Intern Career

Task 1: Data analysis

Initial database

For creating the database to use to analyze the dataset, a Python code was used to create it. This code will only generate one table which we will use to perform some analysis, in order to fully understand the dataset. This process will be done in the analysis stage. The first step taken was to create the database in our preferred SQL database. I used PostgreSQL for the SQL database. After this, the next step will be to run the code bellow.

- You will need to have the csv file and the python code saved in the same directory if you will use the same code, word for word.
- You will also be required to have the following packages installed :
    1. pandas
    2. sqlalchemy

```
import pandas as pd
from sqlalchemy import create_engine

postgresql_username = 'postgres'
postgresql_password = ''
postgresql_host = 'localhost'
postgresql_database = 'spotify_data'

database_url = f'postgresql+psycopg2://{postgresql_username}:
engine = create_engine(database_url)

csv_file_path = 'spotify-data.csv'

try:
    df = pd.read_csv(csv_file_path)
    table_name = 'music'
    df.to_sql(table_name, engine, index=False, if_exists='rep
```

```
except Exception as e:
    print(f"Error: {e}")

engine.dispose()
```

Analysis

From first glance the data seems to contain duplicates, and after further analysis there are duplicates. The problem with the duplicates is appearing when one wants to collect the duplicates together and put them in one table. As we use more columns to filter with, the number of duplicates keeps on decreasing in number until there are no duplicates. All the entries are similar in one way or the other. This means some entries are similar. So when it comes to identifying the key tables, one will choose almost all columns as a table and create primary keys in each, or use the whole dataset as one table and create queries from there. After further analysis, a conclusion was made of not including the id. The id was found to only uniquely identifies each row even if there are duplicate entries. A query is used which gets the duplicates and groups them according to all the columns, except the id column. Each entry will also include how many times the same data is occurring in the the table. This is archived by using the query :

```
select
    name, artists, duration_ms, release_date, year, acousticn
    instrumentalness, liveness, loudness, speechiness, tempo,
    key, popularity, explicit, count(name)
from music
group by(
    name,   artists,
    duration_ms, release_date,
    year,   acousticness,   danceability,
    energy, instrumentalness,
    liveness, loudness,
    speechiness, tempo,
    valence,    mode,
```

```
    key, popularity, explicit)
having count(name)>1;
```

If we want to get the remaining data we simply replace the greater than (>) sign with the equal to (=) sign in the above query. This will look as :

```
select
    name, artists, duration_ms, release_date, year, acousticn
    instrumentalness, liveness, loudness, speechiness, tempo,
    key, popularity, explicit, count(name)
from music
group by(
    name,   artists,
    duration_ms, release_date,
    year,   acousticness,   danceability,
    energy, instrumentalness,
    liveness, loudness,
    speechiness, tempo,
    valence,    mode,
    key, popularity, explicit)
having count(name)=1;
```

After further analysis there were no null values found in the table. This was done by using the query :

```
select * from music
where
id is NULL
or name is NULL
or artists is NULL
or duration_ms is NULL
or release_date is NULL
or year is NULL
or acousticness is NULL
or danceability is NULL
or energy is NULL
```

```
or instrumentalness is NULL
or liveness is NULL
or loudness is NULL
or speechiness is NULL
or tempo is NULL
or valence is NULL
or mode is NULL
or key is NULL
or popularity is NULL
or explicit is NULL;
```

The queries above will only separate the duplicates using all the columns except the id. As said above, when it comes to finding the duplicates depending on which columns one uses. For this reason some of the columns will not be used for displaying some of the data. This due to the fact that a some of the data displayed does not have meaning unless one has the knowledge of it.

Data exploration

From the dataset there seems to be two tables which could be created. One table will store the audio characteristics of the song and the other table stores the main song data which is commonly known. these two table are defined below and the keys which determine their relationship.

Tables to create

- songs (id(primary key), name, artists, duration_ms, release_date, year)
- audio_features (song_id(foreign key),acousticness, danceability, energy, instrumentalness, liveness, loudness, speechiness, tempo, valence, mode, key, explicit)

Database creation

For creating the database, a python code was used. This was caused by the large dataset. The code will only create a single table, which is the same csv file. After performing analysis on the table ,that’s when the creation of the tables will be done. To create the tables mentioned above the following query was used

```
creat table songs as
select
    id,
    name,
    artists,
    duration_ms,
    release_date,
    year
from music;

alter table songs
add constraint pk_songs primary key (id);

create table audio_features as
select
    id as song_id,
    acousticness,
    danceability,
    energy,
    instrumentalness,
    liveness,
    loudness,
    speechiness,
    tempo,
    valence,
    mode,
    key,
    explicit
from music;

alter table audio_features
```

```
add constraint fk_audio_features
foreignn key (song_id) references songs(id);
```

Basic queries

Bellow are some basic queries that can be performed on the tables we created above :

1. The query bellow groups the explicitness of of the songs(if explicit or not explicit) and shows the total number songs that are explicit or not explicit:

```
select explicit, count(explicit) from audio_features group
by explicit;
```

2. The query bellow shows the total average duration of all the songs

```
select avg(duration_ms) from songs;
```

3. The query bellow shows the year and the total popularity of the songs of that year. The output is ordered by descending order of the years

```
select year, count(popularity) from music group by(year) orde
```

4. The query bellow shows all the songs where the year is 2018

```
select  * from songs where year = 2018;
```

These queries provide a starting point for exploring and analyzing the data in the created tables.

Changing data type

To change the data type of a column in the music table, you can use the alter table statement with the alter column clause. For example, if you want to change the data type of the year column from its current type to integer, you can use the following query:

```
alter table music
alter column year type integer;
```

Remember to make sure that the new data type is compatible with the data in the column before making any changes.

Joins

1. the query bellow will retrieve all the songs with their corresponding audio features:

```
SELECT songs.name, artists, duration_ms, release_date, yea
r, acousticness, danceability, energy, instrumentalness, li
veness, loudness, speechiness, tempo, valence, mode, key, e
xplicit
FROM songs
INNER JOIN audio_features
ON songs.id = audio_features.song_id;
```

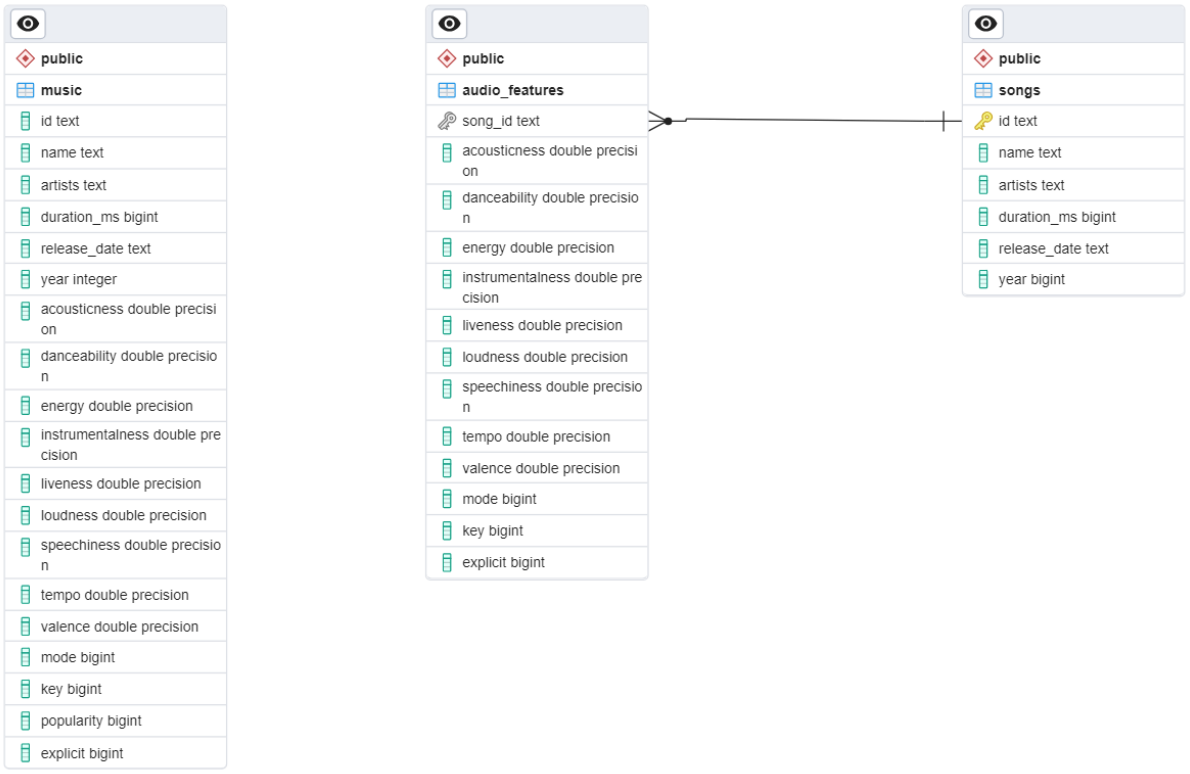
2. the query bellow will find the top 5 songs with the highest energy:

```
SELECT name, artists, energy
FROM songs
INNER JOIN audio_features
ON songs.id = audio_features.song_id
ORDER BY energy DESC
LIMIT 5;
```

Task 2: Database Design and Optimization

Database design (with Normalization and Denormalization)

The database ERD provided both the a normalized tables and the denormalized table. The design is shared below in the ERD image generated by pgAdmin4. From first glance you'll be able to see that the normalized table doesn't have more tables. The reasoning was to avoid breaking the table into smaller tables, hence creating a larger dataset to explore and making it take longer to search through them. Both the normalized and denormalized tables are present to allow the use of any of two types of database table when using the queries.



Indexing and tuning (with performance monitoring and query optimization)

A test was carried out by using two commonly used columns when searching for music. These are artists which display the artist(s) of the particular song in each row, and the year which is the release year of the song. There are two images below with one showing the performance of the queries before indexing and the other after indexing. After looking at the execution times you can clearly see that there is a very big difference in the execution time. This shows the impact indexing has done to the performance of the the query execution. You can see this bellow. The two queries which were run are :

1. For getting the data about the year

```
explain analyse * from music where year = 2020;
```

2. For getting the data about the artist

```
explain analyse * from music where artists = '[''Eminem'']';
```

Before indexing

QUERY PLAN
Gather (cost=1000.00..6940.70 rows=887 width=194) (actual time=8.289..144.126 rows=1756 loops=1)
Workers Planned: 2
Workers Launched: 2
-> Parallel Seq Scan on music (cost=0.00..5852.00 rows=370 width=194) (actual time=4.368..73.044 rows=585 loops=3)
Filter: (year = 2020)
Rows Removed by Filter: 56051
Planning Time: 0.271 ms
Execution Time: 144.314 ms

QUERY PLAN
Seq Scan on songs (cost=0.00..4875.86 rows=193 width=94) (actual time=4.830..94.151 rows=181 loops=1)
Filter: (artists = '[''Eminem'']'::text)
Rows Removed by Filter: 169728
Planning Time: 0.201 ms
Execution Time: 94.217 ms

After indexing

QUERY PLAN
Bitmap Heap Scan on music (cost=10.88..2217.18 rows=850 width=194) (actual time=0.506..1.863 rows=1756 loops=1)
Recheck Cond: (year = 2020)
Heap Blocks: exact=69
-> Bitmap Index Scan on year_idx (cost=0.00..10.67 rows=850 width=0) (actual time=0.447..0.448 rows=1756 loops=1)
Index Cond: (year = 2020)
Planning Time: 1.228 ms
Execution Time: 2.297 ms

QUERY PLAN
Bitmap Heap Scan on music (cost=5.69..567.62 rows=164 width=194) (actual time=0.098..0.283 rows=181 loops=1)
Recheck Cond: (artists = '["Eminem"]'::text)
Heap Blocks: exact=144
-> Bitmap Index Scan on artists_idx (cost=0.00..5.65 rows=164 width=0) (actual time=0.069..0.070 rows=181 loops=1)
Index Cond: (artists = '["Eminem"]'::text)
Planning Time: 0.226 ms
Execution Time: 0.342 ms

All of this analysis and indexing was done on the denormalized table. This was done to show the importance of indexing key columns that are normally used in the given dataset. Even though indexing is good it's not a good practice to use/create a lot of indexes but only the necessary columns. But with the help of these indexes when running some of the queries which use the year and artists filter