CPSC 304 Project Cover Page

Milestone #4

Date: 08-06-2023

Group Number: 32

Name	Student Number	CS Alias (Userid)	Preferred E-mail Address
Jiawei Liu	55362669	b7j6x	1943743535@qq.com
Flora Deng	14085211	d1i2t	floraa817@gmail.com
Tammie Liang	52445806	c1g1c	tammieliang@hotmail.com

By typing our names and student numbers in the above table, we certify that the work in the attached assignment was performed solely by those whose names and student IDs are included above. (In the case of Project Milestone 0, the main purpose of this page is for you to let us know your e-mail address, and then let us assign you to a TA for your project supervisor.)

In addition, we indicate that we are fully aware of the rules and consequences of plagiarism, as set forth by the Department of Computer Science and the University of British Columbia

Project Description

Music is gradually becoming an integral part of people's lives. People with headphones can be seen everywhere. Various styles of music are played in shopping malls or stores, and every day there are different concerts or lives happening around the world. At the same time, artists and the record labels behind them are becoming more and more known and a topic of conversation. Some of the loved artists or songs are known all over the world.

Data plays a huge role in the dissemination of information about records, artists and the labels associated with them. Most music apps can see the current most searched artists, songs or albums, but their main functions are still focused on the listening function rather than songs or artists' information browsing. The music information contained is not complete, even some niche music information is completely blank.

Through this project, our team hopes to create a more concise and clear platform for searching music information through the establishment of a database.

Our Melodex is a comprehensive web platform tailored for music lovers. Built using cutting-edge technologies such as React (frontend design), SpringBoot (backend design) and MySQL (database). It stores music artists and discographies, as well as being able to find them easily. It is useful for, say, a music wikipedia staff that needs to store and be able to find information about all the music artists and discographies in the world. The platform aims to make it easy for users to access, manage, and find music-related information.

Our database includes navigation bars that can navigate users to artists, music, records, labels, and the home page. The platform's administrative features allow the user to add information into appropriate pages, or make editing changes, as well as delete invalid information.

In addition to this, our platform provides users with a number of additional functions for interacting with the data:

Selection operation —— find a discography that meets a certain condition.

Join operation —— find albums with the same name as the discography.

Projection operation —— find the name and release date of all albums.

Nested Aggregation —— find the artist with the lowest average number of songs per album among the average number of songs per album.

Division operation —— find listeners who have attended all live events.

The above queries take into account most of the lookup requirements and allow the user to observe the overall data comparison.

Final Schema

Our database has been focused on the acquisition of information content about songs, records, artists and music companies from the beginning of the project, and the platform's primary focus has stayed the same. However, during the implementation of the project, we removed some of the entities and attributes.

First of all, at the beginning of the design, we designed different account interfaces for artists and listeners. However, considering that the restriction that users need an account to use the platform would affect user experience and would not be conducive to quick and easy instant searches, we decided to remove this feature. Therefore, we removed 'email' and 'userPassword' from 'Artist_ContractedWith.'

In addition, our platform focuses on managing, collecting, and querying music information. The list of songs that users listen to changes many times a day, and the 'ListenTo" entity requires users to have a personal account and does not contribute to the collection of music information, as well as placing an unnecessary burden on the platform regarding information storage. Therefore, we removed the 'ListensTo' schema.

We also removed 'numOfLikes' and 'ranking' attributes from 'Song' and 'Album' entities because most users need an official source of the number of favorites when adding information, which they may not have, and the number changes too quickly to rely solely on user updates. The lack of favorites will lack a unified reference for ranking.

(NOTE: By “mainly implemented”, we mean that is where you will find the actual SQL query, but the backend logistics always begin from tracing the Controller function that uses it)

INSERTION QUERY

MAINLY implemented in Artist_ContractedWithRepository, insertNewArtist function (backend)

```
INSERT INTO Artist_ContractedWith (artistName, age, country, biography,
numOfMembers, labelID)
VALUES (:artistName, :age, :country, :biography, :numOfMembers, :labelID);
```

DELETE QUERY

MAINLY implemented in Artist_ContractedWithRepository, deleteArtist function (backend)

```
DELETE FROM Artist_ContractedWith
WHERE artistID = :artistID
```

UPDATE QUERY

MAINLY implemented in Artist_ContractedWithRepository, updateArtist function(backend)

```
UPDATE Artist
SET artistName = :artistName, age=:age, biography=:biography,
numOfMembers=:numOfMembers-- user should specify which attribute to update
WHERE artistID = :artistID;
```

SELECTION QUERY

MAINLY implemented in GeneralController, getDynamicSelection function (backend)

The user is able to select from either Artist_ContractedWith, or Discography_Main joined with Album table (because it is meaningless to get Discography_Main on its own–Discography_Main is either an Album or Song according to ER diagram concept).
They are then allowed to select **any number of attributes** regardless of table choice (so as long as the attributes exist for the respective table, of course)

- If the user chooses Artist_ContractedWith…
  - they can pick Field1 to be either “artistName” or “country”. Val1 can be anything
  - They can pick Field2 to be either “numOfMembers” or “age”. Val2 can be anything

  For example,

  ```
  SELECT artistID, age
  FROM Artist_ContractedWith
  WHERE country='South Korea'  where 'South Korea' is an example value for
  :val1 AND numOfMembers>=1  where 1 is an example value for :val2
  ```

  For example,

  ```
  SELECT artistName, age, biography
  ```

```
FROM Artist_ContractedWith
WHERE artistName='(G)I-DLE' AND age>=1
```

For example,

```
SELECT *
FROM Artist_ContractedWith
WHERE country='Canada' AND age>=20
```

- If the user chooses Discography_Main joined with Album…
  - They can pick Field1 to be either “genre” or “discoName”. Val1 can be anything
  - They can pick Field2 to be either “dID” or “numOfSongs”. Val2 can be anything

For example,

```
SELECT discoName, dID
FROM Discography_Main, Album
WHERE genre='Kpop' where 'Kpop' is an example value for {:val1} AND
dID>=1  where 1 is an example value for {:val2}
```

For example,

```
SELECT discoName, dID, genre
FROM Discography_Main, Album
WHERE discoName='GITTU RAP' AND numOfSongs>=1
```

For example,

```
SELECT dID, genre, numOfSongs
FROM Discography_Main, Album
WHERE genre='Kpop' AND numOfSongs>=2
```

PROJECTION QUERY

MAINLY implemented in Artist_ContractedWithRepository, projection function (backend)

```
SELECT ATTRIBUTES_FROM_USER_INPUT
FROM Artist_ContractedWith;
```

JOIN QUERY

MAINLY implemented in AlbumRepository, getAllAlbumsWithName function (backend)

```
SELECT *
FROM Album A, Discography_Main D
WHERE A.albumID = D.dID AND D.discoName = :discoName
```

AGGREGATION WITH GROUP BY QUERY

MAINLY implemented in Artist_ContractedWithRepository, aggregationGroupBy function (backend)

```
SELECT R.artistID, COUNT(dID) AS numOfDiscography
FROM Releases R
GROUP BY R.artistID;
```

AGGREGATION WITH HAVING QUERY

MAINLY implemented in Artist_ContractedWithRepository, aggregationHaving function (backend)

```
SELECT A.artistName, MIN(D.releaseDate) as earliestReleaseDate
FROM Artist_ContractedWith A, Discography_Main D, Releases R
GROUP BY A.artistName
HAVING COUNT(D.dID) > 1;
```

NESTED AGGREGATION WITH GROUP BY QUERY

MAINLY implemented in Artist_ContractedWithRepository, nestedAggregation function (backend)

```
SELECT artistID
FROM Artist_ContractedWith A
WHERE (
    SELECT AVG(songCount)
    FROM (
        SELECT albumID,
               (SELECT COUNT(*) FROM Song S WHERE S.songID = Album.albumID) AS
songCount
        FROM Album
        WHERE A.artistID = Album.albumID
    ) AS inner_query
) =
(
    SELECT MIN(avgSongs)
    FROM (
        SELECT artistID,
               AVG(
                    (SELECT COUNT(*) FROM Song S WHERE S.songID = Album.albumID)
               ) AS avgSongs
        FROM Album
        GROUP BY artistID
    ) AS overall_avg_query
);
```

DIVISION QUERY

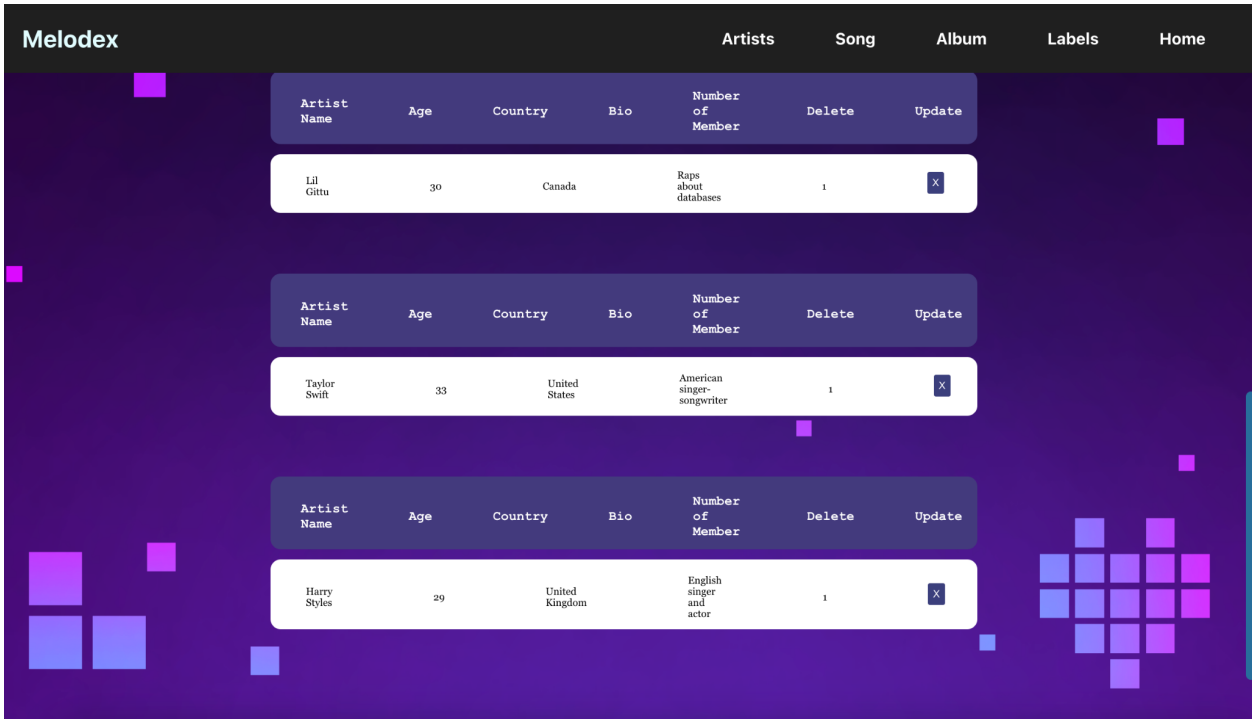
MAINLY implemented in Artist_ContractedWithRepository, division function (backend)

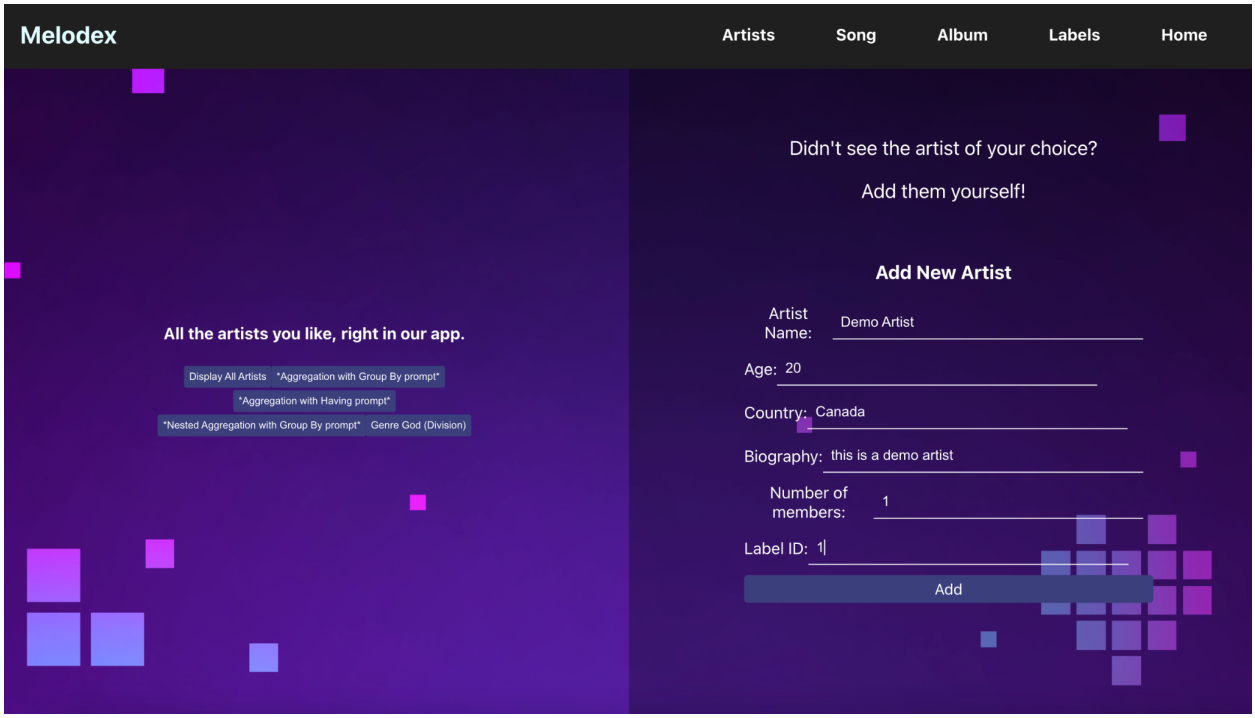
```
SELECT artistID
FROM Artist_ContractedWith A
WHERE NOT EXISTS (
    SELECT DISTINCT genre
    FROM Discography_Main D
    WHERE D.genre NOT IN (
        SELECT DISTINCT Dm.genre
        FROM Discography_Main Dm, Song S, Releases R
        WHERE Dm.dID = R.dID AND S.songID = R.dID AND R.artistID = A.artistID
    )
);
```

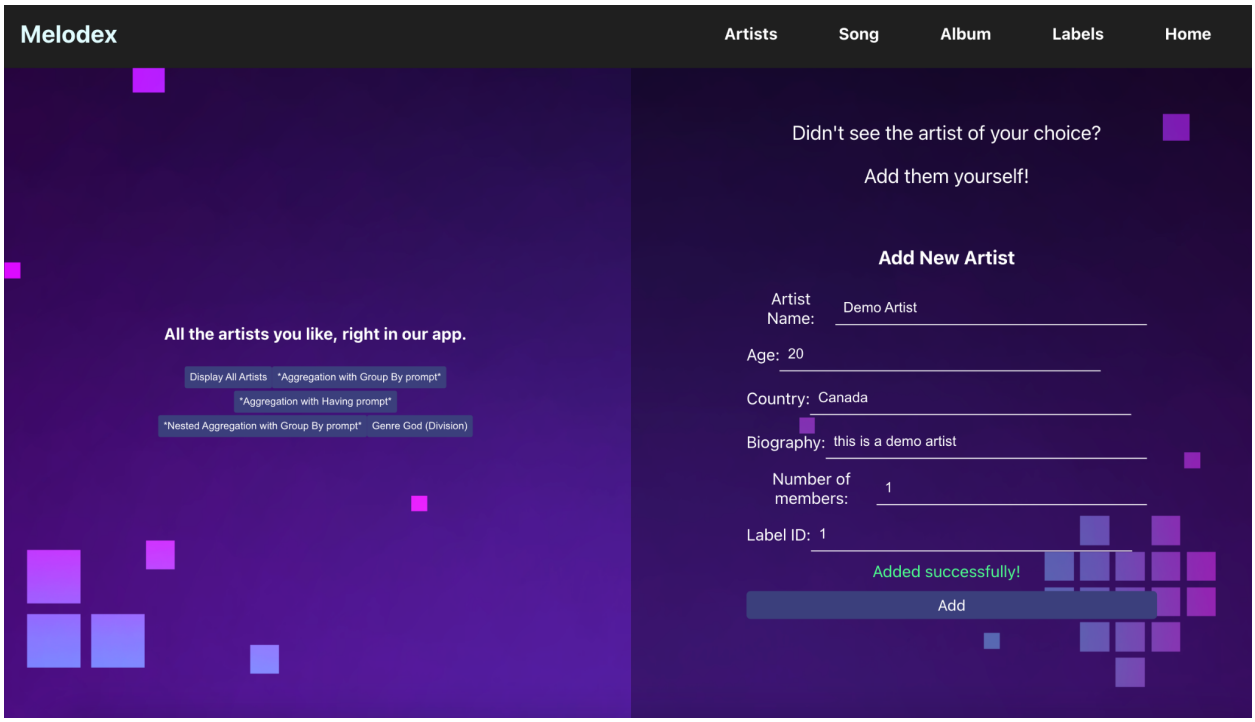
Screenshots of the sample output of the queries using the GUI

INSERTION QUERY:

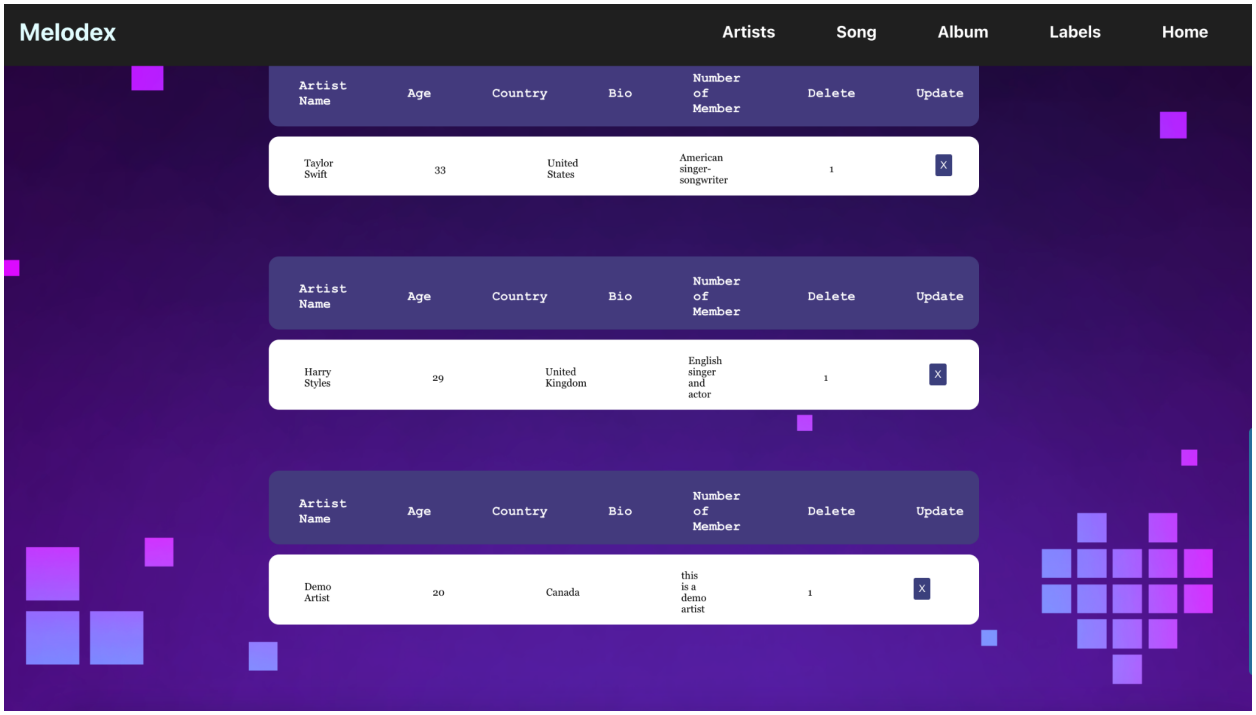
Before insertion:




Melodex
Artists
Song
Album
Labels
Home
All the artists you like, right in our app.
Display All Artists
Aggregation with Group By prompt
Aggregation with Having prompt
Nested Aggregation with Group By prompt
Genre God (Division)
Didn't see the artist of your choice?
Add them yourself!
Add New Artist
Artist Name:
Demo Artist
Age:
20
Country:
Canada
Biography:
this is a demo artist
Number of members:
1
Label ID:
1
Add

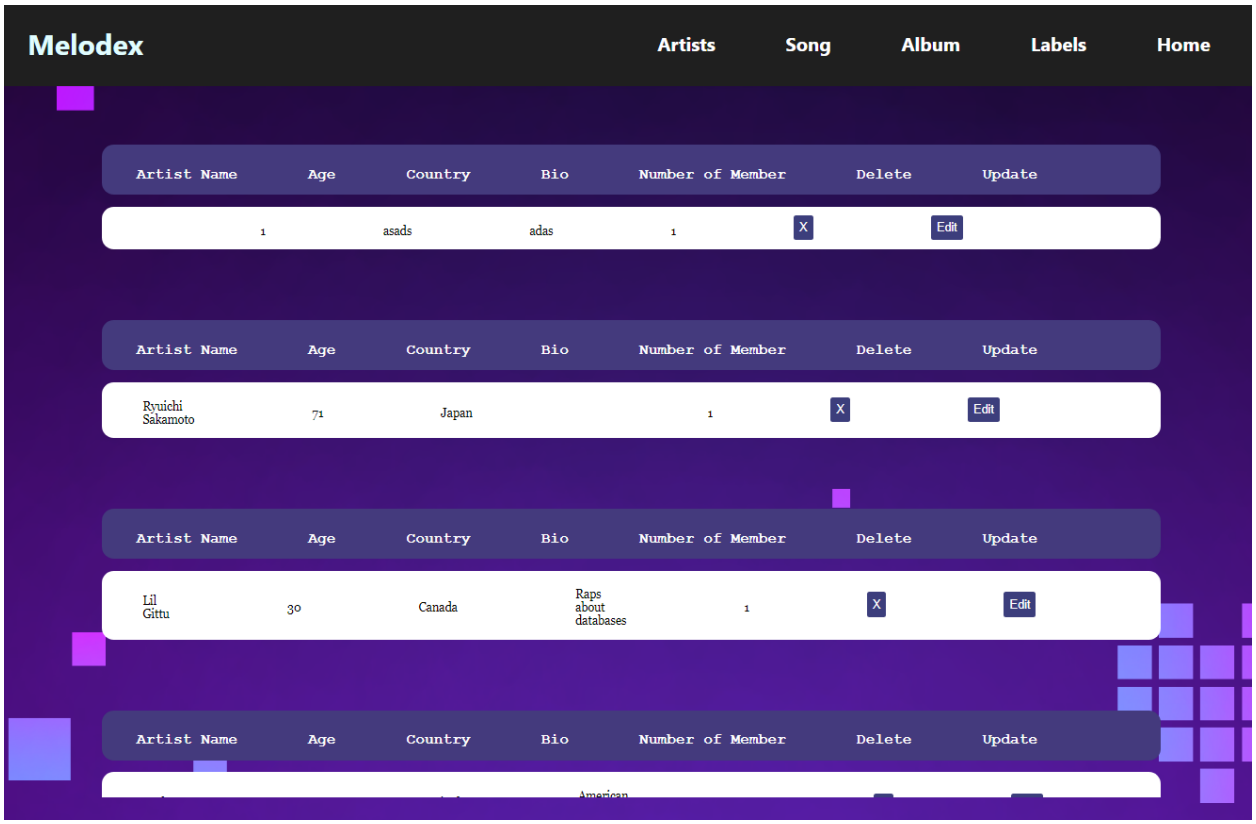

Melodex
Artists
Song
Album
Labels
Home
All the artists you like, right in our app.
Display All Artists
Aggregation with Group By prompt
Aggregation with Having prompt
Nested Aggregation with Group By prompt
Genre God (Division)
Didn't see the artist of your choice?
Add them yourself!
Add New Artist
Artist Name:
Demo Artist
Age:
20
Country:
Canada
Biography:
this is a demo artist
Number of members:
1
Label ID:
1
Added successfully!
Add

After insertion:

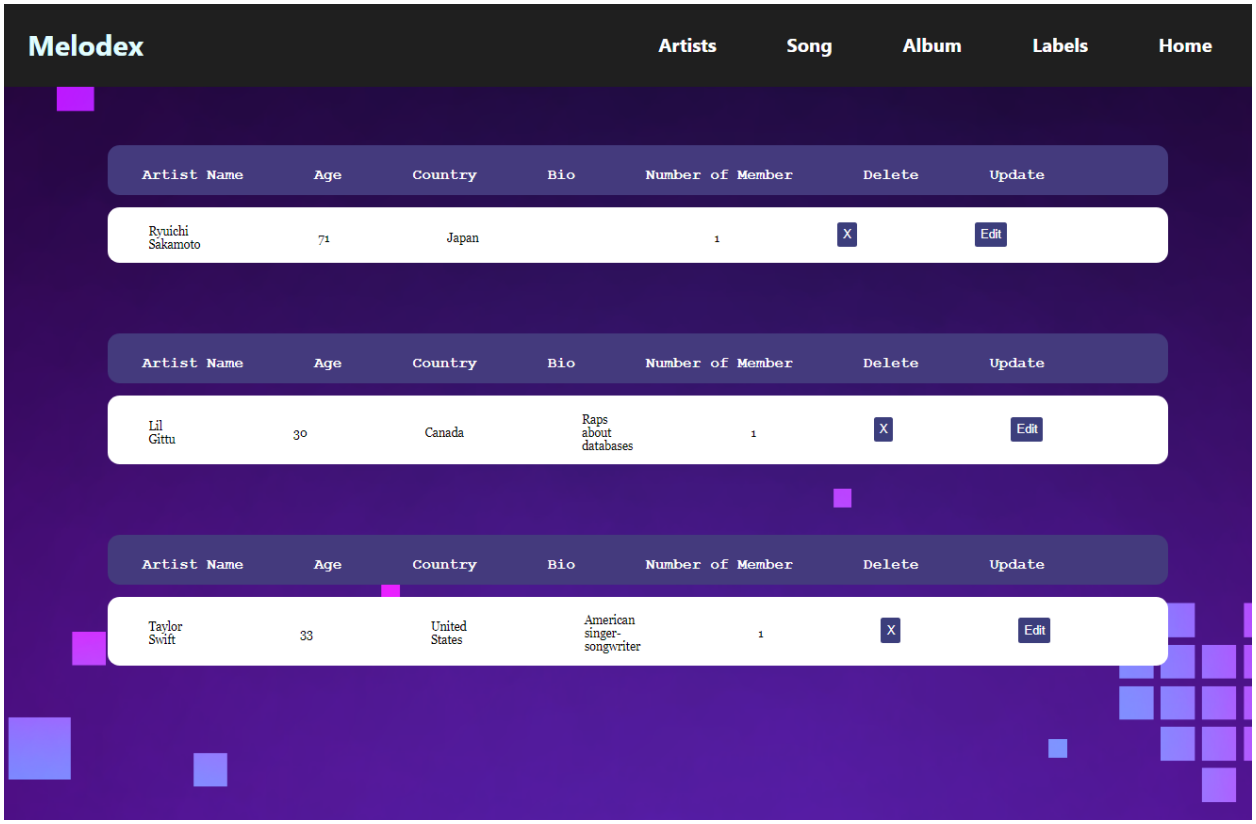


DELETION QUERY:

Before deletion:

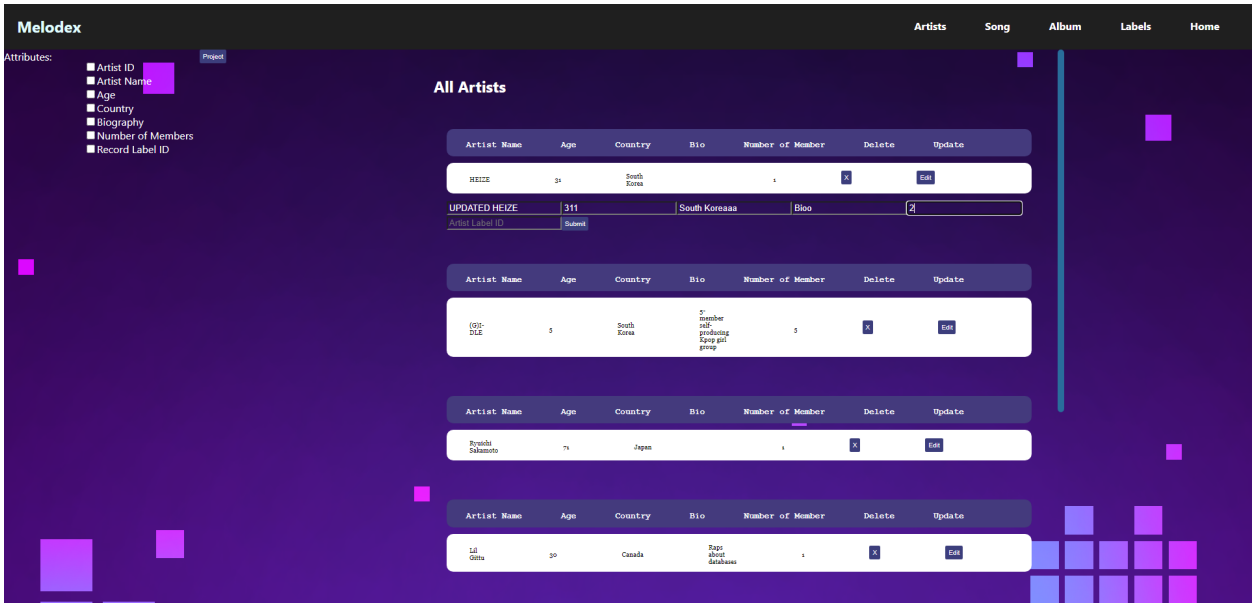


After deletion:

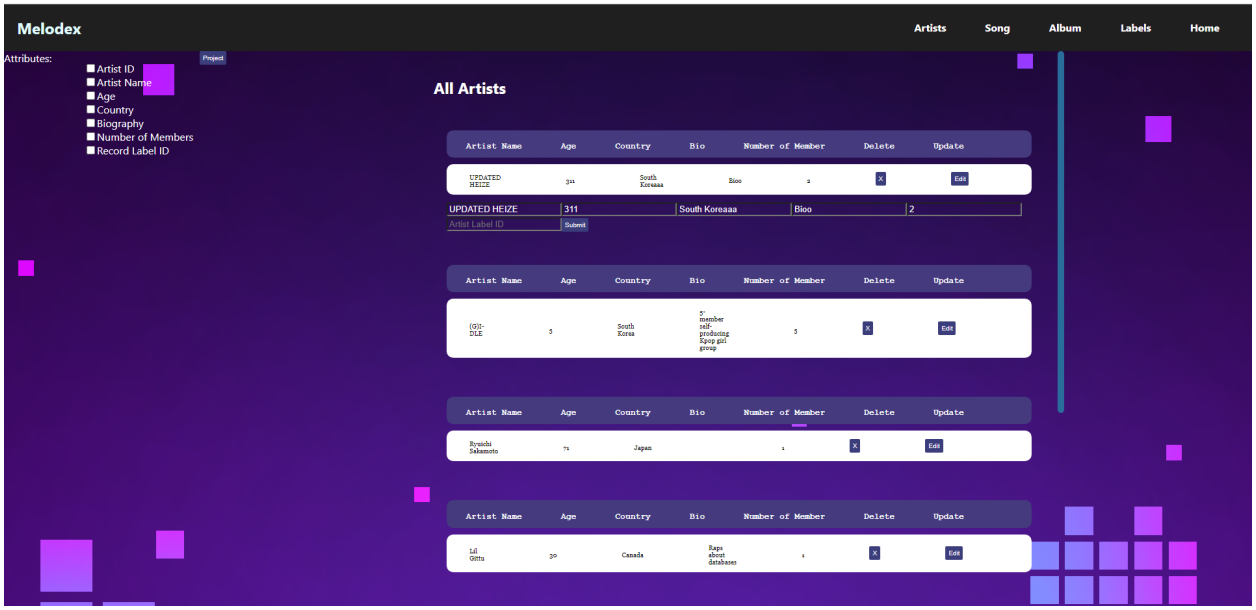

Melodex
Artists
Song
Album
Labels
Home
Artist Name
Age
Country
Bio
Number of Member
Delete
Update
Ryuichi Sakamoto
71
Japan
1
X
Edit
Artist Name
Age
Country
Bio
Number of Member
Delete
Update
Lil Gittu
30
Canada
Raps about databases
1
X
Edit
Artist Name
Age
Country
Bio
Number of Member
Delete
Update
Taylor Swift
33
United States
American singer-songwriter
1
X
Edit

UPDATE QUERY

Before clicking “submit”:

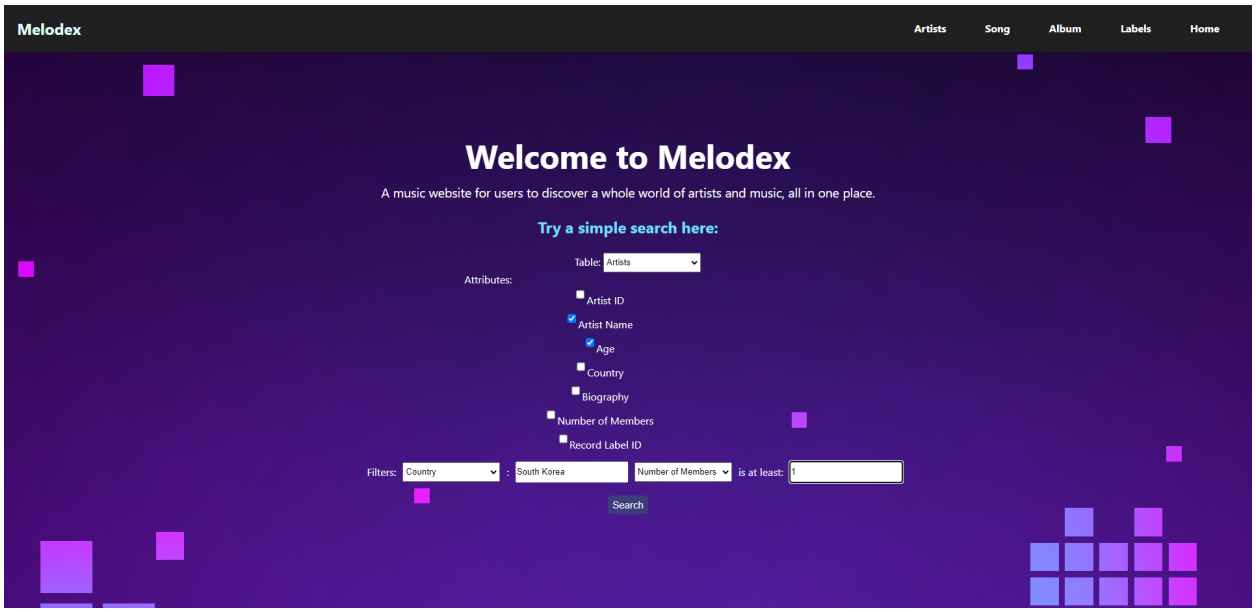


After clicking “submit”:

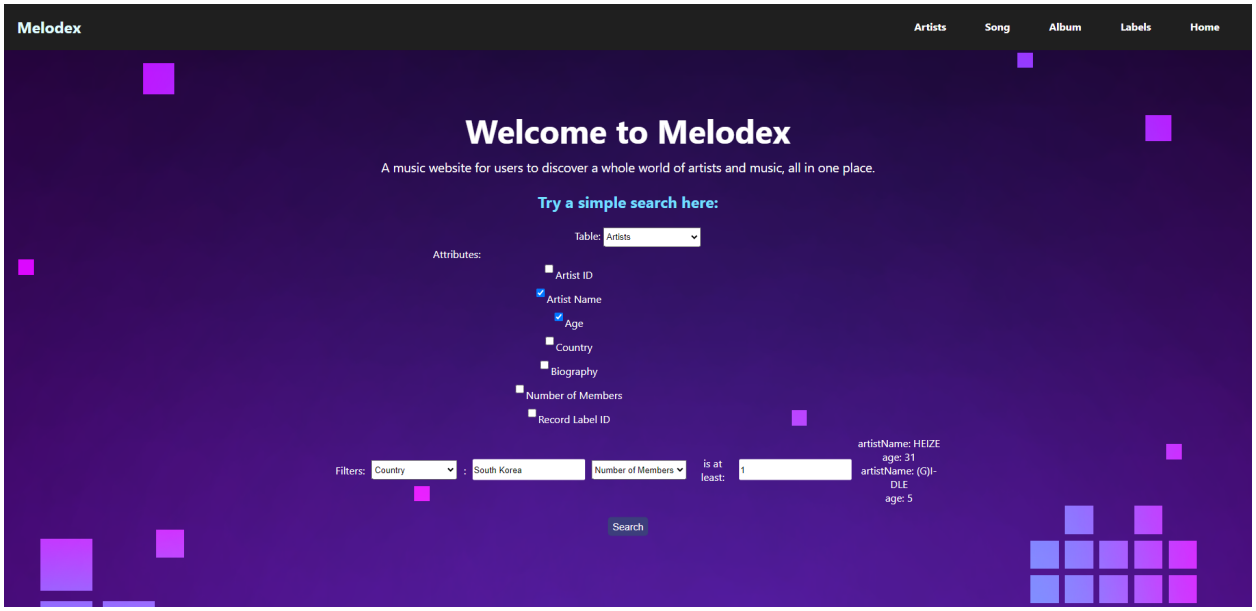


SELECTION QUERY:

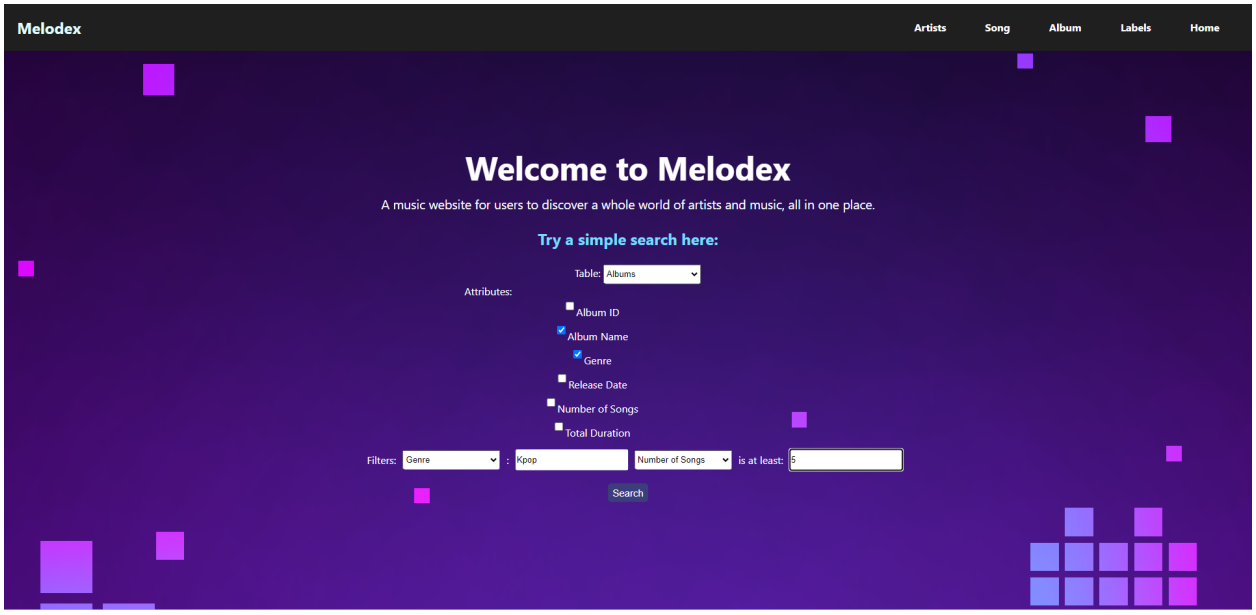
Before clicking “search”:



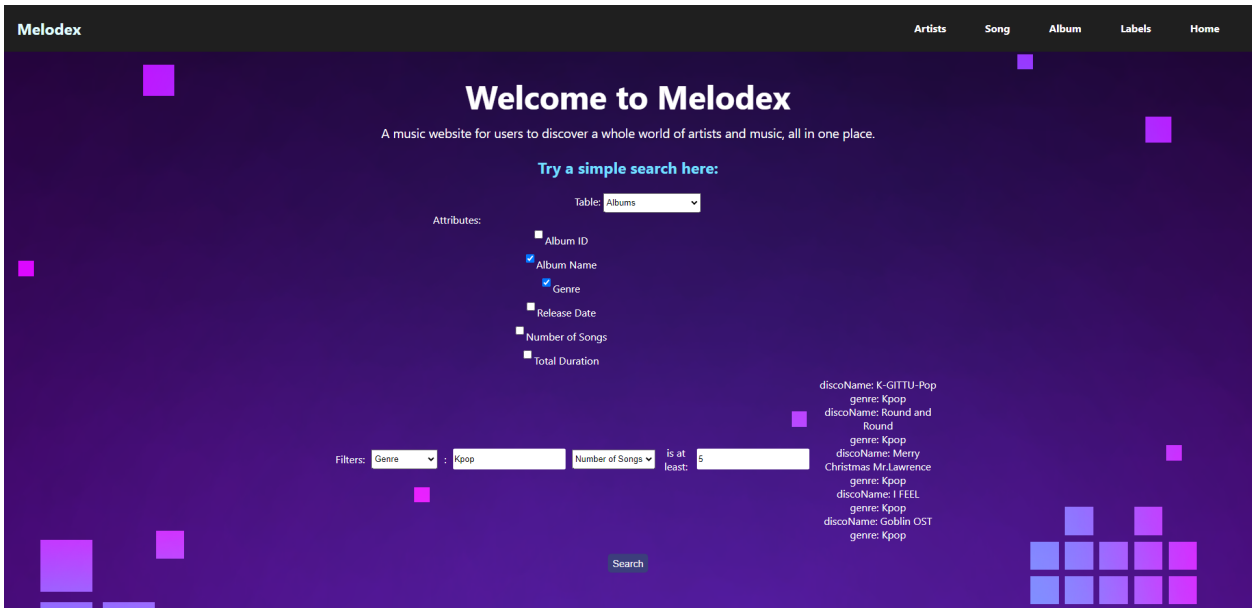
After clicking “search”:



Before clicking search:

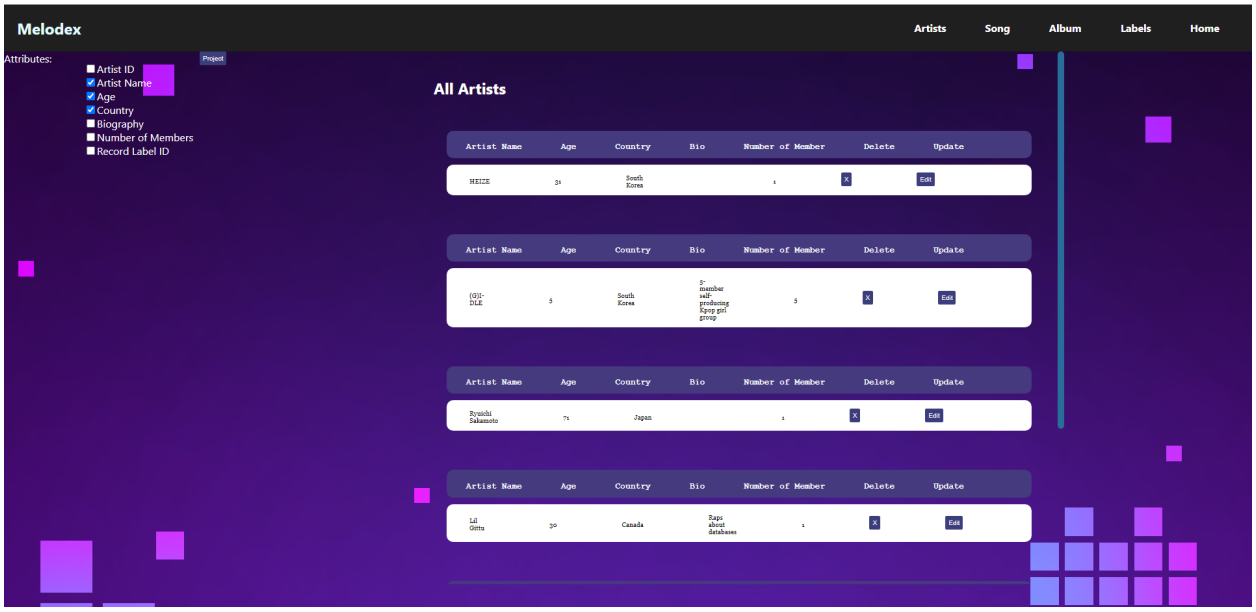


After clicking search:

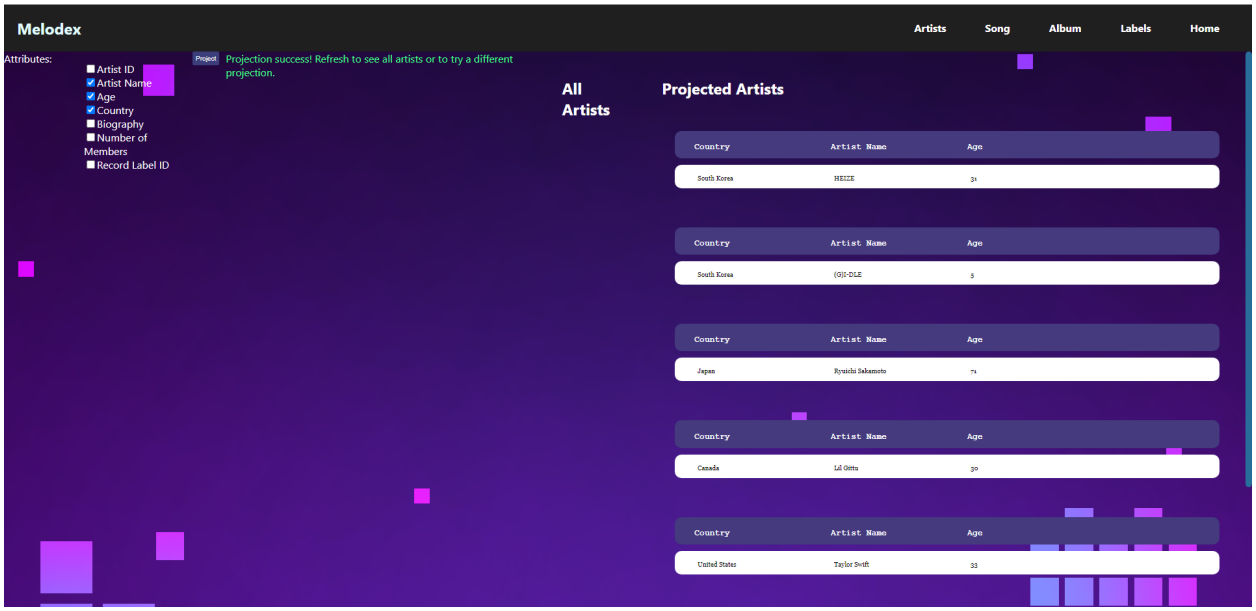


PROJECTION QUERY

Before clicking “project”:

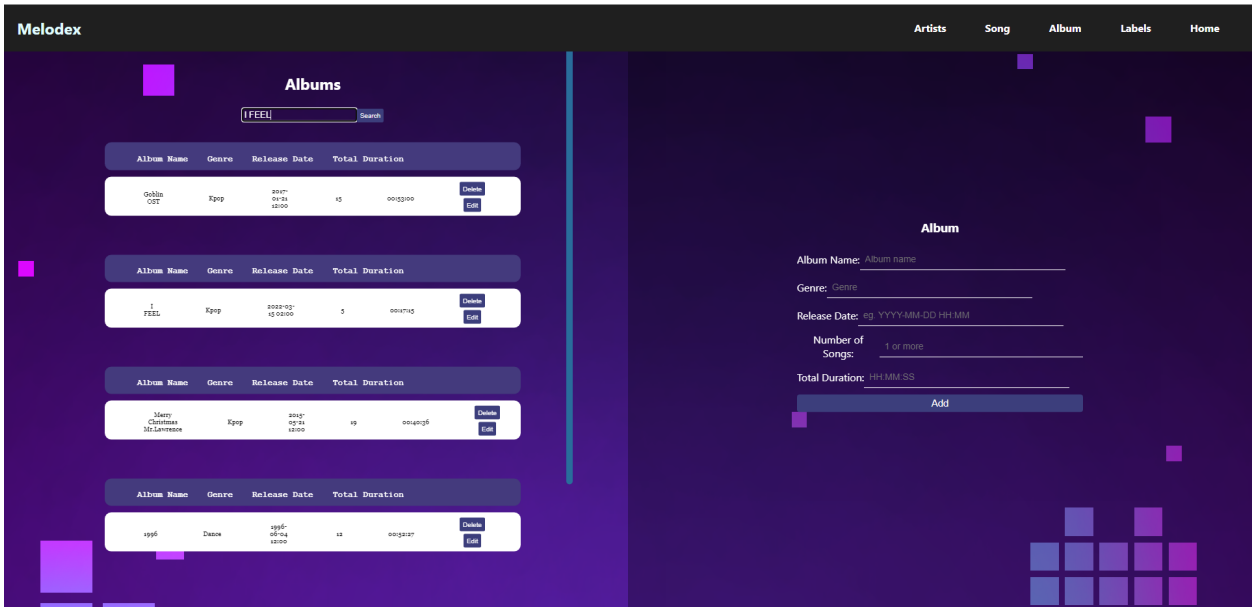


After clicking project:

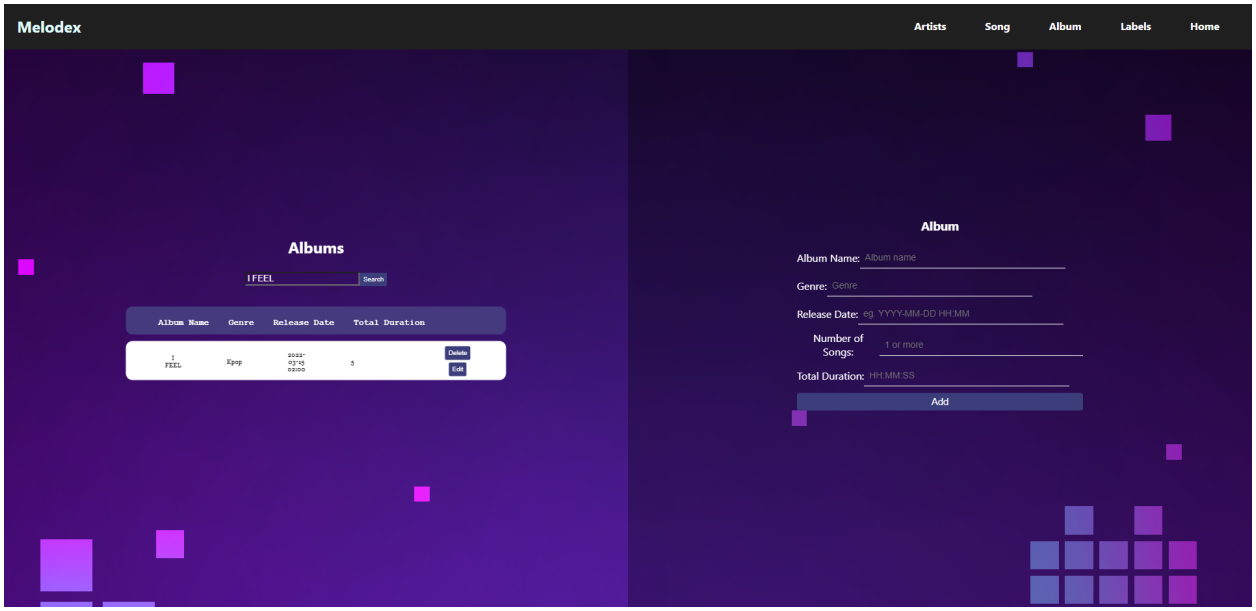


JOIN QUERY

Before clicking search:

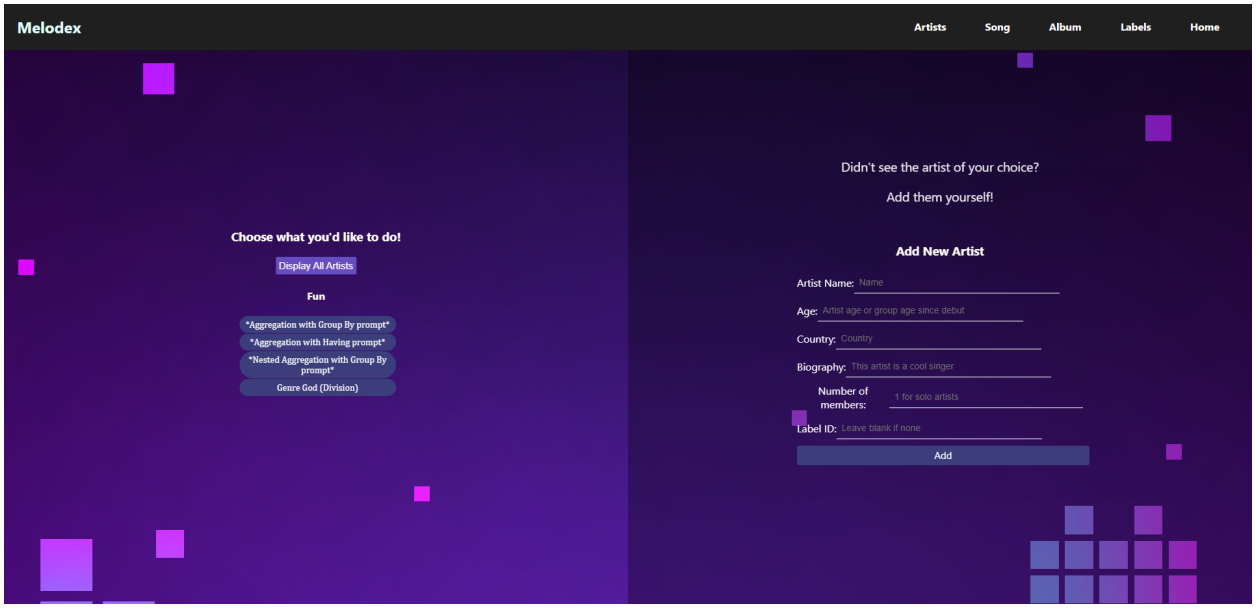


After clicking search:

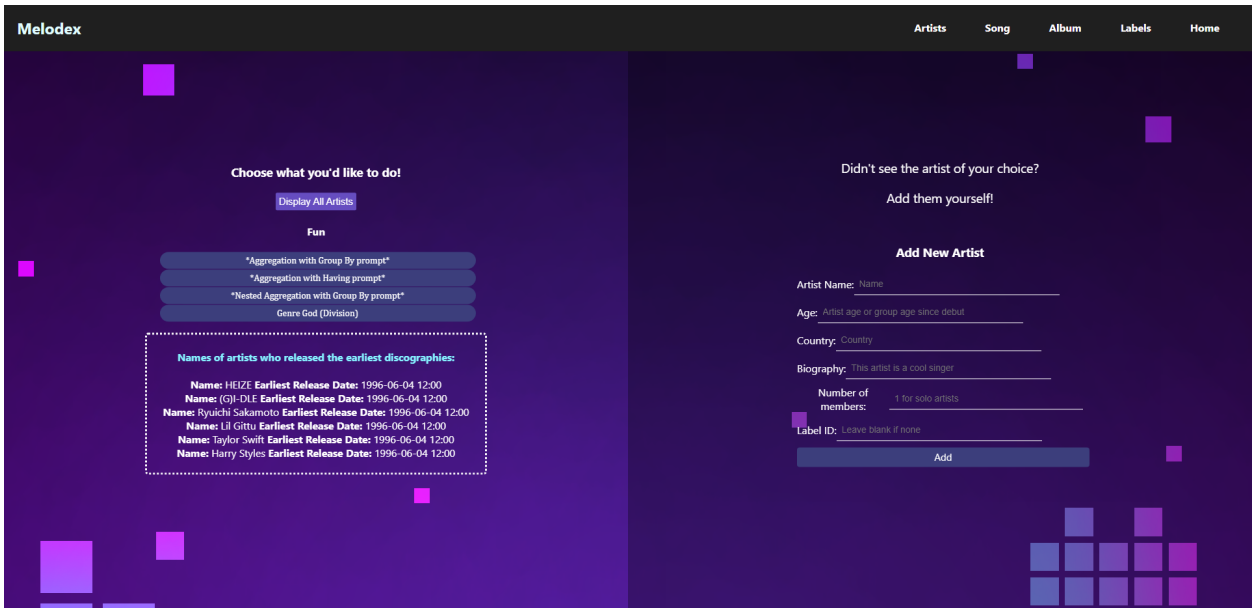


For Division and all the Aggregations, all their buttons are on the same page so I will only show one “before” image.

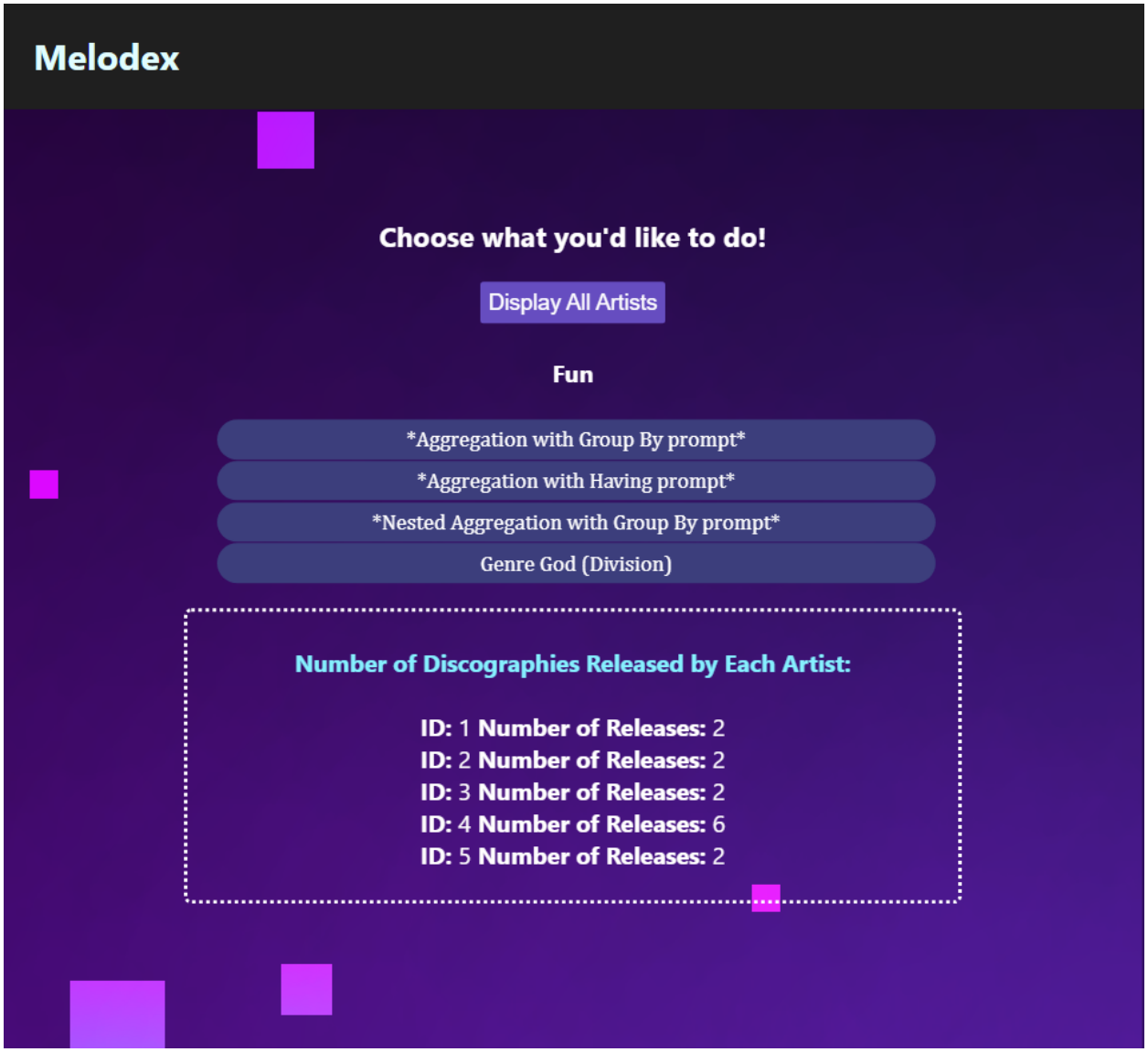
Before:



AGGREGATION WITH HAVING

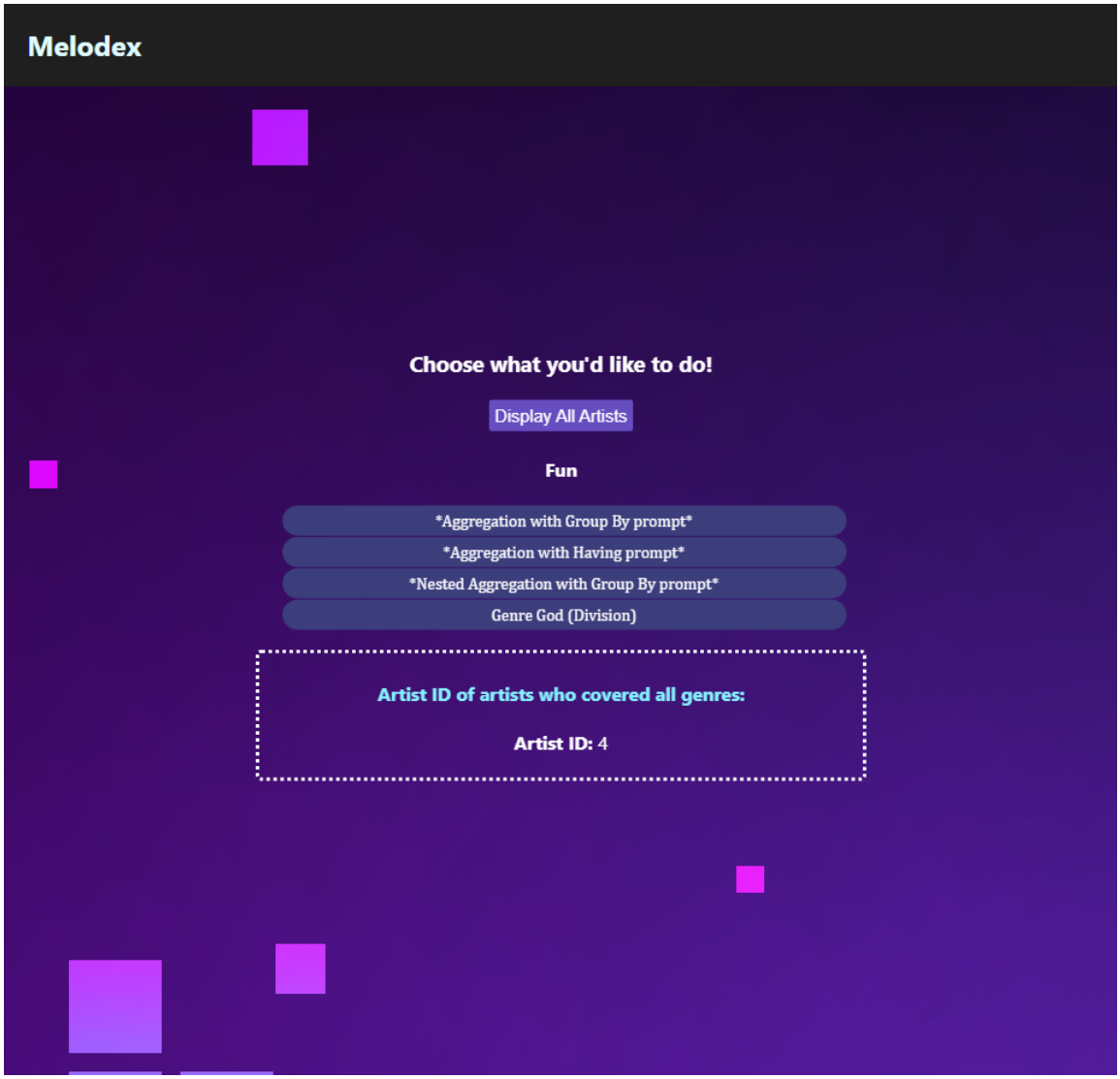


AGGREGATION WITH GROUP BY



NESTED AGGREGATION WITH GROUP BY PROMPT:

DIVISION:


Melodex
Choose what you'd like to do!
Display All Artists
Fun
Aggregation with Group By prompt
Aggregation with Having prompt
Nested Aggregation with Group By prompt
Genre God (Division)
Artist ID of artists who covered all genres:
Artist ID: 4

Example snippet of how we use data from backend to frontend:
Here is an asynchronous function that makes a call to the API route that will fetch all the albums with the name we specified for us. We parse what we get back into JSON so we can use it. The implementation for the API routes are implemented in our Java backend.

```
const handleSearchClick = async (e) => {
  e.preventDefault();
  if (name === '') {
    return;
  }
  const response = await fetch('http://localhost:8080/api/albums/all/' + name, {
    method: 'GET'
  }).catch((err) => {
    console.log(err);
  })

  if (response.ok) {
    const json = await response.json();
    setAlbums(json);
    console.log("Fetched artists! ", json)
  } else {
    console.log("Could not fetch artists");
  }
}
```

Typically, a button will be clicked to activate a function like this.

```
<div className={styles.searchArea}>
    <h2>Albums</h2>
    <input type="text" placeholder="Search by Name" onChange={(e) => setName(e.target.value)}></input>
    <button onClick={handleSearchClick}>Search</button>
</div>
```

Any requests that require a body, such as POST or UPDATE requests will have a JSON stringified body (see highlighted line). For example, in this function to add an artist, we create an artist object from whatever the user inputs from our frontend (its fields will be stored in states that “react” to user inputs). Then, again, we make the appropriate call to the according API route. Since we are not getting anything back from POST or UPDATE requests (it is set up like this in the backend), we do not do response.json() this time, i.e. we do not parse the response into json..

```
const handleAddClick = async (e) => {
  e.preventDefault();

  const artist = {
    artistName,
    age,
    country,
    biography: bio,
    numOfMembers,
    labelID,
  };

  const response = await fetch("http://localhost:8080/api/artists/add", {
    method: "POST",
    body: JSON.stringify(artist),
    headers: {
      "Content-Type": "application/json",
    },
  });

  if (response.ok) {
    console.log("New artist added!");
    setSuccessAdd(true);
    setError(false);
  } else {
    setError(true);
    setSuccessAdd(false);
    console.log("Error!");
  }
};
```

Once again, this function will be used by a button, like this:

```
)}
<div className={styles.btn} onClick={handleAddClick}>
  Add
</div>
```

Quick backend overview: For the API routes in the backend, they can all be found in the classes under the Controller folder.
For example, anything that is "http://localhost:8080/artists/SOMETHING" will be in Artist_ContractedWithController

```
package com.musicmanagementsystem.controller;

import ...

@RestController
@RequestMapping("/api/artists")
@CrossOrigin
public class Artist_ContractedWithController {
    @Autowired
    public Artist_ContractedWithService artist_contractedWithService;

    @Autowired
    public JdbcTemplate jdbcTemplate;

    @PostMapping("/add")
    public void addNewArtist(@RequestBody Artist_ContractedWithBody reqBody) {
        artist_contractedWithService.insertNewArtist(reqBody.getArtistName(), reqBody.getAge(), reqBody.getCountry()
                reqBody.getBiography(), reqBody.getNumOfMembers(), reqBody.getLabelID());
        System.out.println("Added new artist!");
    }
```

For the native queries (SQL) that the controllers actually use, they are MOSTLY found in the repositories. But, queries that don't return all attributes may be found in the entity classes (for DTO mapping), and dynamic queries may be found in the Controller (eg. selection query is in GeneralController, projection query is in Artist_ContractedWithController). The Service classes mainly add as a middle man for the Controller classes and Repositories.

Example of query implementation in Artist_ContractedRepository for the add a new artist function:

```java
package com.musicmanagementsystem.repository;

import ...

@Repository
public interface Artist_ContractedWithRepository extends JpaRepository<Artist_ContractedWith, Integer> {

    String insertNewArtistQuery = "INSERT INTO Artist_ContractedWith (artistName, age, country, biography, numOfMembers, labelID) " +
            "VALUES (:artistName, :age, :country, :biography, :numOfMembers, :labelID)";
    String getAllArtistsQuery = "SELECT * FROM Artist_ContractedWith";
    String updateArtistQuery = "UPDATE Artist_ContractedWith " +
            "SET artistName= :artistName, age=:age, country=:country, biography=:biography, numOfMembers=:numOfMembers, labelID=:labelID " +
            "WHERE Artist_ContractedWith.artistID= :artistID";
    String dynamicSelection = "SELECT :attributes FROM :table ";
    String deleteArtistByIdQuery = " DELETE FROM Artist_ContractedWith WHERE artistID = :artistID";
    String nestAggregationQuery = "SELECT artistID\n" +
            "FROM Artist_ContractedWith A\n" +
            "WHERE (\n" +
            "    SELECT AVG(songCount)\n" +
            "    FROM (\n" +
            "        SELECT albumID,\n" +
            "               (SELECT COUNT(*) FROM Song S WHERE S.songID = Album.albumID) AS songCount\n" +
            "        FROM Album\n" +
            "        WHERE A.artistID = Album.albumID\n" +
```

```java
    @Modifying
    @Transactional
    @Query(value=insertNewArtistQuery, nativeQuery = true)
    public void insertNewArtist(@Param("artistName") String artistName, @Param("age") int age, @Param("country")String country,
                                @Param("biography") String biography, @Param("numOfMembers") int numOfMembers, @Param("labelID") Integer labelID);

    @Query(value=getAllArtistsQuery, nativeQuery = true)
    public List<Artist_ContractedWith> getAllArtists();

    @Modifying
    @Transactional
    @Query(value=updateArtistQuery, nativeQuery = true)
    public void updateArtist(@Param("artistID") int artistID, @Param("artistName") String artistName, @Param("age") int age, @Param("country")String co
                             @Param("biography") String biography, @Param("numOfMembers") int numOfMembers, @Param("labelID") Integer labelID);

    // SELECTION OPERATION to submit
    @Query(value = dynamicSelection, nativeQuery = true)
    List<Object> getDynamicSelection(@Param("table") String table, @Param("attributes") String attributes);

    @Modifying
    @Transactional
    @Query(value = "INSERT INTO Releases (artistID, dID) VALUES (:artistID, :dID)", nativeQuery = true)
    public void releaseDiscography(@Param("artistID") Integer albumID, @Param("dID") Integer discoID);
```