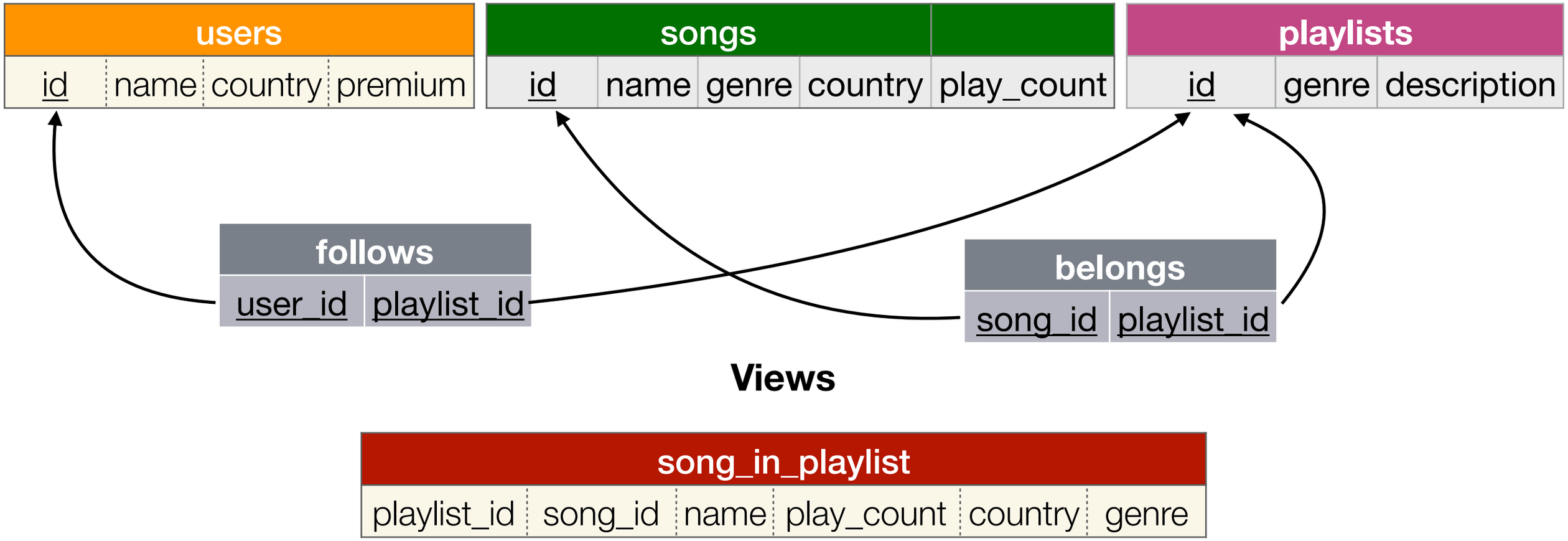
Design and short documentation



Views

song_in_playlist					
playlist_id	song_id	name	play_count	country	genre

playlist_statistics	
id	play_count

similar_users	
user_1_id	user_2_id

Comments

- One can force constraint on genre being equal for song with id *song_id* and playlist with id *playlist_id* in table **belongs** by adding dummy column *genre* to belongs table and making (*song_id,genre) and (playlist_id, genre)* foreign keys referencing (*id, genre*) in **songs** and **playlists** and *(id, genre)* unique in **songs** and **playlists**

- Other tables check required constraints in trivial way

- The numberOfCommonPlayListsSQL subquery used to build **similar_users** relation was firstly build with assumption that users that follow 0 playlists should be taken info account in getSimilarUsers so it contains 0 for users that have no playlists in common. Then constraint numberOfCommonPlayListsSQL being greater than 0 was introduces to fix the solution in a fast way.

Low level functions

- insert(String tableName, Object… arguments)
- select(String tableName, String[] columnNames, String fieldName, Object fieldValue)
- selectSingleRow(String tableName, String[] columnNames, String fieldName, Object fieldValue)
- selectAndAggregate(String aggregator, String tableName, String aggregatingFieldName, String whereName, Object whereValue)
- selectMost(String tableName, String fieldName, String groupBy, String orderBy, T emptyResult, boolean returnNullInCaseOfFailure)
- selectTop(String tableName, String fieldName, String where, String groupBy, String having, String orderBy, Integer limit, boolean returnNullInCaseOfFailure, Object… arguments)
- delete(String tableName, Pair<> fieldNamesAndValues)
- execute(String sql)
- update(String tableName, String whereFieldName, Object whereFieldValue, setFieldName, Object setFieldValue)
- clearTable(String tableName)
- dropTable(String tableName)
- dropView(String viewName)

Middle level functions

- createUsersTable()
- createSongsTable()
- createPlaylistsTable()
- createBelongsTable()
- createFollowsTable()
- createSongInPlayListView()
    - Just join tables **songs**, **belongs** and **playlists**
- createPlayListStatisticsView()
    - **playlists** left join **song_in_playlist**
    - groupby **playlist**.*id,* sum *play_count* and coalesce with 0
- createSimilarUsersView()
    - use numberOfComonPlayLists and numberOfUser1PlayLists subqueries
    - select from cross product of **users** with itself entries which satisfy the definition
- changeUserPremium(Integer userId, boolean premium)
- getInsertReturnValue(SQLException e)
- getUpdateAndDeleteReturnValue(Integer numberOfRecords, SQLException e)

High level functions
Basic database functions

- createTables
- clearTables
- dropTables

CRUD

- addUser
- getUserProfile
- deleteUser
- updateUserPremium
- updateUserNotPremium

- addSong
- getSong
- deleteSong
- updateSongName
- addPlaylist
- getPlaylist
- deletePlaylist
- updatePlaylist

Basic API

- songPlay
    - just update *playCount* with itself plus times in **songs**
- addSongToPlaylist
    - In current implementation just insert into **belongs** information selected from **songs** and **playlist** checking *genre equality* in where clause
- removeSongFromPlaylist
- followPlaylist
- stopFollowPlaylist
- getPlaylistTotalPlayCount
    - just select from **play_list_statistics** view the relevant row
- getPlaylistFollowersCount
    - just count number of rows In **follows** table when filtering by *user_id*
- getMostPopularSong
    - Select top by count song name from **song_in_playlist** grouped by *song_id* and *song_name*
- getMostPopularPlaylist()
    - Select top *playlist_id* by sum of *play_count* from **play_list_statistics**
-

Advanced API

- hottestPlaylistsOnTechnify
    - Simply select top <= 10 from **song_in_playlist** grouped by *playlist_id* ordered as defined
- getSimilarUsers
    - Simply select top <= 10 from **similar_users** view ordered as defined
- getPlaylistRecommendation
    - use similarUsers and userPlayLists subqueries returning users similar to current and playlists user follows and then just select from **follows** relation top <= 5 rows where user is one of similar group them by playlist id and then filter ones (having) not in one of playlists user follows
- getTopCountryPlaylists
    - Use isUserPremium, userCountry, playListsWithSongsInUserCountry subqueries and then just select top <= 10 playlists from **play_list_statistics** ordered as defined where user is premium and playlist has songs in his country
- getSongsRecommendationByGenre
    - User songsInPlayListsUserFollows subquery (by joining follows and belongs tables) and simply select from **songs** the ones which does not belong there but have current genre.