MixTech

Created by: *Taijing Chen, Nicholas Kimball, Alex Do & Darshna Doshy*

I. Introduction

a. Motivation

Music is something that all of our group members are passionate about. In a discussion, we identified that Professional Musicians, DJs and even music lovers are always looking for songs that go together. This could be a collection of songs they would want to play for an event, mixing two songs, or just find similar songs. That is how we came up with the idea of MixTech. This web-based application will help overcome the lack of professional-oriented music applications in the market and it will help users find and keep track of music compatible with certain styles using data and song metrics.

b. Description of Application

MixTech is a web-based application mainly designed for Professional Musicians and DJs who need songs to mix or have a list of songs that go well together. Authenticated users will be able to create, modify and delete playlists. They will also be able to allow other users to see their playlists by setting the privacy to public, or hide it by setting privacy to private. Users will be able to add and delete songs from a playlist with ease. If the user is certain that two songs go well together, they can create a Match. A match consists of two songs that surely go together. The user can add one song to the match and add the other one later. Complete matches will be seen by anyone who searches for songs in those matches. They can also search for songs either by their name, or by song metrics offered in the Advance Search.

c. Organization of the Report

This report will lead you to our journey on MixTech development. In the first part of the report, we will present an overview of Mixtech: why we choose this topic, what MixTech does, and how each of us has participated in the project. In the second part, we will introduce where the data came from, and how MixTech was designed, implemented and evaluated. The last part of the report concludes with our reflections and future plans of MixTech.

d. Task Assignment for each member

Member	Task
Alex Do	- Frontend Development - Data Collection + Cleaning
Darshna Doshy	- SQL - Testing - Assisted with Java Backend Development
Nicholas Kimball	- Java Backend Development - Server configuration - Application deployment
Taijing Chen	- Java Backend Development - Assisted with Frontend Development

II. Implementation

a. Description of the System Architecture

Data collection + Cleaning: Python/Spotify

Front-end: React.js / Redux

Back-end: SpringBoot+Java

Database: MySQL MariaDB

Server: Amazon AWS Linux Server

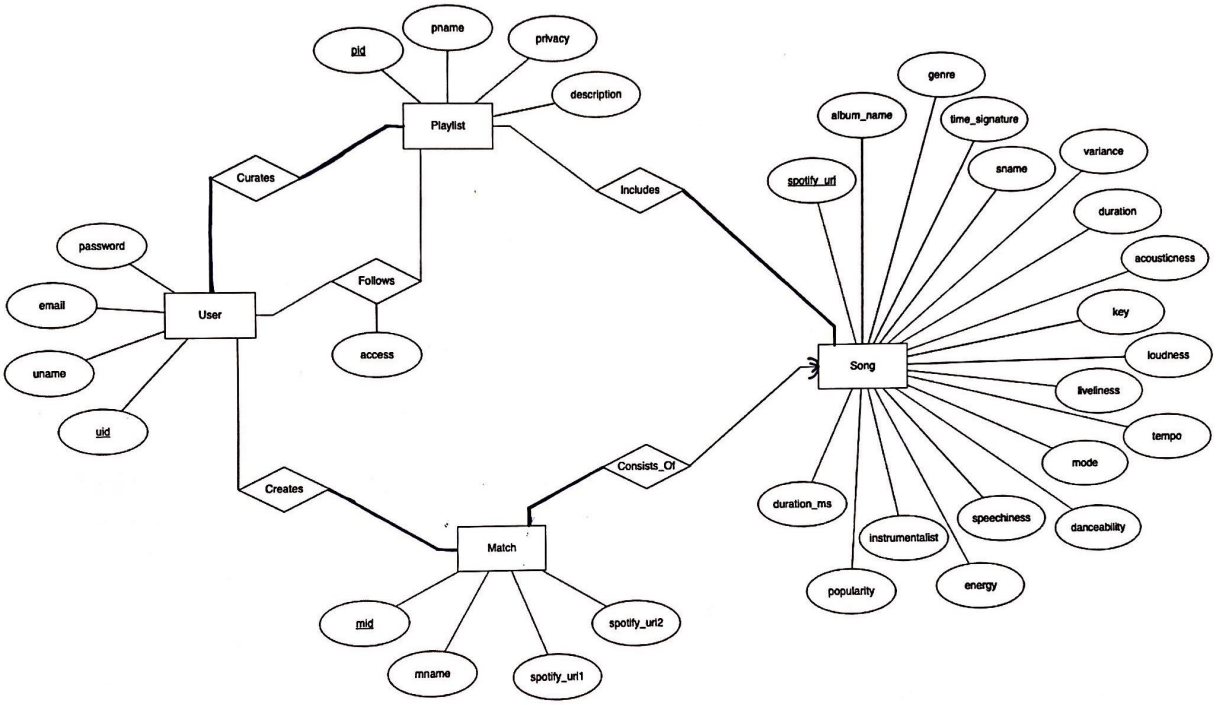
b. Description of the dataset

Spotify, being one of the largest audio streaming companies in North America, South America and Europe, provides digital music service to millions of people. Owing to its popularity, we were able to have access to millions of songs, their musicians, their

trendings, and many other important music features (keys, tempo, etc.). This size and variety gave us flexibility to manipulate the data and build the application we wanted.

Scraping data was a hassle because we had to indirectly query everything. We could only get a list of albums by scraping all albums from Billboard's top 200 albums since Jan. 1963 to July. 2019. We then had to query every album by name through the spotify API python wrapper, spotipy. From there, we grabbed every track in each album and then for each track we got the track's audio features. The final result was a dataset consisting of 316,312 tracks that each had 14 audio features, the artist of the track, its album and popularity.

c. ER Diagram



d. Relational Model

i. Users(<u>uid: BIGINT</u>, uname: STRING, email: STRING, password: STRING)

ii. Playlists(<u>pid: BIGINT</u>, pname: STRING, privacy: INT, description: STRING)

iii. Song(spotify_uri: STRING, sname: STRING, album_name: STRING, danceability: REAL, energy: REAL, skey: INT, loudness: REAL, smode: INT, speechiness: REAL, acousticness: REAL, instrumentalness: REAL, liveness: REAL, valence: REAL, tempo: REAL, duration_ms: BIGINT, time_signature: INT, popularity: INT)

iv. Matches(mid: BIGINT, mname: STRING, spotify_uri1: STRING, spotify_uri2: STRING)

v. Curates(uid: BIGINT, pid: BIGINT)

vi. Include(spotify_uri: STRING, pid: BIGINT)

vii. Creates(mid: BIGINT, uid: BIGINT)

viii. Follows(uid: BIGINT, pid: BIGINT, access: INT)

e. Implementation: Description of the Prototype

We have several different functionalities the user can perform and get desired results -

i. User Registration and Login - A new user can register or an existing user can login to use the application. Authentication has been done with the help of base64 encoding and a unique token is passed from the frontend every time the user logs in which allows them to explore all the functions of the app. The frontend stores the token in the browser's local storage in order to persist the user's information and application session. The home page will have routes to the user's Matches, Playlists and two Search tabs which will give them access to the Basic and Advanced Search respectively.

ii. Basic and Advance Search - A list of songs and all its audio features will appear when a user searches for a song just by its name (basic search). When the user searches for songs by their audio features (advanced search), a list of songs that match the criteria will be displayed along with its audio features. All songs are returned in descending order of popularity.

iii. Create Playlists - The user will be able to create their own playlist, populate it with songs that fit the genre/mood or just mix it up. They can decide if they want to publish their playlist so that all the users can see it, or keep their creation just to themselves. They can also add a description to the playlist.

iv. Delete Playlists - Once deleted, the playlist will not exist in the user's account and will not be visible to other users if it was public.

v. Update Privacy - If the user changes their mind and wants to change their privacy settings from either Public or Private to Private or Public, they can do it without any hassle.

vi. Add/Delete Songs from a Playlist - The user can modify their existing playlist by adding or deleting songs from it. They can add the song from the results of their Basic/Advanced Search. To delete a song, they must simply choose the song they want to discard and hit the delete button.

viii. Create/Delete Matches - Matches are two songs that definitely go together and can be mixed according to the user. The user can create a match by selecting a result tuple from the search results and add it to an incomplete match. They can also create an entirely new match that'll be matched later. This new match will go in incomplete matches. They can also delete the matches they create from the matches route.

viii. See Completed Matches as a part of Basic Search - When searching for a song, the users will be able to see a list of completed matches created by any user that have that particular song. In this way users can share their matches and also other users can find songs that go together for inspiration and add it to their own library.

f. Testing and Evaluation

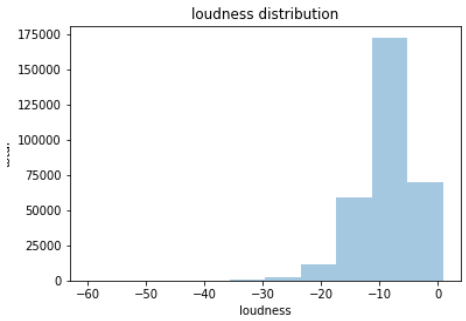
We tested our application using Postman and on the Frontend with React/Redux dev tools Chrome extension with various trivial and non-trivial test cases.

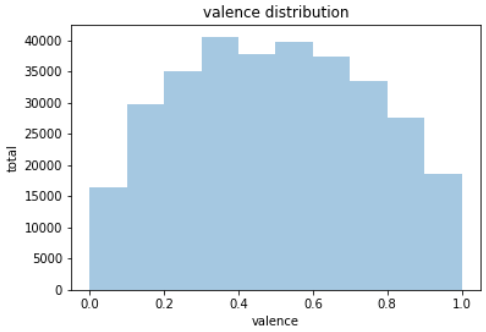
We also made sure that our queries were optimized. We calculated the cost of each query, before optimization and after optimization. We found that there was a huge drop in cost after optimization. Here is an example -

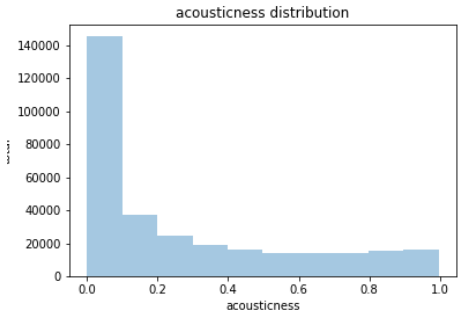
Query	Cost Before Optimization	Cost After Optimization
Select m.* from Creates c, Matches m where c.uid =?1	40,600	1,003

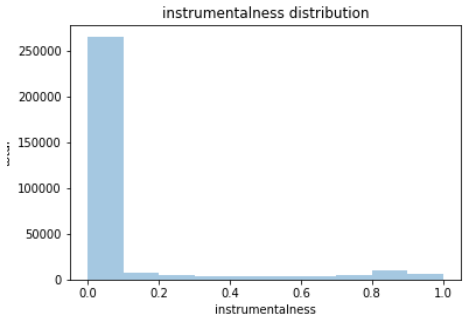
and c.mid = m.mid;		
Select m.mid, s1.sname, s2.sname from Matches m, Song s1, Song s2, Creates c where m.spotify_uri1 = s1.spotify_uri and m.spotify_uri2 = s2.spotify_uri and m.mid = c.mid and c.uid = ?1;	1,241,200	641,101

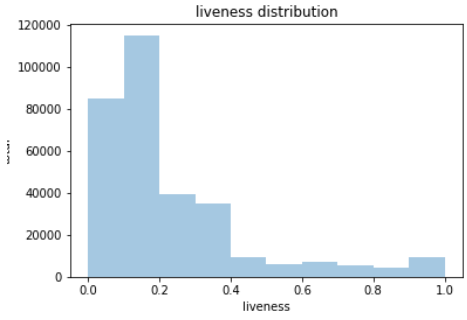
We calculated the distributions of attributes by writing a script in python and then compared our expected results to our actual results. We found that our queries returned the tuples correctly and in a timely manner.

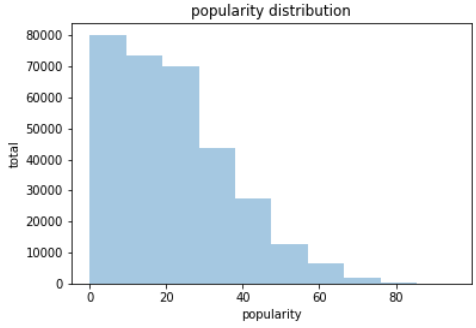


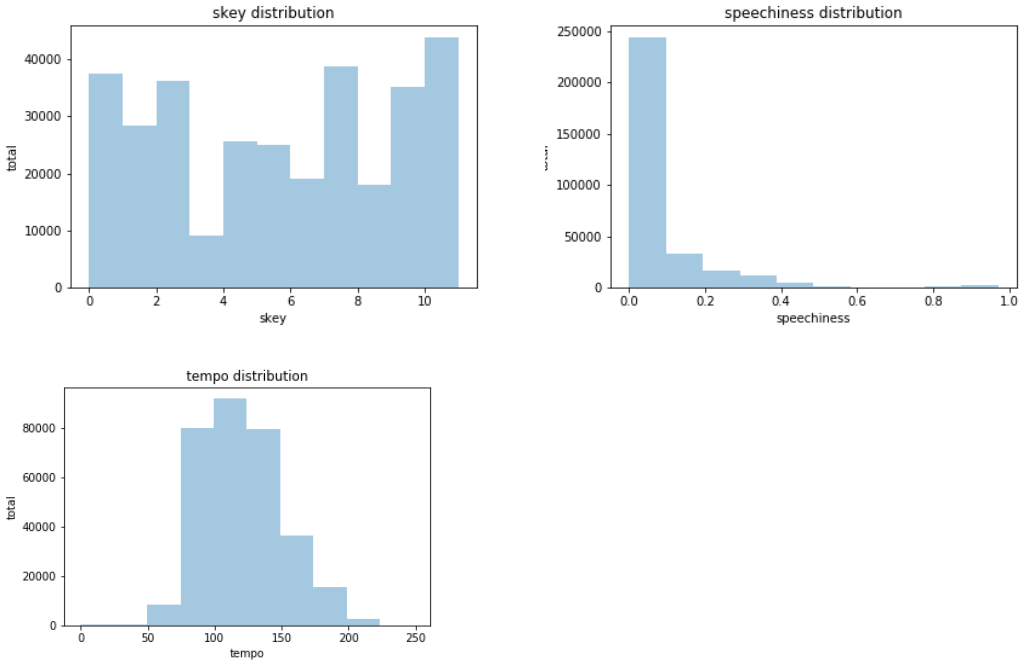












Distribution of Attributes of Song Metrics

Attribute	Distribution Stats	Expected Result	Actu
Valence	> .5	~150,000	150,858 (1.1 sec)
Acousticness	> .8	~31,200	31,696 (.3 sec)
Instrumentalness	> .8	~15,000	18,044 (.25 sec)
Liveness	> .8	~ 15,000	13,227 (.22 sec)
Loudness	< -40	~0	101 (.17 sec)
Popularity	> 80	~5000	5821 (.18 sec)
Skey	> 10	~30,000	22,495 (.27 sec)
Speechiness	> .8	~15,000	13,146 (.21 sec)
Tempo	< 40	~15,000	20,108 (.24 sec)

Table: Comparison of Expected vs Actual results

III. Conclusion

a. Lessons Learned

Building this project was a roller coaster ride. We were prepared for some turns but we also met with a lot of unexpected challenges and surprises. Solving those issues helped us get a better understanding of the various components of development. We learned various tools in this process - IntelliJ, Spring Boot, Maven, Redux, React, Postman, etc. Learning the Spring framework, setting up the project, building it, compiling it, managing dependencies was stimulating. We learned about dev ops and explored AWS. There were a lot of intricacies that we had to take care of. We had to make sure there was no redundancy in the code, that it worked well with the database and that our database returned the tuples correctly. While doing this we stumbled upon many ways to accomplish one task. We learned that we could write native queries in Spring, or write them in Java. We could also write them completely in Spring due to the built in functions Spring had to offer.

Frontend development was difficult but overtime the workflow became easier. One thing that was difficult about developing the user interface was that it required a lot of communication with the backend implementation - making sure routes were properly being requested, figuring out exactly what was returning from the backend and how to send data back to the backend in proper format to ultimately interface with the MySQL database. Another challenge was figuring out how to organize and maintain data flow. At first, we simply used React's one way data flow by passing state from parent to child components. However, as our application grew, we realized that this methodology of maintaining state would quickly become convoluted. In response, we adopted a popular paradigm in front end application development called Redux that maintains a global state. Redux allowed us to decouple state from presentation so we could take care of interfacing with the backend and have the user interface dynamically change in response to a change in state in a functional way. From there, workflow improved tremendously. Finally, the styling of the user interface was also challenging. Figuring out what components, html tags and event handlers to use meant a lot of stack overflow and documentation reading. Eventually the frontend development became very streamlined and we learned a lot about how to present our data to the end user in a digestible way.

Overall, the learning curve of this project was very steep but the experience was invaluable in helping us solidify our understanding of the course as well as garnering technical skills to engineer projects in the future.

b. Have you found any relevant database knowledge you have learned in this course helpful?

The materials we've learned from this course greatly help us with the design and development process. Strictly following the instructions on the ER diagram and the rules for relational model translation, we had our functional dependencies in 3NF easily. With a well-organized database, we were able to come up with the clear logic flows of the desired functionalities and implement the corresponding Java methods efficiently.

Although Spring framework offered us a powerful data tool, Spring JPA, which allows us to use a large number of built-in database operations rather than write queries by ourselves, we cannot use it to query on more than one table. If we solely use Spring JPA's built-in functionalities, to implement some complex functionalities (especially for those involved joining two or more tables), we have to fetch the complete sets of tuples from each table separately and use if-conditions and getters to retrieve data we want. It brings two problems: 1) we would not be able to fully utilize database's own optimization, and 2) we need to manage some complicated functionality logic on the service level. This course has given us extensive practice on SQL query. After practicing, we feel confident that we can implement complex functionalities in long native SQL queries. By writing native queries inside the direct access objects (DAOs), we had better abstraction, and could let MySQL optimize our queries completely.

c. Have you encountered any database relevant issues that have been discussed in this course?

While testing the application in the backend, we noticed that there was some inconsistency in our records, tables weren't being updated as they should have. After giving it a closer look, we realised that our results were inconsistent because of the foreign key constraint. We had not taken into account the foreign key constraint in many places and fixed the bug as soon as it was found. We also tried our best to optimize each query and make sure that it runs in the expected time frame. We struggled with optimization in the beginning, but once we practiced the algorithms, over time it became easy. It was not easy to think of every possible test case and test the application, but we made up scenarios and tried to come up with ways that would bring out the vulnerabilities of the application, so that we could go ahead and fix them.

d. Future Goals

i. More functionalities, especially the social aspects. Now, following/unfollowing are only implemented on the playlists. For the future, we would like to migrate this functionality to matches as well, so that users can see how other music professionals match songs.
ii. Use Spring Security for user authentication. Currently, we authenticates users using JWT token and route interceptors because Spring Security hasn't been configured into our dependency file. We will integrate Spring Security to our project for better security.
iii. Deploy the app on Amazon Web Server (AWS). Our original plan was to host the website on aws. We've already had database set up on the server. However, it turns out that deploying a website on the amazon server is very different from hosting it on a local machine; there are many more configurations needed. We are continuing to work on those configurations to deploy our application in the near future.

IV. References

1. "Web API Tutorial." *Spotify for Developers*, developer.spotify.com/documentation/web-api/quick-start/.
2. "Get Audio Features for Several Tracks." *Spotify for Developers*, developer.spotify.com/documentation/web-api/reference/tracks/get-several-audio-features/.
3. "Connecting to Your Linux Instance Using SSH." *Amazon*, Amazon, docs.aws.amazon.com/AWSEC2/latest/UserGuide/AccessingInstancesLinux.html.
4. "Get Audio Features for a Track." *Spotify for Developers*, developer.spotify.com/documentation/web-api/reference/tracks/get-audio-features/.
5. "Java, Spring and Web Development Tutorials." *Baeldung*, www.baeldung.com/.