Garden"
isAdmin : false
```

Figure 3.1: User collection.

```
    _id: ObjectId('66cf226fe9e9a4e3c501cb78')
    gameId : 40692
    name : "Small World"
    description : "In Small World, players vie for conquest and control of a world that i…"
  ▾ boardgamecategory : Array (3)
      0: "Fantasy"
      1: "Fighting"
      2: "Territory Building"
    rating : 7.24546480178833
    year : 2009
    minplayers : 2
    maxplayers : 5
    playingtime : 80
    minage : 8
  ▸ boardgamemechanic : Array (8)
  ▸ boardgamedesigner : Array (1)
  ▸ boardgameartist : Array (2)
  ▾ reviews : Array (5)
    ▸ 0: Object
    ▾ 1: Object
        username : "Ravenhoe"
        rating : 7.75
        review text : "The perfect mix of randomness and strategy/skill. Simple rules but ver…"
    ▸ 2: Object
    ▸ 3: Object
    ▸ 4: Object
    averageplayingtime : 66
```

Figure 3.2: Boardgame collection.

```
    _id: ObjectId('66c3591c34d505924e467a99')
    username : "mitnachtKAUBO-I"
    rating : 10
    review text : "Hands down my favorite new game of BGG CON 2007.  We played it 5 times…"
    id : 30549
    game : "Pandemic"
    date : "2022-03-02 08:42:40"
```

Figure 3.3: Review collection.

```
    _id: ObjectId('66c3622134d505924e699339')
    user : "liq3"
    game : "Wizard's Academy"
    numgiocatori : 6
    result : true
    duration : 93
    date : "2020-03-15"
    gameId : 154895
```

Figure 3.4: Match collection.

Considerations on Reviews and Matches

Reviews could be embedded within the BoardGame collection, and matches within the User collection. However, we decided to create two separate collections because the size of reviews, especially those with extensive text, can grow significantly. This would make the BoardGame collection too large. Additionally, we chose this approach because we want to display only the latest five reviews for each board game, while a separate query will be required to view all the reviews.

Regarding matches, although the size of an individual match is not large, the quantity can be substantial, with many matches per game. Therefore, having a separate collection is beneficial.

3.2.2 Neo4j

In Neo4j, we have two types of nodes:

- **User (Username)**

- **BoardGame (ID,name)**

And two types of relationships:

- **Follow (User -> User)**

- **Liked (User -> BoardGame)**

This allows us to track the network's status and implement recommendation queries.

Considerations on Follow Relationships

We chose to implement this relationship as **unidirectional**. An alternative would be to introduce a property to track reciprocal follows, but despite our solution requiring more relationships and more memory, we believe it is the best choice because Neo4j is highly efficient at traversing these connections rather than handling attributes particularly for recommendation queries.
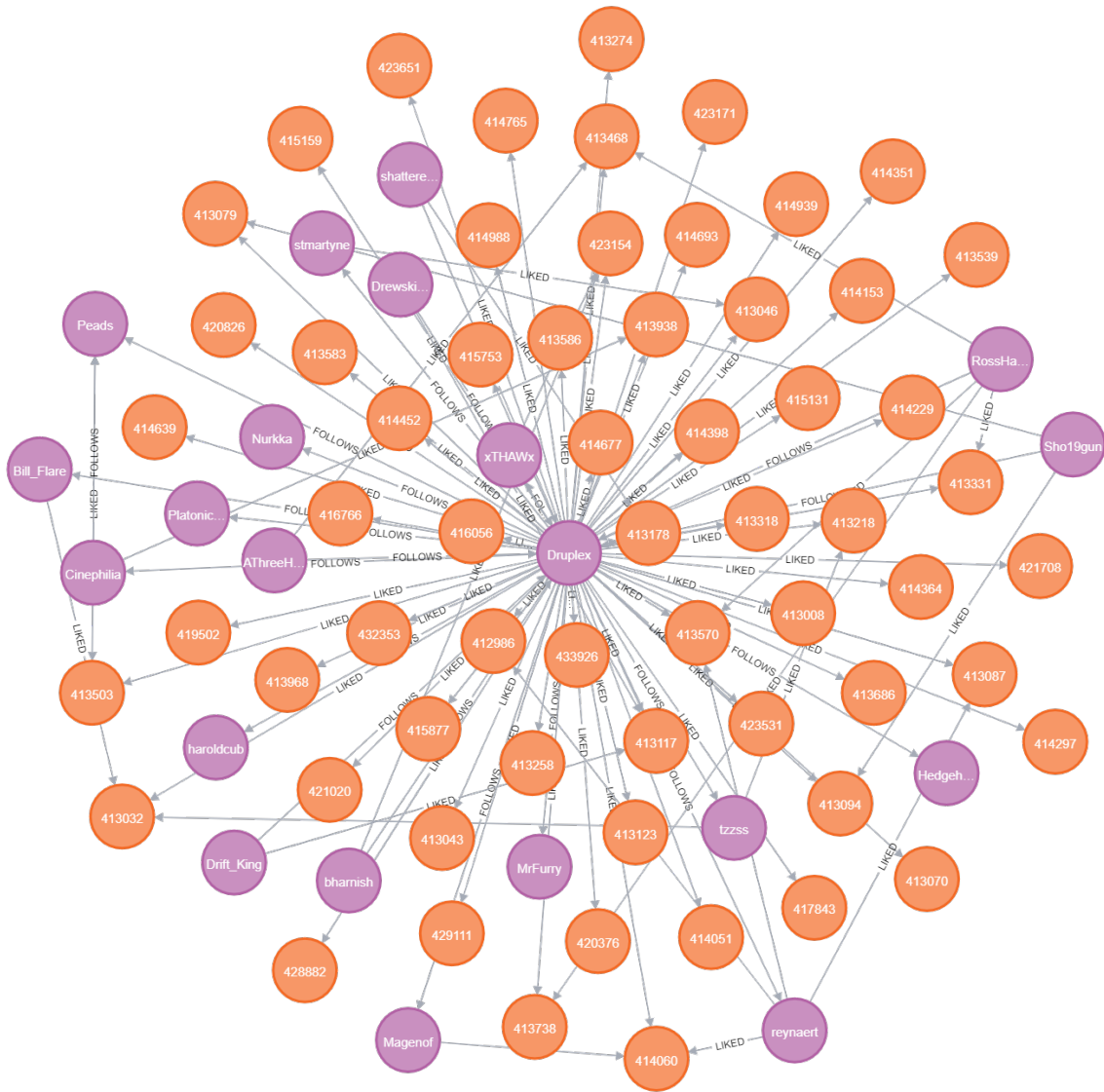
Figure 3.5: Example of a graph.

Chapter 4

Implementation

4.1 Implementation Frameworks

In this project, the implementation of our web server was greatly facilitated by leveraging the Spring Boot framework. Spring Boot was selected primarily due to its seamless integration of an embedded Tomcat web server, which significantly simplified the development and deployment of the server. This feature allowed us to avoid the complexities typically associated with web server configuration, enabling a more straightforward development process.

Furthermore, the use of the `HttpSession` class has been particularly advantageous in conjunction with the `@RestController` annotation. The `@RestController` annotation allowed us to efficiently map HTTP protocol requests to Java methods, thus providing the flexibility and efficiency required for handling HTTP requests and responses within our application. Overall, the Spring Boot framework played a crucial role in streamlining the development of our web-based components.

4.2 MVC Design Pattern

To ensure a clear separation of concerns within our application, we adopted the Model-View-Controller (MVC) design pattern. The MVC pattern is well-suited for non-trivial applications, as it divides the application into three interconnected components:

- **Model:** The Model is responsible for managing the data and business logic of the application. This component interacts directly with the database and is designed to facilitate efficient data handling and future modifications.

- **View:** The View handles the presentation layer of the application, managing the user interface and ensuring that the application's visual appearance can be easily maintained and updated.

- **Controller:** The Controller acts as an intermediary between the Model and the View, managing the flow of data and user interactions. It processes incoming HTTP requests, calls the necessary services, and returns the appropriate responses.

The adoption of the MVC pattern allowed us to maintain a clean and modular codebase, which improved collaboration among the development team and simplified the debugging process.

4.3 Project Structure and Package Organization

The main package is `it.unipi.lsmsd.LSMSD_Project` and the project is organized into several sub-packages. Each package serves a specific function within the application, ensuring a clear separation of concerns and facilitating maintainability and scalability. Below is a detailed description of each package and its key components.

4.3.1 it.unipi.lsmsd.LSMSD_Project.config

The `it.unipi.lsmsd.LSMSD_Project.config` package is responsible for configuring and customizing the application's connection to MongoDB. The primary class here is `MongoConfig`, which handles the specific setup required for connecting to the MongoDB instance. This includes defining the database name, configuring any custom converters, and setting up additional MongoDB properties if necessary.

```
@Bean
public MongoClient mongoClient() {
    MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(new com.mongodb.ConnectionString("mongodb://localhost:27017/Progetto"))
            .writeConcern(WriteConcern.W1)
            .readPreference(ReadPreference.nearest())
            .build();

    return MongoClients.create(settings);
}
```

Figure 4.1: Example configuration of MongoConfig in the config package.

4.3.2 it.unipi.lsmsd.LSMSD_Project.controller

The `it.unipi.lsmsd.LSMSD_Project.controller` package contains the Controllers responsible for exposing the APIs to access the web server and retrieve data. The main classes in this package include:

- `BoardGameController`: Manages API requests related to board games, such as retrieving game details and searching for games.

- `MatchController`: Handles API requests related to matches, including creating, updating, and retrieving match data.

- `RelationController`: Manages relationships between users, such as following and unfollowing users.

- `UserController`: Handles user-related API requests like user registration, login, and profile management.

- **ReviewController**: Manages review-related requests, such as adding, updating, and retrieving game reviews.


```java
@RestController
@RequestMapping(⊕∨"/api/matches")
public class MatchController {

    @Autowired
    private MatchService matchService;

    @PostMapping(⊕∨"/add")
    public ResponseEntity<?> addMatch(@RequestBody Match match, HttpSession session) {
        User currentUser = (User) session.getAttribute( s: "user");
        if (currentUser != null) {
            Match newMatch = matchService.addMatch(match);
            return ResponseEntity.ok(newMatch);
        } else {
            return new ResponseEntity<>( body: "Operazione non autorizzata", HttpStatus.UNAUTHORIZED);
        }
    }
}
```

Figure 4.2: Example of controllers in the controller package.

4.3.3 it.unipi.lsmsd.LSMSD_Project.dao

The `it.unipi.lsmsd.LSMSD_Project.dao` (Data Access Object) package includes the repositories that provide the interface to the databases. These repositories are crucial for data management and retrieval within the application. They not only manage basic CRUD (Create, Read, Update, Delete) operations but also implement more complex and significant queries tailored to the application's needs. The key classes include:

- **BoardGameRepository** and **UserRepository**: These interfaces manage CRUD operations for board games and users in MongoDB, along with advanced queries for searching and filtering data based on various criteria.

- **ReviewRepository** and **MatchRepository**: Handle CRUD operations for reviews and matches in MongoDB, and include custom queries to retrieve aggregated data and match history.

- **BoardGameNodeRepository** and **UserNodeRepository**: Manage data interactions with Neo4j, particularly for entities like board games and users that require graph-based data management. These repositories also include queries to explore and manipulate relationships within the graph.

17

```
public interface ReviewRepository extends MongoRepository<Review, Long> {  3 usages
    List<Review> findByUsername(String username);  1 usage
    List<Review> findByUsernameAndGameId(String username, Long gameId);  3 usages
    List<Review> findByGameId(Long gameId);  2 usages

    @Query("{ 'date': { $gte: ?0 } }")  2 usages
    List<Review> findReviewsAfterDate(String date);
}
```

Figure 4.3: Example of repository classes in the dao package.

4.3.4 it.unipi.lsmsd.LSMSD_Project.model

The `it.unipi.lsmsd.LSMSD_Project.model` package contains the domain models representing the application's core entities. These models map the data from the database to Java objects used throughout the application. Key classes include:

- **User**: Represents a user entity with attributes like username, email, and password.

- **Review**: Represents a review for a board game, including fields such as rating and comments.

- **Match**: Represents a match between users, including details like participants and scores.

- **BoardGame**: Represents a board game, encompassing attributes such as title, genre, and description.

In addition to these core classes, the `model` package also includes several other model classes that are less prominent but are essential for the comprehensive management of data within the application. These additional models support various features and ensure a seamless data handling experience across different components of the application.

```java
public class Review {
    @Id
    private String mongoId;

    @Field(name = "gameId")
    private Long gameId;

    @Field(name = "username")
    private String username;

    @Field(name = "rating")
    private float rating;

    @Field(name = "review text")
    private String reviewText;

    @Field(name = "game")
    private String game;

    @Field(name = "date")
    private String date;
}
```

Figure 4.4: Example of a model class in the model package.

4.3.5 it.unipi.lsmsd.LSMSD_Project.projections

The `it.unipi.lsmsd.LSMSD_Project.projections` package defines interfaces for data projections, allowing the application to return specific subsets of data when executing queries. This is especially useful for optimizing performance by only retrieving the necessary fields from the database, rather than loading entire entities.

4.3.6 it.unipi.lsmsd.LSMSD_Project.service

The `it.unipi.lsmsd.LSMSD_Project.service` package contains the business logic of the application. These classes implement the functionalities exposed by the controllers and interact with the repositories. Key services include:

- `BoardGameService`: Handles operations related to board games.

- `UserService`: Manages user-related operations like registration, login, and profile updates.

- `MatchService`: Handles the creation and management of matches.

- **RelationService**: Manages user relationships, including following and unfollowing functionalities.

- **ReviewService**: Handles the creation, updating, and retrieval of game reviews.

Figure 4.5: Example of service classes in the service package.

4.3.7 it.unipi.lsmsd.LSMSD_Project.utils

The `it.unipi.lsmsd.LSMSD_Project.utils` package includes utility classes that are reused across the application. Notable exceptions include:

- **InvalidCredentialException**: Thrown when a user provides invalid credentials during login.

- **UserAlreadyExistException**: Thrown when trying to register a user that already exists in the system.

4.4 Database Integration

For data persistence, we integrated two databases: MongoDB and Neo4j. MongoDB was used as our primary document store, handling large-scale data storage with its NoSQL capabilities. Neo4j, on the other hand, was employed to manage complex relationships between entities through its graph database capabilities.

Spring Boot's Spring Data module was instrumental in simplifying database interactions. We utilized Spring Data MongoDB and Spring Data Neo4j, both of which provided repositories with built-in CRUD operations, significantly reducing the amount of boilerplate code required for database interactions.

4.5 Testing and Deployment

To test the functionalities of our web server, we used Postman, a powerful tool for testing APIs. With Postman, we were able to simulate client requests, test various endpoints, and verify that our application handled different HTTP methods correctly. By using Postman, we ensured that the application behaved as expected under various scenarios without needing to implement a dedicated main interface for testing.

Chapter 5

MongoDB query

5.1 Read

- BoardGames

  - GetBoardGames(filterByCategories, filterByMechanics, filterByRating, Get-BoardGameByName, GetBoardGamesLimited)
  - GetBoardGamesDetails

- User

  - GetUsername, GetUsernameByName
  - GetOwnProfile,GetOtherUserProfile

- Review

  - GetReview(ByGame, ByUsername, ByReviewUsernameAndGame, ByRat-ing, GetOrderedByRating)

- Match

  - GetMatches(MatchByUser,MatchByGame,MatchByUserAndGame)

And other statistics query that we analyze later on the aggregation query

5.2 Write, Update and Delete

- BoardGames

  - AddBoardGame
  - DeleteBoardGame
  - UpdateBoardGame
  - UpdateRatings
  - UpdateReviews

- – UpdateAveragePlayingTime

- **User**

  - – RegisterUser
  - – UpdateProfile
  - – DeleteUser

- **Review**

  - – AddReview
  - – DeleteReview

- **Match**

  - – AddMatch

5.3 Relevant query

5.3.1 `UpdateReviews`

We decided to store the last five reviews in the BoardGame collection. When a user adds a review, it is initially inserted only into the Review collection to avoid writing to two different collections simultaneously. This means we may have some temporary inconsistency. Our solution is to run the `UpdateReviews` query every night, updating only the board games affected by new reviews.

5.3.2 `UpdateRatings`

The board game ratings also need to be updated periodically. Similar to reviews, we update this field at the end of each day using the `UpdateRatings` query, which only updates the board games that received new reviews.

5.3.3 `UpdateAveragePlayingTime`

The `averagePlayingTime` field is updated after users log new matches. However, we also update this field once per day using the `UpdateAveragePlayingTime` query.

5.4 Aggregation query

1. `getGameStatistics(Integer minMatches, Integer limit, Integer sortOrder)`

This query returns game statistics by aggregating the matches for each game.

- **`$group`**: Aggregates the data by the `game` field and calculates:

- totalMatches: The total number of matches played.
- totalWins: The total number of wins (summing 1 for each win, determined by $cond which checks the value of result).
- avgDuration: The average duration of the matches.

- $addFields: Adds a winRate field, calculated as the ratio of totalWins to totalMatches, multiplied by 100 to obtain a percentage.

- $match: Filters the results to include only games with at least minMatches.

- $sort: Sorts the results by winRate, based on the value of sortOrder (1 for ascending, -1 for descending).

- $limit: Limits the number of results to limit.

- $project: Projects only the relevant fields (game, totalMatches, totalWins, avgDuration, winRate).

```
@Aggregation(pipeline = {
    "{ $group: { _id: '$game', totalMatches: { $sum: 1 }, totalWins: { $sum: { $cond: ['$result', 1, 0] } }, avgDuration: { $avg: '$duration' } } }",
    "{ $addFields: { winRate: { $multiply: [ { $divide: ['$totalWins', '$totalMatches'] }, 100 ] } } }",
    "{ $match: { $expr: { $gte: [ '$totalMatches', ?0 ] } } }",
    "{ $sort: { winRate: ?2 } }",
    "{ $limit: ?1 }",
    "{ $project: { _id: 0, game: '$_id', totalMatches: 1, totalWins: 1, avgDuration: 1, winRate: 1 } }"
})
List<GameStatistic> getGameStatistics(Integer minMatches, Integer limit, Integer sortOrder);
```

Figure 5.1: getGameStatistics

2. getUserGameStatistics(Integer limit, Integer sortOrder)

This query calculates statistics for each user by aggregating the matches played by each user.

- $group: Aggregates the data by the user field and calculates:
  - totalMatches: The total number of matches played by each user.
  - totalWins: The total number of wins.
  - mostPlayedGame: The first game played by the user (according to insertion order).
  - leastPlayedGame: The last game played by the user.

- $addFields: Adds a winRate field to calculate the percentage of wins.

- $sort: Sorts the results by totalMatches based on the value of sortOrder.

- $limit: Limits the number of results to limit.

- $project: Projects only the relevant fields (user, totalMatches, winRate, mostPlayedGame, leastPlayedGame).

```
@Aggregation(pipeline = {
    "{ $group: { _id: '$user', totalMatches: { $sum: 1 }, totalWins: { $sum: { $cond: ['$result', 1, 0] } }, mostPlayedGame: { $first: '$game' }, " +
        "leastPlayedGame: { $last: '$game' } } }",
    "{ $addFields: { winRate: { $multiply: [ { $divide: ['$totalWins', '$totalMatches'] }, 100 ] } } }",
    "{ $sort: { totalMatches: ?1 } }",
    "{ $limit: ?0 }",
    "{ $project: { _id: 0, user: '$_id', totalMatches: 1, winRate: 1, mostPlayedGame: 1, leastPlayedGame: 1 } }"
})
List<UserGameStatistic> getUserGameStatistics(Integer limit, Integer sortOrder);
```

Figure 5.2: getUserGameStatistic

3. `getTopPlayersForEachGame(Integer limit, Integer minMatches)`

This query returns the top players for each game, based on wins and matches played.

- **$group:** Aggregates the data by a combination of `game` and `user`, calculating:
  - `totalMatches:` The total number of matches played by that user in that game.
  - `totalWins:` The total number of wins.
- **$addFields:** Adds a `winRate` field to calculate the percentage of wins.
- **$match:** Filters the results to include only those with at least `minMatches` played.
- **$group:** Groups the data again by `game`, selecting the top player (`topPlayer`) for each game.
- **$sort:** Sorts the results by the `_id` field (which corresponds to `game`).
- **$project:** Projects only the relevant fields (game, user, totalMatches, winRate).
- **$limit:** Limits the number of results to `limit`.

```
@Aggregation(pipeline = {
    "{ $group: { _id: { game: '$game', user: '$user' }, totalMatches: { $sum: 1 }, totalWins: { $sum: { $cond: ['$result', 1, 0] } } } }",
    "{ $addFields: { winRate: { $multiply: [ { $divide: ['$totalWins', '$totalMatches'] }, 100 ] } } }",
    "{ $match: { $expr: { $gte: [ '$totalMatches', ?1 ] } } }",
    "{ $group: { _id: '$_id.game', topPlayer: { $first: { user: '$_id.user', totalMatches: '$totalMatches', winRate: '$winRate' } } } }",
    "{ $sort: { '_id': 1 } }",
    "{ $project: { _id: 0, game: '$_id', user: '$topPlayer.user', totalMatches: '$topPlayer.totalMatches', winRate: '$topPlayer.winRate' } }",
    "{ $limit: ?0 }"
})
List<TopPlayerStatistic> getTopPlayersForEachGame(Integer limit, Integer minMatches);
```

Figure 5.3: getTopPlayerForEachGame

4. `findUsersWithHighestAvgNumPlayers(Integer limit, Integer minMatches)`

This query finds the users with the highest average number of players (`numGiocatori`) in the matches played.

- **$group:** Aggregates the data by `user`, calculating:

- – `totalMatches`: The total number of matches played by the user.

- – `totalWins`: The total number of wins.

- – `avgNumGiocatori`: The average number of players (`numgiocatori`).

- `$addFields`: Adds the `winRate` and `weightedWinRate` fields (the latter is the winRate weighted by the average number of players).

- `$match`: Filters the results to include only users who have played at least `minMatches`.

- `$sort`: Sorts the results by `avgNumGiocatori` in descending order.

- `$project`: Projects only the relevant fields (user, avgNumGiocatori, winRate, weightedWinRate, totalMatches).

- `$limit`: Limits the number of results to `limit`.

```
@Aggregation(pipeline = {
    "{ $group: { _id: '$user', totalMatches: { $sum: 1 }, totalWins: { $sum: { $cond: ['$result', 1, 0] } }, avgNumGiocatori: { $avg: '$numgiocatori' } } }",
    "{ $addFields: { winRate: { $multiply: [ { $divide: ['$totalWins', '$totalMatches'] }, 100 ] }, weightedWinRate: { $multiply: [{ $divide: ['$totalWins', " +
        "'$totalMatches'] }, '$avgNumGiocatori'] } } }",
    "{ $match: { $expr: { $gte: [ '$totalMatches', ?1 ] } } }",
    "{ $sort: { avgNumGiocatori: -1 } }",
    "{ $project: { _id: 0, user: '$_id', avgNumGiocatori: 1, winRate: 1, weightedWinRate: 1, totalMatches: 1 } }",
    "{ $limit: ?0 }"
})
List<TopAvgPlayersStatistic> findUsersWithHighestAvgNumGiocatori(Integer limit, Integer minMatches);
```

Figure 5.4: findUsersWithHighestAvgNumPlayers

5. `findTopPlayerByGameId(long gameId)`

This query finds the top player for a specific game (identified by `gameId`).

- `$match`: Filters the data by `gameId`.

- `$group`: Aggregates the data by `user`, calculating:

  - – `totalMatches`: The total number of matches played by the user in that game.

  - – `totalWins`: The total number of wins.

- `$addFields`: Adds a `winRate` field to calculate the percentage of wins.

- `$sort`: Sorts the results by `winRate` in descending order.

- `$limit`: Limits the result to the top player.

- `$project`: Projects only the relevant fields (user, game, winRate, total-Matches).

26

Figure 5.5: findTopPlayerByGameId

6. `findMostPlayedGameByMatches()`

This query finds the game with the highest number of matches played.

- `$group:` Aggregates the data by `game`, calculating the total number of matches played (`totalMatches`).

- `$sort:` Sorts the results by `totalMatches` in descending order.

- `$limit:` Limits the result to the game with the most matches.

- `$project:` Projects only the relevant fields (game, totalMatches).

Figure 5.6: findMostPlayedGameByMatches

7. `findMostPlayedGameByTime()`

This query finds the game with the highest total time played.

- `$group:` Aggregates the data by `game`, calculating the total time played (`totalTimePlayed`).

- `$sort:` Sorts the results by `totalTimePlayed` in descending order.

- `$limit:` Limits the result to the game with the most time played.

- `$project:` Projects only the relevant fields (game, totalTimePlayed).

```
@Aggregation(pipeline = {
        "{ $group: { _id: '$game', totalTimePlayed: { $sum: '$duration' } } }",
        "{ $sort: { totalTimePlayed: -1 } }",
        "{ $limit: 1 }",
        "{ $project: { _id: 0, game: '$_id', totalTimePlayed: 1 } }"
})
TopGameStatistic findMostPlayedGameByTime();
```

Figure 5.7: findMostPlayedGameByTime

These queries enable the extraction of complex statistics about games and users by leveraging the power of MongoDB's aggregation pipelines.

5.4.1   All query

Nome della Query	Permessi per eseguire la query	Accessi effettuati
AddBoardGame	Admin	
DeleteBoardGame	Admin	
getBoardGames(and Filter)	All	
getBoardGamesLimited		
getBoardGamesDetails		
getBoardGameByName		
UpdateBoardGame	Admin	
UpdateRatings	Admin	
UpdateReviews	Admin	
UpdateAveragePlayingTime		
AddReview		
DeleteReview	Different for User and Admin	
getReviewByUsername		
getReviewByGame		
ReviewUsernameAndGame		
getReviewsByRating		
getOrderedByRating		
RegisterUser		
GetUsername		
GetOwnProfile		
GetOtherUserProfile		
getUsernamebyName		
ProvaLogin		
UpdateProfile		
Logout		
DeleteUser		
GetMatches		
MatchByUser		
addMatch		
getGameStatistics		
getUserStatistics		
getTopPlayer		
getTopPlayerByGame		
getmostPlayedGameByMatches		
getmostPlayedGameByTime		
getTopAvgPlayer		
addFollow		
addLike		
Deletefollowuser		
Deletelikeuser		
followed		
followers		
likedGames		
getTopFollowedUser		
getTopLikedGame		
userMostSimilar		

Table 5.1: Tabella delle query con permessi e accessi

Chapter 6

Neo4j query

6.1 Follows

- AddFollow
- DeleteFollowedUser
- getFollowed
- getFollowers

6.2 Like

- AddLike
- DeleteLike
- getLikedGame

6.3 Recommendation Queries Overview

This section provides an overview of the recommendation queries implemented in the system. These queries are designed to suggest board games and users based on interactions and preferences of other users. Each query serves a specific purpose in providing recommendations.

6.3.1 Recommended Board Games for the User

```
@Query("MATCH (u:User {username: $username})-[:FOLLOWS]->(followed:User)-[:LIKED]->(b:BoardGame) " +
       "WHERE NOT (u)-[:LIKED]->(b) " +
       "RETURN b.id AS id, b.name AS name, count(b) AS likeCount " +
       "ORDER BY likeCount DESC " +
       "LIMIT $n")
List<Map<String, Object>> findTopBoardGamesForUser(String username, int n);
```

Figure 6.1: getTopBoardGamesForUser

This query identifies the most popular board games among the users that a given user is following. It **finds games that have been liked by users followed by the current user**, excluding those that the current user has already liked. The results are ordered by the number of likes in descending order, providing a list of recommended games that the user might enjoy.

6.3.2 Most Followed Users

```
@Query("MATCH (u:User {username: $username})-[:FOLLOWS]->(followed:User)-[:FOLLOWS]->(target:User) " +
        "RETURN target.username AS followedUser, count(target) AS followCount " +
        "ORDER BY followCount DESC " +
        "LIMIT $n")
List<Map<String, Object>> findTopFollowedUsers(String username, int n);
```

Figure 6.2: getTopFollowedUsers

This query identifies the **most followed users among those that a given user is following**. It examines users who are followed by users followed by the current user and ranks the results based on the number of followers in descending order. This helps in recommending influential or popular users based on the current user's network of connections.

6.3.3 Most Similar Users

```
@Query("MATCH (u:User {username: $username})-[:LIKED]->(b:BoardGame)<-[:LIKED]-(other:User) " +
        "WHERE u <> other " +
        "WITH other, count(b) AS commonGames " +
        "ORDER BY commonGames DESC " +
        "RETURN other.username AS username, commonGames " +
        "LIMIT $n")
List<Map<String, Object>> findMostSimilarUsers(String username, int n);
```

Figure 6.3: getUserMostSimilar

This query finds users who are most similar to a given user based on the board games they have both liked. It excludes the user themselves from the search and orders the results by the number of common games in descending order. It provides a list of users with similar preferences in board games, helping to discover people with aligned tastes.

These queries are designed to enhance the user experience by suggesting relevant content and connections based on real user interaction data.

Chapter 7

Database Design Decisions

This chapter outlines the critical design decisions made for the database architecture, including indexing strategies, sharding, replication, and considerations of the CAP theorem. These decisions are essential to ensure the system's performance, scalability, and reliability.

7.1 Indexing in MongoDB

Indexing is a crucial part of database design in MongoDB, used to improve the performance of queries and updates. As datasets expand, proper indexing becomes essential for maintaining efficient operations. This section describes the creation and impact of indexes on our system, focusing on optimizing specific operations in our web application.

7.1.1 Index on User Collection for Specific Games

To efficiently check if a particular game is in a user's library, important for allowing users to add Matches or write Reviews, we initially created a simple index on the `username` field. This optimization aids in speeding up retrieval processes.

Index Creation

- **Index:** `db.User.createIndex({ username:1 })`

Performance Analysis

Without the index, MongoDB must scan the entire collection to find matching documents, which can be slow. The index enables MongoDB to directly locate relevant documents based on the username, thereby minimizing search time and improving performance.

- **Query:** `db.User.find({username:  "torguem", library: { $in:  ["Caylus"] }}).explain("executionStats")`

```
executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 10071,
    totalKeysExamined: 0,
    totalDocsExamined: 412816,
```

Figure 7.1: Performance Without Index for Checking User Library

```
executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 25,
    totalKeysExamined: 0,
    totalDocsExamined: 0,
```

Figure 7.2: Performance With Index for Checking User Library

7.1.2   Index on Matches Associated with Users

Another common operation is retrieving all matches associated with a specific user. To optimize this, we created an index on the `user` field in the `Match` collection, ensuring that match data is retrieved efficiently.

Index Creation

- **Index:** `db.Match.createIndex({ user:  1 })`

Performance Analysis

Indexing the `user` field reduces the need for a full collection scan, allowing MongoDB to quickly find and retrieve all relevant matches for a user. This optimization is crucial for maintaining responsive performance as the dataset grows.

- **Query:** `db.Match.find({ user:  "liq3" })`
  `.explain("executionStats")`

Figure 7.3: Performance Without Index for Retrieving User Matches

Figure 7.4: Performance With Index for Retrieving User Matches

7.1.3 Indexes on Match and Review Collections for Boardgame Updates

To efficiently update boardgames with average playing times, ratings, and reviews, we created indexes on the `gameId` field in both the `Match` and `Review` collections. These indexes are essential for quickly aggregating the necessary data to update each boardgame's statistics.

Index Creation

- **Match Collection Index:** `db.Match.createIndex({ gameId:1 })`

- **Review Collection Index:** `db.Review.createIndex({ gameId:1 })`

Performance Analysis

Indexing the `gameId` field in both collections greatly improves the performance of queries that aggregate data for boardgame updates. The 'Match' index allows MongoDB to quickly find and compute the average playing time, while the 'Review'

index facilitates the rapid retrieval of ratings and reviews associated with a specific game.

Match Collection

- **Query:** `db.Match.find(gameId:30549).explain("executionStats")`

Figure 7.5: Performance Without Index for Calculating Average Playing Time

Figure 7.6: Performance With Index for Calculating Average Playing Time

Review Collection

- **Query:** `db.Review.find(gameId:30549).explain("executionStats")`

```
executionStats: {
    executionSuccess: true,
    nReturned: 109342,
    executionTimeMillis: 19211,
    totalKeysExamined: 0,
    totalDocsExamined: 19077279,
```

Figure 7.7: Performance Without Index for Calculating Average Rating and Reviews

```
executionStats: {
    executionSuccess: true,
    nReturned: 109342,
    executionTimeMillis: 639,
    totalKeysExamined: 109342,
    totalDocsExamined: 109342,
```

Figure 7.8: Performance With Index for Calculating Average Rating and Reviews

Impact on System Performance

The indexes on the `gameId` field in both the 'Match' and 'Review' collections ensure that updates to boardgames—such as recalculating average playing time, average rating, and total reviews—are performed efficiently. These indexes minimize the need for full collection scans, significantly reducing query execution times and maintaining high performance as the dataset grows. This optimization is essential for keeping the system scalable and responsive, even as the volume of match and review data increases.

7.2 Index on Neo4j

Indexes in Neo4j can be extremely important for improving query performance. We decided to create two indexes: one on the **User** label for the username field, and another on the **BoardGames** label for the id field.

7.2.1 User Index

Queries involving user searches are very common and frequent. With more than 400,000 users in the system, this index significantly improves performance.
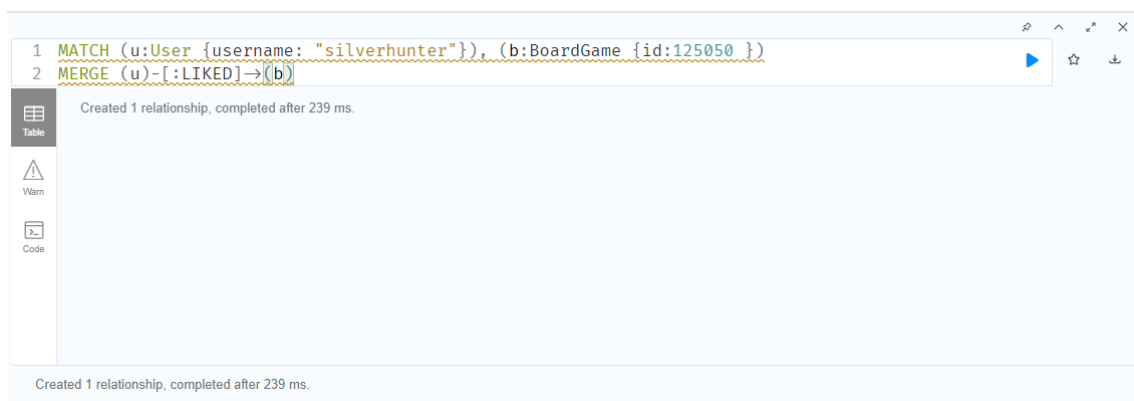

Figure 7.9: Insert follow relation with index

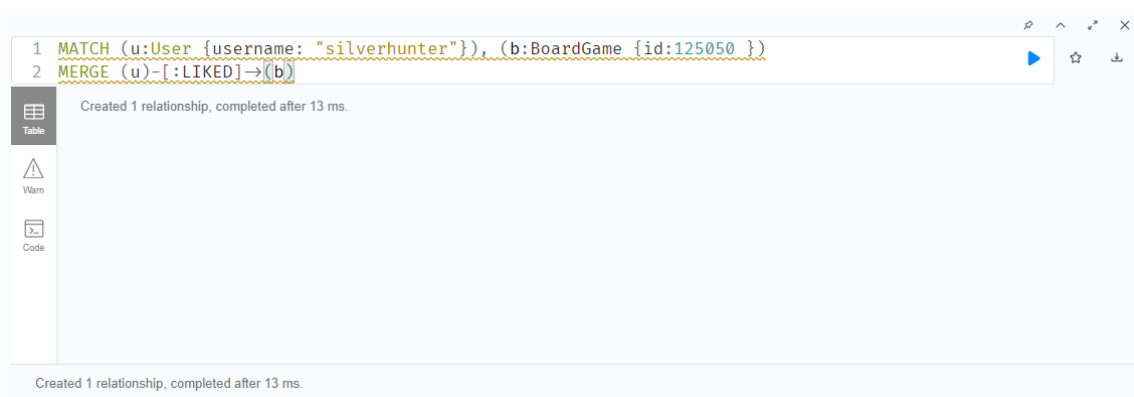Figure 7.10: Insert like relation without index

Figure 7.11: Insert like relation with index

We also observed improvements in recommendation queries like the following:

Figure 7.12: Find most similar users without index

Figure 7.13: Find most similar users with index

There are other queries that showed improvements as well, but for simplicity, we have only included these examples, which are sufficient to demonstrate that implementing this index is a good decision.

7.2.2 BoardGame Index

The main query that showed a significant improvement with the BoardGame index is the "like" insertion query.

As we can see in Figure 7.10, the time without indexes is 239 ms. In Figure 7.11, after adding the index on the user, the time drops to 13 ms. Additionally, when we add the index on the BoardGame, the result is as follows:
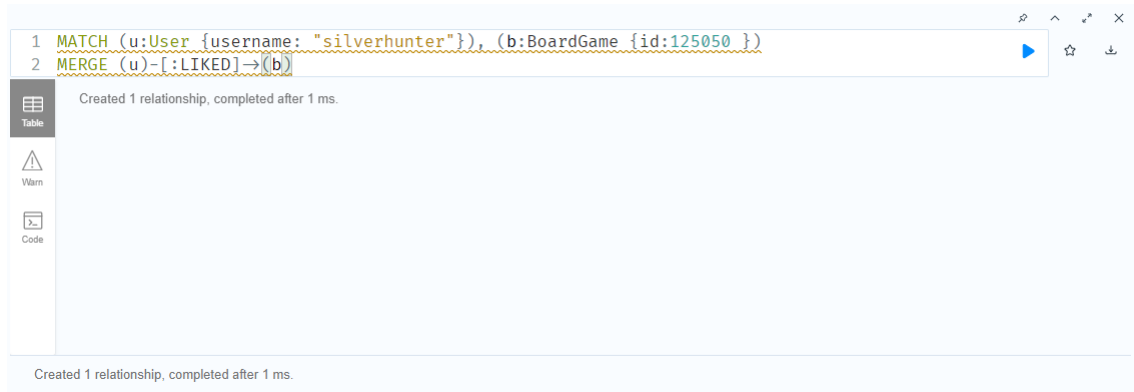

Figure 7.14: Insert like relation with both indexes

We observe a further improvement of 12 ms. Since the "like" insertion is likely the most frequent query in the application, we decided to keep the index.

7.3 Sharding Strategy for MongoDB Collections

Sharding is essential for scaling MongoDB databases by distributing data across multiple servers. This approach improves performance and allows the system to handle larger datasets effectively. This chapter outlines the proposed sharding strategy for each collection in our MongoDB database, detailing the chosen shard keys and sharding methods along with their rationale.

7.3.1 User Collection

Shard Key: `username`
Sharding Method: Hash

Hash-based sharding on the `username` field ensures an even distribution of user data across shards. This method prevents hotspots and balances the load, which is crucial for queries related to user authentication and profile access. By hashing `username`, we distribute user-related queries evenly, which enhances performance and avoids scenarios where some shards may become overloaded.

7.3.2 BoardGame Collection

Shard Key: `gameId`
Sharding Method: Hash

Using `gameId` as the shard key with hash-based sharding ensures uniform data distribution for board game information. Unlike using `name`, which could suffer from

issues related to non-unique or similar names, `gameId` is unique and provides a more consistent distribution of data across shards. This approach helps avoid hotspots and ensures scalable performance as the number of board games grows.

7.3.3 Match Collection

Shard Key: `username`
Sharding Method: Hash

Hash-based sharding on the `username` field is chosen for the `Match` collection to ensure an even distribution of user-specific game session data. As the number of matches grows over time, hashing on `username` prevents hotspots and maintains balanced query loads across shards. This method supports efficient performance and scalability, accommodating the increasing volume of match data without affecting system performance. It also aligns with the need to frequently access match data by user, ensuring that user-specific queries remain efficient.

7.3.4 Review Collection

Shard Key: `gameId`
Sharding Method: Hash

For the `Review` collection, sharding by `gameId` with a hash-based method ensures balanced data distribution and prevents hotspots. Given that the number of reviews will likely grow as more games are reviewed, this approach optimizes the retrieval of reviews and maintains consistent performance. Hashing `gameId` ensures that review data is evenly spread across shards, accommodating the expansion of the review collection efficiently.

7.3.5 Performance Evaluation of the Proposed Strategy

The proposed sharding strategy utilizes hash-based sharding for collections where data growth and distribution patterns could lead to hotspots or uneven load, such as in the `Match` and `Review` collections. For collections like `User` and `BoardGame`, hashing on unique identifiers (`username` and `gameId`) ensures balanced data distribution and efficient query performance. This approach supports scalability, enhances performance, and avoids issues related to uneven data distribution, making it well-suited to handle the anticipated growth in these collections. Regular monitoring and adjustments will be essential to maintaining optimal system performance as data volumes and access patterns evolve.

7.4 Replication

MongoDB supports high availability and data redundancy through replication, which ensures that data is consistently available across multiple servers. This section describes the replication setup for our MongoDB deployment and highlights the benefits of the chosen configuration.

7.4.1 Replication Configuration

Our MongoDB deployment utilizes a replica set with one primary server and two secondary servers. The replica set configuration is as follows:

```
rsconf = {
  _id: "rs0",
  members: [
    {
      _id: 0,
      host: "localhost:27018",
      priority: 5
    },
    {
      _id: 1,
      host: "localhost:27019",
      priority: 3
    },
    {
      _id: 2,
      host: "localhost:27020",
      priority: 1
    }
  ]
}
```

In this setup:

- **Primary Server:** The server at `localhost:27018` is configured with the highest priority (`priority: 5`). This server is responsible for handling all write operations and is the primary point of data consistency.

- **Secondary Servers:** The servers at `localhost:27019` and `localhost:27020` serve as secondary replicas, with priorities of 3 and 1, respectively. These servers replicate data from the primary and can be elected as the new primary in case of a failure.

7.4.2 Replication Benefits

The replication setup provides several key advantages:

- **High Availability:** By maintaining multiple copies of the data across different servers, MongoDB ensures that data remains accessible even if the primary server fails. The secondary servers can take over as the primary in the event of a failure, minimizing downtime.

- **Data Redundancy:** Replication protects against data loss by keeping multiple copies of the data. If one server encounters an issue, the data can still be retrieved from the other replicas.

- **Read Scalability:** Secondary servers can be used to distribute read requests, which helps balance the load and improves performance for read-heavy operations. This setup allows the primary server to focus on write operations while the secondary servers handle read queries.

- **Automatic Failover and Election:** In the event of a primary server failure, the replica set automatically performs an election to choose a new primary. This process ensures continuity of operations with minimal manual intervention.

7.4.3 Read Preferences and Write Concerns

To optimize performance and ensure data consistency, we have configured the following parameters:

- **Read Preference: `nearest`**
  The `nearest` read preference setting directs read operations to the closest available replica. This configuration reduces read latency by minimizing the distance between the client and the server handling the read request. It also helps distribute read loads among secondary servers, enhancing overall system performance.

- **Write Concern: `1`**
  The `writeConcern: 1` setting ensures that a write operation is acknowledged by the primary server before it is considered successful. This level of write concern provides a balance between performance and data durability, ensuring that writes are committed to the primary while avoiding the overhead of waiting for acknowledgment from secondary servers.

7.5 CAP Theorem and System Design Choices

The CAP theorem, also known as Brewer's theorem, is a fundamental principle in distributed systems that states that it is impossible for a distributed data store to simultaneously achieve all three of the following guarantees:

- **Consistency:** All nodes in the system see the same data at the same time.

- **Availability:** Every request receives a response, either with the requested data or an error, even if some nodes are unavailable.

- **Partition Tolerance:** The system continues to operate despite network partitions or communication failures between nodes.

Given these constraints, a distributed system must choose which of these properties to prioritize. In our MongoDB deployment, we have made the following design choices:

7.5.1 Chosen Approach: AP (Availability and Partition Tolerance)

In our MongoDB setup, we prioritize Availability and Partition Tolerance (AP) over Consistency. This choice is motivated by the following considerations:

- **High Availability:** By configuring our replica set with one primary and two secondary servers, we ensure that the system remains operational even if one or more servers fail. The primary server handles all write operations and can fail over to a secondary server if necessary, maintaining the system's overall availability.

- **Partition Tolerance:** Our setup is designed to handle network partitions and communication failures. MongoDB's replica set architecture allows the system to continue functioning even if some nodes are temporarily unreachable. The replica set can elect a new primary server if the current primary becomes unavailable, ensuring that the system remains responsive and functional despite partitioning issues.

- **Trade-off with Consistency:** In prioritizing Availability and Partition Tolerance, we accept a trade-off with Consistency. In our configuration, there may be a delay before all secondary nodes are updated with the latest data. We use a `writeConcern` of 1, which ensures that a write is acknowledged by the primary server but does not wait for acknowledgments from secondaries. This approach reduces the impact of write latency but may lead to scenarios where some reads return stale data if the secondary servers have not yet synchronized with the primary.

7.5.2 Implications and Benefits

Choosing AP ensures that our system can handle high levels of traffic and network issues without sacrificing overall availability. Users of the system will experience minimal disruptions even if certain servers are down or experiencing connectivity issues. The trade-off with Consistency means that while the system may occasionally return slightly outdated data, it remains operational and responsive under adverse conditions.

In summary, our decision to adopt the AP approach aligns with the need for a robust and highly available distributed system that can handle network partitions and continue to operate reliably even in the face of server failures or communication issues. This design choice reflects our commitment to providing a dependable and resilient system for handling data in a distributed environment.