

Project overview and description of input datasets

Document overview

This document answers the following three questions:

1. What is the objective and use case for this project?
2. How would the workflow be different, given a few hypothetical changes around scope and size?
3. What were the raw data inputs used for this project?
4. Which steps were taken to clean and prepare the data?
5. After analysis, what were the key takeaways from the data?

Contents

- I. [Project overview and objectives](#)
- II. [Modified workflow after hypothetical changes](#)
- III. [Raw data inputs](#)
- IV. [Steps for cleaning and preparation](#)
- V. [Key takeaways](#)

I. Project overview and objectives

For my capstone project, I've combined various data sources from Spotify to answer **four key questions** around hit songs. The four questions are:

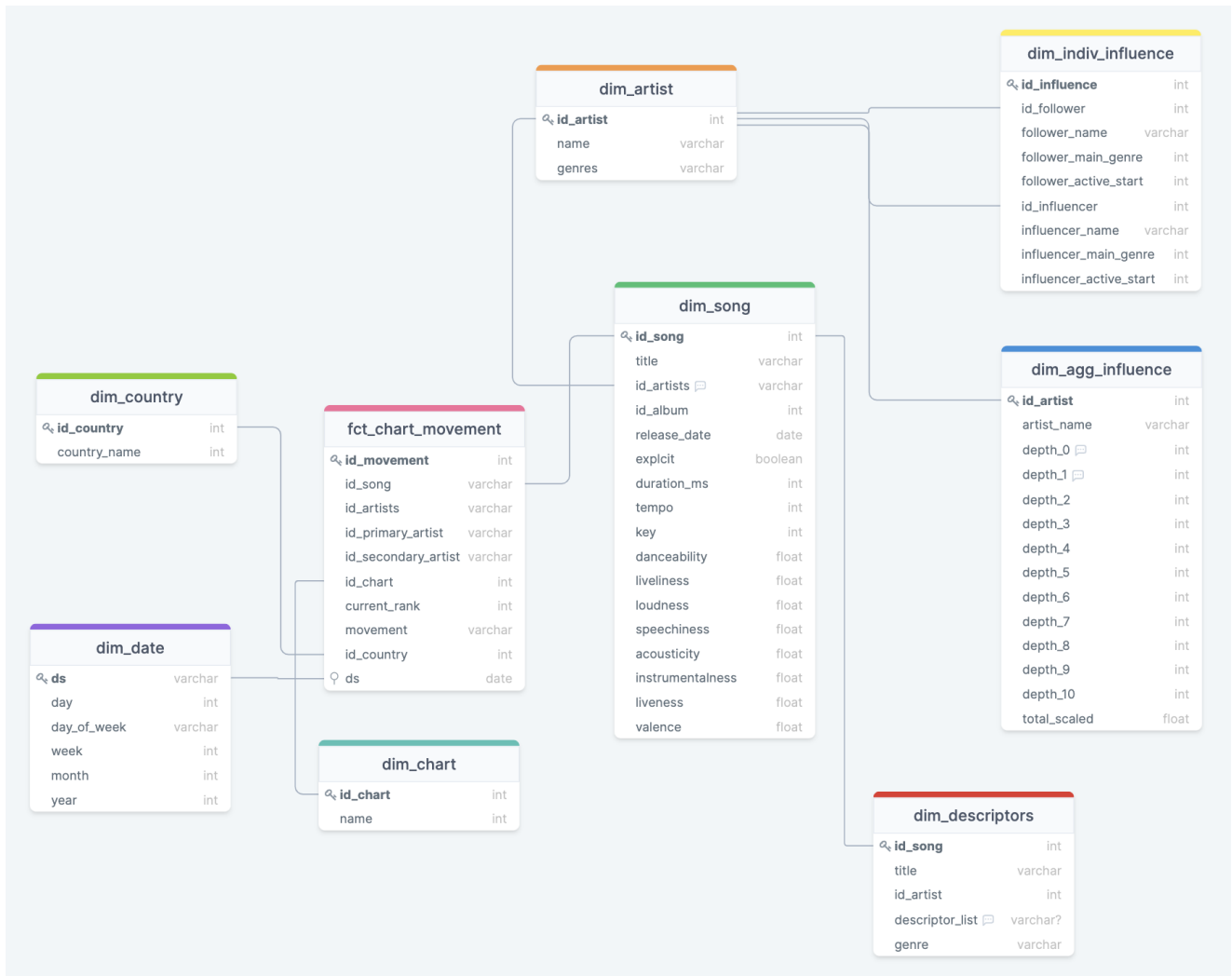
1. How do the musical characteristics of hit songs — including duration, explicit language, tempo, and danceability — differ when compared to all songs on Spotify?
2. Does this vary by country, or are the differences between pop songs and other songs universal across the world?
3. Which countries tend to be the first places where global hits will appear on their charts, and which countries tend to trail behind?
4. Which musical artists have had the biggest impact on the artists who are currently topping the charts?

This data could be of interest to many people, including songwriters, DJs, and those who work at record labels. For example, if an artist wanted to expand her fan base across new countries, she might look at the characteristics of popular songs there and adjust her music accordingly. A&R representatives could use it to keep track of big trends in the music industry when deciding which up-and-coming artists to sign to their labels. Lastly, Spotify itself could use this data when they are deciding which music to add to their different curated playlists.

Data models

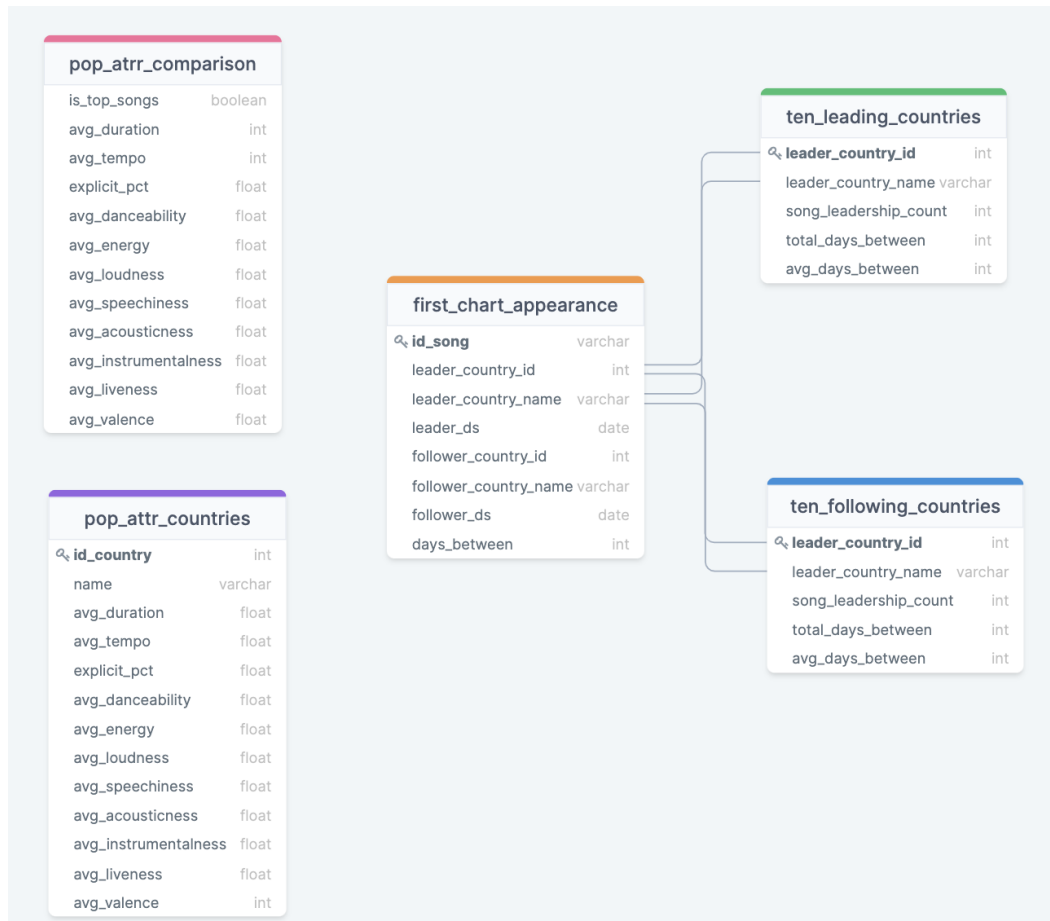
Based on my research questions, I ended up creating a data model that is somewhat reminiscent of a star schema, with *fact_chart_movement* being at the center and having some dimension tables linked via foreign keys. Although it is not at the center of the star schema per se, *dim_song* also has connections with a few different tables that are not connected to *fact_chart_movement*, namely *dim_artist*. The reason for this decision was because the artist information from the fact table is not as reliable as it is from the *dim_song* table, and we felt that artist information about a song more naturally links with the song table than it does with the fact table of all of the chart movements. Instead, it is safer to only get the song ID from that table and then use the *dim_song* information table to gather more information about each song.

The data model can be seen below:



In addition to the central data model depicted above, I created a separate model for the analytical tables. These are all tables derived from the core tables above, and are created specifically to answer the four research questions above. Instead of analysts needing to manually run the analytical queries anytime they needed the data, I decided to provide this already. That way, technical and non-technical users can easily query these analytical tables to look at the results using basic *SELECT * FROM table* queries instead.

The model for these can be seen below:



Technological and architectural decisions

The primary technologies used in this pipeline are Redshift, pure python, and a small amount of bash scripting. All of the analytical queries take place using Redshift, with python being used to orchestrate what should be run on Redshift, as well as defining and executing the various data quality checks specified by the user. The decision was made not to use spark or Airflow, which is explained below.

When deciding between a simpler technology (SQL on Redshift) and a more complex one (Spark), The simpler option should be chosen as the default unless it would not be sufficient. Based on the size of the data in my project and the types of transformations needed to answer the research questions, Redshift was sufficient across the board. Even with millions of records, no portions in the pipeline take more than a few minutes on Redshift, and writing pure SQL, rather than working with Sparks APIs, is more comfortable for many data analysts. As such, Redshift was perfectly sufficient for this project.

I opted not to use Airflow for this pipeline, although it would have set relatively well into the project. My main reasoning behind this decision was that the questions I was asking of the data are more periodical / ad-hoc in nature. While the results could be of interest to artists, record labels, etc., it is

not the type of thing one would expect to change substantially from day-to-day. An example of this might be daily pipelines to understand how one of spotify's A/B tests is going. In this case, data from each day is important because it can impact whether the A/B test is scaled up or down, and we would want a tool like Airflow to orchestrate the pipeline and run it every day. However, in my case, it is the type of analysis that can be run at any given point by a curious analyst, who only needs to call one shell script, but need not be executed on a rigid timetable. Based on how often music trend data changes, I believe the pipeline could be rerun every few weeks to see whether any major changes took place. Songs typically stay on top charts for at least a week, so running the pipeline every day would be excessive.

II. Modified workflow after hypothetical changes

In the project assignment, we were asked to answer how our pipeline would change under the following three scenarios:

1. If the data were increased by 100x
2. If the pipelines were around on a daily basis by 7 a.m.
3. If the database needed to be accessed by 100+ people

If the data were increased by 100x

In this case, the fundamental project architecture would not need to change, because Redshift is built to handle very large amounts of data, and the same goes for storage on S3. Even with 100x more data, the tables would still be less than 1 billion records. The most important change would be to increase the size of the Redshift cluster, both vertically and horizontally, meaning adding more nodes and increasing the computational resources of each node. This would ensure that the data pipeline is able to run in a timely fashion.

Another option would be to rewrite the pipeline to leverage spark instead, but this would require much more work and is not strictly necessary, even if there were 100x more data, so that there are more, powerful nodes to process the data in parallel.

If the pipelines were around on a daily basis by 7 a.m.

Given this requirement, Airflow would be a good solution. We could schedule the DAG to run every morning, rather than having to manually run it ourselves.

If the database needed to be accessed by 100+ people


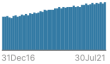

The most important change in this case would be similar to the first recommendation, which is to scale up the Redshift cluster. The current number of nodes in the cluster would be easily overwhelmed if it were being accessed by 100+ people. Another modification might be to add some kind of locking system so that the tables cannot be queried by all of these people while they are in the process of being added to, but this is something else that Redshift can handle automatically with STV_LOCKS. However, this can create bottlenecks for productivity and is not always recommended.

III. Raw data inputs

List of input datasets

- [Top_charts.csv](#)
- [1m_songs_features.csv](#)
- [90000_song_adjectives.csv](#)
- [Influence_data.csv](#)
- [artist_uri_mapping](#)

top_charts.csv

▲ title str	# rank int	📅 date date	▲ artist str	🔗 url str	🌐 region category	▲ chart category	▲ trend category	📊 streams int
154626 unique values			Ed Sheeran 2% Billie Eilish 1% Other (23393479) 98%	203788 unique values		top200 78% viral50 22%	MOVE_DOWN 43% MOVE_UP 37% Other (4718949) 20% Other (18658994) 78%	[null] 1003 78%
Sirenas	1	2017-08-01	Taburete	https://open.spotify.com/track/4un0evr9C2Ng19aJXrp0y	Andorra	viral50	SAME_POSITION	
Something Just Like This	2	2017-08-01	The Chainsmokers, Coldplay	https://open.spotify.com/track/68UXPb4LETWmr3iAEQktW		viral50	SAME_POSITION	
Bella y Sensual	3	2017-08-01	Romeo Santos, Daddy Yankee, Nicky Jam	https://open.spotify.com/track/8ERBk7qVqveCaBWTiYCr13	Andorra	viral50	SAME_POSITION	
Deja Que Te Bese	4	2017-08-01	Alejandro Sanz, Marc Anthony	https://open.spotify.com/track/8sy98ZF11LvYHnGRXfga	Andorra	viral50	SAME_POSITION	
Have You Ever Seen The Rain	5	2017-08-01	Creedence Clearwater Revival	https://open.spotify.com/track/2LwezPeJhN4A8uSB8GTAU	Andorra	viral50	SAME_POSITION	
Soldadito marino	6	2017-08-01	Fito y Fitipaldis	https://open.spotify.com/track/8eYborSuxUe5gmeWY69GZ	Andorra	viral50	SAME_POSITION	
I Don't Want to Miss a Thing - From the Touchstone film, "Armageddon"	7	2017-08-01	Aerosmith	https://open.spotify.com/track/8c1ghntWjKDTQ8C8s99sq	Andorra	viral50	SAME_POSITION	
Someone Like You	8	2017-08-01	Adele	https://open.spotify.com/track/4kfl10fjdZJ84ot2ioixTB	Andorra	viral50	MOVE_UP	

- **Description:** Each row represents one song being on either the 'Top 200' or 'Viral 50' Spotify playlist in a certain region on a given day
- Different countries have different top playlists. Spotify publishes this data every few days, so it's not exactly each day but similar idea
- Doesn't have a column for the URI but has it in the URL so we can just do a regex
- Around 3 million rows
- <https://www.kaggle.com/dhruvildave/spotify-charts>

1m_songs_features.csv

id	name	album	album_id	artists	artist_ids	track_num...	disc_num...	explicit	danceability
71meHLHB4nmXzuXc0HDjk	Testify	The Battle Of Los Angeles	2e1a0myWFgoHuttJytCxcgX	['Rage Against The Machine']	['2d0h0y0Q5ynDBnkvAbJKORj']	1	1	False	0.47
1wsR4tfRRtWYEp10q22o8	Guerrilla Radio	The Battle Of Los Angeles	2e1a0myWFgoHuttJytCxcgX	['Rage Against The Machine']	['2d0h0y0Q5ynDBnkvAbJKORj']	2	1	True	0.599
1hr0fIFK2qRG3f3RF70pb7	Calm Like a Bomb	The Battle Of Los Angeles	2e1a0myWFgoHuttJytCxcgX	['Rage Against The Machine']	['2d0h0y0Q5ynDBnkvAbJKORj']	3	1	False	0.315
21bASgTS0D07MTuLAX1TW0	Mic Check	The Battle Of Los Angeles	2e1a0myWFgoHuttJytCxcgX	['Rage Against The Machine']	['2d0h0y0Q5ynDBnkvAbJKORj']	4	1	True	0.44
1M0TnpY0Z6fcMQc56Hd07T	Sleep Now In the Fire	The Battle Of Los Angeles	2e1a0myWFgoHuttJytCxcgX	['Rage Against The Machine']	['2d0h0y0Q5ynDBnkvAbJKORj']	5	1	False	0.426
2LXPnLSMAauNJfnC581SqY	Born of a Broken Man	The Battle Of Los Angeles	2e1a0myWFgoHuttJytCxcgX	['Rage Against The Machine']	['2d0h0y0Q5ynDBnkvAbJKORj']	6	1	False	0.298
3moeHk8eIajvUEzVocXukf	Born As Ghosts	The Battle Of Los Angeles	2e1a0myWFgoHuttJytCxcgX	['Rage Against The Machine']	['2d0h0y0Q5ynDBnkvAbJKORj']	7	1	False	0.41700000000000000000

- **Description:** has a little more detail about 1 million songs, so probably almost all of the ones above. Has id, artist and album info, if it's explicit, and danceability. The screenshot wasn't able to fit most of them but there are a few dozen other attributes about each song, which is amazing! Time signature, Tempo, duration, liveliness, release year, node, loudness, acousticity, etc. A ton I can do with that and the one above!
- Despite the large size of this data set, it only captures a fraction of Spotify's approximately 70 million songs
- <https://www.kaggle.com/rodolfofigueroa/spotify-12m-songs>

90000_song_adjectives.csv

lastfm_url	track	artist	seeds	number_of...	valence_ta...	arousal_tags	dominanc...	mbid	spotify_id
https://www.last.fm/music/eminem/_/%2527till%2bi%2bcollapse	'Till I Collapse	Eminem	['aggressive']	6	4.55	5.273124999999999	5.698625	cab93def-26c5-4fb8-bedd-26ec4c1619e1	4xx0aSrKexMc1UuogZKVTS
https://www.last.fm/music/metallica/_/st.%2banger	St. Anger	Metallica	['aggressive']	8	3.71	5.832999999999999	5.427250000000000	727a2529-7ee8-4868-aef6-7959884895cb	3f0c9x061KJBhz435mInIH
https://www.last.fm/music/rick%2bross/_/speedin%2527	Speedin'	Rick Ross	['aggressive']	1	3.08	5.87	5.49		3Y96xd4Ce0J47dc a1LrEC8
https://www.last.fm/music/m.i.a._/_/bamboo%2bbanga	Bamboo Banga	M.I.A.	['aggressive', 'fun', 'sexy', 'energetic']	13	6.555071428571428	5.537214285714286	5.691357142857143	99dd2c8c-e7c1-413e-8ea4-4497a80ffa18	6tqFC1D10phJKCwrjVzPmg
https://www.last.fm/music/dope/_/die%2bm%2bdie	Die MF Die	Dope	['aggressive']	7	3.771176470588235	5.348235294117648	5.441764705882353	b9eb3484-5e0e-4698-ab5a-ca91937032a5	5bU4KX47KqtDKKaLM4QCzh
https://www.last.fm/music/drowning%2bpool/_/step%2bup	Step Up	Drowning Pool	['aggressive']	9	2.971388888888889	5.537499999999999	4.726388888888889	49e7b4d2-3772-4301-ba25-3cc46ceb342e	4Q1w4Ryyi8K0NxxaF10QC1K

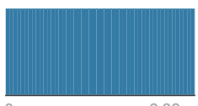
- **Description:** For about 90,000 songs, this has some adjectives pulled from Last FM about each song, like 'aggressive', 'energetic', etc. (adjectives not found in the previous data source), and scores on the valence / arousal / dominant dimensions
- One really nice thing is that it has Spotify URIs for each song, so it's easy to join!
- <https://www.kaggle.com/cakiki/muse-the-musical-sentiment-dataset>

influence_data.csv

Δ follower_n...	Δ follower_m...	# follower_a...	∞ influencer_...	Δ influencer...	Δ influencer...	# influencer_...	∞ follower_id	Δ follower_n...
Slightly Stoopid	Pop/Rock	1990	785380	Lee "Scratch" Perry	Reggae	1960	20729	Slightly Stoopid
Slightly Stoopid	Pop/Rock	1990	786613	Dead Kennedys	Pop/Rock	1970	20729	Slightly Stoopid
Slightly Stoopid	Pop/Rock	1990	796045	Fishbone	Pop/Rock	1970	20729	Slightly Stoopid
Slightly Stoopid	Pop/Rock	1990	883318	Red Hot Chili Peppers	Pop/Rock	1980	20729	Slightly Stoopid
Slightly Stoopid	Pop/Rock	1990	1416172	Jack Johnson	Religious	1930	20729	Slightly Stoopid
Sizzla	Reggae	1990	16944	Alpha Blondy	Reggae	1980	20740	Sizzla
Sizzla	Reggae	1990	129817	Shabba Ranks	Reggae	1980	20740	Sizzla
Sizzla	Reggae	1990	138063	Barrington Levy	Reggae	1970	20740	Sizzla

- **Description:** each row represents one of the musical influences on another artist. See above. 5,000 different artists, and each artist has 5-10 influencers, so the entire dataset is about 40,000 rows
- No artist IDs, so we'll have to make the artist names lowercase and do a join
- <https://www.kaggle.com/ironicninja/icm-problem-d>

Artist_uri mapping

#	name	genres	spotify_uri
 0 6.69m	5866470 unique values	<div> <div></div> 95% </div> <div> <div></div> 0% </div> <div> <div></div> 5% </div>	6690302 unique values
0	Liszt Jenő Cimbalom Project	<div> <div></div> </div>	spotify:artist:0NxUJYwYtXrUHNwQznLwL
1	Stewie	<div> <div></div> </div>	spotify:artist:0VLF0xJlBB2LYNttSejrEv
2	G.O'd	<div> <div></div> </div>	spotify:artist:0ijndZDbEsSTgDS8qyw6fI
3	Zico Aka Lost Identity	<div> <div></div> </div>	spotify:artist:0xB16BIgyn5Uy5TtNGVEew
4	Suburban Drag	<div> <div></div> </div>	spotify:artist:3FxfEhNrYCMiZl6f9ajrbE
5	Ya nos vamos	<div> <div></div> </div>	spotify:artist:3y4Utgj0pBG1ktIRKcwz5No

- **Description:** Has the name and mapping 66 million artists. Good for joining
- <https://www.kaggle.com/toluwafayemi/66m-spotify-artists>

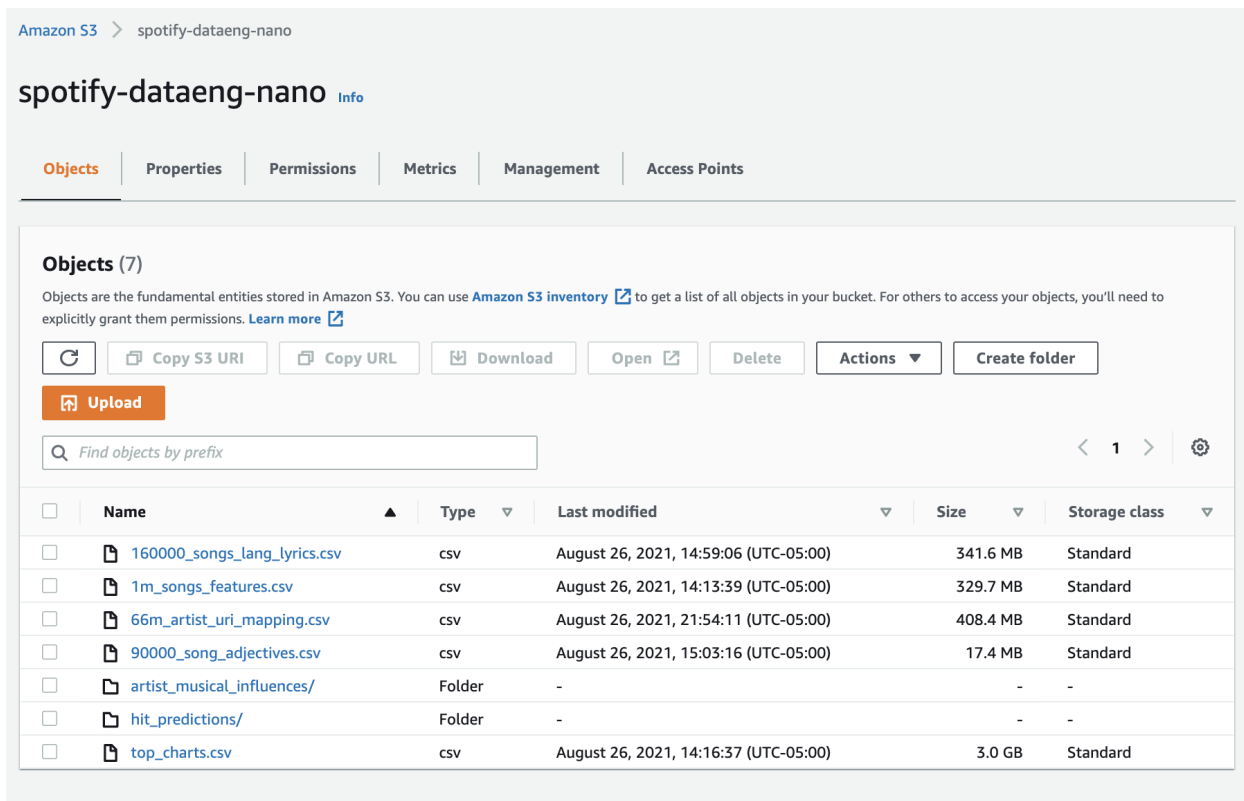
IV. Steps for cleaning and preparation

Below, I have documented many of the steps needed to load, clean, and transform the data for analysis, as well as a few data quality issues that came up along the way.

High-level steps

- I. Load data into S3
- II. Transfer data from S3 into redshift staging tables
- III. Move data from staging tables into cleaned star schema
- IV. Conduct analysis, which also uncovered data quality issues

Load data into S3



Amazon S3 > spotify-dataeng-nano

spotify-dataeng-nano [Info](#)

Objects | Properties | Permissions | Metrics | Management | Access Points

Objects (7)

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

[Refresh](#) [Copy S3 URI](#) [Copy URL](#) [Download](#) [Open](#) [Delete](#) [Actions](#) [Create folder](#)

[Upload](#)

<input type="checkbox"/>	Name	Type	Last modified	Size	Storage class
<input type="checkbox"/>	160000_songs_lang_lyrics.csv	csv	August 26, 2021, 14:59:06 (UTC-05:00)	341.6 MB	Standard
<input type="checkbox"/>	1m_songs_features.csv	csv	August 26, 2021, 14:13:39 (UTC-05:00)	329.7 MB	Standard
<input type="checkbox"/>	66m_artist_uri_mapping.csv	csv	August 26, 2021, 21:54:11 (UTC-05:00)	408.4 MB	Standard
<input type="checkbox"/>	90000_song_adjectives.csv	csv	August 26, 2021, 15:03:16 (UTC-05:00)	17.4 MB	Standard
<input type="checkbox"/>	artist_musical_influences/	Folder	-	-	-
<input type="checkbox"/>	hit_predictions/	Folder	-	-	-
<input type="checkbox"/>	top_charts.csv	csv	August 26, 2021, 14:16:37 (UTC-05:00)	3.0 GB	Standard

The first step was to load all of the different raw data files, collected from Kaggle, into an S3 bucket. Some of the datasets were actually collections of datasets, so I decided to store those inside of sub-directories in the main bucket. All of the Initial input files were csv's, which made the uploading process fairly smooth.

Transfer data from S3 into Redshift staging tables

```
create_staging_tables.sql
1 CREATE TABLE IF NOT EXISTS staging_charts (
2     title VARCHAR NOT NULL
3     , rank VARCHAR -- Should be int but had letters in there
4     , date VARCHAR -- Should be date but had invalid characters
5     , artists VARCHAR
6     , url VARCHAR
7     , region VARCHAR
8     , chart VARCHAR
9     , trend VARCHAR
10    , streams VARCHAR -- Should be int but had letters in there
11 )
12
13
14 -- Like above, final table won't be VARCHAR but staging column had some records with letters
15 CREATE TABLE IF NOT EXISTS staging_songs_full (
16     id VARCHAR
17     , name VARCHAR
18     , album VARCHAR
19     , album_id VARCHAR
20     , artists VARCHAR
21     , artist_ids VARCHAR
22     , track_number VARCHAR
23     , disc_number VARCHAR
```

For each of the CSV files above, I created a new staging table and copied the data into it. See below for an example COPY statement:

```
staging_songs_full_copy = ""
COPY {} FROM 's3://spotify-dataeng-nano/1m_songs_features.csv' TRUNCATECOLUMNS
IAM_ROLE {}
CSV
IGNOREHEADER 1
"".format('staging_songs_full', ARN)
```

In the first screenshot, one might notice that every column type in the staging tables is of type VARCHAR, even though some of the columns, e.g., *streams*, should be INTEGER. The justification for using VARCHAR in this initial stage is that most of the csv columns that should be INTEGER contained at least one value that is not a valid integer, like 'ab'. By making the staging table column be of type INTEGER, the entire COPY statement fails if there are any non-INTEGER values. I decided to make the staging tables have a flexible column type of VARCHAR, and then later on, filter out records that have invalid values in these columns.

Move data from staging tables into cleaned star schema

Much of the cleaning needed to load the data into the star schema cleaning took place within *WHERE* clauses in this section. For example, only including chart movement events with a numeric *rank* column, and only using songs where the release date was structured like *####-##-##*. Some of the data was malformed so these types of filters helped remove those records.

I had to add *AND LENGTH(artist_ids) < 150* because some of the songs had so many artists that they did not fit into the column, which led to artist ID columns that would be cut off. Since the array did not complete, meaning there was no right square bracket, my UDF could not parse the array correctly, so these records were removed. The artist IDs are approximately 24 characters long, so this only removed songs with seven or more artists. An exceedingly small number of songs featured more artists than could fit in the column, (about 0.2%) so most data was maintained.

After loading the 90,000 songs with adjectives, I noticed that only about 7,600 of them linked to a song from the *dim_song* table when doing an inner join. This is because *dim_song*, despite its large size, only captures a fraction of Spotify's approximately 70 million songs, so most songs with adjectives in the table are not included in *dim_song*. In general, whenever I would be joining tables to insert into a final dimensional table, I always first made sure that a significant number of rows would not be unintentionally dropped due to the join.

As seen below, there were about 6.7 million distinct artist IDs, but only 5.9 million distinct artists names, which is a challenge because some of the other tables, like the influence tables, only have the artist name and not their artist IDs, so I need to know which ID I should be joining with. For example, multiple bands being named 'The Eagles'.

```
1  %%sql
2  SELECT count(distinct id_artist)
3  FROM dim_artist
4  UNION
5  SELECT count(distinct name)
6  FROM dim_artist

* postgresql://capstoneuser:***@capstone-proj.c0d
2 rows affected.

count
6690302
5866304
```

My solution was to join the *dim_artist* table with the *fact_chart_movement* table when creating *dim_artist*. That way, the artist table would only have artists who show up at least once in the chart. This is a better solution in general, because all of my analysis will focus around artists who were in the

top charts and what influence they had on other artists, so it's not ideal but okay to exclude the others. See below for updated results:

```
1 %%sql
2 SELECT count(distinct id_artist)
3 FROM dim_artist
4 UNION
5 SELECT count(distinct name)
6 FROM dim_artist

* postgresql://capstoneuser:***@capstone-proj.c0df8
1 rows affected.

count
150
```

Another idea I tried was to include every artist from the influence table in the *dim_artist* table. Below shows the results from that query:

```
1 %%sql
2 SELECT count(distinct id_artist)
3 FROM dim_artist
4 UNION
5 SELECT count(distinct name)
6 FROM dim_artist

* postgresql://capstoneuser:***@capstone-proj.c0df8unkd
2 rows affected.

count
5215
13652
```

This runs into the same issue, where there are multiple artists with the same name but different IDs. Since there was no ground truth that I could use to determine which of the 3 artists named *The Eagles* had the idea of the popular band, I opted to discard this approach. The most important artists are the 150 artists who appeared in the top charts.

Another challenge I ran into was when preparing the dimensional tables that pertain to the influence that musical artists have had on one another. Each row represents one influence that an artist has had on another artist. In this case, there were 42,771 influencers. However, I only wanted to include influences where the influencer or follower were represented in the top charts, since this is the basis of my analysis (and those are the only artists included in *dim_artist*). If I only included influences where the follower was in the top charts, that would cut it from 42,771 influences to only 357, and if I only included influences where the influencer was in the top charts, that would cut it from 42,771 influences to only 610. See below:


```

1 %%sql
2 SELECT count(*)
3 FROM staging_all_influences;

* postgresql://capstoneuser:***@capstone-proj.c0df8unkdobb.us-west-2.reds
1 rows affected.

count
42771

1 %%sql
2 SELECT count(*)
3 FROM staging_all_influences sai
4 JOIN dim_artist da
5     ON sai.follower_name = da.name
6
7 UNION
8
9 SELECT count(*)
10 FROM staging_all_influences sai
11 JOIN dim_artist da
12     ON sai.influencer_name = da.name

* postgresql://capstoneuser:***@capstone-proj.c0df8unkdobb.us-west-2.reds
2 rows affected.

count
357
610

```

However, I wanted influences where **either** the influencer for the follower were in the charts. This required a more complex join that joins on follower name, if there's a match, or influencer name if there's a match there and no match on the follower name. This correctly returned 967 results, the combination of 357 and 610 above. See below:

```

1 %%sql
2 SELECT count(*)
3 FROM staging_all_influences sai
4 JOIN dim_artist da
5     ON CASE
6         WHEN sai.follower_name = da.name THEN 1
7         WHEN sai.influencer_name = da.name THEN 1
8         ELSE 0 END = 1

* postgresql://capstoneuser:***@capstone-proj.c0df8unkdobb
1 rows affected.

count
967

```

One other issue I noticed after loading the aggregate influence data was that there were duplicates for each artist, because of the way I did my join. See below:

```

1 %%sql
2 SELECT *
3 FROM dim_agg_influence
4 ORDER BY total_scaled DESC
5 LIMIT 4;

```

* postgresql://capstoneuser:***@capstone-proj.c0df8unkdobb.us-west-2.redshift.amazonaws.com:5432

4 rows affected.

influencer_id	influencer_name	depth_0	depth_1	depth_2	depth_3	depth_4	depth_5	depth_6	depth_7	depth_8	depth_9	depth_10	total_scaled
234	Elvis Presley	1	167	1975	3632	4162	4415	4533	4598	4634	4847	4880	4884
234	Elvis Presley	1	167	1975	3632	4162	4415	4533	4598	4634	4847	4880	4884
169	Cab Calloway	1	28	396	2594	4634	4847	4880	4884	4884	4884	4884	4884
169	Cab Calloway	1	28	396	2594	4634	4847	4880	4884	4884	4884	4884	4884

My solution was to use the *ROW_NUMBER()* function to de-duplicate the data. See below for a portion of the code:

```

INSERT INTO dim_agg_influence

WITH duplicate_df AS (
SELECT
    CAST (influencer_id AS INT)
    , influencer_name
    , CAST (depth_0 AS INT)
    , CAST (depth_1 AS INT)
    , CAST (depth_2 AS INT)
    , CAST (depth_3 AS INT)
    , CAST (depth_4 AS INT)
    , CAST (depth_5 AS INT)
    , CAST (depth_6 AS INT)
    , CAST (depth_7 AS INT)
    , CAST (depth_8 AS INT)
    , CAST (depth_9 AS INT)
    , CAST (depth_10 AS INT)
    , CAST (total AS FLOAT)
    , ROW_NUMBER() OVER (PARTITION BY influencer_name ORDER BY influencer_id) AS row_num
FROM staging_influence_depth sid
JOIN dim_artist da
    ON sid.influencer_name = da.name
)

SELECT
    influencer_id
    , influencer_name
    , depth_0
    , depth_1
    , depth_2
    , depth_3
    , depth_4
    , depth_5
    , depth_6
    , depth_7
    , depth_8
    , depth_9
    , depth_10
    , total
FROM duplicate_df
WHERE
    row_num = 1;

```

This approach worked successfully. After applying that filter, you can see below that we no longer have the issue of duplicate data:

```
1 %%sql
2 SELECT *
3 FROM dim_agg_influence
4 ORDER BY total_scaled DESC
5 LIMIT 4;
```

* postgresql://capstoneuser:***@capstone-proj.c0df8unkdobb.us-west-2.redshift.amazonaws.com:5439/capstone?sslmode=require

4 rows affected.

influencer_id	influencer_name	depth_0	depth_1	depth_2	depth_3	depth_4	depth_5	depth_6	depth_7
234	Elvis Presley	1	167	1975	3632	4162	4415	4533	4591
169	Cab Calloway	1	28	396	2594	4634	4847	4880	4881
679	The Clash	1	137	897	1670	1972	2284	2541	2811
927	Willie Nelson	1	67	487	1453	2495	3143	3532	3811

Data quality issues uncovered when conducting analysis

After loading all of the data into the star schema, I started to create analytical tables that would allow for different research questions to be answered. For example, common characteristics of top hits and whether this varies by region.

When preparing these analytical tables, a few more data quality issues were uncovered. The first was when looking at common characteristics of top hits. See below for the results of a query to determine which song spent the most days on the top charts globally:

```

SELECT fcm.id_song, count(*), title, ds.id_artists
FROM fct_chart_movement fcm
JOIN dim_song ds
    ON fcm.id_song = ds.id_song
WHERE id_chart = 102
GROUP BY fcm.id_song, ds.title, ds.id_artists
ORDER by count(*) DESC
LIMIT 15;

```

* postgresql://capstoneuser:***@capstone-proj.c0df8unkdobb.us-west-2.redshift.amazonaws.com:5439/dev
15 rows affected.

id_song	count	title	id_artists
0FE9t6xYkqWXU2ahLh6D8X	63215	Shape of You	['6eUKZXaKkcviH0Ku9w2n3V']
3dCC7aodARJimlvDnlh1mo	47181	Someone You Loved	['4GNC7GD6oZMSxPGyXy4MNB']
2XU0oxnq2qxCPomAAuJY8K	40807	Dance Monkey	['2NjfBq1NflQcKSeiDooVjY']
5ZULALlmTm80tzUbYQYM9d	40807	Dance Monkey	['2NjfBq1NflQcKSeiDooVjY']
4TK18Te3S79HAL2K4pcPBV	39142	bad guy	['6qqNVTkY8uBg9cP3Jd7DAH']
1nb87LGeIO9Uh4BBBY4iaj	37047	Say You Won't Let Go	['4lWBUUAFIplrNtaOHcJPRM']
5uCax9HTNlZGyblStD3vDh	37047	Say You Won't Let Go	['4lWBUUAFIplrNtaOHcJPRM']
3BVgrFWuH01GmCUy9Y2EE8	36350	Señorita	['7n2wHs1TKAczGzO7Dd2rGr', '4nDoRrQiYLoBzwC5BhVJzF']
4U54mWY0leFsVzjNXys3pH	35986	Don't Start Now	['6M2wZ9GZgrQXHCffjv46we']
2wVytz13toqr1i9bceYCo8	35986	Don't Start Now	['6M2wZ9GZgrQXHCffjv46we']
6Wrl0LAC5M1Rw2MnX2ZvEg	35986	Don't Start Now	['6M2wZ9GZgrQXHCffjv46we']
3PflrDoz19wz7qK7tYeu62	35986	Don't Start Now	['6M2wZ9GZgrQXHCffjv46we']
1Zh1T8dryY8RC3ibopOqTu	35986	Don't Start Now	['6M2wZ9GZgrQXHCffjv46we']
1AVtceapuF36oZqI9gzp0o	35986	Don't Start Now	['6M2wZ9GZgrQXHCffjv46we']
1dNIETp7AY3oDAKCGg2XkH	33948	Something Just Like This	['69GGBxA162lTqCwzJG5jLp', '4gzpq5DPGxSnKTe4SA8HAU']

For some of the songs, like “Dance Monkey” and “Don’t Start Now”, there are multiple instances of the same song, by the same artists, and with the same number of days on the charts, but with different song IDs. I noticed that this was only an issue for approximately 10,000 of the songs in the *dim_song* table.

To fix the issue, I used the query below. The main trick here is that if a song has the same exact name, artists and release date, then it is likely a duplicate, perhaps being a single and then the same song from an album. In cases where alternate versions of a song are released, like with acoustic versions, this is usually included in the title. See below for the query:

```

323 INSERT INTO dim_song
324
325 WITH duplicate_songs AS (
326     SELECT *, ROW_NUMBER() OVER (PARTITION BY name, artist_ids, release_date ORDER BY id) AS row_num
327     FROM staging_songs_full
328 )
329
330 SELECT
331     id
332     , name
333     , artist_ids
334     , album_id
335     , CAST(release_date AS DATE)
336     , CASE WHEN explicit = 'True' then TRUE else FALSE END
337     , CAST(duration_ms AS INT)
338     , ROUND(CAST(tempo AS FLOAT), 0)
339     , CAST(key AS INT)
340     , ROUND(CAST(time_signature AS FLOAT), 1)
341     , ROUND(CAST(danceability AS FLOAT), 2)
342     , ROUND(CAST(energy AS FLOAT), 2)
343     , ROUND(CAST(loudness AS FLOAT), 2)
344     , ROUND(CAST(speechiness AS FLOAT), 2)
345     , ROUND(CAST(acousticness AS FLOAT), 2)
346     , ROUND(CAST(instrumentalness AS FLOAT), 2)
347     , ROUND(CAST(liveness AS FLOAT), 2)
348     , ROUND(CAST(valence AS FLOAT), 2)
349 FROM duplicate_songs
350 WHERE
351     release_date ~ '[0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]'
352     AND row_num = 1;

```

The last step is to revise the *fct_chart_movement* table creation to join with the *dim_songs* table, so that duplicates are not included as chart movement events. It reduced the number of records in the *fct_chart_movements* table from over 8.3 million to 7.6 million.

After the fix, there were no longer any duplicates when looking at the songs that spent the most time in the top charts. See below:

```

%%sql
SELECT fcm.id_song, count(*), title, ds.id_artists
FROM fct_chart_movement fcm
JOIN dim_song ds
    ON fcm.id_song = ds.id_song
WHERE id_chart = 102
GROUP BY fcm.id_song, ds.title, ds.id_artists
ORDER by count(*) DESC
LIMIT 15;

```

* postgresql://capstoneuser:***@capstone-proj.c0df8unkdobb.us-west-2.redshift.amazonaws.com:5432/dev

15 rows affected.

id_song	count	title	id_artists
2XU0oxnq2qxCpomAAuJY8K	12824	Dance Monkey	['2NjfBq1NflQcKSeiDooVjY']
1ZxELEYmloUoTfFVHqpiTe	8002	Sweet but Psycho	['4npEfmQ6YuiwW1GpUmaq3F']
5HIksDGugWk5SM7q9Ua7hY	7963	death bed (coffee for your head)	['6bmlMHgSheBauioMgKv2tn', '35l9BRT7MXmM8bv2WDQiyB']
3fefQTqeZ0dkOAvoJwHfko	7106	Calma - Remix	['4QVBYiaglaa6ZGSPMbybpy', '329e4yvlujlSKGKz1BZZbO']
1KnrEzLrdw6Gx2iAmGfYMh	6912	Falling	['7uaim6Pw7xplS8Dy06V6pT']
3dCC7aodARJimlvDnlh1mo	6528	Someone You Loved	['4GNC7GD6oZMSxPGyXy4MNB']
3Dv1eDb0MEgF93GpLXLucZ	6378	Say So	['5cj0ILjcoR7YOSnhnX0Po5']
0E9ZjEayAwOXZ7wJC0PD33	6335	In My Mind	['3v6Ji4uoWtKRkhuDUaxi9n', '10AjDaKgg00KCUYqDe68un']
51Fjme0JiitpyXKuyQiCDo	6187	Lalala	['6USMTwO0MNDnKte5a5h0xx', '41X1TR6hrK8Q2ZCpp2EqCz']

V. Key takeaways

In the section, I will walk through some of the key findings from the analytical queries. Since I structured the tables to be made for analysis, all of the queries in the section are very simple.

<pre>SELECT * FROM pop_attr_comparison;</pre> <p>* postgresql://capstoneuser:***@capstone-proj.c0df8unkdobb.us-west-2.redshift.amazonaws.com:5439/dev 2 rows affected.</p>										
is_top_songs	avg_duration	avg_tempo	explicit_pct	avg_danceability	avg_energy	avg_loudness	avg_speechiness	avg_acousticness	avg_instrumentalness	avg
TRUE	206171	122	0.250	0.682	0.632	-6.584	0.107	0.233	0.028	
FALSE	249290	117	0.074	0.495	0.511	-11.73	0.085	0.445	0.285	

In this table, the top row represents popular songs that were in the global top 1,000, and the bottom represents the broader Spotify library. Also, it's worth noting that only two columns on the right did not fit into the screenshot, but there was no significant difference between the two. This is not every song on Spotify per se, but it is a sample with multiple million songs.

There are a few clear and interesting differences between popular songs and all songs in general. Popular songs tend to be much shorter, a bit higher tempo, higher energy, louder, and more danceable. Another interesting finding was that 25% of popular songs are explicit, versus only 7.4% for songs in general. However, this likely pertains at least in part due to the difference in instrumentalness, as seen in the last column. Many more songs in general tend to be fully instrumental and without any vocals, which is quite rare for popular songs. Therefore, it makes sense that more pop songs are more explicit than the broader Spotify library, because instrumental songs don't have the opportunity to do so.

As a musician and close follower of music, a lot of these findings above are consistent with my own observations, which provided more confidence that the data pipeline was done correctly, but seeing specific numbers and the magnitude of differences was still quite interesting.

<pre>SELECT * FROM pop_attr_countries LIMIT 14;</pre>									
<pre>* postgresql://capstoneuser:***@capstone-proj.c0df8unkdobb.us-west-2.redshift.amazonaws.com:5439/dev 14 rows affected.</pre>									
id_country	name	avg_duration	avg_tempo	explicit_pct	avg_danceability	avg_energy	avg_loudness	avg_speechiness	avg_acousticness
163	Poland	206117	122	0.276	0.672	0.644	-6.919	0.105	0.24
128	Honduras	213840	122	0.209	0.677	0.657	-6.246	0.097	0.228
139	Japan	224225	122	0.148	0.65	0.703	-5.702	0.085	0.176
110	Denmark	194419	124	0.378	0.7	0.625	-6.875	0.123	0.227
106	Colombia	214682	123	0.220	0.69	0.654	-6.323	0.105	0.247
113	Guatemala	208276	122	0.185	0.7	0.641	-6.281	0.098	0.249
150	India	204896	120	0.242	0.696	0.6	-7.255	0.101	0.289
102	Bulgaria	205678	122	0.318	0.698	0.638	-6.366	0.122	0.208
100	Australia	202632	121	0.286	0.675	0.637	-6.688	0.118	0.22
159	Philippines	208741	120	0.212	0.662	0.6	-6.853	0.091	0.28
175	Taiwan	209219	123	0.181	0.639	0.621	-6.76	0.088	0.256
103	Andorra	225887	119	0.112	0.65	0.643	-7.081	0.073	0.237
117	Ireland	205469	121	0.270	0.673	0.636	-6.661	0.115	0.224
171	Spain	211445	122	0.246	0.687	0.659	-6.298	0.109	0.254

Above, we can see the characteristics of popular songs from different countries across the world. Interestingly, there are some characteristics that tend to be shared across all pop songs, but there are other characteristics that differ very much by country. For example, the average tempo is quite consistent. In almost every country, the average is between 120-123 BPM. Other characteristics like energy and danceability also appear to be fairly similar across countries.

On the other hand, there's a wide range in the percentage of popular songs that are explicit. Some countries like Andorra and Japan are only 11% and 15% explicit respectively, whereas Bulgaria is 32% and Denmark is almost 38% explicit! Much of this comes down to countries having different cultural norms and different amounts of tolerance for profanity, but it is one of the most clear distinctions from country to country.

<pre>SELECT * FROM ten_leading_countries ORDER BY song_leadership_count DESC;</pre>					
<pre>* postgresql://capstoneuser:***@capstone-proj.c0df8unkdobb.us-west-2.redshi 10 rows affected.</pre>					
leader_country_id	leader_country_name	song_leadership_count	total_days_between	avg_days_between	
168	United States	19692	1269079	64	
101	Canada	18149	1187009	65	
164	United Kingdom	17023	810608	47	
117	Ireland	16843	889409	52	
100	Australia	15672	595630	38	
122	Hungary	15591	689870	44	
151	New Zealand	15536	925142	59	
124	Greece	14472	1219656	84	
140	Lithuania	14067	1146470	81	
143	Latvia	13973	1518089	108	

The next table, seen above, highlights the ten countries that tend to lead the music trends. In other words, if a song ends up becoming a global hit, in which countries does it usually first appear on the top charts? A closer look at the code is needed to fully understand the process for calculating this, but at a high level, I was looking for any time when a song showed up on two different countries' top music charts. For example, if a song first appeared on the United States charts on April 5th, and that same song first appeared on Japan's top music charts a month later, I would classify this as a 30-day gap between two. Anytime this type of phenomenon took place, that country's *song_leadership_count* value would increase by 1, so in theory, the countries with the highest values in that column tend to be the first places where global pop songs appear on the charts.

As expected, the United States was first here, which makes sense given that a lot of the most popular global artists hail from here. Otherwise, the top few countries were the ones I was expecting to see based on my own observations. I also did the same analysis to look at countries least likely to lead the pack, and some of the names that came out were Vietnam, Saudi Arabia, and Nicaragua. This was also intuitive, as I am not aware of many globally-renowned artists from those countries.

<pre> SELECT * FROM dim_agg_influence ORDER BY total_scaled DESC LIMIT 10; </pre>													
<pre> * postgresql://capstoneuser:***@capstone-proj.c0df8unkdobb.us-west-2.redshift.amazonaws.com:5439/dev 10 rows affected. </pre>													
influencer_id	influencer_name	depth_0	depth_1	depth_2	depth_3	depth_4	depth_5	depth_6	depth_7	depth_8	depth_9	depth_10	total_scaled
234	Elvis Presley	1	167	1975	3632	4162	4415	4533	4598	4619	4630	4631	1571.224609375
169	Cab Calloway	1	28	396	2594	4634	4847	4880	4884	4884	4884	4884	1029.90625
679	The Clash	1	137	897	1670	1972	2284	2541	2818	3197	3538	3818	783.69921875
927	Willie Nelson	1	67	487	1453	2495	3143	3532	3817	4126	4313	4462	708.294921875
1407	Jethro Tull	1	15	182	742	1753	2264	2522	2751	2900	2996	3141	409.275390625
2537	Phosphorescent	1	4	76	499	1456	2496	3143	3532	3817	4126	4313	359.470703125
1230	Little Eva	1	4	61	521	1574	2389	2740	2911	3145	3336	3648	345.88671875
1075	Donny Hathaway	1	32	206	611	1008	1491	1676	1752	1774	1789	1808	306.298828125
38	Red Hot Chili Peppers	1	40	289	529	703	908	1031	1114	1239	1372	1453	264.857421875
1443	Celia Cruz	1	22	61	156	426	1302	2668	3675	3974	4122	4272	214.37890625

Lastly, I was curious to see who the most influential artists were. The way to make sense of this data is that the number in the *depth_1* column is the number of artists that influencer impacted directly, *depth_2* is the number of artists who were impacted by artists that were impacted by the influencer, etc. Because this data was a fairly small sample, it could not be joined with the data above, and the sample nature also explains why there are some unexpected artists listed. However, Elvis being the top influencer is very much consistent with the musical historians I've, and some of the other names like Willie Nelson and Donny Hathaway are thought to have had large impacts on artists from the last few decades.

Although not without its challenges, this project ended up being very informative, a great opportunity to put to use many of these skills I've picked up from the nanodegree, and particularly interesting for me as a musician and close follower of music!