mOOd

by the meeseekses

# Website main features

1. User login and registration.
2. User home page, containing:
   1. Saved playlist history,
   2. General and custom recommendations based on previous searches and “wisdom of the crowd”.
3. Search page, containing:
   1. Track search by mood and tags,
   2. Save search to playlist,
   3. Free text search in artist names.
4. Results page, containing:
   1. List of 20 top matching tracks,
5. Track page, containing:
   1. Track name and details (album, artist),
   2. Track lyrics,

# DB design

## General considerations

1. We built our database considering the features we want, in addition to a clear logical division of entities. As a result, most joins and searches are done on keys, so that the relevant information is indexed.
2. We also added unique constraints to our database were logically needed, to make sure our data is not corrupted and the use of it giving the expected results.  
   Generally speaking, we tried to flatten our queries as much as possible to support the RDBMS optimizations.
3. In cases in which the queries complex, we did use some subqueries so that the queries are somewhat readable and possible to manage in the future (given, of course, that we did not see a big impact on performance).
4. We tried to avoid subqueries that are dependent on the parent query completely.

## DB

### Sketch

### Additional highlights:

1. Generally, we used auto-incremented IDs in most tables to more easily and effectively manage connections between entities.
2. Users\_tbl - in the absence of effective constraints in MySQL, there is a trigger validating that password hashes inserted to the DB are 32 characters long exactly.
3. Playlists\_tbl – we added a unique constraint on playlist\_name + user\_id to have a deterministic choice when users load playlist by name.
4. Playlist mood and tags – we initially considered adding the mood and tag IDs of the preliminary search to the playlist table. After some considerations, we decided against it, because a playlist can eventually consist of tracks with other moods and tags, so we will have to get them by looking into the tracks anyway.
5. Tags – we chose to use a connecting table given the many-to-many nature of the connection between tracks and tags. This table has no PK because it is not, and not expected to ever be, an entity on its own with further attributes. We do keep a unique constraint on the table, to avoid redundant data. The constraint also creates an index on the two, which is helpful for our queries.
6. Playlists – treated similar to tags, for the same reasons.
7. Track details – we debated how to divide the track details into tables. Logically, a track, an artist and an album are all individual entities. In our form of use, albums are never referenced separately from a track, but artists do have queries and searches specific for them. Given these considerations, we decided to keep the album name as an attribute of the track, but keep the artists separate.
8. Lyrics – after much debate, we chose to keep the lyrics as text in the Tracks\_tbl. As most searches only require specific fields from this table, the TEXT type enables us to keep the table lean (only stores a pointer to the text), and the overhead of retrieving the lyrics is negligible because only one line is retrieved at a time.
9. Artists\_tbl – artist names are not unique, because that is the case in real life. This required addition of artists and tracks to be coordinated.
10. ArtistsAsText\_tbl – because of InnoDB technical limitations, we were forced to create a shadow table for the artists using the MyISAM engine. To make management somewhat easier, we added a trigger that, after every insert into the original Artists\_tbl, inserts the same row into its shadow table. We created the appropriate full-text index on the shadow table.
11. Moods\_tbl – the original data we retrieved is in the form of a floating point number, which is harder to accurately search. We considered changing it to a string or multiplying it to get an integer, but decided to leave it in its original form for two reasons:
    1. Changing it to an integer might prove ineffective later, as precision levels may increase (now we have 4 digits after the decimal point, so we will multiply by 1,000. What if later we will get a mood with 5-digit precision?)
    2. Changing it to string form will prevent us from searching on it with variable precision levels. If a user didn’t get anything for his or hers exact mood, we may want to try searching for similar moods.

## Queries

Our application demands simple queries for user authentication, and complex queries for both the main feature – mood and tag based searches, and the additional recommendations we supply based on those searches. We will discuss some of the more complex queries:

### Search and add to playlist

The main feature of our app requires searching for tracks by mood and tags.  
To enable it, we have a query to create an new playlist, another query to search for the tracks, and the last query adds the tracks to the playlist.

Search by mood and tag:

SELECT tb.track\_id,

tb.mood\_id,

track\_name,

album\_name,

tb.artist\_id,

artist\_name

FROM Tracks\_tbl AS tb

JOIN Artists\_tbl

ON tb.artist\_id = Artists\_tbl.artist\_id

JOIN Moods\_tbl AS mt

ON tb.mood\_id = mt.mood\_id

JOIN TracksToTags\_tbl AS ttt

ON tb.track\_id = ttt.track\_id

JOIN Tags\_tbl

ON ttt.tag\_id = Tags\_tbl.tag\_id

WHERE ABS(danceability – {danceability}) < 0.1 AND ABS(energy – {energy}) < 0.1

AND tag\_name = {tag\_name}

GROUP BY track\_name

ORDER BY ABS(danceability - {danceability}) + ABS(energy - {energy}) ASC

LIMIT 20

The query leaves a pretty loose barrier on the difference between the mood searched and the mood of the track, in order to allow for tracks to be recommended even if none fit the user’s exact mood. We use *Order by* to make sure the most fitting tracks be recommended.  
This query was flattened completely, to avoid use of *IN* and subqueries, which seemed to have a massive cost in this case.  
A similar query enables search by mood only.

### User-specific recommendations

Other than a simple list of tracks, we also provide the user with recommendations based on his or hers current and previous searches.

Recommended tags:  
This query returns at most 5 tags that are most common in a user’s previously searched tracks.

SELECT ttt.tag\_id, tag\_name

FROM TracksToTags\_tbl AS ttt

JOIN PlaylistToTracks\_tbl AS ptt

ON ttt.track\_id = ptt.track\_id

JOIN

(

SELECT playlist\_id

FROM Playlists\_tbl

JOIN Users\_tbl

ON Playlists\_tbl.user\_id = Users\_tbl.user\_id

WHERE user\_name = {username}

) AS ptu

ON ptt.playlist\_id = ptu.playlist\_id

JOIN Tags\_tbl AS tt

ON tt.tag\_id = ttt.tag\_id

GROUP BY ttt.tag\_id

HAVING COUNT(ttt.tag\_id) >= ALL

(

SELECT COUNT(tag\_id)

FROM TracksToTags\_tbl AS ttt

JOIN PlaylistToTracks\_tbl AS ptt

ON ttt.track\_id = ptt.track\_id

JOIN

(

SELECT playlist\_id

FROM Playlists\_tbl

JOIN Users\_tbl

ON Playlists\_tbl.user\_id = Users\_tbl.user\_id

WHERE user\_name = {username}

) AS ptu

ON ptt.playlist\_id = ptu.playlist\_id

GROUP BY tag\_id

)

LIMIT 5

This query is used once for each user, so we decided to leave a subquery for finding the relevant playlists.

Recommended artist:  
This query returns the most artists that have the most tracks matching the last laylist’s moods (or one of them if there’s more than one).

SELECT art.artist\_id, artist\_name

FROM Artists\_tbl AS art

JOIN Tracks\_tbl AS tt

ON art.artist\_id = tt.artist\_id

JOIN

(

-- moods in last playlist

SELECT DISTINCT mood\_id

FROM Tracks\_tbl

JOIN

(

SELECT ptt2.track\_id

FROM PlaylistToTracks\_tbl AS ptt2

JOIN Playlists\_tbl AS pt2

ON ptt2.playlist\_id = pt2.playlist\_id

JOIN Users\_tbl AS ut2

ON pt2.user\_id = ut2.user\_id

WHERE user\_name = {username}

AND pt2.playlist\_timestamp =

(

SELECT MAX(playlist\_timestamp)

FROM Playlists\_tbl

JOIN Users\_tbl

ON Playlists\_tbl.user\_id = Users\_tbl.user\_id

WHERE user\_name = {username}

)

) AS tracks\_in\_playlist

ON Tracks\_tbl.track\_id = tracks\_in\_playlist.track\_id

WHERE mood\_id IS NOT NULL

) AS moods\_in\_pl

ON tt.mood\_id = moods\_in\_pl.mood\_id

GROUP BY art.artist\_id

HAVING COUNT(art.artist\_id) >= ALL

(

SELECT COUNT(artist\_id)

FROM Tracks\_tbl AS tt3

JOIN

(

-- moods in last playlist

SELECT DISTINCT mood\_id

FROM Tracks\_tbl

JOIN

(

SELECT ptt4.track\_id

FROM PlaylistToTracks\_tbl AS ptt4

JOIN Playlists\_tbl AS pt4

ON ptt4.playlist\_id = pt4.playlist\_id

JOIN Users\_tbl AS ut4

ON pt4.user\_id = ut4.user\_id

WHERE user\_name = {username}

This query is particularly complex, because it needs to both extract the mood from a user’s last playlist count the amount of matching tracks per artist.  
We chose to use the MAX function instead of the >= ALL syntax in finding the last playlist, since the value is in the table and not constructed from it (as it is the case with COUNT on a column).  
We also changed all usages of the *IN* function to joins, for performance reasons.  
Lastly, we chose to not flatten subqueries in this query, to leave it readable enough to make sure it works properly. We did not encounter great changes in performance when flattening it.

AND pt4.playlist\_timestamp =

(

SELECT MAX(playlist\_timestamp)

FROM Playlists\_tbl

JOIN Users\_tbl

ON Playlists\_tbl.user\_id = Users\_tbl.user\_id

WHERE user\_name = {username}

)

) AS tracks\_in\_playlist

ON Tracks\_tbl.track\_id = tracks\_in\_playlist.track\_id

WHERE mood\_id IS NOT NULL

) AS moods\_in\_pl2

ON tt3.mood\_id = moods\_in\_pl2.mood\_id

GROUP BY artist\_id

)

LIMIT 1

More songs for your last playlist:  
This query returns up to 20 tracks that match a user’s last playlist mood (but didn’t appear in that playlist).

SELECT tt.track\_id,

tt.track\_name,

tt.album\_name,

tt.artist\_id,

artist\_name

FROM Tracks\_tbl AS tt

JOIN Artists\_tbl AS art

ON tt.artist\_id = art.artist\_id

LEFT JOIN

(

-- tracks in last playlist

SELECT track\_id

FROM PlaylistToTracks\_tbl AS ptt

JOIN Playlists\_tbl AS pt

ON ptt.playlist\_id = pt.playlist\_id

JOIN Users\_tbl AS ut

ON pt.user\_id = ut.user\_id

AND user\_name = {username}

AND pt.playlist\_timestamp =

(

SELECT MAX(playlist\_timestamp)

FROM Playlists\_tbl

JOIN Users\_tbl

ON Playlists\_tbl.user\_id = Users\_tbl.user\_id

WHERE user\_name = {username}

)

) AS last\_playlist\_tracks

ON tt.track\_id = last\_playlist\_tracks.track\_id

JOIN

(

-- moods in last playlist

SELECT DISTINCT mood\_id

FROM Tracks\_tbl

JOIN

(

SELECT ptt2.track\_id

FROM PlaylistToTracks\_tbl AS ptt2

JOIN Playlists\_tbl AS pt2

ON ptt2.playlist\_id = pt2.playlist\_id

JOIN Users\_tbl AS ut2

ON pt2.user\_id = ut2.user\_id

WHERE user\_name = {username}

AND pt2.playlist\_timestamp =

(

SELECT MAX(playlist\_timestamp)

FROM Playlists\_tbl

JOIN Users\_tbl

ON Playlists\_tbl.user\_id = Users\_tbl.user\_id

WHERE user\_name = {username}

)

) AS tracks\_in\_playlist

ON Tracks\_tbl.track\_id = tracks\_in\_playlist.track\_id

Similar to the recommended artist query, this query is quite complex because it needs to both extract the mood from a user’s last playlist and make sure the tracks it returns are not in that list.  
We used similar choices in this query regarding the use of *MAX* function and refraining from the use of the *IN* function.  
We also chose to use an outer join and get the values not in the adjoined table, instead of using *NOT IN*, in order to boost performance.  
Lastly, we again chose to not flatten subqueries in this query, to leave it readable enough to make sure it works properly. We did not encounter great changes in performance when flattening it.

WHERE mood\_id IS NOT NULL

) AS moods\_in\_playlist

ON tt.mood\_id = moods\_in\_playlist.mood\_id

WHERE last\_playlist\_tracks.track\_id IS NULL

LIMIT 20

### Global recommendations

Our app also gives some recommendations based on the entire playlist history of its users.

Top artist:  
This query returns an artist that has the most tracks in user searches.

SELECT Artists\_tbl.artist\_id, artist\_name

FROM Artists\_tbl

JOIN Tracks\_tbl

ON Artists\_tbl.track\_id = Tracks\_tbl.track\_id

JOIN PlaylistToTracks\_tbl

ON Tracks\_tbl.track\_id = PlaylistToTracks\_tbl.track\_id

GROUP BY artist\_id

HAVING COUNT(artist\_id) >= ALL

(

SELECT COUNT(artist\_id)

FROM Tracks\_tbl

JOIN PlaylistToTracks\_tbl

ON Tracks\_tbl.track\_id = PlaylistToTracks\_tbl.track\_id

GROUP BY artist\_id

)

LIMIT 1

Top track:  
This query returns a track that appears the most times in user’s playlists.

SELECT Tracks\_tbl.track\_id,

track\_name,

album\_name,

Tracks\_tbl.artist\_id,

artist\_name

FROM Tracks\_tbl

JOIN Artists\_tbl

ON Tracks\_tbl.artist\_id = Artists\_tbl.artist\_id

JOIN PlaylistToTracks\_tbl

ON Tracks\_tbl.track\_id = PlaylistToTracks\_tbl.track\_id

GROUP BY track\_id

HAVING COUNT(PlaylistToTracks\_tbl.track\_id) >= ALL

(

SELECT COUNT(PlaylistToTracks\_tbl.track\_id)

FROM PlaylistToTracks\_tbl

GROUP BY track\_id

)

LIMIT 1

### Misc

We enable users to add tracks from recommendations etc. to their last playlist using the following query:

INSERT INTO PlaylistToTracks\_tbl(playlist\_id, track\_id)

SELECT playlist\_id, {track\_id}

FROM Playlists\_tbl AS pt

JOIN Users\_tbl AS ut

ON pt.user\_id = ut.user\_id

WHERE user\_name = {username}

AND playlist\_timestamp =

(

SELECT MAX(playlist\_timestamp)

FROM Playlists\_tbl AS pt2

JOIN Users\_tbl AS ut2

ON pt2.user\_id = ut2.user\_id

WHERE user\_name = {username}

)

This query was original looking for the timestamp bigger or equal to all other timestamps, and using the parent query’s user\_id in the subquery. It was changed to use the MAX function and to not have a dependency between the parent and subquery, so that the optimizer will be able to run the subquery only once.

We also enable a full text search for artists:

SELECT \*

FROM ArtistsAsText\_tbl

WHERE MATCH(artist\_name)

AGAINST({substr} IN NATURAL LANGUAGE MODE)

LIMIT 20