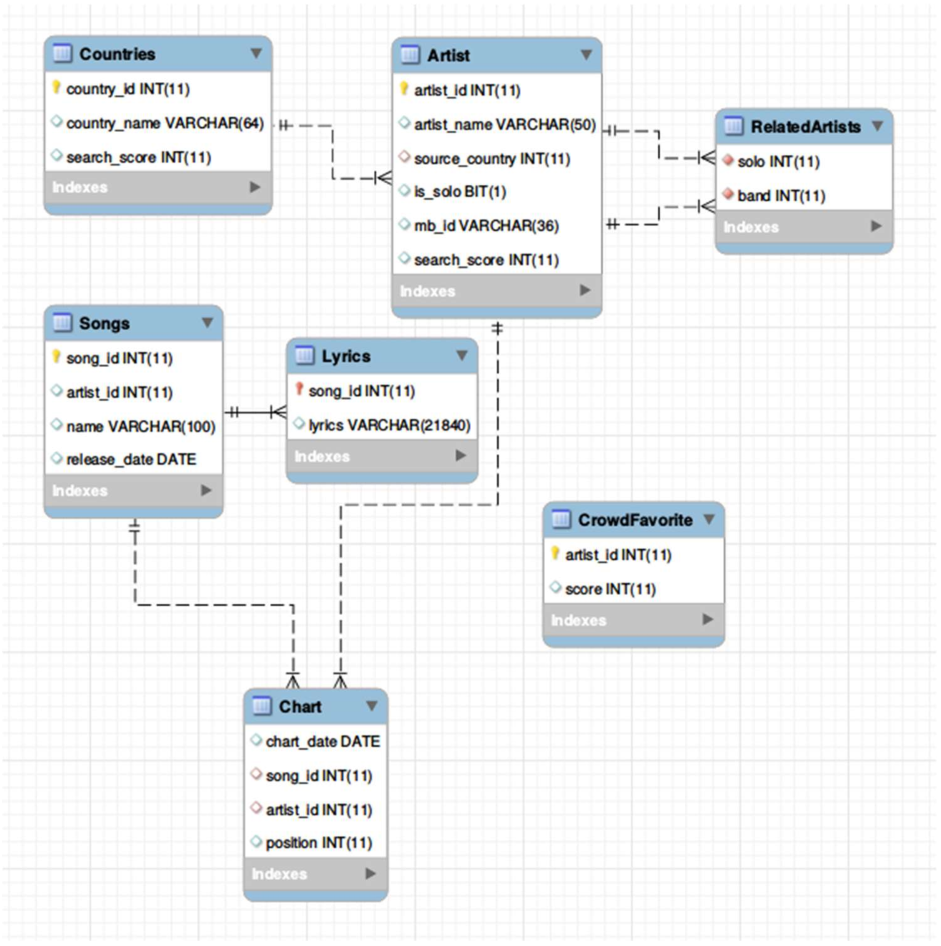
Top Hits

Software Documentation

DB Scheme Structure



The design is pretty straightforward, we constructed the tables with the information required for our application: songs, lyrics, countries of origin and artists. The Chart table contains songs/artists IDs listed in accordance with their appearance at the top charts tables of billboard.com. We'll go over the nontrivial information stored in our DB.

We store, and later use, connections between individual artists and their bands in the **'RelatedArtists'** table. This is done by processing the information received from musicbrainz.com, one of our sources, which provides artists membership in a band or a group. We later use this information in artist-based searches.

Additionally, we store the value of mb_id, which is the ID used by musicbrainz.com, since artists are commonly referred to by different aliases, yet their ID remains the same.
We use **'CrowdFavorite'** table to keep track on the users aggregated scores for each artist. Since most artist in 'Artist' table will probably not be voted, we decided to initialize an empty table and add rows to it according to the users votes.

Optimizations

We tried avoiding superfluous information by linking the tables using only IDs, which are merely integers.
We use indexes on the information searched the most, mainly Ids in the Chart and RelatedArtists tables. We use additional indexes for our full search query on the Lyrics table, and the MyISAM engine which is required for this index.
We also chose to index mb_id, since it is queried extensively during the construction of our DB.

Queries

1. For a given artist, returns all the song where his name appears that were sung by him or by any band he was in.

```
querySongsOnMe =
    SELECT s.artist_name AS col1, s.name AS col2
    FROM
    (
    (SELECT Artist.artist_name, Songs.song_id, Songs.name
    FROM Songs INNER JOIN Artist ON Artist.artist_id = Songs.artist_id
    WHERE Artist.artist_name = @artist_name
    AND Songs.release_date IS NOT NULL
    AND Songs.release_date BETWEEN @start_date AND @end_date)
    UNION
    (SELECT a2.artist_name, Songs.song_id, Songs.name
    FROM Artist AS a1, Artist AS a2, RelatedArtists, Songs
    WHERE a1.artist_name = @artist_name
    AND a1.artist_id = RelatedArtists.solo
    AND a2.artist_id = RelatedArtists.band
    AND a2.artist_id = Songs.artist_id
    AND Songs.release_date IS NOT NULL
    AND Songs.release_date BETWEEN @start_date AND @end_date)
    ) AS s
    INNER JOIN Lyrics ON s.song_id = Lyrics.song_id
    WHERE MATCH(Lyrics.lyrics) AGAINST(@artist_name IN BOOLEAN MODE)
    GROUP BY s.song_id;
```

Both ‘Artist’ and ‘Songs’ tables are indexed by ‘artist_id’, to make the queries faster.
Also, ‘Lyrics’ table has FullText index to allow full-text search.
Since the ‘Match Against’ takes the most time, we do it only after filtering by all the other conditions.

2. Returns the top charted songs in a specific timeframe based on its artist/band.

```
queryTopOfArtist =
    SELECT s.artist_name AS col1, s.name AS col2
    FROM
    (
    (SELECT Artist.artist_name, Songs.song_id, Songs.name
    FROM Songs INNER JOIN Artist ON Artist.artist_id = Songs.artist_id
    WHERE Artist.artist_name = @artist_name)
    UNION
    (SELECT a2.artist_name, Songs.song_id, Songs.name
    FROM Artist AS a1, Artist AS a2, RelatedArtists, Songs
    WHERE a1.artist_name = @artist_name
    AND a1.artist_id = RelatedArtists.solo
    AND a2.artist_id = RelatedArtists.band
    AND a2.artist_id = Songs.artist_id)
    ) AS s
    INNER JOIN Chart ON s.song_id = Chart.song_id
    WHERE Chart.chart_date IS NOT NULL
    AND Chart.chart_date BETWEEN @start_date AND @end_date
    GROUP BY s.song_id
    ORDER BY sum(100-Chart.position) DESC
    LIMIT 100;
```

Just like in previous query, ‘Artist’ table is indexed by ‘Artist.artist_name’, ‘Songs’ is indexed by ‘Songs.artist_id’, which optimizes the JOIN between those two tables.

‘RelatedArtist’ is indexes by ‘RelatedArtist.solo’ to optimize the WHERE clause in the second part of the UNION.

Also, ‘Chart’ is indexed by ‘Chart.chart_date’ to filter faster based on dates.

3. Top 10 best years of artist, including as part of a band.

```
queryBestYears =
SELECT YEAR(Chart.chart_date) AS col1, sum(100-Chart.position) AS col2
    FROM
    (
    (SELECT Songs.song_id
    FROM Songs INNER JOIN Artist ON Artist.artist_id = Songs.artist_id
    WHERE Artist.artist_name = @artist_name)
    UNION
    (SELECT Songs.song_id
    FROM Artist AS a1, Artist AS a2, RelatedArtists, Songs
    WHERE a1.artist_name = @artist_name
    AND a1.artist_id = RelatedArtists.solo
    AND a2.artist_id = RelatedArtists.band
    AND a2.artist_id = Songs.artist_id)
    ) AS s
    INNER JOIN Chart ON s.song_id = Chart.song_id
    WHERE Chart.chart_date IS NOT NULL
    AND Chart.chart_date BETWEEN @start_date AND @end_date
    GROUP BY YEAR(Chart.chart_date)
    ORDER BY sum(100-Chart.position) DESC
    LIMIT 10;
```

In contrast to the previous queries, in this one the UNION returns only the relevant song_ids of an artist, since the end result is a table with the year and the artist total score.

The same optimizations are applied here.

4. Returns the artists whose songs are top charted during the timeframe in a specific country.

```
queryTopArtistsOfCountryInTimeRange =
    SELECT Artist.artist_name AS col1, sum(100-Chart.position) AS col2
    FROM Songs INNER JOIN Artist ON Songs.artist_id = Artist.artist_id
    INNER JOIN Chart ON Artist.artist_id = Chart.artist_id
    INNER JOIN Countries ON Artist.source_country = Countries.country_id
    WHERE Countries.country_name = @country
    AND Chart.chart_date IS NOT NULL
    AND Chart.chart_date BETWEEN @start_date AND @end_date
    GROUP BY Artist.artist_id
    ORDER BY sum(100-Chart.position) DESC
    LIMIT 100;
```

Same optimization considerations as in the previous queries.

Also, ‘Countries’ is indexed by ‘Countries.country_name’ to find the relevant country_id faster.

5. Returns the top charted songs in a certain timeframe.

```
queryTopSongsInTimeRange =
    SELECT Artist.artist_name AS col1, Songs.name AS col2
    FROM Songs INNER JOIN Artist ON Songs.artist_id = Artist.artist_id
    INNER JOIN Chart ON Songs.song_id = Chart.song_id
    WHERE Chart.chart_date IS NOT NULL
    AND Chart.chart_date BETWEEN @start_date AND @end_date
    GROUP BY Songs.song_id
    ORDER BY sum(100-Chart.position) DESC
    LIMIT 100;
```

Same optimization considerations as in the previous queries.

6. Returns the top artists in a certain timeframe.

```
queryTopArtistsInTimeRange =
    SELECT Artist.artist_name AS col1, sum(100-Chart.position) AS col2
    FROM Songs INNER JOIN Artist ON Songs.artist_id = Artist.artist_id
    INNER JOIN Chart ON Artist.artist_id = Chart.artist_id
    WHERE Chart.chart_date IS NOT NULL
    AND Chart.chart_date BETWEEN @start_date AND @end_date
    GROUP BY Artist.artist_id
    ORDER BY sum(100-Chart.position) DESC
    LIMIT 100;
```

Same as the previous query, only based on artists.

7. Returns songs that contain the given country in the lyrics.

```
querySongsOnCountry =
    SELECT Artist.artist_name AS col1, Songs.name AS col2
    FROM Songs INNER JOIN Artist ON Songs.artist_id = Artist.artist_id
    INNER JOIN Chart ON Songs.song_id = Chart.song_id
    INNER JOIN Lyrics ON Songs.song_id = Lyrics.song_id
    WHERE Songs.release_date BETWEEN @start_date AND @end_date
    AND Chart.chart_date IS NOT NULL
    AND Chart.chart_date BETWEEN @start_date AND @end_date
    AND MATCH(Lyrics.lyrics) AGAINST(@country IN BOOLEAN MODE)
    GROUP BY Artist.artist_name, Songs.name
    ORDER BY sum(100-Chart.position) DESC;
```

The fulltext index on the lyrics at the 'Lyrics' table helps optimize this query (along with the indexing on IDs as in the previous queries).
The results are ordered by their score, since as in other queries, we wish to present the top songs first.

8. Returns the 10 (or less) most searched artists by the users of the web API.

```
queryMostSearchedArtists =
    SELECT Artist.artist_name AS col1, Artist.search_score AS col2
    FROM Artist
    WHERE Artist.search_score > 0
    ORDER BY Artist.search_score DESC
    LIMIT 10;
```

Since the search-score is changeable, and the query depends on a single table, we decided there was no need for optimization.

9. Returns the 10 (or less) most searched countries by the users of the web API.

```
queryMostSearchedCountries =
    SELECT Countries.country_name AS col1, Countries.search_score AS col2
    FROM Countries
    WHERE Countries.search_score > 0
    ORDER BY Countries.search_score DESC
    LIMIT 10;
```

Since the search-score is changeable, and the query depends on a single table, we decided there was no need for optimization.

10. Returns the 10 (or less) searched artists with the highest scores (> 0), as voted by the users of the web API.

```
queryMostPopularArtists =
    SELECT Artist.artist_name AS col1, CrowdFavorite.score AS col2
    FROM CrowdFavorite INNER JOIN Artist ON CrowdFavorite.artist_id = Artist.artist_id
    WHERE CrowdFavorite.score > 0
    ORDER BY CrowdFavorite.score DESC
    LIMIT 10;
```

The index on ‘Artist.artist_id’ helps make the JOIN faster.

11. Update the number of searches for an artist.

```
updateSearchCountArtist =
    UPDATE Artist
    SET Artist.search_score = Artist.search_score + 1
    WHERE Artist.artist_name = @artist_name;
```

Each time the user searches for a specific artist, we increment his search_count by one.

12. Update the number of searches for a country.

```
updateSearchCountCountry =
    UPDATE Countries
    SET Countries.search_score = Countries.search_score + 1
    WHERE Countries.country_name = @country;
```

Each time the user searches for a specific country, we increment its search_count by one.

13. Update the score of an artist based on users votes.

```
DELIMITER //
create procedure UpdateVote(
    in artist_name_in varchar(256),
    in user_score INT)
begin
    if exists (select artist_name from Artist where artist_name_in = Artist.artist_name) then
        INSERT INTO CrowdFavorite(artist_id, score)
        Values( (select artist_id from Artist where Artist.artist_name = artist_name_in), user_score )
        ON DUPLICATE KEY UPDATE
        CrowdFavorite.score = CrowdFavorite.score + user_score;
    End if;
End //
```

Since ‘CrowdFavorite’ table in empty at the beginning and we populate it according to users votes, we use INSERT … ON DUPLICATE KEY UPDATE to easily insert new rows for artist who got voted for the first time, or update their score if they already exist in the table.

Code Structure

create_schema.sql – Contains the code for creating the DB.

filler.py – Main python code used to populate the initial database. It pulls data from the various APIs, performs necessary parsing and analysis, and injects it to the DB. Full documentation of each method used is within the file itself.

fix_lyrics.py – After discovering that the lyrics provided by lyrics.ovh are inserted with a fixed erroneous prefix (sometimes containing the name of the artist, or just padded with superfluous chars), we wrote this script to clean the DB from those unnecessary additions.

server.py – The server side code. It uses flask with gevent.wsgi
to handle multiple requests from users.
Each time a user submits a request, the server connects to the DB to handle it, and redirects the result to the appropriate url.

queries.py – Queries used by the server side code. Partly documented in this file above, fully documented within the code itself.

Client side code – The web API was written in HTML with the use of Bootstrap and Javascript.

APIs

billboard.com - keeps record of all the top charted songs since 1970. Its API allows the extraction of the artists and song names who appeared in those charts, returned from a search based on charts' dates. We parsed this information, injected it to our website and expanded on it.

musicbrainz.com – collects information about songs/artists, including their membership in a band, and various aliases.

lyrics.ovh – provided the lyrics for the songs.

External Packages/Libraries

The billboard.com and musicbrainz.com APIs come in the form of external python packages. We imported them into our filler.py file and used them to extract the required data. An additional external package required for the use of billboard.com API is BeautifulSoup, which is used to parse web pages.

General Flow

The database is populated with data by filler.py. This is a one-time process, which injects the top charts information of the last 50 years.
Afterwards the website itself is accessible via http://delta-tomcat-vm.cs.tau.ac.il:40663, the description of its use is fully detailed in the user manual.
Some information in the database, mainly search and user rank statistics, can be updated by the user post the population described above.
The server reacts to each request made by the user. If the user searched by a country or an artist, it first updates the appropriate search_score before handling the actual request made by the user. The server queries the DB and then sends the result along other data needed by the API, to the client-side.