Documentation

Concerts Trip, by Crazy Group

Software documentation

DB scheme structure

When designing the DB we understood we are going to generate a connection of data from multiple sources:

- SongKick API - artists & events & venues
- DiscOgs API - artists & players
- Spotify API - artists & genres
- Wikipedia - cities & countries & continents

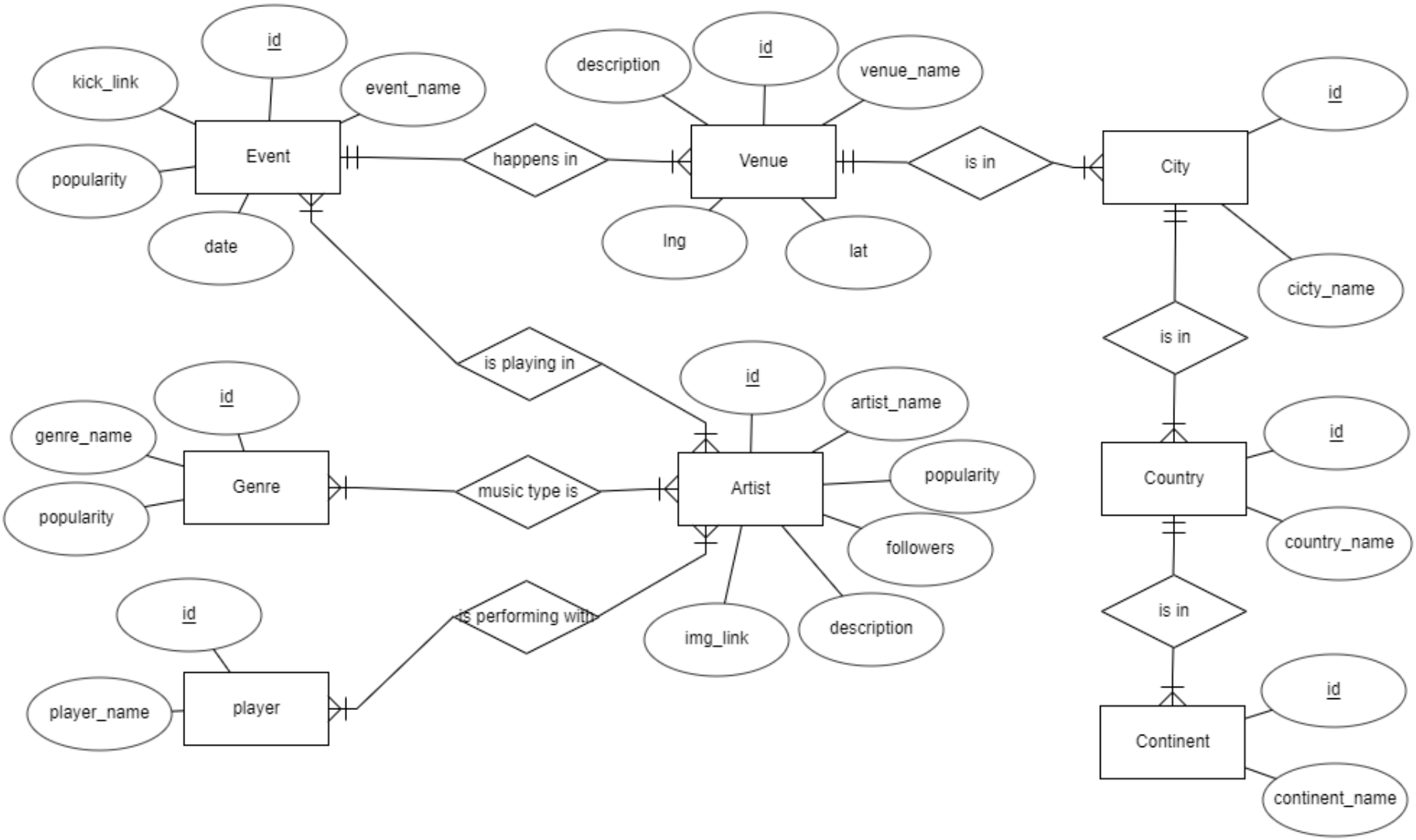
The center of our product is the **artist**, as he is the performer that the user wants to watch. All the data around the artists is aimed to help the user decide which of them he wants to see and under which conditions (**genre, location, players**).

Those conditions are relevant for multiple artists, and artists can contain multiple conditions. That is why we had to create a many-to-many relationships, leading us to create cross-tables.

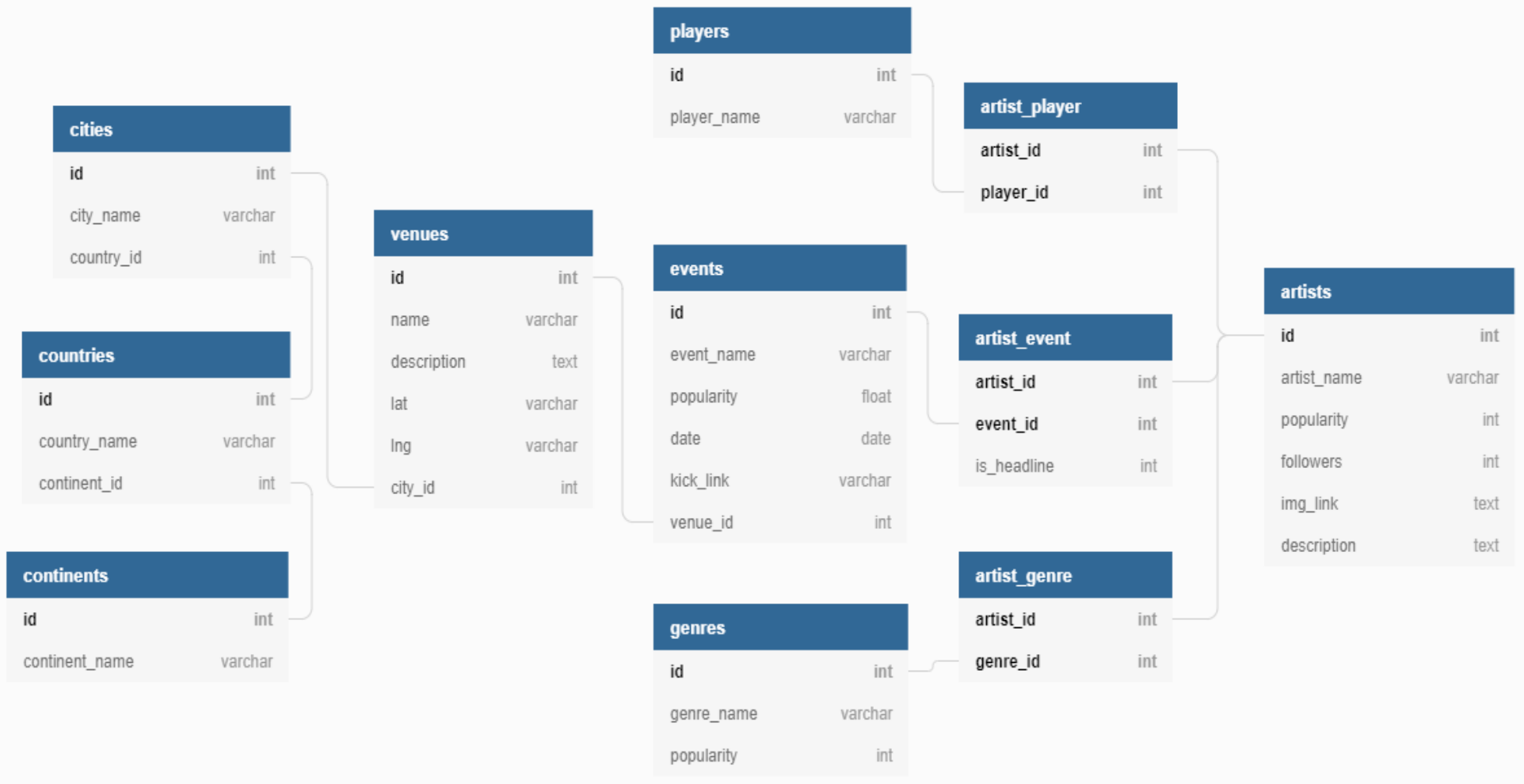
The decision of the user is eventually narrowed down to specific locations. This condition is a combination of several inner-conditions. Events are the point that connects the location with the artist, while venue gives us more details about the permanent location where the event takes place, and city-country-continent are providing us more general information about the location.

For that reason we've connected **event->venue->city->country->continent** from the most specific to the most general, in a connection of one-to-many, leading us to holding foreign keys in those table accordingly.

ER Diagram


id
kick_link
event_name
Event
popularity
date
happens in
description
id
venue_name
Venue
lng
lat
is in
City
id
cicty_name
is in
Country
id
country_name
is in
Continent
id
continent_name
is playing in
id
artist_name
popularity
followers
Artist
img_link
description
music type is
id
genre_name
Genre
popularity
is performing with
id
player_name
player

Relational Schema



Queries & DB Optimization

We've made several indices, in order to optimize our work with the DB:

- CREATE UNIQUE INDEX events_id_date ON **Events** (id, date);
- CREATE UNIQUE INDEX events_id_popularity ON **Events** (id, popularity);
- CREATE UNIQUE INDEX venues_id_city ON **Venues** (id, city_id);
- CREATE UNIQUE INDEX artists_id_popularity ON **Artists** (id, popularity);
- CREATE UNIQUE INDEX artists_id_followers ON **Artists** (id, followers);
- CREATE UNIQUE INDEX city_country_key ON **Cities** (id, country_id);
- CREATE UNIQUE INDEX country_continent_key ON **Countries** (id, continent_id);
- CREATE UNIQUE INDEX players_id_name ON **Players** (id, player_name);
- CREATE UNIQUE INDEX genre_id_name ON **Genres** (id, genre_name);
- CREATE UNIQUE INDEX genre_id_popularity ON **Genres** (id, popularity);
- CREATE UNIQUE INDEX artist_to_event ON **artist_event** (event_id, artist_id, is_headline);
- CREATE UNIQUE INDEX artist_to_player ON **Artist_Player** (artist_id, player_id);
- CREATE UNIQUE INDEX artist_to_genre ON **Artist_Genre** (artist_id, genre_id);
- CREATE UNIQUE INDEX fulltext_venue_desc ON **venues** (description);
- CREATE UNIQUE INDEX fulltext_artist_desc ON **artists** (description);

With those indices in mind, we wrote our queries:

1. Are_there_concerts_on_these_dates

This query returns whether there are concerts in the specified range of dates.

We built indices on the major attributes of this query, this way, we succeeded in optimizing the Search done behind the scenes in order to provide us with the results we needed.

Attributes with indices:

- events_id_date

The database is designed to support the same date format used in the query (the one we receive from the server)

```
SELECT E.id
FROM events as E
WHERE E.date BETWEEN "2020-02-14" AND "2020-02-15"
```

2. Query_get_genres

This query returns all the genres available in the specified dates, it does that by checking which artists are performing in the given dates and then checking their genres.

We had to do a nested query because of the combination between DISTINCT and ORDER BY.

We built indices on the major attributes of this query, this way, we succeeded in optimizing the Search and Join done behind the scenes in order to provide us with the results we needed.

Attributes with indices:

- events_id_date
- genre_id_name
- genre_id_popularity

The database is designed to support the same date format used in the query (the one we receive from the server)

For the Join part, the database holds foreign keys in each table used in the query to enable the Join, such as: artist id in artist_genre relation. (it is a primary key in ‘artists’ relation)

```
SELECT g.genre_name AS genre
FROM genres as g,
    (SELECT DISTINCT G.id
     FROM events AS E, genres as G, artist_genre as AG, artists as A,
          artist_event as AE
     WHERE A.id = AG.artist_id AND AG.genre_id = G.id
     AND AE.artist_id = A.id AND E.id = AE.event_id
     AND E.date >= "2020-02-14" AND  E.date <= "2020-02-15") AS available_genre
WHERE g.id = available_genre.id
ORDER BY g.popularity DESC
```

3. Query_get_artists

This query returns all the artists performing in the given dates at the given locations. It does that by joining all the relevant tables, reviewing all the records in a nested loop and checking which of them satisfies the conditions.

We built indices on the major attributes of this query, this way, we succeeded in optimizing the Search and Join done behind the scenes in order to provide us with the results we needed.

Attributes with indices:

- events_id_date
- venues_id_city
- artists_id_popularity
- artist_to_event
- artist_to_player
- players_id_name

The database is designed to support the same date format used in the query (the one we receive from the server).

For the Join part, the database holds foreign keys in each table used in the query to enable the Join, such as: event_id in artist_event relation. (it's a primary key in events relation).

(see full query in the next page)

```
SELECT artist
FROM (
      (SELECT a.artist_name AS artist, a.popularity as popularity
       FROM artists a, (SELECT DISTINCT A.id
                        FROM artists as A,
                             (SELECT AE.artist_id as artist_id, c.city_name as city,
                                     co.country_name as country
                              FROM events as E, artist_event as AE, venues as V,
                                   cities as c, countries as co
                              WHERE c.country_id = co.id AND V.city_id = c.id
                              AND AE.event_id = E.id AND E.venue_id = V.id
                              AND (E.date BETWEEN "2020-02-14" AND "2020-02-15"))
                              As event_artist_by_date
                        WHERE A.id = event_artist_by_date.artist_id
                        AND (('Toronto' = event_artist_by_date.city
                        AND 'Canada'=event_artist_by_date.country))) As available_artist
       WHERE a.id = available_artists.id
       ORDER BY a.popularity DESC, a.followers DESC)
   UNION (
       SELECT p.player_name, -1 as popularity
       FROM artists a, artist_player ap, player p
       WHERE a.artist_name IN (SELECT a.artist_name AS artist
                               FROM artists a,
                                   (SELECT DISTINCT A.id
                                    FROM artists as A,
                                         (SELECT AE.artist_id as artist_id,
                                          c.city_name as city,
                                          co.country_name as country
                                          FROM events as E, artist_event as AE,
                                          venues as V, cities as c, countries as co
                                          WHERE c.country_id = co.id
                                          AND V.city_id = c.id AND AE.event_id = E.id
                                          AND E.venue_id = V.id
                                          AND (E.date BETWEEN "2020-02-14" AND
                                          "2020-02-15")) As event_artist_by_date
                                    WHERE A.id = event_artist_by_date.artist_id
                                    AND (('Toronto' =  event_artist_by_date.city
                                    AND 'Canada' =  event_artist_by_date.country)))
                                    As available_artists
                               WHERE a.id = available_artists.id
                               ORDER BY a.popularity DESC, a.followers DESC)
      AND a.id = ap.artist_id AND p.id = ap.player_id ) Order by popularity DESC) as t
```

4. Query_get_locations

This query returns all the locations in which artists with the given genre are performing in the given dates. It does that by joining all the relevant tables, reviewing all the records in a nested loop and checking which of them satisfies the conditions.

We built indices on the major attributes of this query, this way, we succeeded in optimizing the Search and Join done behind the scenes in order to provide us with the results we needed.

Attributes with indices:

- events_id_date
- genres_id_name
- venues_id_city
- city_country_key
- country_continent_key
- artist_to_genre

The database is designed to support the same date format used in the query (the one we receive from the server).

For the Join part, the database holds foreign keys in each table used in the query to enable the Join, such as: country_id in city relation. (it's a primary key in Country relation)

```
SELECT c.city_name as city, co.country_name AS country ,
       con.continent_name AS continent
FROM cities c, countries co, continents con,
    (SELECT C.id AS city_id, COUNT( DISTINCT E.id) as event_sum
     FROM genres as G, artist_genre as AG, artist_event as AE,
          events as E, venues as V, cities as C, countries as CO, continents as CON
     WHERE G.id = AG.genre_id AND AG.artist_id = AE.artist_id
     AND AE.event_id = E.id AND E.venue_id = V.id AND V.city_id = C.id
     AND C.country_id = CO.id AND CO.continent_id = CON.id
     AND (E.date BETWEEN "2020-02-14" AND "2020-02-15")
     AND (G.genre_name LIKE '%rock%')
     GROUP BY C.id) as city_event_sum
WHERE c.id = city_event_sum.city_id AND c.country_id = co.id
AND co.continent_id = con.id
ORDER BY city_event_sum.event_sum DESC
```

5. Query_get_concerts

This query returns all the concerts\events happening at one of the specified locations with at least one of the specified genres in the range of the given dates

Combined with all the concerts happening in the range of the given dates at one of the specified locations in which one of the given artists perform.

In other words, we would like to provide to our users the possibility to schedule a concerts' tour that includes both, their favorite genres and favorite artists despite their genres.

We built indices on the major attributes of this query, this way, we succeeded in optimizing the Search and Join done behind the scenes in order to provide us with the results we needed.

Attributes with indices:

- event_id_date
- genre_id_name
- venue_id_city
- city_country_key
- country_continent_key
- artist_to_event
- artist_to_player
- artist_to_genre

The database is designed to support the same date format used in the query (the one we receive from the server)

For the Join part, the database holds foreign keys in each table used in the query to enable the Join, such as: artist id in Artist_Genre relation. (it’s a primary key in Artist relation)

(see full query in the next page)

```
WITH ORDERED As(SELECT distinct id, date, artist_id, artist, event, e_popularity,
                    country, city, sort_key, ROW_NUMBER() OVER (PARTITION BY id ORDER
                    BY headline DESC, popularity desc) AS rn
               FROM ((SELECT DISTINCT E.id, E.date AS date, A.id as artist_id,
                       A.artist_name as artist, AE.is_headline as headline,
                       A.popularity as popularity, E.event_name AS event,
                       E.popularity as e_popularity, CO.country_name as country,
                       C.city_name as city, 0 as sort_key
                     FROM artists as A, artist_event as AE, events as E, venues as V,
                          cities as C, countries as CO
                     WHERE A.id = AE.artist_id AND AE.event_id = E.id
                     AND E.venue_id = V.id AND V.city_id = C.id AND C.country_id = CO.id
                     AND AE.is_headline = 1 AND (E.date BETWEEN "2020-02-14"
                     AND "2020-02-15") AND ('Post Malone' = A.artist_name )
                     AND (('Toronto' =  C.city_name AND 'Canada' = CO.country_name)))
       UNION (
              SELECT DISTINCT E.id, E.date AS date, A.id as artist_id, A.artist_name as
                    artist, AE.is_headline as headline, A.popularity as popularity,
                    E.event_name AS event, E.popularity as E_popularity,
                    CO.country_name as country, C.city_name as city, 0 as sort_key
              FROM (SELECT AP.artist_id AS id
                    FROM players P, artist_player AP
                    WHERE P.id = AP.player_id
                    AND ('Post Malone' = P.player_name)) AS artists_with_players,
                 artists as A, artist_event as AE, events as E, venues as V,
                 cities as C, countries as CO
              WHERE A.id = artists_with_players.id AND A.id = AE.artist_id
              AND AE.event_id = E.id AND E.venue_id = V.id AND V.city_id = C.id
              AND C.country_id = CO.id
              AND AE.is_headline = 1 AND (E.date BETWEEN "2020-06-12" AND "2020-06-14")
              AND (('Toronto' =  C.city_name AND 'Canada' =  CO.country_name)))
      UNION (
              SELECT DISTINCT E.id, E.date AS date, A.id as artist_id, A.artist_name as
                    artist, AE.is_headline as headlinen A.popularity as popularity,
                    E.event_name AS event, E.popularity as e_popularity,
                    CO.country_name as country, C.city_name as city, 1 as sort_key
              FROM genres as G, artist_genre as AG, artist_event as AE, events as E,
                   venues as V, cities as C,countries as CO, artists AS A
              WHERE G.id = AG.genre_id AND AG.artist_id = AE.artist_id
              AND AE.event_id = E.id AND AE.is_headline = 1 AND E.venue_id = V.id
              AND V.city_id = C.id AND C.country_id = CO.id AND AG.artist_id = A.id
              AND (E.date BETWEEN "2020-02-14" AND "2020-02-15")
            AND (G.genre_name LIKE '%rock%')
            AND (('Toronto' =  C.city_name AND 'Canada' = CO.country_name)))) as mytable)
SELECT *
FROM ORDERED WHERE rn = 1
ORDER BY sort_key, date, e_popularity DESC
```

6. Query_get_summary

This query returns all the concerts whom ids are in the list received, which means the events the user chose to be part of the concert tour he's planning.

This query provides informative additional attributes to the user to help him order the ticket, such as: a link to songkick to order tickets, a link to the exact location of the venue at which the event is happening and a cute photo of the lead artist of the event to motivate the user and encourage him to "go for it"

We built indices on the major attributes of this query, this way, we succeeded in optimizing the Search and Join done behind the scenes in order to provide us with the results we needed.

Attributes with indices:

- artist_to_event
- events_id_date
- venues_id_city
- city_country_key
- counrty_continent_key

For the Join part, the database holds foreign keys in each table used in the query to enable the Join, such as: city id in Venue relation. (it's a primary key in City relation)

```
SELECT DISTINCT e.date AS date, e.event_name as event, e.kick_link,
       co.country_name as country, c.city_name as city, a.img_link as photo,
       v.name as venue_name, v.lat, v.lon
FROM artists as a, artist_event as ae, events as e, venues as v, cities as c,
     countries as co, continents as con
WHERE a.id = ae.artist_id AND ae.event_id = e.id AND e.venue_id = v.id
AND v.city_id = c.id AND c.country_id = co.id AND co.continent_id = con.id
AND (e.id, a.id) IN ((3, 1794), (4, 6592))
```

7. Get_best_city_per_main_genre

The results of this query are a top city to every one of the top 10 genres.

It sorts the genres by genre's popularity in a descending order, then it sorts the cities with events of artists with one of the top 10 genres by artist's popularity also in a descending order then it outputs the first city for each genre.

We built indices on the major attributes of this query, this way, we succeeded in optimizing the Search and Join done behind the scenes in order to provide us with the results we needed.

Attributes with indices:

- Genres_id_popularity
- Genres_id_name
- Venues_id_city
- Artists_id_popularity
- Artist_to_event
- artist_to_genre

For the Join part, the database holds foreign keys in each table used in the query to enable the Join, such as: artist id in Artist_Genre relation. (it’s a primary key in Artist relation).

(see full query in the next page)

```
SELECT t1.genre, t1.city, t2.max_avg
FROM genres g, (SELECT MG.genre_name as genre, c.city_name AS city,
                    AVG(a.popularity) as avgArtistPopularity
                FROM (SELECT c.id AS id, COUNT(e.id) AS cnt
                      FROM cities AS c, venues AS v, events AS e
                      WHERE v.id = e.venue_id AND v.city_id = c.id
                      GROUP BY c.id
                      HAVING cnt > 10) AS city_event,
                   venues AS v, artist_event AS ae, countries AS co, cities AS c,
                   events AS e, artists AS a, artist_genre as ag,
                   (SELECT g.id, g.genre_name, g.popularity
                    FROM genres as g
                    ORDER BY g.popularity DESC LIMIT 10) As MG
               WHERE city_event.id = v.city_id AND v.id = e.venue_id
               AND e.id = ae.event_id AND ae.artist_id = a.id AND ae.is_headline = 1
               AND ag.artist_id = a.id AND co.id = c.country_id AND c.id = city_event.id
               AND MG.id = ag.genre_id AND ag.artist_id = a.id
               GROUP BY MG.genre_name, c.city_name
               HAVING COUNT(*) > 10
               ORDER BY MG.genre_name, avgArtistPopularity DESC) AS t1,

              (SELECT genre, MAX(avgArtistPopularity) AS max_avg
               FROM (SELECT MG.genre_name as genre, c.city_name AS city,
                           AVG(a.popularity) as avgArtistPopularity
                     FROM (SELECT c.id AS id, COUNT(e.id) AS cnt
                           FROM cities AS c, venues AS v, events AS e
                           WHERE v.id = e.venue_id AND v.city_id = c.id
                           GROUP BY c.id
                           HAVING cnt > 10) AS city_event,
                          venues AS v, artist_event AS ae, countries AS co, cities AS c,
                          events AS e, artists AS a, artist_genre as ag,
                          (SELECT g.id, g.genre_name, g.popularity
                           FROM genres as g
                           ORDER BY g.popularity DESC LIMIT 10) As MG
                     WHERE city_event.id = v.city_id AND v.id = e.venue_id
                     AND e.id = ae.event_id AND ae.artist_id = a.id AND ae.is_headline = 1
                     AND ag.artist_id = a.id AND co.id = c.country_id
                     AND c.id = city_event.id
                     AND MG.id = ag.genre_id AND ag.artist_id = a.id
                     GROUP BY MG.genre_name, c.city_name
                     HAVING COUNT(*) > 10
                     ORDER BY MG.genre_name, avgArtistPopularity DESC) AS t
               GROUP BY genre) as t2
WHERE t2.max_avg = t1.avgArtistPopularity AND t2.genre = t1.genre
AND g.genre_name = t1.genre
ORDER BY g.popularity desc
```

8. Get_top_3_city_per_continent_by_event_number

This query (as its name indicates) returns the top 3 cities per continent after ordering them by a descending events' number per country.

We use this query to recommend to the user where to travel in case number of events (which may indicate high tourists' visits' quality) is a priority for them.

We built indices on the major attributes of this query, this way, we succeeded in optimizing the Search and Join done behind the scenes in order to provide us with the results we needed.

Attributes with indices:

- Venues_id_city
- City_country_key
- country_continent_key

For the Join part, the database holds foreign keys in each table used in the query to enable the Join, such as: venue id in Events relation. (it's a primary key in Venue relation).

```
SELECT rs.con, rs.city, rs.cnt
FROM (SELECT con, city, cnt, Rank() over (Partition BY con ORDER BY cnt DESC ) AS rnk
      FROM (SELECT con.continent_name as con, c.city_name as city, cnt
            FROM (SELECT c.id as cityid, COUNT(e.id) as cnt
                  FROM cities as c, events AS e, venues v
                  WHERE e.venue_id = v.id AND v.city_id = c.id
                  GROUP BY c.id) AS cityEvent,
               continents as con, cities AS c, countries AS co
            WHERE con.id = co.continent_id AND cityEvent.cityid = c.id
            AND c.country_id = co.id
            ORDER BY cnt DESC) AS t
       ) AS rs
WHERE rnk <=  3
ORDER BY con, cnt desc
```

9. Get_top_3_city_per_continent_by_artist_followers

This query (as its name indicates) returns the top 3 cities per continent after ordering them in a descending value of artists' followers (we think that the number of followers indicates how preferable the artist is).

We use this query to recommend to the user where to travel in case events with commonly preferable artists (which may indicate an enjoyable, high quality events) is a priority for them.

We built indices on the major attributes of this query, this way, we succeeded in optimizing the Search and Join done behind the scenes in order to provide us with the results we needed.

Attributes with indices:

- Venues_id_city
- Artists_id_followers
- City_country_key
- country_continent_key

For the Join part, the database holds foreign keys in each table used in the query to enable the Join, such as: city id in Venue relation. (it's a primary key in City relation).

```
SELECT rs.con, rs.city, rs.sum
FROM (SELECT con, city, sum, Rank() over (Partition BY con ORDER BY sum DESC) AS rnk
      FROM (SELECT con.continent_name as con, c.city_name as city, sum
            FROM (SELECT city_ten.id as cityid, SUM(a.followers) as sum
                  FROM events as e, artists AS a, venues AS v, artist_event AS ae,
                       (SELECT c.id AS id, COUNT(e.id) AS cnt
                        FROM cities AS c, venues AS v, events AS e
                        WHERE v.id = e.venue_id AND v.city_id = c.id
                        GROUP BY c.id
                        HAVING cnt > 3) AS city_ten
                  WHERE e.venue_id = v.id AND v.city_id = city_ten.id
                  AND e.id = ae.event_id AND ae.artist_id = a.id
                  AND ae.is_headline = 1
                  GROUP BY city_ten.id) AS cityEvent,
                continents as con, cities AS c, countries AS co
            WHERE con.id = co.continent_id AND cityEvent.cityid = c.id
            AND c.country_id = co.id
            ORDER BY sum DESC) AS t
      ) rs
WHERE rnk <= 3
ORDER BY con, sum DESC
```

10. Get_last_2_city_per_continent_by_artist_followers

This query (as its name indicates) returns the last two cities per continent after ordering them in an ascending value of the number of artists' followers (we think that the number of followers indicates how preferable the artist is).

We use this query to recommend to the user where NOT to travel in case events with commonly preferable artists (which may indicate an enjoyable, high quality events) is a priority for them.

We built indices on the major attributes of this query, this way, we succeeded in optimizing the Search and Join done behind the scenes in order to provide us with the results we needed.

Attributes with indices:

- Venues_id_city
- Artists_id_followers
- City_country_key
- country_continent_key

For the Join part, the database holds foreign keys in each table used in the query to enable the Join, such as: artist id in Artist_Event relation. (it's a primary key in Artist relation).

```
SELECT rs.*
FROM (SELECT con, city, sum, Rank() over (Partition BY con ORDER BY sum) AS rnk
      FROM (SELECT con.continent_name as con, c.city_name as city, sum
            FROM (SELECT c.id as cityid, SUM(a.followers) as sum
                  FROM cities AS c, events as e, artists AS a, venues AS v,
                  artist_event AS ae
                  WHERE e.venue_id = v.id AND v.city_id = c.id AND e.id = ae.event_id
                  AND ae.artist_id = a.id AND ae.is_headline = 1
                  GROUP BY c.id) AS cityEvent,
               continents as con, cities AS c, countries AS co
            WHERE con.id = co.continent_id AND cityEvent.cityid = c.id
            AND c.country_id = co.id
            ORDER BY sum) AS t
      ) rs
WHERE rnk <= 2
ORDER BY con, sum DESC
```

11. Query_get_filtered_concerts

This is our Full-Text query.

With this query we offer the user the possibility to filter the concerts he chose depending on genres, dates, locations and artists fields. Filtering is done by entering a keyword representing a feature which characterizes the artists the user wishes to see performing.

It uses the MATCH AGAINST keywords of SQL to implement the search using the index we created on the "description" field in "Artist" relation.

We built an index on the major attribute of this query, this way, we succeeded in optimizing the Search of the keyword entered by the user.

- An index on Artist.description was created

For the Join part, the database holds foreign keys in each table used in the query to enable the Join, such as: event id in Artist_Event relation. (it's a primary key in Events relation)

```
SELECT DISTINCT E.id
FROM artists AS A, events AS E, venues AS V, artist_event AS AE
WHERE ((MATCH(A.description) AGAINST('"keyword"'))
    OR (MATCH(V.description) AGAINST('"keyword"')))
AND E.id IN (3, 4) AND E.id = AE.event_id
AND AE.artist_id = A.id AND V.id = E.venue_id
```

Code structure

The code structure is pretty straight forward:

- We created the DB first using the code found in CREATE-DB-SCRIPT.sql.
- Then we retrieved the data and populated the DB using the scripts under API-DATA-RETRIEVE folder.
- We made manipulations and connections on the data, via csv files, then we upload the data to the DB using WEBAPI.py.
- We use the APPLICATION-SOURCE-CODE / DBConnection.py in order to run queries on the MySQL server.
- These queries can be found in APPLICATION-SOURCE-CODE / queries.py
- We use flask as our web-server framework and its code can be found in APPLICATION-SOURCE-CODE / server.py
- The web-server runs queries from queries.py using DBConnection.py and renders the templates found under APPLICATION-SOURCE-CODE / templates.
- APPLICATION-SOURCE-CODE / static is used by the different HTML pages for static content, such as the background image.
- In order to run the application, you only need to run APPLICATION-SOURCE-CODE / server.py. All the other work (setting up the DB) was already done.

API description & Usage

In this project we constructed our main database using several APIs:

Songkick

originally a website containing information over past and upcoming performances, mainly concerts. The site offers an API which allows the user to query events, venues, locations and artists. Access to the API required the usage of credentials key, which was supplied to us after contacting the website's developers.

Since most of the above information required the Specific ID of the searched criteria, we created a list of selected cities to be contained in the DB, and queried Songkick for the ID of each one. Once we had the ID, we queried all upcoming shows of each city (starting on the 26th of January) until the site "ran out" of available information to give. The results, which came in the form of JSON, were parsed for relevant information using python:

- Songkick Event ID
- Event name
- Artists appearing on that event
- Date and time of the event (also in the form of a "datetime" object)
- Event popularity – a float describing the popularity of the event on Songkick.
- Is Headline – if the appearing artist is the main show, or just supporting.
- Venue name and ID

Each one was inserted into a CSV for further inspection. Once we removed duplicate entries, we queried for the information of each Venue in separate to get full info about the event's location. Parsing the Venue object gave us the following:

- Songkick Venue ID
- Venue Name
- Lat/Lon of Venue (if exist)
- Venue Address (including city and country)
- Venue website and phone number (if exist)
- Venue description – a long text describing the Venue.

Which were then used to create the venues table. Both of the above tables became the basis for all other API usage and queries.

Discogs

a massive online database containing millions of artists with accompanying info. Once we had enough events from Songkicks, we exported the list of all artists found, which we then used to query the Discogs API for each. We did so by running a “Search” query on Discogs and matching the artist’s name with all the query results. If the artist’s info was found, we used the result JSON for his entry to get its Discogs API Page, from which we obtained the following info:

- Artist name
- Artist Description – A long text with information about the artist, sometimes pages long (if exists).
- Current lineup – if the artist is a band, the JSON brought a list with all band members, each having “True” or “False” regarding their current status with the band. We selected only those with “True”.

The above info allowed us to construct some of the artist’s table. Artists who were not found during this search were removed.

Spotify

was used mainly to determine the artist’s popularity and the popularity of specific music genres. Once we finished querying Discogs and eliminating unfound results, we queried Spotify for each of the remaining artists. The spotify query returned the following info:

- Artist’s genres – a list containing all genres associated with the artist.
- Artist’s Popularity – an integer between 0 and 100 describing the artist’s popularity. The higher the number, the more the artist is popular among Spotify’s listeners.
- Artist’s Followers – the number of Spotify followers the artist currently has.

The above info allowed us to order results in terms of popularity, so that the more the band is popular, the bigger the chances it’ll appear first on the search results.

External libraries

Requests

Usually an external library. This module allows programmers to construct a “get” or “post” HTTP packets to be sent to a specific URL, alongside specific headers and parameters (given as an optional dictionary object). All queries sent to our chosen APIs were constructed using this module (with the credentials key and the search query).

CSV

a module that allows easier read/write to CSV format. It Was used for its DictRead and DictWrite methods, which allow to read each row as a dictionary with the headers as its keys.

Flask / Flask-Table

Flask is a web application framework. We use it in our project for the web-server actions.

Flask-Table library is a plugin to Flask, that we use in order to render nice looking tables.

my-sql-connector

My-sql-connector is a package used for connection with the MySQL DB, as we saw in class.

Client-side packages

On the client-side, we use a number of packages:

- Bootstrap.
- jquery - for javascript.
- daterangepicker (jquery) and moment (jquery) - used for the calendar on the first page.
- modal (jquery) - used for the modals on the first page.
- flexdatalist (jquery) - used for the dropdown lists on the first page.

The general flow of the application

The flow of the application can be broken down into the following steps:

1. The user fills in an input (client-side).
2. The website sends the input using one of the two methods:
    a. POST - in case the user's input is used on the same page (e.g. sending the genres the user picked for generating the locations' list).
    b. GET - in case the user is redirected to a different page (e.g. redirecting to the results page after user picked his concerts) or for getting information for the current page (e.g. returning the list of locations after it was generated based on the genres' list).
3. The web-server runs a query using the user's input as parameters.
4. The web-server then either sends a result back to the same page or renders a new one (client-side).