University of Pisa

Large Scale and Multi Structured Databases

Project

BOOK-HUB

Marco Bologna
Chang Liu
Giacomo Moro

A.Y. 2023-2024

Contents

1 Introduction

Book-Hub is a Java application designed for avid readers who seek a centralized platform to rate and review their favorite books and get inspired with new things to read. The application not only facilitates individual book assessments but also fosters a community where users can follow friends, favorite authors, and receive personalized book recommendations.
Emphasizing simplicity and efficiency, Book-Hub leverages non-relational databases to manage substantial volumes of data seamlessly.

All the code of the application is accessible following this link:
https://github.com/MarcoBolo001/Book-Hub

1.1 Objective

The main aims of the project are essentially 2:

- **Unified Book Database:** Book-Hub strives to consolidate a diverse array of literary works from various sources into a singular, user-friendly database. By merging books from open-access archives, the application provides users with swift and efficient search capabilities, ensuring a comprehensive collection of titles at their fingertips.

- **Social Network for Readers:** Beyond individual interactions with books, Book-Hub aspires to create a vibrant social network for readers. Users can express opinions through reviews, follow their favorite authors to keep updated to their new books and follow each other to get inspired by others book's tastes. The platform encourages a dynamic exchange of ideas among its community of book enthusiasts.

1.2 Notes

Emphasizing simplicity and efficiency, Book-Hub leverages non-relational databases to manage substantial volumes of data seamlessly. As opposed to traditional relational databases, non-relational databases offer flexibility in handling unstructured and diverse data types associated with books, reviews, and user interactions. With the command line interface serving as the application's primary mode of interaction, non-relational databases provide **scalability** and efficient data retrieval, ensuring optimal performance even as the user base grows.

2 Dataset

The dataset used in our application contains information about Books, Users and Reviews.
In order to respect the variety constraint we downloaded reviews, users and books information from two different sources:

- Amazon Book Reviews, conveniently provided in CSV format, required minimal preprocessing.
- GoodReads Dataset, initially comprising disparate JSON files, underwent a comprehensive preprocessing phase.

The total amount of Raw Data was around 12 GB.

2.1 Merge and Preprocessing

In the preprocessing phase we converted all data in the same format (csv) to easily merging them.
Than we merged data using the ISBN code of books, and we differ Users using the Username.
At the end off the preprocessing phase we used only a sampled version of the database and then we built the collections to put in our document DB;
introducing some redundancies (to better support some of our requirements) we end up with around 300MB of document DB data.
Then we created the nodes and relationship for the graph DB and we used a Python script to populate the social network part of the application creating random follow relationships between Users, Authors and Genres.
At the end of this phase we end up with around 120MB of graph DB data.

3 Application Design

3.1 Actors

The application has 3 main actors:

- **Unregistered User**: Not yet registered user, must sign-up to access the application
- **Registered User**: User that is registered, can access the application by logging in
- **Administrator**: User with highest level of privilege, he can ban users and delete reviews

3.2 Requirements

This section describes the requirements that the application provides

3.2.1 Functional Requirements

Unregistered User can:

- Register to the platform
- Log in to the platform
- Search and view Books

Registered User can:

- Search and view Books
- Search and view Users
- Review and rate a Book
- Follow or Unfollow other users
- Follow or Unfollow his favorites authors
- Select his preferred genre
- Get recommendation of books based on his favorite genre and followed users, or based on liked authors
- Get recommendation of new users to follow
- View his own profile
- Update account information
- Log out

Administrator can:

- Search and view Books
- Search and view Users
- View his own profile
- Ban Users
- Delete reviews
- Add new books to the collection
- Update account information
- View User Statistics (most versatile Users, most followed users, most active users, top *bad* users)
- View Book Statistics (top categories, most rated Authors)
- Log out

3.2.2 Non-Functional Requirements

- The application needs to be highly available and always online.
- The system needs to be tolerant to data loss and to single point of failure.
- The application needs to provide fast response search results to improve the user experience.
- The application is implemented using a object-oriented programming language.
- The application is implemented by a user-friendly Command-Line Interface.
- The application needs to manage the consistency between databases of different types.

3.3 Use case Diagram

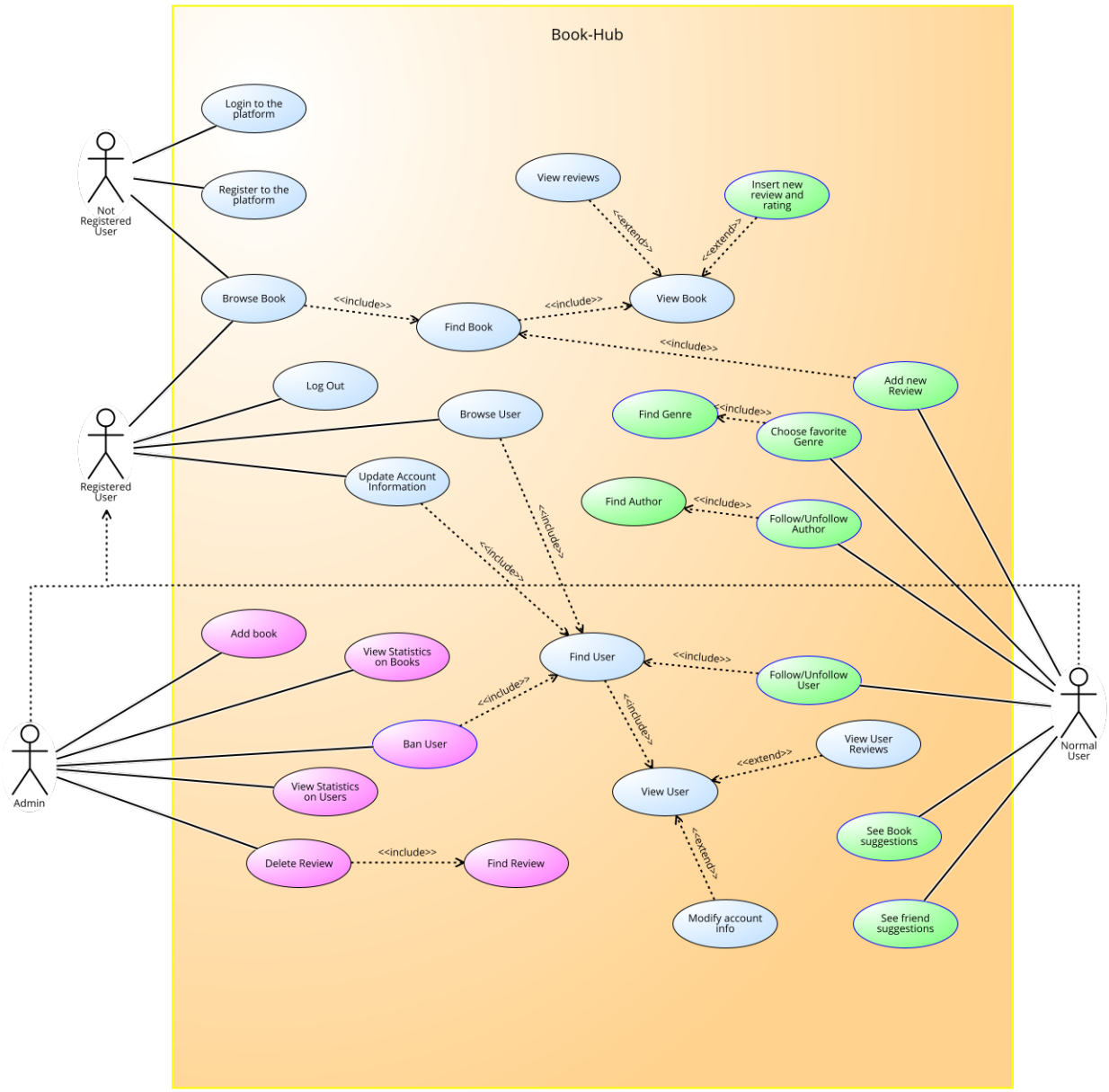


Figure 1: UML Use Case Diagram

3.4 UML class diagram

In the UML class diagram, five main entities are depicted: "Book," "User," "Review", "Author" and "Genre".
The "User" entity serves as a generalization for the two primary actors identified in the use case diagram, namely, "Registered User" and "Administrator."
This hierarchy signifies that both specialized actors inherit the core actions associated with the more generalized "User" entity, streamlining the representation of user-related functionalities in the system.

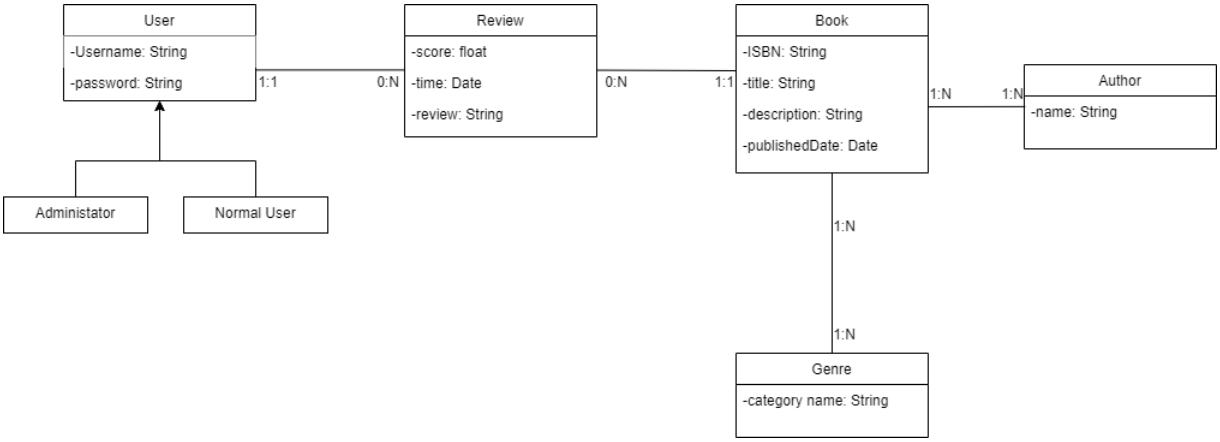


Figure 2: UML Analysis Classes Diagram

We resolve the generalization adding one attribute to the class user for specify the role of the generic user.

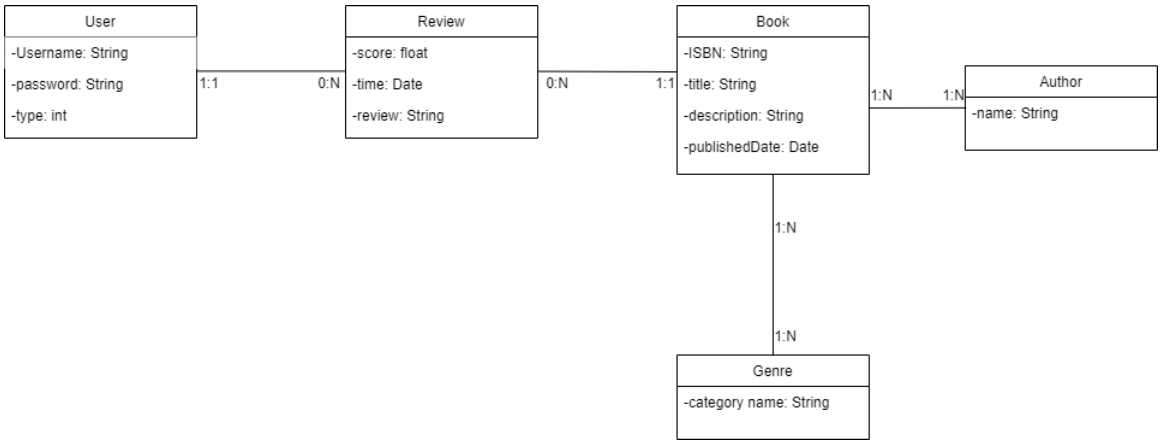
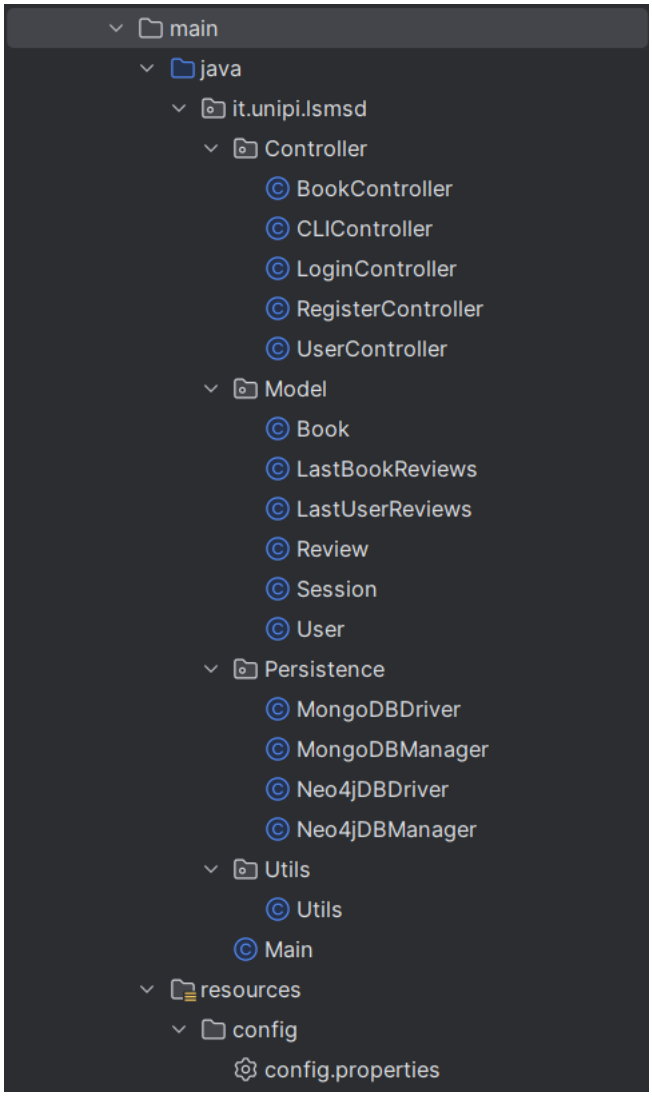


Figure 3: UML Analysis Classes Diagram

3.5 Application structure



We organized the structure of our application following a Model-View-Controller (**MVC**) architecture which is a design pattern commonly used in software development to organize code and separate concerns.
The separation makes the code more maintainable and easier to understand and also enhance code efficiency and scalability.
The MVC pattern consists of three main components: Model, View, and Controller.
This architectural choice was complemented by the efficient management of dependencies using **Maven**.
Maven streamlined the process of handling libraries and external dependencies, contributing to a more structured and organized development environment.

3.5.1 Model

The Model represents the application's data and business logic. It encapsulates the application's state and behavior, providing an abstraction of the underlying data. The classes contained in this package are:

- **Book**: This class contains informations of the book
- **Review**: This class contains information on the reviews
- **User**: This class contains informations of the user
- **Session**: This class contain information about the ongoin session of the logged user
- **LastBookReviews**: This class contains informations about the last review left under a book

- **LastUserReviews**:This class contains informations about the last review left by a user

3.5.2 Controller

The Controller acts as an intermediary between the Model and the user Interface, in our case a CLI (Command Line Interface). It receives user input from the interface, processes it, and updates the Model accordingly. In this is implemented also the View component, responsible for presenting the data to the user and receiving user input. It displays the information provided by the Model and captures user interactions. The classes in this package are:

- **BookController**: this class manages the actions performed on books, such as adding or deleting a review
- **CLIController**: this class manages all the interactions between the user and the applications through the command line interface, displaying all the possibilities and instructions for the users and functioning as a bridge with the Model package.
- **LoginController**: this class manages the login process of the application.
- **RegisterController**: this class manage the register process of the application.
- **UserController**: this class manages the generic user information to display and also all the "social" actions a user can do, such as following other users, authors and more.

3.5.3 Persistence

The Persistence component handles the storage and retrieval of data from the databases. It ensures that the application's data is persisted across sessions. The classes in this package are the following:

- **MongoDBDriver**:Implements methods essential for establishing and maintaining a connection to the MongoDB database.
- **MongoDBManager**:Implements MongoDB's queries. Provides a set of methods for efficient data retrieval and manipulation in MongoDB
- **Neo4jDriver**:Implements methods for establishing and maintaining connectivity to Neo4j.
- **Neo4jManager**:Implements queries tailored for Neo4j. Provide methods for interacting with the Neo4j graph database, for data retrieval and manipulation.

3.5.4 Utils

This package contains utility functions.

- **Utils**: this class contains function for manage the configuration parameters

3.6 Data Models

3.6.1 Document DB

In the context of our application on books, leveraging a document-oriented database offers distinct advantages, particularly in optimizing performance and data retrieval. The decision was made to embed reviews within both book and user documents, aligning with a schema-less approach that accommodates flexible data structures.

3.6.2 Graph DB

We opted for a graph database to efficiently manage the social network features within the application. Our strategy focuses on keeping the graph database lightweight but functional, utilizing it not only for handling social network relationships, and conducting analytics rooted in social network interactions between users and their preferences about authors and genres, but also using it for our recommendation System.

3.6.3 Data among databases

In our application, we made the strategic decision to store certain information redundantly in both the graph and document databases. This dual storage approach is aimed at enhancing overall efficiency. Distributing the load between the graph and document databases helps balance resource utilization, preventing potential bottlenecks and ensuring optimal performance across various application functions. Our storing strategy is as follows:

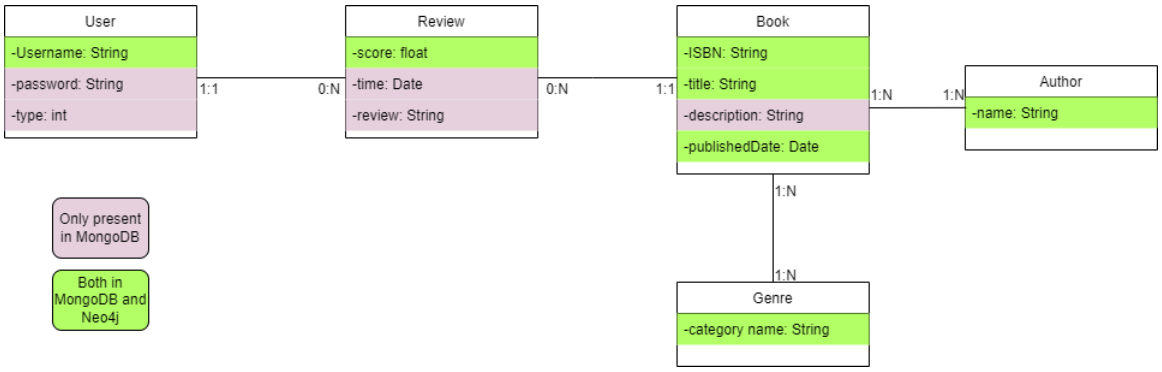


Figure 4: Storing strategy

3.7 Distributed Database Design

According to the Non-Functional Requirements, our system must provide high availability, fast response times and to be tolerant to data lost and single point of failure.
To achieve such results, we orient our application on the **A** (*Availability*), **P** (*Partition Protection*) edge of the *CAP triangle* and we adopt the **Eventual Consistency** paradigm on our dataset. .
To ensure these features we have to sacrifice consistency between the replicas;
offering a high availability of the content, even if an error occurs on the network layer,

means that users may read data which is not always accurate.

3.7.1 Consistency between Replicas

The document database of our application is shared with 3 replicas, one for each machine of the provided cluster.
The consistency between all the replicas is managed using **Eventual Consistency**.
This type of consistency allows the system to continue operating even when some replicas have not yet received the latest updates. In fact, after receiving a write operation we will update one server, and the replicas will be updated in a second moment. In this way writes operation don't keep the server busy for too much time. This means reads and writes can continue, ensuring **high availability** of the service.
This approach supports **Partition Protection** that is guarantee by the presence of replicas in our cluster; if one server is down, we can continue to offer our service by searching the content in the replicas.
Unfortunately, to guarantee these two services we can't ensure that the user will retrieve the most updated content but the replicas will eventually catch up and become consistent.

3.7.2 Consistency between different Databases

The operations which require cross-database consistency management are Add/Remove a User, Add/Remove a Book, Add a Review.
To manage the cross-database consistency we use **MongoDb Transactions**.
By using MongoDB's transactional capabilities, we ensure that all parts of an operation succeed or fail together. This is crucial when operations involve changes that need to reflect in both MongoDB and Neo4j, preserving the integrity of data across systems.
If an operation fails after modifying MongoDB but before Neo4j reflects the changes, the transactional control allows us to roll back changes, ensuring that the databases do not end up in an inconsistent state.

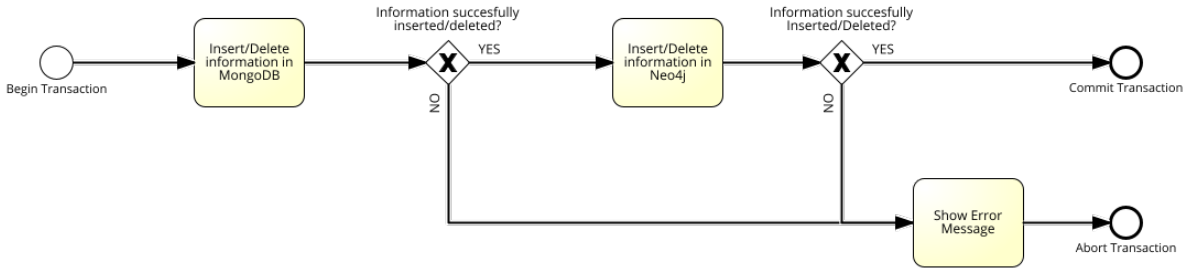


Figure 5: Consistency between MongoDB and Neo4j

3.7.3 Sharding

The possibility to implement a sharding would permit to evenly balance the workload among the nodes, improving users' experience.
The Sharding could be implemented by selecting three different sharding keys, one for each collection of the document database (User Collection, Book Collection and Review collection).
For all collections we use as partitioning method the *Consistent hashing*, because if the number of nodes of the cluster will change, it will be easier to relocate data.
The sharding keys are:

- For the User Collection we choose the attribute "*profileName*" as *sharding key*, which is unique among the users.
- For the Book Collection we choose the attribute "*ISBN*" as *sharding key*, which is unique among the books.
- For the Review Collection we choose the attribute "*_id*", which is automatically generated by MongoDB, as *sharding key*.

4 Document DB Design and Implementation

In the document DB we have 3 collections: **Users**, **Reviews** and **Books**.
One particular aspect of the database is the redundancy of the reviews:
Reviews,in fact are also embedded within both book and user documents (only the lasts 5 for each user/book), emphasizing a denormalized structure to minimize the need for expensive join operations. To enhance performance, the application displays a concise preview of the last five reviews associated with a book or user, offering users quick insights without requiring extensive data retrieval. We put some examples of our collection underneath:

Figure 6: Example of a document of Book collection

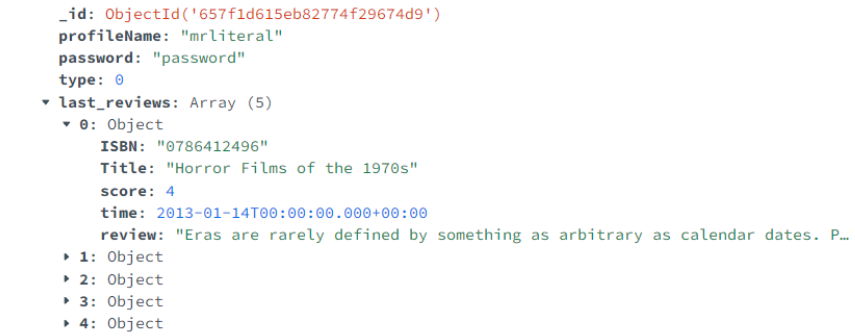


Figure 7: Example of a document of User collection

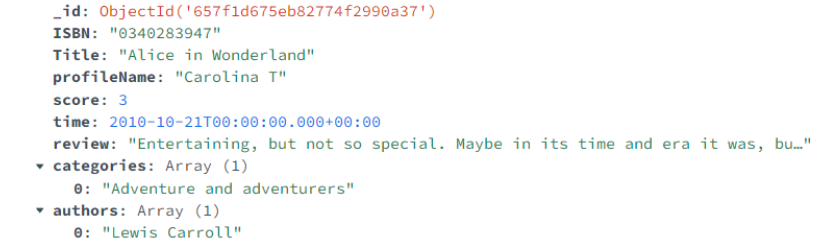


Figure 8: Example of a document of Review collection

4.1 Queries Implementation

in This paragraph we present the CRUD operations and the more complex queries written in Cypher.

4.1.1 CRUD Operations

Create

Operation	Implementation in MongoDB

Create a User	db.Users.insertOne(
	{
	username: "username",
	password: "password",
	Type: userType,
	Last reviews: []
	}
	)

Create a Book	db.BookCollection.insertOne(
	{
	ISBN: "ISBN",
	Title: "Title",
	description: "Description",
	authors: ["author1", "author2"],
	categories: ["category1", "category2"],
	publishedDate: new Date("yourPublishedDate"),
	last_users_review: []
	}
	)

Create a Review	db.ReviewCollection.insertOne(
	{
	ISBN: "ISBN",
	Title: "Title",
	profileName: "ProfileName",
	score: Score,
	time: new Date("ReviewDate"),
	review: "Review",
	categories: ["category1", "category2"],
	authors: ["author1", "author2"]
	}
	)

Delete

Operation	Implementation in MongoDB
Delete a user	db.UserCollection.deleteOne({ profileName: "yourProfileName" }); db.BookCollection.updateMany({ "last_users_review.profileName": "ProfileName" }, { $pull: { last_users_review: { profileName: "ProfileName" } } }); db.ReviewCollection.deleteMany({ profileName: "ProfileName" });

Read

Operation	Implementation in MongoDB

Get user by profilename	db.userCollection.findOne(
	{
	profilename: “ProfileName”
	}
	);

Get book by isbn	db.BookCollection.findOne(
	{
	ISBN: “BookISBN”
	}
	);

Get book by title	db.BookCollection.find(
	{
	title: “BookTitle”
	}
	);

Get review by isbn and profilename	db.ReviewCollection.findOne(
	{
	ISBN: “BookISBN”,
	profilename: “ProfileName”
	}
	);

Update

Operation	Implementation in MongoDB
Update user	db.UserCollection.updateOne({ profileName: "ProfileName" }, { $set: { password: "NewPassword" } })
Update review	var reviewDeleted = "Review deleted by admin because it doesn't follow the politeness"; db.ReviewCollection.updateOne({ ISBN: "yourBookISBN", profileName: "ProfileName" }, { $set: { review: reviewDeleted } }); db.UserCollection.updateOne({ profileName: "ProfileName", "last_reviews.ISBN": "BookISBN" }, { $set: { "last_reviews.$.review": reviewDeleted } }); db.BookCollection.updateOne({ ISBN: “BookISBN", "last_users_review.profileName": "ProfileName" }, { $set: { "last_users_review.$.review": reviewDeleted } });

4.1.2 Analytics

Get most rated Books, choosing one or more genres:
Java:

```
public List<Book> getTopBooks(int numReview,List<String>
   categories,int limit,int skip,ArrayList<Double>scores){
        List<Book> results=new ArrayList<>();
        List<Document>pipeline=new ArrayList<>();
        if(categories!=null&&categories.isEmpty()){
            pipeline.add(new Document("$match",
                    new Document("categories",
                            new Document("$in",categories))));
        }
        pipeline.addAll(Arrays.asList(
                new Document("$group",
                        new Document("_id","$ISBN")
                                .append("averageScore",
                                        new Document("$avg","
                                           $score"))
                                .append("totalReviews",
                                        new Document("$sum",1)))
                                           ,
                new Document("$match",
                        new Document("totalReviews",
                                new Document("$gte",numReview)))
                                   ,
                new Document("$sort",
                        new Document("averageScore",-1)),
                new Document("$project",
                        new Document("ISBN","$_id")
                                .append("averageScore",1)
                                .append("totalReviews",1)
                                .append("_id",0)),
                new Document("$skip",skip),
                new Document("$limit",limit)
        ));
        AggregateIterable<Document> documentAggregateIterable  =
           reviewCollection.aggregate(pipeline);
        for(Document document:documentAggregateIterable){
            results.add(getBookByISBN(document.getString("ISBN")
               ));
            scores.add(document.getDouble("averageScore"));
        }
        return results;
    }
```

MongoDB:

```
    db.reviewCollection.aggregate([
    {
        $match: {
            categories: { $in: categories }
        }
    },
    {
        $group: {
            _id: "$ISBN",
            averageScore: { $avg: "$score" },
            totalReviews: { $sum: 1 }
        }
    },
    {
        $match: {
            totalReviews: { $gte: numReview }
        }
    },
    {
        $sort: {
            averageScore: -1
        }
    },
    {
        $project: {
            ISBN: "$_id",
            averageScore: 1,
            totalReviews: 1,
            _id: 0
        }
    },
    {
        $skip: skip
    },
    {
        $limit: limit
    }
]);
```

Explanation of each stage:

- **match**: Filters documents based on the condition that the "categories" field is an array containing at least one element.
- **group**: Groups documents by the "ISBN" field. Calculates the average score and the total number of reviews for each book.
- **match**: Filters books based on the condition that the total number of reviews is greater than or equal to the specified numReview.

- **sort**: Sorts the results based on the average score in descending order.
- **project**: Projects the desired fields, renaming "_id" to "ISBN" and excluding "_id".
- **skip**: Skips a specified number of results.
- **limit**: Limits the total number of returned results.

Get most versatile Users:
(Users that reviews the greatest number of different genres)
Java:

```
        public List<User> getMostVersatileUsers(int skip, int
           limit){
        List<Document> pipeline=Arrays.asList(
                new Document("$group",
                        new Document("_id","$profileName")
                                .append("uniqueCategories",
                                        new Document("$addToSet"
                                           ,"$categories"))),
                new Document("$project",
                        new Document("profileName","$_id")
                                .append("uniqueCategories","
                                   $uniqueCategories")
                                .append("numUniqueCategories",
                                        new Document("$size","
                                           $uniqueCategories")))
                                        ,
                new Document("$sort",
                        new Document("numUniqueCategories",-1)),
                new Document("$skip",skip),
                new Document("$limit",limit)
        );
        AggregateIterable<Document> documentAggregateIterable=
           reviewCollection.aggregate(pipeline);
        ArrayList<User> result=new ArrayList<>();
        for(Document document:documentAggregateIterable){
            result.add(getUserByProfileName(document.getString("
               profileName")));
        }
        return result;
    }
```

MongoDB:

```
db.reviewCollection.aggregate([
    {
        $group: {
            _id: "$profileName",
            uniqueCategories: { $addToSet: "$categories" }
        }
    },
    {
        $project: {
            profileName: "$_id",
            uniqueCategories: 1,
            numUniqueCategories: { $size: "$uniqueCategories" }
        }
    },
    {
        $sort: { numUniqueCategories: -1 }
    },
    {
        $skip: skip
    },
    {
        $limit: limit
    }
])
```

Explanation of each stage:

- **match**: Filters reviews based on the "time" field within the specified date range.
- **group**: Groups reviews based on the "profileName" field. Calculates the count of reviews for each user profile.
- **sort**: Sorts the results based on the count of reviews in descending order.
- **project**: Reformat the result of grouping. Renames the "_id" field to "profileName". Retains only the necessary fields for output.
- **skip**: Skips a specified number of results.
- **limit**: Limits the total number of returned results.

Get most published genres:

Java:

```
    public List<String> getTopCategoriesOfNumOfBookPublished(int
        skip,int limit){
        List<Document> pipeline=Arrays.asList(
                new Document("$unwind","$categories"),
                new Document("$group",
                        new Document("_id","$categories")
                                .append("count",
                                        new Document("$sum",1)))
                                           ,
                new Document("$sort",
                        new Document("count",-1)),
                new Document("$skip",skip),
                new Document("$limit",limit)
        );
        AggregateIterable<Document> documentAggregateIterable=
           bookCollection.aggregate(pipeline);
        List<String> results=new ArrayList<>();
        for(Document document:documentAggregateIterable){
            results.add(document.getString("_id"));
        }
        return results;

    }
```

MongoDB:

```
db.bookCollection.aggregate([
    {
        $unwind: "$categories"
    },
    {
        $group: {
            _id: "$categories",
            count: { $sum: 1 }
        }
    },
    {
        $sort: { count: -1 }
    },
    {
        $skip: skip
    },
    {
        $limit: limit
    }
])
```

Explanation of each stage:

- **unwind**: Deconstructs the "categories" array, creating a new document for each category.
- **group**: Groups the documents based on the "categories" field. Calculates the count of books for each category.
- **sort**: Sorts the results based on the count of books in descending order.
- **skip**: Skips a specified number of results.
- **limit**: Limits the total number of returned results.

Get most active Users in a specific date range:
Java:

```
public List<String> getMostActiveUsers(String startDate,
   String endDate,int skip,int limit,ArrayList<Integer>
   counts){

List<Document> pipeline;

if (startDate != null && endDate != null && !startDate.
   isEmpty() && !endDate.isEmpty()) {
    try {
        SimpleDateFormat dateFormat = new
           SimpleDateFormat("yyyy-MM-dd");
        Date start = dateFormat.parse(startDate);
        Date end = dateFormat.parse(endDate);
        pipeline=Arrays.asList(
                new Document("$match",
                        new Document("time",
                                new Document("$gte",
                                   start)
                                        .append("$lte",
                                           end))),
                new Document("$group",
                        new Document("_id","$profileName
                           ")
                                .append("count",
                                        new Document("
                                           $sum",1))),
                new Document("$sort",
                        new Document("count",-1)),
                new Document("$project",
                        new Document("_id",0)
                                .append("profileName","
                                   $_id")
```

```
                                        .append("reviewCount","
                                            $count")),
                        new Document("$skip",skip),
                        new Document("$limit",limit)
                );
            } catch (ParseException e) {
                System.out.println("problems␣with␣parse␣the␣date
                    ␣in␣getMostActiveUsers");
                e.printStackTrace();
                return null;
            }
            AggregateIterable<Document> results=reviewCollection
                .aggregate(pipeline);
            List<String> topReviewersName=new ArrayList<>();
            for (Document document:results){
                String profileName=document.getString("
                    profileName");
                topReviewersName.add(profileName);
                counts.add(document.getInteger("reviewCount"));
            }
            return topReviewersName;
        }else{
            System.out.println("enter␣a␣good␣start␣and␣end␣date"
                );
            return null;
        }
    }
```

MongoDB:

```
    db.reviewCollection.aggregate([
    {
        $match: {
            time: {
                $gte: ISODate("startDate"),
                $lte: ISODate("endDate")
            }
        }
    },
    {
        $group: {
            _id: "$profileName",
            count: { $sum: 1 }
        }
    },
    {
        $sort: { count: -1 }
    },
    {
        $project: {
            _id: 0,
```

```
            profileName: "$_id",
            reviewCount: "$count"
        }
    },
    {
        $skip: skip
    },
    {
        $limit: limit
    }
])
```

Explanation of each stage:

- **match**: Filters documents based on the specified time range.
- **group**: Groups the documents based on the "profileName" field. Calculates the count of reviews for each user.
- **sort**: Sorts the results based on the count of reviews in descending order.
- **project**: Projects the desired fields, renaming "_id" to "profileName" and "count" to "reviewCount".
- **skip**: Skips a specified number of results.
- **limit**: Limits the total number of returned results.

Get most rated Authors:
(Based on the total rating of their books)
Java:

```
        public List<String> getMostRatedAuthors(int skip,int
           limit,int numReviews,ArrayList<Double>score){
        List<String> results= new ArrayList<>();
        List<Document> pipeline=Arrays.asList(
                new Document("$unwind","$authors"),
                new Document("$group",
                        new Document("_id","$authors")
                                .append("totalReviews",
                                        new Document("$sum",1))
                                .append("avgScore",
                                        new Document("$avg","
                                           $score"))),
                new Document("$match",
                        new Document("totalReviews",
                                new Document("$gte",numReviews))
                                   ),
                new Document("$sort",
                        new Document("avgScore",-1)),
                new Document("$project",
                        new Document("author","$_id")
                                .append("totalReviews",1)
                                .append("avgScore",1)
                                .append("_id",0)),
                new Document("$skip",skip),
                new Document("$limit",limit)
        );
        AggregateIterable<Document> documentAggregateIterable=
           reviewCollection.aggregate(pipeline);
        for (Document document:documentAggregateIterable){
            results.add(document.getString("author"));
            score.add(document.getDouble("avgScore"));
        }
        return results;
    }
```

Mongo:

```
    db.reviewCollection.aggregate([
    {
        $unwind: "$authors"
    },
    {
        $group: {
            _id: "$authors",
            totalReviews: { $sum: 1 },
            avgScore: { $avg: "$score" }
        }
    },
    {
        $match: { totalReviews: { $gte: numReviews } }
    },
    {
        $sort: { avgScore: -1 }
    },
    {
        $project: {
            author: "$_id",
            totalReviews: 1,
            avgScore: 1,
            _id: 0
        }
    },
    {
        $skip: skip
    },
    {
        $limit: limit
    }
])
```

Explanation of each stage:

- **unwind**: Deconstructs the "authors" array, creating a separate document for each author.
- **group**: Groups the documents based on the "authors" field. Calculates the total number of reviews and the average score for each author.
- **match**: Filters authors based on the specified minimum number of reviews.
- **sort**: Sorts the results based on the average score in descending order.
- **project**: Projects the desired fields, renaming "_id" to "author", excluding "_id".
- **skip**: Skips a specified number of results.
- **limit**: Limits the total number of returned results.

Get *Bad* Users:
(Users with the greatest number of deleted reviews)
Java:

```
public List<User>getBadUsers(int skip,int limit){
        List<User> results=new ArrayList<>();
        ArrayList<Document>pipeline=new ArrayList<>();
        pipeline.add(new Document("$match",
                new Document("review",reviewDeleted)));
        pipeline.add(new Document("$group",
                new Document("_id","profileName").append("count"
                   ,
                        new Document("$sum",1))));
        pipeline.add(new Document("$sort",
                new Document("count",-1)));
        pipeline.add(new Document("$project",
                new Document("profileName","$_id").append("count
                   ",1).append("_id",0)));
        pipeline.add(new Document("$skip",skip));
        pipeline.add(new Document("$limit",limit));
        Iterable<Document>result=reviewCollection.aggregate(
           pipeline);
        for(Document document:result){
            User user=getUserByProfileName(document.getString("
               profileName"));
            if(user!=null){
                results.add(user);
            }
        }
        return results;
    }
```

MongoDB:

```
    db.reviewCollection.aggregate([
    {
        $match: { review: reviewDeleted }
    },
    {
        $group: {
            _id: "$profileName",
            count: { $sum: 1 }
        }
    },
    {
        $sort: { count: -1 }
    },
    {
        $project: {
            profileName: "$_id",
            count: 1,
            _id: 0
        }
    },
    {
        $skip: skip
    },
    {
        $limit: limit
    }
])
```

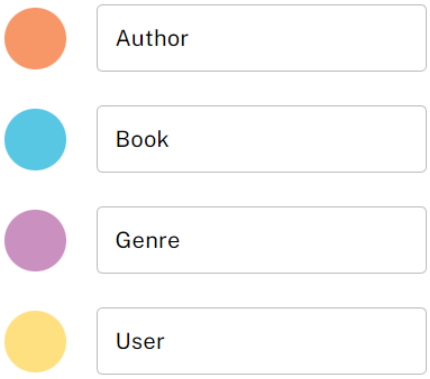
Explanation of each stage:

- **match**: Filters documents based on the condition that the "review" field is equal to "reviewDeleted".
- **group**: Groups documents by the "profileName" field and Calculates the count of reviews for each profileName.
- **sort**: Sorts the results based on the count of reviews in descending order.
- **project**: Projects the desired fields, renaming "_id" to "profileName" and excluding "_id".
- **skip**: Skips a specified number of results.
- **limit**: Limits the total number of returned results.

5 Graph DB Design and Implementation

In Neo4j database there are the following entities: **Book**, **User**, **Author** and **Genre**. Every type of node contains only a string containing the name of the node (Username, Name of the Author, Name of the Genre, Title of the book), the Book node contains also the ISBN code because is possible that two books with different ISBN have the same Title.
There are also the following relationships between entities:

- USER – REVIEWS → BOOK: Each user can reviews one or more books, the relation contains the score of the review, it is useful to showing suggestions based on the most liked books of your friends.
- USER – LIKES → AUTHOR: Each user can like one or more Authors; it is useful to showing suggestions based the books of the favourite authors.
- USER – FOLLOWS → USER: Each user can follow one or more users; it is useful to showing suggestions of books and users to follow, but also to see followers and following people.
- USER – PREFERS → GENRE: Each user can prefer only one genre (if he chooses to like another genre, the first relationship will be eliminated; it is useful to showing suggestions based on the preferred genre.
- AUTHOR – WROTE → BOOK: Each Author can write one or more book and one book may have more than one authors; the relation contains the publication date of the book and it is useful to showing suggestions of the most recent published books by the favorite authors based.
- BOOK – BELONGS_TO → GENRE: Each book can belongs to one or more genre; it is useful for showing suggestions based on the preferred genre.



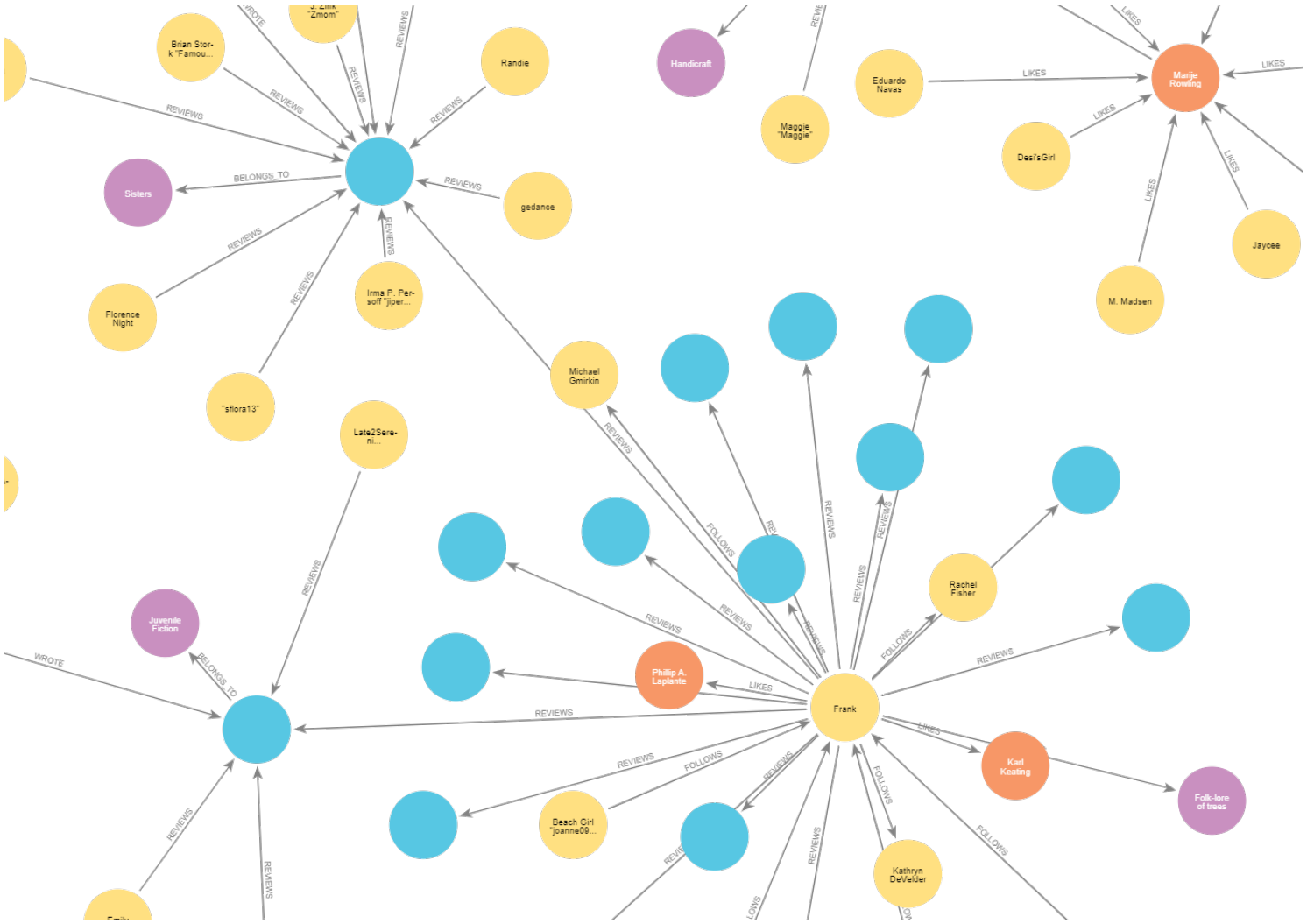


Figure 9: Example of our graph

5.1 Queries Implementation

in This paragraph we present the CRUD operations and the more complex querys written in Cypher.

5.1.1 CRUD Operations

Create

Operation	Implementation in Cypher
Create User	**CREATE** (u:User {name:$name})
Create Book	**MERGE** (b:Book {ISBN: $isbn}) SET b.title = $title
Create User – FOLLOW → User	**MATCH** (a:User {name: $usernameA}), (b:User {name: $usernameB}) **MERGE** (a)-[:FOLLOWS]->(b)
Create User – LIKES → Author	**MERGE** (u:User {name: $username}) **MERGE** (a:Author {name: $authorName}) **MERGE** (u)-[:LIKES]->(a)
Create User – PREFERS → Genre (In our Application one User can select at most only 1 genre, so this query is preceded by a check and in case by a delete)	**MERGE** (u:User {name: $username}) **MERGE** (g:Genre {name: $newGenre}) **MERGE** (u)-[:PREFERS]->(g)
Create User – REVIEWS → Book	**MATCH** (u:User {name: $profileName}), (b:Book {ISBN: $isbn, title: $title}) **MERGE** (u)-[r:REVIEWS]->(b) **SET** r.score = $score
Create Book – BELONGS_TO → Genre	**MERGE** (b:Book {ISBN: $isbn}) **MERGE** (g:Genre {name: $newGenre}) **MERGE** (b)-[:BELONGS_TO]->(g)
Create Author – WROTE → Book	**MERGE** (b:Book {ISBN: $isbn}) **MERGE** (a:Author {name: $author}) **MERGE** (a)-[w:WROTE]->(b) **SET** w.publicationDate = $publicationDate

Read

Operation	Implementation in Cypher
Get number of User's followers	**MATCH** (:User {name: $username})<-[r:FOLLOWS]-() **RETURN COUNT**(DISTINCT r) AS numFollowers
Get list of User's followers	**MATCH** (u:User {name: $username})<-[r:FOLLOWS]-(f:User) **RETURN DISTINCT** f.name AS userName
Get number of User's followings	**MATCH** (:User {name: $username})-[r:FOLLOWS]->() **RETURN COUNT**(DISTINCT r) AS numFollowing
Get list of User's followings	**MATCH** (u:User {name: $username})-[r:FOLLOWS]->(f:User) **RETURN DISTINCT** f.name AS userName
Check if the user has an existing genre preference	**MATCH** (u:User {name: $username})-[:PREFERS]->(oldGenre:Genre) **RETURN** oldGenre.name AS oldGenreName

Delete

Operation	Implementation in Cypher
Delete User	**MATCH** (u:User {name: $name}) DETACH DELETE u
Delete User – PREFERS → Genre	**MATCH** (u:User {name: $username})-[r:PREFERS]->(oldGenre:Genre {name: $oldGenreName}) **DELETE** r
Delete User – LIKES → Author	**MATCH** (u:User {name: $username})-[r:LIKES]->(a:Author {name: $authorName}) **DELETE** r
Delete User – FOLLOWS → User	**MATCH** (a:User {name: $usernameA})-[r:FOLLOWS]->(b:User {name: $usernameB}) **DELETE** r

5.1.2 Recommendation System and other queries

Get most popular users:

```
MATCH (follower:User)-[:FOLLOWS]->(u:User)
WITH u, COUNT(DISTINCT follower) AS numFollowers
ORDER BY numFollowers DESC LIMIT $limit
RETURN DISTINCT u.name AS mostFollowed
```

The query returns a list of the most followed users, is possible to specify a limit to the number of element of the list.

Get recommendations of Users:

```
MATCH (me:User {name: $userName})-[:FOLLOWS]->(u:User)-[:FOLLOWS
   ]->(u2:User)
WHERE NOT EXISTS((me)-[:FOLLOWS]->(u2))
WITH u2, COUNT(DISTINCT u) AS numFollowers
ORDER BY numFollowers DESC
LIMIT $limit
RETURN DISTINCT u2.name AS recommendedUserName
```

The query returns a list of suggested Users to follow.
In particular, recommends users who are followed by the most followers of the specified user.

Get recommendations of books based on liked Authors:

```
MATCH (u:User {name: $userName})-[:LIKES]->(a:Author)-[w:WROTE
   ]->(b:Book)
WHERE NOT EXISTS((u)-[:REVIEWS]->(b))
WITH b, a, w
ORDER BY w.publicationDate DESC
RETURN DISTINCT b.ISBN AS ISBN
LIMIT $limit
```

The query returns a list of books written by the liked Authors of the User, giving priority to the latest books released.

Get recommendations of books based on preferred Genre:

```
MATCH (u:User)-[:REVIEWS]->(b:Book)-[:BELONGS_TO]->(g:Genre)
WITH b, g, COUNT(DISTINCT u) AS numReviews
WHERE g.name = $preferredGenre AND numReviews >
   $numReviewsThreshold
RETURN DISTINCT b.ISBN AS ISBN, b.title AS title, numReviews
ORDER BY numReviews DESC
LIMIT $limit
```

The query returns a list of books that belongs to the preferred genre of the User.

Get recommendations of books based on followed Users:

```
MATCH (u:User {name: $userProfileName})-[:FOLLOWS]->(f:User)-[r:
   REVIEWS]->(b:Book)
WHERE r.score>=3 AND NOT EXISTS((u)-[:REVIEWS]->(b))
WITH b, COUNT(DISTINCT r) AS numComments
ORDER BY numComments DESC
LIMIT $limit
RETURN DISTINCT b.ISBN AS ISBN, b.title AS title
```

The query returns a list of books based on the most commented books by friends, counting only the ones with score greater than 3 or equal.

Get recommendations of books based on followed Users and preferred Genre:

```
 MATCH (u:User {name: $userName})-[:FOLLOWS]->(f:User)-[r:
    REVIEWS]->(b:Book)-[:BELONGS_TO]->(g:Genre)
WHERE r.score>=3 AND NOT EXISTS((u)-[:REVIEWS]->(b))
WITH b, g, COUNT(DISTINCT r) AS numComments
WHERE g.name = $preferredGenre
WITH b,g,numComments
ORDER BY numComments DESC
LIMIT $limit
RETURN DISTINCT b.ISBN AS ISBN, b.title AS title
```

The query is a combination of the previews two, returning a list of books based on the most commented books by friends, counting only the ones with score greater than 3 or equal that belongs to the preferred genre of the user.

6 Test and Statistical Analysis

The table below illustrates the frequency of usage for the primary queries in our application, focusing on the search functionality for books:

Query	Frequency
Search Books by Title	High
Search Books by Author	High
Search Books by Published Date	Medium
Search Books by Category	Medium
Search Books by ISBN Code	High
Search Users by Profile Name	Medium
Search Review by ISBN and Profile Name	Low
Search Review by Time	Low

Table 1: Queries frequency analysis

Please note that while the table outlines specific search queries, in the application's code, the first four queries are consolidated into a single parametric search. This unified query is utilized within the application, enabling users to execute these searches individually by setting the corresponding parameters. The adaptation enhances efficiency in query execution while maintaining a seamless user experience in searching for books.

6.1 Index Analysis

In our pursuit of performance optimization, a key strategy we've embraced is the deliberate application of indexes. Indexes function as practical signposts within the database, facilitating swift data retrieval by reducing the time required to pinpoint specific information. Our decision to incorporate indexes is particularly rooted in the observation that our queries, as previously mentioned, lean heavily towards reading operations.In this section we explore how their implementation contributes to improved overall performance in our application. To avoid impacting write operations, deliberate decisions were made to refrain from adding indexes to the reviewsCollection in our project. By not introducing indexes to this specific collection, we aimed to maintain the efficiency of write processes and prevent unnecessary overhead.

6.1.1 Index ProfileName

By leveraging indexes for profile names, the search performance experiences a significant enhancement, resulting in improved query efficiency.

Query	Results without index	Results with index
SearchUserByProfileName	executionTimeMillis: 33 totalKeysExamined: 0 totalDocsExamined: 117096	executionTimeMillis: 2 totalKeysExamined: 1 totalDocsExamined: 1

Figure 10: Username index performances

the size is currently around 2.4 MB and is expected to increase proportionally with the growth of the User database.

6.1.2 Index ISBN

Query	Results without index	Results with index
SearchBookByISBN	executionTimeMillis: 22 totalKeysExamined: 0 totalDocsExamined: 52217	executionTimeMillis: 1 totalKeysExamined: 1 totalDocsExamined: 1

Figure 11: ISBN index performances

the size is 794.6KB

6.1.3 Index Titles

In our performance analysis focusing on the query "Search Books by Title," which retrieves all books matching a specified keyword, we observed a notable statistical improvement.

Query	Results without index	Results with index
SearchBookByTitle	executionTimeMillis: 25 totalKeysExamined: 0 totalDocsExamined: 52216	executionTimeMillis: 1 totalKeysExamined: 1 totalDocsExamined: 1

Figure 12: Titles index performances

the size is currently around 2.9 MB and is expected to increase proportionally with the growth of the book database. Despite the potential increase in size, the statistical outcomes were highly favorable, especially considering the frequency of this query. Given its significance and frequent occurrence, we have decided to incorporate the index into the application to optimize the performance of the "Search Books by Title" query.

6.1.4 Index Publication Date

The table below presents the outcomes of the "Search Books by Published Date" query, designed to retrieve all books falling within a specified date range.

Query	Results without index	Results with index
SearchBookByPubDate(specific date)	executionTimeMillis: 33 totalKeysExamined: 0 totalDocsExamined: 52216	executionTimeMillis: 3 totalKeysExamined: 1824 totalDocsExamined: 1824

Figure 13: Publication date index performances

Despite the query not being highly frequent, the associated index is relatively lightweight at 569.3 KB and exhibits a slower growth rate compared to previous indexes. More importantly, the performance improvement achieved is substantial. Considering these factors, we have chosen to include the "published" index in the database, enhancing the efficiency of the "Search Books by Published Date" query.

7 User Manual

This section is a brief User Manual for you to navigate our application. Our application can be used by a simple command line interface.
The available commands vary among different user types, including Admins, Normal Users, and Unregistered Users.
When you start the application you find yourself in the first command menu, which shows the option as a Unregistered/Unlogged user:

```
*****************************************
*            Welcome to BOOK-HUB          *
*   Your Ultimate Destination for Reading  *
*                                          *
* Discover, Explore, and Immerse Yourself  *
*    in the Enchanting World of Books      *
*                                          *
*****************************************
Hello, Reader! Ready to dive into the world of books?
Use the menu options to explore different features.
Enter the corresponding number to make your choice.

1-Login
2-Sign Up
3-Search for Books
4-Exit
------------
Please enter your choice:
```

As shown, the CLI is guided step by step and it's easy to understand.

After the registration or the login, users are directed to one of two distinct menus, depending on whether they are a normal user or an administrator:

```
Please enter your choice: 1
username:LiuChang
password:password
Welcome, Dear admin:LiuChang!
1-Show profile
2-Find User
3-Change Password
4-Find Book
5-Delete Review
6-Add Book
7-Ban User
8-User Statistics
9-Book Statistics
10-LogOut
------------
Enter your choice:
```

Figure 14: Admin menu

```
Please enter your choice: 1
username:A Customer
password:password
Welcome, Dear user:A Customer!
0-Show following and/or Followers
1-Search book
2-Search user
3-Show profile
4-Add review
5-Change Password
6-Add Your Preferred genre
7-Follow an Author
8-Follow a User
9-Unfollow an Author
10-Unfollow a User
11-Recommendation of Books Based On Friend
12-Recommendation of Books Based On Friend and Preferred Genre
13-Recommendation of Books Based On The Preferred Genre
14-Recommendation of Books Based On Best Authors
15-Recommendation of Users
16-LogOut
------------
Enter your choice:
```

Figure 15: Normal User menu

As evident from the provided images, the available options vary significantly between the two user types, each offering unique actions. Normal users can leverage our recommendation system by selecting options 11 to 15, obtaining new book suggestions based on various parameters. They have the ability to review books, customize their profiles with preferred genres, and access all social features.

On the contrary, administrators possess extensive capabilities, including access to comprehensive statistics on users and books. Additionally, administrators can actively moderate the application by removing reviews and imposing bans on users.

When you exit our application you receive this message:

```
Please enter your choice: 4
*****************************************
*          Thank you for visiting       *
*               BOOK-HUB                *
*     We hope you had an amazing time    *
*      exploring the world of books!     *
*                                        *
*         Keep reading and come back     *
*              soon. Goodbye!            *
*****************************************
Connection closed ...
```