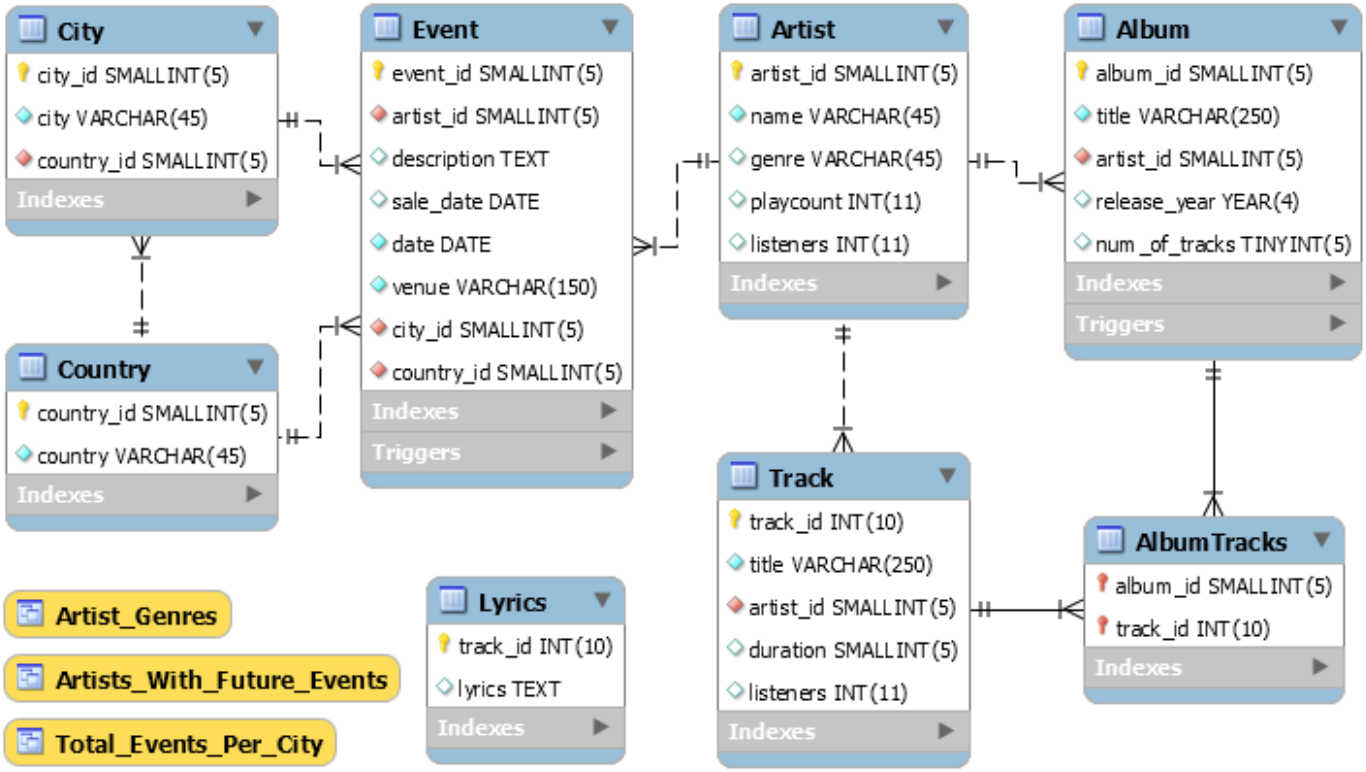
Software Documentation

EER DIAGRAM:



DATABASE DESIGN PROCESS

It was clear to us that we needed separate tables for Artists, Albums, Tracks and Events.

That being said, we made several database design decisions:

1. We wanted to use full-text search on the tracks' lyrics. We contemplated whether we should add a lyrics column to the Track table. When looking into this option, we found out that the mySQL version installed on TAU servers does not support full-text search in InnoDB storage engines. Therefore, had no better choice than to store the lyrics in a separate table, "Lyrics". Unfortunately, this table cannot be referred to by a foreign key from the Track table because MyISAM storage engine doesn't support this feature.
   It is important to note that the lyrics data is relatively large. The fact that the lyrics are in a separate table speeds up any of the queries that involve the Track table without using the lyrics.

2. When deciding where to store data regarding the Artist's genre, we contemplated storing the data in a separate table. The pros of this approach are to keep the database consistent, and to enable easy and simple updates, inserts or deletions of the genres in our database. Once we decided on our queries, we realized that an index on the genre could speed up the query execution time. Therefore, we decided to join store the genre in the Artist table, and use an index on this column. It is important to note that the genre field is relatively small (45 characters long at most), and therefore does not increase the record size in a significant manner.

3. We had a difference of opinions regarding where to store the data regarding the event's countries and cities. In the end, we decided to create separate tables for the cities and the countries. The reason for that is to enable simple updates, inserts and deletion. In addition, this way we were able to save space in the event table (which keep a column of foreign keys to the city and country tables, instead of varchar variables). In addition, we were able to easily fetch all the cities and countries in order to display them in the user interface.

4. It was clear to us that the tracks and albums are to be connected. However, when looking at the application in a broader view, we realized that we wanted to enable the existence of tracks without having an album connected to it by a foreign key (tracks that were released as "singles"). Following this decision, we enabled the existence of tracks with "NULL" in the album ID column. On the other hand, we decided that it is best to use a cascading delete when an artist is deleted, which means all relevant data that is referred to by the artist (events, tracks, albums) will be deleted. This fact caused a conflict regarding the track table. To handle this problem, we created a new table, AlbumTracks, which consists of the Album ID and the Track ID connected to it. This table is meant to connect the Albums and their Tracks, and enabled cascading deletion, and preserve the dependencies we wanted to preserve.

TABLES

Artist Table

The Artist table lists information regarding all artists.
The Artist table is referred to by a foreign key in the Event, Album and Track tables.

Columns

- artist_id: A surrogate primary key used to uniquely identify each artist in the table.
- name: The artist's name.
- genre: The name of the artist's genre.
- playcount: The artist's play count given by LastFM.
- listeners: The artist's listeners given by LastFM.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
artist_id	SMALLINT(5)	V	-	V	V	-
name	VARCHAR(45)	V	-	-	-	-
genre	VARCHAR(45)	-	-	-	-	NULL
playcount	INT(11)	-	-	V	-	NULL
listeners	INT(11)	-	-	V	-	NULL

Indexes		
Index Name	Type	Index Column
PRIMARY	PRIMARY	artist_id
listeners_idx	INDEX	listeners
genre_idx	INDEX	genre

Event Table

The Event table lists all the artists' events, including information regarding the ticketing sales date, the event date, and where the event takes place (name of venue).
The Event table refers to the Artist, Country and City table using a foreign key.

Columns

- event_id: A surrogate primary key used to uniquely identify each event in the table.
- artist_id: A foreign key identifying the artist which performs in the event.
- description: Event details.
- sale_date: The date of the start of tickets sales.
- date: The date of the event.
- venue: The name of the venue where the event takes place.
- city_id: A foreign key identifying the city where the event takes place.
- country_id: A foreign key identifying the country where the event takes place.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
event_id	SMALLINT(5)	V	-	V	V	-
artist_id	SMALLINT(5)	V	-	V	-	-
description	TEXT	-	-	-	-	NULL
sale_date	DATE	-	-	-	-	NULL
date	DATE	V	-	-	-	-
venue	VARCHAR(150)	V	-	-	-	-
city_id	SMALLINT(5)	V	-	V	-	-
country_id	SMALLINT(5)	V	-	V	-	-

Indexes		
Index Name	Type	Index Column
PRIMARY	PRIMARY	event_id
fk_event_artist_idx	INDEX	artist_id
fk_event_city_idx	INDEX	city_id
fk_event_country_idx	INDEX	country_id
event_date_idx	INDEX	date

Foreign Keys			
Foreign Key Name	Refers to	On Update	On Delete
fk_event_artist	Artist.artist_id	CASCADE	CASCADE
fk_event_city	City.city_id	CASCADE	RESTRICT
fk_event_country	Country.country_id	CASCADE	RESTRICT

Triggers		
Trigger Name	Type	Checks
event_BEFORE_INSERT	BEFORE INSERT	event date > current date sale date <= event date city_id corresponded to country_id
check_dates	BEFORE UPDATE	sale date <= event date

When an Artist is deleted/updated, relevant events are also deleted/updated.
If a country or city is updated, the relevant country or city is also updated but we do not allow deletion of these columns if there is an event that refers to them.

Country Table

The Country table contains a list of countries.
The Country table is referred to by a foreign key in the City and Event tables.

Columns

- country_id: A surrogate primary key used to uniquely identify each country in the table.
- country: The name of the country.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
country_id	SMALLINT(5)	V	-	V	V	
country	VARCHAR(45)	V	V	-	-	

Indexes		
Index Name	Type	Index Column
PRIMARY	PRIMARY	country_id
country_UNIQUE	UNIQU	country

City Table

The City table contains a list of cities.
The City table refers to the Country table using a foreign key, and is referred to by a foreign key in the Event table.

Columns

- city_id: A surrogate primary key used to uniquely identify each city in the table.
- city: The name of the city.
- country_id: A foreign key identifying the country that the city belongs to.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
city_id	SMALLINT(5)	V	-	V	V	-
city	VARCHAR(45)	V	-	-	-	-
country_id	SMALLINT(5)	V	-	V	-	-

Indexes		
Index Name	Type	Index Column
PRIMARY	PRIMARY	city_id
city_idx	INDEX	city
fk_city_country_idx	INDEX	county_id

Foreign Keys			
Foreign Key Name	Refers to	On Update	On Delete
fk_city_country	Country.country_id	CASCADE	RESTRICT

When a Country is updated, relevant cities are also updated.
We do not allow deletion of a country if there is an city that refers to it.

Album Table

The Album table lists all the artists' albums.
The Album table refers to the Artist table using a foreign key, and is referred to by a foreign key in the AlbumTracks table.

Columns

- album_id: A surrogate primary key used to uniquely identify each album in the table.
- title: The album's title.
- artist_id: A foreign key identifying the artist which released the album.
- release_year: The release year of the album.
- num_of_tracks: The number of tracks in the album.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
album_id	SMALLINT(5)	V	-	V	V	-
title	VARCHAR(45)	V	-	-	-	-
artist_id	SMALLINT(5)	V	-	V	-	-
release_year	YEAR(4)	-	-	-	-	NULL
num_of_tracks	TINYINT(5)	-	-	V	-	NULL
album_id	SMALLINT(5)	V	-	V	V	-

Indexes		
Index Name	Type	Index Column
PRIMARY	PRIMARY	album_id
fk_artist_album_idx	INDEX	artist_id
album_release_year_idx	INDEX	release_year

Foreign Keys			
Foreign Key Name	Refers to	On Update	On Delete
fk_artist_album	Artist.artist_id	CASCADE	CASCADE

Triggers		
Trigger Name	Type	Checks
check_release_year	BEFORE INSERT	release year <= current year

When an Artist is deleted/updated, relevant Albums are also deleted/updated.

Track Table

The Track table lists all the artists' tracks.
The Track table refers to the Artist table using a foreign key, and is referred to by a foreign key in the AlbumTracks table.

Columns

- track_id: A surrogate primary key used to uniquely identify each track in the table.
- title: The track title.
- artist_id: A foreign key identifying the artist who released the track.
- duration: The track's duration time (in seconds).
- listeners: The track's listeners given by LastFM.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
track_id	INT(10)	V	-	V	V	-
title	VARCHAR(45)	V	-	-	-	-
artist_id	SMALLINT(5)	V	-	V	-	-
duration	SMALLINT(5)	-	-	V	-	NULL
listeners	INT(11)	-	-	V	-	NULL
track_id	INT(10)	V	-	V	V	-

Indexes		
Index Name	Type	Index Column
PRIMARY	PRIMARY	track_id
fk_track_artist_idx	INDEX	artist_id

Foreign Keys			
Foreign Key Name	Refers to	On Update	On Delete
fk_track_artist	Artist.artist_id	CASCADE	CASCADE

When an Artist is deleted/updated, relevant Tracks are also deleted/updated.

AlbumTracks Table

The AlbumTracks table lists all the tracks of the album.
The AlbumTracks table refers to the Album and Track tables using a foreign key.

Columns

- album_id: A foreign key identifying the album.
- track_id: A foreign key identifying the track.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
album_id	SMALLINT(5)	V	-	V	V	
track_id	INT(10)	V	-	V	-	

Indexes		
Index Name	Type	Index Column
PRIMARY	PRIMARY	album_id, track_id
fk_track_idx	INDEX	track_id

Foreign Keys			
Foreign Key Name	Refers to	On Update	On Delete
fk_album	Album.album_id	CASCADE	CASCADE
fk_track	Track.track_id	CASCADE	CASCADE

When an Album or a Track is deleted/updated, the relevant keys are also deleted/updated.

Lyrics Table

The Lyrics table lists all the tracks' lyrics. This tables uses MyISAM engine, and the reason for that is to be able to use FULLTEXT index on lyrics column. The track_id does not refer to Track table, because MyISAM engine does not support foreign keys. However, the track_id appears in Track table. It is up to the DBA to keep these columns consistent.

Columns

- track_id: A surrogate primary key used to uniquely identify each lyric in the table.
- lyrics: The track's lyrics.

Columns						
Column Name	Data Type	Not NULL	Unique	Unsigned	Auto Increment	Default
track_id	INT(10)	V	-	V	-	-
lyrics	TEXT	-	-	-	-	NULL

Indexes		
Index Name	Type	Index Column
PRIMARY	PRIMARY	track_id
lyrics_idx	FULLTEXT	lyrics

VIEWS

Artist_Genre

```
CREATE OR REPLACE VIEW Artist_Genres AS
SELECT DISTINCT genre
FROM Artist
WHERE genre IS NOT NULL;
```

Description – the Artist_Genre view lists all the genres stored in the database. This view is being used by the user interface.

Artist_With_Future_Events

```
CREATE OR REPLACE VIEW Artists_With_Future_Events AS
SELECT  A.artist_id AS artist_id, A.name AS artist_name,
        A.genre AS genre, A.listeners AS listeners, A.playcount AS playcount,
        E.event_id AS event_id, E.date AS event_date, E.country_id AS country_id,
        E.sale_date AS sale_date, E.venue AS venue,E.city_id AS city_id,
        E.description AS description
FROM Event AS E INNER JOIN Artist AS A ON E.artist_id = A.artist_id
WHERE CURRENT_DATE() <= E.date;
```

Description – the Artist_With_Future_Events view lists all the artists which have future events starting from current date. This view is being used by the queries, so the relevant results will include only future events.

Total_Events_Per_City

```
CREATE OR REPLACE VIEW Total_Events_Per_City AS
SELECT E.country_id, E.city_id, City.city, COUNT(E.event_id) AS numOfEvents
FROM Event AS E INNER JOIN City ON E.city_id = City.city_id
GROUP BY E.city_id
HAVING numOfEvents >= 5;
```

Description – the Total_Events_Per_City view lists all the number of events on each city that has more than 5 events. This view is being used by the queries.

STORED PROCEDURES

sp_insertAlbum

Description – the sp_insertAlbum stored procedure inserts a new album to Album table. If the album already exists (i.e. the DB contains an album by the same artist with the same title), then an error is returned.

Parameters:

- p_title: The title of album.
- p_artist_id: The ID of the artist. Must be a valid ID (i.e. exists in Artist table).
- p_release_year: The release year of the album. Cannot be later than current year.
- p_num_of_tracks: The number of tracks in the album.

sp_insertEvent

Description – The sp_insertEvent stored procedure inserts a new event to Event table.

Parameters:

- p_artist_id: The ID of the artist. Must be a valid ID (i.e. exists in Artist table).
- p_description: Event details.
- p_sale_date: Starting sale date. Cannot be later than the date of the event.
- p_date: The date of event. Cannot be earlier than current date.
- p_venue: The name of the venue where the event takes place.
- p_city_id: The ID of the city. Must be a valid ID (i.e. exists in City table).

sp_updateEventDate

Description – the sp_updateEventDate stored procedure is used to update the event date.

Parameters:

- p_event_id: The ID of the event. Must be a valid ID (i.e. exists in Event table).
- p_event_date: The event date. Cannot be earlier than sale date.

Queries as Stored Procedures

We implemented all of our queries as stored procedures, in order to simplify the server-side script.

- top_artists
- fresh_artists
- latest_artists
- playlist_dur
- bad_words
- trivia_1
- trivia_2

Further details regarding each query can be found in the “COMPLEX QUERIES” section.

COMPLEX QUERIES

The user is asked to pick one of the three queries to find an event of his or her liking:

1. *top_artists:*

```
DROP PROCEDURE IF EXISTS top_artists$$
CREATE PROCEDURE top_artists(IN inGenre VARCHAR(45), IN numListeners INT,
                             IN numSongs INT, IN countryName VARCHAR(45))
BEGIN
SELECT A.artist_id AS artist_id, artist_name, sale_date, event_date,
        country, city, venue, description
FROM    (SELECT Artist.artist_id AS artist_id
        FROM Artist INNER JOIN Track ON track.artist_id = artist.artist_id
        WHERE Track.listeners >= numListeners AND genre = inGenre
        GROUP BY Artist.artist_id
        HAVING COUNT(track_id) >= numSongs) AS A INNER JOIN
        Artists_With_Future_Events AS E ON A.artist_id = E.artist_id
        INNER JOIN Country AS C ON E.country_id = C.country_id
        INNER JOIN City ON E.city_id = City.city_id
WHERE C.country = countryName;
END$$
```

This query receives the parameters: inGenre, numListeners, numSongs and countryName.
The query then returns a list of artists of genre “inGenre”, that released at least “numSongs” tracks with at least “numListeners” listeners, and their events in the given “countryName”.
To optimize this query, we used an index on Track.listeners and on Artist.genre.

2. *fresh_artists:*

```
DROP PROCEDURE IF EXISTS fresh_artists$$
CREATE PROCEDURE fresh_artists(IN times TINYINT, IN in_date DATE)
BEGIN
SET @times = times;
SET @in_date = in_date;
CREATE OR REPLACE VIEW events_60 AS
    # all the events + 60 from in_date
    SELECT *
    FROM Event as E
    WHERE DATEDIFF(E.date,getDate()) <= 60 AND DATEDIFF(E.date,getDate()) >=0;

CREATE OR REPLACE VIEW relevant_events AS
  # all the relevant events as defined above
  SELECT *
  FROM events_60 AS E2
  WHERE EXISTS (
                SELECT E3.artist_id
                FROM Event as E3
                WHERE DATEDIFF(E2.date,E3.date) <= 30 AND
                      DATEDIFF(E2.date,E3.date) > 0   AND
                      E2.artist_id = E3.artist_id
                HAVING COUNT(E3.event_id) < getTimes());

SELECT A.artist_id AS artist_id, A.name AS artist_name, D.sale_date AS sale_date,
       D.date AS event_date, D.venue AS venue, C.country AS country,
       C2.city AS city, D.description AS description
FROM relevant_events AS D INNER JOIN Artist AS A ON D.artist_id = A.artist_id
     INNER JOIN Country AS C ON C.country_id = D.country_id
     INNER JOIN City AS C2 ON C2.city_id = D.city_id
ORDER BY A.playcount DESC ;

DROP VIEW events_60;
DROP VIEW relevant_events;
END$$
```

This query receives the parameter: times and in_date. The query then returns a list of events that takes place between the dates “in_date” and “in_date” + 60 days, whose artist performs less than “times” times in the 30 days before the event. We sorted the output table by Artist.playcount in descending order.
To optimize this query, we used an index on Event.date.

3. *latest_artists:*

```
DROP PROCEDURE IF EXISTS latest_artists$$
CREATE PROCEDURE latest_artists(IN numYears TINYINT, IN numAlbums TINYINT)
BEGIN
SET @numYears = numYears;
SET @numAlbums = numAlbums;

# view to hold the most recent artists
CREATE OR REPLACE VIEW latest_artists AS
SELECT  E.artist_id as artist_id, COUNT(A.release_year) as cnt_albums
FROM Artist AS E INNER JOIN Album AS A ON E.artist_id = A.artist_id
WHERE A.release_year<=YEAR(current_date())
        AND YEAR(current_date()) - A.release_year <= getNumYears()
GROUP BY E.artist_id;

SELECT  E.artist_id as artist_id, artist_name, sale_date, event_date, country, city,
        venue, description
FROM    latest_artists AS T INNER JOIN Artists_With_Future_Events AS E
                            ON T.artist_id = E.artist_id
        INNER JOIN Country as C ON C.country_id = E.country_id
        INNER JOIN City as C2 ON C2.city_id = E.city_id
WHERE T.cnt_albums >= @numAlbums
ORDER BY E.listeners DESC;

DROP VIEW latest_artists;
END$$
```

This query receives the parameters: numYears and numAlbums. The query returns a list of events whose artists released a minimum of “numAlbums” in the last “numYears”.
To optimize this query, we used an index on album.year. In addition, we used the view called Artists_With_Future_Events which filtered out artists who did not have any future events in our database.

Once the user picked an event, we may use the date contained in his or her choice for the following queries:

4. *playlist_dur:*

```
DROP PROCEDURE IF EXISTS playlist_dur$$
CREATE PROCEDURE playlist_dur(IN artistId SMALLINT(5), IN playlistDuration INT)
BEGIN

DECLARE numOfSongs  INT;
DECLARE i INT;
DECLARE currentDur INT;
SET @artistId = artistId;

CREATE OR REPLACE VIEW ArtistTracks AS
SELECT  Track.title as track_name, Track.duration AS duration,
        Lyrics.lyrics as lyrics, Track.listeners AS listeners
FROM  Track INNER JOIN Artist ON Artist.artist_id = Track.artist_id
INNER JOIN Lyrics ON Track.track_id = Lyrics.track_id
WHERE Artist.artist_id = getArtistId() AND Lyrics.lyrics IS NOT NULL
AND duration IS NOT NULL;

SET i = 0;
SET currentDur = 0;
SET numOfSongs = (SELECT COUNT(*)
                  FROM ArtistTracks);
SET playlistDuration = playlistDuration*60;
WHILE currentDur <= playlistDuration  AND i <= numOfSongs DO
    SET  i = i + 1;
    SET currentDur = (SELECT (SUM(duration)) AS playlist_duration
                      FROM (SELECT * FROM ArtistTracks ORDER BY listeners DESC LIMIT i ) AS R);

END WHILE;


SET i = IF(currentDur>playlistDuration,i-1,i);
SELECT track_name AS title, (duration/60) AS duration, listeners, lyrics
FROM ArtistTracks
ORDER BY ArtistTracks.listeners DESC
LIMIT i;

DROP VIEW ArtistTracks;
END$$
```

This query receives a “playlistDuration” parameter from the user, and uses the “artist.artist_id” parameter from the event that the user chose in the first part of the application (the artist ID is received as a parameter from the server side).
The query begins by creating the view ArtistTracks, which contains information regarding all of the artist’s tracks whose lyrics are in our database, and whose duration is not NULL. The view minimized our data so it contains only the relevant data to the artist that was chosen, and therefor optimized the query.
The view receives the artist’s ID from the function getArtistId(), because it is impossible to create views that use variable parameters, but it is possible to create views that use parameters received by functions.
The procedure then iterates over the view. In each iteration, the procedure adds one more song to the output playlist(ordered by Track.listeners) until the total duration of the playlist is greater than the input duration, or until there are no more tracks to iterate over in the view table.
To optimize this query, we created the view ArtistTracks outside the while loop, and used it inside.

5. *bad_words:*

```
DROP PROCEDURE IF EXISTS bad_words$$
CREATE PROCEDURE bad_words(IN artistId SMALLINT(5), IN badWords VARCHAR(255))
BEGIN

SET @badWords = badWords;
SET @artistId = artistId;
#return num of tracks which has lyrics per artist of input artist's genre
CREATE OR REPLACE VIEW ALL_SONGS AS
SELECT  A.artist_id, COUNT(T.track_id) AS num_of_songs, A.listeners
FROM Track AS T INNER JOIN Artist AS A ON T.artist_id = A.artist_id
    JOIN Lyrics AS L ON T.track_id = L.track_id
WHERE A.genre = (SELECT Artist.genre FROM Artist WHERE artist_id = getArtistId())
        AND L.lyrics IS NOT NULL
GROUP BY A.artist_id
HAVING num_of_songs>10;


SELECT Artist.name AS artist_name, Track.title AS title, (Track.duration/60) AS duration
FROM Artist INNER JOIN Track ON Artist.artist_id = Track.artist_id
WHERE Artist.artist_id =
        (SELECT M.ALL_SONGS_ID
          FROM (
                SELECT ALL_SONGS.artist_id as ALL_SONGS_ID
                FROM ( SELECT Artist.artist_id as artist_id, Track.track_id AS track_id
                        FROM Artist INNER JOIN Track ON Artist.artist_id = Track.artist_id
                        INNER JOIN Lyrics ON Lyrics.track_id = Track.track_id
                        WHERE MATCH (lyrics) AGAINST (@badWords IN BOOLEAN MODE)) AS BAD_LYRICS
                RIGHT JOIN ALL_SONGS ON ALL_SONGS.artist_id = BAD_LYRICS.artist_id
                GROUP BY ALL_SONGS.artist_id
                ORDER BY (COUNT(BAD_LYRICS.track_id)/ALL_SONGS.num_of_songs) asc,
                          listeners desc
                limit 1) AS M)

ORDER BY Track.listeners DESC
LIMIT 20;

DROP VIEW ALL_SONGS;
END$$
```

This query receives a string from the user, called “badWords”, and uses the “Artist.artist_id” parameter from the event that the user chose in the first part of the application, in order to extract the requested genre.
The query first creates the view ALL_SONGS. The view contains the number of songs for each artist in the relevant genre, whose lyrics we have in our database (the lyrics of the track in the “lyrics” table is not NULL), and who has more than 10 songs (so that we can build a playlist consisting of at least 10 songs for each artist that is chosen).
We then create a nested query, to receive the artist id whose percentage of “bad songs” is the lowest. The percentage of “bad songs” is the number of tracks containing one or more of the words we received as input, *divided by* the number of tracks by the artist (whose lyrics are not NULL – in ALL_SONGS).
We used nested queries instead of views because we use full-text search to order to find the number of “bad songs”. As we previously mentioned, views do not support receiving variable parameters, and the match-against function does not support receiving parameters from functions, and therefore we could not create a view with this data.
In the abovementioned nested query, we used the RIGHT JOIN operator in order to preserve the artists who did not use one of the “bad words” in any of their songs. If we were to use INNER JOIN, these records would not have returned, in contrast to the fact that they have the lowest percentage of “bad songs”. Afterwards, we ordered the records by ascending percentage of “bad songs”, and then ordered again by descending number of listeners per artists (in order to resolve cases where more than one artist received the lowest “bad songs” percentage).
Once we received the artist ID whose percentage of “bad songs” is the lowest, we were able to create a playlist of up to 20 of the artist’s most listened to tracks (ordered by Track.listeners).
To optimize this query we used full-text index on the Lyrics.lyrics column in the lyrics table.
In addition, implementing the “bad words” query using the nested query (called “M” in the “where” clause) prevents the redundant recalculation of this query for *every* artist, which would have occurred if we would have selected the ALL_SONGS_ID directly from the nested query (without defining M in the “from” clause first).

6. *trivia_1:*

```
DROP PROCEDURE IF EXISTS trivia_1$$
CREATE PROCEDURE trivia_1(IN artistId SMALLINT(5), IN word VARCHAR(45), IN numTracks INT)
BEGIN
SET @artistId = artistId;
SET @word = word;
SET @numTracks = numTracks;

CREATE OR REPLACE VIEW artistAlbumTracks AS
SELECT Album.album_id AS album_id, Album.title AS album_title,
       Track.track_id AS track_id, Track.listeners AS listeners
FROM Artist INNER JOIN Album ON Artist.artist_id = Album.artist_id
    INNER JOIN Albumtracks ON Albumtracks.album_id = Album.album_id
    INNER JOIN Track ON Track.track_id = Albumtracks.track_id
WHERE Artist.artist_id = getArtistId();


SELECT album_title,SUM(listeners)
FROM artistAlbumTracks AS A INNER JOIN Lyrics ON A.track_id = Lyrics.track_id
WHERE MATCH(lyrics) AGAINST(@word)
GROUP BY album_id
HAVING COUNT(listeners)>=@numTracks
ORDER BY SUM(listeners)
LIMIT 1;

DROP VIEW artistAlbumTracks;
END$$
```

This query receives the parameters “word” and “numTracks” from the user, and uses the “Artist.artist_id” parameter from the event that the user chose in the first part of the application. The query returns the details of the artist’s most popular album that contains at least “numTracks” tracks with the word “word”.
The “most popular album” is the album with the largest sum of listeners count of the tracks containing the inserted words.
To optimize this query we used full-text index on the lyrics.lyrics column in the lyrics table.

7. *trivia_2:*

```
DROP PROCEDURE IF EXISTS trivia_2$$
CREATE PROCEDURE trivia_2(IN p_genre VARCHAR(45))
BEGIN

SET @genre = p_genre;
# return num of events of spcific genre per city
CREATE OR REPLACE VIEW TOTAL_EVENTS_IN_CITY_PER_GENRE AS
SELECT City.city_id, City.city, COUNT(E.event_id) AS numOfEvents
FROM Artist INNER JOIN Event AS E on Artist.artist_id = E.artist_id
            INNER Join City ON E.city_id = City.city_id
WHERE Artist.genre = getGenre()
GROUP BY City.city_id;

#return which city has the highest (total events of genre/total events) ratio
SELECT Total_Events_Per_City.country_id, Country.country,Total_Events_Per_City.city_id,
        Total_Events_Per_City.city,
        (TOTAL_EVENTS_IN_CITY_PER_GENRE.numOfEvents / Total_Events_Per_City.numOfEvents * 100)
        AS percent
FROM Total_Events_Per_City JOIN TOTAL_EVENTS_IN_CITY_PER_GENRE
            ON Total_Events_Per_City.city_id = TOTAL_EVENTS_IN_CITY_PER_GENRE.city_id
    JOIN Country ON Total_Events_Per_City.country_id = Country.country_id
ORDER BY percent DESC
LIMIT 1;

DROP VIEW TOTAL_EVENTS_IN_CITY_PER_GENRE;
END$$
```

This query receives a genre from the user, and returns the city who “likes the genre the most”. The city who “likes a genre the most” is a city whose percentage of number of events of that genre is the greatest.
To implement this query, we first create the view Total_Events_Per_City, which contains the number of events in each city.
We then created the view TOTAL_EVENTS_IN_CITY_PER_GENRE, which contains the number of events of the specified genre in each city.
We then divided the number of events per city in TOTAL_EVENTS_IN_CITY_PER_GENRE, by the number of events per city in Total_Events_Per_City and multiplied by 100, to get the percentage of the events of the genre in each city. The query returns the name of the city and the name of the country the city is in, that received the highest percentage per genre.
It is important to note that some of the cities in our database had relatively few events.
Therefore, to minimize the number of instances we receive 100 percent of shows of a specific genre, we decided to run this query on cities that had at least 5 events taking place in them.
To optimize this query we used index on the city.city_id column in the city table, which optimized the “group by” command.

STORED FUNCTIONS

All the function below were defined for the sole purpose of inserting user input into views:

- getArtistId – returns the artist_id inserted by the user
- getDate – returns the date inserted by the user
- getGenre – returns the name of the genre chosen (via scroll-down menu) by the user
- getNumYears – returns an integer inserted by user
- getTimes – returns an integer inserted by user

API USE / INTEGRATION PROCESS

At the beginning of our thought process, we wanted to find a way to retrieve a large amount of unbiased data from one of the API's in a single query. We found that the Last.fm API enabled us to retrieve raw data for the top 1000 artists in their database. We retrieved the data using a python script and saved it in JSON format. The reason for saving the data in JSON format is to keep our API requests at a minimum.
Afterwards, we created a python script to filter the relevant data we were interested in from the JSON format, and saving it in CVS files.

In the process of converting our data from JSON format to CSV files we noticed that some artist names included undefined characters in the English language (for example – Beyoncé). We decided to exclude these records. We were left to work with 887 artist records.
Saving our data in CSV files enabled us to isolate problematic records that needed fixing using the filter and sorting features in EXCEL.

We wanted to add information to our Artists database, for example, their music's genre, and we found this data from the MusicGraph API. The MusicGraph query was based on the artists name, but because the name was not a key, we were missing some artists. When we went back to check the reason for this, we noticed that the artist name was different in each API (for example, with or without the word "the" in the beginning). To fix this, we wrote a python script to overcome differences in the artist names and to receive all the data we needed.

We were interested in adding information regarding our albums. We considered using the Last.fm API but unfortunately their queries returned many duplicated and inconsistent data (for example different release dates for the same albums), and so we decided against using this API. We decided to use the MusicGraph API instead to retrieve this data. The MusicGraph API returned some duplicate albums as well, but we solved this issue using the excel features.

Regarding the tracks, we decided to retrieve all the songs in at most three albums per artist (we wanted to have at least 20,000 records but we didn't want to have too much data). This data, too, contained duplicates which we eliminated with EXCEL features.

Afterwards, we retrieved the lyrics for each track in our database using the Lyrics.ovh API. To increase the success rate of the lyrics query for each song, we removed redundant characters from the track names in the track database (for example "(live)", "[remix]", etc.).

After retrieving the lyrics to our tracks, we noticed that many of the lyrics texts started with the same sequence of Spanish words (which are not a part of the actual lyrics song). Therefore, we removed this sequence using a python script.

Lastly, we retrieved the data for our events database using BandsInTown API. After retrieving the data, we noticed that many artists did not have any events, and the reason for that is that these artists stopped preforming.

BONUS

1. Use of stored procedures:
    a. When thinking of the idea for the query *playlist_dur*, we searched for a way to iterate over the relevant tracks. One option was to have a very simple query, and to have most of the logic executed in python on the server side, but considering the project is DB oriented, we were not satisfied with our decision. The staff brought to our attention the stored procedure feature, which was not taught in class. That way, we were able to implement the whole query, along with the complex logic behind it, in SQL.
    b. After learning about stored procedures with realized that executing all of our 7 complex queries using this feature would simplify executions on server-side.

2. User interface:
    The user interface of our application is based on material design for bootstrap, so the user can experience the application in the most clear, comfortable manner. To optimize the clarity of our applications features, each one has a short paragraph describing what it does, and specifies the parameters needed to insert. The user interface is responsive, so that it can be accessed from every technological device.
    In addition, our application is immune to SQL injection. We pre-defined the input variables before the user inserts them, as taught in class, in order to keep our data base as safe as possible.