Cool Kids Playing with SQL (Group 16)

Raman Zatsarenko – rz4983@rit.edu
Austin Couch – alc9026@rit.edu
Srikar Sundaram – ss8931@rit.edu
Ricky Gupta – rg4825@rit.edu

November 23, 2021

1 Introduction

1.1 Team Information

1. **Team Name:** Cool Kids Playing with SQL.
2. **Team Number:** 16
3. **Members**
   (a) Raman Zatsarenko
   (b) Austin Couch
   (c) Srikar Sundaram
   (d) Ricky Gupta
4. **Domain:** Music

1.2 Description

The goal of this project is to create a music-focused database. Our user interface (view) will be a shell-based command line application written in Java. Project codebase will be version controlled using git and synchronized using GitHub and IntelliJ IDEA.

At any given time, we plan to have two front-end developers and two-back end developers working simultaneously. This report is collaborative effort between all four team members. In person meetings are conducted between all team members at least once per week for two hours, with more sessions added as needed. Soft deadlines are laid out ahead of course-defined deadlines and will correspond with in-person team meetings to review progress. In order to avoid last-minute brainstorming, any notable ideas will be recorded.

This is of course subject to change as progress is made through the project.

2 EER and Reduction to Tables

2.1 Decisions Made with EER Diagram

2.1.1 Entity Types

The first order of business when creating an EER diagram is to identify the entity types (ETs) needed. In our case, we decided to make ETs based on our knowledge of existing music streaming services. As such, the final set that we decided on was: User, Artist, Song, Album, Playlist and Genre.

For our first set of ETs, we considered "what are the human aspects of this database?" In accordance with project guidelines, we decided to make "User" and "Artist" ETs, as they represented the most basic kinds of accounts on the database. However, since they serve fundamentally different functions within any kind of music-related service (an artist creates music, whilst a user listens to music), we separated them into their own ETs.

Next, arguably the most crucial aspect of a music database are the songs stored within its system. Since every song has its own attributes distinct from both one another and other entities within the database,"Song" was decided as its own ET.

The following set of ETs that we deemed necessary were based around collections of music. Artists are the creators of songs, but also collections of songs, known as albums. Since this collection has differing attributes than than that of a singular song, we made this collection its own ET, an "Album". Furthermore, a playlist is nothing more than a collection of songs made by a user, instead of an artist. However, we determined that playlists serve an intrinsically different function to albums on top of having many differing attributes to albums. As such, "Playlist" became an ET.

Out final ET was Genre. Initially, we decided that both the ETs "Song" and "Album" would posses a multivalued attribute called "genre" that would represent the category that the given music would fall into. However, since multiple ETs have this attribute, we decided that it would be better to re-class "genre" the attribute to "Genre" the ET.

2.1.2 Attribute Types

Note: The symbols at the end of some attribute explanations are citing as to where in the project constraints this attribute is referenced, if at all.

I. **User**

a. Email: This attribute is a part of user and is unique due to project constraints. (§2 ¶1)

b. Creation Date: This attribute is a part of user due to project constraints. (§2 ¶1)

c. Last Access Date: This attribute is a part of user due to project constraints. (§2 ¶1)

d. Username: This attribute is a part of user and is unique due to project constraints. (§2 ¶1)

e. Password: This attribute is a part of user due to project constraints. (§2 ¶1)

f. Name: Is a composite attribute including a first and last name that do not need to be unique. We are limited by the constrains of this project to include first name an last name. It is our decision to make available a name composed by first and last name.

g. NumOfFollows: Indicates the number of users who follows this user. by this user. Inspired by music services, this is the only number the user is able to see.

II. **Artist**

a. Name: This attribute is a part of artist and unique due to project constraints. (§2 ¶2)

III. **Song**

a. Release Date: This attribute is a part of song due to project constraints. (§2 ¶2)

b. Title: This attribute is a part of song due to project constraints. (§2 ¶2)

c. Genre: This attribute is a part of song due to project constraints. (§2 ¶2). This is multivalued because if one album can have one song and multiple genres, and song must have multiple genres.

d. Length: This attribute is a part of song due to project constraints. (§2 ¶1)

e. ID: The necessity of this attribute comes from the unavailability of any other attribute and any combination of other attributed such that the combination is unique for the song. This is the primary key.

IV. **Album**

a. Name: This attribute is a part of album due to project constraints. (§2 ¶2)

b. Genre: This is a multivalued set of elements picked from a predefined set of categories that a music can belong to.

c. Release Date: This attribute is a part of album due to project constraints. (§2 ¶2)

d. ID: The necessity of this attribute comes from the unavailability of any other attribute and any combination of other attributed such that the combination is unique for the album. This is the primary key.

V. **Playlist**

a. Name: We are limited by project constraints to have a name for the collection of music. We call this collection, inspired by audio streaming services, a playlist. This playlist has an attribute name.

b. ID: The necessity of this attribute comes from the unavailability of any other attribute and any combination of other attributed such that the combination is unique for the playlist. This is the primary key.

VI. **Genre**

a. GenreName: A unique name identifying the genre, for example, rock, rap.

b. GenreID: A unique id used to refer to the the genre.

2.1.3 Relations and Cardinality

I. **Artist—Album**

Relation An artist can display their art using either a collection of songs. We are inspired by music streaming services, where artists often release a number of songs as an album. An album cannot exist without an artist. We believe it is a requirement for control of publication.

Cardinality An artist can have as few as zero albums, or many albums. This is a many to many relationship.

II. **Song—Album**

Relation In addition a song existing on its own, it may also exist via an album. When it exists this way, the song is related to an album.

Cardinality An song can be in as few as zero albums, or many albums. This is a many to many relationship.

III. **Artist—Song**

Relation An artist can display their art using either a collection of songs or a standalone song. This is the latter. We are inspired by music streaming services, where artists can release a single song. A song cannot exist without an artist. We believe it is a requirement for control of publication.

Cardinality An artist can have as few as zero songs, or many songs. This is a many to many relationship.

IV. **Song—Playlist**

Relation A song can exist in a user-generated collection of songs. Project constraints require us to maintain functionality of collection of songs.

Cardinality A song can belong to as few as zero playlist or many songs. Its cardinality is many to many.

V. **User—Playlist**

Relation A user is allowed to maintain collection of playlist, this is required by project constraints. A playlist is a way for the user to maintain songs.

Cardinality A user may have as few as zero or many playlists. Its cardinality is many to many.

VI. **Genre—Album**

Relation By project constraint, all albums must the ability to have genres. Having an independent relation for genres gives us the ability to quickly query all albums of a given genre.

Cardinality An genre may have as few as zero albums, or many. We believe that a genre can have zero album associated with it. For example, rock and jazz are two genres out of many. A database with one song might be associated with jazz, but rock is still a genre albeit unused. This is a many to many relation.

VII. **Genre—Song**

Relation By project constraint, all songs must the ability to have genres. Having an independent relation for genres gives us the ability to quickly query all songs of a given genre.

Cardinality An genre may have as few as zero songs, or many. We believe that a genre can have zero songs for the same reason discussed above. This is a many to many relation.

VIII. **User—User**

Note to reader: Its counterpart is not included like other relation in the subsection below because this relation is related to itself.

Relation By project constraints, a user must be able to follow another user. We need to have this as a relation because this is unique relation between two instances.

Cardinality A user may follow zero or many users. This is a many to many relation.

I. **Album—Artist**

Relation Album is a collection of songs but connected to artist or artists. Albums are a way for a group of songs made by the same group of artist. The album provides a way to discover similar songs in music services.

Cardinality An album must have at least one artist or many. This is a many to many relation.

II. **Album—Song**

Relation An album is a collection of songs connected by a common group of artists. The songs are related to this album as they are made up of them.

Cardinality An album must have one song or many songs. This is a many to many relationship. We decided to force a minimum of one song because an album must be able to added to a playlist via adding all its songs. If the minimum were zero, we think it would mean a playlist must be able to accommodate an album directly and not necessarily via songs.

III. **Song—Artist**

Relation The song can exist directly to the artist This is inspired by artists publishing single songs — rather than a group of them. However, all songs must be connected to an artist.

Cardinality A song is always connect to at least one artist, it may also be connected to many. This is a many to many relationship.

IV. **Playlist—Song**

Relation By project constraints, we must have a collection of songs. The playlist is related to these songs by the necessity of being able to play them.

Cardinality A playlist must not have a song, it may have zero or many songs. It's cardinality is many to many.

V. **Playlist—User**

Relation By project constraints a user must have the ability to have collection of songs. Multiple users may share a playlist.

Cardinality The playlist is connected to at least one user and may have many users. This is a many to many relation.

VI. **Album—Genre**

Relation By the necessity of project constraints, we needed to accommodate the connection between genres and albums. This relation helps us in connecting multiple albums to the same limited genres.

Cardinality At a minimum, an album must have at least one genre. It must also be able to have multiple because its genres must be at least the union of all its songs' genre, and this is true by project constraints. This is a many to many relation.

VII. **Song—Genre**

Relation By the necessity of project constraints, we needed to accommodate the connection between genre and song. This relation helps us in connecting multiple albums to the same limited genres.

Cardinality At a minimum, a song must have at least one genre. It must also be able to have multiple because a single-song album with multiple genre should logically follow a song with multiple genre. This is a many to many relation.

2.2 Decisions made with Reduction to Tables

2.2.1 Entity Types and Primary Keys

When looking to translate our EER diagram into relational tables, we began with the six ETs of User, Artist, Song, Album, Playlist, and Genre as described in section

2.1.1. These were decided as the most basic building blocks we could use to construct our database. Each of these ETs carried its own primary key that was aptly named via prefix. For example, the primary key of the SONG() is its songID whereas the PLAYLIST() ET's primary key is playlistID. The naming convention decisions were made in this Reduction to Tables process upon realizing the confusion between each ET's name or ID value that represented their primary key and how this would make the reductions to tables incredibly difficult to perform. This would have become increasingly difficult as we went on to form the relationship tables.

2.2.2 Multi-valued Attributes into Entity Types

Attribute Types lead us to our next hurdle in multi-valued attributes. The genre attribute of a SONG() or ALBUM() ET needed to be multi-valued in order to account for songs that expressed more than one genre, as is often the case in the real-world music databases we sought to emulate. We later realized that this would be a misuse of a multi-valued attribute as this genre is exhibited by multiple ETs. Creating a GENRE() ET itself proved more effective as we could now simply create separate relation tables for SONGGNR() and ALBUMGNR() between GENRE() and the respective ETs. Our reduction to tables process was bound by the standard relational domain constraint requiring that each attribute type must be atomic and single-valued. In order to abide by these rules, we created the 2 new relationship tables previously mentioned that would each hold a now atomic and single-valued foreign key to its genreID and a foreign key back to their respective parent's songID or albumID ET.

2.2.3 Relationships and Foreign Keys

Our last steps were in laying out the remaining relationship tables. These tables named after the action that they perform were CREATE(), COMPOSE(), COMPOSEDOF(), CONTAINS(), MAKE(), and FOLLOWSUSER(). Each of these relational tables held first the foreign key from the account performing the action, whether that be an ARTIST() or a USER(). Next would be the foreign key of the ET they are acting on, for example, a USER() could CREATE() a PLAYLIST(), and this CREATE() would then contain the PLAYLIST()'s primary key as this foreign key of this ET being acted on. Additionally, the music collection creation relationship tables of an ALBUM()'s CREATE() or a PLAYLIST()'s MAKE() would lastly contain the respective index value that would be needed in the implementation of actual collection formations aptly named as albumSongIndex and playlistSongIndex.

3 Application Implementation

3.1 Updates made to the EER and Reduction to Tables

Based on the feedback we received, a number of updates was made to the EER diagram and the reduction to tables. Because a Genre attribute type exists for both the Album ET and the Song ET, it was decided to create a separate ET named Genre that would

have its own attribute types (GenreName and GenreID) and would relate to the Song ET and the Album ET. As a result, a new table was added to the reduction of tables.

Another update was the addition of a PLAYS() relation between the user and the song, which would allow the user to play any song they wish. We also added functionality to support the ability to play a whole album or a playlist. The EER and the reduction to tables were updated accordingly.

3.2 Database design, Uploading Data and Populating Tables

3.2.1 Creating tables

Tables for the database were created based on the reduction to tables with all the necessary updates to fully reflect the state of our application's data. The following SQL statements were used to create the tables:

CREATE TABLE "User" (email text PRIMARY KEY, username text NOT NULL UNIQUE, password text NOT NULL, "creationDate" text NOT NULL, "lastAccessDate" text NOT NULL, "firstName" text NOT NULL, "lastName" text NOT NULL, "userNumFollowers" int);

CREATE TABLE "Playlist" ("playlistID" text PRIMARY KEY, "playlistName" text NOT NULL, "runtime" int NOT NULL DEFAULT 0);

CREATE TABLE "AlbumGNR" ("genreID" text, "albumID" text);

CREATE TABLE "Contains" ("playlistSongIndex" int);

CREATE TABLE "AlbumGNR" ("genreID" text, "albumID" text);

CREATE TABLE "Create" ("artistName" text, "albumID" text);

CREATE TABLE "FollowsUser" ("followerEmail" text, "followeeEmail" text);

We then used the software DataGrip to add foreign key relationships between tables where ever two tables were related as seen in the reduction table.

3.2.2 Populating the data

Some data, like the users with their credentials, followers lists, and playlists, was randomly generated and populated using SQL procedures and queries:

Adding random followers for users

```
CREATE PROCEDURE addFollower ("emailFollower" text,
"emailFollowee" text)
LANGUAGE plpgsql
AS $$ BEGIN
    INSERT INTO "FollowsUser" ("followerEmail", "followeeEmail")
    VALUES ("emailFollower", "emailFollowee");
    UPDATE "User" SET "userNumFollowers" = "userNumFollowers"
```

```
    + 1 WHERE "email" = "emailFollowee";
END;$$;

CREATE OR REPLACE PROCEDURE add_random_follower ()
LANGUAGE plpgsql
AS $$
DECLARE randEmail1 text;
DECLARE randEmail2 text;
BEGIN
    SELECT "email" FROM "User" OFFSET floor(random() * 1000)
    LIMIT 1 INTO randEmail1;
    SELECT "email" FROM "User" OFFSET floor(random() * 1000)
    LIMIT 1 INTO randEmail2;
    IF randEmail1 != randEmail2 THEN
        IF NOT EXISTS(SELECT * FROM "FollowsUser"
        WHERE "followerEmail" = randEmail1 AND
        "followeeEmail" = randEmail2) THEN
            CALL add_follower(randEmail1, randEmail2);
        end if;
    END IF;
END;$$;

CREATE OR REPLACE PROCEDURE add_multiple_rand_followers (num int)
LANGUAGE plpgsql
AS $$
BEGIN
    FOR i IN 1..num LOOP
    CALL add_random_follower();
    end loop;
END;$$;
CALL add_multiple_rand_followers(500);
CALL add_multiple_rand_followers(3000);
CALL add_multiple_rand_followers(6000);
CALL add_multiple_rand_followers(40000);
CALL add_multiple_rand_followers(10000);
```

Assigning random playlist creators Same methods were used to create

```
CREATE OR REPLACE PROCEDURE assign_random_playlist_creators()
LANGUAGE plpgsql
AS $$
DECLARE randEmail text;
DECLARE make_row RECORD;
BEGIN
    FOR make_row IN SELECT * FROM "Make" LOOP
```

```
            SELECT "email" FROM "User" OFFSET floor(random() * 1000)
            LIMIT 1 INTO randEmail;
            UPDATE "Make" SET email = randEmail
            WHERE make_row."playlistID" = "playlistID";
        END LOOP;
    END;$$;
    CALL assign_random_playlist_creators();
```

Adding random songs to playlists

```
    CREATE OR REPLACE PROCEDURE add_songs_to_playlists()
    LANGUAGE plpgsql
    AS $$
    DECLARE sID text;
    DECLARE pID text;
    DECLARE playlist_row RECORD;
    BEGIN
        FOR playlist_row IN SELECT * FROM "Playlist" LOOP
            SELECT "playlistID" FROM "Playlist"
            WHERE playlist_row."playlistID" = "playlistID" INTO pID;
            FOR i IN 0..floor(random() * 30) LOOP
                SELECT "songID" FROM "Song" OFFSET floor(random() *
                (SELECT count(*) FROM "Song")) LIMIT 1 INTO sID;
                CALL add_song(sID, pID);
            END LOOP;
        END LOOP;
    END;$$;
    CALL add_songs_to_playlists();
```

3.2.3 Loading the Data into the Database

The initial data file we decided to use was a csv file found on kaggle.com. It contained some information about songs, their authors, release date, but the data had to be formatted properly. All of the data from the csv was formatted and loaded into a table named **ImportedMusicData**. ImportedMusicData table was only used to properly load the data into the actual tables. As an example of what kind of formatting was performed with ImportedMusicData, we provide a code snippet used to trim the artists' names:

```
UPDATE "ImportedMusicData" SET artists = right(artists, -2);
UPDATE "ImportedMusicData" SET artists = left(artists, -2);
SELECT '''' || "ImportedMusicData".artists
    as artists FROM "ImportedMusicData";
SELECT "ImportedMusicData".artists
    || '''' as artists FROM "ImportedMusicData";
UPDATE "ImportedMusicData" SET artists = ''''
```

```
    || "ImportedMusicData".artists;
UPDATE "ImportedMusicData"
    SET artists = "ImportedMusicData".artists || '''';
```

Finally, we provide a sample of SQL insert statements used to insert data into the tables:

```
INSERT INTO "Compose" ("artistName", "songID")
SELECT "artistName", "songID"
FROM "ImportedMusicData";

INSERT INTO "Create" ("artistName", "albumID")
SELECT DISTINCT "artistName", "albumID"
FROM "ImportedMusicData";

INSERT INTO "ComposedOf" ("albumID", "songID", "albumSongIndex")
SELECT "albumID", "songID", "trackIndex"
FROM "ImportedMusicData";

SELECT DISTINCT * INTO "Album" FROM (SELECT "albumID",
"albumName", "albumReleaseDate" FROM "ImportedMusicData") as a;

SELECT DISTINCT * INTO "Genre" FROM (SELECT "genreName"
FROM "ImportedMusicData") as g;

SELECT DISTINCT * INTO "Song" FROM (SELECT "songID", length,
title, "albumReleaseDate" FROM "ImportedMusicData") as s;
```

4 Analysis and Recommendations

4.1 Analysis

4.1.1 Preprocessing

The user related data was generated using a stored procedure which randomly associated a user with a song to play. Our original dataset provided the following data for each song. In bold are the data we select for analysis.

1. **id**
2. name
3. album
4. album id
5. artists
6. artist ids
7. track number
8. disc number
9. **explicit**
10. **danceability**
11. **energy**
12. **key**
13. **loudness**
14. **mode**
15. **speechiness**

16. **acousticness**
17. **instrumentalness**
18. **liveness**
19. **valence**
20. **tempo**
21. **duration**
22. time signature
23. **year**
24. release date

We find outliers in the years. We treat them as missing values and the row is not considered in of any of the next steps.

To transform the data as a step to reduce noise, we analyze the data by by look at the year. We look at the average of that data for that year. The R programming language was used for this, along with packages provided in the Tidyverse collection.

```
summary <- (data %>%
filter(year > 0) %>%
group_by(year) %>%
summarise(
    loudness = mean(loudness),
    energy = mean(energy),
    danceability = mean(danceability),
    speechiness = mean(speechiness),
    acousticness = mean(acousticness),
    instrumentalness = mean(instrumentalness),
    liveness = mean(liveness),
    valence = mean(valence),
    tempo = mean(tempo),
    duration = mean(duration_ms) / 60000,
    explicit = mean(explicit),
    n = n() # num of songs for that year.
  ));
```

Since `R` is an interpreted language, we used Jupyter Notebooks along with the the R kernel. Analysis was performed on this data using time series and correlation plots. We also used the cluster and factoextra packages to perform k-means clustering and plotting them. The summarized data was moved to a csv file in order to analyze it in Minitab.

Minitab was used to discretize and perform decision tree-like tree for predicting the data based on the year.

4.1.2 Modeling the data

Using the correlation plot, we investigated the relationships between the features of our data. It was evident that energy and acousticness were very strongly inversely related to each other, and it was therefore decided to eliminate energy from the final model to avoid multicollinearity. We built various linear regression models using least squares method and analyzed them using the R squared adjusted and the Mallow's Cp score. P-values of the parameters were also investigated to verify the statistical significance.

We finally found a model that produced good results and fit the data well. To verify that a linear model indeed fits the data properly, a residuals analysis process was also

conducted. Based on the results of the residuals analysis, we concluded that the linear model we used indeed fits the data. Below is the equation we found:

$$year = 2384.99 + 4.58x_{i1} - 123.06x_{i2} + 70.74x_{i3} - 1.86x_{i4} - 7.18x_{i5} - 96.46x_{i6}, \quad (1)$$

Here $4.58 = \beta_1 = loudness$,
$-123.06 = \beta_2 = acousticness$,
$70.74 = \beta_3 = instrumentalness$,
$-1.86 = \beta_4 = tempo$,
$-7.18 = \beta_5 = duration$,
$-96.46 = \beta_6 = valence$.

4.2 SQL Statements

```
-- 1. My playlists (Count)
select count(*)
from "Make"
where email = '__EMAIL__';
-- 2. My followings (Count)
select count(*)
from "FollowsUser"
where "followerEmail" = '__EMAIL__';
-- 3.  My followers (Count)
select count(*)
from "FollowsUser"
where "followeeEmail" = '__EMAIL__';
--4. My top aritsts (collections & played)
select topArts."artistName"
from (
        (
            select "artistName"
            from "Artist"
                natural join "Compose"
                natural join "Plays"
            where email = '__EMAIL__'
        )
        union all
        ( top_artists_collections
            select "artistName"
            from "Artist"
                natural join "Compose"
                natural join "Make"
                natural join "Contains"
            where email = '__EMAIL__'
        )
    ) as topArts
```

```
group by topArts."artistName"
order by count(topArts."artistName") desc;
--5. Top 50 songs of the month.
select "songID",
    length,
    title,
    "songReleaseDate"
from "Plays"
    natural join "Song"
where email = '__EMAIL__'
    and "datePlayed" > 'year-month-day'
group by "songID",
    length,
    title,
    "songReleaseDate"
order by count("songID") desc
limit 50;
--6. Top 50 Songs among my followers.
select "songID",
    length,
    title,
    "songReleaseDate"
from "Plays"
    natural join "Song"
where email in (
        select "followeeEmail"
        from "FollowsUser"
        where "followerEmail" = '__EMAIL__'
    )
group by "songID",
    length,
    title,
    "songReleaseDate"
order by count("songID") desc
limit 50;
--7. Top 5 Genres of the month.
select "genreID",
    "genreName"
from "Plays"
    natural join "Song"
    natural join "SongGNR"
    natural join "Genre"
where "datePlayed" > 'year-month-01'
group by "genreID",
    "genreName"
order by count("genreID") desc
```

```
limit 5;
--8 Similar songs based on your Genre.
select "Song".*
from "Song"
    natural join "SongGNR"
where "genreID" in (
        select "genreID"
        from "Plays"
            natural join "Song"
            natural join "SongGNR"
            natural join "Genre"
        where email = '__EMAIL__'
        group by "genreID",
            "genreName"
        order by count("genreID") desc
    )
limit 20;
```