

# Project 1: Spotify Simulator

Priyanshu Gautam, 2017CS10362

Rachit Kumar, 2017CS10364

## 1 Project Description

Our project involves making a website to help a music aficionado create his own library of music. We'll have a library of songs with all its stats(album, artists, genres, danceability, number of streams etc) and will provide search functionalities like find the artist with most number of genres, find the most popular song by an artist with low danceability, find the most popular genres, etc and update functionalities like add/delete a song, add/delete an album, add/delete an artist etc. We can search for a song and then play it, it essentially creates a playlist for us, given specifications.

The ER Diagram is given below:

## 2 Data Source

We got the data from the Spotify Web API using [spotipy](#). With Spotipy, we got full access to all of the music data provided by the Spotify platform. We extracted all data from 2000 onwards. We found out about spotipy and the data from [kaggle](#).

The code for extraction is present in the github repository. There was no need for cleaning of any data as there was no noise.

The data has:

1. More than 40,000 songs (all songs from the dataset mentioned above after the year 2000)
2. More than 16000 artists
3. More than 15000 albums
4. More than 2500 genres
5. Data about popularity, danceability, acousticness, energy, instrumentalness, liveness, loudness, speechiness, tempo of songs. These features are computed using proprietary owned by spotify, but the name itself discloses a lot on the meaning and implication of these features.
6. Data about type, popularity, release date, label, etc of albums
7. Data about popularity, type, followers etc of artists

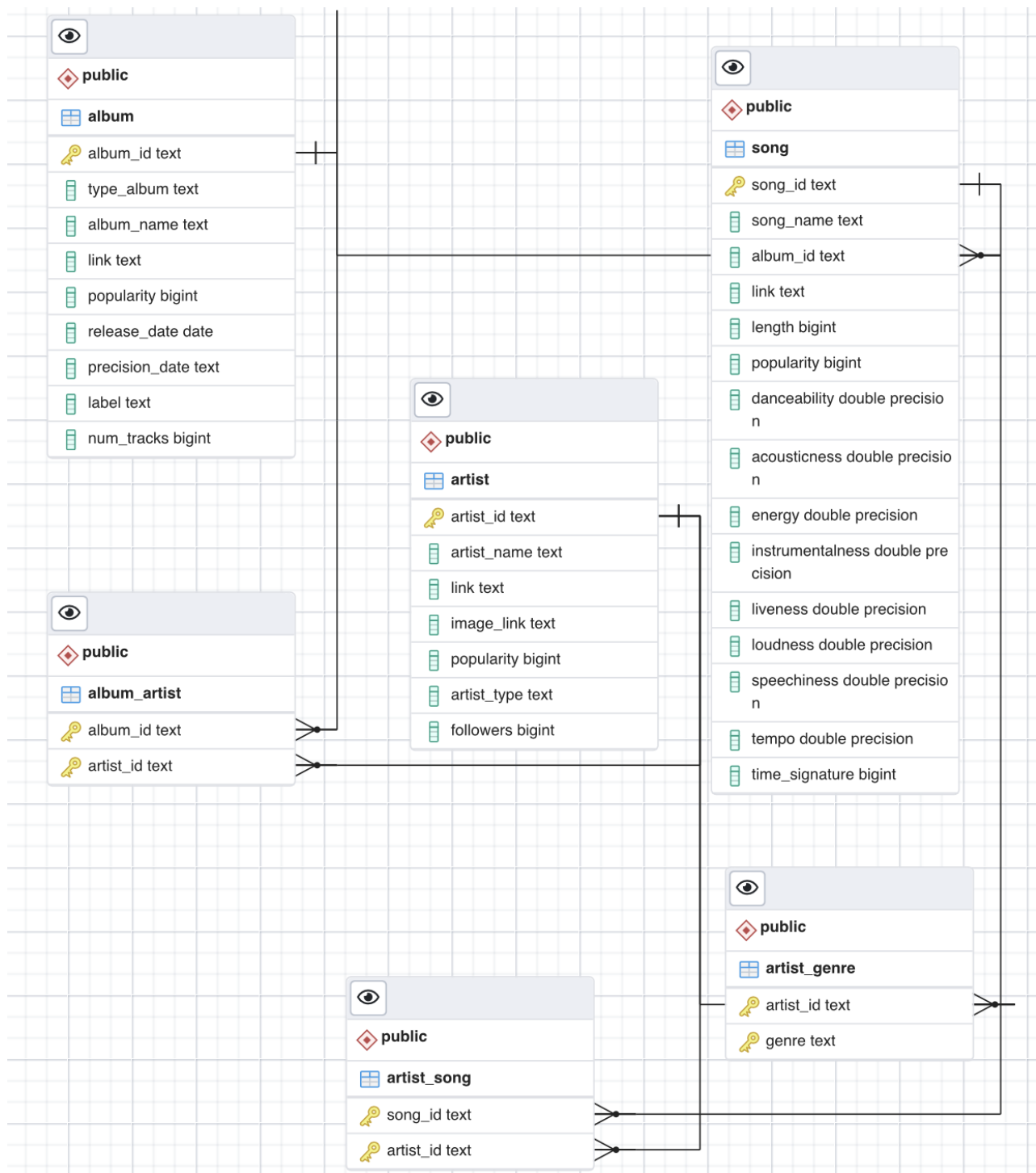


Figure 1: The main ER diagram showing all the relations as well as attributes, marking the primary keys and foreign key references as well

Table 1: Data Stats

Table	Time(in ms)	Size	Tuples originally	Unique Tuples	# of Attributes
Album	1083	6.1 MB	42368	15713	9
Artist	915	11 MB	68219	16028	7
Song	2309	7.6 MB	42368	40324	15
Artist-Song	2681	2.6 MB	57036	54478	2
Artist-Genre	3357	6.6 MB	184863	33403	2
Album-Artist	1435	2.0 MB	45468	17068	2

During the scraping process, many records were duplicated, therefore the cleanup required was to select distinct, therefore we used a select distinct on the full dataset available for each table. The summary stats are given here:

### 3 Functionality

#### 3.1 User's view of the system

The home page has 9 options for the user to select from. 3 of the operations are privileged as the data is manually added to the database using the three operations. Where it is necessary for consistent data to be entered, and therefore these operations are privileged. On clicking, let's say, search a song, we are faced with a form like interface. In this interface we have to fill in what kind of result we want(song in this case: to be popular or not, by a particular artist, have a particular name, from a particular album) etc. And we finally click submit and voila we get our results.

The results look like these, with the option to play the song directly, so not only we can just search the song, we can play the song.

In case of insert and delete functions, a page will come up conveying whether the operation was successful or not.

#### 3.2 System view

Before we set upon this endeavour to enlist various views in our system, we must state some design choices we chose. There were two points of contention.

1. Materialized View vs Normal Views: The trade-off was faster search queries owing to *indexes* as without indexes, the run time produced was quite similar and at times faster for normal views. But, the time for updating materialized views seem to be bottle neck. The time needed for insert/delete becomes of the order of 4 6 seconds in the case of materialized view, but some of the search queries were faster by order of almost 25 times, but the time taken was in the order

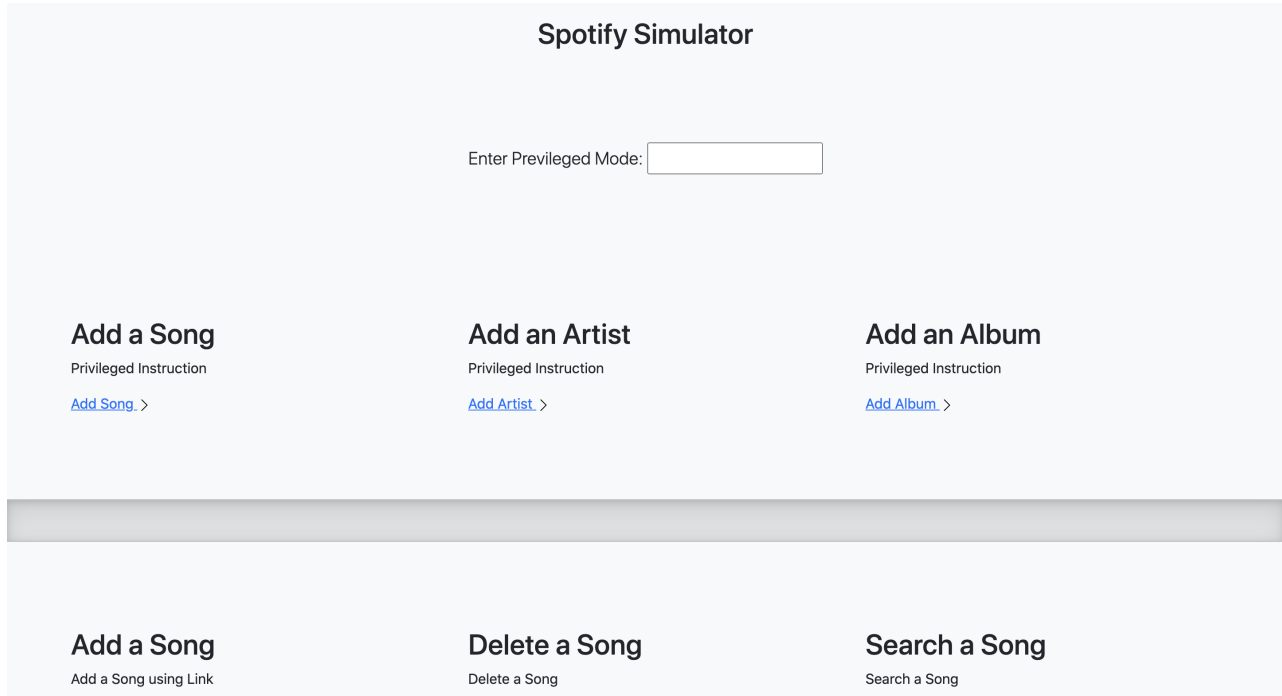


Figure 2: Home Page

of ms. With the given knowledge that insert or delete queries are extremely rare compared to search operation. There are 70 million songs on spotify accumulated over 15 years, while on a daily basis almost 1 billion streams happen on spotify on a daily basis. So, if we are to simulate a real application we must choose the way where search operations are as fast as possible, while the update operations happening on the server side can take a few seconds. But for demonstration purposes maybe an approach where update as well as search operations are fast can be considered a better deal.

2. How to update Materialized views. Using triggers vs updating in the server code itself as an update operation happens.

Triggers are *triggered* at each update operation. This is not desirable as when we add a song, we can perform many update operations for a single song, therefore the time taken is multiplied by as many operations taken. This at an average lead to insert and delete taking 30 seconds for adding a single song, we realised this is not practical therefore using triggers was decided against.

The various views used:

1. **songalbum**: This materialized view is a join of *song* and *album* table used for searching for songs having particular albums. There are indexes associated with the materialized view. The indexes are:

## Spotify Simulator

Artist

Album

Song

Popularity

Choose... ▼

Danceability

Choose... ▼

Acousticness

Choose... ▼

Energy

Choose... ▼

Instrumentalness

Choose... ▼

Liveness

Choose... ▼

Loudness

Choose... ▼

Speechiness

Choose... ▼

Tempo

Choose... ▼

Number of Results Needed

Order of Results

Choose... ▼

Submit

Figure 3: Search page






Songs			
Serial No	Artist	Song	Song Link
1		drivers license	<a href="#">Play &gt;</a>
2		Mood (feat. iann dior)	<a href="#">Play &gt;</a>
3		Astronaut In The Ocean	<a href="#">Play &gt;</a>
4		Bandido	<a href="#">Play &gt;</a>
5		The Business	<a href="#">Play &gt;</a>

Figure 4: Result of Query

- (a) `song_name_index`
- (b) `song_id_index`
- (c) `album_id_index`
- (d) `album_name_index`
- (e) `release_date_index`
- (f) `label_index`

The functionality of these indexes are understandable from their names itself. All these indexes are hash indexes which facilitate easy searching for queries which have equality check. The times are sped up by almost 40-50 times when analysed using explain keyword.

2. **songart**: As with materialized view **songalbum** this too is a join, this time of *song* and *artist*. This view is used for searching for songs by particular artists. This also has all the various indexes as mentioned above to speed up search. The speed up observed was of the order of 10-20 times on using the indexes vs not using the indexes.
3. **my\_genre\_view**: This is also a materialized view. This view helps in the **Get Genre Trends over Artist**, this materialized view aggregates genres over artists. With the help of next view it makes the query faster as well as convenient to use.
4. **simple\_genre\_view**: This is not a materialized view. This view is created on **my\_genre\_view**. We unnest the genres here, the reason for doing so here and not in **my\_genre\_view** is that we look to optimise storage. As otherwise number of rows would become of the order of 33000 compared to 16000 presently. Therefore we create a view when we unnest so that storage is optimised and then we can aggregate over genres to see which genres are the most frequent.
5. **song\_year**: This creates a join of song table and album table to get the year for the songs, this makes the computation of next view faster. This is also a materialized view
6. **pop\_year\_song**: This materialized view is used to help in the query of **Getting Song Trends** over the year, here we make the view such that it stores the rank of each song in terms of popularity over the year it was released.
7. **pop\_year\_album**: This materialized view is used to help in the query of **Getting Album Trends** over the year, here we make the view such that it stores the rank of each album in terms of popularity over the year it was released within the years specified by the user. For the above two views we define two indexes each:
  - (a) **Year Index**: This indexes the year, it helps in finding the range of year specified by the user.

- (b) **Rank Index:** This indexes the rank of each entity, and helps to select all those entities above a certain rank specified by the user.

Year index has both hash and BTree indexes, and rank index is only based on BTree.

## 4 List of queries Used

### 4.1 Add a Song(Privileged)

Song Name : Keep A Song In Your Soul

Song Link : <https://open.spotify.com/track/0cS0A1fUEUd1EW3FcF8AEI?si=9e45917aedc84d03>

Album ID : 2xjQOixp5YLkhVDcAh8MY0

Song Length : 168333

Popularity : 12

Danceability : 0.598

Acousticness : 0.991

Energy : 0.22399999999999998

Instrumentalness : 0.000522

Liveness : 0.379

Loudness : -12.628

Speechiness : 0.0936

Tempo : 149.976

Time Signature : 1920

#### Query Created

```
Insert into song(song_id,song_name,link,album_id,length,Popularity,Danceability,
acousticness,Energy,Instrumentalness,Liveness,Loudness,Speechiness,Tempo,time_signature)
values('0cS0A1fUEUd1EW3FcF8AEI', 'Keep A Song In Your Soul',
'https://open.spotify.com/track/0cS0A1fUEUd1EW3FcF8AEI?si=9e45917aedc84d03',
'2xjQOixp5YLkhVDcAh8MY0', '168333', '12', '0.598', '0.991', '0.22399999999999998',
'0.000522', '0.379', '-12.628', '0.0936', '149.976', '1920');
```

### 4.2 Add a Song(Using Spotify Link)

Song Link : <https://open.spotify.com/track/7lPN2DXiMsVn7XUKtOW1CS>

Query Created:

```
Insert into album values ('66FPnVL9G4CMKy3wvaGTcr', 'album', 'drivers license',
'https://open.spotify.com/album/66FPnVL9G4CMKy3wvaGTcr', 94, '2021-01-08', 'day',
'Olivia Rodrigo PS', 1) ON CONFLICT DO NOTHING;
```

```

Insert into song values ('71PN2DXiMsVn7XUKtOW1CS', 'drivers license',
'66FPnVL9G4CMKy3wvaGTcr', 'https://open.spotify.com/track/71PN2DXiMsVn7XUKtOW1CS',
242013, 100, 0.585, 0.721, 0.436, 1.31e-05, 0.105, -8.761, 0.0601, 143.874, 4);
Insert into artist_song values ('71PN2DXiMsVn7XUKtOW1CS', '1McMsnEElThX1knmY4oliG')
ON CONFLICT DO NOTHING;
Insert into album_artist values ('66FPnVL9G4CMKy3wvaGTcr', '1McMsnEElThX1knmY4oliG')
ON CONFLICT DO NOTHING;
Insert into artist_genre values ('1McMsnEElThX1knmY4oliG', 'alt z')
ON CONFLICT DO NOTHING;
Insert into artist_genre values ('1McMsnEElThX1knmY4oliG', 'pop')
ON CONFLICT DO NOTHING;
Insert into artist_genre values ('1McMsnEElThX1knmY4oliG', 'post-teen pop')
ON CONFLICT DO NOTHING;

```

### 4.3 Add an Artist(Privileged)

Artist: "Cats" 1981 Original London Cast

Artist Link: <https://open.spotify.com/artist/5PwgSnrvG48DYvdsf2Y2ZD>

Image Link: <https://open.spotify.com/artist/5PwgSnrvG48DYvdsf2Y2ZD>

Popularity:

Artist Type: artist

Followers: 176643

Query Created:

```

Insert into artist(artist_id,Artist_name,link,image_link,artist_type,followers)
values('5PwgSnrvG48DYvdsf2Y2ZD', '"Cats" 1981 Original London Cast',
'https://open.spotify.com/artist/5PwgSnrvG48DYvdsf2Y2ZD',
'https://open.spotify.com/artist/5PwgSnrvG48DYvdsf2Y2ZD', 'artist', '176643');

```

### 4.4 Add an Album(Privileged)

Album : Cats (Original London Cast Recording / 1981)

Album Link: <https://open.spotify.com/album/66UrYsmX0lRzAOwLxkPgjl>

Album Type: album

Popularity: 51

Release Date: 1981-05-11

Precision Date

Label: Parlophone UK

Number of Tracks: 23



Query Created:

```
Insert into album(album_id,Album_name,link,type_album,Popularity,Release_Date,
precision_date,Label,num_tracks) values('66UrYsmX0lRzA0wLxkPgjl',
'Cats (Original London Cast Recording / 1981)',
'https://open.spotify.com/album/66UrYsmX0lRzA0wLxkPgjl', 'album', '51', '1981-05-11',
'11/05/1981', 'Parlophone UK', '23');
```

#### 4.5 Delete a Song(Using Link)

Song Link: <https://open.spotify.com/track/7lPN2DXiMsVn7XUKtOW1CS>

Query Created:

```
delete from song where link = 'https://open.spotify.com/track/7lPN2DXiMsVn7XUKtOW1CS';
```

#### 4.6 Delete a Song(Using Name)

Song: drivers license

Query Created:

```
delete from song where song_name = 'drivers license';
```

#### 4.7 Search a Song

Artist:

Album:

Song:

Popularity: Medium

Danceability: Very High

Acousticness: High

Energy: High

Instrumentalness: Very Low

Liveness:

Loudness:

Speechiness:

Tempo:

Number of Results Needed: 10

Order of Results: Decreasing Popularity

Query Created:

```
select song_name, song_link, array_agg(image_link) from songart
where true and Danceability>=0.8 and Danceability<1.1 and Acousticness>=0.6 and
Acousticness<0.8 and Energy>=0.6 and Energy<0.8 and Instrumentalness>=0 and
Instrumentalness<0.2 and Speechiness>=0.2 and Speechiness<0.4 and Popularity>=40 and
Popularity<60 group by song_name, song_link ,songart.Popularity
order by songart.Popularity desc limit 10;
```

#### 4.8 Genre Trends

Minimum Followers: 40

Maximum Followers: 100000000

Popularity: High

Query Created:

```
select genre, count(genre) from simple_genre_view
where true and followers >=40 and followers <=100000000 and Popularity>=60 and Popularity<80
group by genre order by count desc limit 10;
```

#### 4.9 Song Trends

Start Year: 2005

End Year: 2009

Limit: 10

Query Created:

```
select song_id, song_name, song_link, rank, year from pop_year_song
where rank<=10 and year <=2009 and year>=2005;
```

#### 4.10 Album Trends

Start Year: 2001

End Year: 2005

Limit: 10

Query Created:

```
select album_id, album_name, album_link, rank, year from pop_year_album
where rank<=10 and year <=2005 and year>=2001;
```

Table 2: Query Runtimes

Query Number	Time(in ms)
1	25.089
2	18.219 + 2924(to refresh)
3	14.160 + 70(to refresh)
4	10.552 + 78(to refresh)
5	26.638 + 2924(to refresh)
6	18.990 + 2924(to refresh)
7	21.252
8	15.403
9	11.997
10	6.587