University of Pisa

Personality & Music

Large Scale and MultiStructured Databases

Cecchetti Jacopo
Terranova Franco

Academic Year 2021/2022

1. Project Idea

Many contemporary personality psychologists believe that there are five basic dimensions of personality, often referred to as the "Big 5" personality traits. The five broad personality traits described by the theory are extraversion, agreeableness, openness, conscientiousness and neuroticism. Each of the five personality factors represents a range between two extremes. In the real world, in fact, most people lie somewhere in between the two polar ends of each dimension.

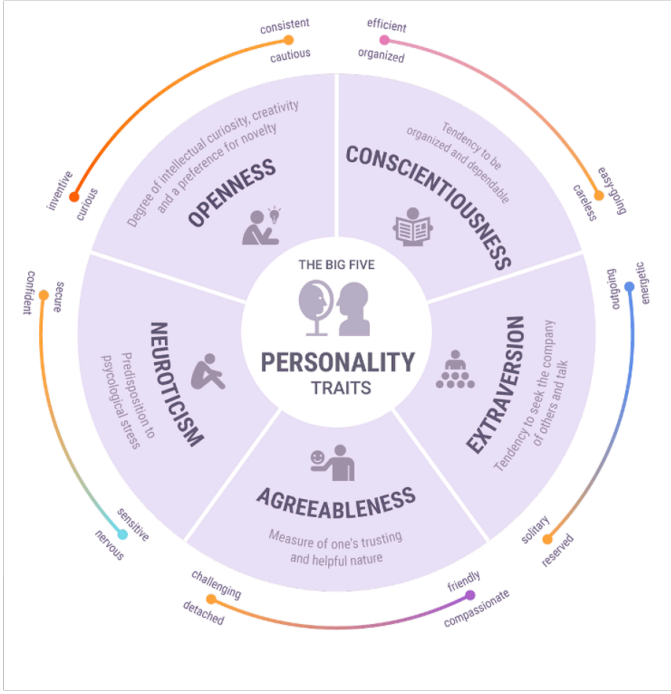


Figure 1. Personality Traits

- Conscientiousness. Standard features of this dimension include high levels of thoughtfulness, good impulse control, and goal-directed behaviors.
- Extraversion. This dimension is characterized by excitability, sociability, talkativeness, assertiveness, and high amounts of emotional expressiveness.
- Agreeableness. This personality dimension includes attributes such as trust, altruism, kindness, and affection.
- Neuroticism. This trait is characterized by sadness, moodiness, and emotional instability.
- Openness. This trait is characterized by curiosity, creativity, and preferences for novelty.

1.1. Summarization of Machine Learning Analysis

The dataset we used comes from Kaggle (https://www.kaggle.com/tunguz/big-five-personality-test) and was collected through an interactive on-line personality test. The personality test was constructed with the "Big-Five Factor Markers" from the IPIP (https://ipip.ori.org/newBigFive5broadKey.htm).
After applying a data preprocessing pipeline to the dataset and compared different clustering algorithms, we've determined that K-Means with K = 5 was the best clustering algorithm.

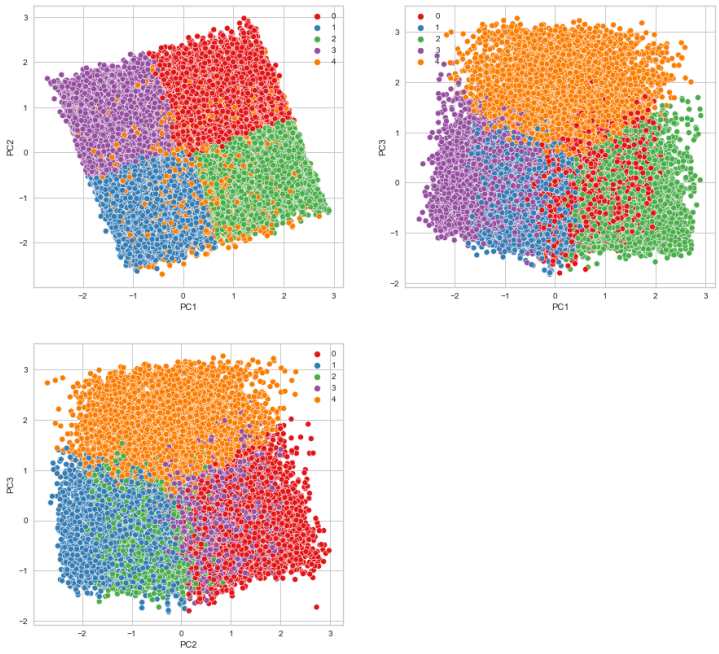


Figure 2. Principal Components

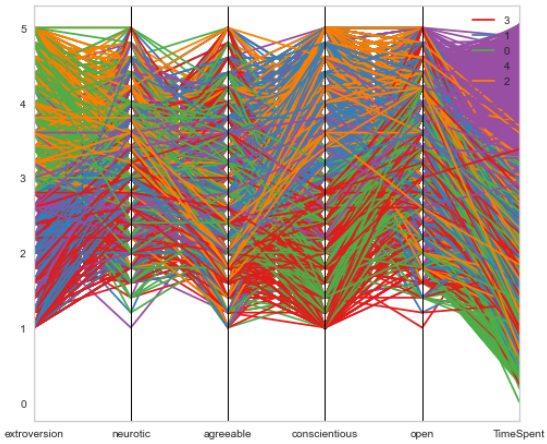


Figure 3. Parallel Coordinates

	Silhouette Score	Silhouette mean for each cluster	Std of clusters' silhouette	Number of negative single silhouette values	CHS	DBI	Time
K-Means	0,176	[0,20, 0,18, 0,14, 0,17, 0,18]	0,019	1524	10543,06	1,48	0,35 seconds

Further information about the Analysis can be found in the following Github Repository: https://github.com/terranovaa/PersonalityClusteringAnalysis.

1.2. Psychology of music preference

The psychology of music preference is the study of the psychological factors behind peoples' different music preferences.
Music can affect people in various ways from emotion regulation to cognitive development, along with providing a means for self-expression.
Numerous studies have been conducted to show that individual personality can have an effect on music preference.
The relationship between musical preference and personality has remained a long-standing topic of contention for researchers due to the variability in results and the low-predictive power that personality has historically demonstrated on music preferences.

1.3. Application and Requirements

Our idea is to determine clusters of people among candidates who answered to the Big 5 Personality Test, to create an application that will allow people with similar personality to know each other and to recommended songs most suited for their personality.
In particular, the recommendation of the songs will be made according to the music preferred by people with similar personality.
The application provides general information about each recommended song and their correlation with personality traits. Users will be able to provide a feedback to the songs recommended by the platform creating an intelligent recommendation system that will adapt to the preferences of the user.
The application will also recommend friends to the user and the user will be able to send them a friend request.
If the friend request will be accepted, the two users will be able to see each other’s information and start a conversation using email or phone numbers.
If two users become friends, the recommendation of songs between them will become stronger.
The application will also provide stats on the personality of the user and its music interests.

1.3.1. Functional Requirements

The users of the application will be divided into three categories: *Anonymous Users, Administrators*, which are allowed to provide new songs and limit accounts, and the *Standard Users*, who are allowed to use the main functionalities of the application.
Access to the application is guaranteed only through a login system using username and password, through which the user will be correctly identified.

A registration form will allow new users to register within the application as standard users. The registration will require the user to answer the survey, in such a way that the personality traits of the user will be determined.

Anonymous Users:

- Anonymous Users can register as Standard User and take the survey.
- Anonymous Users can log in as Standard User or Administrator.

Administrators:

- Administrators can search and delete users' accounts.
- Administrators can search and quarantine users' accounts, removing friends and friends requests of the user and isolating the user from the recommendation system.
- Administrators can search and delete songs.
- Administrators can add new songs into the application.
- Administrators can see stats of cluster and the recommendation system.

Standard Users:

- Standard Users can browse recommended songs.
- Standard Users can express preference to songs, using like and unlike button.
- Standard Users can see information of a songs, including comments and stats.
- Standard Users can comment songs.
- Standard Users can search for songs by name.
- Standard Users can browse recommended users.
- Standard Users can browse nearby users.
- Standard Users can update their recommended users' list, in case new users have been registered or the clustering algorithm has been re-executed.
- Standard Users can send friend requests to recommended users.
- Standard Users can browse friend requests.
- Standard Users can accept/decline incoming friend requests.
- Standard Users can browse friends and see their personal information.
- Standard Users can see their personality traits stats.
- Standard Users can see their music stats.
- Standard Users can edit their information.
- Standard Users can logout.

1.3.2. Non-Functional Requirements

- The application needs to provide low latency, high availability and *tolerance to single points of failures* (SPOF) and *network partitions*, for which the application has been designed in order to prefer the Availability (A) and Partition Protection (P) vertices of the CAP triangle. For this reason, the *Eventual Consistency* paradigm has been adopted.
- *Usability*. The application needs to be user friendly, providing a *GUI*.

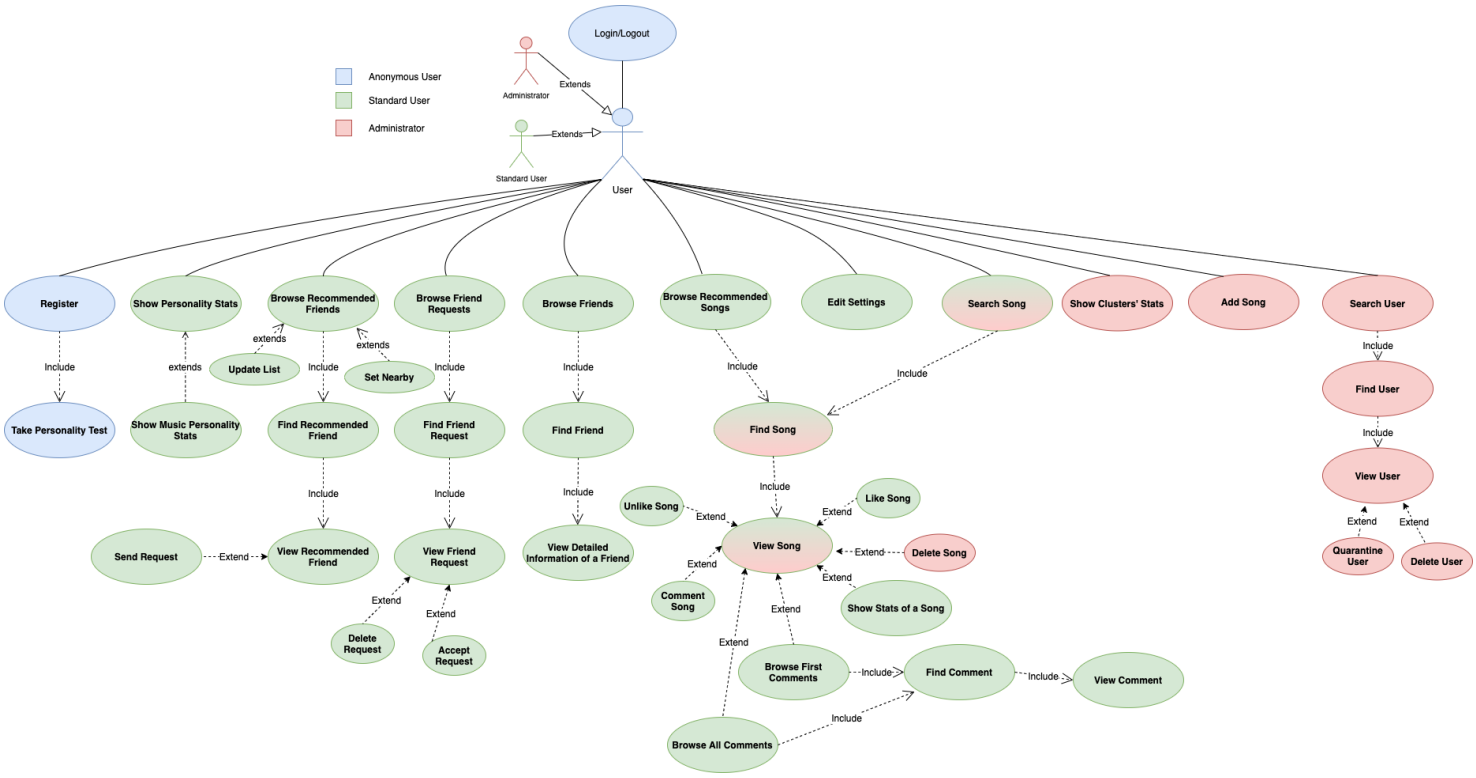
- *Privacy*. The application shall provide the security of users' credentials.
- *Scalability*. The system shall work on a non-prefixed number of cluster nodes.
- *Portability*. The system shall be environment independent.
- *Maintainability*. The code shall be readable and easy to maintain.

1.4. Specification

1.4.1. Actors and Use Case Diagram

Based on the software's functional requirements, we can distinguish three different actors:

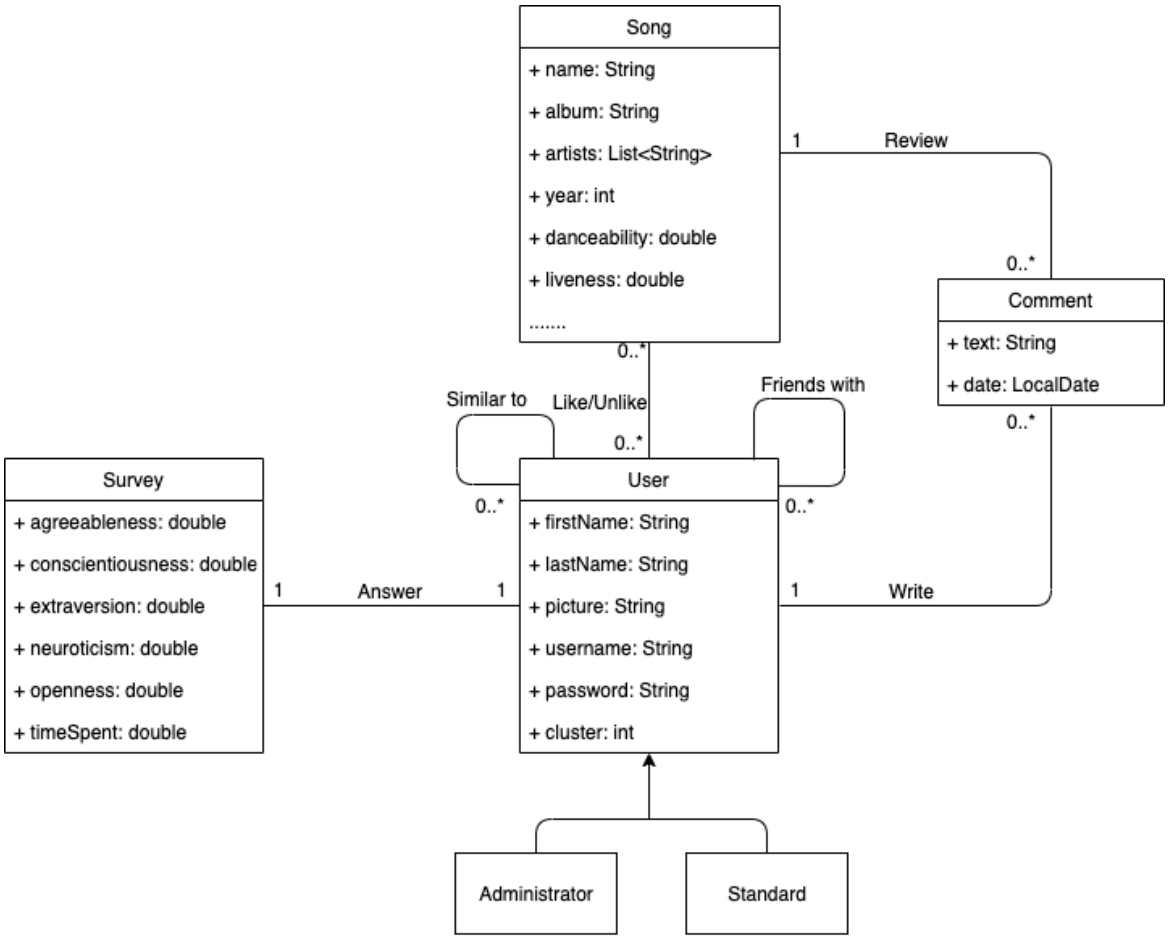
- *Anonymous Users*, who are only allowed to log in via username and password or register within the application and take the survey.
- *Standard Users*, who are allowed to interact with most of the application's features such as browse, search, express preference, see their information and comment recommended songs, browse and send a friend request to recommended users, browse friend requests and answer to them, browse friends and see their information, see stats and edit their personal information.
- *Administrators*, who are allowed to manage users and songs.



Use Case Diagram

1.4.2. Analysis Classes Diagram

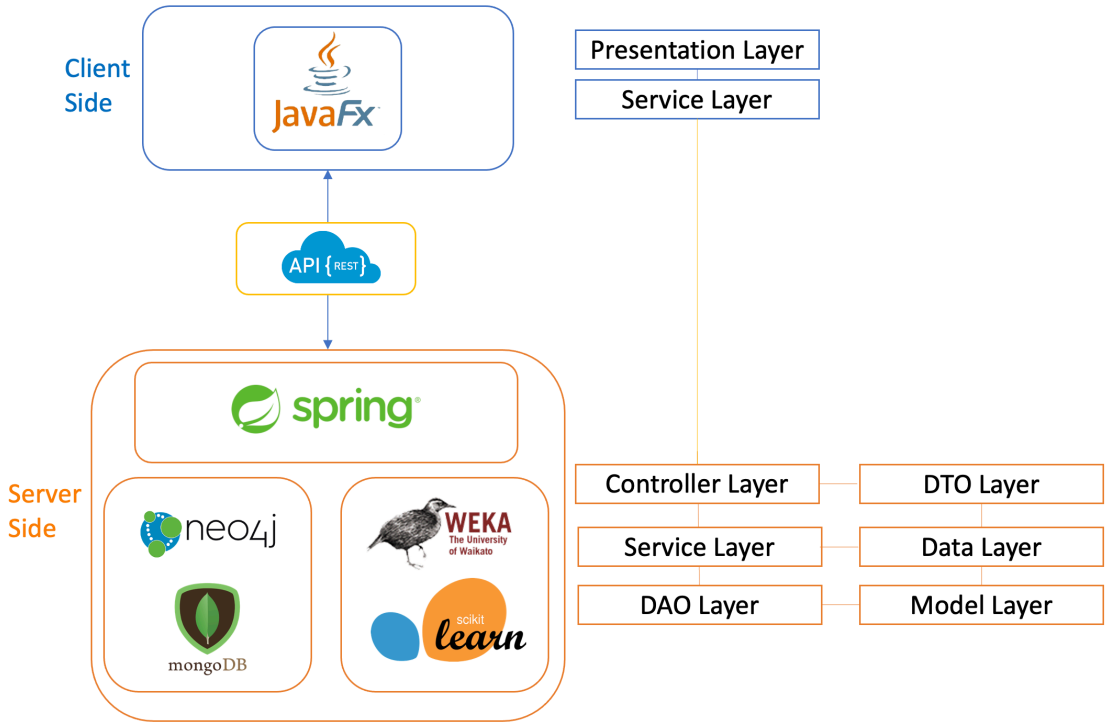
Based on the software's functional requirements, in the following diagram we can see the main entities of the application and the relationships among them.



Each user may send/receive one or more friend requests to/from other users.
One user has associated a survey and each survey is associated with a user.
Each survey is composed by the summarized value of personality traits.
One user can like/unlike zero or more songs and each song can be liked/unliked by one or more users.
Each user can be similar to zero or more users and each user can be friend with zero or more other users.
Each user can write zero or more comments and each comment is written by a user.
Each user can be a Standard User or an Administrator.
Each comment regards one song and each song can have zero or more comments.

1.4.3. System Architecture

The architectural pattern used for the design of the overall system is the client-server pattern. The client and the server interact with the HTTP protocol sending JSON data to each other that is then converted to Java DTO (Data Transfer Objects) classes.



System Architecture

1.4.4. Server Side

The server implements a REST API with the Spring Boot Java Framework.
Every functionality of the server is accessed using Uniform Resource Identifiers, also known as URIs (e.g. http://localhost:8080/users).
The server is divided into different layers, each of them enclosed in a specific package:

- The controller layer is in charge of handling a request from the moment when it is intercepted to the generation of the response and its transmission. This layer calls one or more service layer functions and manages the deserialization of the request and the serialization of the response through the DTO layer;
- The service layer encapsulates the business logic of the server. The service layer of this server also contains the functions needed for the clustering process;

- The DAO (Data Access Object) layer is responsible for encapsulating the details of he persistence layer and provide a CRUD interface for a single entity;
- The model layer contains the entities the represent the data stored in the databases;
- The DTO layer is used to decouple data representation to model objects;
- The data layer contains specific data structures needed for passing data from controllers to services and vice versa;

For the MongoDB access, Spring Data MongoDB was used, which provides integration with the MongoDB document database. Key functional areas of Spring Data MongoDB are a POJO (Plain Old Java Object) centric model for interacting with a MongoDB DBCollection and easily writing a Repository style data access layer.

The server side uses a cluster of 3 virtual machines made available by the University of Pisa, which are used to host a sharded MongoDB Cluster and a single instance of a Neo4j database.

	MongoDB Instances	**Neo4j Instances**
172.16.4.108	MongoDB Server (mongod)	
172.16.4.109	MongoDB Server (mongod)	
172.16.4.110	MongoDB Server (mongod)	Neo4j Server

1.4.5. Client Side

On the client side, we can find the Presentation Layer and the remaining part of the Service Layer. The Presentation Layer consists of a Graphical User Interface with which the users can interact. The portion of the Service Layer in the client, is in charge of processing requests coming from the user, interact with the server and provide the information requested.

1.4.6. Frameworks

The application code will be written in Java, using JavaFX and SceneBuilder for the GUI. Concerning the database management systems, Neo4j and MongoDB have been used. The server implements a REST API with the Spring Boot Java Framework. Apache Maven has been used as a tool for building and managing the whole application, while for version control, Git has been used.

1.4.7. Dataset Organization and Database Population

1.4.7.1. User Information

The dataset we used for applying the clustering algorithm comes from Kaggle (https://www.kaggle.com/tunguz/big-five-personality-test) and was collected through an interactive on-line personality test. In the dataset, 10 questions are available for each personality trait, with answers on a five point scale, labeled 1 = Disagree, 3 = Neutral and 5 = Agree. The corresponding time needed to answer each question is also present in ms. Since our application works as a social network, users are a relevant part of the application. We decided to trait each instance of the table as a user of our application, generating random information from https://randomuser.me.

Population of the user's database

Each user will present a similarity edge in our Graph Database with the K Nearest Neighbors inside its Cluster. We decided to use K = 10.

1.4.7.2. Song Information

The dataset we used for the songs comes from Kaggle (https://www.kaggle.com/rodolfofigueroa/spotify-12m-songs) and contains the audio features of 1.2M+ Spotify Songs obtained with the Spotify API.
Among the information for each song, we have: Name, Album, Artists, danceability and other interesting music features.

1.4.7.2. Preferences

Since we use two different datasets which do not interact with each other and since we need starting preferences to make our application work, we decided to use an heuristic to allocate the first edges regarding preferences of the users. To avoid to allocate random edges, we exploited the following correlation matrix, obtained from a paper regarding the correlation among the big 5 personality traits and music streaming behavior of Spotify Users: https://dl.acm.org/doi/pdf/10.1145/3468784.3469854.

Table 2: Results of the Correlation Analysis.

	Openness to Experience	Conscientiousness	Extraversion	Agreeableness	Neuroticism
Acousticness	.076	.112	**.405***	.304	-.128
Danceability	-.056	.240	.102	-.006	-.244
Energy	-.215	-.230	**-.475****	-.287	.065
Instrumentalness	.002	-.253	.140	-.028	**.479****
Liveness	**.376***	.198	-.217	.096	.004
Speechiness	.120	.013	**-.456****	-.182	-.015
Valence	.022	-.029	-.122	.020	-.092
Tempo	-.190	-.192	-.329	**-.452****	.070

**. Correlation is significant at the 0.01 level (2-tailed).
*. Correlation is significant at the 0.05 level (2-tailed).

Correlation Matrix

In particular, starting from the predominant personality trait of each user, we decided to allocate preferences to songs that have the highest or lowest values in the song's feature mostly correlated with that trait, respectively using like and unlike relationships.

```java
public void addLikesHeuristic() {
    List<String[]> csvRelationships = new ArrayList<>();
    List<MongoUser> mongoUsers = customUserRepository.findAllWithSurveyAndCluster();
    List<MongoUser> extroverseUsers = new ArrayList<>();
    List<MongoUser> agreeableUsers = new ArrayList<>();
    List<MongoUser> conscentiousUsers = new ArrayList<>();
    List<MongoUser> neuroticUsers = new ArrayList<>();
    List<MongoUser> openUsers = new ArrayList<>();
    for (MongoUser mongoUser: mongoUsers) {
        switch (getBiggestPersonalityTrait(mongoUser)) {
            case 0 -> extroverseUsers.add(mongoUser);
            case 1 -> agreeableUsers.add(mongoUser);
            case 2 -> conscentiousUsers.add(mongoUser);
            case 3 -> neuroticUsers.add(mongoUser);
            case 4 -> openUsers.add(mongoUser);
        }
    }
    addRelationships(openUsers, type: "liveness", csvRelationships, Sort.Direction.DESC, Sort.Direction.ASC);
    addRelationships(conscentiousUsers, type: "instrumentalness", csvRelationships, Sort.Direction.ASC, Sort.Direction.DESC);
    addRelationships(extroverseUsers, type: "energy", csvRelationships, Sort.Direction.ASC, Sort.Direction.DESC);
    addRelationships(agreeableUsers, type: "tempo", csvRelationships, Sort.Direction.ASC, Sort.Direction.DESC);
    addRelationships(neuroticUsers, type: "instrumentalness", csvRelationships, Sort.Direction.ASC, Sort.Direction.DESC);
    File csvOutputFile = new File(CSV_FILE_NAME_LIKES);
    try (PrintWriter pw = new PrintWriter(csvOutputFile)) {
        csvRelationships.stream()
                .map(this::convertToCSV)
                .forEach(pw::println);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

Script for the heuristic regarding the allocation of preferences

1.4.7.3. Comments

For the same reason, we decided to add random comments from some of the users to some of the songs of our application, starting from a set of predefined comments, typical for songs' applications.

```java
public void generateComments(){

    Random random = new Random();

    String[] comments = {"Nice song.", "Really dope!", "Awesome song!", "Best song!", "Wow, this sucks.", "Really amazing song!",
            "Really bad song.", "This song couldn't be worse", "The first album was better.", "The guitarist is not good at all",
            "Awesome sound", "Really chill", "Average song.", "Nothing special.", "This song could have been better.",
            "Really disappointed", "I love Taylor Swift"};

    List<MongoUser> users = customUserRepository.findToGenerateComments();
    List<MongoSong> songs = customSongRepository.findToGenerateComments();
    List<Comment> toBeCreated = new ArrayList<>();
    List<MongoSong> toBeUpdated = new ArrayList<>();

    for (MongoSong song: songs) {

        for (int i = 0; i < 15; i++) {
            MongoUser user = users.get(random.nextInt(users.size()));
            String text = comments[random.nextInt(comments.length)];
            toBeCreated.add(createComment(user, text, song));
        }
        List<Comment> commentList = customCommentRepository.bulkInsertComments(toBeCreated);
        song.setComments(new ArrayList<>());

        for (int i = 0; i < 10; i++) {
            song.addComment(createCommentSubset(commentList.get(i)));
        }

        toBeUpdated.add(song);
        toBeCreated.clear();
    }
```

Script for initial random comments

2. NoSQL Databases

We decided to use a Document Database, with MongoDB as DBMS, to store all main information about the entities of our application.
We also decided to exploit a Graph Database with Neo4J as DBMS to store information about friendships and similarity among users, but also for storing the songs' preferences of our users.

2.1. MongoDB Design

Information about songs, users, survey, clusters and comments are stored in MongoDB in three different collections.

2.1.1. User Collection

The document of a user contains all the personal information of the user and the summarized results of the survey. We decided to consider only this summarization since after the survey, the only information we need are the average values of the personality traits and the time spent to answer the test. Since the only information we have about a cluster is the ID and the users contained, we decided to represent this information inside our document using the cluster field.


```
_id: ObjectId("61e91bc613c1e6256f65bf79")
first_name: "Blake"
last_name: "Sirko"
date_of_birth: 1977-06-18T22:00:00.000+00:00
gender: "male"
country: "Albania"
username: "blackbutterfly1630"
phone: "480-613-8037"
email: "blake.sirko2@example.com"
password: "$2a$05$pFKKJhMFsumUlQOIaPQydumjMHaOAC3i/cUTg6seGmIHgKQquHFua"
registration_date: 2004-01-17T23:00:00.000+00:00
picture: "https://randomuser.me/api/portraits/men/91.jpg"
extraversion: 2.4
agreeableness: 2.1
conscientiousness: 3.3
neuroticism: 3.4
openness: 3.1
time_spent: 3.86
cluster: 1
```


User's Collection Example

This collection is also used at the login, for checking if the credentials are correct.
To satisfy the privacy non-functional requirement, the password is encrypted using BCrypt hashing.

2.1.2. Song Collection

The document of a song contains all the general information and features of a song and also an array of embedded documents describing the comments created by the users. It has been decided to embed the comment document inside the song's collection due to the one-to-many relationship between these two entities. In this way, each time a new song is displayed, users will be able to see the comments of other users to the song. We decided to embed a subset of information of the user, allowing the general information of a comment to be retrieved with a single read operation and avoiding join operations, at the cost of an additional small redundancy within the database. In fact, for each comment we need to be able to see the name and surname of the user who wrote it.


```
_id: ObjectId("61e87d0113c1e6256f575fac")
name: "Flypaper IV"
album: "Flypaper"
artists: Array
  0: "Oren Ambarchi"
  1: " Keith Rowe"
track_number: 4
disc_number: 1
explicit: false
danceability: 0.119
energy: 0.461
key: 2
loudness: -10.215
mode: 1
speechiness: 0.068
acousticness: 0.308
instrumentalness: 0.892
liveness: 0.106
valence: 0.0306
tempo: 90
duration: 902560
time_signature: 4
year: 2002
comments: Array
  0: Object
    comment_id: ObjectId("61ebc3c61344522b02fc7c8d")
    user_id: ObjectId("61e91bcb13c1e6256f65ea80")
    name: "Rodrigo"
    surname: "Santana"
    text: "Awesome sound"
    date: 2022-01-21T23:00:00.000+00:00
cluster: 1
likes: Array
  0: Object
    cluster: 1
    numLikes: 320
    numUnlikes: 0
  1: Object
    cluster: 2
    numLikes: 63
    numUnlikes: 0
  2: Object
    cluster: 3
    numLikes: 135
    numUnlikes: 0
  3: Object
    cluster: 4
    numLikes: 62
    numUnlikes: 0
  4: Object
    cluster: 5
    numLikes: 104
    numUnlikes: 0
```


Song's Collection Example

Regarding comments, a potential problem with document embedding is that it can lead to large documents and overcome the limitation size imposed by

MongoDB (16MB). This situation may be encountered, for example, in front of a song that everyone talks about.
For this reason, we used the subset pattern to only store a subset of the comments, instead of the entire set of embedded data. We embedded the 10 most recent comments and we decided to store all the comments in another collection, since we will show them in the interface only if the user will require it, using the 'Show All Comments' button.
We can notice also the introduction of a redundancy for each song document, which is an array of cluster's number of likes and unlikes.
This redundancy will be useful for improving the performances of almost all our analytics but also to determine the total number of likes and unlikes of the song, that can be calculated as the sum of all these values, avoiding other two redundant fields in our document.
We also decided to use a cluster field, representing the id of the most predominant cluster for that song, which is the cluster for which we have the highest difference between number of likes and unlikes.
Using this field will allow us to perform more efficiently most our aggregations and also, as we will see, to implement a good strategy of sharding. Of course, each time one of the users will give new preference, we will need to update our redundancies, updating the number of likes and unlikes for its cluster. We will also need to check if the predominant cluster has changed, but for how our recommendation system works, once the cluster has become predominant for a song, that song will be strongly recommended among users of that cluster, increasing the probability to have more likes from that cluster than others.

2.1.3. Comment Collection

If a user wants to see additional documents, the application will use the `comment's` collection, in which we will find all general information of a comment, again with a small redundancy to avoid join operations.

```
_id: ObjectId("61ebc3c61344522b02fc7c8a")
user_id: ObjectId("61e91c0213c1e6256f678582")
song_id: ObjectId("61e87d0113c1e6256f575fac")
name: "Matthew"
surname: "Brown"
text: "Really bad song."
date: 2022-01-21T23:00:00.000+00:00
```

Comment's Collection Example

This collection will also store the most recent 10 comments which are already contained in the song, in such a way that if a song has more than 10 comments, each time a new comment arrives we don't need to move the oldest comment among the previous 10 in this collection. This will improve our performances, with the cost of another small redundancy.

2.1.4. Queries Analysis

Starting from the analysis of the use case, we determined the following read and write queries involving our Document Database. Each operation is presented with its expected frequency and cost.

Read Operations		
Operation	**Expected Frequency**	**Cost**
Show User Information	High	Low (1 Read)
Show Song Information	High	Low (1 Read)
Show First Song Comments	High	Low (1 Read)
Show All Song Comments	Low	High (Multiple Reads)
Show Personality Stats	Low/Medium	Low (1 Read)
Show Cluster's Personality Stats	Low/Medium	High (Aggregation)
Show Cluster Deviation	Low/Medium	High (Aggregation)
Show Cluster's Music Stats	Low	High (Aggregation)
Search Song by Name	Medium	Low (1 Read)
Validate User Login Attempt	High	Low (1 Read)
Retrieve User Information at Login	High	Low (1 Read)

Write Operations		
Operation	**Expected Frequency**	**Cost**
Update User Information	Very Low	Medium (Multiple attributes write)
Registration of a New User	Low	Medium (Insertion of a Document)
Update Song's Cluster's likes/unlikes	High	Medium (2 Attribute Writes)
Add a Comment	Low	Low (Document Insertion)

Admin Read Operations		
Operation	**Expected Frequency**	**Cost**
Search User By Username	Low	Low
Search Song By Name	Low	Low
Show Cluster's Stats	Very Low	High (Aggregation)
Show Country's Stats	Very Low	High (Aggregation)

Admin Write Operations		
Operation	**Expected Frequency**	**Cost**
Delete User	Very Low	Medium (Document deletion)
Add Song	Low/Medium	Medium (Document Insertion)
Delete Song	Very Low	Medium (Document deletion)

From the query analysis we can conclude that document database's queries are predominantly read intensive, while write operations, except for the updates of likes and unlikes, are less frequent and with a low cost.
For this reason, we can confirm our privilege to response time and availability requirements, choosing the following combination of concern settings:

- **Write Concern Type**: W1, waiting for an acknowledgment from a single member. It allows the system to be as fast as possible during write operations, introducing the need to implement some kind of eventual consistency paradigm.
- **Write Timeout**: 0, because at least one write operation should be performed.
- **Read Concern**: Nearest, in such a way that read operations are performed on the nearest node, considering responsiveness, which is measured in pings.

2.1.5. CRUD Operations

In this section, we can find CRUD Operations implemented with the interface MongoTemplate of the Spring Data MongoDB.

2.1.5.1. Creation

Operation	
Create Song	mongoTemplate.insert(song);
Create User	mongoTemplate.insert(user);
Create Comment	mongoTemplate.insert(comment);

2.1.5.2. Reading

Operation	
Find Song by Id	mongoTemplate.findById(id, MongoSong.class);
Find User by Id	mongoTemplate.findById(id, MongoUser.class);
Find User by Email	mongoTemplate.findOne(Query.*query*(Criteria.*where*("email").is(email)), MongoUser.class);

Find Songs Sort by Field	Query query = new Query();
	query.with(Sort.*by*(direction, field));
	query.fields().include("id");
	query.limit(1000);
	mongoTemplate.find(query, MongoSong.class);

Get Song's Comments	Query query = new Query();
	query.addCriteria(Criteria.*where*("song_id").is(id));
	query.with(Sort.*by*("date").descending().and(Sort.*by*("id").descending()));
	mongoTemplate.find(query, Comment.class);

Operation	

Get Songs with Name starting with	Query query = new Query();
	query.addCriteria(Criteria.*where*("name").regex("^" + name));
	query.with(Sort.*by*(Sort.Direction.*ASC*, "name"));
	query.fields().include("name").include("album").include("artists");
	mongoTemplate.find(query, MongoSong.class);

Get Users with Username starting with	Query query = new Query();
	query.addCriteria(Criteria.*where*("username").regex("^" + username));
	query.with(Sort.*by*(Sort.Direction.*ASC*, "username"));
	query.fields().include("first_name").include("last_name").include("picture");
	mongoTemplate.find(query, MongoUser.class);

2.1.5.3. Update

Operation	

Update Comment	Update update = new Update();
	update.set("comments", song.getComments());
	mongoTemplate.updateFirst(Query.*query*(Criteria.*where*("id").is(song.getId())), update, MongoSong.class).getModifiedCount();

Update Likes	Update update = new Update();
	update.set("likes", song.getLikes());
	update.set("cluster", song.getCluster());
	mongoTemplate.updateFirst(Query.*query*(Criteria.*where*("id").is(song.getId())), update, MongoSong.class).wasAcknowledged();

Operation	

Update User Information	Update update = new Update();
	for (Map.Entry<String, String> field:
	toBeUpdated.entrySet()) {
	  update.set(field.getKey(), field.getValue());
	}
	mongoTemplate.updateFirst(Query.*query*(Criteria.*where*("id").is(id)), update,
	MongoUser.class).wasAcknowledged();

Update Cluster	Query query = new Query(Criteria.*where*("id").is(user.getId())); Update update = new Update(); update.set("cluster", user.getCluster()); mongoTemplate.updateFirst(query, update, MongoUser.class).wasAcknowledged();

2.1.5.4. Delete

Operation	
Delete Song by Id	Query query = new Query(); query.addCriteria(Criteria.*where*("id").is(id)); mongoTemplate.remove(query, MongoSong.class).wasAcknowledged();
Delete User by Id	mongoTemplate.remove(Query.*query*(Criteria.*where*("id").is(id)), MongoUser.class).wasAcknowledged();
Delete Song's Comments	Query query = new Query(); query.addCriteria(Criteria.*where*("song_id").is(id)); mongoTemplate.remove(query, Comment.class).wasAcknowledged();
Delete User's Comments	Query query = new Query(); query.addCriteria(Criteria.*where*("user_id").is(id)); mongoTemplate.remove(query, Comment.class).wasAcknowledged();

2.1.6. Analytics and Statistics

Different aggregations for analytics and statistics involving our Document Database are described in this section.

2.1.6.1. Cluster with the highest variation

The following aggregation returns the cluster with the highest variation, which we defined as the sum of differences among the minimum and maximum in the range of each feature.

```
{$group: {
    _id: '$cluster',
    AGRMax: { $max: '$agreeableness' },
    AGRMin: { $min: '$agreeableness' },
   OPNMax: { $max: '$openness' },
    OPNMin: { $min: '$openness' },
    CSNMax: { $max: '$conscientiousness' },
    CSNMin: { $min: '$conscientiousness' },
    EXTMax: { $max: '$extraversion' },
    EXTMin: { $min: '$extraversion' },
    ESTMax: { $max: '$neuroticism' },
    ESTMin: { $min: '$neuroticism' },
    TimeSpentMax: { $max: '$time_spent'},
    TimeSpentMin: { $min: '$time_spent'}}},
   {$project: {
    differenceAGR: { $subtract: ['$AGRMax','$AGRMin']},
    differenceOPN: { $subtract: ['$OPNMax','$OPNMin']},
    differenceCSN: { $subtract: ['$CSNMax','$CSNMin']},
    differenceEXT: { $subtract: ['$EXTMax','$EXTMin']},
    differenceEST: { $subtract: ['$ESTMax','$ESTMin']},
    differenceTS: { $subtract: ['$TimeSpentMax','$TimeSpentMin']}
   }},
   {$project: {
    difference: {
     $sum: [
      '$differenceAGR',
      '$differenceCSN',
      '$differenceEXT',
      '$differenceEST',
      '$differenceOPN',
      '$differenceTS'
     ]
    }
   }},
   {$sort: {difference: -1}},
   {$limit: 1}]
```

2.1.6.2. Top K Countries with the highest average of personality traits

The following aggregation returns the top K countries with the highest average values of personality traits, giving information about the countries of the citizens that used higher scores in our survey.

```
[{$group: { _id: '$country',
 AGR: { $avg: '$agreeableness' },
 OPN: { $avg: '$openness'},
 CSN: { $avg: '$conscientiousness' },
 EXT: { $avg: '$extraversion' },
 EST: { $avg: '$neuroticism' }
}},
{$project: { cluster: 1, avg: { $avg: [ '$AGR', '$EST', '$EST', '$CSN', '$OPN' ]}}},
{$sort: {avg: -1}},
{$limit: K},
{$project: {cluster: 1}}]
```

2.1.6.3. Top K Albums with the highest preferences inside a cluster

The following aggregation returns the top K Albums in which a given cluster is predominant, taking in consideration the difference between the number of likes and the number of unlikes.

```
[{$match: { cluster: $id}},
{$unwind: { path: '$likes'}},
{$project: {
 album: 1,
 likes: 1,
 result: {
  $eq: ['$cluster','$likes.cluster']
 }
}},
{$match: {result: true}},
{$project: {
 album: 1,
 algebricSum: {
  $subtract: ['$likes.numLikes', '$likes.numUnlikes']}
}},
{$group: {
 _id: '$album',
 strength: {$sum: '$algebricSum'}
}},
{$sort: {strength: -1}},
{$limit: K}]
```

2.1.6.4. Most Danceable Cluster

The following aggregation returns the most predominant cluster among the songs with the highest danceability.

```
[{$sort: {danceability: -1}},
 {$limit: 500},
 {$group: {
 _id: '$cluster',
 strength: {$sum: 1}
}},
{$sort: {strength: -1}},
 {$limit: 1},
 {$project: {
 _id: 1
}}]
```

2.1.6.5. Cluster Personality Average Characteristic

The following aggregation returns the average personality characteristics for each cluster.

```
[{
$group: { _id: "$cluster",
  AGR: { "$avg": "$agreeableness" },
  OPN: { "$avg": "$openness"},
  CSN: { "$avg": "$conscientiousness"},
  EXT: { "$avg": "$extraversion"},
  EST: { "$avg": "$neuroticism"},
  TimeSpent: { "$avg": "$time_spent"}
}]
```

2.1.6.6. Average Music Characteristics of the Songs Associated with a Cluster.

The following aggregation returns the average music features of the songs in which a cluster is predominant, for each cluster.

```
[{
   $group: { _id: '$cluster',
        danceability: { $avg: '$danceability'},
        acousticness: { $avg: '$acousticness' },
        energy: { $avg: '$energy' },
        instrumentalness: { $avg: '$instrumentalness' },
        liveness: { $avg: '$liveness' },
        valence: { $avg: '$valence' }
    }
}]
```

2.1.7. Indexes

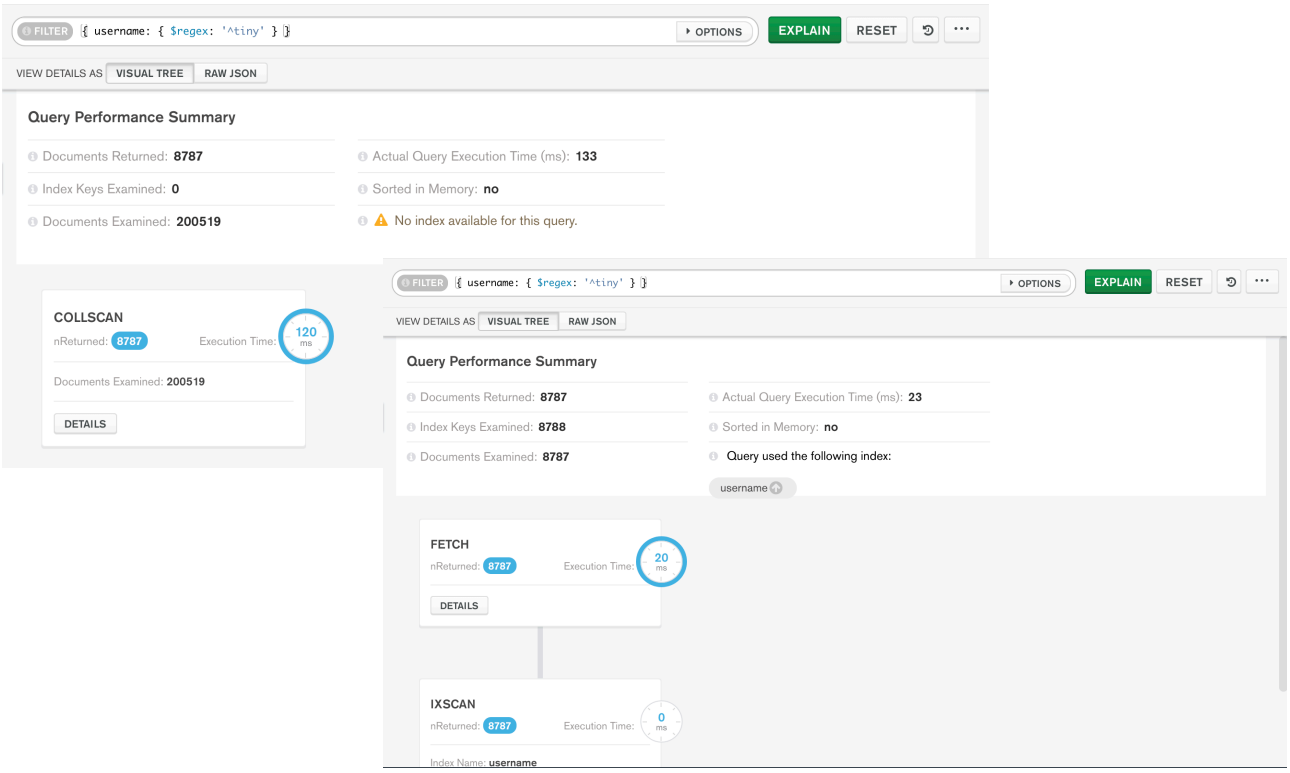
In order to improve the read operations' performance, the following indexes are introduced:

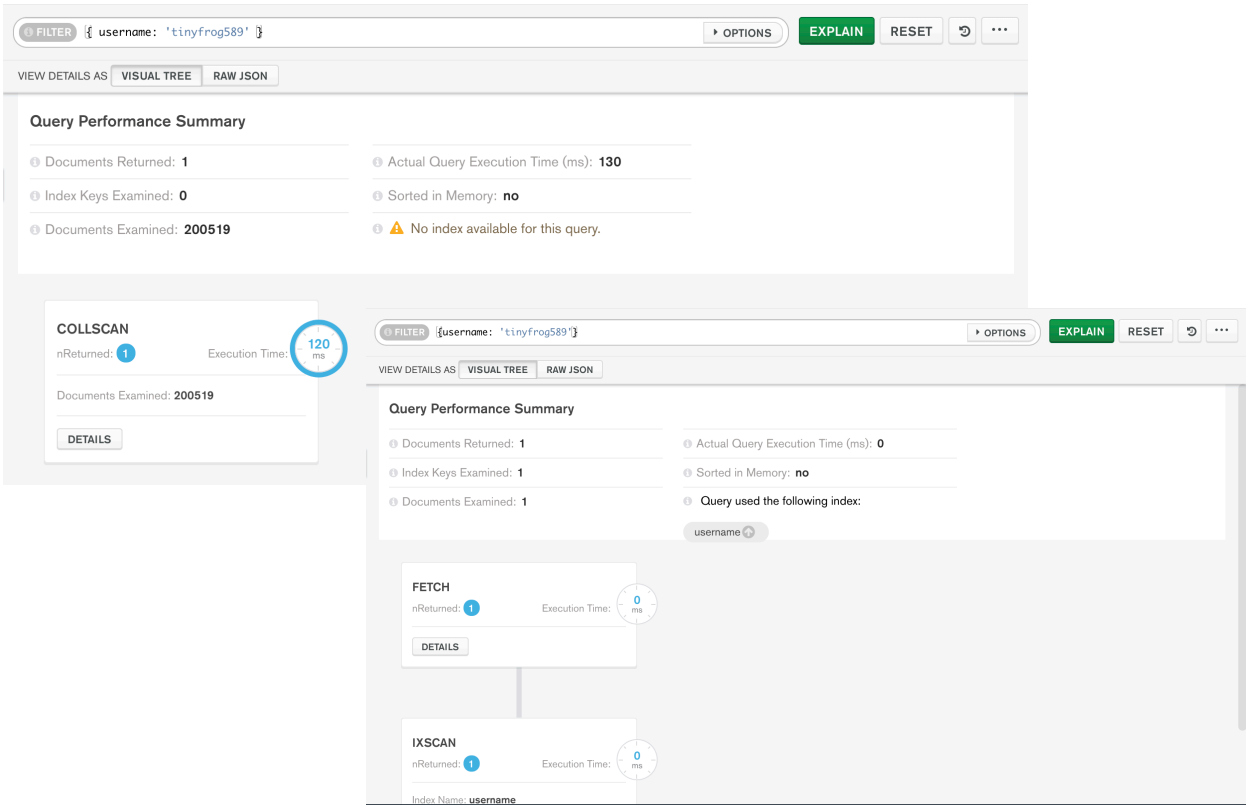
Index Type	Collection	Index Name	Index Field
Hashed Index (default)	User	id	_id
Simple Index	User	Username	username
Simple Index	User	Email	email
Simple Index	Song	Name	name
Simple Index	Comment	SongID	song_id

2.1.7.1 Indexes Performance Analysis

2.1.7.1.1. Username Index

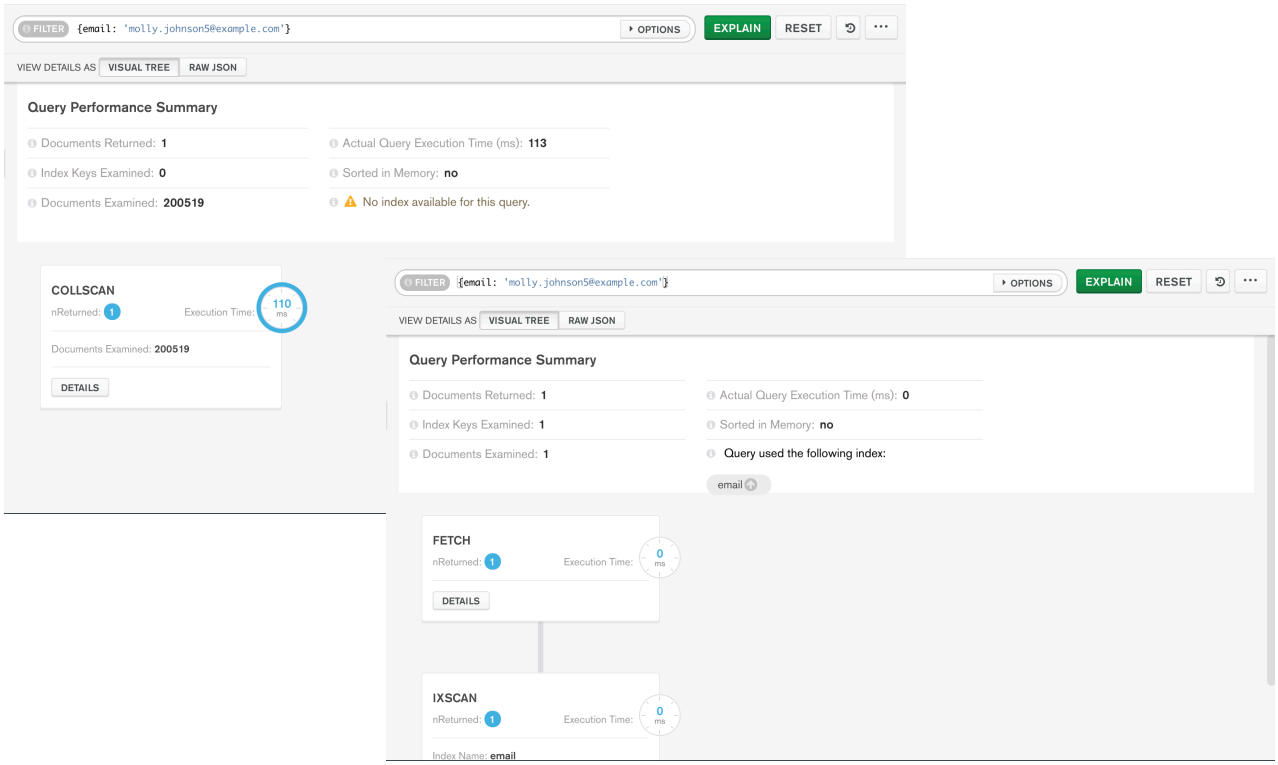
In this performance analysis we can see the improvements on equality matches using the username Index.
Using the index, only documents that match the value or regular expression are examined, with an execution time that strongly decreases.

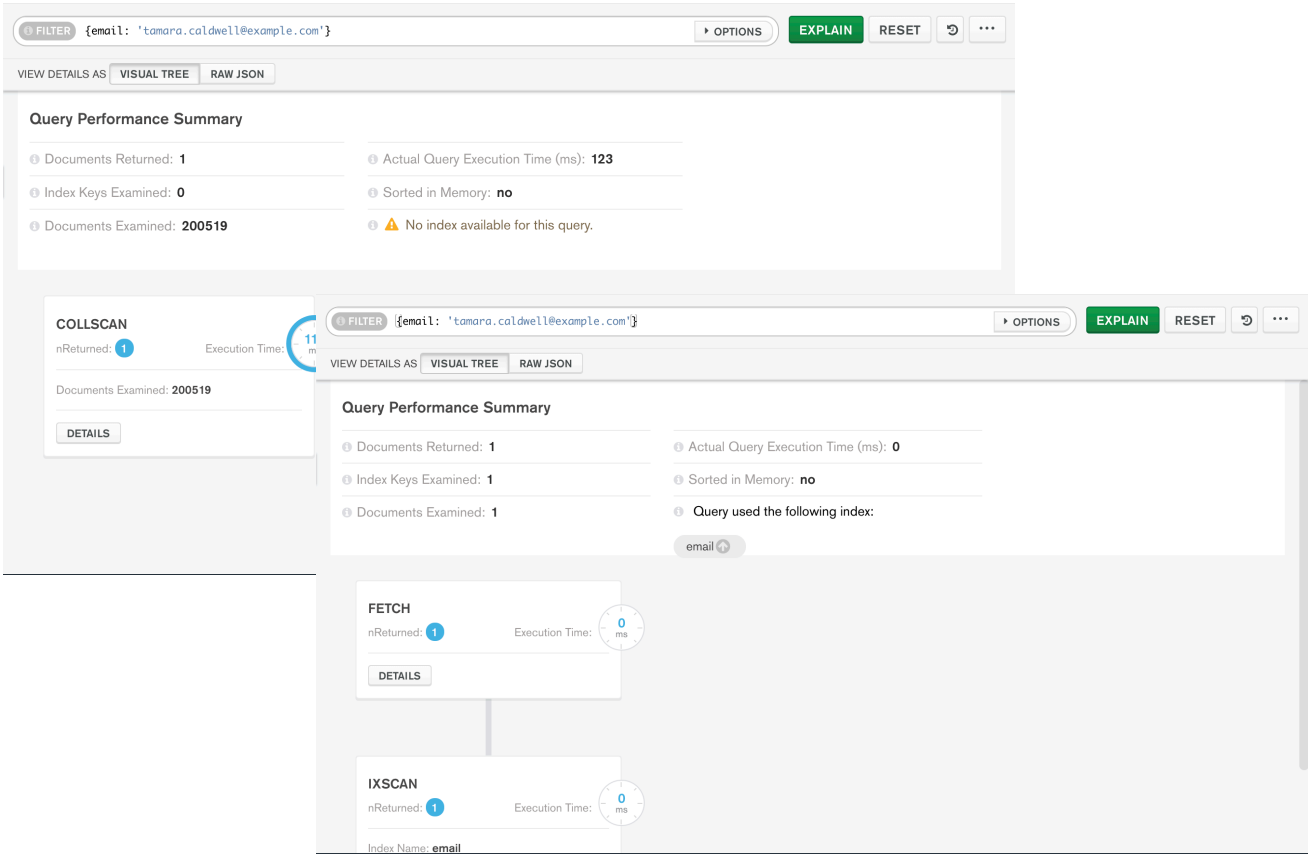




2.1.7.1.2. Email Index

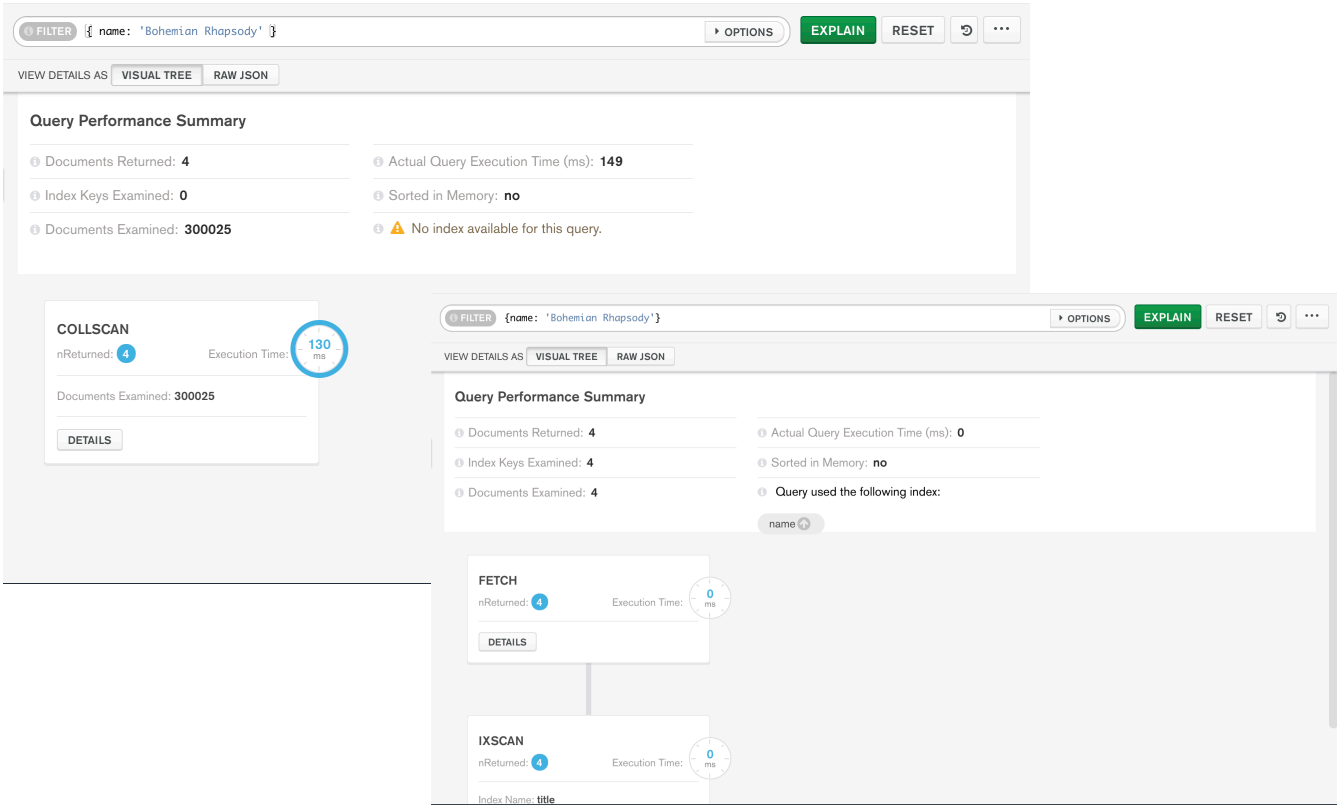
In this performance analysis we can see the improvements on equality matches using the email index.

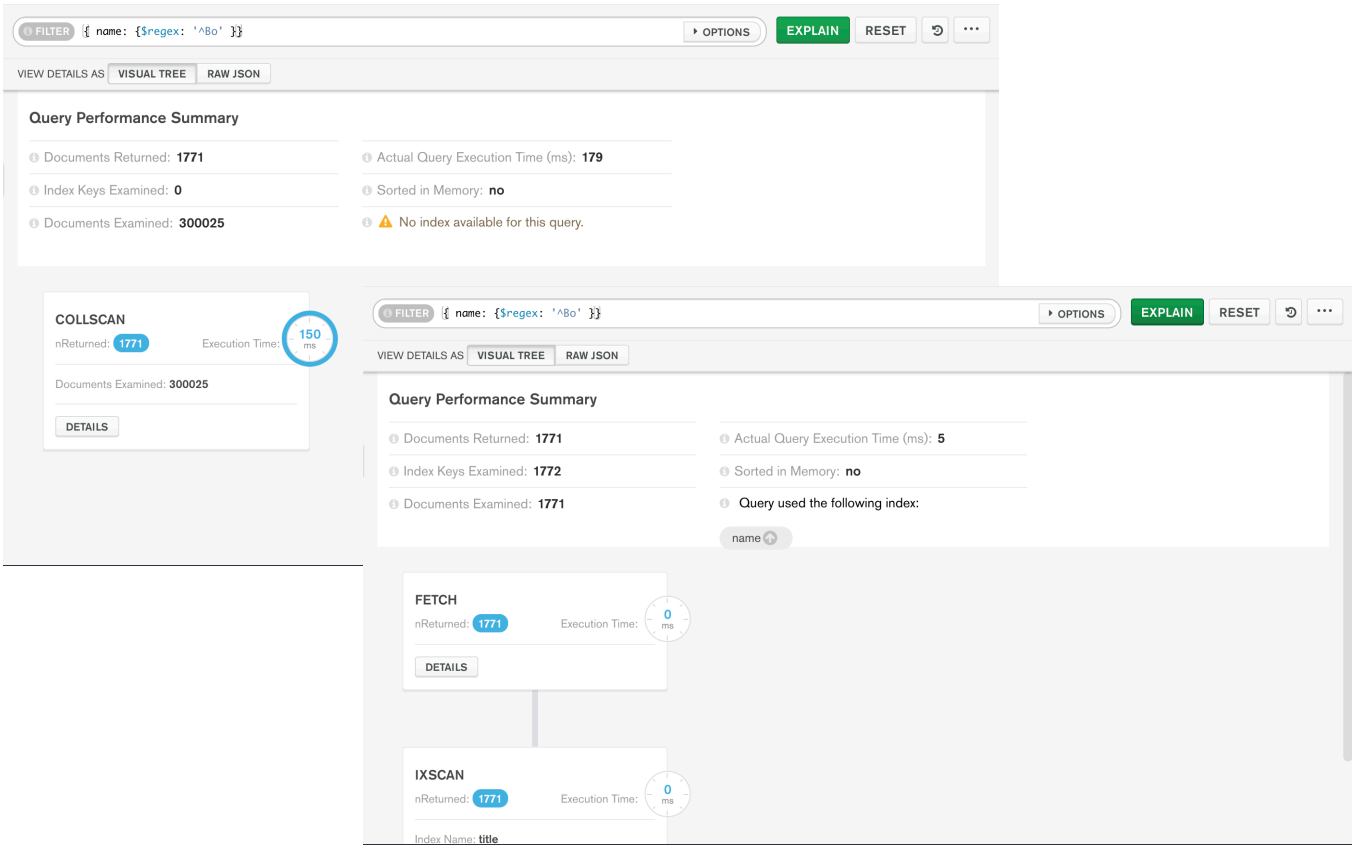




2.1.7.1.3. Song Name Index

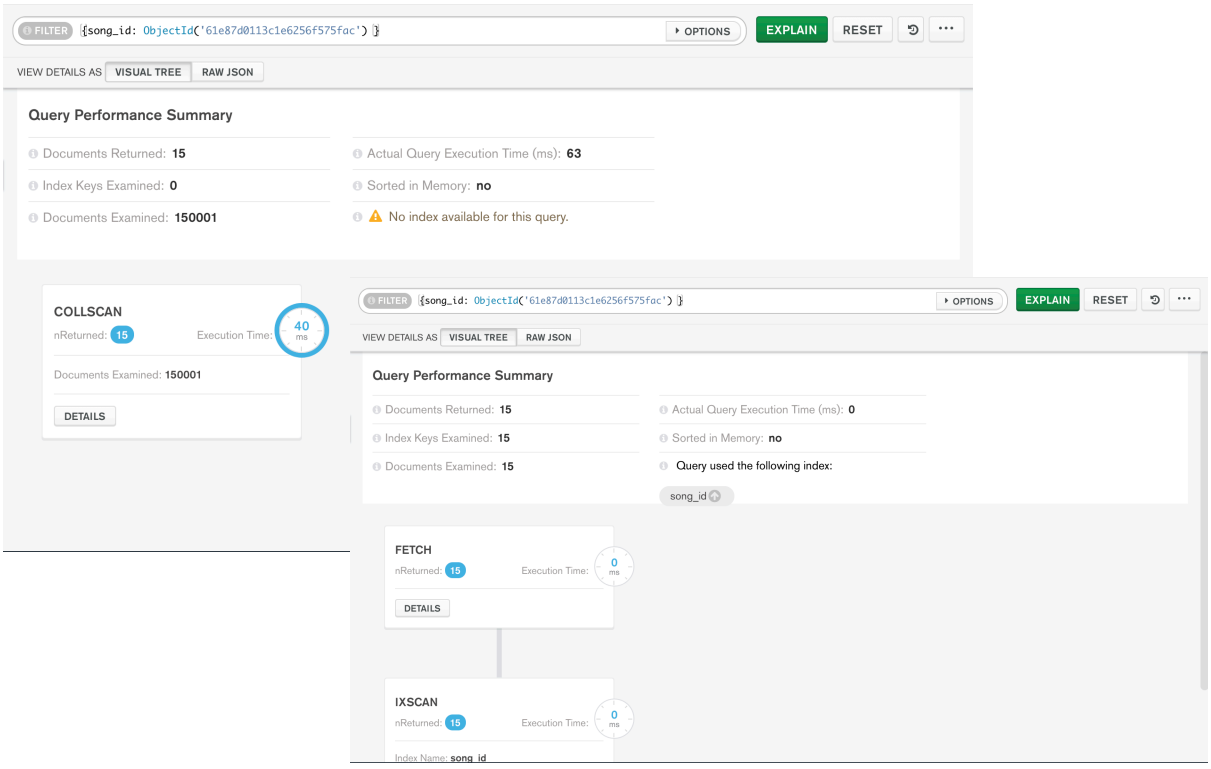
In this performance analysis we can see the improvements on equality matches using the song name index.





2.1.7.1.4. Song ID Index

In this performance analysis we can see the improvements on equality matches using the song ID index.



2.1.8. Horizontal Partitioning

We've chosen the most appropriate shard key considering:
1. The distribution of reads and writes.
2. The size of the chunks.
3. The number of chunks each query uses.

2.1.8.1. User Collection

As possible candidates for the sharding key in the User Collection we considered the *_id*, the *country* and the *cluster* fields.
Considering only the monotonic _id we would always be adding documents to the last replica set and supposing that, as typically happens, users who subscribed recently to the application are the ones who uses it the most, we would always involve the last replica set for reading operations.
We would also have a high write-lock-% ending in slow write operations.
Also, analytics would involve, basically always, multiple servers.
Given the fact that our personality clusters are quite balanced in terms of number of users and given the fact that all statistics a user can ask for, regard its cluster, the cluster field could be an optimal shard key.
Analytics queries of a user would always use the same server, which have information of its cluster.
Given that our clusters are also balanced in terms of number of users, we will also avoid to overload just few servers.
In our clustering algorithm, the optimal value of K was 5, obtaining 5 possible values for this field and, as we know, if we have too many documents with the same shard key we may end up with jumbo chunks.
Considering that we want to achieve that most queries hit as few shards as possible and we want pretty balanced chunks, the compound index {cluster, _id} is the best solution for the User collection.
The addition of the _id field will break large clusters into multiple chunks, which will still be adjacent in index, and so most likely to end up on the same shard.
Also, we have that most recent registered users of the same cluster are typically the ones who uses the application the most, and so they are more likely to be recommended, to become friends and see each other's personal information, involving read operations on the same shard.

2.1.8.2. Song Collection

Noticing that each song will be recommended and though probably preferred by people of one particular cluster, and exploiting the cluster redundant field, we can imagine to create also clusters of songs. In particular, a song will belong to the most predominant cluster, the one which have the highest difference between the number of likes and unlikes.

We decided, though, to use the same combination of fields {cluster, _id} for the songs as well.
In fact, considering our analytics available for the user, a user can ask for statistics involving songs in which its cluster is predominant, involving documents present in just one server.
If the number of users or songs inside a cluster would grow too much, documents of one cluster could not be stored anymore in the same server, requiring read operations from multiple servers for our analytics. To solve this problem, we could also think to limit our analytics only to the most recent users added to the Cluster, in such a way to limit the analytics just on documents stored in one server.
Exploiting the _id in the shard key we will also have that songs added in the same period will be stored into the same server.

2.1.8.3. Comment Collection

The best sharding approach for the comment's collection is to shard comments associated with a song into the same shard of the song.

2.2. Neo4J Design

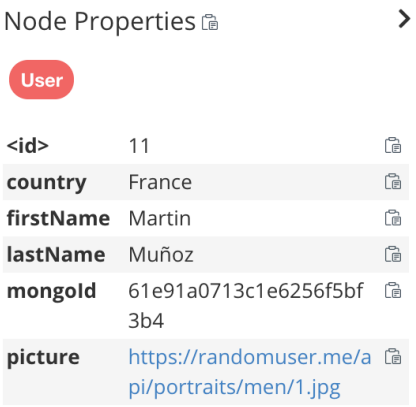
We used Neo4j to store information about friendships and similarity among users and preferences of users on the songs.

2.2.1. Nodes

We used a node for each main entity of our application, in particular for songs and users. Among the properties, we just have essential information needed to display previews of the entity in the application.
In order to maintain a connection between the same user in both databases a field mongoId containing the MongoDB ObjectId was added to Neo4j.

2.2.1.1. User Node

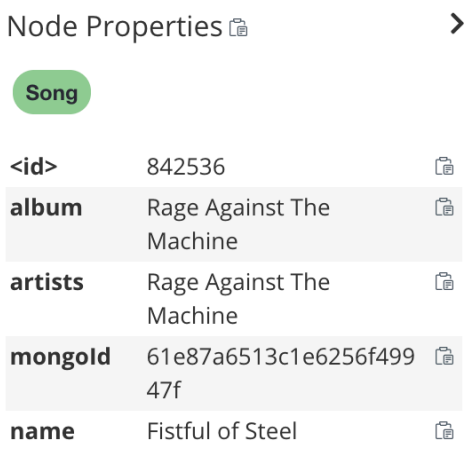
The user node will maintain as properties just the essential information to show a preview of the user.
In fact, before creating a friendship, a user can just see essential information about recommended users, and only once a friendship request is accepted, the user can see the other user's information.
Storing also the country of a user we are able to realize the 'Show Nearby Users' query without accessing into the Document DB.



User Node Properties Example

2.2.1.2. Song Node

The song node will store as properties just the essential information to show a preview of the song to the user.
Once interested, the user will be able to access the document database to obtain more information.



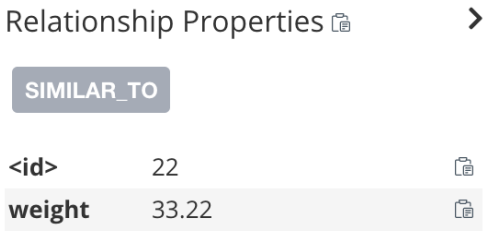
Song Node Properties Example

2.2.2. Relations

2.2.2.1. Similarity

The directed relation SIMILAR_TO will connect each user to the K Nearest Neighbors on its cluster.
Among all people with similar personality in fact, these users will be the ones expected to be more similar to the user.
The similarity edge will maintain as property the weight, which is determined as the inverse of the square of the distance of the personality traits' features and it will be used in the recommendation of users and songs.

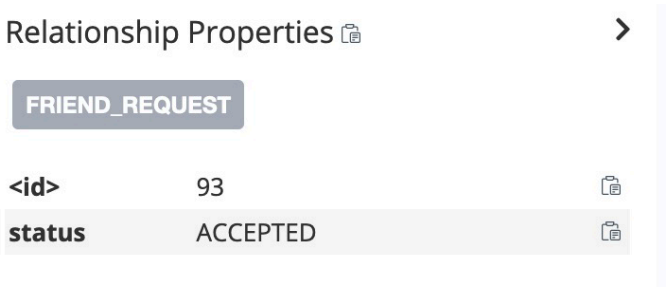
$$w_i = \frac{1}{dist^2(x,y)} \qquad dist(x,y) = \sqrt{\sum_{i=1}^{6}(personality_x - personality_y)^2}$$



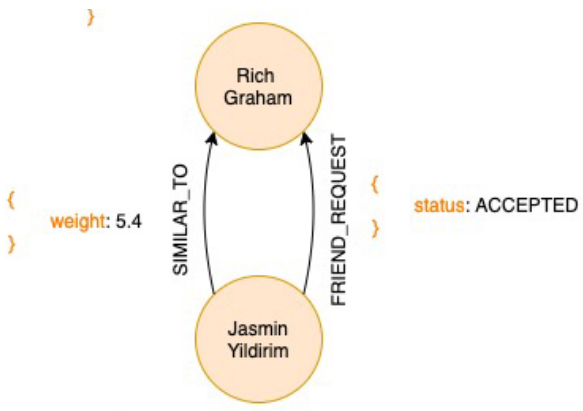
SIMILAR_TO Relation Properties Example

2.2.2.2. Friend Request

The directed relation FRIEND_REQUEST will be added between users if one of them wants to establish a friendship to the other.
Each user can establish a friendship only with recommended users, so one of the K Nearest Neighbors in its cluster.
The relation will maintain a status property that will store the status of the request. It can be ACCEPTED or UNKNOWN.
The relation will also be used to enhance the recommendation of songs.



FRIEND_REQUEST Relation Properties Example

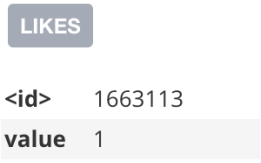


Example of relations among users

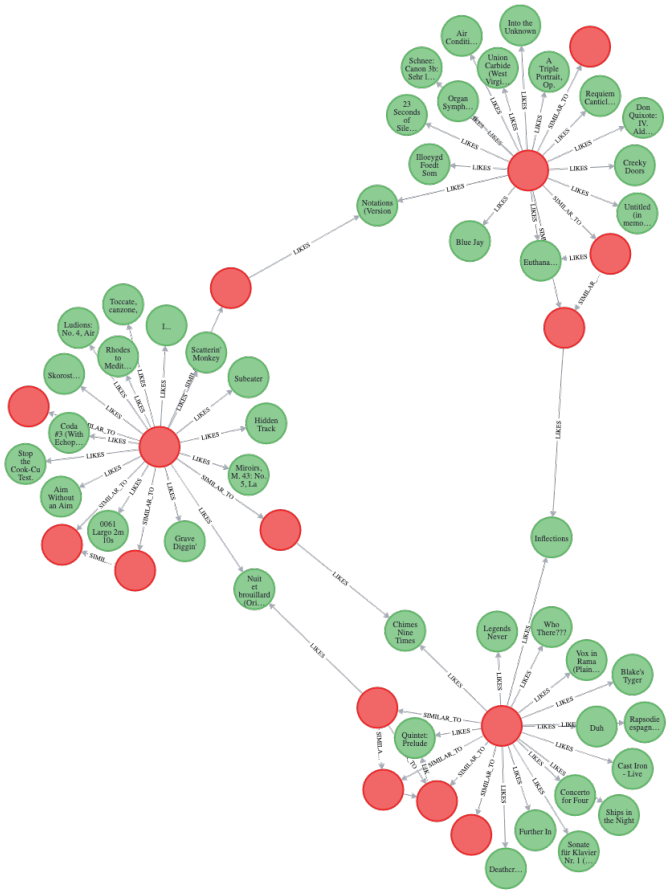
2.2.2.3. Preference

The directed relation LIKE will be added between a user and a song if the user express a preference (like/unlike) on the song.
The relation will maintain a value property that will be equal to 1 if the user likes the song, -1 if it unlikes the song.
The relation will be used to realize the recommendation songs' system.

Example of LIKES Relationships



LIKES Relation Properties Example



Snapshot from the Graph

2.2.3. Queries Analysis

The following read and write queries involves the Graph Database. Each operation is presented with its expected frequency and cost.

Read Operations		
Operation	**Expected Frequency**	**Cost**
Show Recommend Users	High	High (Multiple reads and adjancies)
Show Nearby Users	Medium	High (Multiple reads and adjancies)
Show Recommended Songs	Very High	Very High (Multiple reads and adjancies with complex calculations)
Show Friendships Summarized Information	Medium	Medium (Multiple reads and adjancies)
Show Friend Requests	Low/Medium	Medium (Multiple reads and adjancies)
Show Deviation from Nearest Neighbor	Low/Medium	Low (1 Read)

Write Operations		
Operation	**Expected Frequency**	**Cost**
Register new user into the database	Low	Very high (Create node and allocated edges with the KNN in the cluster)
Update User Information	Very Low	Medium (Multiple Attribute Modifications)
Send Friend Request	High	Low (1 Relation Creation)
Accept/Decline Friend Request	Low/Medium	Low (1 Relation Update)
Express preference on a song	High	Low (1 Relation Creation/ Update)
Check for Cluster Evolution	Very Low	Very high if clusters have changed (Multiple reallocation), very low otherwise

Admin Write Operations		
Operation	**Expected Frequency**	**Cost**
Delete User	Very Low	Medium (Nodes and Relations Deletion)
Quarantine User	Very Low	Medium (Relations Updates)
Add Song	Low/Medium	Low (Node Creation)
Delete Song	Very Low	Medium (Nodes and Relations Deletion)

Admin Read Operations		
Operation	**Expected Frequency**	**Cost**
Show Users By Username	Low	Medium (Nodes Retrieval)
Show Songs By Name	Very Low	Medium (Nodes Retrieval)

From this analysis we can conclude that the graph database's queries are predominantly read intensive.
The only write-heavy operation is the allocation of edges but this is done only at registration and when clusters are updated and the user requires the update of the list. In the lifespan of a user, this operation is typically required few times.

2.2.4. CRUD Operations

2.2.4.1. Create

Operation	
Create Song	CREATE (s: Song {mongoId: $mongo_id, name: $name, artists: $artists, album: $album})
Create User	CREATE (u: User {mongoId: $mongo_id, firstName: $first_name, lastName: $last_name, username: $username, country: $country, picture: $image})
Create Friend Request	MATCH(a: User) MATCH(b: User) WHERE a.mongoId = $from_mongo_id AND b.mongoId = $to_mongo_id CREATE(a)-[r:FRIEND_REQUEST {status: $status}]->(b

2.2.4.2. Read

Operation	
Get Song by Name	MATCH (s:Song) WHERE s.name = $name RETURN s
Get User by MongoId	MATCH (u:User) WHERE u.mongoId = $mongo_id RETURN u
Get User's Friends	MATCH (u: User)-[r:FRIEND_REQUEST]-(friend:User) WHERE r.status = 'ACCEPTED' AND u.mongoId = $mongo_id RETURN friend
Get Incoming Friend Requests	MATCH (a: User {mongoId: $mongo_id})<-[r:FRIEND_REQUEST {status: $status}]-(b:User) RETURN b

Operation	
Get Most Similar User	MATCH (a: User {mongoId: $mongo_id})- [r:SIMILAR_TO]-(b:User) RETURN b ORDER BY r.weight DESC LIMIT 1

2.2.4.3. Update

Operation	
Update Like Relationship	MATCH(u: User) MATCH(s: Song) WHERE u.mongoId = $from_mongo_id AND s.mongoId = $to_mongo_id MERGE(u)-[r:LIKES]-(s) SET r.value = like
Update User	MATCH (u: User {mongoId: $mongo_id}) SET u.firstName: $first_name, u.lastName: $last_name, u.username: $username, u.country = $country, u.picture = $image RETURN u
Update Friend Request	MATCH(a: User) MATCH(b: User) WHERE a.mongoId = $from_mongo_id AND b.mongoId = $to_mongo_id MERGE(a)-[r:FRIEND_REQUEST]-(b) SET r.status = $status

2.2.4.4. Delete

Operation	
Delete Song	MATCH (s:Song) WHERE s.mongoId = $mongo_id DETACH DELETE s

Operation	
Delete User	MATCH (u:User) WHERE u.mongoId = $mongo_id DETACH DELETE u
Delete Friend Request	MATCH (a: User {mongoId: $from_mongo_id})- [r:FRIEND_REQUEST]-(b: User {mongoId: $to_mongo_id}) DELETE r

2.2.5. On-graph queries

2.2.5.1. Recommended Songs

For the recommendation of songs our system uses a smart evolving personality clustering recommendation.
In particular, for each song preferred by similar users (users with which the similarity edge is present) for which our user didn't expressed already a preference, we determine the STRENGTH of the recommendation.
This strength of recommendation is given by a linear combination of a factor given by the choice of the personality clustering algorithm and a value given by the application's use.
The factor given by the clustering algorithm will take in consideration the difference on personality traits of the users.
This weight will be doubled if the two users are friends.
If two users are friends, in fact, probably the clustering algorithm has provided good suggestions and the user may want to prefer songs preferred by its friends with respect to similar users which are not friends.
The other factor takes in consideration the coherence among users' preferences and will permit our clusters to evolve.
In particular, if the users have liked/unliked the same songs the coherence will increase, otherwise the coherence will decrease.
So, it represents the accuracy of our system on detecting the correct similar users for a person.
In conclusion, we will consider all the songs preferred by similar users and for each path we have to reach them, that involves a similarity edge and a preference edge, we calculate this linear combination and multiply by the value property of the preference edge (that can be +-1).

$$Strength = \sum_{paths:User x \to User y \to Song} (f(\alpha) * Weight + \beta * Coherence) * PreferenceValue_y$$

$$\alpha \in [0,1] \quad \beta \in [0,1]$$

1) $Weight = \frac{1}{dist^2(x,y)}$ $\quad dist(x,y) = \sqrt{\sum_{i=1}^{6} (personality_x - personality_y)^2}$

$$f(\alpha) = \begin{cases} 2 & if\ x\ and\ y\ are\ friends \\ 1 & otherwise \end{cases}$$

2) $$Coherence = \frac{\#CoherentPreferences}{\min(\#PreferencesX,\ \#PreferencesY)}$$

$$\#CoherentPreferences = \#edges\ towards\ the\ same\ song\ with\ same\ value$$

A higher value of alpha will prefer the results given by the clustering algorithm, while a higher value of Beta will prefer the coherence, and so the number of songs with which they both agree on preference.

$$Strength_{Userx \rightarrow Song} = \sum_{paths:Userx \rightarrow Usery \rightarrow Song} (f(\alpha) * Weight + \beta * Coherence) * PreferenceValuey$$

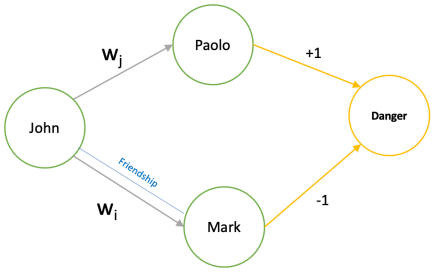
$\alpha \in [0,1]$ $\beta \in [0,1]$

Clustering Algorithm & Survey

1) $Weight = \frac{1}{dist^2(x,y)}$

$$dist(x,y) = \sqrt{\sum_{i=1}^{6} (personality_x - personalityy)^2}$$

$$f(\alpha) = \begin{cases} 2 & if\ x\ and\ y\ are\ friends \\ 1 & otherwise \end{cases}$$



John->Danger = w_i*(+1) + 2*w_j*(-1)

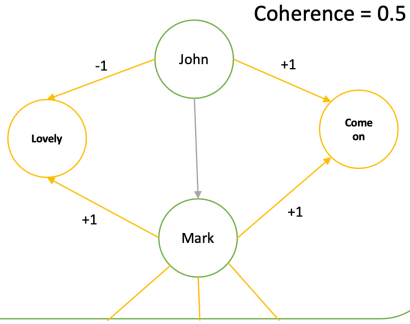
$$Strength_{Userx \rightarrow Song} = \sum_{paths:Userx \rightarrow Usery \rightarrow Song} (f(\alpha) * Weight + \beta * Coherence) * PreferenceValuey$$

$\alpha \in [0,1]$ $\beta \in [0,1]$

Use of the application

2) $Coherence = \frac{\#CoherentPreferences}{\min(\#PreferencesX,\ \#PreferencesY)}$

$\#CoherentPreferences =$
$\#edges\ towards\ the\ same\ song\ with\ same\ preference$

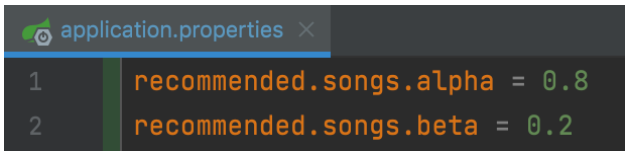


For each of these song we will determine the strength and return the list of the first 50 songs on decreasing order of strength.
We can notice that the evolution of the clusters, according to the coherence will regards only the first K Nearest Neighbors.
Due to this consideration, this algorithm will far away some of the neighbors and get closer others, if they will appear to prefer the same songs.
Using only the first K Nearest Neighbors we will not alter our clustering structure in a massive way, but it can be seen as a way to only modifying the distance among nearest neighbors, if they're similar in terms of music's taste.

```
MATCH (u:User)-[r:SIMILAR_TO]->(su:User)-[p:PREFERENCE]->(s:Song)
WHERE NOT (u)-[:PREFERENCE]->(s) AND u.mongoId = $mongoId
OPTIONAL MATCH (u)-[f:FRIEND_REQUEST {status: "accepted"}]->(su)
WITH s AS Song, u AS User, su as SimilarUser, p as Preference,
    CASE when count(f)>0 then r.weight * 2 else r.weight end AS Weight
OPTIONAL MATCH (u:User)-[p:PREFERENCE]->(s:Song)<-[p2:PREFERENCE]-(su:User)
WHERE su.mongoId <> u.mongoId AND p.value = p2.value AND u = User AND su = SimilarUser
WITH User, Song, SimilarUser, count(p) AS Coherence, Weight, Preference
OPTIONAL MATCH (u:User)-[pr:PREFERENCE]->(s:Song)
WHERE u = User
WITH count(pr) AS numLikes1, Song, SimilarUser, User, Coherence, Weight, Preference
OPTIONAL MATCH (u:User)-[pr:PREFERENCE]->(s:Song)
WHERE u = SimilarUser
WITH count(pr) AS numLikes2, numLikes1, Song, SimilarUser, User, Coherence, Weight, Preference
UNWIND [numLikes1,numLikes2] AS numLikes
WITH User, SimilarUser, Song, Preference, CASE when min(numLikes) <> 0 then Coherence/(toFloat(min(numLikes))) else 0 end AS BetaStrength, Weight
RETURN Song, sum($alpha) * Weight * Preference.value + $beta * BetaStrength * Preference.value) AS Strength
ORDER BY Strength DESC
LIMIT 50
```

Cypher Query

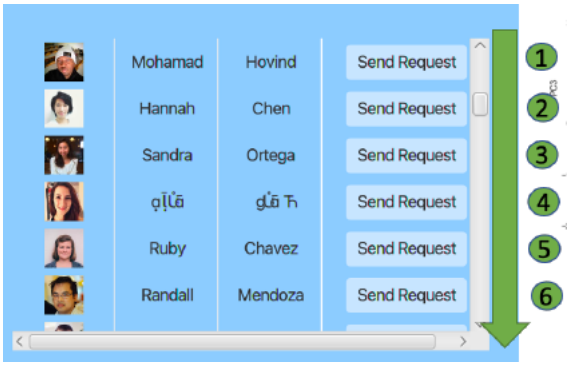
Alpha and Beta will be two parameters editable by configuration parameters' file.

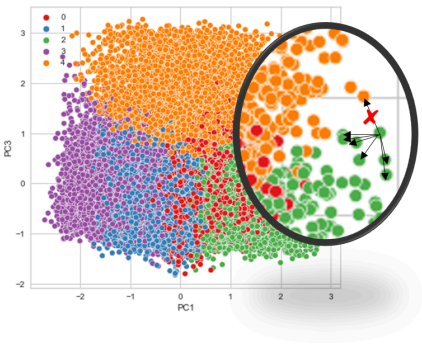


Application Properties File

2.2.5.2. Suggested Users

For the recommendation of users, our system uses the first K Nearest Neighbors among the people of the same cluster.
The users will so be ordered in an increasing order of distance, and so a decreasing order of weight.





$$w_i = \frac{1}{dist^2(x,y)}$$

```
MATCH (a:User)-[f:FRIEND_REQUEST]-(b:User)
WHERE a.mongoId = '$mongo_id'
AND f.status <> 'UNKNOWN'
WITH collect(b) AS excluded
MATCH (a:User {mongoId:$mongo_id })-[r:SIMILAR_TO]-(b:User)
OPTIONAL MATCH (a)-[outgoing:FRIEND_REQUEST]->(b)
OPTIONAL MATCH (a)<-[incoming:FRIEND_REQUEST]-(b)
WITH excluded, r, outgoing, incoming, collect(b) AS users
WHERE NONE(b IN users WHERE b IN excluded)
RETURN users, outgoing, incoming
ORDER BY r.weight DESC
```

Cypher Query

2.2.5.3. Top Cluster with the highest number of Coherent similar users

The following on-graph query will determine the Top cluster with the highest amount of similar users' couple with a strong strength, which are the similar users suggested by the clustering algorithm that actually like the same songs.

```
MATCH (u:User)-[p:LIKES]->(s:Song)<-[p2:LIKES]-(su:User)
WHERE su.mongoId <> u.mongoId AND p.value = p2.value AND EXISTS ((u)-[:SIMILAR_TO]-(su))
WITH u AS user, su AS similarUser, count(*) AS coherence
MATCH (u:User)-[pr:LIKES]->(s:Song)
WHERE u = user
WITH count(*) AS numLikes1, similarUser, user, coherence
MATCH (u:User)-[pr:LIKES]->(s:Song)
WHERE u = similarUser
WITH count(*) AS numLikes2, numLikes1, similarUser, user, coherence
UNWIND [numLikes1,numLikes2] AS numLikes
WITH  user, similarUser, coherence/(toFloat(min(numLikes))) AS Strength
WHERE Strength > 0.75
RETURN user.cluster, count(*) AS NumUsers
ORDER BY NumUsers DESC
LIMIT 1
```

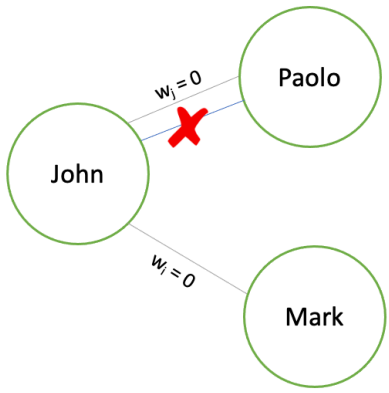
Cypher Query

2.2.5.4. Quarantine Users

In case one user will perform anomalous actions, the admin can decide to quarantine the user for a certain amount of time, removing all its friends and avoiding to use the user in the recommendation system.

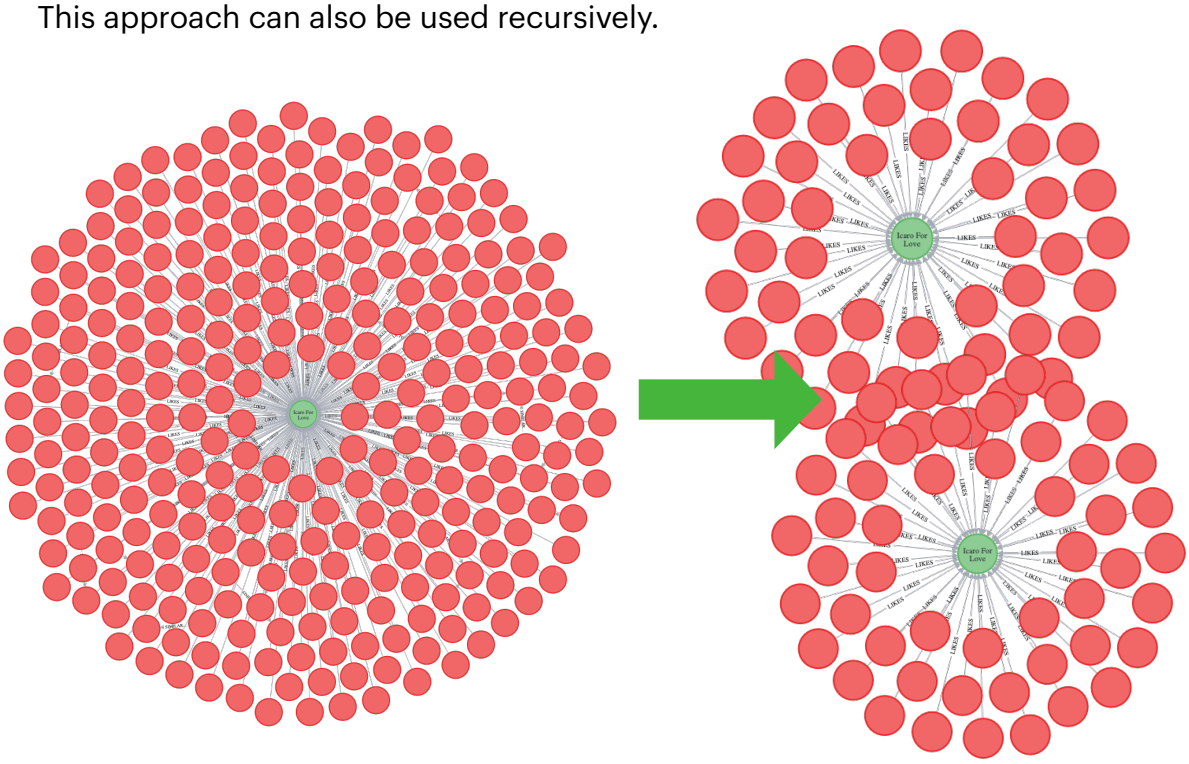
```
MATCH (u:User)-[f:FRIEND_REQUEST]-(:User)
WHERE u.mongoId = $mongo_id
DELETE f

MATCH (u:User)-[r:SIMILAR_TO]-(:User)
WHERE u.mongoId = $mongo_id
SET r.weight = 0
```

Cypher Query



2.2.6. Supernodes

In our graph database, we may have songs which are liked by everyone, so with a huge amount of relationships.
Let's think for example at some classic songs, like Bohemian Rhapsody.
These super-nodes may be problematic because they considerably slow down graph traversal and slow down our read and write operations.
For this reason we decided to refactor these nodes, splitting them into multiple nodes.
The splitting strategy we adopted is the following.
We determine all songs' nodes which have a number of LIKES relationships higher than a certain threshold and determine the two clusters of users which have expressed the highest amount of preferences to that song, which is the highest number of LIKES relationships.
Then, we update the starting song node with a new property, the cluster's property, equal to $MajorityClusterID and we create a copy of the node with the cluster's property equal to $SecondMajorityClusterID.
After that, we move all LIKES relationships regarding user of the $SecondMajorityCluster to the second node and also we distribute all the remaining clusters' nodes' relationships, cluster by cluster, among the two nodes. Of course, our queries will need to be rewritten in such a way to consider now the song's cluster property.
In particular, when a new like/unlike will be added to the split song, we will choose, if available, the song with its same cluster, otherwise if the cluster of the user is not one of the two previously mentioned, the relationships will be given randomly to one of the two nodes.
This approach can also be used recursively.



Example of splitting

```
/* Determine Supernodes */
MATCH (s:Song)<-[l:LIKES]-(u:User)
WITH s AS Song, count(*) AS NumLikes
WHERE NumLikes > 'Threshold'
RETURN Song

/* Determine Top 2 Clusters with the highest number of relationships */
MATCH (s:Song)<-[r:LIKES]-(u:User)
WHERE s.mongoId = $mongoId
RETURN u.cluster AS Cluster, count(*) AS Count
ORDER BY count DESC

/* Add cluster field to the supernode */
MATCH (s:Song{mongoId : $mongoId})
UPDATE SET s.cluster = $firstId

/* Create duplicate */
MATCH (s:Song{mongoId : $mongoId})
CREATE (s2:Song{mongoId : $mongoId, songName: $songName, artists: $artists, album: $album, cluster: $secondId})

/* Move relationships regarding the second highest cluster */
MATCH (s:Song{mongoId : $mongoId, cluster: $firstId})<-[r:LIKES]-(u:User {cluster: $secondId})
MATCH (s2:Song{mongoId: $mongoId, cluster: $secondId})
CREATE (u)-[rNew :LIKES]->(s2)
SET rNew = r
DELETE r

/* Distribute the remaining relationships regarding the other clusters */
/* Example 5 clusters, the 2nd and the 4th clusters goes to the duplicated Supernode */
MATCH (s:Song{mongoId : $mongoId, cluster: $firstId})<-[r:LIKES]-(u:User {cluster: $fourthId})
MATCH (s2:Song{mongoId: $mongoId, cluster: $fourthId})
CREATE (u)-[rNew :LIKES]->(s2)
SET rNew = r
DELETE r
```

Splitting Supernode Cypher Statements

2.2.7. Neo4J Indexes

In order to improve the read operations' performance, the following indexes are introduced:

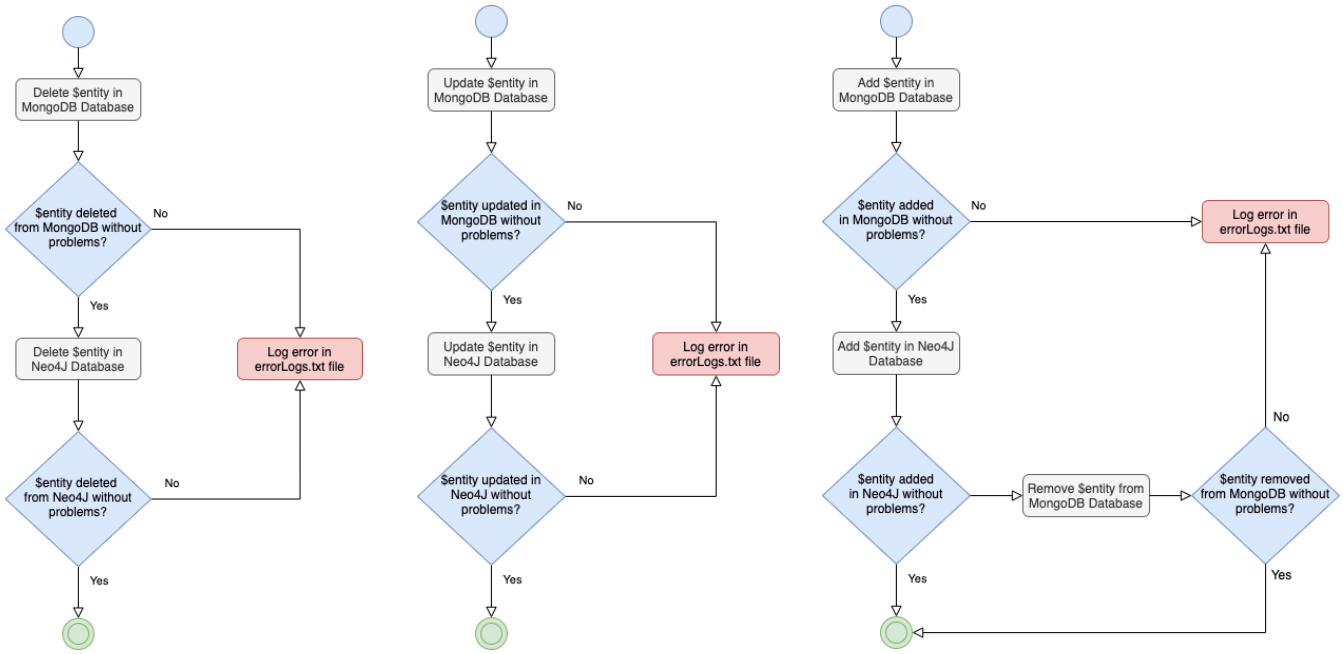
Index Type	Index Name	Index Label	Index Field
BTREE	User Index	:User	mongoId
BTREE	Song Index	:Song	mongoId

2.3. Cross-Database Consistency Management

The operations that require a cross-database consistency management are the insertion, removal and updating of a song, a user or a preference.

- **Addition of a generic entity.** If an error occurs with the first write operation, we just log the error in errorLogs.txt file, while if an error occurs with the second write operation, we log the error in the same file and also try to delete the preceding write operation in the other database.
- **Removal of a generic entity.** If an error occurs with the first removal or second removal, we log the error and terminate.
- **Update of a generic entity.** If an error occurs with the first or second update, we log the error and terminate.

Introducing these consistency management operations, we remain coherent with the side of the CAP triangle we've chosen.
Administrators will manually check the errorLogs.txt file and update/add/remove entities or increment/decrement counters to restore the consistency.

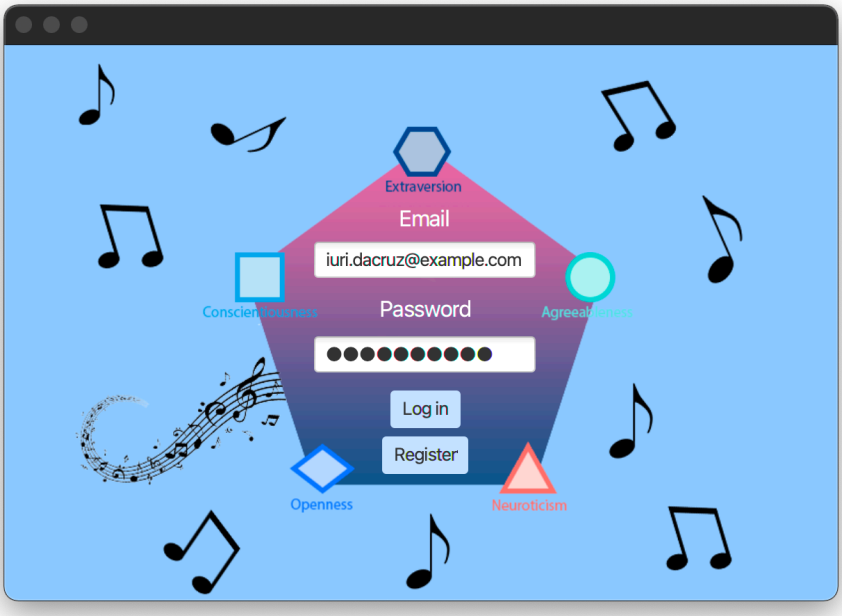


Consistency Management

3. User Manual

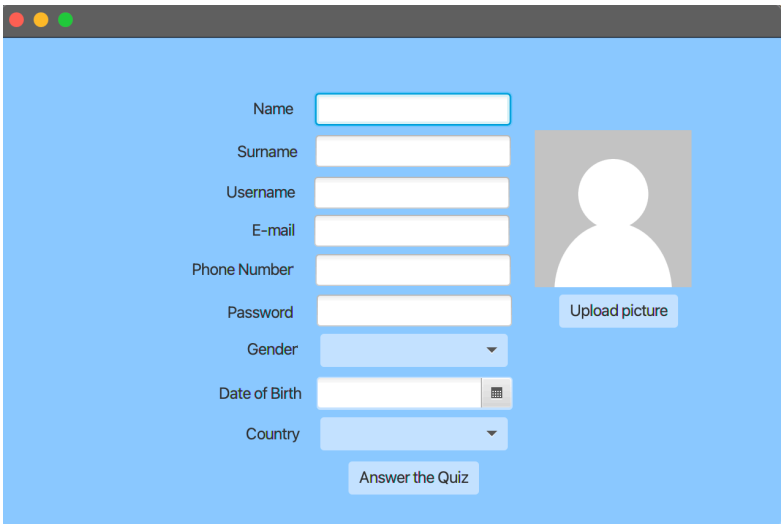
3.1. Login

The user can login to the application using email and password.
If the user is not registered yet, it can use the Register button to start the registration.



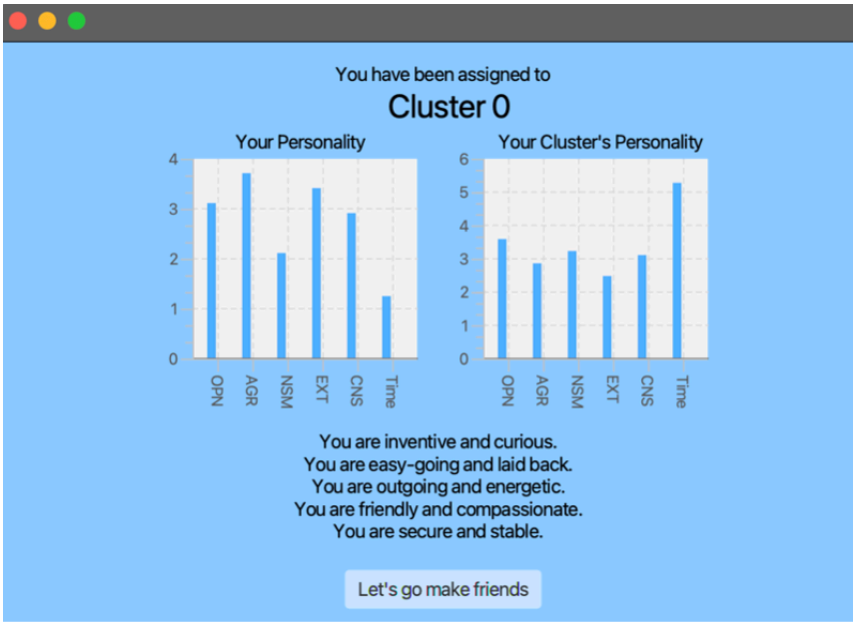
3.2. Registration

The user can use the registration form to register into the application providing personal information.



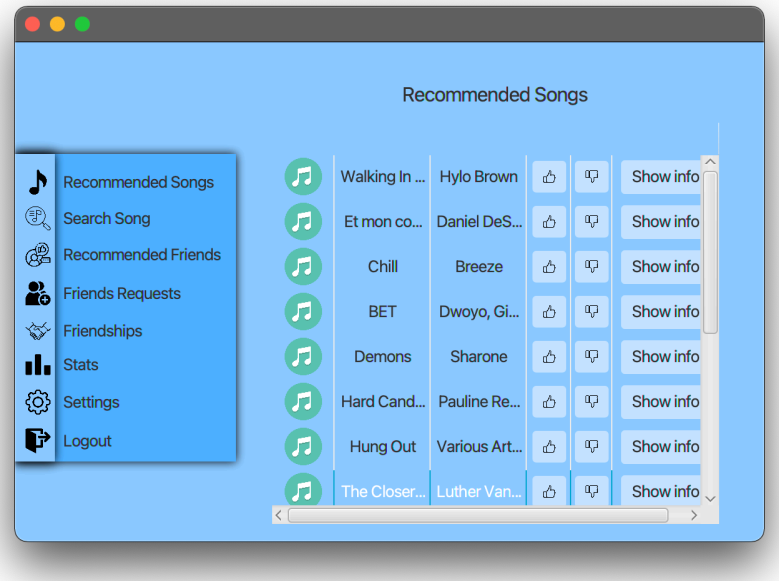
Once provided all the information needed, the user can start to answer the survey.

Fifty different question will require to be answered using a value between 1 (Disagree) and 5 (Agree). The default value is 3 (Neutral).
Once answered to all the questions, the user will receive information about its personality traits and the cluster in which he will be assigned.

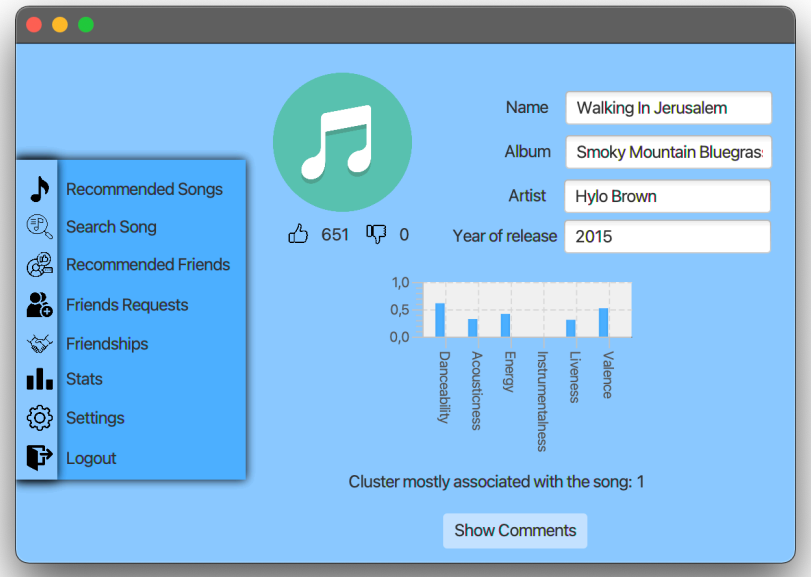


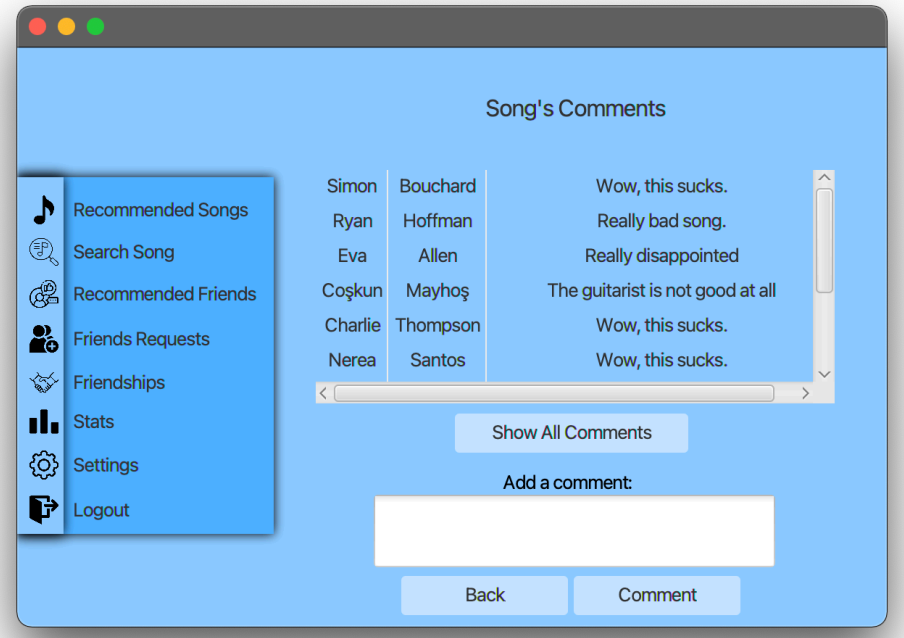
3.3. Recommended Songs

In the home page the user will be able to browse recommended songs and like/unlike them and see their information.



The songs will be listed in a decreasing order of strength.
The user can decide to see more information on the song:

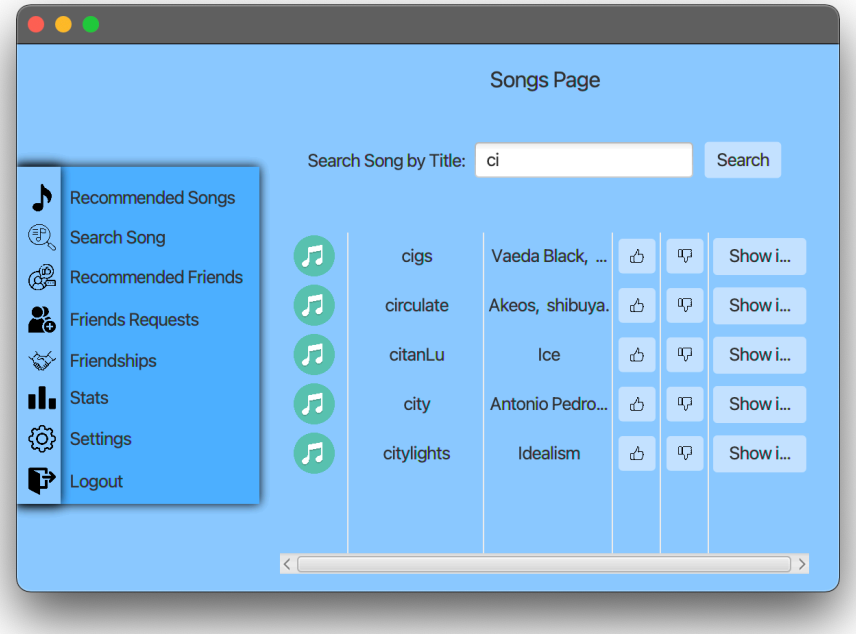




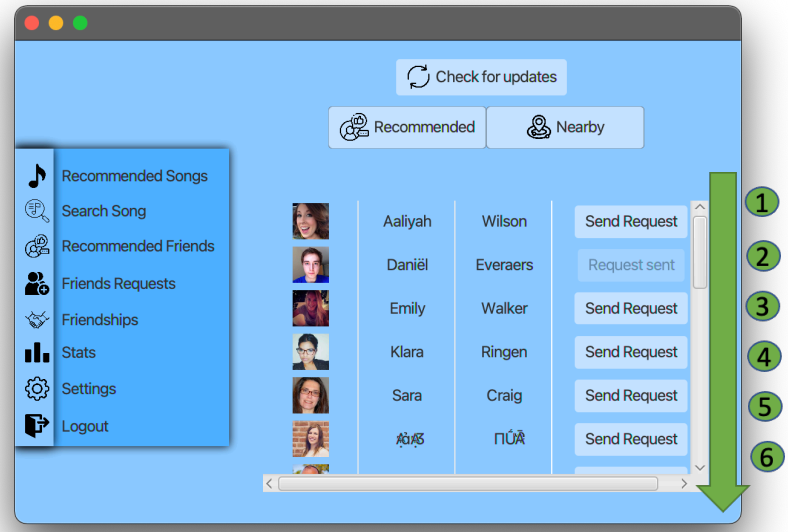
Clicking on 'Show All Comments' the user will be able to see more than the 10 most recent comments, accessing to the comment's collection.

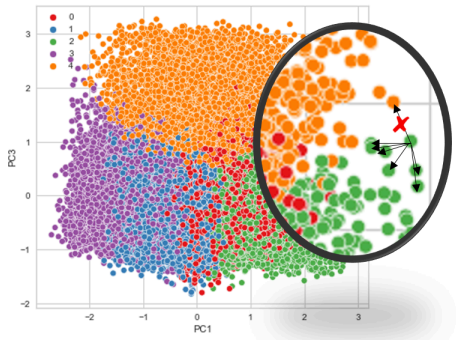
3.4. Search Songs

The user can search for songs by Title using the search page.



3.5. Recommended Users



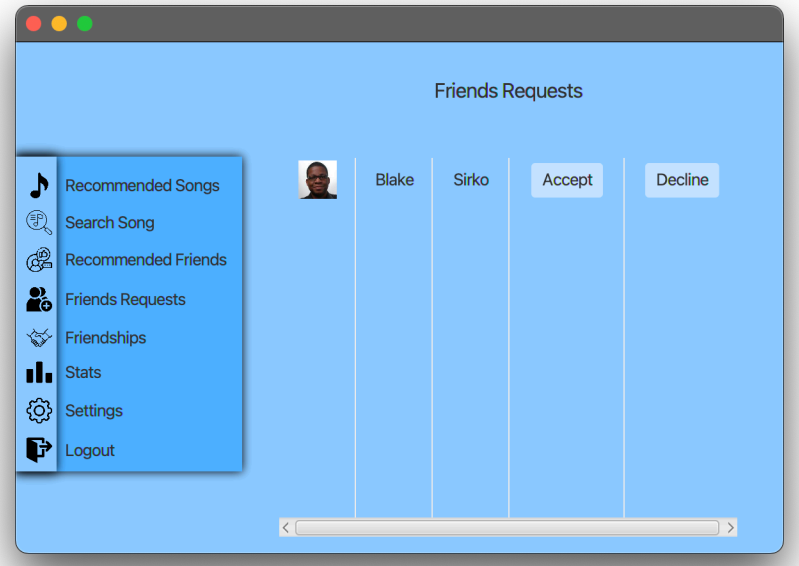


$$w_i = \frac{1}{dist^2(x,y)}$$

The recommended users shown are the thirty nearest neighbors inside the same cluster and they will be ordered according to the euclidian distance on the space of the features, in increasing order.
The user will also be able to browse nearby friends, which are recommended friends in the same country.
Using the 'Check For Updates' button, the user can update the list if the clusters have changed.

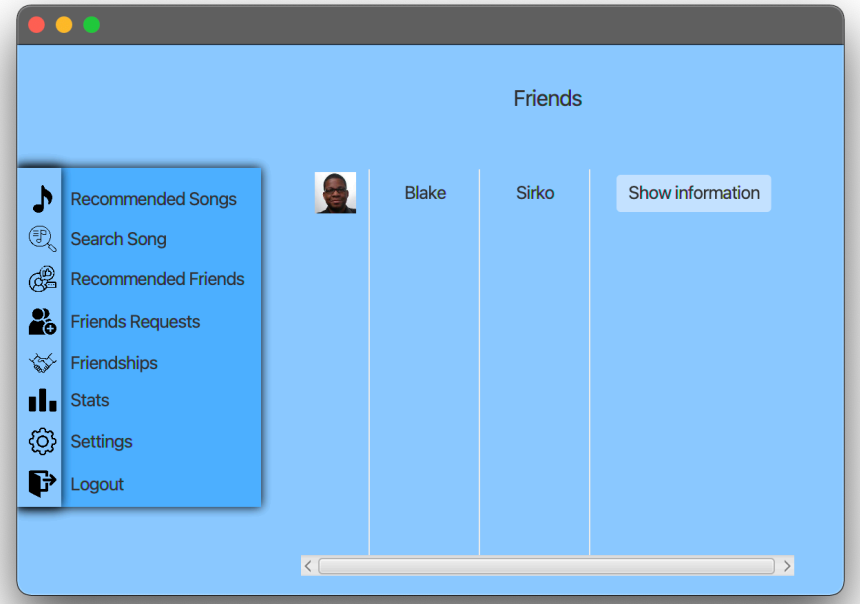
3.6. Friendship Requests

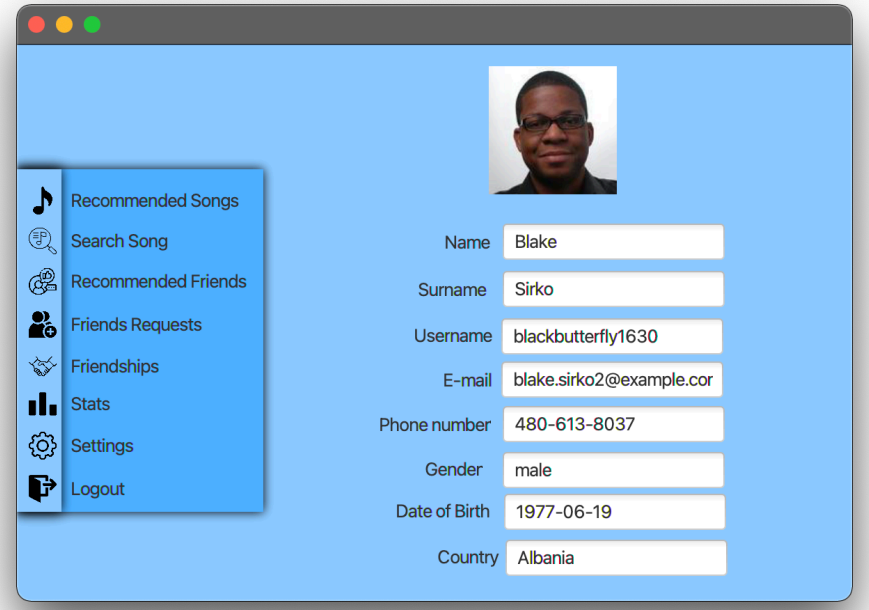
The user will be able to browse incoming friend requests and accept or decline them.



3.7. Friendships

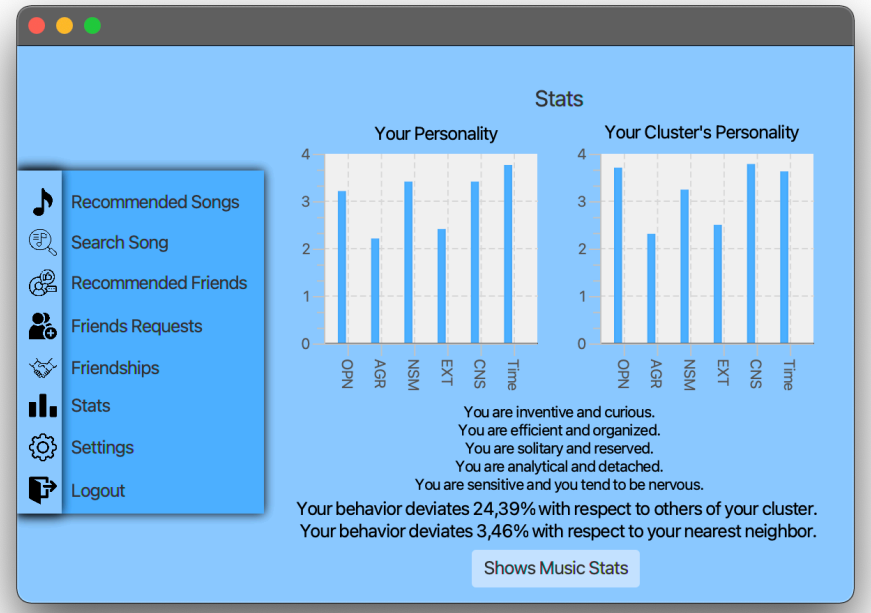
The user will be able to browse friends and see their information to start a conversation.



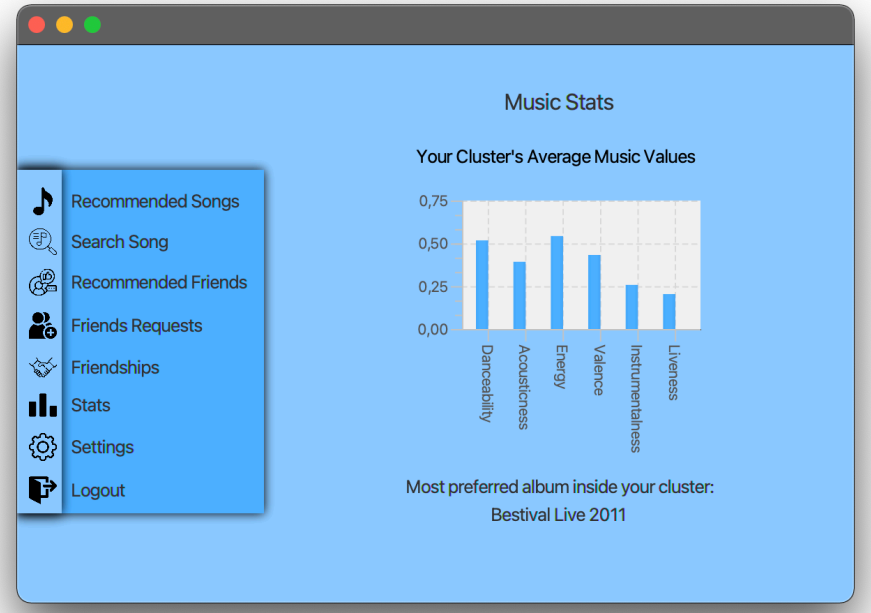


3.8. Stats

In the stats menu, the user can see statistics about its personality and the average personality of people in the same cluster.

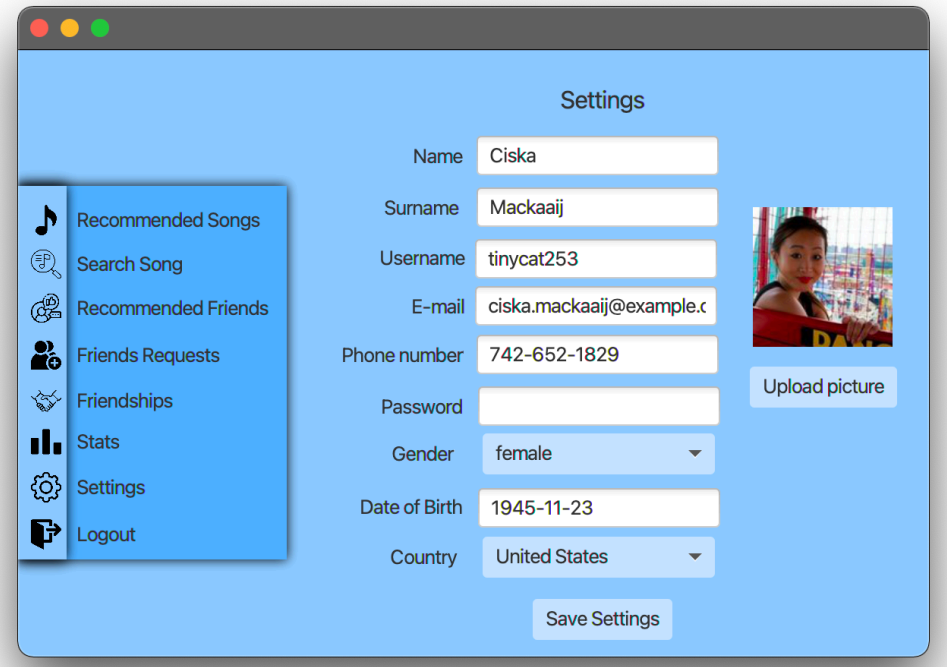


A brief description of the personality in terms of big 5 personality traits will also given to the user, depending on its answers to the survey.
The user can also have information about the deviation of its behavior from the cluster's average behavior, and its deviation from the nearest neighbor in the cluster. Clicking on 'Show Music Stats' the user can see music statistics.



3.9. Settings

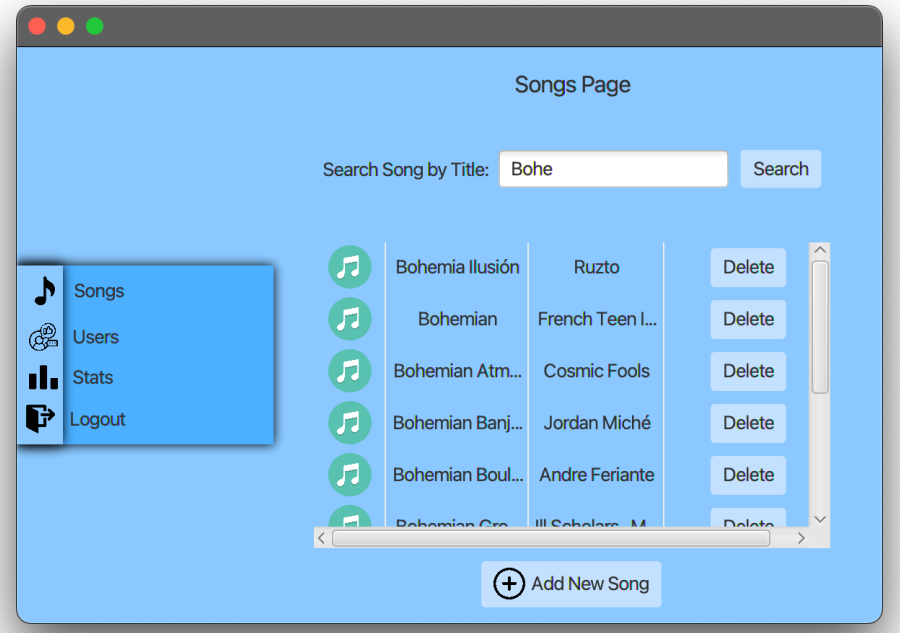
Using the settings tab, the user will be able to change personal information.



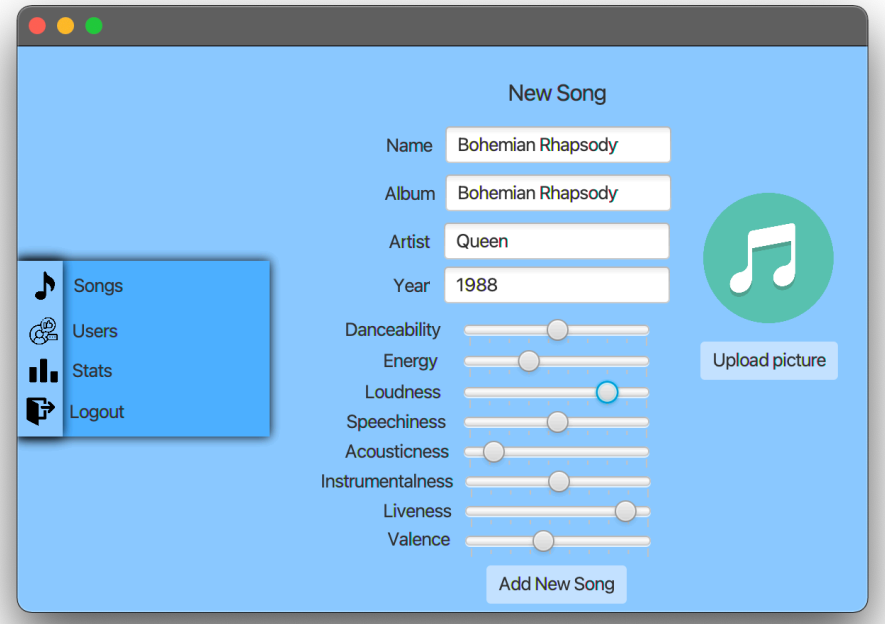
4. Admin Manual

4.1. Songs Page

The administrator can search for songs by title and delete them.

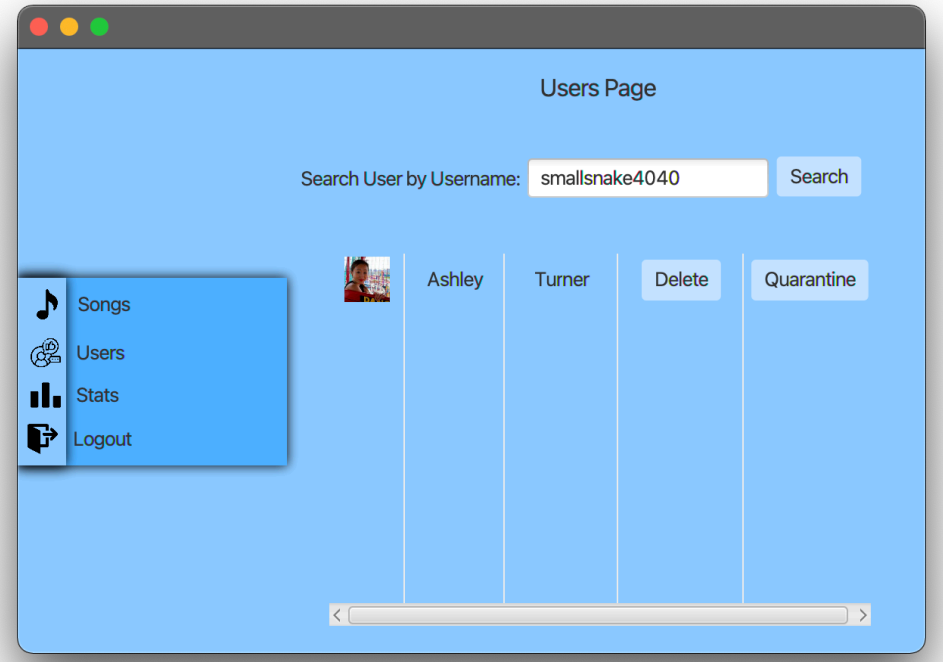


The admin can also add new songs.



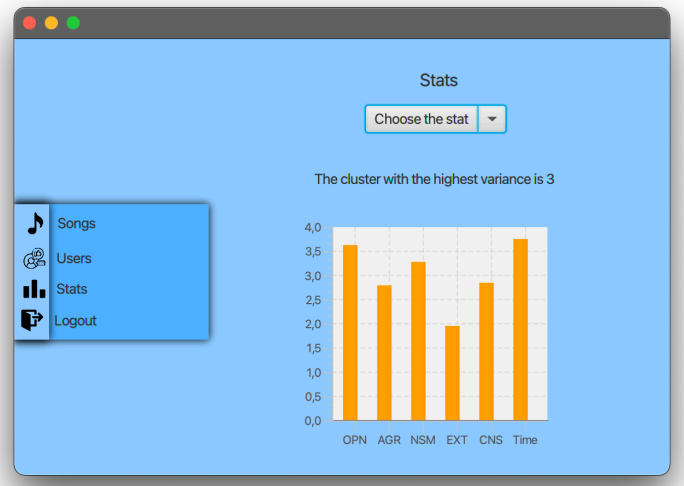
4.2. Users Page

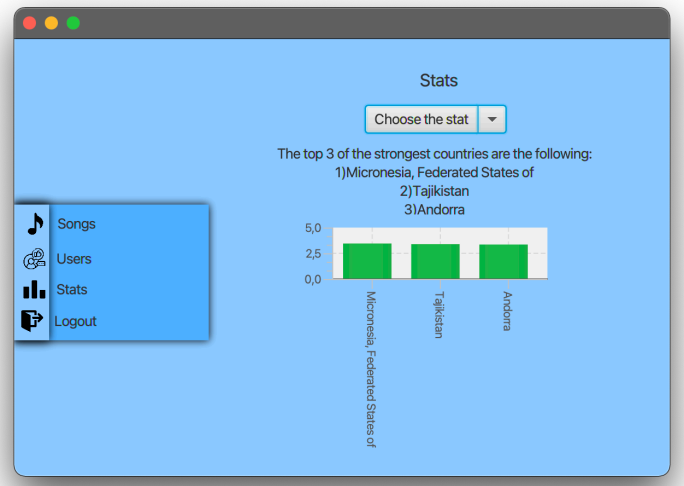
Using the Users Page, the admin can search for users by username and delete or quarantine their account.



4.3. Stats Page

Using the Stats Page, the admin can calculate statistics regarding clusters and countries.





5. Implementation

All the source code of the application can be found on GitHub at the following links:
- Server GitHub Repository: https://github.com/jacopocecch/large-scale-backend
- Client GitHub Repository: https://github.com/jacopocecch/large-scale-frontend
- Dumps and Executables Repository: https://drive.google.com/drive/folders/1CRJmFdyDWe8Nmyb4AEFrhrufuZyEI58V

5.1. Client Side

In this section, the main packages of the client of the application and the contained classes are described.

5.1.1. com.unipi.largescale

This package contains the main class, that starts the application.

Classes:
- PersonalityClustering: this class extends Application, implements the start method and contains the main.

5.1.2. com.unipi.largescale.gui

This package contains the FXML Document Controllers and the classes for input fields' validation.

Classes:
- FXMLPageDocumentController: this class implements Initializable and represents the controller for the GUI of the $Page of the application.
- LoaderFXML: this class provides the method that allows to load the correct FXML file.
- ValidationForm: this class provides the methods for validating input fields.

5.1.3. com.unipi.largescale.util

This package contains the utilities for the application.

Classes:

- ConfigurationParameters: this class provides the configuration parameters of the application and it is built starting from an XML file.
- UtilGUI: this class provides methods that contains utilities for the GUI.

5.1.4. com.unipi.largescale.entities

This package contains the entities of the application.

Classes:
- User: this class stores all the information of a user, in particular personal information, the cluster and the survey associated.
- Survey: this class stores all the information of a survey, in particular the average personality features.
- FriendRequest: this class stores all the information of a friend request, in particular the user associated and the status.
- SimilarUser: this class stores all the information of the relationship between two similar users, in particular the weight.
- Cluster: this class stores all the information regarding mean values of personality traits for a cluster.
- Song: this class stores all the information regarding a song, in particular general information and feature characteristics.
- Comment: this class stores all the information regarding a comment, in particular the user and the text in the comment.
- Aggregation subpackage: this subpackage contains classes for receiving answers to aggregations.

5.1.5. com.unipi.largescale.dtos

This package contains the dtos used to communicate with the application server.

Classes:
- UserDto: this class stores all the information of a user, in particular personal information, the cluster and the survey associated.
- QuestionDto: this class stores all the information of a question, in particular the question label, the answer and the time.
- SurveyDto: this class stores all the information of a survey, in particular the list of questions and answers.
- FriendRequestDto: this class stores all the information of a friend request, in particular the user associated and the status.
- SimilarUserDto: this class stores all the information of the relationship between two similar users, in particular the weight.

- ClusterDto: this class stores all the information regarding mean values of personality traits for a cluster.
- LoginDto: this class stores all the information regarding the login of a user, in particular username and password.
- SongDto: this class stores all the information regarding a song, in particular general information and feature characteristics.
- CommentDto: this class stores all the information regarding a comment, in particular the user and the text in the comment.
- CommentSubsetDto: this class stores all the information regarding an embedded comment, in particular the user and the text in the comment.
- InterfaceSongDto: this class stores a smaller amount of information regarding a song (coming from graph database).
- InterfaceUserDto: this class stores a smaller amount of information regarding a user (coming from graph database).
- LikeDto: this class stores information about preferences.

5.1.6. com.unipi.largescale.beans

This package contains the beans used to represent the rows In the JavaFX table view.

Classes:
- UserBean: this class stores the GUI information of a user.
- SongBean: this class stores the GUI information of a song.
- CommentBean: this class stores the GUI information of a comment.

5.1.7. com.unipi.largescale.API

This package provides all the communication with the server.

Classes:
- API: this class contains the methods required for the communication with the server.

5.1.8. com.unipi.largescale.services

This package provides all the functionalities of the application.

Classes:
- SongService: this class stores all main song's functionalities.
- UserService: this class stores all main user's functionalities.
- AdminService: this class stores all main admin's functionalities.

5.2. Server Side

In this section, the main packages of the server of the application and the contained classes are described.

5.2.1 com.unipi.large.scale.backend

This package contains the Spring Boot Application.

5.2.2 com.unipi.large.scale.backend.configs

This package contains all the configuration classes.

- RecommendedSongsConfigurationProperties: this class contains the configurations for the recommended songs query in neo4j. The values of the variables alpha and beta are taken from the respective fields in the application.properties configuration file.
- Config: this is the configuration class. It contains the configuration for the CORS Mapping and the DTO mapper.

5.2.3 com.unipi.large.scale.backend.controllers

This package implements the controller layer of the application

5.2.3.1 com.unipi.large.scale.backend.controllers.errors
This package is responsible for handling the exceptions and sending them back to the client in an interpretable way

- ControllerExceptionHandler: this class contains the code for generating the error response message based on the type of exception that was thrown by the application.
- ErrorBody: the body of the response with a status code and a message.
- ErrorResponse: the error response class.

5.2.3.2 com.unipi.large.scale.backend.controllers.services

This package handles all the requests and responses.

- ServiceController: this abstract class contains al the instances for the DTO and service layer that need to be used by the controllers.
- UserController: this controller extends ServiceController and implements all the functions regarding the User entity needed by the client by defining a different URI for each one. It implements serialization/deserialization and call the corresponding service layer functions.

- SongController: this controller also extends ServiceController and implements all the functions regarding the Song entity.
- ClusteringController: this class starts the clustering process during the application startup.

5.2.4 com.unipi.large.scale.backend.daos

This package implements the Data Access Object for the Neo4j database. There is no DAO class for MongoDB because we used Spring Data MongoDB.

- Neo4jDao: this abstract class contains the logger and the neo4j driver needed by the DAO classes.
- Neo4jUserDao: this class extends Neo4jDao and implements all the CRUD operations and Cypher queries needed by the service layer on the User node and its relationships.Neo4jSongDao: this class extends Neo4jDao and implements all the CRUD operations and Cypher queries needed by the service layer on the Song node and its relationships.

5.2.5 com.unipi.large.scale.backend.data

This package implements the data layer.

- Survey: this class is needed by the service layer for storing the summary of the survey values for clustering and computing the distances between users.
- Distance: this class is needed by the service layer for storing the distance between two users.
- Login: this class is needed by the service layer to perform the login operation.

6.2.4.1 com.unipi.large.scale.backend.data.aggregations

This package defines the classes needed to store the results of the aggregations.

- Album: this class is used for storing the result of the getClusterKHighestRatedAlbums aggregation.
- AverageMusicFeatures: this class is used for storing the result of the getAverageMusicFeaturesByCluster aggregation.
- Country: this class is used for storing the result of the getTopKCounties aggregation.
- HighestVarianceCluster: this class is used for storing the result of the getClusterWithHighestVariance aggregation.

- Id: this class is used for storing the result of the getMostDanceableCluster aggregation.

5.2.6 com.unipi.large.scale.backend.dtos

This package implements the DTO layer. Each DTO class corresponds to an entity or data class and it is used for the serialization/deserialization at the control layer.

- Mapper: the mapper class implements the mapping between DTO and data/entity classes

5.2.7 com.unipi.large.scale.backend.entities

This package implements the model layer.

5.2.7.1 com.unipi.large.scale.backend.entities.mongodb

This package contains the entities that correspond to the MongoDB documents.

- Comment: this class corresponds to the documents in the comment collection.
- MongoSong: this class corresponds to the documents in the song collection.
- MongoUser: this class corresponds to the documents in the user collection.
- CommentSubset: this class corresponds to the embedded comments document in the song document.
- Like: this class corresponds to the embedded likes document in the song document.

5.2.7.2 com.unipi.large.scale.backend.entities.neo4j

This package contains the entities that correspond to the Neo4j nodes and relationships.

- Neo4jSong: this class corresponds to the Song node.
- Neo4jUser: this class corresponds to the User node.
- FriendRequest: this class is used to store the status of the FRIEND_REQUEST relationship between two users.

5.2.8 com.unipi.large.scale.backend.repository

This package contains the Spring Data MongoDB MongoTemplate, which is used for CRUD queries and aggregations.

- CustomRepository: this abstract class defines the MongoTemplate class for the Repository classes.
- CustomSongRepository: this class extends CustomRepository and implements all the CRUD methods and aggregations regarding the song collection.
- CustomUserRepository: this class extends CustomRepository and implements all the CRUD methods and aggregations regarding the user collection.
- CustomCommentRepository: this class extends CustomRepository and implements all the CRUD methods regarding the comment collection.

5.2.9 com.unipi.large.scale.backend.service

This package implements the service layer.

- Utils: this class implements utility methods needed by both clustering and user service.

5.2.9.1 com.unipi.large.scale.backend.service.clustering

This package implements all the clustering logic.

- Clustering: this class contains the functions needed for the standard KMeans clustering.

5.2.9.2 com.unipi.large.scale.backend.service.db

This package implements the service layer for the entities.

- EntityService: this abstract class contains all the instances of classes needed by the service classes.
- UserService: this class extends EntityService and implements all the user business logic between the Controller Layer and the DAO Layer.
- SongService: this class extends EntityService and implements all the song business logic between the Controller Layer and the DAO Layer.

5.2.9.3 com.unipi.large.scale.backend.service.exceptions

This package contains custom defined RuntimeExceptions.

5.2.10 resources

This package contains the application resources

- application.properties: this configuration file contains the credentials for the database access and the configuration properties for the clustering.
- logback-spring.xml: this XML file contains the configuration for the error logging.