UNIVERSITY OF PISA

Department of Information Engineering

Large Scale and Multi-Structured Databases

BeatBuddy

Work Group:
Arduini Luca
Loni Giovanni Enrico
Mancinelli Lorenzo

ACADEMIC YEAR 2023/2024

INTRODUCTION

Welcome to BeatBuddy, your rhythmic playground! This is where music pulses through the digital space, inviting you to discover vibes with friends, share your album crushes, and let the beats of others lead you to uncharted sonic adventures. Imagine a place where every review, every rating, and every note played is a stepping stone to your next musical revelation.

With BeatBuddy, stay effortlessly in tune with the charts of the hottest albums and songs, letting the diverse tastes of your friends guide you to new tunes. Engage in the vibrant conversation of the music-loving community by sharing your thoughts on the latest talked-about album. And when you crave depth, dive into any album you want, exploring the intricate stats and jotting down the next songs you can't wait to hear.

BeatBuddy isn't just an app; it's a gateway to a world where music is shared, discovered, and cherished in every click, in every playlist, and in every beat. So, plug in, press play, and let the journey begin. Happy listening!

The code is available on GitHub at: github.com/LucaArduini/BeatBuddy

CONTENTS

1 APPLICATION MANUAL

This documentation provides a comprehensive guide to BeatBuddy, our web application developed for the LSMSDB course. BeatBuddy caters to music enthusiasts, offering a platform where users can discover and enjoy their favorite albums, songs, and artists. It also allows users to share their thoughts and connect with other music lovers. A standout feature of BeatBuddy is its **suggestions** system, which recommends new songs based on user preferences.

1.1 APPLICATION HIGHLIGHTS

BeatBuddy combines the features of a social network with a focus on music, emphasizing two main functionalities:

- **Search:** Users can easily search for albums, songs, and artists, and they can access to the dedicated page with detailed information, including track lists, album collections, and user-generated reviews.
- **Discover:** The application is designed to help users uncover new music, showcasing popular tracks and personalized recommendations across the platform.

Additionally, BeatBuddy includes specialized administrative tools for server maintenance, statistical analysis, and potential monetization strategies.

1.2 USER MANUAL

Upon opening the site, users are greeted with options for *login* and *registration*, along with a *button* to open the charts for Songs, Artists and Albums, and a **search bar**.

Unregistered User

Unregistered users have the option to register and become Registered Users or explore the site, without actively interact with it:they can view Album and Artist pages but cannot interact through

likes or reviews. We will discuss these pages in more detail in the Registered User section.

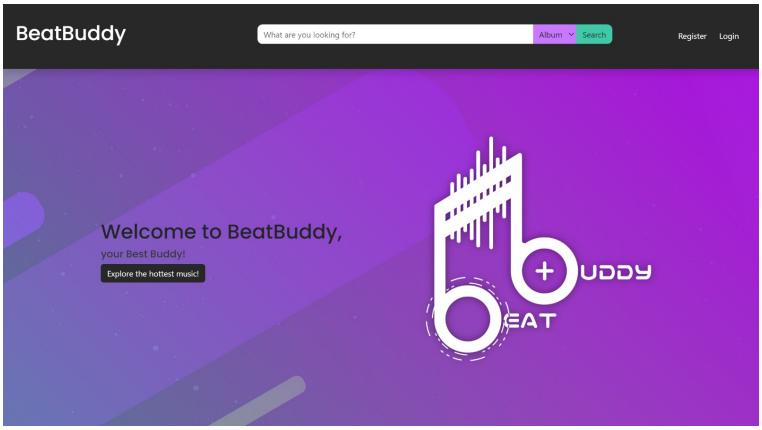


Figure 1.1. Page with search bar and links to Trending page

Registered User

Upon logging in, Registered Users are directed to the main page, from which they can:

- Access their personal profile page;
- Visit the **Most Popular** page;
- Search for Songs, Albums, Artists, or other Users and explore their respective pages;
- Discover new music through the **Discover** page.

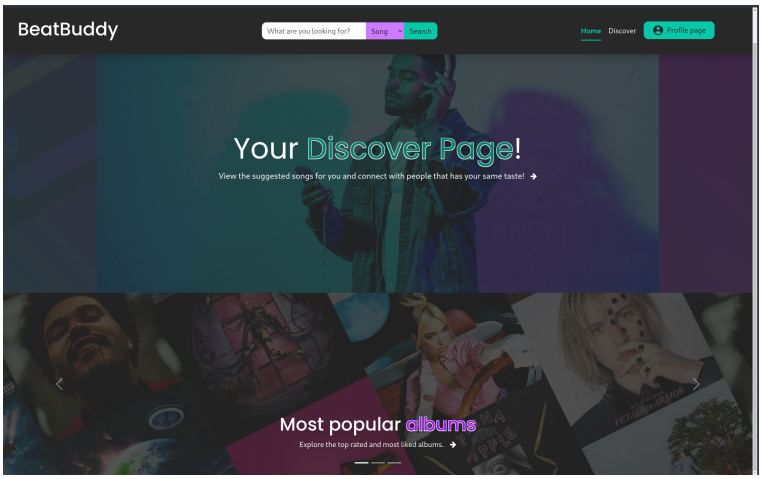


Figure 1.2. Homepage for a registered user

The **personal page** showcases the user's information, previews of their reviews, and sections for followed users, liked albums, and songs. On other users' pages, a **follow** button is available.

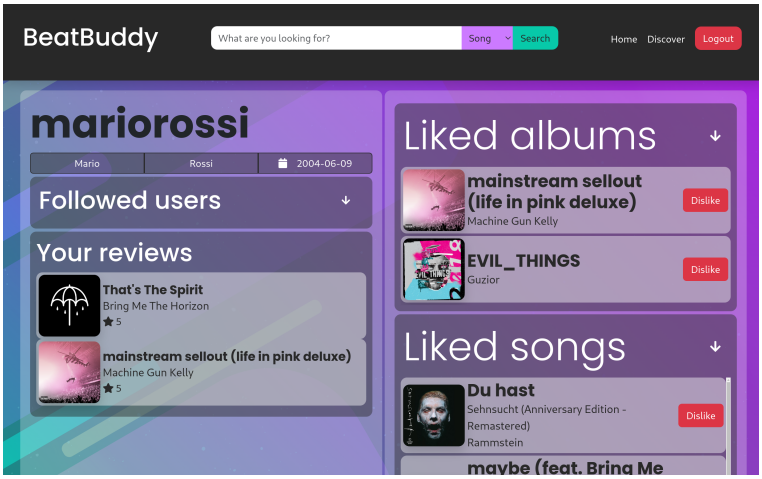


Figure 1.3. User's personal page

The **Most Popular** page allows users to view and access content such as charts for top-rated Artists, Songs, or Albums.

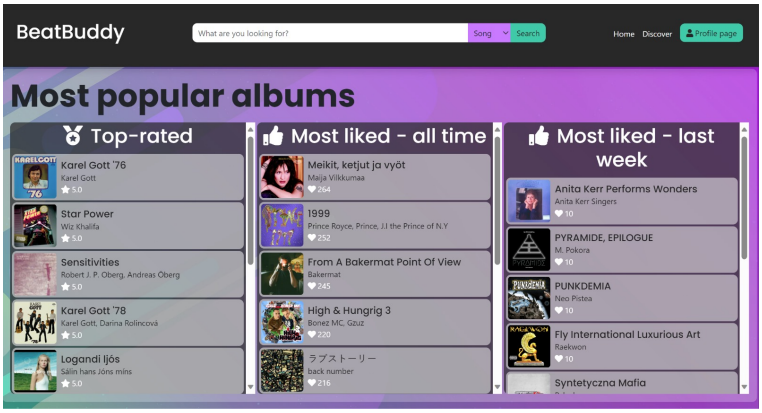


Figure 1.4. Most Popular page for Albums

Using the search bar, users can specify their search for *Songs*, *Albums*, *Artists*, or *Users*, with up to five results displayed. Selections lead to the corresponding pages where further actions can be taken.

On the Album page, users can view track lists, recent reviews, and have the option to like songs or albums, write reviews, or view all reviews related to that album.

The Artist page displays a list of albums, which users can explore further by clicking on their preferred album.

The **Discover** page presents three columns of recommendations: songs suggested based on user's likes, popular songs among its followed users, and a list of recommended users based on current followings.

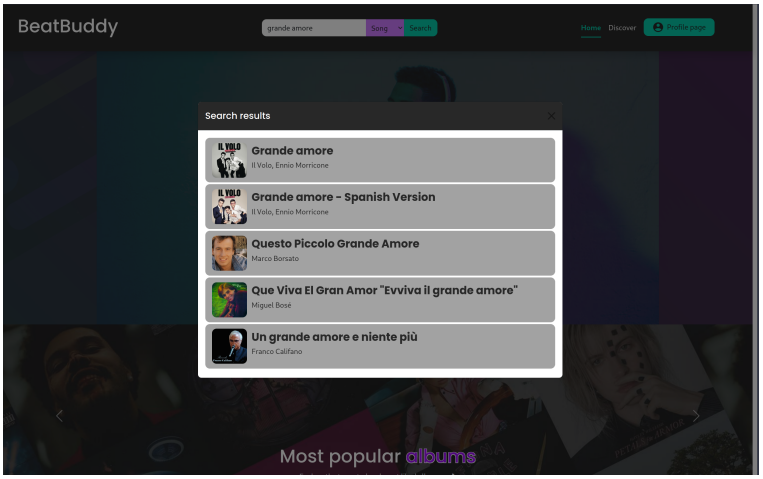


Figure 1.5. Search results for songs using the term *grande amore*

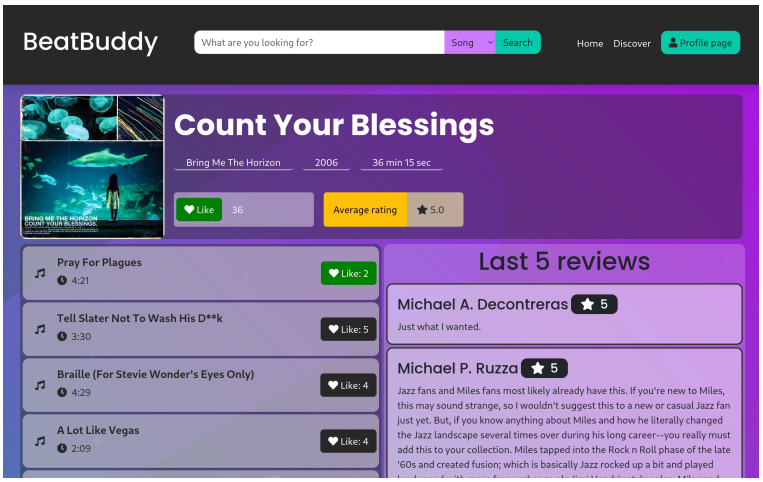


Figure 1.6. Album page as viewed by a registered user who liked a song

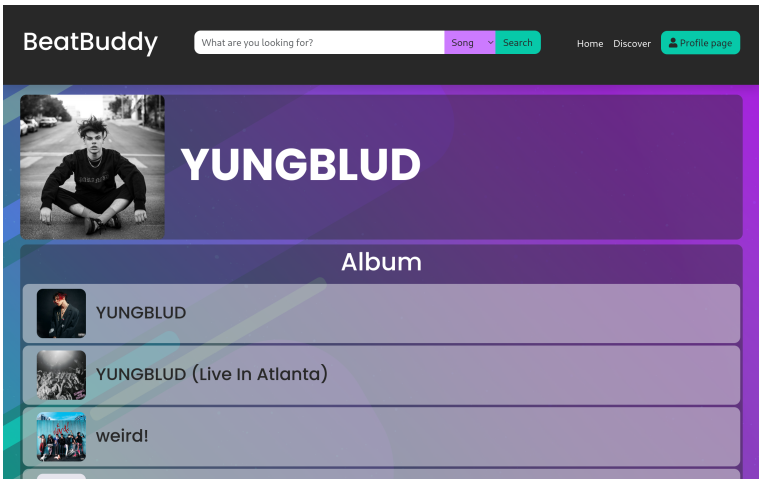


Figure 1.7. Artist page view

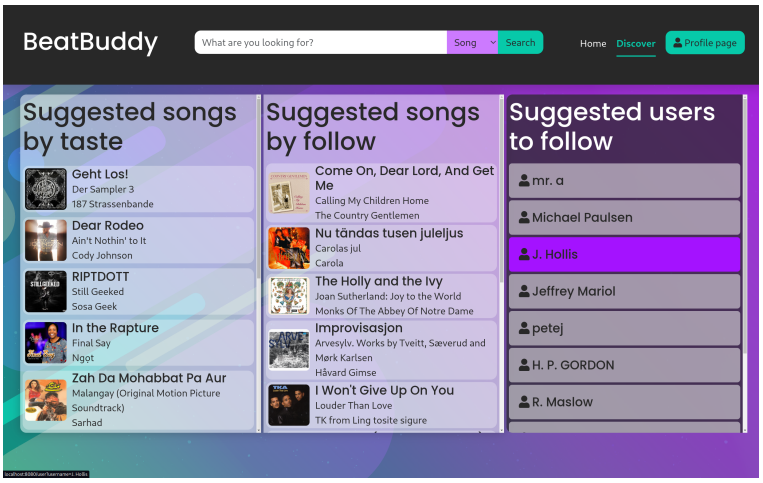


Figure 1.8. Discover Page layout

1.3 ADMIN MANUAL

The **Admin Dashboard** grants administrators access to exclusive controls, such as updating like counters, calculating daily likes and reviews, and updating rankings for popular content. They can also initiate database updates and manage the Most Popular page.

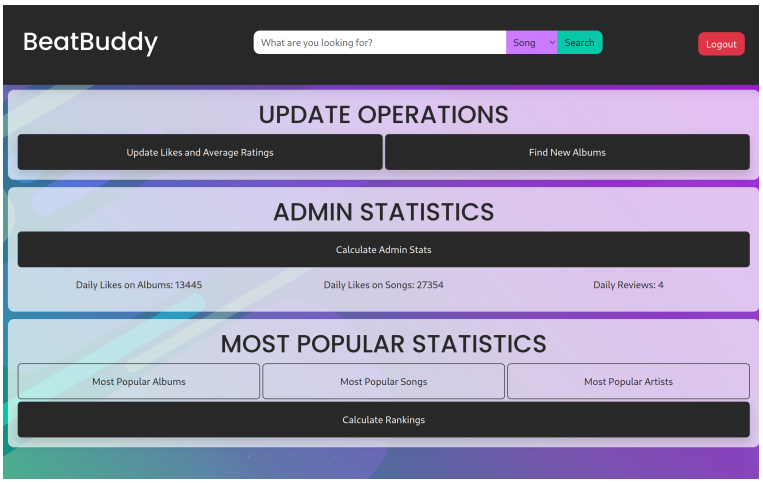


Figure 1.9. Admin Dashboard interface

2 DESIGN OVERVIEW

2.1 SYSTEM OVERVIEW AND SPECIFICATIONS

This chapter provides a detailed exploration of the key components and specifications that form the foundation of our system. Understanding the architecture and the interactions between various elements is crucial for the development, implementation, and scalability of the system.

2.2 MAIN ACTORS

Identifying and understanding the main actors interacting with our system is essential. Actors, which can be users, external systems, or hardware components, play pivotal roles in determining the system's use cases and ensuring it meets all stakeholders' needs. The main actors in our application are:

- **Unregistered Users:** These are visitors who browse through albums and artists on the website without engaging in interactive functionalities.
- **Registered Users:** Users who have signed up for our service and have full access to all its features.
- **Admins:** They manage the website, delve into detailed statistics, update like counters, and refresh the database with new albums.

2.3 REQUIREMENTS

2.3.1 Functional Requirements

This section delineates the specific functionalities and features that our system offers. These requirements are vital for guiding development and ensuring the system aligns with user needs.

Unregistered Users

The application should allow **unregistered users** to:

- **Register** within the application.
- Browse through popular Songs, Albums, and Artists, including:
  - Albums with high average ratings.
  - Albums and Songs with the most likes (both all-time and last week).
  - Artists with numerous likes on their albums or high average album ratings.
- Access detailed pages for selected Songs, Artists, or Albums, including information, statistics, track lists, and user reviews.
- Search for Albums and Songs, with access to relevant album pages upon selection.

Registered Users

Registered users should be able to:

- Log in and out of the application.
- View their personal page, which includes account information, liked and reviewed albums and songs, and followed users, with options to remove likes or unfollow.
- Access and interact with the Most Popular page.
- Perform searches like unregistered users, with additional abilities to like albums and songs, and write reviews.
- Search for other registered users and interact with their profiles.
- Access the Discover page for personalized song and user recommendations.

Admins

Admins should be able to:

- Log in and out of the application.
- Perform administrative tasks like updating likes, initiating database updates, and generating statistical reports.
- Calculate rankings for the most popular content.
- Browse and search content like unregistered users, with restricted interaction capabilities.

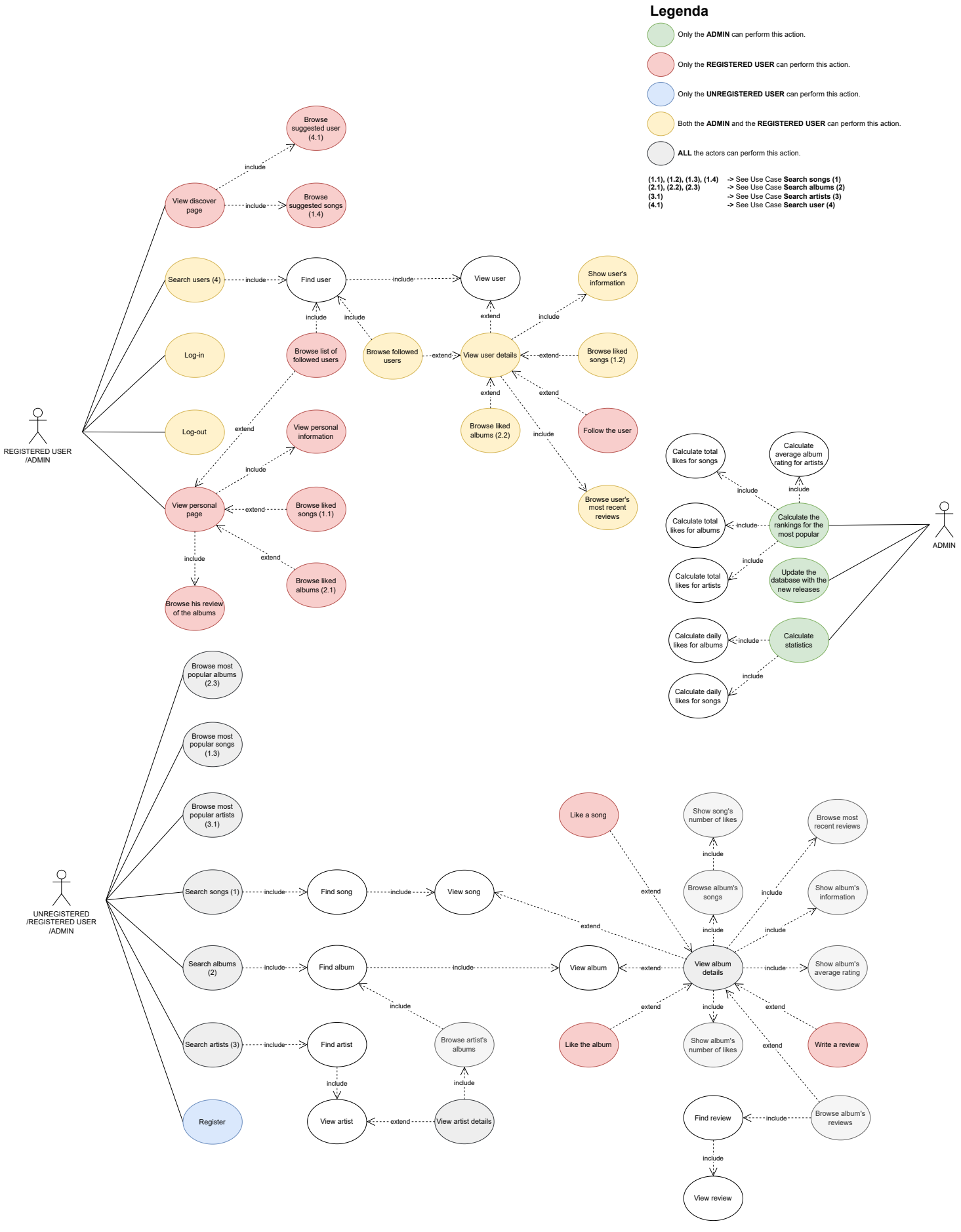
2.3.2 Non-functional Requirements

These requirements address the system's performance, reliability, scalability, and security, ensuring its efficiency and quality:

- Implementation as a web application.
- Minimization of single points of failure.
- Emphasis on high availability, even with occasionally outdated data.
- Use of Object-Oriented Programming languages for code development.
- Tolerance for data loss.
- Encryption of user passwords.

2.4 USE CASE DIAGRAM

To facilitate a comprehensive understanding of the system's structure and behavior, we employ Unified Modeling Language (UML) diagrams. These diagrams offer a graphical overview of the system's components, relationships, and processes.


Legenda
Only the ADMIN can perform this action.
Only the REGISTERED USER can perform this action.
Only the UNREGISTERED USER can perform this action.
Both the ADMIN and the REGISTERED USER can perform this action.
ALL the actors can perform this action.
(1.1), (1.2), (1.3), (1.4) -> See Use Case Search songs (1)
(2.1), (2.2), (2.3) -> See Use Case Search albums (2)
(3.1) -> See Use Case Search artists (3)
(4.1) -> See Use Case Search user (4)
REGISTERED USER /ADMIN
Browse suggested user (4.1)
View discover page
Browse suggested songs (1.4)
Search users (4)
Find user
View user
Show user's information
Log-in
Browse list of followed users
Browse followed users
View user details
Browse liked songs (1.2)
Log-out
View personal information
Browse liked albums (2.2)
Follow the user
View personal page
Browse liked songs (1.1)
Browse user's most recent reviews
Browse liked albums (2.1)
Browse his review of the albums
include
extend
ADMIN
Calculate total likes for songs
Calculate average album rating for artists
Calculate total likes for albums
Calculate the rankings for the most popular
Calculate total likes for artists
Update the database with the new releases
Calculate daily likes for albums
Calculate statistics
Calculate daily likes for songs
UNREGISTERED /REGISTERED USER /ADMIN
Browse most popular albums (2.3)
Browse most popular songs (1.3)
Browse most popular artists (3.1)
Search songs (1)
Find song
View song
Like a song
Show song's number of likes
Browse most recent reviews
Browse album's songs
Show album's information
Search albums (2)
Find album
View album
View album details
Show album's average rating
Like the album
Show album's number of likes
Write a review
Search artists (3)
Find artist
Browse artist's albums
Register
View artist
View artist details
Find review
Browse album's reviews
View review

2.5 UML CLASS DIAGRAM DIAGRAM

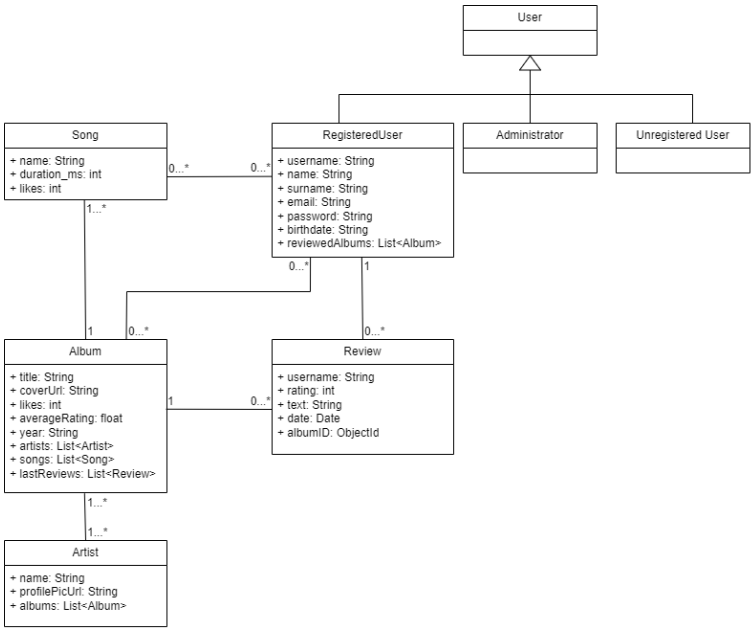


Figure 2.1. BeatBuddy UML class diagram

3 DATA MODELING AND STRUCTURE

3.1 DATASET CREATION

Adhering to the principle of data variety, our dataset was constructed from multiple sources. The foundation consisted of two datasets: one featuring 10,000 artists from Spotify sourced from **Kaggle**, and another comprising 1.5 million Amazon reviews on albums, augmented with approximately 20,000 reviews from musical magazines.

3.1.1 Web APIs

Spotify APIs

Leveraging the **Spotify APIs**, we gathered comprehensive details on artists, including their albums, tracklists, cover URLs, and other relevant information for our dataset.

MusicBrainz and FanArt.tv

FanArt.tv provided artist photos, retrievable using the MusicBrainzID (**MBID**) from **MusicBrainz**. This two-step process involved fetching MBIDs from MusicBrainz before accessing the images.

3.1.2 Building Documents

Reviews

Our review dataset had varying levels of detail. While a smaller subset directly referenced album names, facilitating document creation, the larger dataset lacked this specification. To realistically distribute reviews, we analyzed the **ASIN** distribution from Amazon, observing an exponential pattern. This analysis guided our document creation, blending reviews from different sources into a unified collection.

Users

In constructing user profiles, we preserved original usernames from reviews and supplemented them with randomly generated personal information using the `Faker` Python library. Passwords were hashed using `SHA256`, ensuring robust security.

Albums

Album documents were enriched with information retrieved from Spotify and other sources, including a field for recent reviews.

Artists

Artists' documents featured their albums and images sourced from *FanArt.tv*, essential for displaying on their dedicated pages.

3.2 SCRIPTS AND FUNCTIONS FOR DATASET ASSEMBLY

Python was chosen for its rich library support for data handling. The scripts, organized in the `scraping` directory, facilitated efficient dataset assembly.

3.2.1 Spotipy

We developed our scripts to surpass the limitations of the *Spotipy* library, especially regarding Spotify's rate-limit policies. Our approach optimized the number of requests needed to retrieve complete artist and album information.

3.2.2 Artist Retrieval

Scripts in the `buildArtist.py` file efficiently prepared artist data for database integration.

3.2.3 Reviews

The `studyOnDistribution.py` script analyzed review distribution, while `builder.py` assigned reviews to albums, generated user documents, and prepared review documents for database import.

3.2.4 Database Loading

Using `pymongo` and `neo4j` libraries, we efficiently uploaded our data to MongoDB and neo4j databases. These scripts are found in their respective folders.

3.2.5 Minor Fixes

Scripts for minor data corrections were not included in the repository, but main scripts were refined for increased reliability.

3.3 DATABASES

We decided to use **MongoDB** and **neo4j** for our application. In this chapter we'll go deeper into details about motivations of this choice, how we stored data and all the shrewdness we implemented.

3.3.1 Volume Considerations

We estimated a Daily Active User (DAU) rate at 20%, occasional access at 10%, and an average of 2.5 visits per day, resulting in approximately 25,000 daily site accesses.

% of Utilization	Action	# of Occurrences	MongoDB Ops.	neo4j Ops.	Total (per day)
Song Search	12	120'000	1	0	120'000
Album Search	7	70'000	1	0	70'000
Artist Search	5	50'000	1	0	50'000
Album Page Load	15	150'000	1 + 1[1]	0	(up to) 300'000
User Page Load	10	100'000	1	(up to)3[2]	(up to)400'000
Discover Load	3	30'000	0	3	90'000
Charts Load	5	50'000	0[3]	0[3]	50'000
Like an album	5	50'000	0	1	50'000
Like a song	10	100'000	0	1	100'000
Write a review	3	30'000	3	0	30'000
Load Album review	0.2	2'000	1	0	2'000

Table 3.1. Estimation of daily requests for each action

[1]: because reviews are loaded and displayed only upon request.
[2]: because lists of followed, liked albums and liked songs are loaded and displayed only upon request.
[3]: given the fact that the top-trends songs and albums are updated once a week, we save the results of these weekly queries, and use them to build the page without querying again the DB.

3.4 MONGODB

In addition to being a project requirement, we would have chosen MongoDB to take advantage of the denormalization possibility that a Document DB offers. Its ability to store related data together in a single document, rather than spreading it across tables, simplifies our queries and improves performance. Furthermore, MongoDB's scalability is well-suited to handle the potential growth of our application, with efficient data indexing and powerful aggregation frameworks to manage large volumes of data. Lastly, the JSON-like format of BSON used in MongoDB is intuitive and aligns well with the data structures used in our application, facilitating seamless data integration and manipulation.

3.4.1 MongoDB Collections

Four are the collections used to store and organize our data:

- Users
- Albums
- Artists
- Reviews

Users Collection

An example of a document stored in this collection is the following:


```
{
  "_id": {
    "$oid": "65ab9d44186280a9f3ebc2b5"
  },
  "name": "Ciro",
  "surname": "Paoletti",
  "username": "ttm",
  "password": "11452bbd5cdb25777ba3873d0ff07ebff4982a63bdec19039f535df9f79d2519",
  "birthDate": "1971-08-31",
  "email": "ttm-ciro@Saunders-Riley.com",
  "reviewedAlbums": [
    {
      "artist": "Nipsey Hussle",
      "rating": 5,
      "coverUrl": "https://i.scdn.co/image/ab67616d0000b273cbd8024c0293e4893adf4db1",
      "albumTitle": "X-Tra Laps"
    }
```

```
  ]
}
```

The highlight of this document, despite the personal data, is the `array` **reviewedAlbums**. The latter represent a *one-to-many* relation between **Users** and the **Reviews** they wrote: that choice allowed us to show a summary of their reviews in the User page, either if viewed by their own or by another user, and this **without accessing to another collection**. In this way a user can see them, to make a fast idea of the interests of another user.

Album Collection

An example of a document stored in this collection is the following:

```
{
  "_id": {
    "$oid": "65ab9d6c186280a9f3ecedef"
  },
  "artists": [
    "My Chemical Romance"
  ],
  "averageRating": 4.4,
  "coverURL": "https://i.scdn.co/image/ab67616d0000b273e059074baf87ec49b883e127",
  "last_reviews": [
    {
      "text": "It was great!  The old becomes new once again!  It was almost
      when I started to like them! Very good!",
      "rating": 5,
      "username": "Marion Tracy Barnwell II"
    },
    ...
    {
      "text": "Probably my favorite album since their reunion in 2000.",
      "rating": 5,
      "username": "amazon customer"
    }
  ],
  "likes": 32,
  "songs": [
    {
      "duration_ms": 154306,
      "likes": 7,
      "name": "The End. - Live in Mexico City"
    },
    ...
```

```
    {
      "duration_ms": 81653,
      "likes": 2,
      "name": "Blood - End Credits"
    }
  ],
  "title": "The Black Parade Is Dead!",
  "year": "2008"
}
```


Given by the fact that we decided to store Reviews in a standalone collection (and we'll discuss this choice in the paragraph dedicated to the Reviews collection), we decided to include an array which embeds the five last reviews leaved on that album, and to display them in the Album main page, to enhance the user experience.
The field `artists` is an array because our will was, for joint albums, to show every artist who worked on it.

Artists Collection

An example of a document stored in this collection is the following:


```
{
  "_id": {
    "$oid": "65afcd0328ee62efd4c53988"
  },
  "name": "Il Pagante",
  "image": "http://assets.fanart.tv/fanart/music/abb60b9d-
        9d5f-457c-86ed-8ae9be5173d9/albumcover/entro-in-pass-5a99615838952.jpg",
  "albums": [
    {
      "title": "Devastante",
      "coverURL": "https://i.scdn.co/image/ab67616d0000b273ceaeb9d72a6398f943ca9ad8"
    },
    {
      "title": "Paninaro 2.0",
      "coverURL": "https://i.scdn.co/image/ab67616d0000b273ef32132e1bd4bbababcf23a1"
    },
    {
      "title": "Entro in pass",
      "coverURL": "https://i.scdn.co/image/ab67616d0000b273df3be90c582ebe8addc5c09d"
    }
  ]
}
```


List of albums has been included to display them on the artist main page.

Reviews Collection

An example of a document stored in this collection is the following:


```
{
  "_id": {
    "$oid": "65ab9d80186280a9f3ed6fd5"
  },
  "rating": 5,
  "text": "This is the original formation, with david lee
      roth on vocals. What can go wrong? Amazing guitars,
      revolutionary solos, its the band on its best shape!",
    "albumID": {
    "$oid": "65ab9d7a186280a9f3ed47cc"
  },
  "username": "Vladimir C. Carvalho",
  "date": {
    "$date": "2023-05-11T00:35:25.000Z"
  }
}
```


3.4.2 Rationale for Separate Reviews Collection

Managing Review Growth and Database Efficiency

The decision to create a separate collection for reviews was primarily driven by the need to manage the rapid growth of review data effectively and maintain database efficiency.

Rapid Growth of Reviews: Reviews represent a highly dynamic aspect of our application, with the potential for rapid and substantial growth. Storing them within album documents would lead to a significant increase in document size over time. This increase would not only strain database resources but also impact the performance of read operations, as larger documents take longer to process and retrieve.

Performance Considerations: Separating reviews into their own collection allows us to maintain leaner and more efficient album documents. By avoiding the embedding of all reviews, we prevent the bloating of album documents and ensure faster access to essential album information, which is more frequently sought by users.

Strategic Embedding of Recent Reviews in Albums

While we store the majority of reviews separately, we strategically embed the five most recent reviews within each album document.

User Interaction Patterns: This decision is informed by user interaction patterns, where most users are interested in recent reviews rather than an exhaustive list. Embedding the latest reviews offers a quick snapshot of current opinions, enhancing the user experience by providing immediate insights without the need for additional queries.

Balancing Document Size and Accessibility: By embedding only a limited number of reviews, we strike a balance between keeping album documents manageable in size and providing convenient access to recent reviews. This approach ensures that the album documents are not excessively large, thus maintaining efficient read operations.

Review Details in User Collection

In addition to the separate reviews collection, we include selected details of user-authored reviews in the User collection.

Enhancing User Experience: This embedding offers users immediate access to their own review history and highlights their engagement within the community. It enhances the user experience by providing quick access to personal contributions, encouraging user interaction and engagement.

In summary, our approach to managing reviews reflects a thoughtful balance between operational efficiency and user experience. It ensures optimal database performance while providing users with convenient access to relevant review data, aligning with our application's social networking nature.

3.5 NEO4J: LEVERAGING GRAPH DATABASE ADVANTAGES

Our application's social network features made neo4j an ideal choice. The following sections detail the concepts, entities, and relationships within our neo4j setup.

3.5.1 Concepts

For avoiding useless redundancy between MongoDB and neo4j, we choose to implement likes and follwowing relations **only on neo4j**: a policy of our service is to not show real-time number of likes (these policies are pretty common in other services, such as Spotify which didn't show the number of plays of a song update in real time), to let users focus on their actual taste. Making virtue of necessity, we decide to implement likes and follows relationships only on neo4j, updating

these counters in MongoDB once a day: however, we'll explain and justify this choice in a further chapter.
Furthermore, by exploiting this type of relationship for queries to recommend new content, we avoid, as said before, the redundancy of having the same relationships in both databases, thus embracing the philosophy of eventual consistency, but sacrificing punctual updating, which still falls within the requirements of the application.

3.5.2 Entities

User

User entities in neo4j are simplified, focusing primarily on the username.

Song

Song entities contain essential details like artist, album, and cover URL, aiding in query clarity and user experience: the application shows songs retrieved from neo4j only in the User personal page and in the Discover page. In both cases is useful for the user to visualize the info about the albums they belongs, helping the experience, and this without accessing to mongoDB.

Album

Album entities mirror the considerations made for songs, maintaining consistency across the database.

3.5.3 Relationships

We defined three main relationship types: `FOLLOW`, `LIKES_A`, and `LIKES_S`, each with specific domains and characteristics.

`FOLLOW`

The domain of this relations is `(u1:User) -[r:FOLLOW]-> (u2:User)`, and it's not bidirectional (e.g. u1 follows u2, but u2 didn't follow them back).

In case two Users follow each other, there will be two different arches: we decided to follow this path, instead of, for example, add a property `FOLLOWS_BACK = True` in the relation, because it's faster for neo4j to identify only the outgoing relations instead of considering both outgoing and the incoming ones, and then check if the incoming relations have that property. Our solution, which involves two different arches, is more expensive in terms of memory, but allows faster execution of some crucial queries.

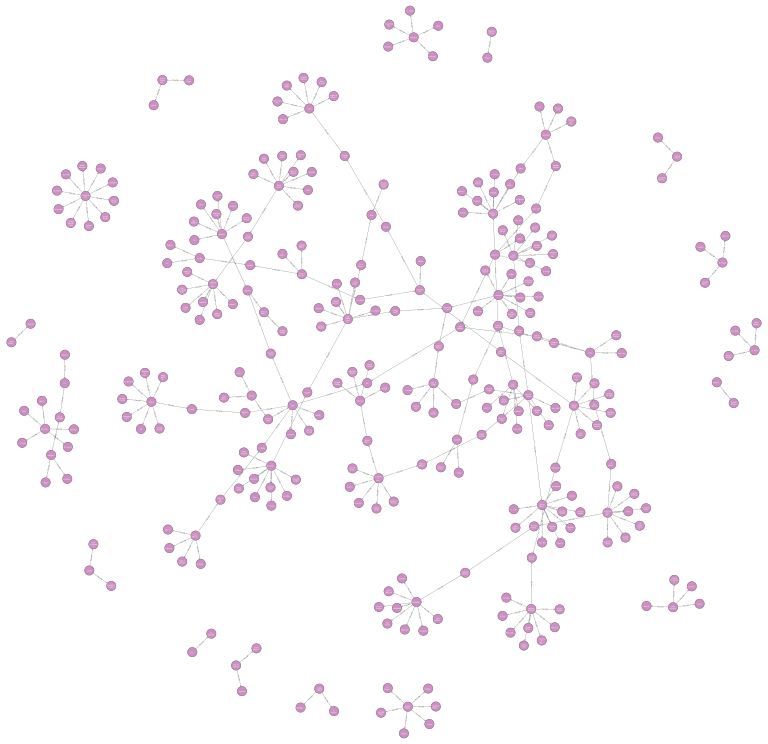


Figure 3.1. User nodes linked with `FOLLOW` relations

`LIKES_A`

The domain of this relations is `(u:User) -[r:LIKES_A]-> (a:Album)`, and it's unidirectional, because of it wouldn't have any sense for an Album to likes a User. The relation also memorize the timestamp of its creation (i.e. when a User likes an Album), which is used in two different main queries we'll discuss further.

`LIKES_S`

The domain of this relations is `(u:User) -[r:LIKES_A]-> (a:Album)`, and it has the same characteristics as the previous one.

Relation between Albums and Likes

In designing our graph database schema, we made a strategic decision to not explicitly define a belonging relationship between Song and Album entities. This decision was driven by our desire to avoid falling into a relational model pattern, which would not fully leverage the strengths of a graph database like neo4j.
By not establishing a direct relationship between Song and Album, we circumvent the more time-consuming process of navigating from a song to its respective album merely to retrieve information, without the actual utilization of the Album node. This approach may introduce a

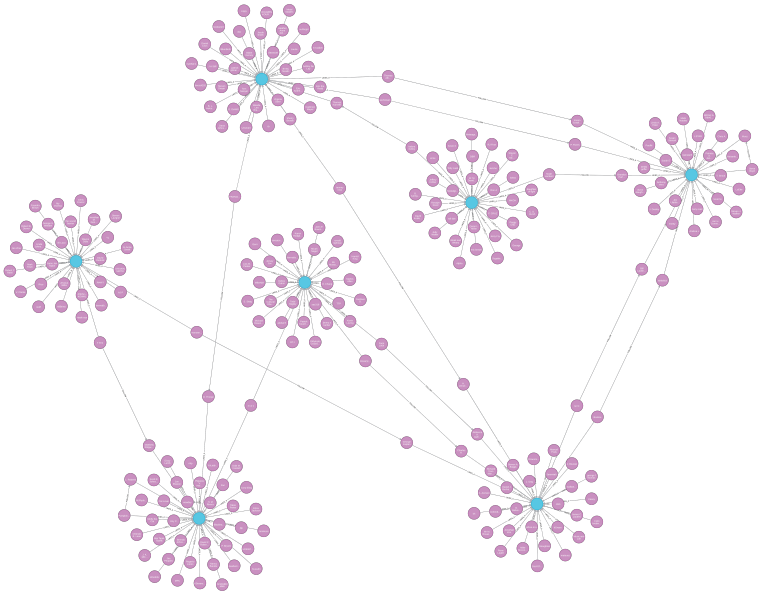


Figure 3.2. Snippet from neo4j: Album nodes (in blue) , User nodes (in violet) with both `FOLLOW` and `LIKES_A` relations

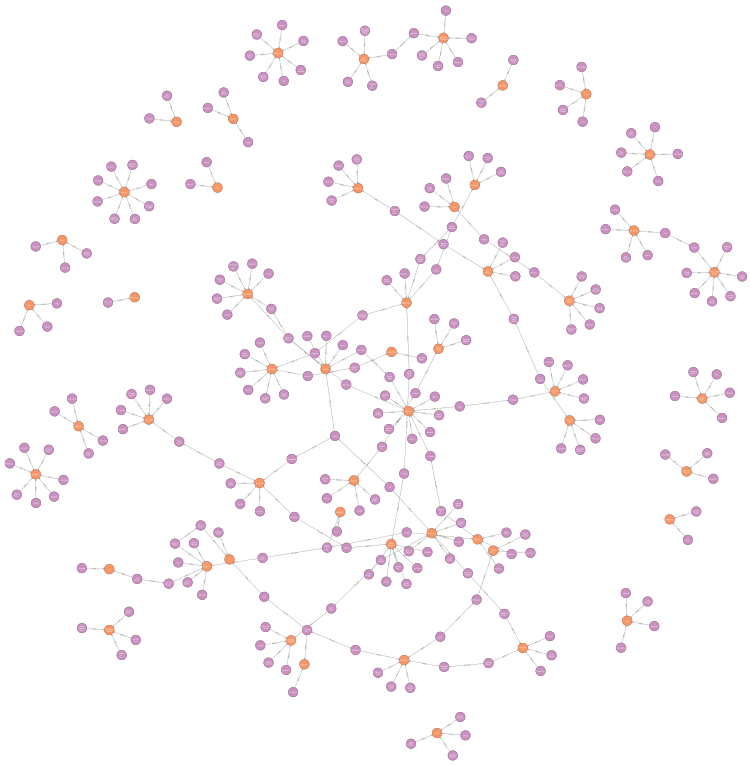


Figure 3.3. Snippet from neo4j: Song nodes (in orange) , User nodes (in violet) with both `FOLLOW` and `LIKES_S` relations

certain level of redundancy, as information about the album is stored within each Song entity, but it significantly enhances the efficiency of our queries.
Our schema design is also a reflection of our application's conceptual model, which prioritizes user interactions over hierarchical data structures. In our social networking, the most meaningful interactions are between users and songs or albums, rather than between songs and albums themselves. This perspective guided our decision to forgo the song-to-album relationship in favor of direct user-to-song and user-to-album links.

4 IMPLEMENTATION

4.1 IMPLEMENTATION FRAMEWORKS

As the cornerstone of our project, we decided to employ the Spring Boot framework, which has greatly facilitated our development process. The most significant reason for embracing Spring Boot is its seamless integration of an embedded Tomcat web server. This feature has made it incredibly convenient for us to develop a web server effortlessly. Moreover, we have found the utilization of `HttpSession` in combination with the `@Controller` and `@RestController` annotations for mapping HTTP protocol requests to Java functions to be exceptionally beneficial. This approach has not only streamlined our development efforts but has also offered us the flexibility and efficiency we needed for handling HTTP requests and responses within our application. In summary, Spring Boot has played a pivotal role in simplifying the development and deployment of our web-based components.

4.1.1 MVC Design Pattern

We have decided to structure our project according to the Model-View-Controller (MVC) pattern for various reasons. The MVC pattern provides a clear separation of responsibilities within the application, which proves crucial when developing a non-trivial application.
The Model manages the data and business logic, ensuring efficient data handling and facilitating future modifications. The View handles the user interface, allowing for flexible design and easy maintenance of the application's visual appearance. Finally, the Controller acts as an intermediary between the Model and the View, managing the flow of data and user interactions.
This organizational structure has also helped us maintain cleaner and more understandable code, facilitating collaboration among members of the development team and software debugging.

4.2 PROJECT STRUCTURE AND PACKAGE ORGANIZATION

In our project, we have structured our codebase to ensure clear separation of concerns and ease of navigation. The code is organized into several packages, each with a specific responsibility within the application.

4.2.1 it.unipi.lsmd.BeatBuddy.config

This package contains configuration classes that are responsible for setting up and customizing the behavior of our application. This includes configurations for data sources, security, and any third-party integrations.

- MongoConfig: responsible for configuring MongoDB settings using Spring Boot's @Configuration annotation. It defines a MongoTemplate bean for interaction with MongoDB;
- Neo4jConfig: this configuration class is used to set up Neo4j-related settings using Spring Boot's @Configuration annotation;
- ScheduledTasks: It defines a scheduled task, `dailyDatabaseUpdate()`, that runs once a week. Its purpose is to retrieve and insert new songs and albums into the database.

4.2.2 it.unipi.lsmd.BeatBuddy.controllers

Controllers are the entry point for handling incoming HTTP requests. This package hosts all the controller classes that manage the interaction between the user and the application's services. These classes are annotated with @Controller and are responsible for returning HTML pages to the user, which are sometimes dynamically modified using Thymeleaf.

- `AdminPage_Ctrl`
- `AlbumDetailsPage_Ctrl`
- `AlbumReviews_Ctrl`
- `ArtistDetailsPage_Ctrl`
- `DiscoverPage_Ctrl`
- `ERROR_Ctrl`
- `HomePage_Ctrl`
- `IndexPage_Ctrl`
- `LoginPage_Ctrl`
- `MostPopularPage_Ctrl`
- `ProfilePage_Ctrl`
- `SignupPage_Ctrl`
- `UserPage_Ctrl`
- `WriteReviewPage_Ctrl`

```java
@Controller
public class ProfilePage_Ctrl {

    @Autowired
    User_Repo_MongoDB user_Repo;

    @RequestMapping("/profilePage")
    public String profilePage(HttpSession session,
                              Model model){

        model.addAttribute("logged", (Utility.isLogged(session)) ? true : false);

        if(!Utility.isLogged(session))
            return "redirect:/login";
        else if(Utility.isAdmin(session))
            return "redirect:/adminPage";
        else{
            User optionalUser = user_Repo.getUserByUsername(Utility.getUsername(session));
            if(optionalUser==null)
                return "error/genericError";
            model.addAttribute("userDetails", optionalUser);
            return "profilePage";
        }
    }
}
```

Figure 4.1. ProfilePage controller

4.2.3 it.unipi.lsmd.BeatBuddy.controllers.api

Here, we have classes responsible for responding to user requests that do not require an entire HTML page but only data necessary for internal functionalities within the page itself. All these classes are annotated with @RestController and send data to the user in JSON format.

- `AdminPage_RESTCtrl`: Responds to admin requests, primarily focused on updating the current database state or handling heavy aggregation requests.
- `Login_RESTCtrl`: Handles user login requests.
- `Neo4jBasicOperations_RESTCtrl`: Responds to user requests involving basic operations on Neo4j, such as adding a user's like to an album.
- `ProfilePage_RESTCtrl`: Within a user's profile page, users can access various pieces of information related to their activities, such as lists of songs or albums they have liked. This class is responsible for providing these lists.
- `Search_RESTCtrl`: This class comes into play when a user performs a search in the search bar. It is responsible for returning a list of items from which the user can choose.
- `Signup_RESTCtrl`: Manages registration requests from new users to the website.
- `WriteReview_RESTCtrl`: This class is used when a user wants to write a new comment on an album.

```java
@RestController
public class Search_RESTCtrl {

    @Autowired
    Album_Repo_MongoDB album_RepoMongoDB;

    @Autowired
    Artist_Repo_MongoDB artist_RepoMongoDB;

    @Autowired
    User_Repo_MongoDB user_RepoMongoDB;

    @GetMapping("/api/search")
    public @ResponseBody String search(@RequestParam("term") String term,
                                       @RequestParam("category") String category){

        if(category.equals("album")){
            return new Gson().toJson(album_RepoMongoDB.find5AlbumsDTO(term));
        }
        else if(category.equals("artist")){
            return new Gson().toJson(artist_RepoMongoDB.find5ArtistsDTO(term));
        }
        else if(category.equals("song")){
            return new Gson().toJson(album_RepoMongoDB.find5SongsDTO(term));
        }
        else if(category.equals("user")){
            return new Gson().toJson(user_RepoMongoDB.find5UserDTO(term));
        }
        else
            return "Invalid category";
    }
}
```

Figure 4.2. `Search_REST` controller

4.2.4 it.unipi.lsmd.BeatBuddy.controllers.model

The model package defines our domain model which captures the business data and the operations that will act upon it. This is the heart of our business logic, encompassing the primary business objects. Specifically, within this package, we find classes that correspond on a one-to-one basis with the documents in the various collections present in MongoDB. This approach, having classes that mirror the attributes of the collection documents, simplifies the interaction with the database, thanks to the features provided by Spring Boot.

- `Album`: This class represents the documents within the "albums" collection. It contains all the information we wish to present to the user about a specific album. For instance, it includes the "songs" array, which holds details about the various tracks that comprise the album.
- `Review`: This class corresponds to documents within the "reviews" collection. It stores the components of a review made by a user on a particular album, including the user's username, the name of the album reviewed, the text of the review, and the assigned rating.
- `Artist`: This class represents the documents from the "artists" collection. It thus contains specific data about an artist and includes an array detailing their discography, providing only the essential information (the complete details of each album are not included here).
- `User`: This class represents the documents from the "users" collection. Here, specific information about an individual user is maintained, such as their username, email, etc.

As with MongoDB, for Neo4j, we also utilize classes that reflect the various properties of nodes with a certain label in our graph database one-to-one. This design allows us to leverage Spring Boot's capabilities to simplify the developer's work.

- `Album_Neo4j`: mirrors the properties of nodes with the **Album** label.
- `Song_Neo4j`: mirrors the properties of nodes with the **Song** label.
- `User_Neo4j`: mirrors the properties of nodes with the **User** label.

Beyond the aforementioned classes, we have several others of lesser prominence:

- `ReviewedAlbum`: used within the User class to record the reviews made by users.
- `ReviewLite`: employed within the Album class to contain the main information of the latest reviews related to the specific album.
- `UserForRegistration`: a class similar to User but with fewer attributes, utilized during the registration phase.

4.2.5 it.unipi.lsmd.BeatBuddy.model.dummy

Within this package, you will find classes primarily used for mapping query or aggregation results. These classes are quite simple and typically contain only constructors and methods for performing mapping operations.

- AdminStats
- AlbumOnlyAvgRating
- AlbumOnlyLikes
- AlbumWithLikes
- ArtistWithAvgRating
- ArtistWithLikes
- SongOnlyLikes
- SongWithLikes

4.2.6 it.unipi.lsmd.BeatBuddy.repository

The repository package in our project follows a strategic design to abstract the data layer, providing a clear separation between the different data storage technologies we use: MongoDB and Neo4j. This organization ensures that data access is consistent and optimized for each database's strengths.

MongoDB Subpackage

The MongoDB subpackage encapsulates all interfaces that define the operations to be performed on the MongoDB database. These interfaces extend from Spring Data MongoDB repositories, allowing us to leverage Spring Data's powerful repository abstraction:

- Album_Mongolnterf: This interface manages data access for album documents within the MongoDB Albums collection. It defines the necessary basic operations to retrieve album data.
- Artist_Mongolnterf: Similar to the album interface, this one handles the data access for artist documents, specifically tailored to the Artists collection in MongoDB. Here we find simple queries like findAlbumByUsername, which returns the information present in a certain document of the Albums collection.
- Review_Mongolnterf: This interface focuses on the Reviews collection, detailing the methods required for fetching review data tied to user feedback on albums.

- `User_Mongolnterf`: It provides the data access methods for user documents within the "users" collection, catering to operations related to user account data. Here too we find only very simple queries, such as the one that returns the data of a certain user starting from their username.
- `UserForRegistration_Mongolnterf`: This specialized interface caters to the registration process, handling the initial data access operations needed when users sign up for the service.

Neo4j Subpackage

The Neo4j subpackage is dedicated to defining repositories for our graph database, allowing us to manage the data in a way that capitalizes on the relationships and connections between data points:

- `Album_Neo4Interf`: This repository interface manages the nodes labeled as Album within Neo4j, providing methods to navigate and manipulate album-related data within the graph structure.
- `Song_Neo4Interf`: Corresponding to the Song nodes in Neo4j, this interface defines operations to interact with song data, taking advantage of the graph database's capabilities to explore the connections between songs and other entities.
- `User_Neo4Interf`: This interface handles the User nodes, providing methods for accessing and managing user-related data within the graph. It focuses on the relationships users have with other nodes, such as albums and reviews. For example, here you can find queries to store a user's likes towards albums or songs, or relationships like user follows.

Other classes belonging to the package

We now move on to the classes, which utilize the interfaces of the two databases to execute more complex operations. It is within these classes that we find operations requiring a greater effort from the database, such as aggregations or more complex queries.

- `Album_Repo_MongoDB`: this class contains various aggregations primarily requested by the system administrator. These aggregations might involve compiling statistics on album sales, user engagement, or other analytics that help in decision-making and reporting.
- `Album_Repo_Neo4j`: in this class, we find the most significant and computationally intensive queries related to albums in Neo4j.
- `Artist_Repo_MongoDB`: this class holds two crucial aggregations related to artists. The purpose is to identify trending artists by analyzing data like streaming counts, social media mentions, or recent sales figures.

- `Review_Repo_MongoDB`: rather than aggregations, this class includes a range of queries, from simple to complex, designed to retrieve reviews for a specific album. These queries may sort, filter, or prioritize reviews based on certain criteria, such as recency or helpfulness.
- `Song_Repo_Neo4j`: this class handles complex queries related to songs. It makes use of Neo4j's graph structure to execute queries that may involve traversing relationships, such as the connection between songs and user preferences. This way, we can generate recommendations.
- `User_Repo_MongoDB`: here, we find queries that enable the registration of new users or update information for existing users, like storing their new reviews. This class ensures that user interactions are reflected in real-time and that the user profiles are up-to-date.
- `User_Repo_Neo4j`: this class mainly utilizes functions from the Neo4j user interface, encapsulating them into methods that can be invoked by other classes within the code. This might include complex graph queries to understand user behavior or to map social connections between users.

4.2.7 `it.unipi.lsmd.BeatBuddy.utility`

The constants and Utility classes serve as fundamental components within our project, centralizing shared resources and functionalities to promote clean code practices and ease of maintenance.

- `Constants`: As indicated by its name, this class serves the sole purpose of maintaining the values of various constants used in the project. We chose to store them all here for easy access, making code maintenance straightforward.
- `Utility`: In this class, can be find functions used by multiple classes to perform operations unrelated to a specific type of document. For example, functions for reading and writing to files can be found here, as well as simpler functions that may be used to check if a user is logged in or retrieve their username.

4.2.8 Static folder

The static folder in our project is a fundamental part of the resource directory and is pivotal for serving static content to the client's browser. It is designed to store all the client-side assets that don't require server-side processing before being sent to the client. Here's a breakdown of its structure and the purpose of each subfolder:

- `css`: This subdirectory contains Cascading Style Sheets (CSS) files, which are used to style the web pages served to the client. CSS is responsible for the look and feel of the website, including layouts, colors, fonts, and other design elements.
- `img`: The image folder stores graphical content such as icons, logos, photos, and any other images that are used in the application's user interface. These images are referenced by the HTML and CSS files to enhance the visual aspect of the web pages.

- `js`: Short for JavaScript, this subdirectory holds all the JavaScript files. JavaScript is used to add interactivity to web pages, from handling form submissions to dynamically updating content without the need to refresh the page.

4.2.9 Templates folder

The templates directory of our project plays a crucial role in the user interface layer of the application. It contains HTML pages that are served to the user by the server. Within this directory, there are subdirectories that categorize the templates based on their use or the type of user interaction they facilitate. The HTML templates in this directory often utilize **Thymeleaf**, which is a modern server-side Java template engine for web environments. Thymeleaf enables the dynamic generation of HTML pages where the content can be rendered on the server before being sent to the client. This approach allows the server to inject dynamic data into the HTML templates, such as user information, query results, or form data, which enhances the interactivity and personalization of the user experience. For instance, `albumDetails.html` display information about an album, dynamically filled with data from the server's database. Similarly, albumReviews.html would list reviews for an album, with content generated based on the current data available for that particular album.

4.3 MONGODB RELEVANT OPERATIONS

4.3.1 Write a review

To write a review, the application has to call the following methods:

- `public @ResponseBody String writeReview(HttpSession session, @RequestParam("rating") int rating, @RequestParam("text") String text, @RequestParam("albumID") String albumID, @RequestParam("username") String username)`
- `public boolean existsById(String id)`
- `public boolean existsByAlbumIDAndUsername(String albumID, String username)`
- `public boolean insertReviewIntoUser(String username, ReviewedAlbum reviewedAlbum)`
- `public int insertReviewIntoAlbum(String albumID, String username, int rating, String text)`

The first method is a @RestController and its task is to handle the POST request perfomed when a user add a review for an album.

Before adding a review to the database, the application must check if the album exists and if there is yet a review for that album (methods 2 and 3). After this controls, the system has to add the

```java
public boolean insertReview(int rating, String text, ObjectId albumObjectId, String username) {
    try {
        Review review = new Review(rating, text, albumObjectId, username, new Date());
        review_RI_MongoDB.save(review);
        return true;
    } catch (DataAccessException dae) {
        if (dae instanceof DataAccessResourceFailureException)
            throw dae;
        dae.printStackTrace();
        return false;
    }
}
```

Figure 4.3. Function to add the document into the Reviews collection

review both in three different collections: Album, Review and User. Since we maintain a list of the User's reviews and a list of "most recent" reviews in the Album, we have to maintain all the collections consistent.

```java
public boolean insertReviewIntoUser(String username, ReviewedAlbum reviewedAlbum) {
    try {
        addReviewedAlbumAtStart(username, reviewedAlbum);
        return true;
    } catch (DataAccessException dae) {
        if (dae instanceof DataAccessResourceFailureException)
            throw dae;
        dae.printStackTrace();
        return false;
    }
}
```

Figure 4.4. Function to update the User document with their last review

4.3.2 Load an Album page

To load an album page, the system calls the following methods:

- `public String albumDetails(HttpSession session, Model model @RequestParam(required = false) String albumId, @RequestParam(required = false) String artist, @RequestParam(required = false) String title)`
- `public Album getAlbumById(String id)` or
  `public Album getAlbumByTitleAndArtist(String title, String artist)`

The first method is a @Controller, and it handles the GET request when a user accesses an album page. To construct the page, one of the functions `getAlbumById(String id)` or `getAlbumByTitleAndArtist(String title, String artist)` must be invoked. The choice of which function to call is determined by the parameters of the first method. If the method used to retrieve the album returns an Album object, the album's details are added to the model, and the page is dynamically loaded using Thymeleaf. We made this decision because when a user want to see the details of the album page the system need to reload every time the page.

```java
public int insertReviewIntoAlbum(String albumID, String username, int rating, String text) {
    try {
        Album album = getAlbumById(albumID);
        if (album == null) {
            return 1; // Album non trovato
        }

        LinkedList<ReviewLite> lastReviews = new LinkedList<>(Arrays.asList(album.getLastReviews()));
        lastReviews.addFirst(new ReviewLite(username, rating, text));
        while (lastReviews.size() > 5) {
            lastReviews.removeLast();
        }

        album.setLastReviews(lastReviews.toArray(new ReviewLite[0]));
        album_RI_Mongo.save(album);
        return 0; // Successo

    } catch (DuplicateKeyException e) {
        e.printStackTrace();
        return 2; // Violazione del vincolo di unicità
    } catch (DataIntegrityViolationException e) {
        e.printStackTrace();
        return 3; // Violazione dell'integrità dei dati
    } catch (DataAccessException dae) {
        if (dae instanceof DataAccessResourceFailureException) {
            throw dae;
        }
        dae.printStackTrace();
        return 4; // Altre eccezioni relative all'accesso ai dati
    }
}
```

Figure 4.5. Function to update the Album document with the most recent review

With this approach, we generate the server-side page every time, which gives us more efficiency than handling client-side page loading. The `findById(String id)` method is provided by `MongoRepository<Album, String>` Spring interface. The `findByTitleAndArtist(String title, String artist)` method, instead, is defined by us:

4.4 NEO4J GRAPH-DOMAIN QUERIES

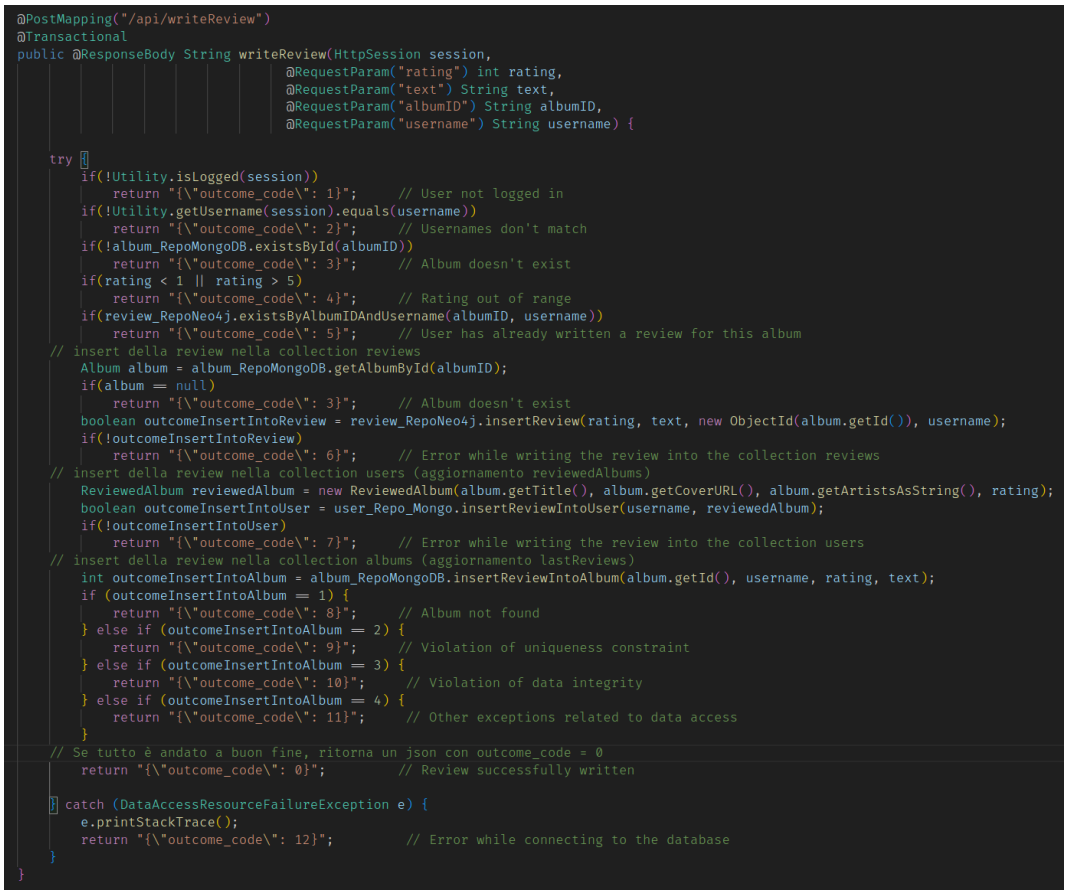
4.4.1 CRUD operations

We designed CRUD queries in order to manage the insertions of new nodes (e.g. a user registers the the application), add a relation between existing nodes (e.g. a user likes an album), or delete an existing relation in the graph (e.g. a user didn't like a song anymore). We present two example queries that respectively add and removes a like to an album:

4.4.2 Main neo4j queries

On-Demand queries

These queries, as the title said, are performed only on-demand, and they're usually used to retrieve entities that have relations with a given user. A real usage of these queries can be found when a user is in its personal page, or a page of another user: they can press the button *Liked*



```
@PostMapping("/api/writeReview")
@Transactional
public @ResponseBody String writeReview(HttpSession session,
                                        @RequestParam("rating") int rating,
                                        @RequestParam("text") String text,
                                        @RequestParam("albumID") String albumID,
                                        @RequestParam("username") String username) {

    try {
        if(!Utility.isLogged(session))
            return "{\"outcome_code\": 1}";      // User not logged in
        if(!Utility.getUsername(session).equals(username))
            return "{\"outcome_code\": 2}";      // Usernames don't match
        if(!album_RepoMongoDB.existsById(albumID))
            return "{\"outcome_code\": 3}";      // Album doesn't exist
        if(rating < 1 || rating > 5)
            return "{\"outcome_code\": 4}";      // Rating out of range
        if(review_RepoNeo4j.existsByAlbumIDAndUsername(albumID, username))
            return "{\"outcome_code\": 5}";      // User has already written a review for this album
    // insert della review nella collection reviews
        Album album = album_RepoMongoDB.getAlbumById(albumID);
        if(album == null)
            return "{\"outcome_code\": 3}";      // Album doesn't exist
        boolean outcomeInsertIntoReview = review_RepoNeo4j.insertReview(rating, text, new ObjectId(album.getId()), username);
        if(!outcomeInsertIntoReview)
            return "{\"outcome_code\": 6}";      // Error while writing the review into the collection reviews
    // insert della review nella collection users (aggiornamento reviewedAlbums)
        ReviewedAlbum reviewedAlbum = new ReviewedAlbum(album.getTitle(), album.getCoverURL(), album.getArtistsAsString(), rating);
        boolean outcomeInsertIntoUser = user_Repo_Mongo.insertReviewIntoUser(username, reviewedAlbum);
        if(!outcomeInsertIntoUser)
            return "{\"outcome_code\": 7}";      // Error while writing the review into the collection users
    // insert della review nella collection albums (aggiornamento lastReviews)
        int outcomeInsertIntoAlbum = album_RepoMongoDB.insertReviewIntoAlbum(album.getId(), username, rating, text);
        if (outcomeInsertIntoAlbum == 1) {
            return "{\"outcome_code\": 8}";      // Album not found
        } else if (outcomeInsertIntoAlbum == 2) {
            return "{\"outcome_code\": 9}";      // Violation of uniqueness constraint
        } else if (outcomeInsertIntoAlbum == 3) {
            return "{\"outcome_code\": 10}";     // Violation of data integrity
        } else if (outcomeInsertIntoAlbum == 4) {
            return "{\"outcome_code\": 11}";     // Other exceptions related to data access
        }
    // Se tutto è andato a buon fine, ritorna un json con outcome_code = 0
        return "{\"outcome_code\": 0}";          // Review successfully written

    } catch (DataAccessResourceFailureException e) {
        e.printStackTrace();
        return "{\"outcome_code\": 12}";         // Error while connecting to the database
    }
}
```

Figure 4.6. Rest controller for inserting a review into the database

```java
@GetMapping("/albumDetails")
public String albumDetails(HttpSession session,
                           Model model,
                           @RequestParam(required = false) String albumId,
                           @RequestParam(required = false) String artist,
                           @RequestParam(required = false) String title) {
    Album album;

    if (albumId != null) {
        // Logica per gestire la richiesta basata su albumId
        album = album_RepoMongoDB.getAlbumById(albumId);
    }
    else if (artist != null && title != null) {
        // Logica per gestire la richiesta basata su artist e title
        album = album_RepoMongoDB.getAlbumByTitleAndArtist(title, artist);
    }
    else {
        // Caso in cui nessuno dei parametri è fornito
        return "error/albumNotFound";
    }

    if(album == null)
        return "error/albumNotFound";
    else {
        //calcolo le durate delle canzoni in min e sec
        album.calculateDurations_MinSec();

        model.addAttribute("albumDetails", album);
        model.addAttribute("albumTotalDuration", album.calculateTotalDuration());
    }

    model.addAttribute("logged", (Utility.isLogged(session)) ? true : false);
    model.addAttribute("admin", (Utility.isAdmin(session)) ? true : false);

    return "albumDetails";
}
```

Figure 4.7. Controller for the load of the page

```java
public Album getAlbumById(String id){
    try {
        Optional<Album> result = album_RI_Mongo.findById(id);
        return result.orElse(null);
    } catch (DataAccessException dae) {
        if (dae instanceof DataAccessResourceFailureException)
            throw dae;
        dae.printStackTrace();
        return null;
    }
}

public Album getAlbumByTitleAndArtist(String title, String artist){
    try {
        List<Album> result = album_RI_Mongo.findByTitleAndArtist(title, artist);
        if (result.isEmpty())
            return null;
        else if (result.size() > 1){
            //throw new IllegalStateException("Multiple albums with same title and artist");
            System.err.println("Multiple albums found with the same title and artist");
            return result.get(0);
        }
        else
            return result.get(0);
    } catch (DataAccessException dae) {
        if (dae instanceof DataAccessResourceFailureException)
            throw dae;
        dae.printStackTrace();
        return null;
    }
}
```

Figure 4.8. Methods to find an album

```java
public Album getAlbumById(String id){
    try {
        Optional<Album> result = album_RI_Mongo.findById(id);
        return result.orElse(null);
    } catch (DataAccessException dae) {
        if (dae instanceof DataAccessResourceFailureException)
            throw dae;
        dae.printStackTrace();
        return null;
    }
}

public Album getAlbumByTitleAndArtist(String title, String artist){
    try {
        List<Album> result = album_RI_Mongo.findByTitleAndArtist(title, artist);
        if (result.isEmpty())
            return null;
        else if (result.size() > 1){
            //throw new IllegalStateException("Multiple albums with same title and artist");
            System.err.println("Multiple albums found with the same title and artist");
            return result.get(0);
        }
        else
            return result.get(0);
    } catch (DataAccessException dae) {
        if (dae instanceof DataAccessResourceFailureException)
            throw dae;
        dae.printStackTrace();
        return null;
    }
}
```

Figure 4.9. Query to find an Album by title and artist

```
@Query("MATCH (u:User {username: $username}), (a:Album {albumName: $albumName, artistName: $artistName}) " +
        "MERGE (u)-[l:LIKES_A]->(a) " +
        "WITH u, l, a, CASE WHEN l.timestamp IS NULL THEN true ELSE false END AS isNew " +
        "LIMIT(1)" +
        "SET l.timestamp = datetime() " +
        "RETURN CASE WHEN isNew = true THEN 'CREATED' ELSE 'EXISTING' END")
String addLikes_A(String username, String albumName, String artistName);

@Query("MATCH (u:User {username: $username})-[r:LIKES_A]->(a:Album {albumName: $albumName, artistName: $artistName}) " +
        "DELETE r")
void removeLikes_A(String username, String albumName, String artistName);
```

Figure 4.10. Queries that performs an insertion and a delete of a `LIKES_A` relation

songs, *Liked albums* or *Followed users*, and only if one of these buttons is pressed (i.e. demanding the data), a list is showed.
In the following code, we show a typical query used in the User page to retrieve the songs they liked:

```
public Song_Neo4j[] findLikedSongsByUsername(String username) {
    String cypherQuery = "MATCH (u:User {username: $username})-[:LIKES_S]->(s:Song) " +
                         "RETURN s.songName AS songName, s.albumName AS albumName, " +
                         "s.artistName AS artistName, s.coverUrl AS coverUrl";

    return Song_Neo4j.getSongNeo4js(username, cypherQuery, neo4jClient).toArray(new Song_Neo4j[0]);
}
```

Figure 4.11. Queries that performs a search of all the songs liked by a given User

This on-demand querying approach significantly enhances the user experience by ensuring that the information displayed is always current and relevant to the user's interests. By dynamically generating queries to Neo4j when a user requests data, such as "Liked songs" or "Followed users," the system remains efficient and responsive. This method not only reduces the load on the database by avoiding unnecessary data retrieval but also provides a personalized and interactive experience, fostering user engagement and exploration of new content.

On-graph queries

The main function offered by our application is, without any doubt, the **Discover page**. In there Users can find suggestions on new songs the could like, based on different parameters, or new users to follow. Three different mode are available, and each of them has a specific query, so we'll analyze them once at time.

Suggestions based on user own tastes Every music lover want to find new songs that fit exactly their tastes, and we want to help our users in that way. Speaking in terms on domain, the query that accomplish this function can be defined as *given an user, finds the songs that they like, then find users that also like these songs, call them similarUsers, and find songs liked by the latters, ordering them on a score, where the score is defined as the number of similarUsers that likes that song*. This description can be translated in graph-domain terms: *given a username, find the User*

which has the property "username" equal to the given parameter, then find all the outgoing arches with the label `LIKES_S`, and the node Song that are connected with the very property to the initial user. Then find all the incoming arches with the property LIKES_S, and retrieve the node users that have the same property as outgoing arc connected to that song. Then find the songs that are connected with these user with the usual property, but not with the initial User, and order them by the count of incoming arches that starts from the similarUsers. The query that perform what we just described is the following:

```
private ArrayList<Song_Neo4j> findSuggestedSongs_ByTaste(String username) {
    String cypherQuery = "MATCH (targetUser:User {username: $username})-[:LIKES_S]->(likedSong:Song) " +
                         "MATCH (similarUser:User)-[:LIKES_S]->(likedSong) " +
                         "WHERE targetUser <> similarUser " +
                         "MATCH (similarUser)-[:LIKES_S]->(recommendedSong:Song) " +
                         "WHERE NOT (targetUser)-[:LIKES_S]->(recommendedSong) " +
                         "WITH recommendedSong, COUNT(*) AS recommendationStrength " +
                         "RETURN recommendedSong.songName AS songName, " +
                                "recommendedSong.albumName AS albumName, " +
                                "recommendedSong.artistName AS artistName, " +
                                "recommendedSong.coverUrl AS coverUrl " +
                         "ORDER BY recommendationStrength " +
                         "LIMIT 10";

    return Song_Neo4j.getSongNeo4js(username, cypherQuery, neo4jClient);
}
```

Figure 4.12. Cypher query called by `findSuggestedSongs_ByTaste()`

Suggestions based on popular songs among followed users A social network, by definition, should include some form of interaction between users: in our application, this interaction is represented by this feature that suggests songs that are popular among followed users. As a query in a graphDB, it can be described in this way: *starting from a User node with a given username, find all the User nodes reached by the property `FOLLOW` that starts from the initial User, and call them Followed for sake of simplicity. Then find all the songs that are reached by an arc with the `LIKES_S` property that starts from one or more nodes of Followed, but not from the initial User, and order them by a score*. The score is always defined as number of Friends that liked a specific song (i.e. the number of arches that go from Friends to that Song node). The query that perform what we just described is the following:

Suggestions of new Users based on the following To complete the social network part, we thought we would give users the opportunity to find new users to follow, basing these suggestions on the Users that they already follow. We describe the query that let the user to use this feature as: *starting from a User node, identified by the `username` property, find all the User node reached by an arc `FOLLOW` that starts from the initial User, and call them again Followed. Then find all the User nodes reached by one or more arches marked as `FOLLOW` by one or more of the Followed node, excluding the initial user, and return them ordering by score*. Once again, the score counts the number of relations that starts from Followed nodes and reach a specific User node. The query that perform what we just described is the following:

```
private ArrayList<Song_Neo4j> findSuggestedSongs_ByFollowed(String username) {
    String cypherQuery = "MATCH (targetUser:User)-[:FOLLOW]->(friend:User) " +
                         "WHERE targetUser.username=$username " +
                         "MATCH (friend)-[r:LIKES_S]->(recommendedSong:Song) " +
                         "WHERE NOT (targetUser)-[:LIKES_S]->(recommendedSong) " +
                         "WITH recommendedSong, COUNT(*) AS recommendationStrength " +
                         "RETURN recommendedSong.songName AS songName, " +
                                "recommendedSong.albumName AS albumName, " +
                                "recommendedSong.artistName AS artistName, " +
                                "recommendedSong.coverUrl AS coverUrl " +
                         "ORDER BY recommendationStrength DESC " +
                         "LIMIT 10";

    return Song_Neo4j.getSongNeo4js(username, cypherQuery, neo4jClient);
}
```

Figure 4.13. Cypher query called by `findSuggestedSongs_ByFollowed()`

```
public ArrayList<User_Neo4j> findSuggestedUserstoFollow(String username) {
    String cypherQuery = "MATCH (targetUser:User)-[:FOLLOW]->(friend:User) " +
                         "WHERE targetUser.username=$username " +
                         "MATCH (friend)-[r:FOLLOW]->(recommendedUser:User) " +
                         "WHERE NOT (targetUser)-[:FOLLOW]->(recommendedUser) " +
                         "WITH recommendedUser, COUNT(*) AS recommendationStrength " +
                         "RETURN recommendedUser.username AS username " +
                         "ORDER BY recommendationStrength DESC " +
                         "LIMIT 10";

    return User_Neo4j.getUserNeo4js(username, cypherQuery, neo4jClient);
}
```

Figure 4.14. Cypher query called by `findSuggestedUserstoFollow()`

4.5 ADMIN OPERATIONS

In our platform, the administrator primarily takes on the role of keeping the system up to date. The adminPage hosts three buttons, each of which triggers a corresponding back-end function. Each of these functions comprises multiple MongoDB aggregations that are utilized to either update data within the database or retrieve information for statistical purposes. Not all operations result in updates to the database; some are conducted purely to compile analytics and insights.
The three backend functionalities discussed are computationally demanding, which implies they have the potential to slow down the service for users if executed during peak hours. Consequently, these operations are designed to be invoked during periods of low server load, ideally during off-peak hours such as at night when user activity is typically reduced. Let's delve into these three macro functions

4.5.1 `CalculateAdminStats`

Among the three macro functions, this one is decidedly the simplest, with the sole objective of calculating three statistical values that can be useful for the administrator to monitor how much the application is being used and to make important decisions accordingly. Specifically, we aim to calculate the daily number of Likes on Albums, daily number of Likes on Songs, and the daily number of Reviews.

It's worth noting that this segment of code has already been analyzed in the "Neo4j Relevant Operations" section. The developers anticipate that this functionality will be invoked by the admin approximately once er day.

4.5.2 `UpdateNewLikesAndAvgRatings`

The most computationally intensive functionality that is performed daily, as dictated by the requirements of our application, is the updating of like counts and average ratings for albums. To keep the application as responsive as possible to users, these statistics are not updated in real-time when a user performs one of these actions. Although likes and reviews (with corresponding ratings) are saved in the databases immediately, the counters for these statistics are not updated with each individual action.

This update is conducted once a day during off-peak hours by a specific function. It is primarily divided into six steps:

1. `getNewLikesForAlbums()` This Neo4j aggregation first identifies albums that have received at least one like in the last 24 hours. For each of these albums, all the likes placed by users (including those from previous days) are recalculated. This operation may seem very heavy, but due to Neo4j's architecture, where nodes have counters for each type of relationship, the complexity is significantly reduced. Each node stores the count of incoming edges of a certain type, which simplifies the query process.
2. `setLikesToAlbums()` This function takes as input a list of pairs (album, like count) and

```
private ArrayList<AlbumOnlyLikes> findNewLikesForAlbums() {
    String cypherQuery = "MATCH (a:Album) <-[r:LIKES_A]- (:User) " +
            "WHERE date(r.timestamp) >= date() - duration({days: 1}) " +
            "MATCH (a) <-[l:LIKES_A]- (:User) " +
            "WITH a, COUNT(l) AS likes " +
            "RETURN a.albumName AS albumName, a.artistName AS artistsString, likes";

    return AlbumOnlyLikes.getAlbumOnlyLikes(cypherQuery, neo4jClient);
}
```

Figure 4.15. Query that returns the new total likes for Albums

```
@Transactional
public boolean setLikesToAlbums(AlbumOnlyLikes[] likeList) {
    try {
        BulkOperations bulkOps = mongoTemplate.bulkOps(BulkOperations.BulkMode.ORDERED, entityClass:Album.class);

        for (AlbumOnlyLikes like : likeList) {
            Query query = new Query(Criteria.where(key:"title").is(like.getAlbumName())
                    .andOperator(Criteria.where(key:"artists").is(like.getArtistsArray())));
            Update update = new Update().set(key:"likes", like.getLikes());
            bulkOps.updateOne(query, update);
        }

        bulkOps.execute();
        return true;

    } catch (DataAccessException dae) {
        if (dae instanceof DataAccessResourceFailureException)
            throw dae;
        dae.printStackTrace();
        return false;
    }
}
```

Figure 4.16. Query that sets updated value of likes for Albums

is responsible for updating the likes field of those albums. To improve efficiency, the individual update operations are executed as bulk operations. Bulk operations allow a group of database operations to be executed together in a single database call, rather than one by one, which is more efficient.

3. `getNewLikesForSongs()`: An aggregation function very similar to getNewLikesForAlbums() from step 1.

4. `setLikesToSongs()`: Very similar to setLikesToAlbums(), this function updates the likes count for songs.

5. `getAverageRatingForRecentReviews()` This function, operating on MongoDB, is similar to setLikesToAlbums but with the difference that it initially searches for albums that have received at least one review in the last 24 hours. It then performs an aggregation to calculate the average rating of all its reviews.

6. `setAverageRatingForRecentReviews()`: Similar to the previous functions, utilizing BulkOperations, this updates the averageRating field for only those albums that require an update, i.e., those that have been reviewed in the last 24 hours.

4.5.3 calculateRankings

The previously discussed macro-function is computationally intensive because it involves a series of read aggregations from the databases followed by a large number of write operations. This next macro-function is equally demanding, not because we update a large number of documents as before, but because we execute a sequence of seven aggregations. We acknowledge the heavyweight nature of this function but are equally aware that it is intended to be called by the system administrator only once a week, possibly during times when the servers are less busy.
The purpose of this function is to calculate the rankings of the most popular albums, songs, and artists of the past week (based on interactions like likes and reviews written in the last week) and the all-time rankings. During the design phase, it was decided that it doesn't make much sense to update these rankings more frequently.

How do we manage to provide users with quick access to the most popular tracks screen, which they visit multiple times a day? The solution is straightforward. Once the administrator calculates the rankings for the week, they are saved on the server in the form of a .json file. Therefore, whenever a user requests a page with rankings, the database is not asked to recalculate them; instead, the server reads the result from the file without sending any request to the database.

Let's now look at the various aggregations, and for the sake of brevity, we will detail only two of them.

getAlbumsWithMinReviewsByAvgRating_AllTime()

This aggregation aims to calculate the all-time ranking of albums based on their average rating. To ensure the credibility of this ranking, it includes only albums with at least ten reviews. This criterion is set to prevent an album with a single five-star review from unfairly topping the list.
The aggregation pipeline for this function consists of six stages:

```
public List<Album> getAlbumsWithMinReviewsByAvgRating_AllTime() {
    int minReviews = 5;
    Pageable pageable = Pageable.ofSize(pageSize:10);

    Aggregation aggregation = Aggregation.newAggregation(
            Aggregation.lookup(from:"reviews", localField:"_id", foreignField:"albumID", as:"albumReviews"), // Join con la collection reviews
            Aggregation.unwind(field:"albumReviews", preserveNullAndEmptyArrays:true), // Appiattisce l'array di reviews
            Aggregation.group( ... fields:"_id")
                    .first(reference:"title").as(alias:"title")
                    .first(reference:"artists").as(alias:"artists")
                    .first(reference:"coverURL").as(alias:"coverURL")
                    .first(reference:"songs").as(alias:"songs")
                    .first(reference:"likes").as(alias:"likes")
                    .first(reference:"averageRating").as(alias:"averageRating")
                    .first(reference:"year").as(alias:"year")
                    .first(reference:"lastReviews").as(alias:"lastReviews")
                    .count().as(alias:"reviewCount"), // Conta il numero di recensioni
            Aggregation.match(Criteria.where(key:"reviewCount").gte(minReviews)), // Filtra gli album con almeno 5 recensioni
            Aggregation.sort(Sort.by(Sort.Direction.DESC, ... properties:"averageRating")), // Ordina in base alla media delle recensioni
            Aggregation.limit(pageable.getPageSize()) // Applica la paginazione
    ).withOptions(AggregationOptions.builder().allowDiskUse(allowDiskUse:true).build()); // Permette l'utilizzo del disco

    AggregationResults<Album> results = mongoTemplate.aggregate(aggregation, collectionName:"albums", outputType:Album.class);
    return results.getMappedResults();
}
```

Figure 4.17. Aggregation to find albums sorted by average rating

1. `Group`: We start with the "reviews" collection and perform a grouping operation on the albumID field. This step classifies the reviews according to the album they pertain to. Additionally, it counts the number of reviews for each album.

2. `Match`: We then filter these review groups, requiring a minimum of ten reviews. Any album groups with fewer are discarded.

3. `Lookup`: At this stage, we execute a join operation with the ”albums” collection. This is necessary because we want to display not just the ratings but also the album cover to the user.

4. `Project`: This aggregation phase specifies which fields we are interested in receiving in the documents that will be returned by the database.

5. `Sort`: The albums are then ordered in descending order based on the average rating.

6. `Limit`: Since we aim to show only the top ten albums in this ranking to the user, we limit the results to just ten from MongoDB.

Considerations on this Aggregation We are aware that the lookup operation is heavy and generally best avoided. However, in our case, it becomes necessary because we want to show the user the album cover, information that resides only within the ”albums” collection, hence necessitating the join.
We have also tried to minimize the join’s impact as much as possible: we perform the join as late as possible in the pipeline, specifically only for albums that are candidates for this ranking, meaning only those with more than ten reviews. Moreover, it’s crucial to emphasize that the join with the ”albums” collection is made on the indexed _id field. Consequently, our tests have shown that the join on this field is executed very swiftly.
A final important note is that we have currently set the minimum number of reviews at ten because it was a reasonable figure at present. However, in the future, as the top albums accrue more reviews, this threshold will need to increase to reduce the number of documents involved in the join.

`getArtistsWithMinAlbumsByAvgRating_AllTime()`

This MongoDB aggregation pipeline meticulously crafts a ranking of leading artists based on their albums’ average ratings. It particularly focuses on artists with a minimum of three albums, aiming to reflect a consistent standard of quality across their body of work. Below is an enhanced breakdown of the pipeline’s operational stages:

1. `Unwind`: The $unwind operator deconstructs the ’artists’ array from each document, laying the groundwork for individual artist evaluation based on the albums they’ve contributed to.

2. `Group`: At the $group stage, documents are consolidated by artist identifier, where the average album rating (avgRating) and a count of albums (albumCount) for each artist are computed.

3. `Match`: The pipeline then employs a $match filter to include only those artists with a substantial body of work, specifically those with three or more albums.

```java
public List<ArtistWithAvgRating> getArtistsWithMinAlbumsByAvgRating_AllTime() {
    MongoClient myMongoClient = MongoClients.create(new ConnectionString("mongodb://10.1.1.18:27017,10.1.1.17:27017,"+
        "10.1.1.19:27017/?replicaSet=BB&w=1&readPreference=nearest&retryWrites=true"));
    MongoDatabase database = myMongoClient.getDatabase("BeatBuddy");
    MongoCollection<Document> collection = database.getCollection("albums");

    Bson unwindOp1  = unwind("$artists");
    Bson groupOp    = group("$artists",
            avg("avgRating", "$averageRating"),
            sum("albumCount", 1));
    Bson matchOp    = match(gte("albumCount", 3));
    Bson lookupOp   = lookup("artists", "_id", "name", "artistDetails");
    Bson unwindOp2  = unwind("$artistDetails");      // se un documento non ha il campo "artistDetails" o se il campo
                        //"artistDetails" è un array vuoto, il documento verrà escluso dalla pipeline di aggregazione.
    Bson sortOp     = sort(descending("avgRating"));
    Bson limitOp    = limit(10);
    Bson projOp     = project(fields(
            computed("_id", "$artistDetails._id"),
            computed("name", "$artistDetails.name"),
            computed("profilePicUrl", "$artistDetails.profilePicUrl"),
            include("avgRating")
    ));

    AggregateIterable<Document> result = collection.aggregate(Arrays.asList(
            unwindOp1, groupOp, matchOp, lookupOp,
            unwindOp2, sortOp, limitOp, projOp
    ));

    List<ArtistWithAvgRating> artistsWithLikes = new ArrayList<>();
    result.forEach(doc -> artistsWithLikes.add(ArtistWithAvgRating.mapToArtistWithLikes(doc)));

    myMongoClient.close();
    return artistsWithLikes;
}
```

Figure 4.18. Aggregation to find albums sorted by average rating

4. `Sort`: A $sort operation arranges the artists in descending order by average rating, ensuring the cream of the crop rises to the top.
5. `Lookup`: This stage links the interim results with the 'artists' collection, matching on the _id field to enrich the artist records with additional details, including names and profile images.
6. `Match` (second occurrence): A subsequent $match weeds out any incomplete artist records, denoted by an empty 'artistDetails' list, following the join.
7. `Limit`: The results are then narrowed down to the top 10 artists.
8. `Project`: The final $project phase defines the specific fields to be included in the output.

Considerations on this Aggregation In contrast to the previous case, this time the join operation is not performed on an indexed field, but at least the number of documents on which the join must be executed is significantly lower. It would have been reasonable to apply the limit of 10 before performing the join to minimize the number of documents involved. However, in our scenario, this is not feasible since our database, for previously discussed reasons, does not have an artist document for every artist featured in the albums. Therefore, we are compelled to apply the limit of 10 only after performing the join and verifying that it was successful.

During the drafting of this aggregation, we considered using a limit of 30 before the join to further reduce the number of necessary joins. However, our tests revealed that adding this additional step did not offer significant benefits. This is because MongoDB, when executing the pipeline, notes that there is a sort operation followed by a limit of 10. Consequently, it optimizes the join to only consider the top 10 documents from the sorted list, rendering the limit of 30 redundant.

Here, as with the previous aggregation, the minimum threshold required for an artist to appear in this ranking (which is currently at 3 albums) will need to be updated in the future to ensure that the aggregation remains highly efficient. As the data grows and the top artists accumulate more albums, this threshold will be adjusted to keep the join operation streamlined and the overall process performant.

4.5.4 DB update

As we said before, we decided to implement a Python script to add new Albums, that is automatically launched every Saturday at 1.00AM. The launch of that Python script is done by Java, using this function:

```java
@Service
class MyDatabaseUpdaterService {
    //Schedule every Saturday at 1:00 AM
    @Scheduled(cron = "0 0 1 * * SAT")
    public void updateDatabase() {
        try {
            String pathToPythonScript = Constants.folderName_DatabaseUpdateScript + File.separator + "update_database.py";
            ProcessBuilder processBuilder = new ProcessBuilder("python3", pathToPythonScript);
            processBuilder.inheritIO();
            processBuilder.start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 4.19. Java function to call the Python script that update the database

Using the *ProcessBuilder.inheritIO()* we are able to see the possible logs produced by the Python code directly into the shell that launches our Web Server.

5 DATABASES CHOICES

5.1 INTRODUCTION TO VM ORGANIZATION

Before delving into the details of this chapter, it is beneficial to provide an infographic illustrating the overall organization of the virtual machines (VMs) at our disposal, along with descriptions of the processes running on each of them.

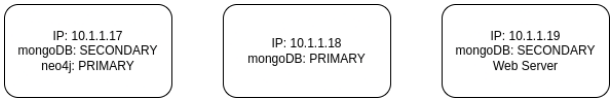


Figure 5.1. An Overview of Our VMs and Their Roles in the Application

5.2 INDEXES

5.2.1 MongoDB - Users: `textIndex(username)`

The primary queries that interact with the User collection involve searching for a user with a specific username, such as those related to the search feature or adding a new review to the User document. Considering the anticipated query volumes outlined in the Volume Table, these queries are not insignificant. Therefore, before introducing the actual index, we conducted an analysis to determine if its implementation could enhance query performance.

To assess the potential improvement, we employed the *MongoCompass* tool and the results demonstrated a noticeable enhancement in query performance with the index in place. Consequently, we decided to proceed with the implementation of this index.

The choice of using a text index was driven by the need to support partial matches on usernames. Although this could also be achieved using regular expressions (regex), it was observed that regex-based queries were slower compared to utilizing the index, as evident from the execution time of a query involving regex.

Figure 5.2. Steps and Statistics of the Execution of a Search-type Query Without Indexes on Users

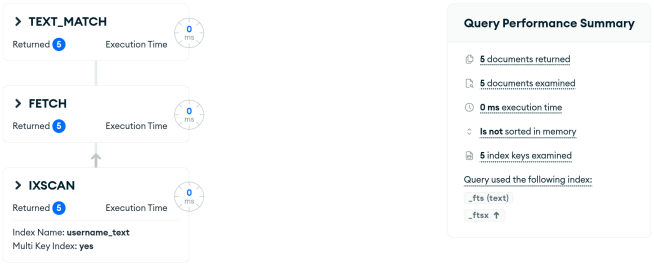


Figure 5.3. Steps and Statistics of the Execution of a Search-type Query with the Index on Users

5.2.2 MongoDB - Albums: `textIndex(songs.name)`

A limitation of MongoDB is that it allows only one text index to be defined in a collection, making it impossible to create separate text indexes for both the ‘title‘ and ‘songs.name‘ fields.

Figure 5.4. Steps and Statistics of the Execution of a Search-type Query Without Any Index on Albums, Searching for `title`

We made the decision to add a text index on the ‘songs.name‘ field based on two key observations:

- Examining the Volume Table, it became evident that searches on songs occur more frequently than searches on albums.
- Given the higher number of songs compared to the number of albums, queries searching for songs were significantly slower (as demonstrated in Figure 5.5).

Utilizing MongoCompass, we observed a substantial improvement in query execution time after

Figure 5.5. Steps and Statistics of the Execution of a Search-type Query Without Any Index on Albums, Searching for `songs.name`

implementing the index (as shown in Figure 5.6):

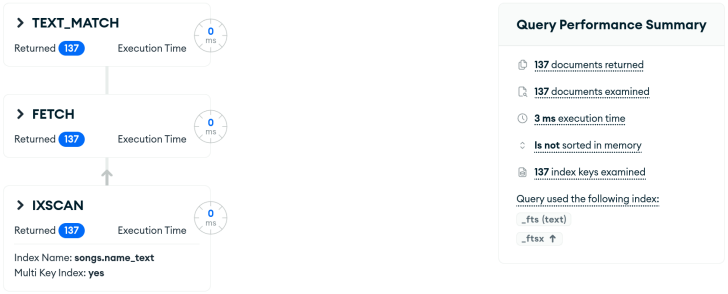


Figure 5.6. Steps and Statistics of the Execution of a Search-type Query with the Index on Albums, Searching for `songs.name`

The notion of a single index covering both ‘title‘ and ‘songs.name‘ fields was dismissed since our intention was to enable users to search by specifying the content type, which would be impossible with a single index.

In conclusion, we can summarize two important points:

- New albums and songs are typically added only once a week, so the reorganization of the index is not time-consuming, especially when compared to the time saved through index usage.
- The ‘songs.name‘ field is static and remains unchanged over time, so the index does not need to be recalculated.

5.2.3 MongoDB - Albums: `index(title, artists)`

Two of the most resource-intensive queries in our application involve updating the number of likes for both Albums and Songs. When these queries are executed without the appropriate index, the execution time, particularly for the albums, becomes unacceptably long (as shown in Figure 5.7).

We were unable to provide a figure illustrating the time taken to update song likes because, after an hour of execution, they had not yet completed. Based on some rough calculations, we estimated that it would take nearly two hours to finish the entire task, which is clearly

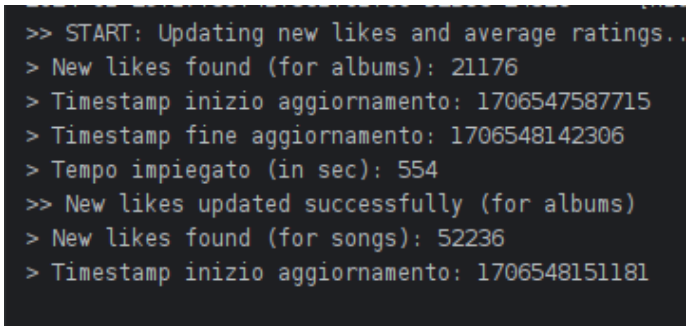


Figure 5.7. Java Console Output for Queries Updating Likes on MongoDB. Note that the second query, for likes on songs, was still running.

unacceptable. However, with the introduction of the compound index, the time required for these updates decreased dramatically (as depicted in Figure 5.8):

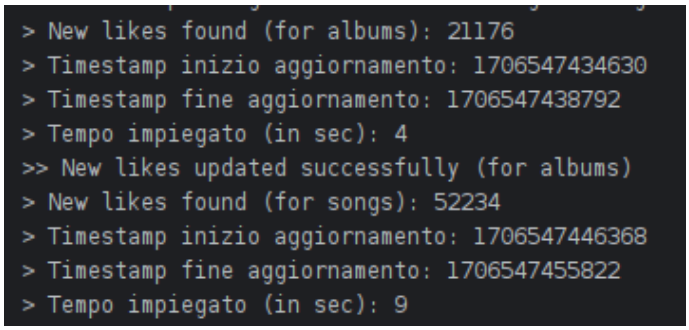


Figure 5.8. Java Console Output for Queries Updating Likes on MongoDB After the Definition of the Index.

While these operations are intended to be performed once a day during periods of lower server load, the excessive time required to complete them without the index is unreasonable. This underscores the necessity of having the index in place. Furthermore, the addition of this index is justified when considering that other requests from active users would also be slowed down by these time-consuming queries, which goes against our High Availability requirements.

5.2.4 Consideration on Reviews

In previous sections, we discussed the reasons for having a standalone collection for Reviews. In this section, we will explain why we decided not to insert any indexes on this collection.

The only index that might have made sense to insert is the one on `albumID`, but we can justify its absence.

We estimated an average of 30,000 new reviews added each day. This implies that there would be 30,000 index reorganization operations on the Reviews collection daily. Additionally, as mentioned earlier, reviews are primarily retrieved from the album page, and this happens very infrequently. Therefore, we would have an index that is reorganized numerous times but used

very rarely. We tested the execution time of a query that retrieves reviews related to a given album, and here are the results:

Figure 5.9. Steps and statistics of the execution of a query that retrieves reviews related to a given album.

We believe that this execution time is still acceptable given the assumption of a low number of requests to read reviews in comparison to the number of added reviews. However, we remain open to reconsidering our decision if the number of requests for reading reviews increases significantly during the lifecycle of our application.

Consideration on Artists

In our application, the number of artists is fixed and cannot increase over time. Given this and the fact that a search by name, using regex, provides a response within an acceptable amount of time, we believe that defining an index for this collection is not required.

Figure 5.10. Steps and statistics of the execution of a search-type query on Artists, searching for `name`.

5.2.5 neo4j

Regarding Neo4j, due to the large number of nodes in our database, we have decided to add three different indexes, one for each entity. For every test conducted to assess query execution time, we used `CALL db.clearQueryCaches()` to clear the Neo4j cache each time we needed it, ensuring that our tests are fairly independent.

User

Since the primary queries involving Users start from a specific user, we studied the execution time for a typical simple query that searches for outgoing relationships of a specific User node. We conducted these tests both before and after the introduction of the index on `username`, as it is the only property we can find in that entity. Before the introduction of the index, we observed:

Figure 5.11. Example of a query execution, including execution time, before the introduction of the index on `username`.

After adding the index, the execution time decreased significantly, as shown in the following image:

Figure 5.12. Example of a query execution, including execution time, after the introduction of the index on `username`.

Given the fact that queries in Neo4j starting from a User are the most common operations in our database (e.g., operations returning User likes and follows occur up to 300,000 times per day, with an additional 100,000 from new likes and 90,000 for accesses to the Discover page), we decided

to introduce this index to significantly improve the performance of these common operations.

Song

Songs are also involved in a significant number of queries, making it useful for our system to exploit indexes for this entity as well. When we examined a simple `MATCH` query to search for a specific song, we noticed that the execution time was quite high:

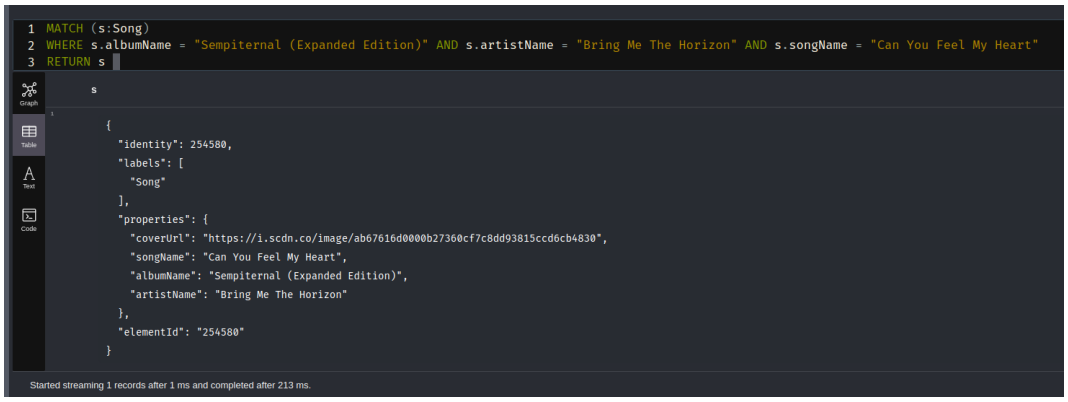


Figure 5.13. Example of a query execution, including execution time, without any index.

We were undecided about which fields to include in the index, so we tested both possibilities available to us. The first test involved an index defined on `(Song.albumName, Song.artistName, Song.songName)`:

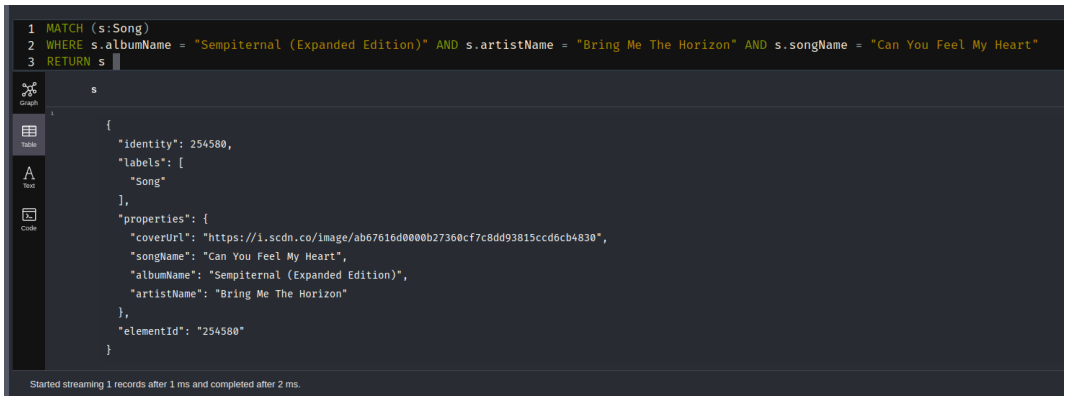


Figure 5.14. Example of a query execution, including execution time, after the introduction of the index on `(Song.albumName, Song.artistName, Song.songName)`.

While the first index provided good results, we also tried an index on `(Song.albumName)`:

The small difference in response time led us to choose the second option. This is because, even though it is slightly slower, it undoubtedly consumes less memory than the first one, making it a good trade-off between execution time and memory usage.

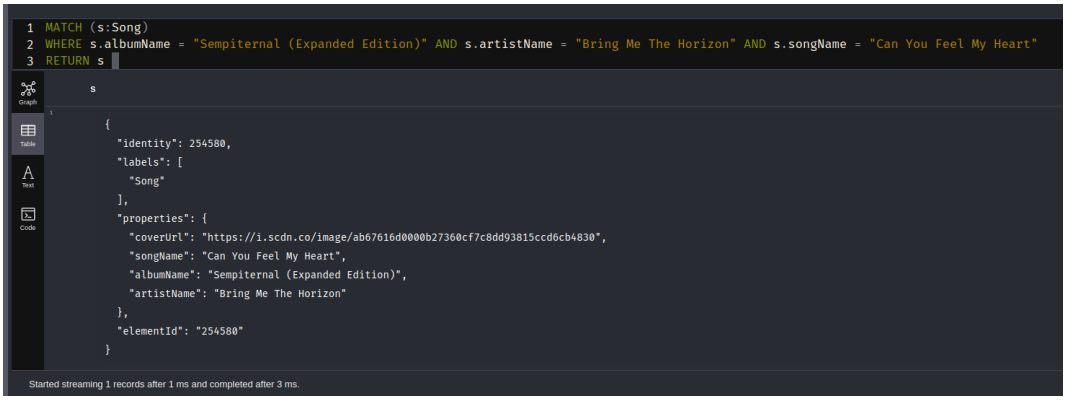


Figure 5.15. Example of a query execution, including execution time, after the introduction of the index on `(Song.albumName)`.

Album

The same reasoning we applied to Songs also applied to Albums. We started by observing the execution time of a query without any index:

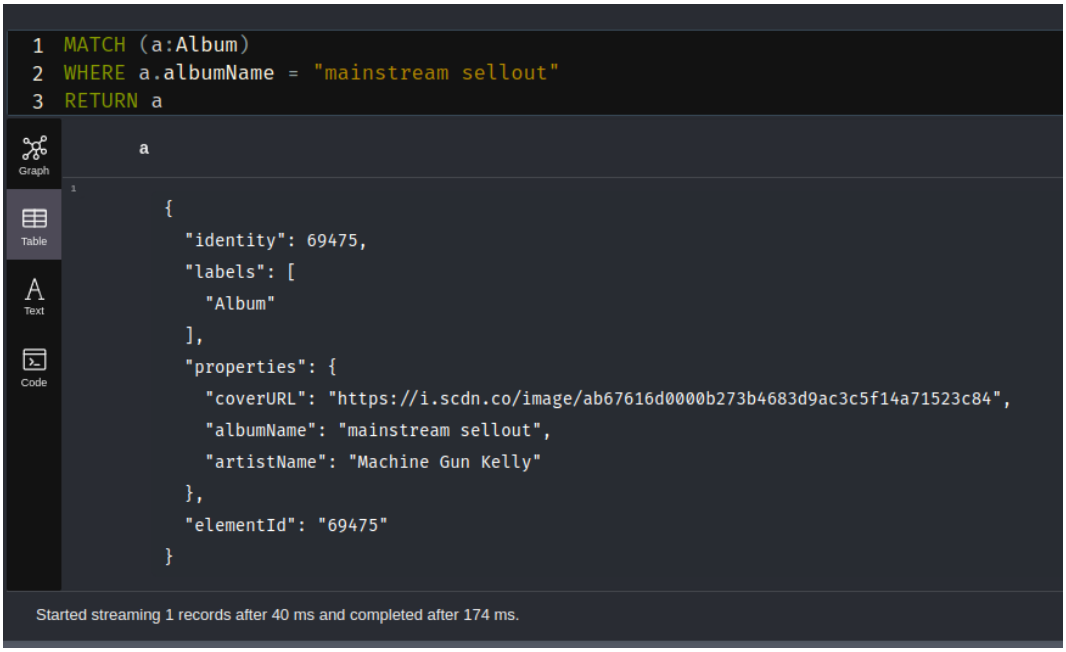


Figure 5.16. Example of a query execution, including execution time, without any index.

We had three possibilities: indexing on `albumName`, indexing on `artistName`, or indexing on both properties. We tested the same query using each of these three indexes, and the results are shown below.

The index that showed the best performance was the one on `a.albumName`, so we decided to use it in our database.

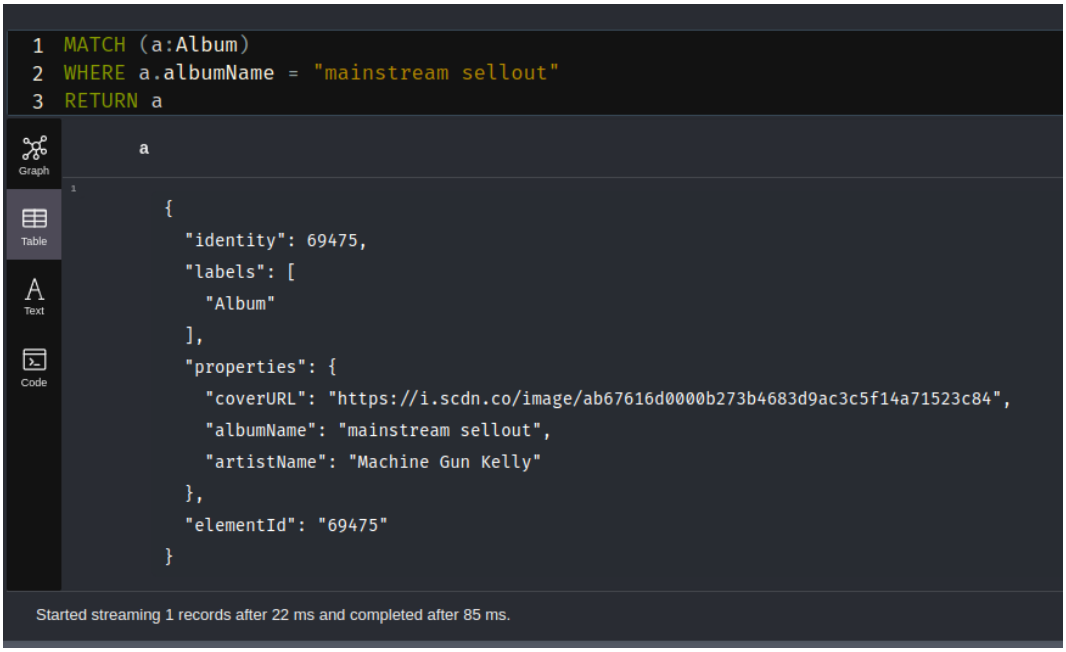


Figure 5.17. Example of a query execution, including execution time, after setting up an index on `a.artistName`.

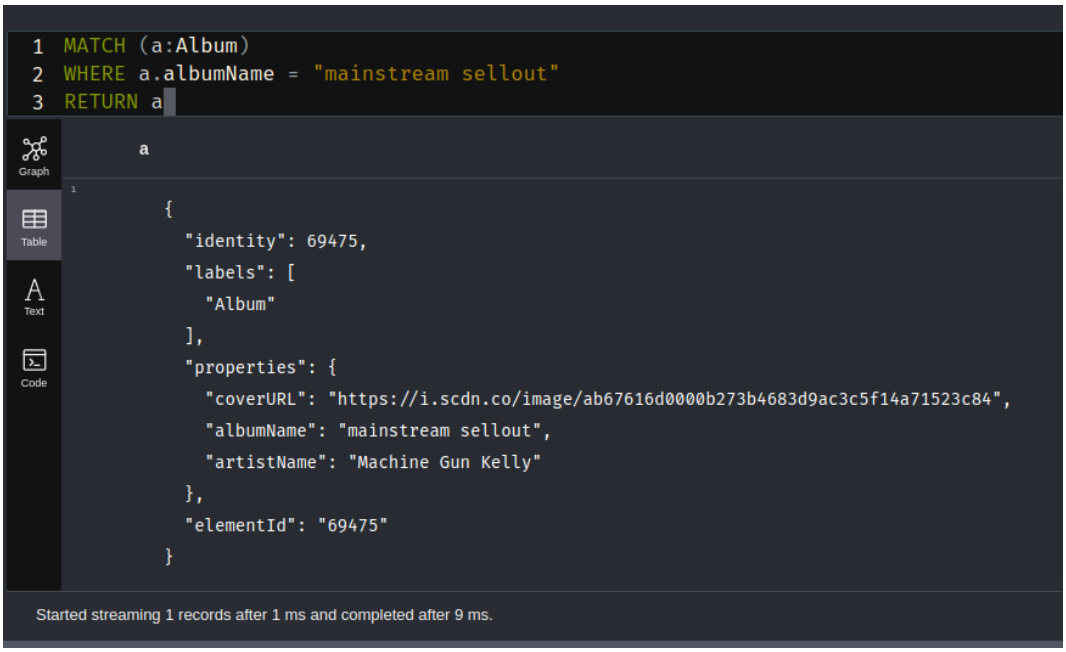


Figure 5.18. Example of a query execution, including execution time, after setting up an index on `a.albumName`.

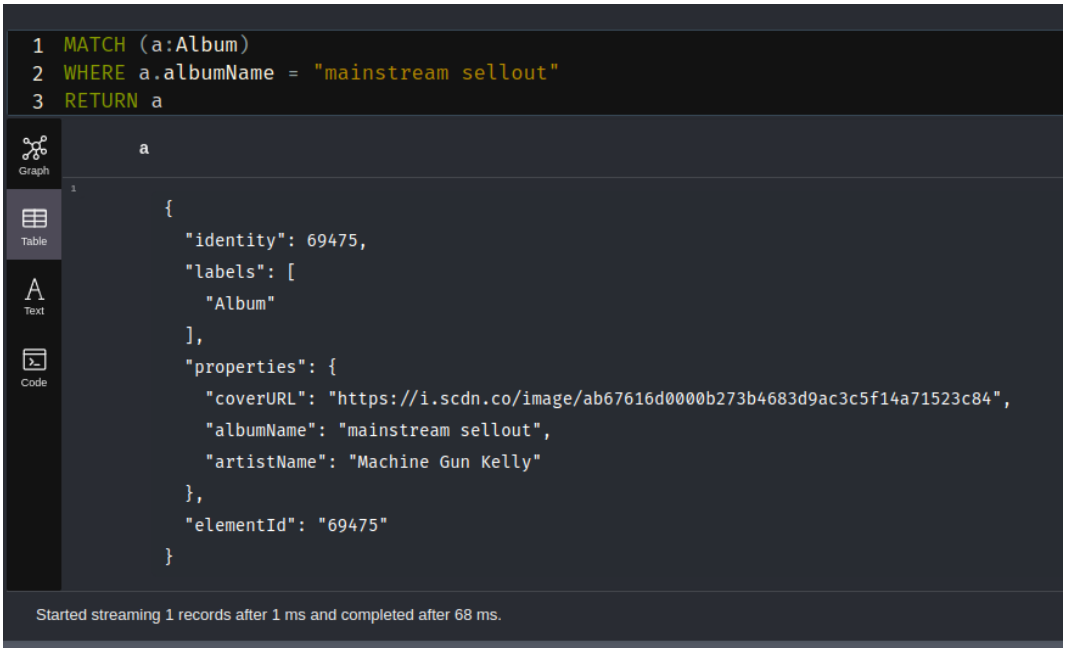


Figure 5.19. Example of a query execution, including execution time, after setting up an index on `(a.albumName, a.artistName)`.

5.3 SHARDING ON MONGODB

Sharding is a method of distributing data across multiple servers, a process essential for large-scale databases to ensure efficient data management and query response. MongoDB, being a NoSQL database, supports horizontal scaling through sharding, which is particularly beneficial for handling massive amounts of data and high throughput. This approach not only enhances performance but also ensures data availability and fault tolerance.

5.3.1 Sharding Strategy for Collections

In our project, we have identified the following collections for sharding: Users, Reviews, Albums, and Artists. The choice of sharding key and strategy is based on data access patterns, growth rate, and query performance.

Albums Collection

Sharding Key: title (First Letter)

The Albums collection is sharded based on the first letter of the album title. This approach is founded on analyzing our dataset to ensure a balanced load across three servers:

```
Sharding Distribution:
Server server1: Titles starting with letters 'A' to 'F' - 17,146 albums
```

```
Server server2: Titles starting with letters 'G' to 'O' - 16,405 albums
Server server3: Titles starting with letters 'P' to 'Z' - 19,507 albums
```

The rationale for this sharding strategy includes:

- **Equal Distribution of Data**: This method enables an approximately equal distribution of albums across servers, preventing any single server from becoming a performance bottleneck due to an excessive load.
- **Efficiency in Query Processing**: Sharding by the first letter of the album title is beneficial for query optimization. Most album-related queries involve the title, so directing these queries to the specific server holding the relevant data subset significantly reduces processing time.
- **Scalability and Maintenance**: Alphabet-based sharding enhances the scalability of our database. It allows for straightforward expansion as more albums are added. Additionally, this clear-cut division facilitates database administration tasks like data backup and recovery, ensuring smoother maintenance and management.

In summary, adopting an alphabet-based sharding approach for the Albums collection is a strategic decision to achieve load balancing, improve query processing efficiency, and ensure easy scalability and maintenance. This methodology supports the overall effectiveness and robustness of our application's database infrastructure.

Users Collection

Sharding Key: username

For the Users collection, the sharding strategy is based on usernames. Given the uniform distribution of usernames, this approach prevents any single shard from becoming overloaded. Additionally, considering geographical location for future sharding could further optimize response times for user profile access.

```
Sharding Example:
Server 1:  Abel  - J. Spradling with 18926 users
Server 2: J. Spurway - Robert E. Fiveash with 18926 users
Server 3: Robert E. Hadik - ~TRUTH~ with 18926 users
```

Reviews Collection

Sharding Key: Hashed albumID

The Reviews collection employs a hash-based sharding strategy on AlbumID. This approach efficiently distributes data across shards, especially considering the potential exponential growth in the number of reviews and their skewed distribution among albums.

Artists Collection

Sharding Key: name

As the Artists collection is relatively static without the provision for adding new artists, sharding is based on the artist's name. This key was chosen as it is the most frequently accessed field in queries related to this collection.

5.4 REPLICATION

5.4.1 MongoDB Replication

To meet the non-functional requirement of **high availability**, we have set up a replica set using the virtual machines available to us.

The organization of the replica set can be summarized as follows:

- A primary node with the IP address 10.1.1.18, responsible for receiving all requests.
- Two secondary nodes with IP addresses 10.1.1.17 and 10.1.1.19, storing copies of the primary server.

We configured the cluster with the following parameters:

```
rsconf = {
  _id: "BB",
  members: [
    { _id: 0, host: "10.1.1.18:27017", priority: 5 },
    { _id: 1, host: "10.1.1.17:27017", priority: 3 },
    { _id: 2, host: "10.1.1.19:27017", priority: 1 }
  ]
};
```

We chose this configuration because:

- We wanted the primary server to have only the MongoDB server as the main process.
- In case the primary server goes down, we still need to deliver the service.
- The server process consumes significantly more system resources than the Neo4j instance.
- Since both secondary nodes have another main process to run, we added the primary workload to the one using fewer system resources.

Furthermore, we introduced constraints such as `writeConcern` set as `w` and `readPreference` in the URI connection string:

```
mongodb://10.1.1.18:27017,10.1.1.17:27017,10.1.1.19:27017/
replicaSet=BB&w=1&readPreference=nearest&retryWrites=true
```

These parameters stand for:

- `w=1`: The `w` parameter in MongoDB specifies the level of write concern. Setting `w=1` means that a write operation is considered successful as soon as the primary node in the replica set has written the data, without waiting for acknowledgments from secondary nodes. This level of write concern provides a balance between speed and data safety.
- `readPreference=nearest`: The `readPreference` setting in MongoDB dictates from which node (primary or secondary) the data is read. Choosing `readPreference=nearest` directs MongoDB to perform read operations on the node closest in network latency, regardless of whether it's the primary or a secondary node. This setting can reduce read latency, as reads are distributed among all nodes in the replica set.

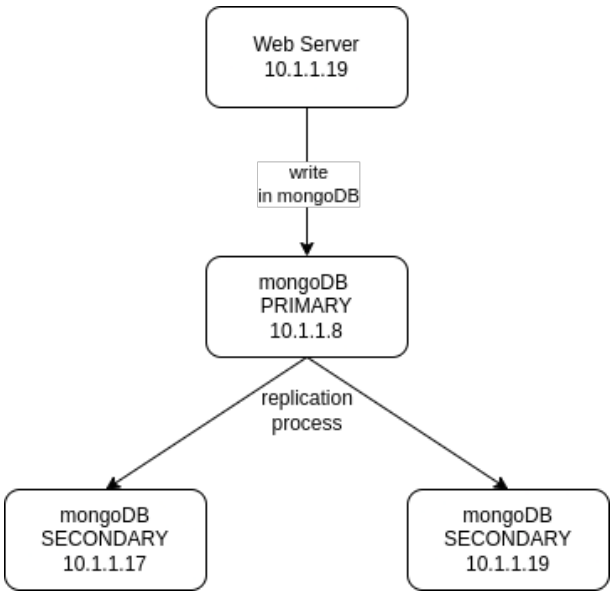


Figure 5.20. A diagram of our replication set

5.4.2 Replication on neo4j

Since the neo4j Community Version don't allow the creation of a replica, we simply decided to place that DB in a different VM from the ones that run the Web Server and the Primary mongoDB Server. Doing this, we avoid to concentrate all the main processes in the same machine, to manage as best as we can a Single-Point-of-Failure situation.

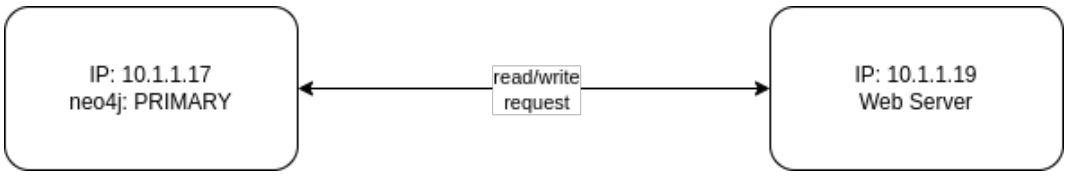


Figure 5.21. The primary, and unique, server have to manage both write and write requests

5.4.3 Consideration on CAP theorem

Looking at the requirements, it turns out that a strict consistency between replicas is not necessary, and even between different databases (for what concerns the same concepts, like the already discussed likes on albums and songs). Because of the first one, we decide to position ourself in the **AP** edge of the CAP triangle. The latter permit us to set the parameters `w=1` and `readPreference=nearest`, with the conclusions we shared previously.

Emphasis on AP in the System's Architecture

In the architectural design of our application, we have deliberately chosen to prioritize Availability and Partition Tolerance (AP) as per the CAP theorem. This decision is critical for an application like ours, which heavily relies on social networking features. By focusing on AP, we ensure that our system remains operational and accessible even in the event of network failures or data partitioning issues. This approach is vital to maintain uninterrupted user engagement with the social aspects of the application. We do this by replicating data across multiple nodes, thus ensuring availability even if some nodes are unreachable. While this might lead to temporary inconsistencies, it's a trade-off we accept to guarantee constant availability, which is crucial for user interactions within our social platform.

Scalability and High Availability

In our system architecture, scalability and high availability are central concerns, particularly considering the design and deployment of our MongoDB replica set. To ensure high availability, we have configured a replica set using three virtual machines, each with specific roles and responsibilities:

- **Primary Node (IP: 10.1.1.18)**: This machine acts as the primary server for MongoDB, handling all incoming requests. Its dedicated role ensures maximum efficiency and performance for write operations. The higher priority assigned to this node ensures that it remains the primary server as long as it's operational.
- **Secondary Nodes (IPs: 10.1.1.17 and 10.1.1.19)**: These machines serve as secondary nodes in the replica set, holding copies of the primary server's data. They provide redundancy and ensure data persistence in case the primary node fails. The secondary node at 10.1.1.19 also hosts the site, while 10.1.1.17 serves as a slave for both MongoDB and Neo4j.

The prioritization within the replica set is strategically designed:

- **Resource Allocation**: Given that the primary node's main task is to run the MongoDB server, it has been allocated the highest priority (5) to ensure it remains the primary node under normal circumstances. This setup minimizes the risk of performance bottlenecks, as the primary node does not have to share resources with other heavy processes.
- **Failover Strategy**: The configuration of the secondary nodes is aligned with our failover strategy. In the event of the primary node's failure, the secondary node at 10.1.1.17, with a priority of 3, is more likely to take over as the new primary node. This ensures continuous service delivery, even during primary node downtimes.
- **Load Balancing and Resource Management**: Considering that both secondary nodes run additional processes, their priorities are set to lower values (3 and 1, respectively). This approach balances the load and resource utilization across the nodes. The secondary node at 10.1.1.19, having the additional responsibility of hosting the site, is assigned the lowest priority (1) for MongoDB operations. This decision is a crucial aspect of resource management, ensuring that the hosting responsibilities do not impact the database's performance.

Overall, this carefully planned replica set configuration underlines our commitment to providing a robust, scalable, and highly available system. By meticulously balancing the load and prioritizing the nodes, we achieve a resilient architecture that can adapt to varying loads and recover swiftly from potential failures.

Balancing Consistency and Availability Trade-Offs

Our architectural decision to lean towards Availability over Consistency involves significant trade-offs. Given the social networking nature of our application, we prioritize ensuring that users can access and interact with the platform without interruption, even if it means dealing with eventual consistency. In scenarios of network partitioning or server failures, our system continues to operate at the cost of some data being temporarily out-of-date. This trade-off is crucial for maintaining user engagement and satisfaction, as immediate access to social features is more critical than real-time data accuracy. However, we closely monitor this balance and remain agile in adjusting our consistency models to adapt to changing user needs and system requirements. This approach allows us to maintain a dynamic and user-centric platform that can evolve and improve over time.

6 FUTURE IMPLEMENTATION

6.1 ALBUM UPDATING

The Python script we wrote to update our database, searching for artists which release one or more albums after the building of our dataset, theoretically works. The main problem is the Spotify API rate limit: for those applications that are in beta mode, the rate limit is around , simplifying, 700 requests per day. That rate is not sufficient for us to update in short time our DB. The solution is the following: request to Spotify the change of our mode to public one, and this could be easily accomplished if we wanted to make our site publicly available on Internet.

6.2 SONG LYRICS

A brand new feature of our application could be the introduction of song lyrics in our dataset, with the possibility for users to search a song using a line of its lyric.

6.3 FOLLOW AN ARTIST

We could implement a feature that allow users to follow the artists, and get suggestions of new ones based on which they follow.

6.4 MUSIC GENRES

The introduction of music genres for artists, songs and albums could lead to a better suggestion algorithm, based on specific genres chosen by the users or the ones they love the most.

6.5 ADD NEW ARTITST

For sake of simplicity, we decided to add in our DBs the 10'000 most listened artists on Spotify across the word. This obviously excluded those artists that have a good amount of listener, but not higher enough to compete with other international artists (e.g. *Francesco Guccini*, *Bello Figo*, *Mina* and many more). In our Database also missing those artists that hadn't an high number on listener on Spotify when the initial dataset was created, but they glew up in the very last years or even months, such as *Baby Gang* (that at this moment has more monthly listener than *Sfera Ebbasta*, who is in the dataset)

6.6 PLOTS FOR ADMIN

It can be useful for the administrator to have graphs available that quickly show the trend of daily likes, accesses and new users for a specific period of time.