UNIVERSITÀ DI PISA

Master's degree in Artificial Intelligence and Data Engineering

Large-Scale and Multi-Structured Databases project

PaperRater

Academic year 2021-2022

Edoardo Ruffoli, Tommaso Baldi, Francesco Hudema

Github link: https://github.com/edoardoruffoli/PaperRater

Table of contents

Introduction

PaperRater is a Java application which allows users to search, rate and comment scientific papers retrieved from different sources. The application has two main purposes:

- Merge in a unique database, papers published in various open access archives providing fast and efficient ways to search papers.
- Realizing a social network, allowing users to interact and express their opinions about papers. Users can also create *Reading Lists* in which they can save papers.

The application is composed by two main programs:

- PaperRaterApp, the application that is thought to be publicly distributed to real users.
- DB_Updater, a command line application that has the task of initializing and keeping updated the database of PaperRaterApp.

Dataset

Papers

The first component of the application database are the papers. In order to respect the variety constraint we downloaded papers information from two different sources: the first and main source is arXiv.org an open-access archive that contains almost 2 million scholarly articles in scientific fields; the second source is viXra.org, an e-print archive set up as an alternative to arXiv.org that contains almost 40K scientific papers.

We obtained arXiv papers metadata using the Python wrapper for the arXiv official API (https://github.com/lukasschwab/arxiv.py) while viXra papers metadata were obtained by performing web page scraping using *Beautiful Soup*, a Python library that it is used for parsing HTML documents. For our scopes we download only 66.8MB of papers data in total (50k entities), 36MB from arXiv and 30MB from viXra.
Then we merged both the metadata and we applied some pre-processing:
We removed possible duplicates using title as unique key.
arXiv and viXra have almost the same paper categories but with different names, for example astrophysics in arXiv is “astro-ph” while in viXra is “Astrophysics” so we adapted the different name formats of every category.

User, Reading Lists and Social Network Relationship

We populated our database using a Python script that randomly generates 1000 users obtained with *randomuser* API, it also generates a random number of comments under every paper previously downloaded and a random number of Reading Lists for every user. We also populated the social network part of the application by adding Like and Follows relations between generated users, papers and reading lists. The initial number of relations is around 30k.

Requirements

Functional Requirements

This section describes the requirements that the application provides.

Unregistered User:

- Unregistered User can register a Registered User account on the platform

Registered User:

- Registered User can **search** for *Papers*:
  - by title
  - by author
  - by category
  - by publication date
- Registered User can **manage** *Reading Lists*:
  - Registered User can **create** a new *Reading List:*
  - Registered User can **add** and **remove** *Papers* from their *Reading Lists*
  - Registered User can **delete** their *Reading Lists*
  - Registered User can **follow** *Reading Lists* made by other Users
- Registered User can **search** other *Users* by username
- Registered User can **search** *Reading Lists* by title
- Registered User can **follow** *Users*
- Registered User can **browse** suggestions about *Papers*, *Reading Lists* and *Users*
- Registered User can **rate** *Papers*
- Registered User can **comment** *Papers*
- Registered User can **log out**

Moderators:

- Moderators can do all the operations that a Registered User can do
- Moderators can **remove** comment from a *Paper*
- Moderators can **browse** all comments between two dates

Admins:

- Admins can do all the operations that a Moderator can do
- Admins can **delete** Users

- Admins can **elect** or **dismiss** moderators
- Admins can **browse** bad users (users whose comments have been removed)
- Admins can compute papers analytics

DB Updater

- DB Updater can **add** *Papers* to the database

Non - Functional Requirements

- The application needs to be highly available and always online.
- The system needs to be tolerant to data lost and to single point of failure.
- The application needs to provide fast response search results to improve the user experience.
- Flexibility is needed for manage different sources id of the papers.
- The application needs to be user-friendly so a GUI must be provided.

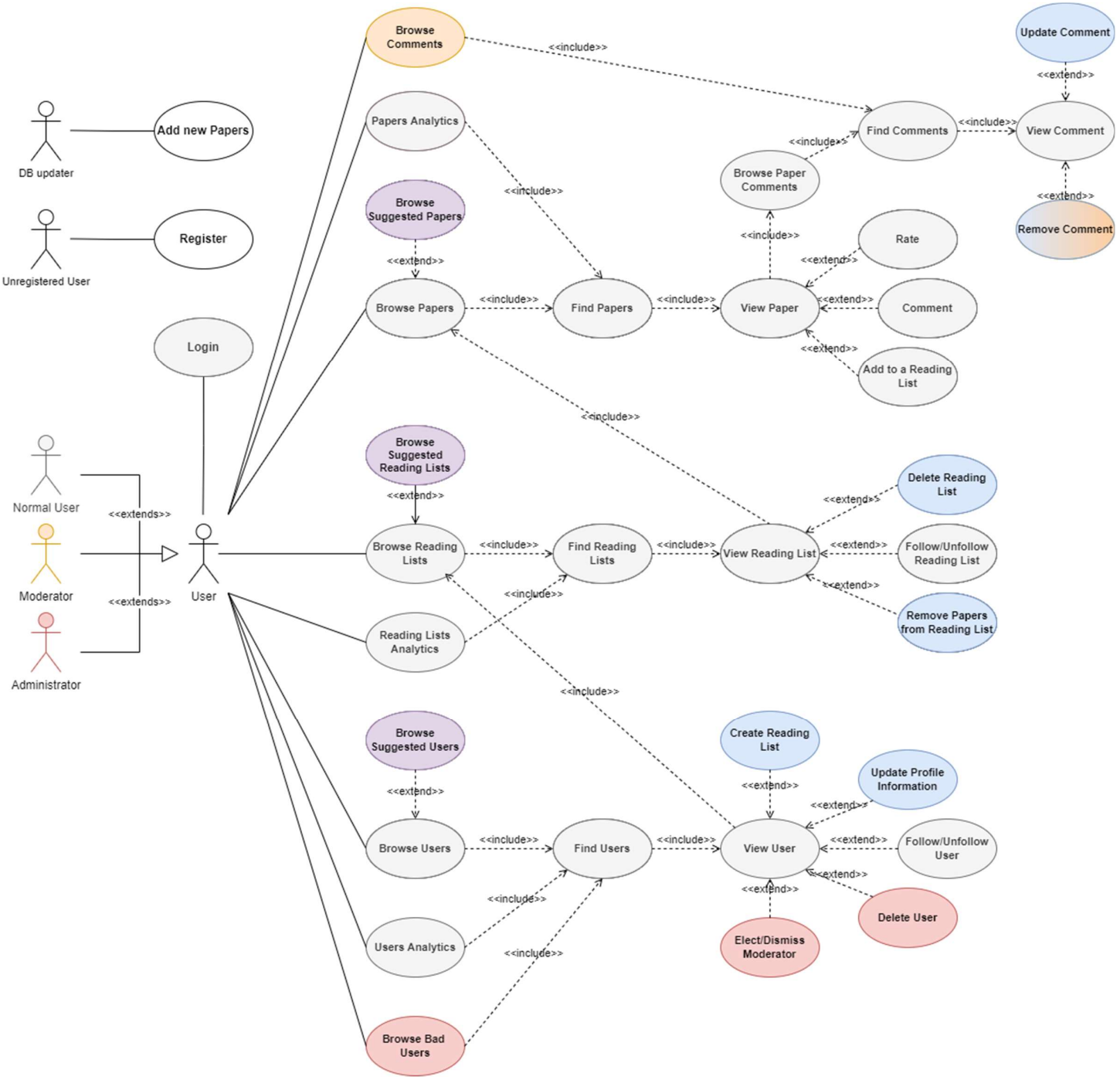
Specifications

Main Actors

The main actors of the application are five:

- *Unregistered User*: User that is not registered on the application, in order to access he must sign-up.
- *Registered User*: User that is registered, he can access application by logging-in.
- *Moderator*: Special User that makes sure that the rules of an internet discussion are not broken, for example by removing any threatening or offensive comments.
- *Administrator*: User with highest level of privilege, he can elect or dismiss moderators, he can delete users. Administrators
- *Database Updater*: Only present on the server side of the application, it imports new papers and add them to the database.

Use Case Diagram



Legend:

- The circles in grey describe actions available to normal Users, Moderators and Administrators.
- The circles in orange describe actions available to Moderators and Administrators.
- The circles in red describe actions available to Administrators only.
- The circles in blue describe actions available only to the owner of the entity. The *Create Reading List* action is available only in the personal page of a user.
- The circles in purple specify graph database typical actions.

UML Class Diagram

There are five main entities: Paper, User, Reading List, Comment. In the diagram, User is a generalization of the three main actors (Registered User, Moderator, Administrator) of the use case diagram, each actor can perform in addition to its own action, the same action of User.

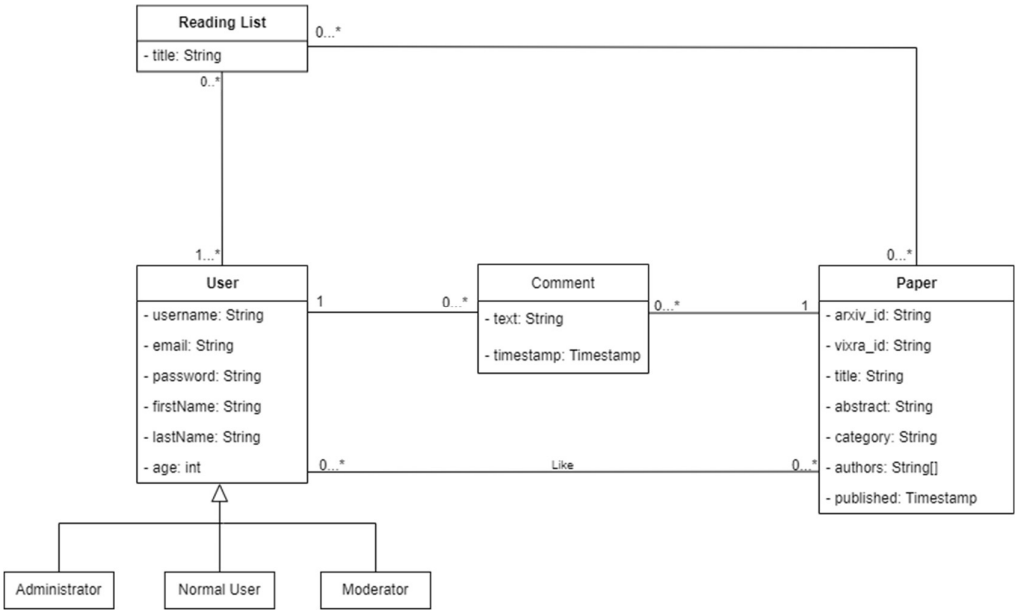


FIGURE 1 UML ANALYSIS CLASSES DIAGRAM

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

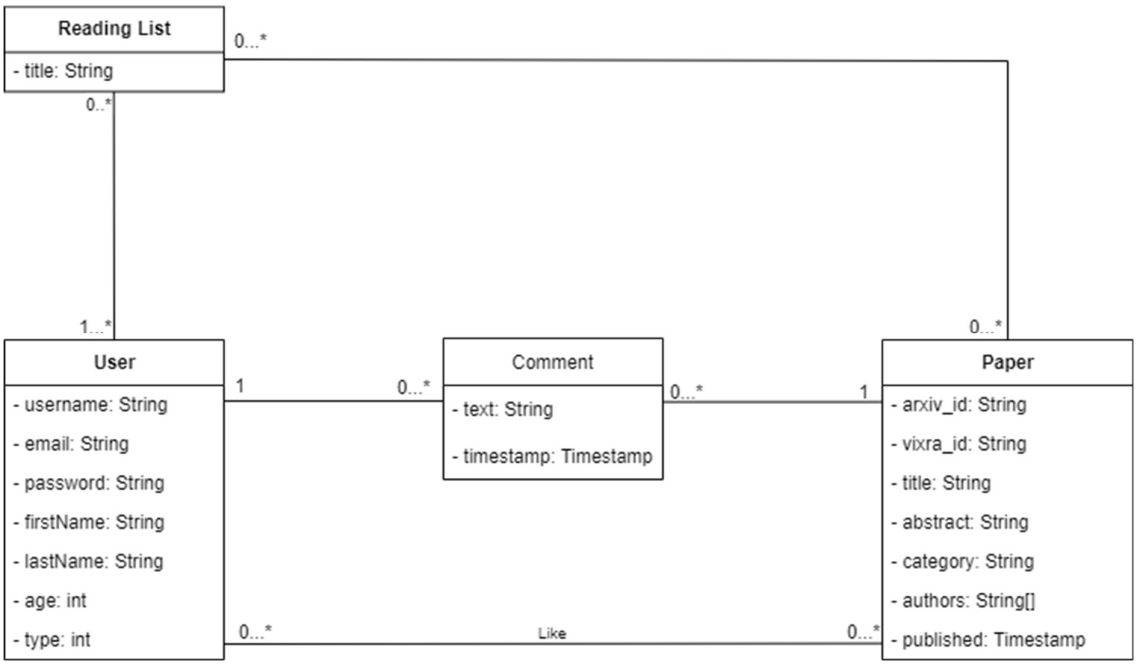


FIGURE 2 UML ANALYSIS CLASSES DIAGRAM

User:
The class implements the main actor of application.
Attributes:

- username: Username of the user
- email: Email of the user
- password: Password of the user
- firstName: First name of the user
- lastName: Last name of the user
- age: Age of the user
- type: Identifies the user's role (0: User, 1: Moderator, 2: Administrator)

Paper:
The class implements the paper entity.
Attributes:

- arxiv_id: Id of Arxiv paper (if the entity is a paper published on Arxiv)
- vixra_id: Id of Vixra paper (if the entity is a paper published on Vixra)
- title: Title of the paper
- abstract: Abstract of the paper
- category: Category to which the paper belongs
- authors: Authors of the paper
- published: Publication date

Reading List:
The class implements the list of paper that can create ad user.
Attributes:

- title: Name of the list

Comment:
The class implements the comment posted by user.
Attributes:

- text: Text of the comment
- timestamp: Creation timestamp of the comment

Architectural Design

Software Architecture

The application was implemented as *client – server architecture,* with middleware implemented on the client side.

Client Side

Client's features can be divided in three parts:

- *Front-end:* this part consists of a graphical interface developed in JavaFX and FXML files. User can use the GUI to interact with the application and managed by controllers used to handle events and views of the informations.
- Middleware: the duty of this part is to handle the connections and communications with the servers, both MongoDB and Neo4j.

DB_Updater Side

The paperRaterUpdater is an application composed by six python scripts:

- *arXivMetadataScraper.py*, script that downloads all the papers from arXiv website published up to start_date argument and saves them in .json format.
- *viXraMetadataScraper.py*, script that downloads all the papers from viXra website published up to start_date argument and saves them in .json format.
- *categoriesRenamer.py,* script that merges in a unique .json the papers downloaded with the previous two scripts, it renames all the categories in order to uniform them.
- *getPapers.py,* script that returns the complete papers dataset in .json format.
- *getUsers.py,* script that returns the complete users dataset in .json format.
- *db_updater.py,* command line Python application used to do create and update the databases.

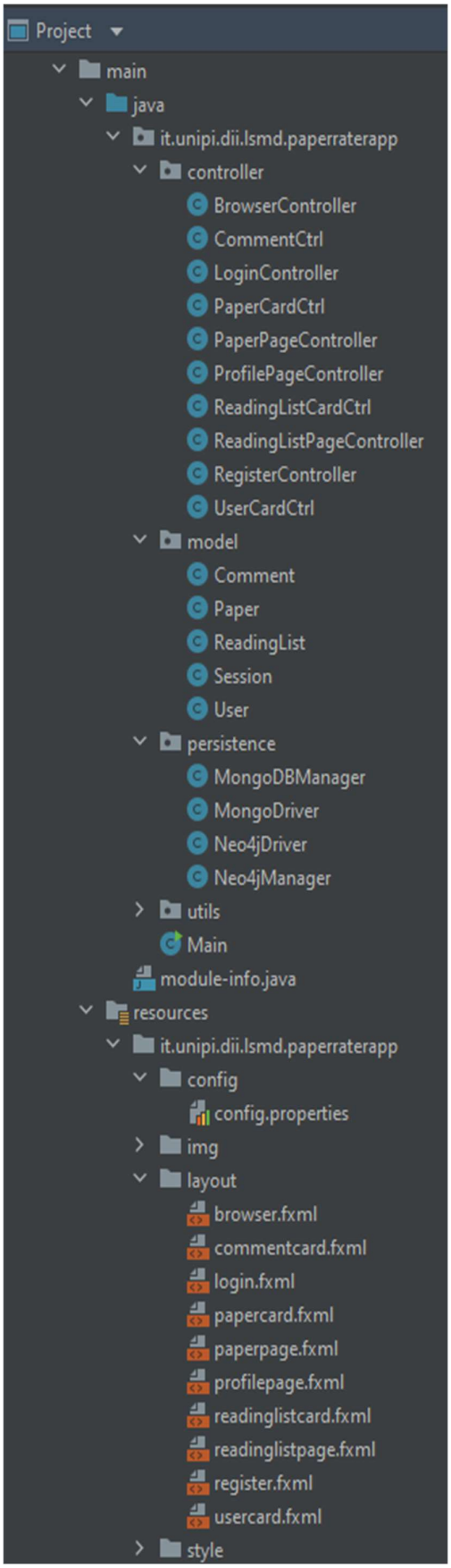
db_updater.py accepts three commands:

- *initDB*, initializes the database of the application by downloading papers data, users data and it populates the database with random *Reading Lists*, *Follows, Comments* and *Likes*.
- *updateDB*, downloads the latest papers published and it uploads them in the database.
- *exit*, close the application.

Application Package Structure

Main Packages and Classes

The PaperRater application is composed of the following packages and classes.



it.unipi.dii.lsmd.paperraterapp.model

This package contains the classes required for the model. These classes are the java bean for our application.
Classes:

- Paper: The class contains paper information.
- User: The class contains user information.
- Reading List: The class contains information of a user's reading list.
- Comment: The class contains comments information.
- Session: The page contains information about the logged user.

it.unipi.dii.lsmd.paperraterapp.persistence

This package contains the classes to interface with databases.
Classes:

- MongoDBDriver: Implement the methods for to manage the connection with MongoDB.
- MongoDBManager: In this class are implemented all the queries to interact with MongoDB.
- Neo4jDriver: Implement the methods for to manage the connection with Neo4j.
- Neo4jManager: In this class are implemented all the queries to interact with Neo4j.

it.unipi.dii.lsmd.paperraterapp.controller

This package implements the controller part of Model-view-controller of the application. Each controller is related to a different page that is shown to the user.
Classes:

- BrowserController: this class manage the homepage and allows the user to navigate to the other pages of application and to carry out search operation and browser.
- CommentCtrl: this class manage the view of comment card.
- LoginController: this class manage the login page of application.
- PaperCardCtrl: this class manage the view of paper card.
- PaperPageController: this class manage the page of a generic paper and allows the user to comment, like or add paper to a reading list.
- ProfilePageController: this class manage the generic user page of the application and allows the user to view/modify his data or view data of another user.
- ReadingListCardCtrl: this class manage the view of reading list card.
- ReadingListPageController: this class manage the reading list page, where information about follower and main category is displayed and the user who owns the reading list can remove papers.
- RegisterController: this class manage the register page of application.
- UserCardCtrl: this class manage the view of user card.

it.unipi.dii.lsmd.paperraterapp.utils

This package contains utility functions.
Classes:

- Utils: this class contains function for manage the configuration parameters necessary for the application and the scene change for pages.

Data Model

DocumentDB

We chose to use a document database in our application to manage most of the data regarding users and papers. This choice allows us to respect the *flexibility* and *high-performance* requirements. We decided to use two collections: **Papers** and **Users**.

With a document DB we can have document with missing fields without having to occupy memory with *null* values: in our application we use different fields to identify the source of a paper, we use an *arxiv_id* field to identify papers coming from arXiv.com and we use *vixra_id* to identify papers coming from viXra.com, and we do not need to set to null the value that is not used.

Regarding the performance, using a document DB allows us to embed objects that are commonly used together to avoid expensive join operations. We chose to embed *reading lists* inside the document of the user that owns it because inside a User Page are shown also all the reading lists that he created. In each Reading List is stored only a subset of information, the ones that are used to display a preview of the papers, while the complete information about papers can be retrieved in the corresponding paper document. The maximum number of reading lists that a user can create is ten, while the maximum number of papers inside a reading list is a hundred, because of the MongoDB document size limit.

We also decided to embed *comments* inside the paper document, for similar reasons: in the Paper Page we show all the comments related to the paper.
The schema less approach offered by document dbs also allows us to handle in a flexible way users without reading lists, reading lists without papers or papers without comments.

In the following figure it is shown an example of a document of *Papers* collection.

```
{  _id: ObjectId("61d6072237ab2f279ae614b1"),
  vixra_id: '1501.0008',
  title: 'Dark Matter-Induced Collapse of Neutron Stars',
  _abstract: 'Fast Radio Bursts (FRBs) are extreme bursts of radio emission
                The origin of these transients is still uncertain — we can
         . . . ',
  category: 'Astrophysics',
  authors: [ 'George Rajna' ],
  published: '2015-01-01',
  comments:
   [ { username: 'heavysnake701',
       text: 'Commento',
       timestamp: '2022-01-05 22:10:58' },
     { username: 'sadbutterfly345',
       text: 'Commento',
       timestamp: '2022-01-05 22:10:58' },
       . . .
    ]
}
```

In the following figure it is shown an example of a document of *Users* collection.

```
{  _id: ObjectId("61d9c43ede99d2747b20f13d"),
  username: 'happytiger362',
  email: 'ozsu.kaplangi@example.com',
  password: 'sabres',
  firstName: 'Özsu',
  lastName: 'Kaplangı',
  age: 67,
  readingLists:
   [ { title: 'r_list0',
       papers:
        [ { arxiv_id: '2111.15574',
            title: 'Polarization in quasirelativistic graphene model ...
            published: '2021-11-30',
            authors: [ 'Halina Grushevskaya', 'George Krylov' ],
            category: 'Condensed Matter' },
          { arxiv_id: '2112.00373',
            title: 'Synthesis and study of (Na, Zr) ...
            published: '2021-12-01',
            authors: ['M. E. Karaeva','D. O. Savinykh',
            category: 'Condensed Matter' },
           ...
        ]},
    { title: 'r_list1',
       papers:
        [       ...
        ]
    ]
}
```

GraphDB

We decided to use a graph database to manage the social network features of the application. Our choice was to keep the graph database as “light” as possible and to use it only for handling social network relationships and social network-based analytics.

The GraphDB nodes with their attributes are the following:

- **User**: {username, email}
- **Reading List**: {title, owner}
- **Paper**: {arxiv_id/vixra_id, title, authors, published, category}

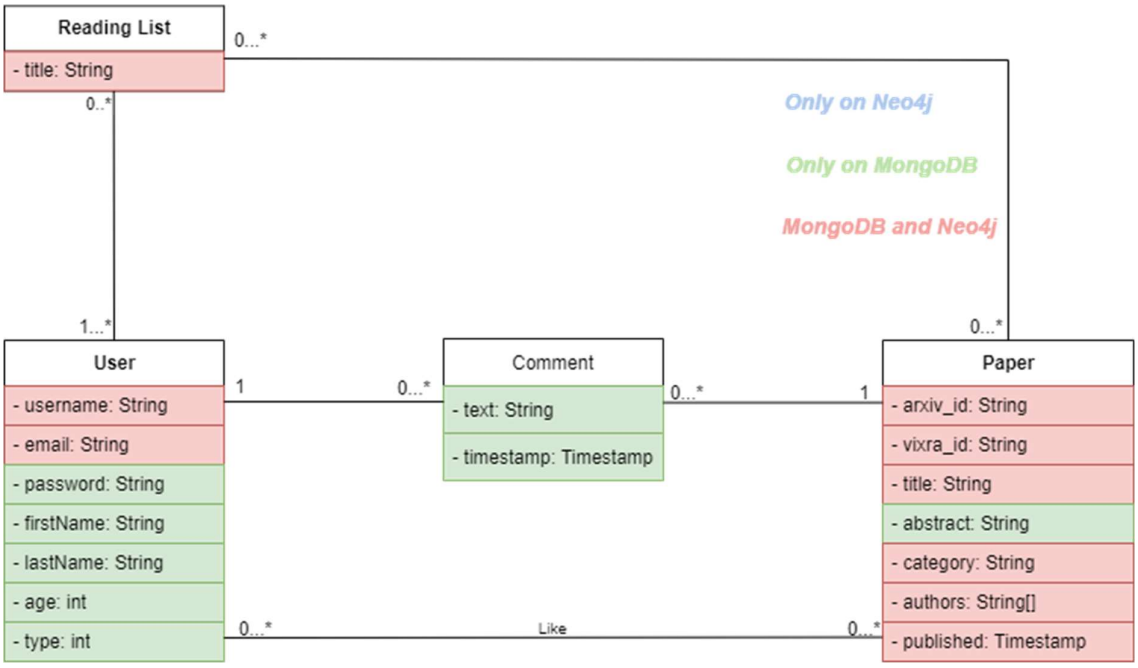
The relationships present in our graph database are the following:

- **Follows**: If a user follows another user (:User)-[:FOLLOWS]->(:User)
  If a user follows a Reading List (:User)-[:FOLLOWS]->(:ReadingList)
- **Like**: If a user likes a paper (:User)-[:LIKES]->(:Paper)

Data among databases

We decided to store some information in both the databases: we stored the information needed to load a preview (a card) of the various entities also in the graph database so we can display in the application the results of the social network analytics without having to access to MongoDB.

In the following figure is shown the storing strategy.



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*. In our application we want to offer a high availability of the content, even if an error occurs on the network layer, at the cost of returning, to the users, data which is not always accurate. For this reason, we adopt the *Eventually Consistency* paradigm on our dataset.

- **High availability** of the service due to the way we handle the writes operation. 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.
- **Partition Protection** of the service 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.

The possibility to implement a sharding would permit to evenly balance the workload among the nodes, improving users' experience.

Replicas

We decide to use replicas due to guarantee Partition protection and Availability of the content. When the user does a write operation just one server will be updated immediately, in this way the write operation will not take so much time and other users that are going to do read operations, which we suppose will be the most frequent ones, don't have to wait too much. The presence of replicas helps us also if we have network problems and we can't reach one server, in this case the query will be redirected to another server which contains the replica. Unfortunately, to guarantee these two services we can't ensure that the user will retrieve the most updated content.

When the "paper update" process runs, we estimate once a day, we want to guarantee that all the replicas will have the same information about new papers, so only in this case we need a consistency constraint, and the write operations will take more time.

In our project we have only 3 replicas are present in our systems: one for each machine of the provided cluster. However, in our implementation we have replicas only for **MongoDB**, because **Neo4J** replicas is a premium feature.

The command which was used to set cluster configuration in MongoDB on the virtual machines will now be shown. We also tried to balance the load by giving a lower priority to the VM running both MongoDB and Neo4J.

```
rsconfig = {
      _id: "lsmdb",
      members: [
          {
                _id: 0,
                host: "172.16.4.66:27020“,
                priority:5
          },
          {
                _id: 1,
                host: "172.16.4.67:27020“,
                priority:2
          },
          {
                _id: 2,
                host: "172.16.4.68:27020“,
                priority:1}
     ]
};
```

```
# mongod.conf

storage:
  dbPath: /root/data
  journal:
    enabled: true

systemLog:
  destination: file
  logAppend: true
  path: "/var/log/mongodb/mongod.log"

net:
  bindIp: localhost,172.16.4.66
  port: 27020

processManagement:
  fork: true

security:
  authorization: enabled
  keyFile: /opt/mongo/security


replication:
  oplogSizeMB: 200
  replSetName: lsmdb
```

Sharding

To implement the sharding we select two different sharding keys, one for each collection of the document database (User Collection and Paper Collection), for both 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 "*username",* which is unique among the users, as sharding key.
- For the Paper Collection we choose the attribute "_id", which is automatically generated by MongoDB, as sharding key.

MongoDB Design and Implementation

The MongoDB database contains the following collections: Papers and Users. We decided to store in each document all the information that we need to build the information pages. Sometimes we use mongo also to display advanced information retrieved by analytic query (basically when we do not need to use the relationships between entities).

Queries Implementation

CRUD operation

Operation	Implementation

Create a User	db.Users.insertOne(
	{
	username: "username",
	email: "email",
	password: "password",
	firstName: "firstname",
	lastName: "lastName",
	age: "age"
	}
	)

Delete User	db.Users.deleteOne(
	{
	username: "username"
	}
	)

Update User	db.Users.updateOne(
	{
	username: "username",
	email: "email",
	password: "password",
	firstName: "firstname",
	lastName: "lastName",
	age: "age",
	type: “type”
	}
	)

Get Users by username	db.Users.findOne(
	{
	username: "username"
	}
	)

Analytics

Get the most versatile users

Select users that have the highest number of categories in their reading list

Mongo java manager:

```
public List<Pair<User, Integer>> getTopVersatileUsers (int skipDoc, int
limitDoc) {
    List<Pair<User, Integer>> results = new ArrayList<>();
    Gson gson = new
GsonBuilder().serializeSpecialFloatingPointValues().create();
    Consumer<Document> convertInUser = doc -> {
        User user = gson.fromJson(gson.toJson(doc), User.class);
        results.add(new Pair(user, doc.getInteger("totalCategory")));
    };

    Bson unwind1 = unwind("$readingLists");
    Bson unwind2 = unwind("$readingLists.papers");
    // Distinct occurrences
    Bson groupMultiple = new Document("$group",
            new Document("_id", new Document("username", "$username")
                    .append("email", "$email")
                    .append("password", "$password")
                    .append("firstName", "$firstName")
                    .append("lastName", "$lastName")
                    .append("age", "$age")
                    .append("category", "$readingLists.papers.category")
            ));
    // Sum all occurrences
    Bson group = new Document("$group",
            new Document("_id",
                    new Document("username", "$_id.username")
                            .append("email", "$_id.email")
                            .append("password", "$_id.password")
                            .append("firstName", "$_id.firstName")
                            .append("lastName", "$_id.lastName")
                            .append("age", "$_id.age"))
                    .append("totalCategory",
                            new Document("$sum", 1)));

    Bson project = project(fields(excludeId(),
            computed("username", "$_id.username"),
            computed("email", "$_id.email"),
            computed("password", "$_id.password"),
            computed("firstName", "$_id.firstName"),
            computed("lastName", "$_id.lastName"),
            computed("age", "$_id.age"),
            include("totalCategory")));
    Bson sort = sort(descending("totalCategory"));
    Bson skip = skip(skipDoc);
    Bson limit = limit(limitDoc);

    usersCollection.aggregate(Arrays.asList(unwind1, unwind2,
groupMultiple, group, project, sort, skip, limit))
            .forEach(convertInUser);

    return results;
}
```

Mongo:

```
db.Users.aggregate(
    {$unwind: "$readingLists"},
    {$unwind: "$readingLists.papers"},
    {
        $group: {
            _id: {username: "$username", email: "$email", password:
"$password", firstName: "$firstName",
                lastName: "$lastName", picture: "$picture", age: "$age",
category: "$readingLists.papers.category"}
        }
    },
    {
        $group: {
            _id: {username: "$_id.username", email: "$_id.email", password:
"$_id.password", firstName: "$_id.firstName",
                    lastName: "$_id.lastName", picture: "$_id.picture",
age: "$_id.age"}, totalCategory: {$sum, 1}
        }
    },
    {
        $project: {
            _id: 0,
            username: "$_id.username",
            email: "$_id.email",
            password: "$_id.password",
            firstName: "$_id.firstName",
            lastName: "$_id.lastName",
            picture: "$_id.picture",
            age: "$_id.age"
            totalCategory: 1
        }
    },
    {
        $skip: 8
    },
    {
        $limit: 8
    }
);
```

Get the most commented papers
Select papers with the highest number of comments.

Mongo java driver:

```
public List<Pair<Paper, Integer>> getMostCommentedPapers(String period, int
skipDoc, int limitDoc) {
    LocalDateTime localDateTime = LocalDateTime.now();
    LocalDateTime startOfDay;
    switch (period) {
        case "all" -> startOfDay = LocalDateTime.MIN;
        case "month" -> startOfDay =
localDateTime.toLocalDate().atStartOfDay().minusMonths(1);
        case "week" -> startOfDay =
localDateTime.toLocalDate().atStartOfDay().minusWeeks(1);
        default -> {
            System.err.println("ERROR: Wrong period.");
            return null;
        }
    }
    String filterDate =
startOfDay.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));

    List<Pair<Paper, Integer>> results = new ArrayList<>();
    Gson gson = new
GsonBuilder().serializeSpecialFloatingPointValues().create();
    Consumer<Document> convertInPaper = doc -> {
        Paper paper = gson.fromJson(gson.toJson(doc), Paper.class);
        results.add(new Pair(paper, doc.getInteger("totalComments")));
    };

    Bson unwind = unwind("$comments");
    Bson filter = match(gte("comments.timestamp", filterDate));
    Bson group = new Document("$group",
            new Document("_id",
                    new Document("arxiv_id", "$arxiv_id")
                            .append("vixra_id", "$vixra_id")
                            .append("title", "$title")
                            .append("_abstract", "$_abstract")
                            .append("category", "$category")
                            .append("authors", "$authors")
                            .append("published", "$published"))
                    .append("totalComments",
                            new Document("$sum", 1)));
    Bson project = project(fields(excludeId(),
            computed("arxiv_id", "$_id.arxiv_id"),
            computed("vixra_id", "$_id.vixra_id"),
            computed("title", "$_id.title"),
            computed("_abstract", "$_id._abstract"),
            computed("category", "$_id.category"),
            computed("authors", "$_id.authors"),
            computed("published", "$_id.published"),
            include("totalComments")));
    Bson sort = sort(Indexes.descending("totalComments"));
    Bson skip = skip(skipDoc);
    Bson limit = limit(limitDoc);
    papersCollection.aggregate(Arrays.asList(unwind, filter, group,
project, sort, skip, limit)).forEach(convertInPaper);

    return results;
}
```

Mongo:

```
db.Papers.aggregate(
    {$unwind: "$comments"},
    {
        $match: {
            "comments.timestamp": {$gte: "2021-12-20 00:00:00"}
        }
    },
    {
        $group: {
            _id: {arxiv_id: "$arxiv_id", vixra_id: "$vixra_id", title:
"$title", _abstract: "$_abstract",
                    category: "$category", authors: "$authors", published:
"$published"}, totalComments: {$sum, 1}
        }
    },
    {
        $project: {
            _id: 0,
            arxiv_id: "$_id.arxiv_id",
            vixra_id: "$_id.vixra_id",
            title: "$_id.title",
            _abstract: "$_id._abstract",
            category: "$_id.category",
            authors: "$_id.authors",
            published: "$_id.published"
            totalComments: 1
        }
    },
    {
        $sort: {
            totalComments: -1
        }
    },
    {
        $skip: 3
    },
    {
        $limit: 3
    }
);
```

Get categories summary by number of papers published
This function returns the categories ordered by the number of papers published.

Mongo java driver:

```
public List<Pair<String, Integer>>
getCategoriesSummaryByNumberOfPaperPublished (String period){
    LocalDateTime localDateTime = LocalDateTime.now();
    LocalDateTime startOfDay;
    switch (period) {
        case "all" -> startOfDay = LocalDateTime.MIN;
        case "month" -> startOfDay =
localDateTime.toLocalDate().atStartOfDay().minusMonths(1);
        case "week" -> startOfDay =
localDateTime.toLocalDate().atStartOfDay().minusWeeks(1);
        default -> {
            System.err.println("ERROR: Wrong period.");
            return null;
        }
    }
    String filterDate =
startOfDay.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));

    List<Pair<String,Integer>> categories = new ArrayList<>();

    Bson match = match(gte("published", filterDate));
    Bson group = group("$category", sum("totalPaper", 1));
    Bson project = project(fields(excludeId(), computed("category",
"$_id"), include("totalPaper")));
    Bson sort = sort(descending("totalPaper"));

    List<Document> results = (List<Document>)
papersCollection.aggregate(Arrays.asList(match, group, project,
sort)).into(new ArrayList<>());

    for (Document document: results)
    {
        categories.add(new Pair(document.getString("category"),
document.getInteger("totalPaper")));
    }
    return categories;
}
```

Mongo:

```
db.Papers.aggregate(
    {
        $match: {
        "published": {$gte: "2021-12-20 00:00:00"}
    },
    {
        $group: {
            _id: "$category", totalPaper: {$sum, 1}
        }
    },
    {
        $project: {
            _id: 0,
            category: "$_id",
```

```
            totalPaper: 1
        }
    },
    {
        $sort: {
            totalPaper: -1
        }
    }
);
```

Get categories summary by comments
This function returns the categories ordered by the number of comments.

Mongo java driver:

```
public List<Pair<String, Integer>> getCategoriesSummaryByComments(String
period) {
    LocalDateTime localDateTime = LocalDateTime.now();
    LocalDateTime startOfDay;
    switch (period) {
        case "all" -> startOfDay = LocalDateTime.MIN;
        case "month" -> startOfDay =
localDateTime.toLocalDate().atStartOfDay().minusMonths(1);
        case "week" -> startOfDay =
localDateTime.toLocalDate().atStartOfDay().minusWeeks(1);
        default -> {
            System.err.println("ERROR: Wrong period.");
            return null;
        }
    }
    String filterDate =
startOfDay.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));

    List<Pair<String, Integer>> results = new ArrayList<>();
    Consumer<Document> rankCategories = doc ->
            results.add(new Pair<>((String) doc.get("_id"), (Integer)
doc.get("tots")));

    Bson unwind = unwind("$comments");
    Bson filter = match(gte("comments.timestamp", filterDate));
    Bson group = group("$category", sum("tots", 1));
    Bson sort = sort(Indexes.descending("tots"));
    papersCollection.aggregate(Arrays.asList(unwind, filter, group,
sort)).forEach(rankCategories);

    return results;
}
```

Mongo

```
db.Papers.aggregate(
    {$unwind: "$comments"},
    {
        $match: {
            "comments.timestamp": {$gte: "2021-12-20 00:00:00"}
    },
    {
```

```
        $group: {
            _id: "$category", tots: {$sum, 1}
        }
    },
    {
        $sort: {
            tots: -1
        }
    }
);
```

Neo4J Design and Implementation

In Neo4j database there are the following entities: Paper, User, ReadingList, that contain only the basic information needed to show a preview of the entity.

There are also the following relationships between entities:

- *USER – LIKES -> PAPER*: Each user can like one or more papers; it is useful for showing suggestions based on paper's like.
- *USER – FOLLOWS -> READING LIST*: Each user can follow one or more reading list created by other user; it is useful for showing suggestions based on reading list.
- *USER – FOLLOWS -> USER*: Each user can follow one or more users; it is useful for showing suggestions based on users.

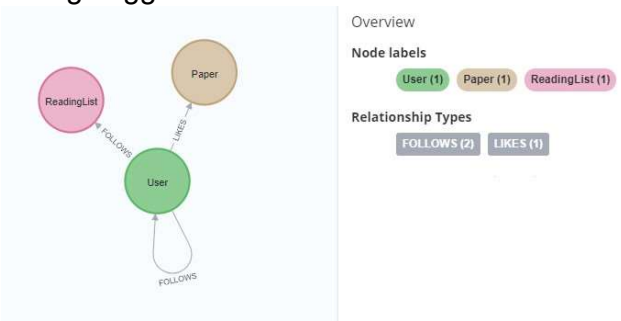


FIGURE 3 NEO4J SCHEMA VISUALIZATION

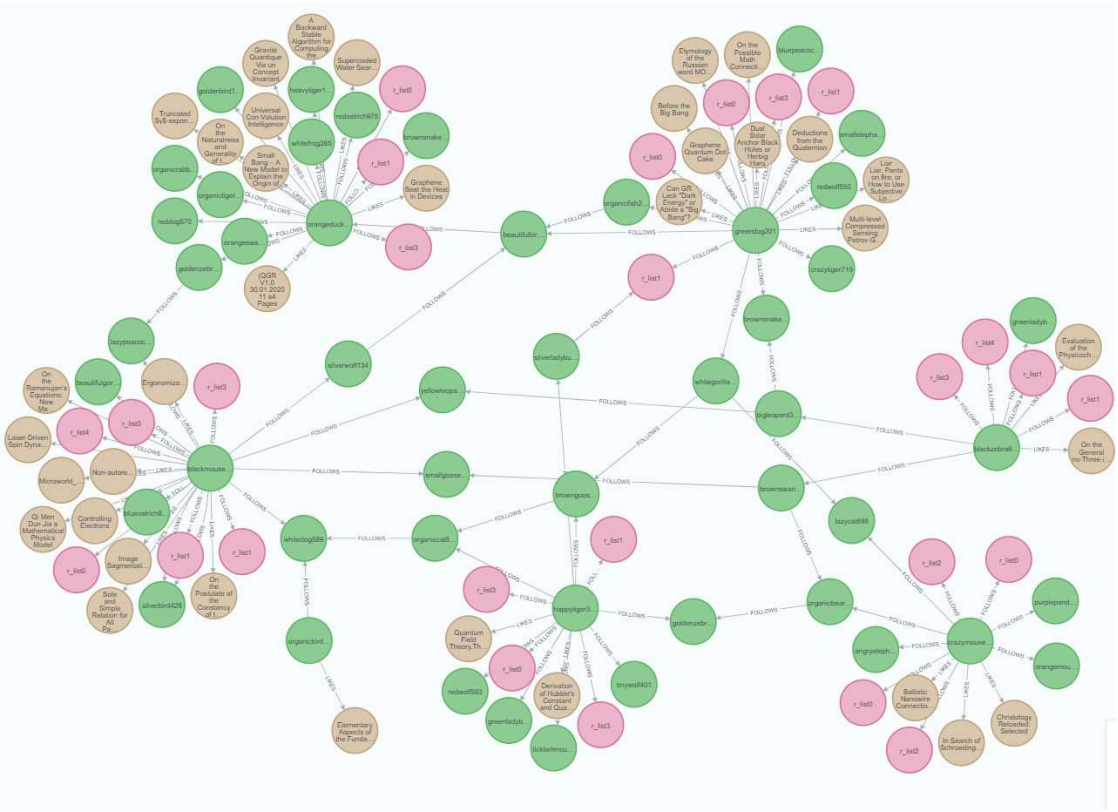


FIGURE 4 PARTIAL EXAMPLE OF THE NEO4J GRAPH

Queries Implementation

CRUD operation

Create

Operation	Cypher Implementation
Create a User node	**CREATE** (u:User {username: $username})
Create a User-LIKES->Paper relation	**MATCH** (a:User), (b:Paper) **WHERE** a.username = $username **AND** (b.arxiv_id = $arxiv_id AND b.vixra_id = $vixra_id) **MERGE** (a)-[r:LIKES]->(b)
Create a ReadingList node and User-FOLLOWS->ReadingList relation	**MATCH** (owner:User {username: $owner}) **MERGE** (r:ReadingList {title: $title, owner: owner}) **MERGE** (owner)-[p:OWNS]->(r) **ON CREATE SET** p.date = datetime()
Create a User-FOLLOWS->User relation	**MATCH** (u:User {username: $username}), (t:User {username: $target}) **MERGE** (u)-[p:FOLLOWS]->(t) **ON CREATE SET** p.date = datetime()
Create a User-FOLLOWS->ReadingList relation	**MATCH** (u:User {username: $username}), (r:ReadingList {title: $title, owner: $owner}) **MERGE** (u)-[p:FOLLOWS]->(r) **ON CREATE SET** p.date = datetime()

Read

Operation	Cypher Implementation
Retrieve the number of user's followers	**MATCH** (:User {username: $username})<-[r:FOLLOWS]-() **RETURN COUNT** (r) AS numFollowers
Retrieve the number of followed user	**MATCH** (:User {username: $username})-[r:FOLLOWS]->() **RETURN COUNT** (r) AS numFollowers
Check if a user follows another user	**MATCH** (a:User{username:$userA})-[r:FOLLOWS]->(b:User{username:$userB}) **RETURN COUNT** (*)
Check if a user follows a reading list	**MATCH** (a:User{username:$user})-[r:FOLLOWS]->(b:ReadingList{title:$title, username:$owner }) **RETURN COUNT** (*)
Check if a user likes a paper	**MATCH** (:User{username:$user})-[r:LIKES]->(p:Paper) **WHERE** (p.arxiv_id = $arxiv_id AND p.vixra_id =$vixra_id) **RETURN COUNT** (*)

Retrieve the number or reading list's followers	**MATCH** (:ReadingList {title: $title, owner: $owner})<-[r:FOLLOWS]-() **RETURN COUNT** (r) **AS** numFollowers
Retrieve the number of paper's likes	**MATCH** (p:Paper)<-[r:LIKES]-() **WHERE** p.arxiv_id = $arxiv_id **AND** p.vixra_id = $vixra_id **RETURN COUNT** (r) **AS** numLikes
Retrieve followed user by keyword	**MATCH** (:User {username: $username})-[:FOLLOWS]->(u:User) **WHERE** toLower(u.username) **CONTAINS** $keyword **RETURN** u.username **AS** Username, u.email **AS** Email **ORDER BY** Username **DESC SKIP** $skip **LIMIT** $limit
Retrieve followed reading list by keyword	**MATCH** (u:User {username: $username})-[r:FOLLOWS]->(b:ReadingList) **WHERE** toLower(b.title) **CONTAINS** $keyword **RETURN** b.owner as username, b.title as title **ORDER BY** username **SKIP** $skip **LIMIT** $limit

Update

Operation	Cypher Implementation
Update User email	**MATCH** (u:User {username: $username}) SET u.email = $newEmail

Delete

Operation	Cypher Implementation
Delete a User node	**MATCH** (u:User) **WHERE** u.username = $username **DETACH DELETE** u
Delete a User-LIKES->Paper relation	**MATCH** (u:User{username:$username})-[r:LIKES]->(p:Paper {arxiv_id:$arxiv_id, vixra_id:$vixra_id}) **DELETE** r
Delete a ReadingList node and User-FOLLOWS->ReadingList relation	**MATCH** (r:ReadingList {title: $title, owner: owner}) **DETACH DELETE** r
Delete a User-FOLLOWS->User relation	**MATCH** (:User {username: $username})-[r:FOLLOWS]->(:User {username: $target **DELETE** r
Delete a User-FOLLOWS->ReadingList relation	**MATCH** (:User {username: $username})-[r:FOLLOWS]->(:ReadingList {title: $title, owner: $owner}) **DELETE** r

Suggestion

Most liked paper

The query returns the list of the most liked papers in a specific period.

- Input: Period of publication of the papers, how many papers to skip and how many paper to show.
- Result: A list of the most liked paper and the number of likes.

```
MATCH (:User)-[l:LIKES]->(p:Paper)
WHERE p.published >= $start_date
RETURN p.arxiv_id AS ArxivId, p.vixra_id AS VixraId, p.title AS Title,
       p.category AS Category, p.authors AS Authors,
COUNT(l) AS like_count ORDER BY like_count DESC
SKIP $skip
LIMIT $limit
```

Most Followed users

The query returns the list of the most followed users.

- Input: how many users to skip and how many users to show.
- Result: A list of the most followed users and the number of followers.

```
MATCH (target:User)<-[r:FOLLOWS]-(:User)
RETURN DISTINCT target.username AS Username, target.email AS Email,
COUNT(DISTINCT r) AS numFollower ORDER BY numFollower DESC
SKIP $skip
LIMIT $num
```

Most Followed Reading lists

The query returns the list of the most followed reading lists.

- Input: how many reading lists to skip and how many reading lists to show.
- Result: A list of the most followed reading lists, the authors of reading list and the number of followers.

```
MATCH (target:ReadingList)<-[r:FOLLOWS]-(:User)
RETURN DISTINCT target.title AS Title, target.owner AS Owner,
COUNT(DISTINCT r) as numFollower ORDER BY numFollower DESC
SKIP $skip
LIMIT $num
```

Suggested papers

The query returns a list of suggested papers for the logged user. Suggestions are based on papers liked by followed users (first level) and papers liked by user that are 2 FOLLOWS hops far from the logged user (second level). Papers returned are ordered by the number of times they appeared in the results, so papers that appear more are most likely to be like the interests of the logged user.

- Input: username of the user, how many papers to skip and how many papers to show from first level, how many papers to skip and how many papers to show from second level.
- Result: A list of the suggested papers.

```
MATCH (target:Paper)<-[r:LIKES]-(u:User)<-
       [:FOLLOWS](me:User{username:$username})
WHERE NOT EXISTS((me)-[:LIKES]->(target))
RETURN  target.arxiv_id AS ArxivId, target.vixra_id AS VixraId,
target.title AS        Title, target.category AS Category, target.authors
AS Authors,
COUNT (*) AS nOccurences
ORDER BY nOccurences DESC
SKIP $skipFirstLevel
LIMIT $firstlevel
UNION
MATCH (target:Paper)<-[r:LIKES]-(u:User)<-[:FOLLOWS*2..2]-
       (me:User{username:$username})
WHERE NOT EXISTS((me)-[:LIKES]->(target))
RETURN target.arxiv_id AS ArxivId, target.vixra_id AS VixraId, target.title
as      Title, target.category AS Category, target.authors AS Authors,
COUNT (*) AS nOccurences
ORDER BY nOccurences DESC
SKIP $skipSecondLevel
LIMIT $secondLevel
```

Suggested users

The query returns a list of suggested users for the logged user. Suggestions are based on most followed users who are 2 FOLLOWS hops far from the logged user (first level), while the second level of suggestion returns most followed users that have likes in common with the logged user.

- Input: username of the user, how many users to skip and how many users to show from first level, how many users to skip and how many users to show from second level.
- Result: A list of the suggested users.

```
MATCH (me:User {username: $username})-[:FOLLOWS*2..2]->(target:User),
       (target)<-[r:FOLLOWS]-()
WHERE NOT EXISTS((me)-[:FOLLOWS]->(target))
RETURN DISTINCT target.username AS Username, target.email AS Email,
COUNT(DISTINCT r) as numFollower
ORDER BY numFollower DESC
SKIP $skipFirstLevel
LIMIT $firstLevel
UNION
MATCH (me:User {username: $username})-[:LIKES]->()<-[:LIKES]-(target:User),
       (target)<-[r:FOLLOWS]-()
WHERE NOT EXISTS((me)-[:FOLLOWS]->(target))
RETURN target.username AS Username, target.email AS Email,
COUNT(DISTINCT r) as numFollower
ORDER BY numFollower DESC
SKIP $skipSecondLevel
LIMIT $secondLevel
```

Suggested reading lists

The query returns a list of suggested reading lists for the logged user. Suggestions are based on most followed reading lists followed by followed users (first level), and most followed reading lists followed by users that are 2 FOLLOWS hops far from the logged user (second level).

- Input: username of the user, how many reading lists to skip and how many reading lists to show from first level, how many reading lists to skip and how many reading lists to show from second level.
- Result: A list of the suggested users.

```
MATCH (target:ReadingList)<-[f:FOLLOWS]-(u:User)<-[:FOLLOWS]
        (me:User{username:$username}), (target)<-[r:FOLLOWS]-(n:User)
WITH DISTINCT me, target,
COUNT (DISTINCT r) AS numFollower,
COUNT(DISTINCT u) AS follow
WHERE NOT EXISTS((me)-[:FOLLOWS]->(target))
RETURN target.owner AS Owner, target.title AS Title, numFollower + follow
AS      followers
ORDER BY followers DESC
SKIP $skipFirstLevel
LIMIT $firstLevel
UNION
MATCH (target:ReadingList)<-[f:FOLLOWS]-(u:User)<-[:FOLLOWS*2..2]-
        (me:User{username:$username}), (target)<-[r:FOLLOWS]-(n:User)
WITH DISTINCT me, target,
COUNT (DISTINCT r) AS numFollower,
COUNT (DISTINCT u) AS follow
WHERE NOT EXISTS((me)-[:FOLLOWS]->(target))
RETURN target.owner AS Owner, target.title AS Title, numFollower + follow
AS      followers
ORDER BY followers DESC
SKIP $skipSecondLevel
LIMIT $secondLevel
```

Get categories summary by likes

The query returns a list of the most liked categories in a specific period of publication of the papers.

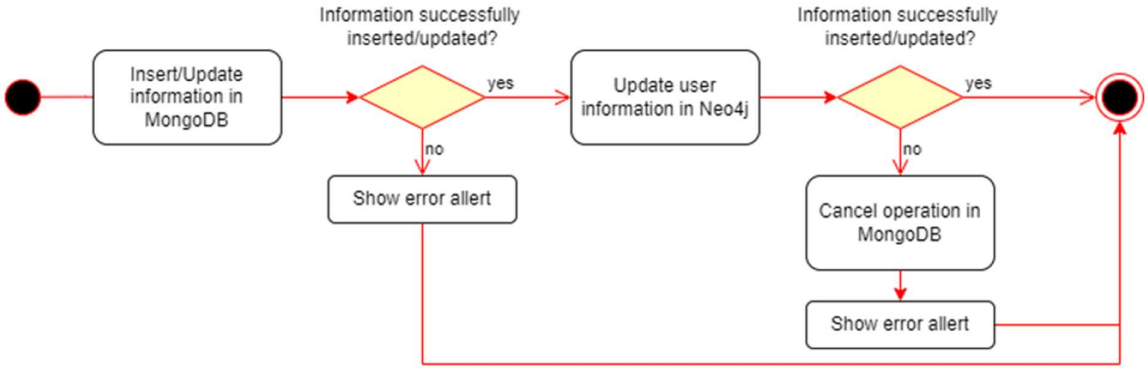
- Input: period of publication of the papers.
- Result: A list of the categories and the sum of like received by the paper in each category.

```
MATCH (p:Paper)<-[l:LIKES]-(:User)
WHERE p.published >= $start_date
RETURN count(l) AS nLikes, p.category AS Category
ORDER BY nLikes DESC
```

Cross-Database Consistency Management

The operations which require cross-database consistency management are Add/Remove/Update a User Add/Remove Reading list. For the update of the users, we have only to keep consistency on the field *“email”,* because it is the only information updatable in Neo4J entity.

In general, if an error occurs with the first write operation we show an error to the user, while if an error occurs with the second write operation, we also show an error, but we must undo the first write operation.



Test and Statistical Analysis

In the following table is shown the frequency of usage of the main queries of our application.

Query	Frequency
Search Papers by Title	High
Search Papers by Author	High
Search Papers by Published Date	Medium
Search Papers by Category	High
Search ReadingLists by Title	Medium
Search Users by Username	Medium

Note: in the application code there is none of the first four queries shown in the table, all of them are “merged” in the “searchPaperByParameters” query that is used in the side search bar; these queries can be executed individually by setting only the related parameter.

Indexes Analysis

As the query analysis has revealed, our application is read heavy, so we introduced indexes to tune queries that are executed more frequently. One of the main purposes of the application is to provide fast search results regarding papers, so we will focus mostly on indexes related to papers.

Indexes regarding “Papers” collection are particularly convenient because paper documents are almost not updatable (only the comments field can be updated but the application does not require frequent queries involving this field). There will not be any overhead related to updates operations and the overhead related to insert operation will affect only the *DB Updater* and not the users (only the DB_Updater can create new papers).

We will perform statistical tests to understand whether the insertion of an index is beneficial or not in terms of query performance.

Index “title”

For the performance analysis of this index, we considered the query “Search Papers by Title” that returns all the papers whose title matches a keyword. As we can see in the picture below, there is a statistical improvement when we specify only a character.

Query	Results without Index	Results with index
SearchPapersbyTitle(“A”)	executionTimeMillis: 89 totalKeysExamined: 0 totalDocsExamined: 50032	executionTimeMillis: 39 totalKeysExamined: 5931 totalDocsExamined: 5930

If we specify a longer keyword, the results are significantly improved.

Query	Results without Index	Results with index
SearchPapersbyTitle(“Architecture”)	executionTimeMillis: 97 totalKeysExamined: 0 totalDocsExamined: 50032	executionTimeMillis: 4 totalKeysExamined: 1 totalDocsExamined: 0

The size is almost 3.5 MB and in general will grow with the number of papers in the database, but the statistical results were very good, and the query is one of the most frequent, so we decided that the index will be included in the application.

Index “authors”

The analysis regarding the authors index gave similar results to the previous one. we considered the query “Search Papers by Authors” that returns all the papers whose at least one of the authors names matches a keyword. As shown in the table the performances are not significantly improved with short keywords, but excellent results are obtained with longer ones. In conclusion, we decided to include the index in our database.

Query	Results without Index	Results with index
SearchPapersbyAuthors(“l”)	executionTimeMillis: 93 totalKeysExamined: 0 totalDocsExamined: 50032	executionTimeMillis: 129 totalKeysExamined: 17859 totalDocsExamined: 3246

Query	Results without Index	Results with index
SearchPapersbyAuthors("Leonardo")	executionTimeMillis: 209 totalKeysExamined: 0 totalDocsExamined: 50032	executionTimeMillis: 8 totalKeysExamined: 58 totalDocsExamined: 57

Index "published"

In the following table is shown the results of the "Search Papers by Published Date" query that returns all the papers between two dates.

Query	Results without Index	Results with index
SearchPapersbyPublishedDate("2021-10-01, "2021-10-01")	executionTimeMillis: 55 totalKeysExamined: 0 totalDocsExamined: 50032	executionTimeMillis: 13 totalKeysExamined: 5 totalDocsExamined: 5

Even though is not highly frequent, the index is not particularly heavy (598 KB) and will grow slower than the previous indexes, also the performance improvement is high. In the end, we included it in the database.

Index "Categories"

In the following table is shown the results of the "Search Papers by Category" query that returns all the papers within a category. Even though the performance has not improved as much as the previous indexes, this index is related to a high frequency query, it is quite small (290KB) and will not grow fast, so it was included in the database.

Query	Results without Index	Results with index
SearchPapersbyCategory("Physics")	executionTimeMillis: 62 totalKeysExamined: 0 totalDocsExamined: 50032	executionTimeMillis: 20 totalKeysExamined: 1761 totalDocsExamined: 1761

Index "ReadingList.title"

Regarding Reading Lists, we analysed an index on the "title" field to increase the performance of the "Search Reading List by Title" query. As shown in the table the performances were good but the index was not included in the application.

Query	Results without Index	Results with index
SearchReadingListsbyTitle("r_list1")	executionTimeMillis: 57 totalKeysExamined: 0 totalDocsExamined: 1005	executionTimeMillis: 37 totalKeysExamined: 664 totalDocsExamined: 664

The main reason is that reading lists can be updated by the user, so the insertion of the index will slow down some of the user's write operations (*addReadingList* and *deleteReadingList*). We thought that the performance improvement was not a good compromise with the write overhead.

Index "username"

Inserting ad index on username is useful to boost up the sharding of the user collection.

Recap

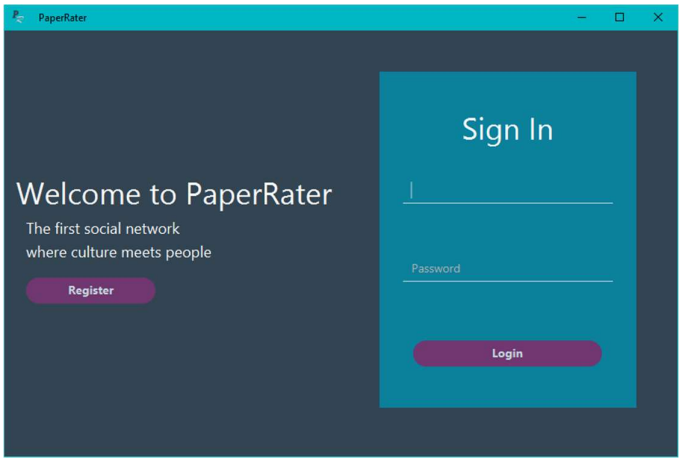
In conclusion, the database has the following indexes.

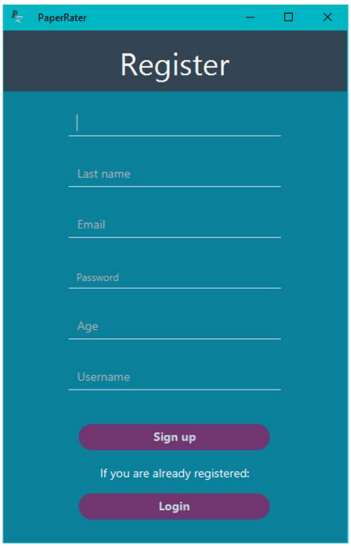
Index Type	Collection	Index Name	Index Field	Size
Hashed Index (default)	Papers	id	_id	1.5MB
Simple Index	Papers	title	title	3.5MB
Simple Index	Papers	authors	authors	4.3MB
Simple Index	Papers	category	category	290KB
Simple Index	Papers	publishedDate	published	598KB
Hashed Index (default)	Users	id	_id	45KB

User Manual

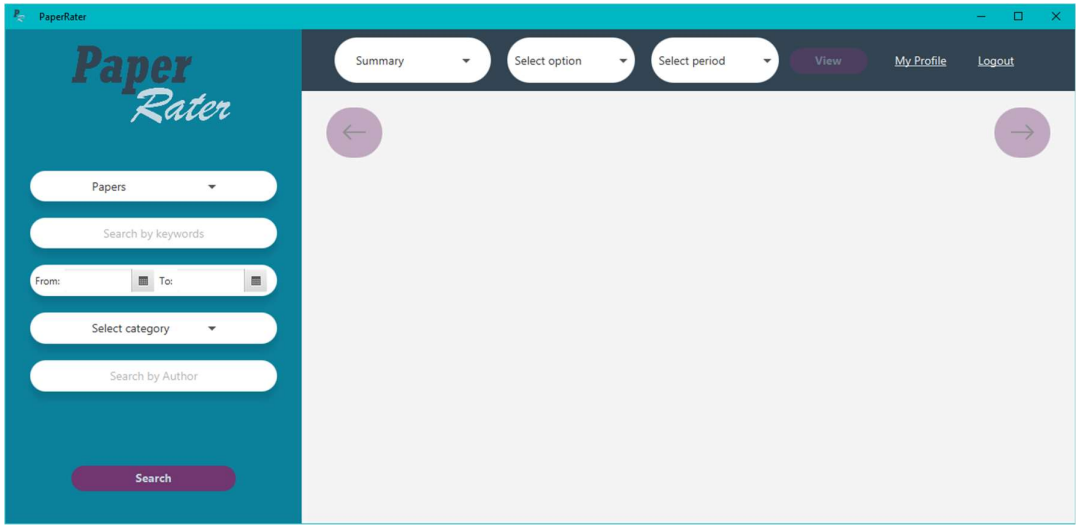
"*PaperRater*" is a user-friendly application, where users can search scientific papers and interact with other users as a social network.

- The first page which a user will see is the **Login** page, where a user can insert his own credentials or go to the **Register** page to create a new account.

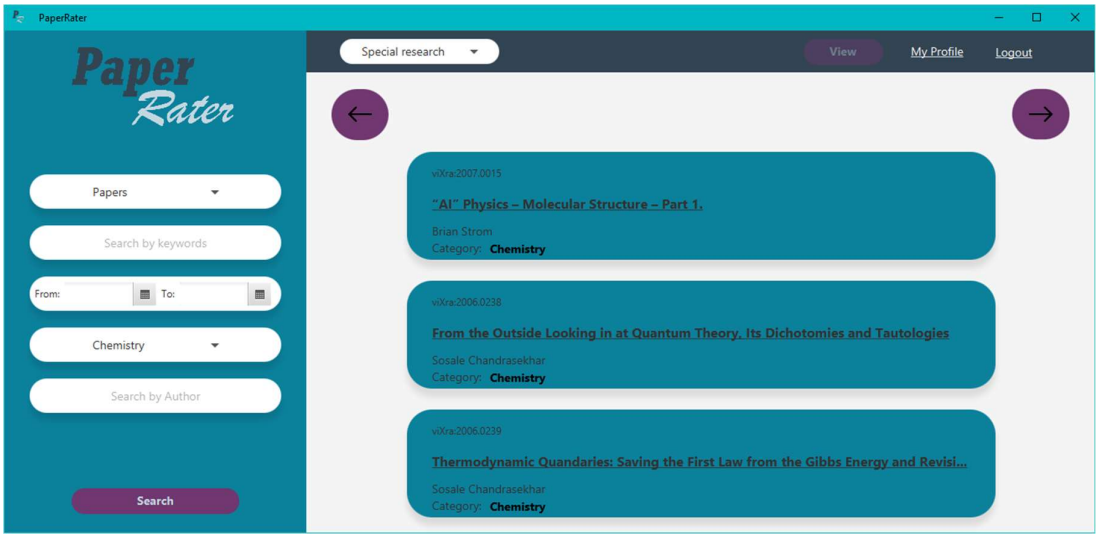




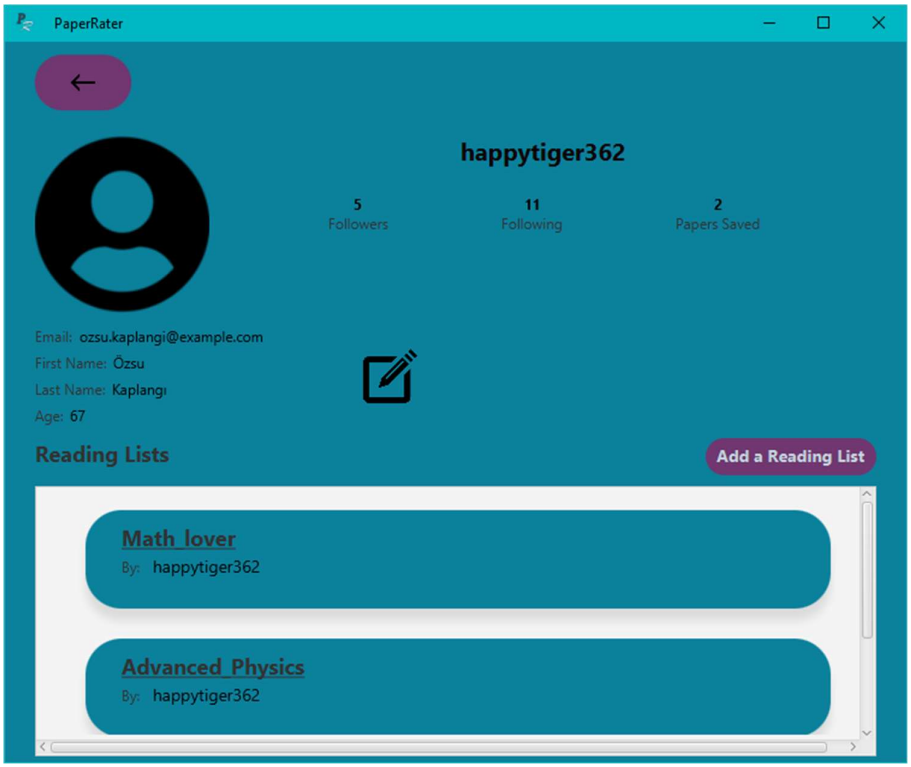
- After the login we are redirected to the **Browser**, in this page users can search by parameters other users, papers or reading lists. There is also the possibility to select special view of the data inside the application, in this way is possible to see some suggestions, analytics or summaries.



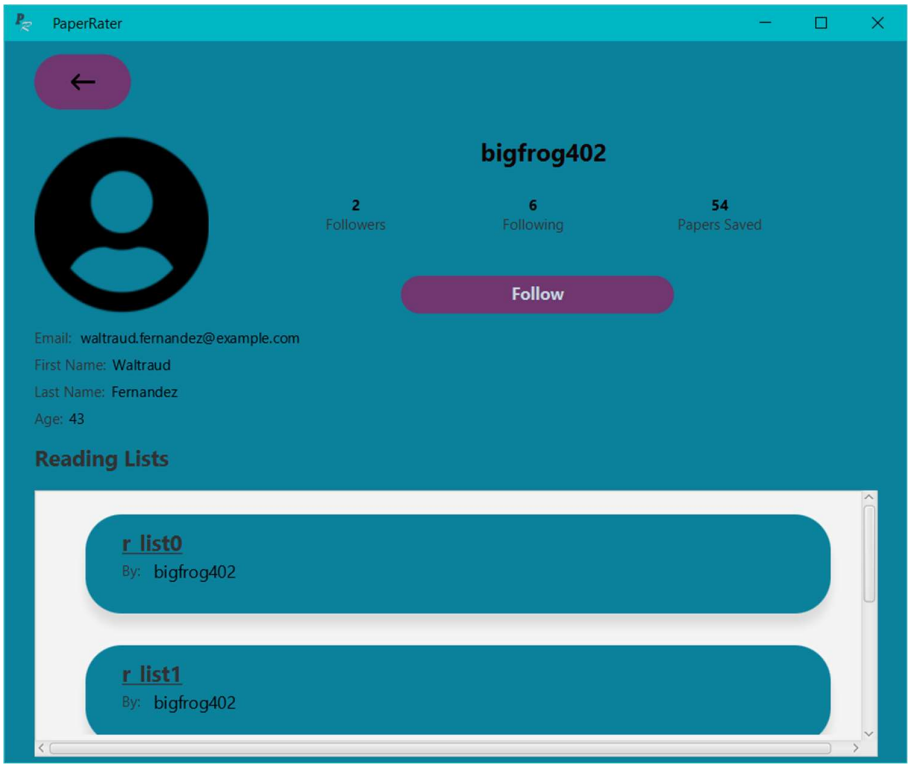
The user can navigate throw his result with two buttons on the top-left and top-right.



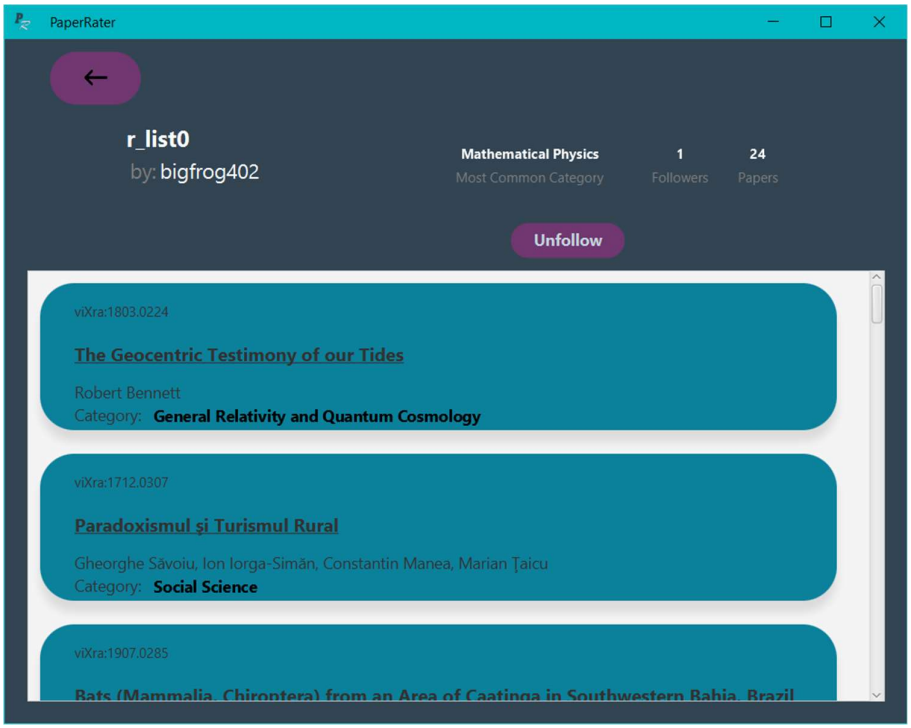
- If the user moves to his **profile page,** he will see his own information, the number of users that he follows and the number of his follower. He can also edit his profile or add new reading list.



- Clicking on the username of a user we will move to his profile page; we can follow or unfollow and view his reading lists.



- Clicking on a reading list a user will be redirected to the **reading list page**. Here a user can follow or unfollow it and see which paper are saved inside.



- Clicking on the title of a paper we will be redirected to the **paper page**. In this page a user can read the basic information about it, open the full document on his browser, add or remove a like and write his opinion about the content. He can also add it to one of his reading lists.