CSCI620-01 Project

Phase 2 Write Up

Group 5:

Vinod Dalavai | vd1605

Ramprasad Kokkula | rk1668

Samson Zhang | sz7651

March 31, 2023

1 Document-Oriented Model

1.1 Data Model Description

For ease of access, here is a basic review of our data set.

Our dataset contains information on one million playlists created by Spotify users. The dataset is sourced from Kaggle, which can be found here.

The dataset is stored in a thousand json files, with each json file storing a thousand playlists.

Because the dataset is already stored as json files, it already fits a document-oriented model.

Each playlist is represented by a json object containing information relevant to the playlist, and each json file contains 1000 of such playlists. Therefore, our document-oriented model treats each playlist as a document

1.2 Data Model Comparison

A very straight forward difference between our two different data models is that our relational model stores the data in multiple "tables", whereas our document-oriented model stores everything in a single "collection".

More specifically, the relational model is normalized whereas the document-oriented model is not. The relational model breaks up the dataset into different entities and relationships between the entities. As such, there is not only a table for playlist, but also a table for track, artist, album, etc. On the other hand, the document-oriented model stores just playlists with information attached to them.

The relational model makes it easier to run queries that are independent from playlists (e.g. a query that only deals with artists), whereas the document-oriented model makes it easier to run queries on playlists as a whole.

1.3 Loading The Data

The loader.py file contains the code responsible for loading the data into a MongoDB database.

To run the program, a host name and a port number is required. The command line argument is as the following:

```
python loader.py -H <hostname> -P <port_number>
```

Here is an example:

```
python loader.py -H localhost -P 27107
```

The program will first handel the input arguments from command line, and use it to establish a connection to the MongoDB database. Then, once the connection is established, it will load the data.

The program assumes that there is a folder named data in the same directory as itself. The data folder contains the json files that stores our data set.

The data loading process is relatively simple, the general process is described with the following psudo-code:

```
1. for each of the json files in the data folder
2.   load the json file
3.   for each playlist within the json file
4.     remove unwanted attributes
5.     adjusts the uri attributes
6.   insert the json file into mongodb
```

The unwanted attributes are:
modified_at, num_tracks, num_albums, num_followers, num_edits, num_artists

The uri adjustment is there because the uri specifies what type of uri it is:

- A track uri always starts with "spotify:track:"
- An artist uri always starts with "spotify:artist:"
- An album uri always starts with "spotify:album:"

The adjustment removes these prefixes.

2 Five Interesting Queries

2.1 Executing Queries

To run the queries, the program executor.py is provided. To run any query, run executor.py and specify the host name, data base name, and query number, as seen below:

```
python executor.py -H <hostname> -D <database name> -Q <query number>
```

Below is an example to run query 1 using local host and database named imdb:

```
python executor.py -H localhost -D imdb -Q 1
```

There are a total of 5 queries. The descriptions of these queries can be viewed by running executor.py with the -h flag:

```
python executor.py -h
```

Which will print the following in the terminal:

```
Here are your options to query the spotify dataset consisting of
250k user created playlists:
1: Find common artists that appear in both happy and sad playlists
3: 10 most common tracks shared across most of the playlists
4: Top 10 "Christmas" themed songs
5: Top 5 playlists with the maximum average track lengh and Top 5
playlists with the minimum average track length
```

2.2 Query 1

2.2.1 Query Description

Find common artists that appear in both happy and sad playlists.

In this query, we are interested in the tracks that appear in both “happy” and “sad” playlists, and the artists that created the most of these kind of tracks. The main idea is that it is possible that while some people associate a track with happy emotions, others associate the the track with sad emotions.

Extending this, we can then find the artists who are associated with the most number of both sad and happy tracks. With that, we will have a set of artists who are perceived as creating music that can be either happy or sad depending on the listener.

2.2.2 SQL Statement

```
SELECT
        happy_sad_tracks.artist_name as artist,
        count(1) as number_of_tracks
FROM (
        SELECT DISTINCT
                track_happy.id AS track_id,
                track_happy.name AS track_name,
                artist_happy.id AS artist_id,
                artist_happy.name AS artist_name
        FROM
                playlist AS playlist_happy
                JOIN track_playlist AS track_playlist_happy ON
                    playlist_happy.playlist_id = track_playlist_happy.playlist_id
                JOIN track AS track_happy ON
                    track_happy.id = track_playlist_happy.track_id
                JOIN track_artist AS track_artist_happy ON
                    track_artist_happy.track_id = track_happy.id
                JOIN artist AS artist_happy ON
                    artist_happy.id = track_artist_happy.artist_id
        WHERE
                lower(playlist_happy.name) LIKE '%happy%'
                AND EXISTS (
                        SELECT
                                track_sad.id AS track_id_sad,
                                track_sad.name AS track_name_sad,
                                artist_sad.id AS artist_id_sad,
                                artist_sad.name AS artist_name_sad
                        FROM
                                playlist AS playlist_sad
                                JOIN track_playlist AS track_playlist_sad ON
                                    playlist_sad.playlist_id =
                                    track_playlist_sad.playlist_id
                                JOIN track AS track_sad ON
                                    track_sad.id = track_playlist_sad.track_id
                                JOIN track_artist AS track_artist_sad ON
                                    track_artist_sad.track_id = track_sad.id
                                JOIN artist AS artist_sad ON
                                    artist_sad.id = track_artist_sad.artist_id
                        WHERE
                                lower(playlist_sad.name) LIKE '%sad%'
                                AND track_happy.id = track_sad.id
                )) happy_sad_tracks
GROUP BY (happy_sad_tracks.artist_id, happy_sad_tracks.artist_name)
ORDER BY count(1) DESC;
```

This is the most complicated query out of the five queries. We will decompose it and explain it in parts.

The most inner subquery is responsible for returning the set of all tracks that are perceived as sad, here is the query on its own:

```
SELECT
track_sad.id AS track_id_sad,
track_sad.name AS track_name_sad,
artist_sad.id AS artist_id_sad,
artist_sad.name AS artist_name_sad
FROM
        playlist AS playlist_sad
        JOIN track_playlist AS track_playlist_sad ON
                playlist_sad.playlist_id = track_playlist_sad.playlist_id
        JOIN track AS track_sad ON
                track_sad.id = track_playlist_sad.track_id
        JOIN track_artist AS track_artist_sad ON
                track_artist_sad.track_idtrack_sad.id
        JOIN artist AS artist_sad ON
                artist_sad.id = track_artist_sad.artist_id
WHERE
        lower(playlist_sad.name) LIKE '%sad%'
        AND track_happy.id = track_sad.id
```

The second “WHERE” condition is used to match the tracks that are considered both happy and sad, which is related to the outer query.

If we take a look at the query that one level higher than the above query, we see that it is essentially doing the same thing, except it searches for all tracks that are considered happy, and only keeps the ones that are also considered sad (the previous query).

```
SELECT DISTINCT
                track_happy.id AS track_id,
                track_happy.name AS track_name,
                artist_happy.id AS artist_id,
                artist_happy.name AS artist_name
FROM
        playlist AS playlist_happy
        JOIN track_playlist AS track_playlist_happy ON
            playlist_happy.playlist_id = track_playlist_happy.playlist_id
        JOIN track AS track_happy ON
            track_happy.id = track_playlist_happy.track_id
        JOIN track_artist AS track_artist_happy ON
            track_artist_happy.track_id = track_happy.id
        JOIN artist AS artist_happy ON
            artist_happy.id = track_artist_happy.artist_id
WHERE
        lower(playlist_happy.name) LIKE '%happy%'
        AND EXISTS (...)
```

Therefore, with the previous two nested queries, we now have a set of tracks that are considered both happy and sad, along with the info of the artists associated with them.

Then, the most outer query simply groups by artists and count how many times each of them appear in tracks hat are perceieved as both happy and sad.

2.2.3 Query Output

```
Executing query#1...
Find common artists that appear in both happy and sad playlists

Execution Time = 18.72 seconds

{'artist': 'Drake', 'number_of_tracks': 107}
{'artist': 'Justin Bieber', 'number_of_tracks': 92}
{'artist': 'Coldplay', 'number_of_tracks': 86}
{'artist': 'Ed Sheeran', 'number_of_tracks': 83}
{'artist': 'One Direction', 'number_of_tracks': 77}
{'artist': 'John Mayer', 'number_of_tracks': 76}
{'artist': 'Taylor Swift', 'number_of_tracks': 68}
{'artist': 'Lana Del Rey', 'number_of_tracks': 65}
```

```
{'artist': 'The Weeknd', 'number_of_tracks': 64}
{'artist': 'Fall Out Boy', 'number_of_tracks': 62}
TOTAL ROWS = 4452
```

For viewing purposes, the program does not print all rows of the query result. We only print out the first 10 rows of the result, representing the top 10 artists who appear in both happy and sad playlists.

2.3 Query 2

2.3.1 Query Description

Find the top 10 most popular artists across all playlists.

In this query we want to find the most popular artists. The variable we use to determine popularity is the number of times an artist appears in all of the playlists. In other words, an artist is popular if they have many tracks in many playlists.

Note that we should not count only distinct tracks, as that would instead be "find artists who created the most tracks in the database". We are not concerned with the specifics of a track, aside from the artist who created the track.

The main idea is that having many tracks across multiple playlists should be a good indicator for popularity. In general, we can suggest that the artist with the most number of tracks (counting repeats) across all playlists should be popular, as many people includes the artist's tracks in their playlist.

2.3.2 SQL Statement

```
SELECT
        artist.name as artist,
        count(1) total_number_of_tracks
FROM
        playlist
        JOIN track_playlist ON track_playlist.playlist_id = playlist.playlist_id
        JOIN track ON track.id = track_playlist.track_id
        JOIN track_artist ON track_artist.track_id = track.id
        JOIN artist ON artist.id = track_artist.artist_id
GROUP BY (artist.id, artist.name)
ORDER BY total_number_of_tracks DESC
LIMIT 10;
```

In order to run our query, we need to denormalize our data first, which is done by joining the tables. We join playlist and track with track_playlist to get a table that contains the full set of playlists and what tracks each playlist contain. We then join with artist using track_artist so we have information on the artist of each track.

Once we have a denormalized table, we can simply group by the artists and count how many times each artist appears. The number of times each artist appears is equivalent to the result we are interested in because there is one row of data for each track.

We sort the result by descending order of the count, and limit the result to only the first 10 rows, as we are interested in the top 10 most popular artists.

2.3.3 Query Output

```
Executing query#2...
Find the 10 most popular artists across all playlists

Execution Time = 29.14 seconds

{'artist': 'Drake', 'total_number_of_tracks': 846937}
{'artist': 'Kanye West', 'total_number_of_tracks': 413297}
{'artist': 'Kendrick Lamar', 'total_number_of_tracks': 353624}
{'artist': 'Rihanna', 'total_number_of_tracks': 339570}
{'artist': 'The Weeknd', 'total_number_of_tracks': 316603}
{'artist': 'Eminem', 'total_number_of_tracks': 294667}
{'artist': 'Ed Sheeran', 'total_number_of_tracks': 272116}
{'artist': 'Future', 'total_number_of_tracks': 249986}
{'artist': 'Justin Bieber', 'total_number_of_tracks': 243119}
{'artist': 'J. Cole', 'total_number_of_tracks': 241556}
TOTAL ROWS = 10
```

2.4 Query 3

2.4.1 Query Description

10 Most common tracks shared amongst playlists

In this query, we are interested in the tracks that appear the most. This is a way to indicate popularity of specific tracks, the track that appears frequently (not counting duplicates) across many playlists would likely be popular.

2.4.2 SQL Statement

```
SELECT
        distinct_tracks.name as track,
        count(1) AS number_of_occurences
FROM
    (SELECT
        track.id, track.name, playlist.playlist_id
    FROM
        playlist
        JOIN track_playlist ON track_playlist.playlist_id = playlist.playlist_id
        JOIN track ON track.id = track_playlist.track_id
    GROUP BY (track.id, track.name, playlist.playlist_id)) as distinct_tracks
GROUP BY (distinct_tracks.id, distinct_tracks.name)
ORDER BY (number_of_occurences) DESC
LIMIT 10;
```

This query is similar to the previous query. Because we do not need any artist information, we do not need to join the artist related tables.

However, we do need to make sure that we do not count duplicate tracks within the same playlist. This is done in the inner subquery.

The subquery joins playlist and track using track_playlist, and then group by track id, track name, and playlist id. This has the same effect as only selecting distinct tuples of (track id, track name, playlist id), because we group every row by those three attributes.

The outer query simply groups by track id and track name (we can't group by just name because there can be multiple tracks with the same name), and count how many times each track appears. Because the inner query already handles the duplicate cases, the outer query does not need to handle it.

2.4.3 Query Output

```
Executing query#3...
10 most common tracks shared across most of the playlists

Execution Time = 1.68 minutes

{'track': 'HUMBLE.', 'number_of_occurences': 45394}
{'track': 'One Dance', 'number_of_occurences': 41707}
{'track': 'Broccoli (feat. Lil Yachty)', 'number_of_occurences': 40659}
{'track': 'Closer', 'number_of_occurences': 40629}
{'track': 'Congratulations', 'number_of_occurences': 39577}
{'track': 'Caroline', 'number_of_occurences': 34765}
{'track': 'iSpy (feat. Lil Yachty)', 'number_of_occurences': 34672}
```

```
{'track': 'Bad and Boujee (feat. Lil Uzi Vert)', 'number_of_occurences': 34157}
{'track': 'XO TOUR Llif3', 'number_of_occurences': 34048}
{'track': 'Location', 'number_of_occurences': 33954}
TOTAL ROWS = 10
```

2.5 Query 4

2.5.1 Query Description

Top 10 "Christmas" themed tracks.

For this query, we are interested in tracks that are perceived with a Christmas theme. In order to get the results, we find all playlists that contains the word "Christmas", and find the most frequently appearing tracks within those playlists.

2.5.2 SQL Statement

```sql
SELECT
        track.name as track,
        count(1) as number_of_occurrences
FROM
        playlist
        JOIN track_playlist ON playlist.playlist_id = track_playlist.playlist_id
        JOIN track ON track.id = track_playlist.track_id
WHERE lower(playlist.name) LIKE '%christmas%'
GROUP BY (track.id, track.name)
ORDER BY (number_of_occurrences) DESC
LIMIT 10;
```

For this query, we only need information on playlists and tracks, therefore, we join playlist, track_playlist, and track together.

We limit our dataset to only playlists that contains the word "Christmas" in its name, then we can group by track id and track name and count each tracks.

2.5.3 Query Output

```
Executing query#4...
Top 10 "Christmas" themed songs

Execution Time = 9.94 seconds

{'track': 'All I Want for Christmas Is You', 'number_of_occurrences': 7715}
{'track': "Rockin' Around The Christmas Tree - Single Version",
                                            'number_of_occurrences': 5782}
{'track': "It's Beginning To Look A Lot Like Christmas",
                                            'number_of_occurrences': 5759}
{'track': 'White Christmas', 'number_of_occurrences': 4930}
{'track': 'The Christmas Song (Merry Christmas To You)',
                                            'number_of_occurrences': 4732}
{'track': "It's the Most Wonderful Time of the Year",
                                            'number_of_occurrences': 4621}
{'track': 'Jingle Bell Rock', 'number_of_occurrences': 4507}
{'track': 'A Holly Jolly Christmas - Single Version',
                                            'number_of_occurrences': 4415}
{'track': 'Holly Jolly Christmas', 'number_of_occurrences': 4112}
{'track': 'Last Christmas', 'number_of_occurrences': 3785}
TOTAL ROWS = 10
```

2.6 Query 5

2.6.1 Query Description

Average track length in each playlist.

In this query, we are interested in comparing long playlists and short playlists. Here we define a long playlist as a playlist with a long duration (as opposed to having many tracks).

Specifically, this query will return the top 5 playlists with maximum average track length, and the top 5 playlists with the minimum average track length.

2.6.2 SQL Statement

```
SELECT *
FROM
        (SELECT
                playlist.playlist_id,
                playlist.name as playlist,
                ROUND(AVG(track.duration)/(1000*60), 2)
                                as "average_track_duration(min)"
        FROM
                playlist
                JOIN track_playlist ON
                    playlist.playlist_id = track_playlist.playlist_id
                JOIN track ON track.id = track_playlist.track_id
        GROUP BY(playlist.playlist_id)
        ORDER BY ("average_track_duration(min)") DESC
        LIMIT 5) as longest
UNION
SELECT *
FROM
        (SELECT
                playlist.playlist_id,
                playlist.name as playlist,
                ROUND(AVG(track.duration)/(1000*60), 2)
                                as "average_track_duration(min)"
        FROM
                playlist
                JOIN track_playlist ON
                    playlist.playlist_id = track_playlist.playlist_id
                JOIN track ON track.id = track_playlist.track_id
        GROUP BY(playlist.playlist_id)
        ORDER BY ("average_track_duration(min)") ASC
        LIMIT 5) as shortest
 ORDER BY ("average_track_duration(min)") DESC;
```

Because this query returns both the top 5 *maximum* and *minimum* average track length playlists, we have one query that selects the maximum, and one query that selects the minimum. The result can be then obtained using the union operation on the two sets of results. The two queries connected by union are very similar, they differ only in their order. Therefore, we will only analyze the first one here.

We want information playlists and tracks, so we start by joining playlist, track_playlist, and track. With the playlist and track information, we can then group by the playlist id, and take the average of the duration of each tracks in individual playlists.

The duration of the tracks are originally in units of milliseconds, we convert it into minutes as that is more intuitive and easier to get a quick sense of duration.

The top 5 longest average duration is sorted by descending order of average track duration, whereas the top 5 shortest average duration is sorted by ascending order of average track duration.

2.6.3 Query Output

```
Executing query#5...
Top 5 playlists with the maximum average track lengh and
Top 5 playlists with the minimum average track length

Execution Time = 1.25 minutes

{'playlist_id': 349308, 'playlist': 'Audiobooks',
            'average_track_duration(min)': Decimal('145.05')}
{'playlist_id': 816157, 'playlist': 'Sets',
            'average_track_duration(min)': Decimal('72.68')}
{'playlist_id': 405235, 'playlist': 'Mixes',
            'average_track_duration(min)': Decimal('71.98')}
{'playlist_id': 582049, 'playlist': 'Audiobooks',
            'average_track_duration(min)': Decimal('68.58')}
{'playlist_id': 174607, 'playlist': 'Mixes',
            'average_track_duration(min)': Decimal('67.52')}
{'playlist_id': 440558, 'playlist': 'Halloween',
            'average_track_duration(min)': Decimal('0.70')}
{'playlist_id': 557279, 'playlist': 'Quran',
            'average_track_duration(min)': Decimal('0.65')}
{'playlist_id': 514289, 'playlist': 'HEARTBEAT',
            'average_track_duration(min)': Decimal('0.60')}
{'playlist_id': 285883, 'playlist': 'Oh Baby ',
            'average_track_duration(min)': Decimal('0.41')}
{'playlist_id': 479170, 'playlist': 'video',
            'average_track_duration(min)': Decimal('0.33')}
TOTAL ROWS = 10
```

It seems reasonable that audio book tracks would have longer average duration. Similarly, it is intuitive that mixes are have longer average duration, as they are tracks that are created by remixing and adding multiple tracks together.

On the other end, it might appear odd to have tracks that are less than one minute in duration. However, it is only because we associate a Spotify track with a song, while most of these short tracks are not songs.

For instance, the playlist "HEARTBEAT" contains tracks that has average duration just a bit over half a minute. The tracks in this playlist are as follows:

track	duration (min)
Heartbeat Speeding Up (Version 2) [Heart Beating Beat Pumping Speed Speeds Pulse Noise Human] [Sound Effect]	0.5
Heartbeat Slow	1.02
Fast Pulsing Heartbeat	0.46
Heartbeat	0.54
Heartbeat (Version 4) [Heart Beating Beat Pumping Pulse Noise Human] [Sound Effect]	0.5

Given the track names, it seems like these tracks are just sound effects. As such, it would make sense that they are mostly less than a minute long.

3 Indexing

Originally the Uris were the unique identifier for the album, track and artist. We added a unique attribute *id* which is of type **SERIAL INT** corresponding to each uri. These *ids* are indexed to optimize query performance. If we had used the Uris in place of ids as the unique identifier, PostgreSQL would have made sequential scans for all of our queries. This is not desirable as it increases the query execution time.

4 Functional Dependencies

A functional dependency is a relationship between two attributes in a relation where the value of one attribute determines the value of another attribute. Functional Dependencies for each relation is described as follows

4.1 album

- **Dependency:** id → album_uri, name
- **Determinant:** id
- **Dependent:** album_uri, name

4.2 track

- **Dependency:** id → track_uri, name, duration, album_id, album_uri
- **Dependency:** album_id → album_uri
- **Determinant:** id, album_id
- **Dependent:** track_uri, name, duration, album_id

4.3 artist

- **Dependency:** id → artist_uri, name
- **Determinant:** id
- **Dependent:** artist_uri, name

4.4 playlist

- **Dependency:** playlist_id → name, collab, duration
- **Determinant:** playlist_id
- **Dependent:** name, collab, duration

4.5 album_artist

- **Dependency:** album_id, artist_id → album_id
- **Determinant:** album_id, artist_id
- **Dependent:** album_id

4.6 track_artist

- **Dependency:** track_id → artist_id
- **Determinant:** track_id
- **Dependent:** artist_id

4.7 track_playlist

- **Dependency:** track_id, playlist_id → position
- **Determinant:** track_id, playlist_id
- **Dependent:** position

5 Normalization

Normalization is the process of organizing data in a database to avoid data redundancy, inconsistencies and improve data integrity.

Entity Relations - album, track, artist, playlist.
Bridge Relations - track_artist, album_artist, track_playlist
Bridge relations are the relations created to normalize the Entity relations.

5.1 First Normal Form (1NF)

All the relations are in first normal form, since they have only atomic groups and not have any repeating groups.

5.2 Second Normal Form (2NF)

The track relation violates the 2NF because the album_uri is functionally dependent on the album_id, but not on track_id. To eliminate this we have split the track relation into two relations track and album. These relations are already present in the database, we added the album_uri to the track relation to be present as an extra attribute.

5.3 Third Normal Form (3NF)

All the relations are in 3NF, as there is no transitive dependency or redundant data.