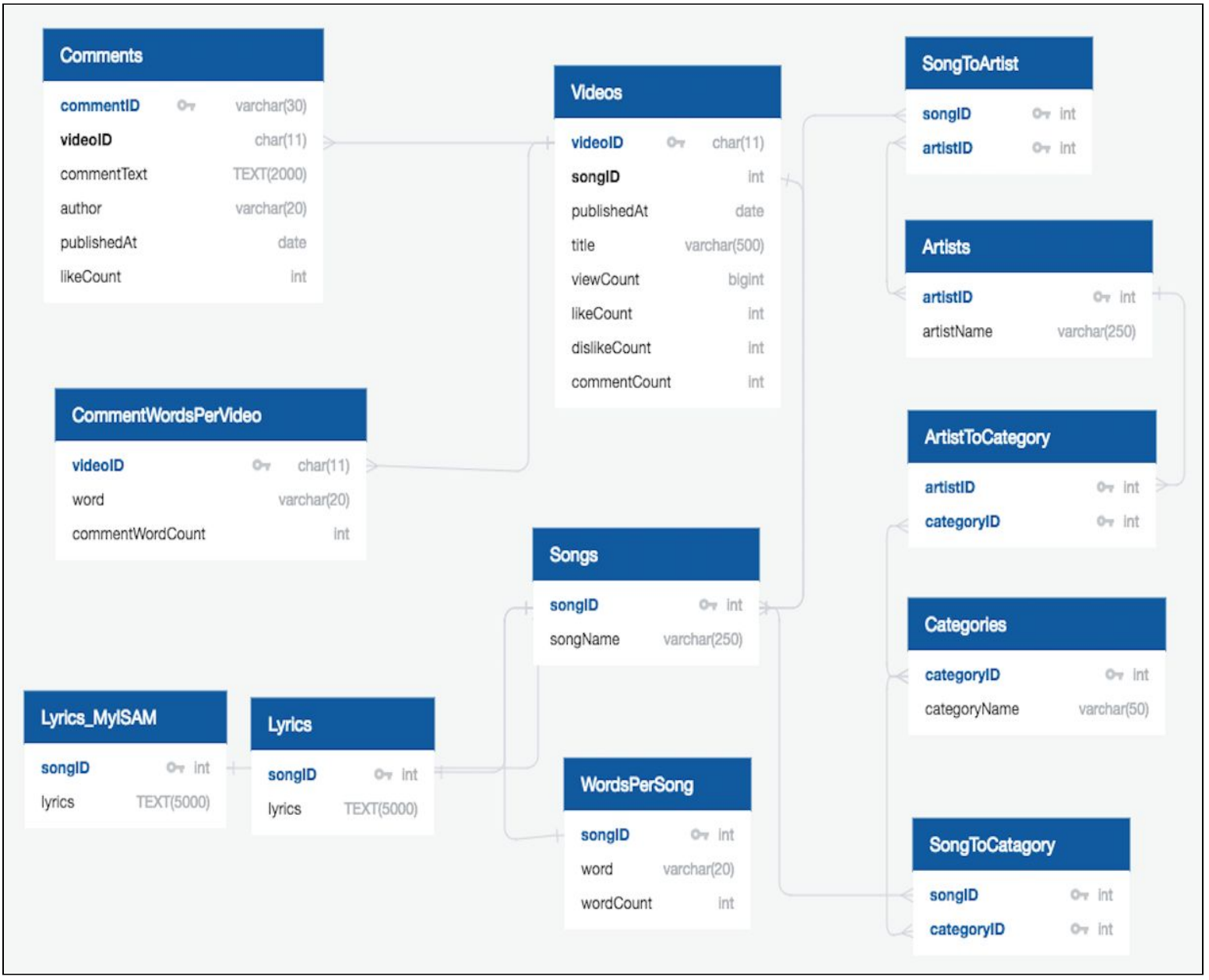
MusicTrends - Software Documentation

DB Scheme Structure:



Categories

API - LastFM

Field	Description	Type	Default	Other
categoryID	INCREMENTAL	int		PK
categoryName	not NULL	varchar(50)		

ArtistToCategory

API - LastFM

Field	Description	Type	Default	Other
artistID		int		PK, FK
categoryID		int		PK, FK

Artists

API - LastFM

Field	Description	Type	Default	Other
artistID	INCREMENTAL	int		PK
artistName	not NULL	varchar(250)		

Songs

API - LastFM

Field	Description	Type	Default	Other
songID	INCREMENTAL	int		PK
songName	not NULL	varchar(250)		

SongToCatagory

API - LastFM

Field	Description	Type	Default	Other
songID		int		PK, FK
categoryID		int		PK, FK

SongToArtist

API - LastFM

Field	Description	Type	Default	Other
songID		int		PK, FK
artistID		int		PK, FK

Lyrics

API - MusixMatch

Field	Description	Type	Default	Other
songID		int		PK, FK
lyrics	not NULL	TEXT(5000)		

Lyrics_MyISAM

API - MusixMatch (Duplicated for FULL TEXT search)

Field	Description	Type	Default	Other
songID		int		PK, FK
lyrics	not NULL, FULL TEXT	TEXT(5000)		

WordsPerSong

Pre-Processing

Field	Description	Type	Default	Other
songID		int		PK, FK
word		varchar(20)		
wordCount		int		

Videos

API - YouTube

Field	Description	Type	Default	Other
videoID	as defined in YouTube	char(11)		PK
songID		int		FK
publishedAt	not NULL	date		
title	not NULL	varchar(500)		
viewCount	not NULL, Max value as YouTube allows	bigint		
likeCount	not NULL	int		
dislikeCount	not NULL	int		
commentCount	not NULL	int		

Comments

API - YouTube

Field	Description	Type	Default	Other
commentID	as defined in YouTube	varchar(30)		PK
videoID		char(11)		FK
commentText	not NULL	TEXT(2000)		
author	not NULL	varchar(20)		
publishedAt	not NULL	date		
likeCount	not NULL	int		

CommentWordsPerVideo

Pre-Processing

Field	Description	Type	Default	Other
videoID		char(11)		PK, FK
word	not NULL	varchar(20)		
commentWordCount	not NULL	int		

DB Design and population process:

Overall we used 3 different API data sources for our database: LastFM, MusixMatch and Youtube.

We got our basic data using LastFM API to populate our first table - Songs. LastFM holds a unique id (called 'mbid') for each song. Having a unique id is crucial in order to store each song only once. Data retrieval from LastFM was performed by calls to multiple endpoints in LastFM API, in the following order:

Top categories -> Top Artists per category -> Top songs per artist

We denote that each artist may belong to more than a single category, and each song may be performed by a combination of multiple artists.

After the above data was retrieved, we could start using the two additional APIs:

- MusixMatch API - to get lyrics for songs (although not all songs actually had lyrics, and the 'free' api plan can give you up to 30% of the lyrics of a given song)
- YouTube API - we wanted to collect interesting data from youtube about the songs we collected from LastFM. One of the first questions that were raised at this point was - how should we define the relation between a song and a video. Of course, we understand that every song might have more than one youtube video, and usually many more. We decided to use a concatenation of the song name and the artists that perform the song, and take the first result we get from Youtube Search API, assuming this is the video that we are looking for with high probability. After getting

the song data, we queried the API again to get the comments of the videos. We decided to limit the amount of comments we collect for each video to the 10 last comments. We decided to stay consistent with the unique identifiers of VideoId and CommentId, which allows us to stay flexible and update our data on-the-fly.

DB Optimizations:

In order to speed-up some of our queries, we've decided to perform pre-processing on the Lyrics and Comments tables by counting how many times each word appears, which allows us to analyze their content quickly. These calculations are stored inside WordsPerSong, and CommentWordsPerSong.

In addition, we've added the following indexes:

- **CREATE INDEX** wordIndex **ON** WordsPerSong(word);
- **CREATE INDEX** commentWordIndex **ON** CommentWordsPerVideo(word);

Since the queries we use to address the table WordsPerSong and CommentWordsPerVideo use the word field we decided to make it an index. This resulted in a significant improvement in running complex lyrics analysis based queries such as 'Top sophisticated songs' and 'Smart fans' (see below)

One of the major challenges was to optimize our queries to support per-category filtering for almost every query we have in the system. At start, we considered filtering queries we have already written to return results for the requested category only. However, we realized that would result with lots of unnecessary records being processed and aggregated where they could already be filtered out in a primal point - inside our inner queries. Such approach would get us better performance for our queries, however it required maintaining two versions for almost each of our queries, which kind of look alike.

Complex queries:

Top Sophisticated Songs
Uses the pre-processing we've done on the Lyrics table, calculates a sophistication score for every song, while taking into account the variety of words used in its lyrics, repetition of words, and the total amount of words in the song. The score is being calculated by the following formula: ***Score = numberOfWords^2 * averageUniqueness / wordCount*** where *uniqueness* for a word is calculated as 1/(word frequency) and *wordCount* is the total count of words (penalty for repetitions).

```
SELECT
   songName,
```

```
   score
FROM
   (
      SELECT
         songID,
         (
            POW(COUNT(WordsPerSong.word), 2) / SUM(wordCount)
         )
         *AVG(uniqueness) AS score
      FROM
         (
            SELECT
               word,
               1 / SUM(wordCount) AS uniqueness
            FROM
               WordsPerSong
            GROUP BY
               word
         )
         AS wordUniqueness,
         WordsPerSong
      WHERE
         wordUniqueness.word = WordsPerSong.word
      GROUP BY
         WordsPerSong.songID
   )
   AS a,
   songs
WHERE
   songs.songID = a.songID
ORDER BY
   score DESC LIMIT % s;
```

Top Sophisticated Discussions

Similarly to sophisticated songs, uses the pre-processing we've done on the Comments table, and use the same algorithm, but this time on the Youtube comments of the song.

```
SELECT
   songName,
   score
FROM
   (
      SELECT
         videoID,
         (
            POW(COUNT(commentWordsPerVideo.word), 2) / SUM(commentWordCount)
         )
         *AVG(uniqueness) AS score
      FROM
         (
            SELECT
               word,
               1 / SUM(commentWordCount) AS uniqueness
            FROM
```

```
                commentWordsPerVideo
            GROUP BY
                word
        )
        AS wordUniqueness,
        commentWordsPerVideo
    WHERE
        wordUniqueness.word = commentWordsPerVideo.word
    GROUP BY
        commentWordsPerVideo.videoID
)
AS a,
songs,
videos
WHERE
    songs.songID = videos.songID
    AND videos.videoID = a.videoID
ORDER BY
    score DESC LIMIT % s;
```

Top Groupies query

Returns the users which are the biggest “groupies” of some artist (in the category). Groupie is defined to be a user which comments on a large number of videos of the same artist.

```
SELECT
    author,
    artistName
FROM
    artists,
    (
        SELECT
            author,
            songToArtist.artistID,
            videos.videoID
        FROM
            comments,
            songToArtist,
            videos
        WHERE
            comments.videoID = videos.videoID
            AND videos.songID = songToArtist.songID
        GROUP BY
            songToArtist.artistID,
            videos.videoID,
            author
    )
    AS authorComments
WHERE
    artists.artistID = authorComments.artistID
GROUP BY
    author,
    artists.artistID
ORDER BY
    COUNT(*) DESC LIMIT % s;
```

Top Hofrim artists

returns the artists with the longest lyrics in average. This query uses the pre-processing we've done on the Lyrics tabel.

```
SELECT
   artists.artistName,
   AVG(wordCount) AS avgWordCount
FROM
   artists,
   songtoartist,
   (
      SELECT
         songID,
         SUM(wordCount) AS wordCount
      FROM
         wordspersong
      GROUP BY
         songID
   )
   AS totalWordsPerSong
WHERE
   songtoartist.artistID = artists.artistID
   AND songtoartist.songID = totalWordsPerSong.songID
GROUP BY
   songtoartist.artistID,
   artists.artistName
ORDER BY
   avgWordCount DESC LIMIT % s;
```

Top viral songs

The songs which attracts a lot of attention in YouTube - has the most comments, and in average, its comments has the largest number of likes.

```
SELECT
   songName
FROM
   Songs,
   (
      SELECT
         songID,
         Videos.videoID,
         commentCount * avg_like_per_comment AS rating
      FROM
         Videos,
         (
            SELECT
               videoID,
               AVG(likeCount) AS avg_like_per_comment
            FROM
               Comments
            GROUP BY
               videoID
         )
         AS avgLikeCount
      WHERE
         avgLikeCount.videoID = Videos.videoID
      ORDER BY
         rating
   )
   AS totalRating WHERE
   Songs.songID = totalRating.songID LIMIT % s;
```

Top day of week (dates) with most comments

Ranks the days of the week according to the number of comments posted on that day.

```
SELECT
   DAYNAME(comments.publishedAt) AS day
FROM
   comments,
   videos
WHERE
   comments.videoID = videos.videoID
GROUP BY
   day
ORDER BY
   COUNT(*) DESC LIMIT % s;
```

Most controversial artists

Returns the artists which has the biggest difference between the number of likes and dislikes on all of their Videos in total.

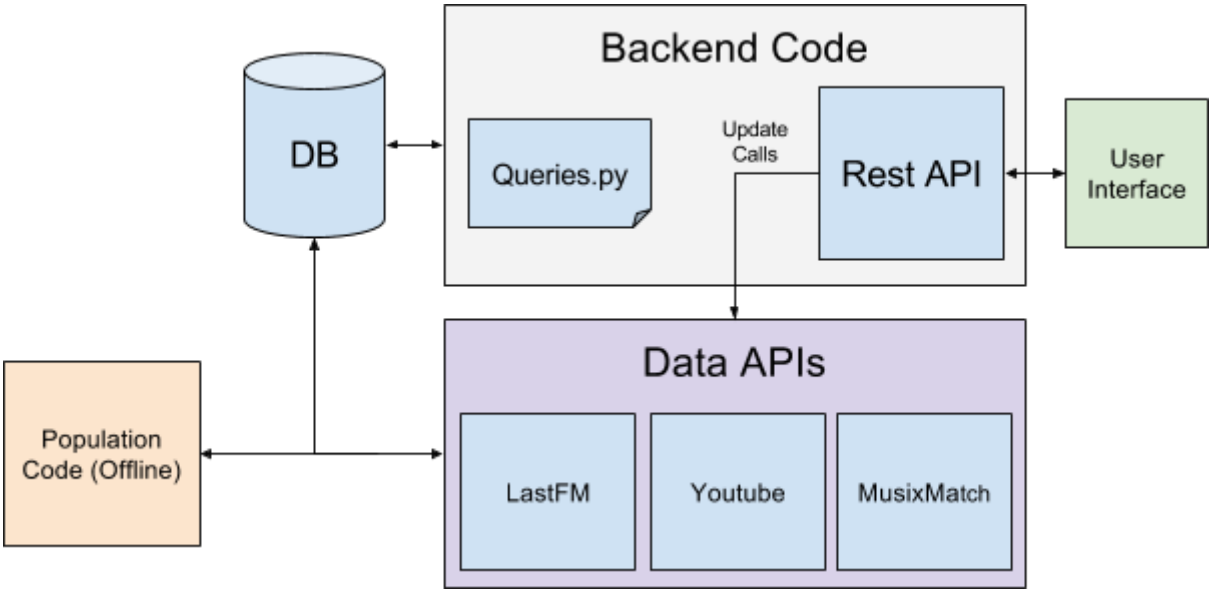
```
SELECT
   artists.artistName,
   AVG(scores.score) AS score
FROM
   artists,
   songtoartist,
   (
      SELECT
         songID,
         dislikeCount / likeCount AS score
      FROM
         videos
   )
   AS scores WHERE
   songtoartist.artistID = artists.artistID
   AND songtoartist.songID = scores.songID
GROUP BY
   artists.artistID,
   artists.artistName
ORDER BY
   score DESC LIMIT % s;
```

* All the queries also have a similar query with the filter for specific category.

Code structure:

All of our code can be found in our GitHub Repository:
https://github.com/alonkol/MusicTrends

High level design schema of our system:



In the zip file provided you can find the code in the requested format:

- SRC:
  - API-DATA-RETRIEVAL
    - LastFM
    - MusixMatch
    - YouTube
    - DBPopulation
  - APPLICATION-SOURCE-CODE
    - Server
    - Frontend
  - CREATE-DB-SCRIPT.sql

Notice that this is not a working code, since libraries, and json files were removed for it to make it lighter and easier to asses. For a full and working code, refer to the github repo.

Rest API:

Public endpoints:

```
/api/songs/likes/top/<int:amount>?category=<category_id>
/api/songs/dislikes/top/<int:amount>?category=<category_id>
/api/songs/views/top/<int:amount>?category=<category_id>
/api/songs/views/bottom/<int:amount>?category=<category_id>
/api/songs/wordscore/top/<int:amount>?category=<category_id>
/api/songs/wordscore/bottom/<int:amount>?category=<category_id>
/api/songs/discussionscore/top/<int:amount>?category=<category_id>
/api/songs/viral_songs/top/<int:amount>?category=<category_id>
/api/songs/days_with_most_comments/top/<int:amount>?category=<category_id>
/api/words/top/<int:amount>?category=<category_id>
/api/words/bottom/<int:amount>?category=<category_id>
/api/groupies/top/<int:amount>?category=<category_id>
/api/artists/head_eaters/top/<int:amount>?category=<category_id>
/api/artists/controversial/top/<int:amount>?category=<category_id>
/api/categories
/api/artists
/api/artists_for_category/<int:category_id>
/api/songs_for_artist/<int:artist_id>
/api/lyrics/get?song=<song_id>
/api/lyrics/search?text=<text>
```

Admin endpoints:

```
/api/blacklist_artist?key=<secret_key>&artist=<artist_id>
/api/lyrics/update?key=<secret_key>&song=<song_id>&lyrics=<lyrics>
/api/youtube/update?key=<secret_key>&song=<song_id>
/api/songs/add?key=<secret_key>&category=<category_id>&artist=<artist_id>&so
ng=<song_name>
```

External packages/libraries:

- json
- requests
- googleapiclient
- swagger.client
- mysql.connector
- Flask
- gevent.wsgi
- hashlib

General Flow of the Application:

At the homepage the user is introduced with the hottest trends on the music industry and inside one of the biggest source of music nowadays - YouTube. If he wishes, he can choose any music category and check out what are the trends on that scene.
On the admin side:

- In case a new hot song comes out, there is an option to add it (while retrieving all of the required data of course)
- In case of a new scandal, we can decide to ban a specific artist and remove all of his related data from our database.
- When a new song becomes a hit, out database can get easily updated accordingly.
- An admin can edit the lyrics of a song in case of an error or in order to remove offensive words

To make the admin menu visible one should add to the url address '?key=<SECRET_KEY>' with the secret key[1] provided for the admin. Once the menu is visible one can perform the actions written above, while the key will be validated before any update operation to our database, and will invoke an 'Unauthorized error' if the key is wrong.

Extras

- **Preprocessing stage**: Since we wanted to give some analytics that rely on words distribution in lyrics and in Youtube video's comment section we decided to compute and create two tables WordsPerSong and CommentWordsPerComment to help us in these queries and to make the queries relating these word distribution be much easier and faster.
- **Cache:** After we used the web application a bit, we noticed that most users will use the same api calls, and will get the same result until the data will change which happens not so often. For that reason and more, we decided to implement a cache for the server. The cache is a simple key:value cache which consists of the url that invoked the request and the request result. Once a new request is accepted by the flask server it checks the cache for its response and only invokes a call to the sql server if it's missing. After the call, if it generates a new response it's added to the cache. The items in the cache don't have a specific timeout, rather they are explicitly invalidated when a relevant update occurs. For example, when a song youtube's data is updated through the update song data page, all the urls regarding youtube data are invalidated from the cache for the 'All' category and for the song's category. That way the cache keeps its relevance, and not generates multiple calls to the sql server. By using these cache mechanism we managed to remove a lot of load from the sql server and give a better user experience while using our web application.

[1] Admin key is ***Wubalubadubdub!***