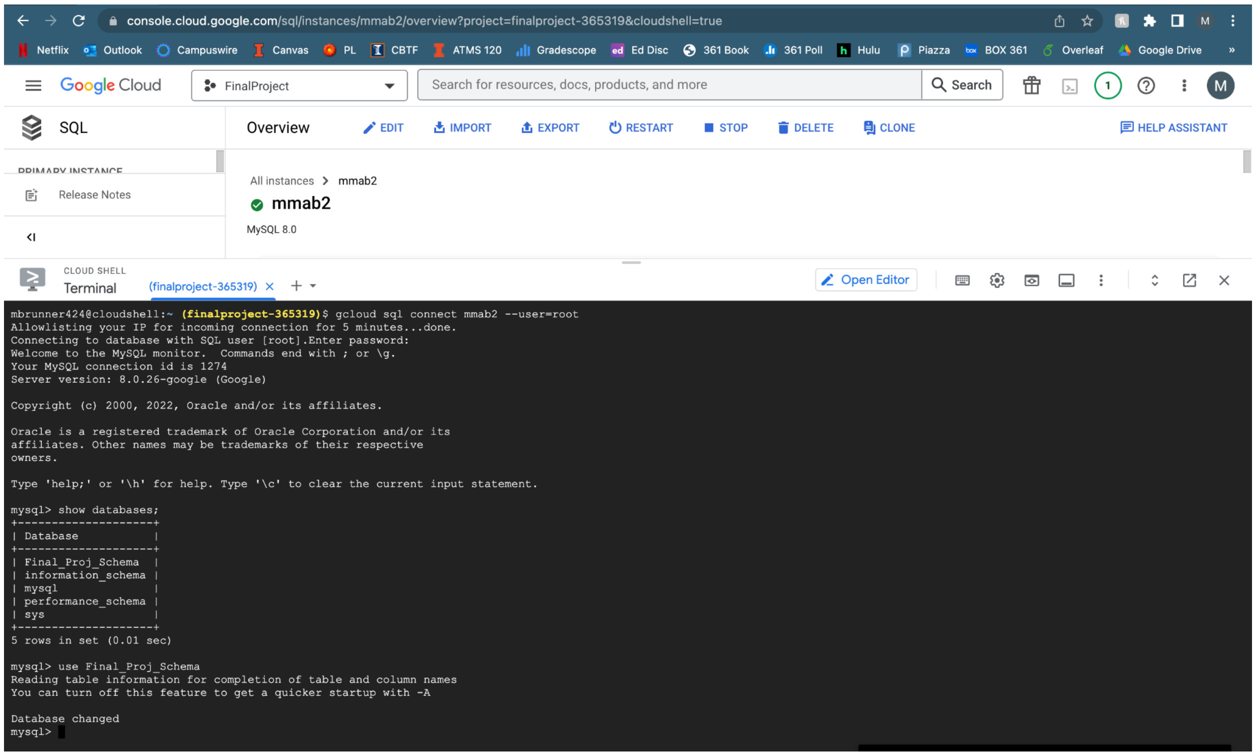
Database Implementation:

1. Here is a screenshot of our database tables implemented on GCP



```
mbrunner424@cloudshell:~ (finalproject-365319)$ gcloud sql connect mmab2 --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1274
Server version: 8.0.26-google (Google)

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| Final_Proj_Schema  |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.01 sec)

mysql> use Final_Proj_Schema
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> 
```

2. Here are the DDL commands for our tables

```
CREATE TABLE `artists` (

 `artistId` varchar(45) NOT NULL,

 `name` varchar(45) DEFAULT NULL,

 PRIMARY KEY (`artistId`)

);

CREATE TABLE `events` (

 `eventId` int NOT NULL,

 `name` varchar(45) DEFAULT NULL,

 `keywords` varchar(45) DEFAULT NULL,

 PRIMARY KEY (`eventId`)

);
```

```
CREATE TABLE `friends` (
  `friendId` int NOT NULL,
  `name` varchar(45) DEFAULT NULL,
  `phone_number` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`friendId`)
);

CREATE TABLE `our_songs` (
  `trackId` varchar(45) NOT NULL,
  `trackTitle` varchar(45) DEFAULT NULL,
  `albumId` varchar(45) DEFAULT NULL,
  `duration` int DEFAULT NULL,
  `danceability` double DEFAULT NULL,
  `loudness` double DEFAULT NULL,
  `energy` double DEFAULT NULL,
  `valence` double DEFAULT NULL,
  `vibe_score` double DEFAULT NULL,
  `artist_Id` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`trackId`),
  KEY `vibe_score_idx` (`vibe_score`),
  KEY `artist_Id_idx` (`artist_Id`),
  CONSTRAINT `artist_Id` FOREIGN KEY (`artist_Id`) REFERENCES `artists`
(`artistId`) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `vibe_score` FOREIGN KEY (`vibe_score`) REFERENCES `vibe`
(`vibe_score`) ON DELETE CASCADE ON UPDATE CASCADE
);
```

```
CREATE TABLE `playlist` (
  `playlistId` int NOT NULL,
  `username` varchar(45) NOT NULL,
  `duration` int DEFAULT NULL,
  `eventId` int DEFAULT NULL,
  PRIMARY KEY (`playlistId`),
  KEY `userId_idx` (`username`),
  KEY `eventId_idx` (`eventId`),
  CONSTRAINT `username` FOREIGN KEY (`username`) REFERENCES `users`
(`username`)
)

CREATE TABLE `spotify_charts` (
  `rank` int DEFAULT NULL,
  `region` varchar(45) NOT NULL,
  `trackId` varchar(45) NOT NULL,
  `streams` varchar(45) DEFAULT NULL,
  `chartId` int NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`chartId`)
);

CREATE TABLE `users` (
  `username` varchar(45) NOT NULL,
  `password` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`username`)
);

CREATE TABLE `vibe` (
  `name` varchar(45) DEFAULT NULL,
  `vibe_score` double NOT NULL,
  PRIMARY KEY (`vibe_score`)
);
```

These are the DDL commands for the many to many relationships we have:

```
CREATE TABLE `playlist_artist` (
  `playlistId` int NOT NULL,
  `artistId` varchar(45) NOT NULL,
  PRIMARY KEY (`artistId`,`playlistId`),
  KEY `playlistId_idx` (`playlistId`),
  CONSTRAINT `artistId` FOREIGN KEY (`artistId`) REFERENCES `artists` (`artistId`),
  CONSTRAINT `playlistId` FOREIGN KEY (`playlistId`) REFERENCES `playlist`
(`playlistId`)
)

CREATE TABLE `playlist_song` (
  `playlistId` int NOT NULL,
  `trackId` varchar(45) NOT NULL,
  PRIMARY KEY (`playlistId`,`trackId`),
  KEY `trackId_idx` (`trackId`),
```

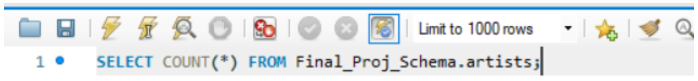
CONSTRAINT `playlist_song_playlistId` FOREIGN KEY (`playlistId`) REFERENCES `playlist` (`playlistId`),

CONSTRAINT `playlist_song_trackId` FOREIGN KEY (`trackId`) REFERENCES `our_songs` (`trackId`)

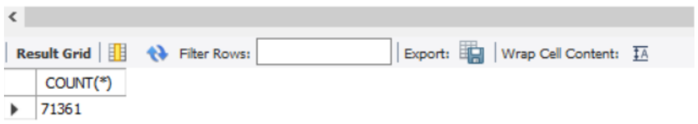
)

3. Here are count queries we used to show that we have at least 1000 rows in the ‘artists’, ‘our_songs’, and ‘spotify_charts’ tables and a screenshot.

Artist table:

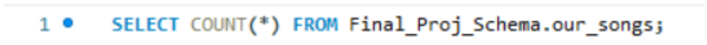


```
SELECT COUNT(*) FROM Final_Proj_Schema.artists;
```

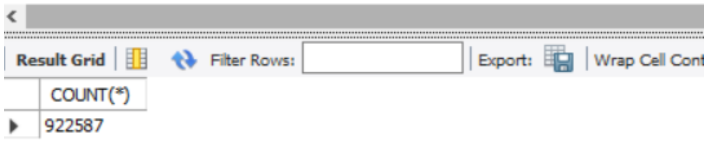


```
COUNT(*)
71361
```

Our_ songs table:

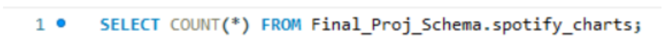


```
SELECT COUNT(*) FROM Final_Proj_Schema.our_songs;
```

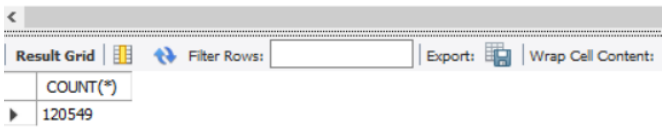


```
COUNT(*)
922587
```

Spotify_charts table:



```
SELECT COUNT(*) FROM Final_Proj_Schema.spotify_charts;
```



```
COUNT(*)
120549
```

Advanced Queries:

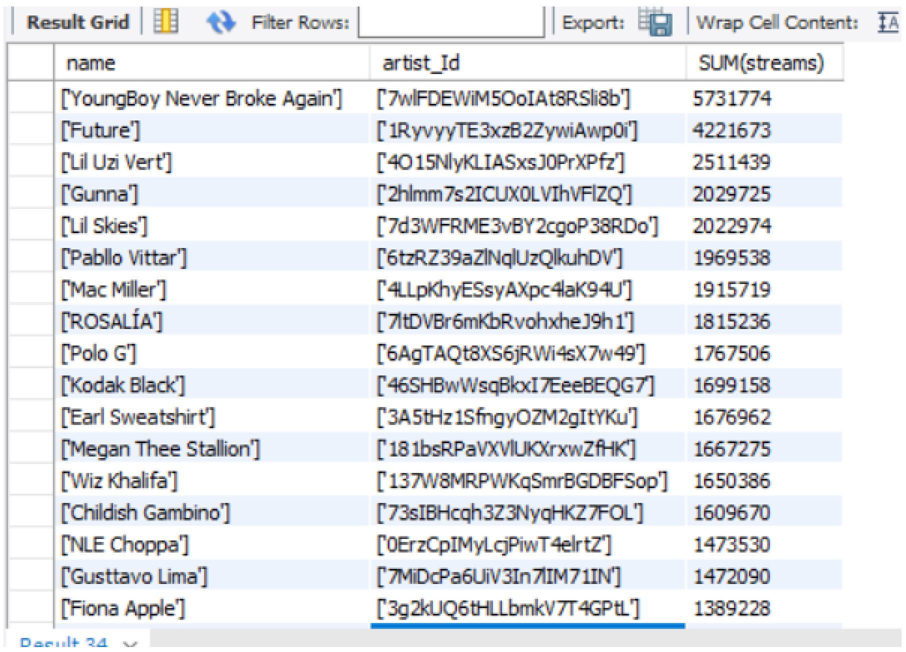
1) Developing two advanced queries

First query

A large part of our project will include trying to figure out what songs to add to a playlist based on certain user preference. If a user inputs many artists, we will not be able to add every artist's songs, and in general, even if a user does not input specific artists, we will want to add songs of artists who are more popular. So, we created the following query that outputs the number of streams that each artist has.

```
SELECT artists.name, our_songs.artist_Id , SUM(streams)

FROM Final_Proj_Schema.artists

INNER JOIN Final_Proj_Schema.our_songs

ON artists.artistId = our_songs.artist_Id

INNER JOIN Final_Proj_Schema.spotify_charts

ON  our_songs.trackId = spotify_charts.trackId

GROUP BY artists.name, our_songs.artist_Id

ORDER BY SUM(streams) DESC, artists.name ASC
```

Our query outputs the following information:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

name	artist_Id	SUM(streams)
['YoungBoy Never Broke Again']	['7wlFDEWiM5OoIAt8RSli8b']	5731774
['Future']	['1RyvyyTE3xzB2ZywiAwp0i']	4221673
['Lil Uzi Vert']	['4O15NlyKLIASxsJ0PrXPfz']	2511439
['Gunna']	['2hlmm7s2ICUX0LVIhVFlZQ']	2029725
['Lil Skies']	['7d3WFRME3vBY2cgoP38RDo']	2022974
['Pabllo Vittar']	['6tzRZ39aZlNqlUzQlkuhDV']	1969538
['Mac Miller']	['4LLpKhyESsyAXpc4laK94U']	1915719
['ROSALÍA']	['7ltDVBr6mKbRvohxheJ9h1']	1815236
['Polo G']	['6AgTAQt8XS6jRWi4sX7w49']	1767506
['Kodak Black']	['46SHBwWsqBkxI7EeeBEQG7']	1699158
['Earl Sweatshirt']	['3A5tHz1SfngyOZM2gItYKu']	1676962
['Megan Thee Stallion']	['181bsRPaVXVlUKXrxwZfHK']	1667275
['Wiz Khalifa']	['137W8MRPWKqSmrBGDBFSop']	1650386
['Childish Gambino']	['73sIBHcqh3Z3NyqHKZ7FOL']	1609670
['NLE Choppa']	['0ErzCpIMyLcjPiwT4elrtZ']	1473530
['Gusttavo Lima']	['7MiDcPa6UiV3In7lIM71IN']	1472090
['Fiona Apple']	['3g2kUQ6tHLLbmkV7T4GPtL']	1389228

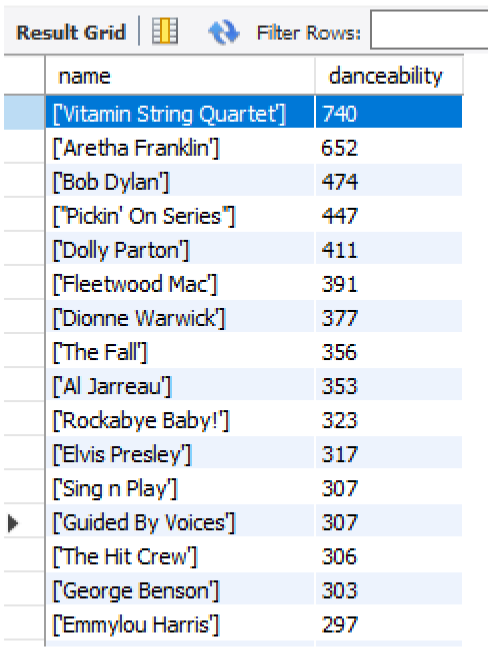
Result 34

Second query

The second query returns the top 25 artists based on the sum of all their songs' danceability scores. This query will be helpful in our project because it will help us see which artists have a higher danceability to know which artists we should add to a more upbeat/dance playlist.

```
SELECT artists.name, ROUND(SUM(our_songs.danceability)) AS danceability

FROM Final_Proj_Schema.artists

INNER JOIN Final_Proj_Schema.our_songs

ON artists.artistId = our_songs.artist_Id

GROUP BY artists.name

ORDER BY 2 DESC

LIMIT 25
```

This query returns the top 25 artists based on the sum of all their song's danceability. The results are shown below:



name	danceability
['Vitamin String Quartet']	740
['Aretha Franklin']	652
['Bob Dylan']	474
["Pickin' On Series"]	447
['Dolly Parton']	411
['Fleetwood Mac']	391
['Dionne Warwick']	377
['The Fall']	356
['Al Jarreau']	353
['Rockabye Baby!']	323
['Elvis Presley']	317
['Sing n Play']	307
['Guided By Voices']	307
['The Hit Crew']	306
['George Benson']	303
['Emmylou Harris']	297

Indexing:

First Query

We start with the following results from EXPLAIN ANALYZE:

```
# EXPLAIN

-> Sort: `SUM(streams)` DESC, Final_Proj_Schema.artists.`name`  (actual
time=592.614..592.749 rows=1606 loops=1)

    -> Table scan on <temporary>  (actual time=0.002..0.232 rows=1606 loops=1)

        -> Aggregate using temporary table  (actual time=591.663..591.984 rows=1606
loops=1)

            -> Nested loop inner join  (cost=96428.99 rows=120354) (actual
time=0.141..582.074 rows=4534 loops=1)

                -> Nested loop inner join  (cost=54305.09 rows=120354) (actual
time=0.133..570.881 rows=4534 loops=1)

                    -> Table scan on spotify_charts  (cost=12171.65 rows=120354) (actual
time=0.045..33.738 rows=120549 loops=1)

                    -> Filter: (Final_Proj_Schema.our_songs.artist_Id is not null)  (cost=0.25
rows=1) (actual time=0.004..0.004 rows=0 loops=120549)

                        -> Single-row index lookup on our_songs using PRIMARY
(trackId=Final_Proj_Schema.spotify_charts.trackId)  (cost=0.25 rows=1) (actual
time=0.004..0.004 rows=0 loops=120549)

                -> Single-row index lookup on artists using PRIMARY
(artistId=Final_Proj_Schema.our_songs.artist_Id)  (cost=0.25 rows=1) (actual
time=0.002..0.002 rows=1 loops=4534)
```

We see that the most costs occur when we perform both of the INNER JOIN commands and when we scan spotify charts for trackId. We can first attempt to index spotify_charts.trackId to organize the songs:

```
CREATE INDEX spotify_charts_trackId

ON Final_Proj_Schema.spotify_charts (trackId);
```

# EXPLAIN

```
-> Sort: `SUM(streams)` DESC, Final_Proj_Schema.artists.`name`  (actual time=580.350..580.491 rows=1606 loops=1)

    -> Table scan on <temporary>  (actual time=0.001..0.265 rows=1606 loops=1)

        -> Aggregate using temporary table  (actual time=579.333..579.695 rows=1606 loops=1)

            -> Nested loop inner join  (cost=96428.99 rows=120354) (actual time=0.158..570.575 rows=4534 loops=1)

                -> Nested loop inner join  (cost=54305.09 rows=120354) (actual time=0.149..559.849 rows=4534 loops=1)

                    -> Table scan on spotify_charts  (cost=12171.65 rows=120354) (actual time=0.042..32.949 rows=120549 loops=1)

                    -> Filter: (Final_Proj_Schema.our_songs.artist_Id is not null)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=0 loops=120549)

                        -> Single-row index lookup on our_songs using PRIMARY (trackId=Final_Proj_Schema.spotify_charts.trackId)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=0 loops=120549)

                -> Single-row index lookup on artists using PRIMARY (artistId=Final_Proj_Schema.our_songs.artist_Id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=4534)
```

We see that all the costs stay exactly the same. This is likely because spotify_charts.trackID is a primary key, so it is automatically indexed when it is read into MySQL. Indexing spotify_charts.trackID had no effect on performance. Let's see if the same applies for artists.artistId, which is also a primary key:

```
CREATE INDEX artists_artistId

ON Final_Proj_Schema.artists (artistId);
```

# EXPLAIN

```
-> Sort: `SUM(streams)` DESC, Final_Proj_Schema.artists.`name`  (actual time=590.601..590.736 rows=1606 loops=1)

    -> Table scan on <temporary>  (actual time=0.001..0.250 rows=1606 loops=1)
```

```
        -> Aggregate using temporary table  (actual time=589.634..589.972 rows=1606
loops=1)

            -> Nested loop inner join  (cost=96428.99 rows=120354) (actual
time=0.138..579.983 rows=4534 loops=1)

                -> Nested loop inner join  (cost=54305.09 rows=120354) (actual
time=0.131..569.082 rows=4534 loops=1)

                    -> Table scan on spotify_charts  (cost=12171.65 rows=120354) (actual
time=0.037..33.616 rows=120549 loops=1)

                    -> Filter: (Final_Proj_Schema.our_songs.artist_Id is not null)  (cost=0.25
rows=1) (actual time=0.004..0.004 rows=0 loops=120549)

                        -> Single-row index lookup on our_songs using PRIMARY
(trackId=Final_Proj_Schema.spotify_charts.trackId)  (cost=0.25 rows=1) (actual
time=0.004..0.004 rows=0 loops=120549)

                -> Single-row index lookup on artists using PRIMARY
(artistId=Final_Proj_Schema.our_songs.artist_Id)  (cost=0.25 rows=1) (actual
time=0.002..0.002 rows=1 loops=4534)
```

We can see that the same applies to indexing on artists.artistID; it does not improve performance because the attribute has already been indexed. Now, let's try indexing our_songs.artist_Id since it is not a primary key; this could potentially help the performance of INNER JOIN and GROUP BY:

```
CREATE INDEX our_songs_ind

ON Final_Proj_Schema.our_songs (artist_Id, trackId);

# EXPLAIN

-> Sort: `SUM(streams)` DESC, Final_Proj_Schema.artists.`name`  (actual
time=588.914..589.050 rows=1606 loops=1)

    -> Table scan on <temporary>  (actual time=0.002..0.290 rows=1606 loops=1)

        -> Aggregate using temporary table  (actual time=587.639..588.018 rows=1606
loops=1)

            -> Nested loop inner join  (cost=96428.99 rows=120354) (actual
time=0.155..578.438 rows=4534 loops=1)

                -> Nested loop inner join  (cost=54305.09 rows=120354) (actual
time=0.147..567.455 rows=4534 loops=1)
```

```
                -> Table scan on spotify_charts  (cost=12171.65 rows=120354) (actual time=0.046..34.130 rows=120549 loops=1)

                -> Filter: (Final_Proj_Schema.our_songs.artist_Id is not null)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=0 loops=120549)

                    -> Single-row index lookup on our_songs using PRIMARY (trackId=Final_Proj_Schema.spotify_charts.trackId)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=0 loops=120549)

            -> Single-row index lookup on artists using PRIMARY (artistId=Final_Proj_Schema.our_songs.artist_Id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=4534)
```

Again, we find no difference in performance even though our_songs.artist_Id is not a primary key. We find the reason to be most likely that artists.artistId takes longer to execute than our_songs.artist_Id and we already know artists.artistId can't be further optimized since it is a primary key. So when we execute INNER JOIN on these two attributes, it will take the same amount of time either way. Since our_songs.trackId is also a primary key, it will not improve performance. Therefore, our program is already performing at an optimum.

Second query:

The following is our starting EXPLAIN ANALYZE output for the second query:

```
# EXPLAIN
-> Limit: 25 row(s)  (actual time=3415.232..3415.235 rows=25 loops=1)
    -> Sort: round(sum(Final_Proj_Schema.our_songs.danceability),0) DESC, limit input to 25 row(s) per chunk  (actual time=3415.231..3415.232 rows=25 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..11.409 rows=70259 loops=1)
            -> Aggregate using temporary table  (actual time=3391.996..3407.394 rows=70259 loops=1)
                -> Nested loop inner join  (cost=317602.95 rows=885524) (actual time=0.083..2778.302 rows=922587 loops=1)
                    -> Table scan on artists  (cost=7599.35 rows=74626) (actual time=0.051..26.853 rows=71361 loops=1)
                    -> Index lookup on our_songs using our_songs_ind (artist_Id=Final_Proj_Schema.artists.artistId)  (cost=2.97 rows=12) (actual time=0.020..0.038 rows=13 loops=71361)
```

We found that significant costs were caused by the GROUP BY aggregation function, the INNER JOIN statement, and the table scan on artists. We thought a good place to start indexing would be by ordering artistId and name under the artists table since artistId is an attribute used in the INNER JOIN and name is the characteristic used in the GROUP BY function. Indexing these attributes could potentially cut down costs, so we did the following:

```
CREATE INDEX artists_artistId
ON Final_Proj_Schema.artists (artistId, name);
```

And got this result:

```
# EXPLAIN
-> Limit: 25 row(s)  (actual time=3724.260..3724.263 rows=25 loops=1)
    -> Sort: round(sum(Final_Proj_Schema.our_songs.danceability),0) DESC, limit input
to 25 row(s) per chunk  (actual time=3724.259..3724.261 rows=25 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..11.827 rows=70259 loops=1)
            -> Aggregate using temporary table  (actual time=3700.531..3716.324
rows=70259 loops=1)
                -> Nested loop inner join  (cost=317602.95 rows=885524) (actual
time=26.271..3081.662 rows=922587 loops=1)
                    -> Index scan on artists using artists_artistId  (cost=7599.35 rows=74626)
(actual time=26.207..386.572 rows=71361 loops=1)
                    -> Index lookup on our_songs using our_songs_ind
(artist_Id=Final_Proj_Schema.artists.artistId)  (cost=2.97 rows=12) (actual
time=0.020..0.037 rows=13 loops=71361)
```

The performance of the query was actually worse. We realized this may be due to the fact that we are both indexing a primary key(which already has an assigned index) and an additional attribute which might add a lot of random and unnecessary indices to the graph. We decide to only index name since it isn't a primary key:

```
CREATE INDEX artists_artistId

ON Final_Proj_Schema.artists (name);

# EXPLAIN

-> Limit: 25 row(s)  (actual time=3255.544..3255.547 rows=25 loops=1)

    -> Sort: round(sum(Final_Proj_Schema.our_songs.danceability),0) DESC, limit input
to 25 row(s) per chunk  (actual time=3255.542..3255.544 rows=25 loops=1)

        -> Stream results  (cost=406155.35 rows=885524) (actual time=0.921..3240.277
rows=70259 loops=1)
```

```
            -> Group aggregate: sum(Final_Proj_Schema.our_songs.danceability)
(cost=406155.35 rows=885524) (actual time=0.917..3218.410 rows=70259 loops=1)

                -> Nested loop inner join  (cost=317602.95 rows=885524) (actual
time=0.851..2963.965 rows=922587 loops=1)

                    -> Index scan on artists using artists_artistId  (cost=7599.35 rows=74626)
(actual time=0.819..249.320 rows=71361 loops=1)

                    -> Index lookup on our_songs using our_songs_ind
(artist_Id=Final_Proj_Schema.artists.artistId)  (cost=2.97 rows=12) (actual
time=0.021..0.037 rows=13 loops=71361)
```

This improved the original runtime slightly as we were able to index the name attribute separate from the primary key of the artists table, so the program executed in a more organized fashion.

Next, we decided to try indexing our_songs.artist_Id since it is also part of the INNER JOIN function where significant costs are used to see if it has any effect on performance:

```
CREATE INDEX our_songs_artist_Id

ON Final_Proj_Schema.our_songs (artist_Id);

# EXPLAIN

-> Limit: 25 row(s)  (actual time=3852.757..3852.761 rows=25 loops=1)

    -> Sort: round(sum(Final_Proj_Schema.our_songs.danceability),0) DESC, limit input
to 25 row(s) per chunk  (actual time=3852.757..3852.758 rows=25 loops=1)

        -> Table scan on <temporary>  (actual time=0.001..11.642 rows=70259 loops=1)

            -> Aggregate using temporary table  (actual time=3828.864..3844.534
rows=70259 loops=1)

                -> Nested loop inner join  (cost=311211.11 rows=867266) (actual
time=0.979..3194.503 rows=922587 loops=1)

                    -> Table scan on artists  (cost=7599.35 rows=74626) (actual
time=0.099..26.760 rows=71361 loops=1)

                    -> Index lookup on our_songs using our_songs_artist_Id
(artist_Id=Final_Proj_Schema.artists.artistId)  (cost=2.91 rows=12) (actual
time=0.024..0.043 rows=13 loops=71361)
```

We found this made performance significantly worse, likely for the same reason as artists.artistId; we are indexing a primary key which makes the entries more disorganized and less efficient to access.

We conclude that option 2 is the best way to index the second query.