Ruby McKillips and Noah Bonner
SI 201 Final Project
December 11, 2025

Repository: https://github.com/rubymckillips/Final-201-Project.git

1. The goals for your project including what APIs/websites you planned to work with and what data you planned to gather (5 points)

Our goals for the project were to use two different sets of data, which were a spotify API (https://developer.spotify.com/documentation/web-api), and an Open Weather API (https://openweathermap.org/api?utm_source). We wanted to track some of the top artists on the US spotify, and track how their music, and music genres, compared across different countries, regions and weather patterns.

2. The goals that were achieved including what APIs/websites you actually worked with and what data you did gather (5 points)

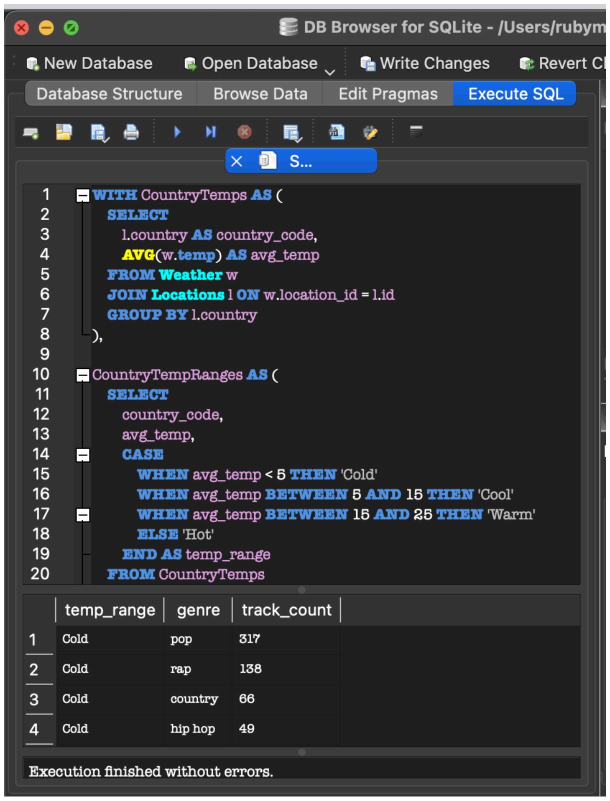
We achieved our goals by pulling song ID, song name, artist name, popularity, genre, and country codes from Spotify's API. From the Weather API, we pulled data including temperature, feels-like temperature, humidity levels, cloud percentage, wind speed, and descriptive weather conditions (like 'clear sky' or 'overcast clouds'). From there we were able to gather data and make calculations about which genres were listened to most across different climates, which artists were listened to the most per climate, and what the average temperature looked like across these countries.

3. The problems that you faced (5 points)

Combining the Spotify database with the weather database was confusing. We had to make sure table names matched, foreign keys aligned, and that we didn't accidentally overwrite data. At one point, we had final_project.db, final_project1.db, final_project2.db, etc., which made it easy to run code on the wrong file, but we needed these files to test out our code. This caused confusion about where the data actually lived. Also, both Noah and I had duplicate string data in some of our database tables. Instead of giving like terms an integer key (like repeated instances of "clear skies" or an artist's name), we listed each instance, and had trouble fixing this problem.

4. The calculations from the data in the database (5 points)

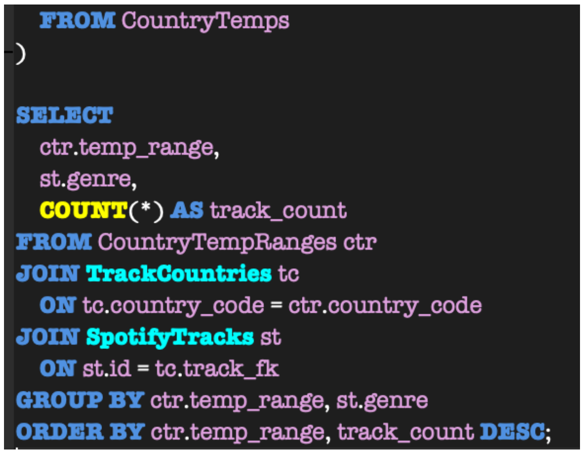
Calculation 1:



```sql
WITH CountryTemps AS (
    SELECT
        l.country AS country_code,
        AVG(w.temp) AS avg_temp
    FROM Weather w
    JOIN Locations l ON w.location_id = l.id
    GROUP BY l.country
),

CountryTempRanges AS (
    SELECT
        country_code,
        avg_temp,
        CASE
            WHEN avg_temp < 5 THEN 'Cold'
            WHEN avg_temp BETWEEN 5 AND 15 THEN 'Cool'
            WHEN avg_temp BETWEEN 15 AND 25 THEN 'Warm'
            ELSE 'Hot'
        END AS temp_range
    FROM CountryTemps
```

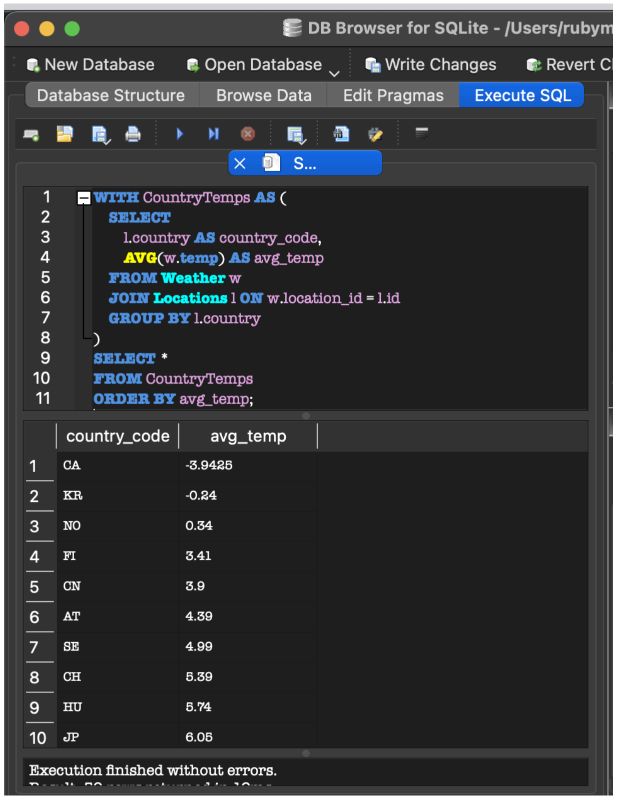
	temp_range	genre	track_count
1	Cold	pop	317
2	Cold	rap	138
3	Cold	country	66
4	Cold	hip hop	49

Execution finished without errors.



```sql
    FROM CountryTemps
)

SELECT
    ctr.temp_range,
    st.genre,
    COUNT(*) AS track_count
FROM CountryTempRanges ctr
JOIN TrackCountries tc
    ON tc.country_code = ctr.country_code
JOIN SpotifyTracks st
    ON st.id = tc.track_fk
GROUP BY ctr.temp_range, st.genre
ORDER BY ctr.temp_range, track_count DESC;
```

Calculation 2:

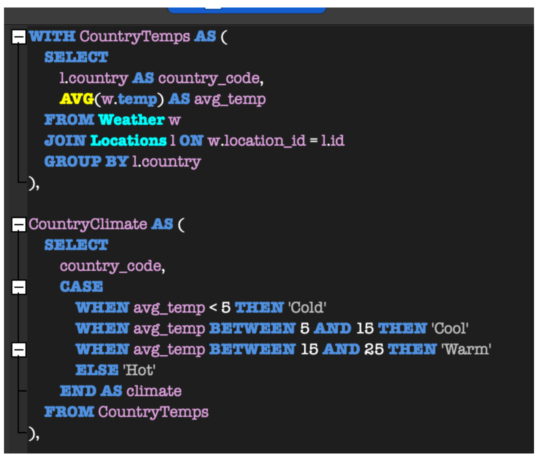


```sql
WITH CountryTemps AS (
  SELECT
    l.country AS country_code,
    AVG(w.temp) AS avg_temp
  FROM Weather w
  JOIN Locations l ON w.location_id = l.id
  GROUP BY l.country
)
SELECT *
FROM CountryTemps
ORDER BY avg_temp;
```

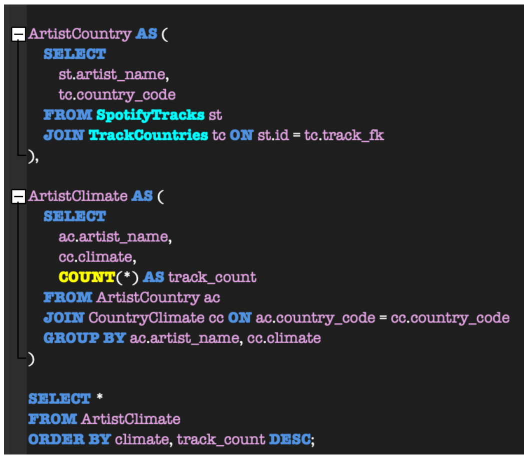
```
    country_code  avg_temp
1   CA            -3.9425
2   KR            -0.24
3   NO            0.34
4   FI            3.41
5   CN            3.9
6   AT            4.39
7   SE            4.99
8   CH            5.39
9   HU            5.74
10  JP            6.05
```

```
Execution finished without errors.
```

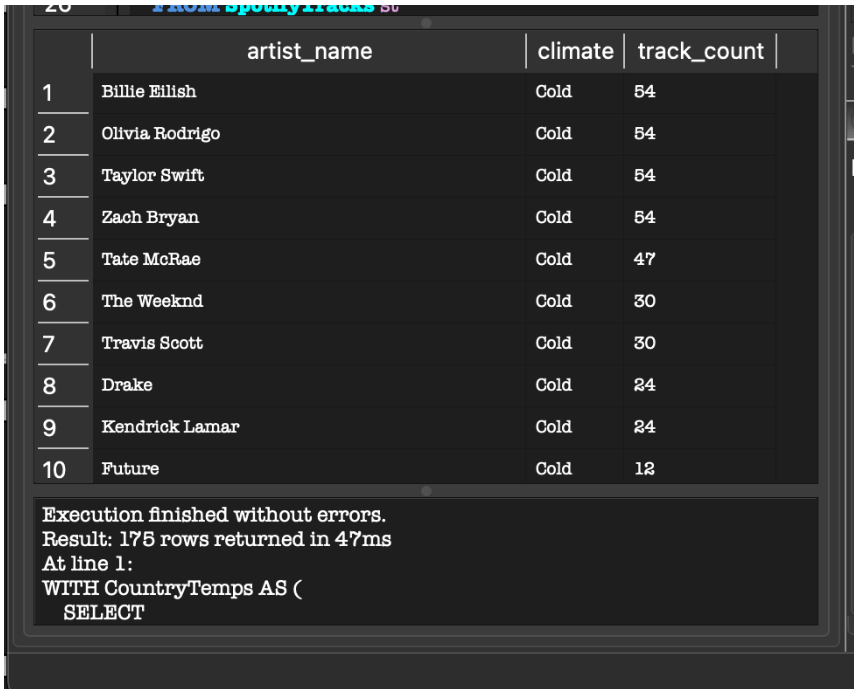
Calculation 3:



```sql
WITH CountryTemps AS (
    SELECT
        l.country AS country_code,
        AVG(w.temp) AS avg_temp
    FROM Weather w
    JOIN Locations l ON w.location_id = l.id
    GROUP BY l.country
),

CountryClimate AS (
    SELECT
        country_code,
        CASE
            WHEN avg_temp < 5 THEN 'Cold'
            WHEN avg_temp BETWEEN 5 AND 15 THEN 'Cool'
            WHEN avg_temp BETWEEN 15 AND 25 THEN 'Warm'
            ELSE 'Hot'
        END AS climate
    FROM CountryTemps
),
```



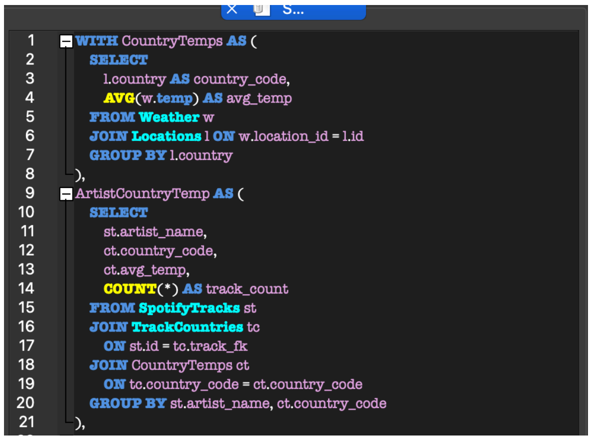
```sql
ArtistCountry AS (
    SELECT
        st.artist_name,
        tc.country_code
    FROM SpotifyTracks st
    JOIN TrackCountries tc ON st.id = tc.track_fk
),

ArtistClimate AS (
    SELECT
        ac.artist_name,
        cc.climate,
        COUNT(*) AS track_count
    FROM ArtistCountry ac
    JOIN CountryClimate cc ON ac.country_code = cc.country_code
    GROUP BY ac.artist_name, cc.climate
)

SELECT *
FROM ArtistClimate
ORDER BY climate, track_count DESC;
```



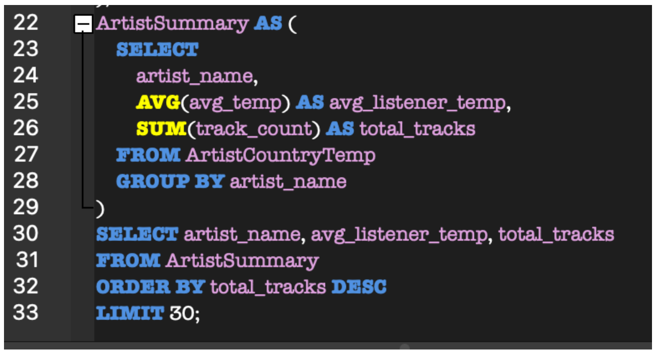
	artist_name	climate	track_count
1	Billie Eilish	Cold	54
2	Olivia Rodrigo	Cold	54
3	Taylor Swift	Cold	54
4	Zach Bryan	Cold	54
5	Tate McRae	Cold	47
6	The Weeknd	Cold	30
7	Travis Scott	Cold	30
8	Drake	Cold	24
9	Kendrick Lamar	Cold	24
10	Future	Cold	12

```
Execution finished without errors.
Result: 175 rows returned in 47ms
At line 1:
WITH CountryTemps AS (
   SELECT
```

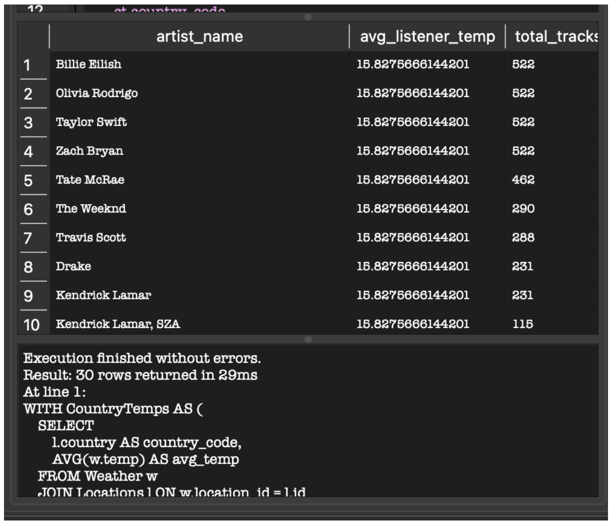
Calculation 4:



```sql
WITH CountryTemps AS (
    SELECT
        l.country AS country_code,
        AVG(w.temp) AS avg_temp
    FROM Weather w
    JOIN Locations l ON w.location_id = l.id
    GROUP BY l.country
),
ArtistCountryTemp AS (
    SELECT
        st.artist_name,
        ct.country_code,
        ct.avg_temp,
        COUNT(*) AS track_count
    FROM SpotifyTracks st
    JOIN TrackCountries tc
        ON st.id = tc.track_fk
    JOIN CountryTemps ct
        ON tc.country_code = ct.country_code
    GROUP BY st.artist_name, ct.country_code
),
```



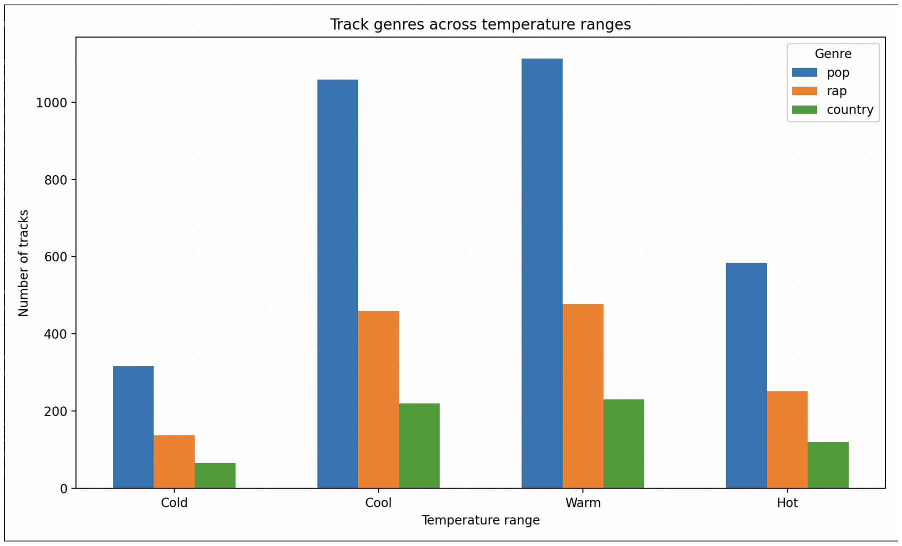
```sql
ArtistSummary AS (
    SELECT
        artist_name,
        AVG(avg_temp) AS avg_listener_temp,
        SUM(track_count) AS total_tracks
    FROM ArtistCountryTemp
    GROUP BY artist_name
)
SELECT artist_name, avg_listener_temp, total_tracks
FROM ArtistSummary
ORDER BY total_tracks DESC
LIMIT 30;
```

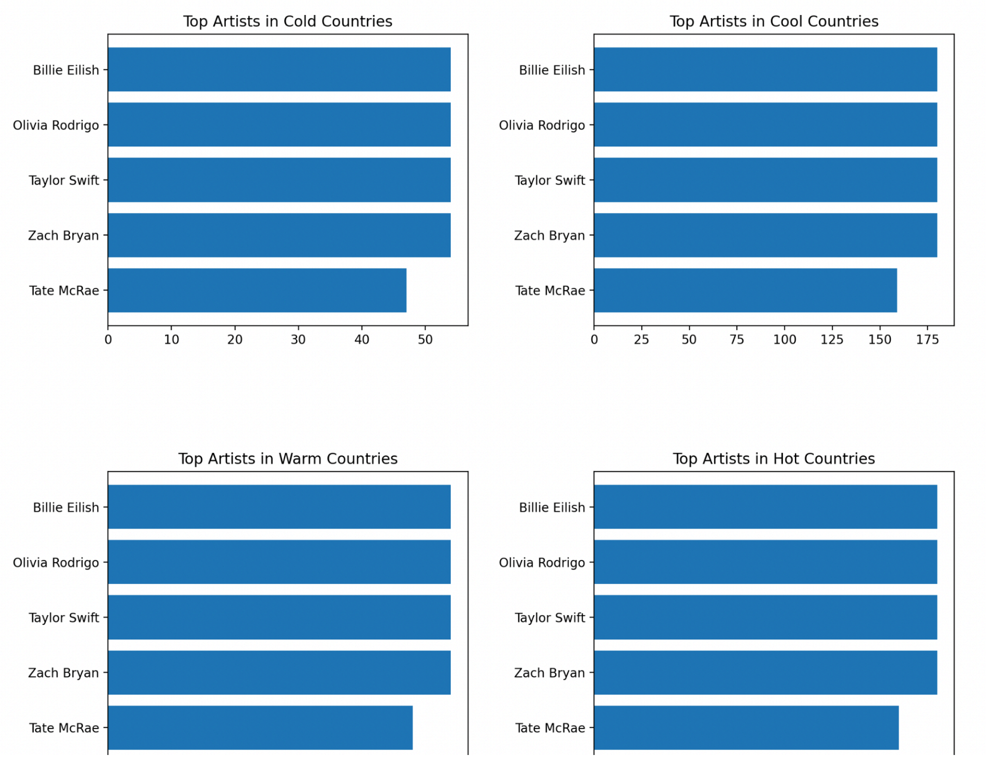


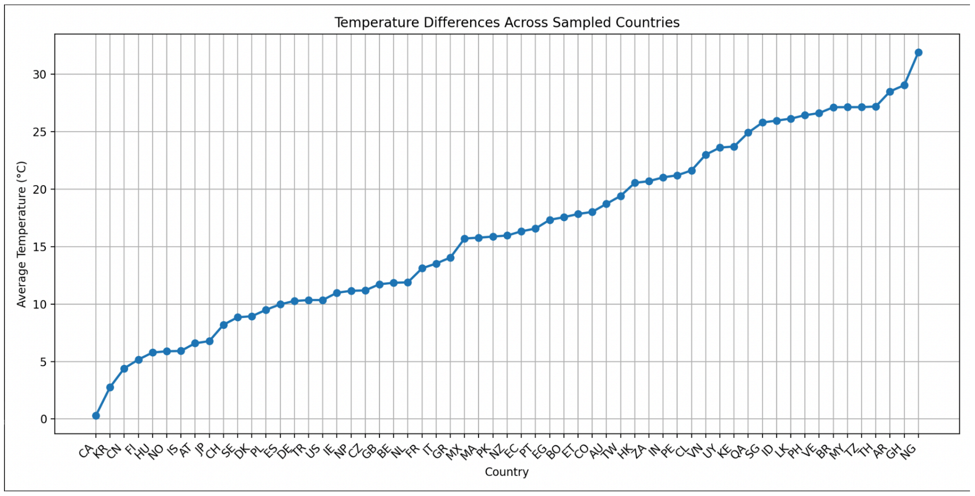
	artist_name	avg_listener_temp	total_tracks
1	Billie Eilish	15.8275666144201	522
2	Olivia Rodrigo	15.8275666144201	522
3	Taylor Swift	15.8275666144201	522
4	Zach Bryan	15.8275666144201	522
5	Tate McRae	15.8275666144201	462
6	The Weeknd	15.8275666144201	290
7	Travis Scott	15.8275666144201	288
8	Drake	15.8275666144201	231
9	Kendrick Lamar	15.8275666144201	231
10	Kendrick Lamar, SZA	15.8275666144201	115

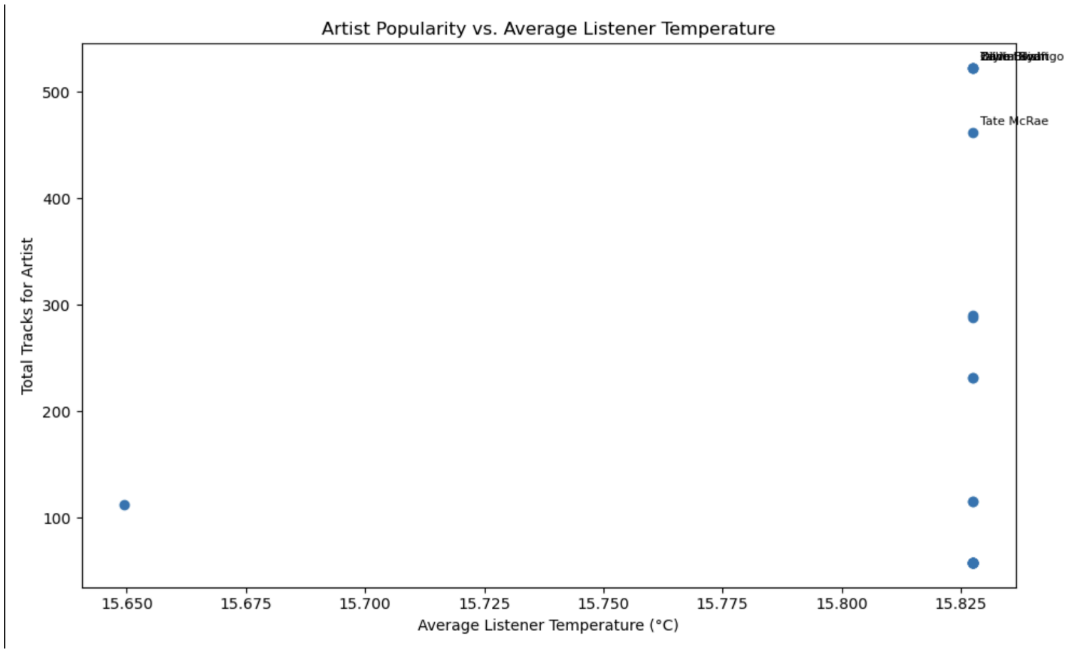
```
Execution finished without errors.
Result: 30 rows returned in 29ms
At line 1:
WITH CountryTemps AS (
  SELECT
    l.country AS country_code,
    AVG(w.temp) AS avg_temp
  FROM Weather w
  JOIN Locations l ON w.location_id = l.id
```

5. The visualizations that you created (5 points)




Top Artists in Cold Countries
Billie Eilish
Olivia Rodrigo
Taylor Swift
Zach Bryan
Tate McRae
0
10
20
30
40
50
Top Artists in Cool Countries
Billie Eilish
Olivia Rodrigo
Taylor Swift
Zach Bryan
Tate McRae
0
25
50
75
100
125
150
175
Top Artists in Warm Countries
Billie Eilish
Olivia Rodrigo
Taylor Swift
Zach Bryan
Tate McRae
Top Artists in Hot Countries
Billie Eilish
Olivia Rodrigo
Taylor Swift
Zach Bryan
Tate McRae


Temperature Differences Across Sampled Countries
Average Temperature (°C)
0
5
10
15
20
25
30
CA KR CN FI HU NO IS AT JP CH SE DK PL ES DE TR US IE NP CZ GB BE NL FR IT GR MX MA PK NZ EC PT EG BO ET CO AU TW HK ZA IN PE CL VN UY KE QA SG ID LK PH VE BR MY TZ TH AR GH NG
Country



6. Instructions for running your code (5 points)

Noah's Code:

1. First, run the database setup file(create_db.py). This creates the database and all the tables used for my Spotify data, including the tracks table, the track–country table, and the genre table.
2. To collect Spotify data, run my Spotify data script. It connects to the Spotify Web API, retrieves artist top tracks, and stores the results in the database. The script automatically limits itself to adding 25 new tracks per run and prevents any duplicate track data from being stored.
3. To update available countries for each track, run my country-update script. This calls Spotify again and fills in all the countries where each track is available, linking them to the tracks using the shared integer key.

Ruby's Code:

1. Run the weather database setup (weather_data.py). This will create the SQL database and all tables (Location and Weather) used for storing weather data.
2. Collect live weather data by running the same script again (python weather_data.py) to collect actual weather data from the OpenWeather Current Weather API. Each run adds 25 new rows until all cities have data.
3. Run the visualization scripts; each script reads from the completed database (final_project.db) and generates a visualization and a CSV file. For each script, run:
   a. python artists_by_climate.py (to produce a .png and .csv)
   b. python genre_vs_temperature_range.py (to produce a .png and .csv)

c. python temperature_across_countries.py (to produce a .png and .csv)
d. python artist_popularity_vs_temperature.py (to produce a .png and .csv)

7. An updated function diagram with the names of each function, the input, and output and who was responsible for that function (10 points)

Final Project Decomposition Diagram

├── A. Spotify Data Pipeline (Noah)

│

├── B. Weather Data Pipeline (Ruby)

│

└── C. Joint Analysis & Calculations (Both)

A. Spotify Data Pipeline

│

├── [1] get_connection()

│ • Input: none

│ • Output: (conn, cur)

│ • Purpose: Opens SQLite DB for Spotify tables.

│

├── [2] get_access_token()

│ • Input: client_id, client_secret

│ • Output: access token

│ • Purpose: Authenticates with Spotify API.

│

```
├── [3] search_artist_id(artist_name, token)
│     • Input: artist_name, token
│     • Output: artist_id
│     • Purpose: Finds Spotify artist ID via Search API.
│
├── [4] get_top_tracks(artist_id, token, market="US")
│     • Input: artist_id, token, market
│     • Output: list of track objects
│     • Purpose: Retrieves top tracks for an artist.
│
├── [5] get_artist_genre(artist_id, token)
│     • Input: artist_id, token
│     • Output: primary genre (string or None)
│     • Purpose: Gets the artist’s main genre.
│
├── [6] ensure_unique_track_index(cur)
│     • Input: cursor
│     • Output: UNIQUE constraint on track_id
│     • Purpose: Prevents duplicate tracks.
│
├── [7] ensure_trackcountries_table(cur)
│     • Input: cursor
│     • Output: Create TrackCountries table if missing
│    • Purpose: Sets up a relational table for country data.
│
├── [8] store_spotify_data(per_run_limit=25, allow_new_tracks=True)
│    • Input: artist list, limit flag
│    • Output: inserts into SpotifyTracks, TrackCountries, Genres
│    • Purpose: Main Spotify pipeline — calls above helpers, stores
│      max 25 new tracks, enforces uniqueness.
│
└── [9] update_countries_for_existing_tracks(limit=25)
     • Input: limit
     • Output: rows inserted into TrackCountries
     • Purpose: Queries Spotify for each track's available markets and
       stores country availability.
```

B. Weather Data Pipeline

```
│
├── [8] get_connection()
│    • Input: none
│    • Output: (conn, cur)
│    • Purpose: Open final_project.db for weather + locations.
│
├── [9] create_tables()
│    • Input: none
│    • Output: creates Locations & Weather tables if missing
│    • Purpose: Initialize schema.
│
├── [10] insert_locations()
│    • Input: none (uses CITIES list)
│    • Output: inserts cities with no duplicates
│
├── [11] get_locations_needing_weather(limit=25)
│    • Input: limit
│    • Output: list of (location_id, lat, lon)
│    • Purpose: Find up to 25 cities needing weather data.
│
├── [12] fetch_weather_for_location(lat, lon)
│    • Input: lat, lon
│    • Output: weather JSON from OpenWeather API
│
├── [13] store_weather_row(location_id, weather_json)
│    • Input: location_id, weather_json
│    • Output: inserts one Weather row (if new)
│
└── [14] get_and_store_weather_batch()
    • Input: none
    • Output: fetches & stores weather for up to 25 cities
```

• Purpose: Orchestrates weather collection (25-per-run rule).

C. Joint Analysis & Calculations

```
│
├── [15] calculate_results(db)
│    • Input: database path
│    • Output: aggregated results object
│    • Purpose: Calls all four calculations below.
│
├── [16] calc_genre_by_temp_range(db)
│    • Input: db (JOIN Spotify + Weather)
│    • Output: genre statistics by temperature range
│
├── [17] calc_temp_differences_across_countries(db)
│    • Input: Locations + Weather
│    • Output: average temperature per country
│
├── [18] calc_top_artists_by_climate(db)
│    • Input: SpotifyTracks + TrackCountries + Weather
│    • Output: ranked artists per climate zone
│
└── [19] calc_popularity_by_listening_temp(db)
   • Input: track popularity + temperature data
```

• Output: relationship between popularity & climate

8. You must also clearly document all resources you used. The documentation should be of the following form (10 points)

Date	**Issue Description**	**Location of Resource**	**Result (did it solve the issue?**

Date	**Issue**	**Location of Resource**	**Result**	
12/5	Spotify tracks were being inserted multiple times and causing duplicate data.	Spotify Web API documentation. Guided by Chat GPT	Yes, adding a UNIQUE index and using "INSERT OR IGNORE" solved the issue.	
12/5	Foreign key errors prevented deleting rows from the SpotifyTracks table.	StackOverflow thread on SQLite foreign key constraints.	Yes, I learned that I had to delete or rebuild the child table first before removing parent rows.	
12/6	TrackCountries table kept blocking deletes and updates because of mismatched keys.	SQLite documentation on foreign keys. Helped explained by ChatGPT	Yes, dropping and recreating TrackCountries fixed the mismatch error.	
12/8	The new Spotify data file was inserting NULL IDs because the PRIMARY KEY wasn't auto-incrementing.	SQLite documentation on INTEGER PRIMARY KEY usage. Helped and guided by CHATGPT	Yes, confirming the schema and using "id INTEGER PRIMARY KEY AUTOINCREMENT" resolved the issue.	
12/9	Couldn't	CHATGPT chat	Yes, switching to	

	demonstrate the 25-item limit because the database already had data in it.	log, explained to me how and what to do.	a new clean database file allowed the demonstration to work.	