CSCI620-01 Project

Phase 1 Write Up

Group 5:

Vinod Dalavai | vd1605

Ramprasad Kokkula | rk1668

Samson Zhang | sz7651

March 2, 2023

1 Dataset Description

1.1 Overveiw and Source:

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

1.2 Files Description:

1.2.1 Overview

The dataset is stored in a thousand json files, with each file containing a thousand playlists (which totals to a million).

The json files follows the naming pattern of:

mpd.slice.[starting playlist number] − [ending playlist number]

For example, "mpd.slice.0-999" contains the first 1,000 playlists, note the use of 0-based numbering.

1.2.2 Formatting

Each of the json files contains the following:

{info, playlists}

Where *info* is the metadata of the file and contains the following:

1. "generated_on": time when the data was generated
2. "slice": the slice of playlists that this file contains (e.g. 0-999)
3. "version" the version of this file/data

A *playlist* contains the following:

1. "name": name of the playlist
2. "collaborative": whether playlist was made by more than one person
3. "pid": playlist id
4. "modified_at": the last time the playlist was modified
5. "num_tracks": number of tracks in the playlist
6. "num_albums": number of albums in the playlist

7. "num_followers": number of followers of the playlist
8. **"tracks"**: a *list* of all the tracks, **details below**
9. "num_edits": number of times playlist was edited
10. "duration_ms": total duration of playlist in milliseconds
11. "num_artists": number of artists in the playlist

A single *track* contains the following:

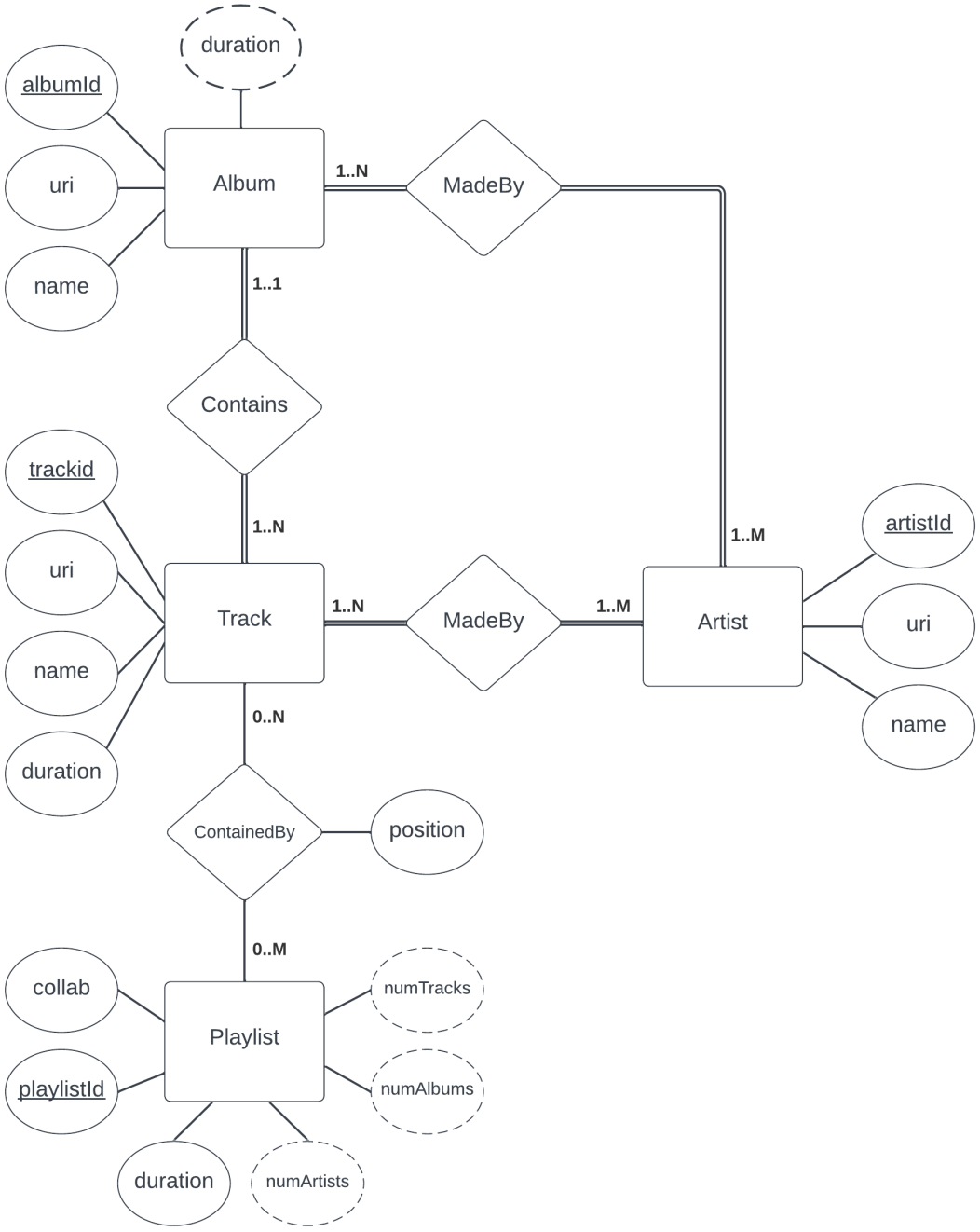
1. "pos": the position of this track in the playlist
2. "artist_name": artist of this track
3. "track_uri": Spotify URI of this track
4. "artist_uri": Spotify URI of the artist of this track
5. "track_name": name of this track
6. "album_uri": Spotify URI of the album of this track
7. "duration_ms": duration of this track in milliseconds
8. "album_name": name of the album of this track

Note that in our relational model, we leave out certain attributes. For example: we do not store "num_edits" and "num_followers" for a playlist. We are excluding these attributes because we don't think they represent useful information for our use case.

Additionally, some of the attributes can be derived. Therefore, these attributes are also not directly stored. For example, "num_trakcs" and "num_albums" in a playlist can be derived through queries.

While a playlist's duration can also be derived, we will need to join tables in order to get the duration of each track contained in the playlist. Because the dataset already provides playlists' duration, we store it for ease of access.

2 ER Diagram


duration
albumId
uri
name
Album
1..N
MadeBy
1..1
Contains
trackid
uri
name
duration
Track
1..N
MadeBy
1..M
Artist
artistId
uri
name
1..M
0..N
ContainedBy
position
0..M
collab
playlistId
Playlist
numTracks
numAlbums
duration
numArtists

3 Relational Model

3.1 Entities DDL

```
CREATE TABLE album(
albumid int PRIMARY KEY,
uri char(255),
name char(255)
)

CREATE TABLE track(
trackid int PRIMARY KEY,
uri char(255),
name char(255),
duration int,
albumid int,
FOREIGN KEY (albumid) REFERENCES album(albumid)
)

CREATE TABLE artist(
artistid int PRIMARY KEY,
uri char(255),
name char(255)
)

CREATE TABLE playlist(
playlistid int PRIMARY KEY
collab boolean,
duration int
)
```

For each attributes that are stored as strings, we assume a max length of 255 characters.

Note that because the "Album Contains Track" relationship is one to many, we just need to store albumid as a Foreign Key in the Track table (we only need to create a new table if the relationship is many to many).

3.2 N:M Relationships DDL

```
CREATE TABLE album_artist(
album int,
artist int,
PRIMARY KEY (album, artist),
FOREIGN KEY (album) REFERENCES album(albumid)
FOREIGN KEY (artist) REFERENCES artist(artistid)
)

CREATE TABLE track_artist(
track int,
artist int,
PRIMARY KEY (track, artist),
FOREIGN KEY (track) REFERENCES track(trackid)
FOREIGN KEY (artist) REFERENCES artist(artistid)
)

CREATE TABLE track_playlist(
track int,
playlist int,
position int,
PRIMARY KEY (track, playlist),
FOREIGN KEY (track) REFERENCES track(trackid)
FOREIGN KEY (playlist) REFERENCES playlist(playlistid)
)
```

Our schema has a total of four relationships, three of them are many to many relationships. For each of the many to many relationships, we create a new table. The new table will store foreign keys referencing the ids of the entities that participates in the relationship.

Additionally, the relationship between Track and Playlist has its own attribute (position), which is included in the table that represents the relationship (Track_Playlist).

4 Loading The Data

4.1 Overview

The data loading process is split into multiple steps.

1. The json files are read, and a single csv file containing all the data is created.
2. Specific columns from the csv file is used to create temporary csv files corresponding to each table of the schema.
3. Temporary tables are created, these tables do not have any constraints to make initial data loading easier.
4. The content of the temporary csv files is inserted into the corresponding temporary tables.
5. The temporary tables are used to create tables that match the actual schema.
6. The temporary csv files are deleted and the temporary tables are dropped.

The program files (along with this write up) can also be found here

4.2 converter.py

The converter reads the json files containing the dataset, and creates a single csv file using it.

Given a path to the folder containing input data files, the program will read and process each of the files one by one. For each file, the json file is converted into a python dictionary using json.load(), which is then used to create a pandas data frame. The data frame is used to create the "output_playlist_data.csv" file. For each files being processed after the first, the data is appended to the end of the csv file (as opposed to overwriting the csv file).

The resulting output csv file contains all of the attributes/columns.

4.3 database_initializer.py

The initializer first connects to a database with the passed in arguments as credentials.

All of the scripts that will be executed by the initializer are stored in the sql_scripts folder, and they are referenced through the scripts.py file, which is an enum class that stores the paths to each of the sql scripts. The enum

classes are used to make code easier to understand, for example, we can refer to 'sql_scripts/create_schema.sql' as just 'SCHEMA'.

The schema is created first by running the script create_schema.sql. This script is similar to the DDL statements seen in the prevous section (Relational Model), however, to make data loading easier, every attribute of every table is initialized as CHAR or VARCHAR type. This is because we use the COPY command when we load the data into the tables, and we treat every data entry from the files as strings. When a table is populated, it is altered so that its attributes are of the correct types (specified in the previous section). We only alter the table at the end because we can speed up the bulk loading process by initially loading the data without any constraints.

Then, to populate each of the tables, the csv file created by converter.py is read in as a pandas data frame. Here we have both table_columns.py and temp_files.py as enum classes to make code more readable. table_columns.py contains information on what columns are needed for a specific table, whereas temp_files.py contains information on the path of each of the temporary csv files to be created.

By specifying which columns of the pandas data frame we want, we create temporary csv files that matches a specific table. For example: the playlist table has columns 'pid', 'name', 'collaborative', and 'duration_ms', therefore, when creating the playlist temporary csv file, only those columns from the data frame are included.

We then copy the data from the temporary csv file into a temporary table with no constraints. Then load the actual tables with the temporary table. The reason behind this is because we can load data into the database more easily if there are no constraints, and maintain those constraints when we insert the data from the temporary table into the actual table. Note that the temporary csv files are deleted and the temporary tables are dropped after they are used.

For example, the following is the script used to populate playlist:

```
COPY temp_playlist
        FROM '/Users/vinoddalavai/LocalDocuments/
          CSCI620_BigData/Project/temp/temp_playlist.csv'
        DELIMITER E','
        CSV HEADER;

SELECT DISTINCT temp_playlist.playlistid, temp_playlist.name,
                temp_playlist.collab, temp_playlist.duration
INTO playlist
FROM temp_playlist;
```

```
ALTER TABLE playlist
ALTER COLUMN playlistid TYPE INT USING playlistid::INTEGER,
ALTER COLUMN collab TYPE BOOLEAN USING collab::BOOLEAN,
ALTER COLUMN duration TYPE INT USING duration::INTEGER;

ALTER TABLE playlist
ADD PRIMARY KEY (playlistid);

DROP TABLE temp_playlist;
```

A temp_playlist table is created using the COPY command and the temp_playlist.csv file. When inserting data into the actual playlist table, a constraint is that we only want distinct rows, which is specified in the SELECT INTO statement. We then ALTER the table to make sure that it matches the schema and, finally, we drop the temporary table.

This process is done for every table of the schema.

4.4 Enum Classes

The enum classes are very simple, as their main purpose is to define variable names to "stand-in" for otherwise long variables, which makes code easier to understand. They are mentioned in the previous subsection (4.3), but the actual code is also included here to more clearly show what they are.

scripts.py

```
from enum import Enum

class Scripts(Enum):
    SCHEMA = 'sql_scripts/create_schema.sql'
    PLAYLIST = 'sql_scripts/populate_playlist.sql'
    ARTIST = 'sql_scripts/populate_artist.sql'
    TRACK = 'sql_scripts/populate_track.sql'
    TRACK_PLAYLIST = 'sql_scripts/populate_track_playlist.sql'
```

table_columns.py

```
from enum import Enum

class TableColumns(Enum):
    PLAYLIST = ['pid', 'name', 'collaborative', 'duration_ms']
    ARTIST = ['artist_uri', 'artist_name']
    TRACK = ['track_uri', 'track_name', 'album_uri']
    TRACK_PLAYLIST = ['track_uri', 'pid', 'pos']
```

temp_files.py

```
from enum import Enum

class TempFiles(Enum):
    PLAYLIST = 'temp/temp_playlist.csv'
    ARTIST = 'temp/temp_artist.csv'
    TRACK = 'temp/temp_track.csv'
    TRACK_PLAYLIST = 'temp/temp_track_playlist.csv'
```

For example, when running a script, instead of specifying to run the scrip 'sql_scripts/create_schema.sql', the Scripts enum class allows us to refer to it as 'SCHEMA'. Similarly, instead of specifying the columns in the playlist table as a list, we can just refer to the columns as 'PLAYLIST'. The same concept applies to TempFiles.