Linnaeus University

2DV513 - Database Theory
Assignment 3

Students
Albert TÉLLEZ
Heysem ASKAN
Student ID
at222vk, ab224hv

Contents

1 Idea

For our project we came up with a multi-functional music searching and sorting program. Our program solves a the difficulties faced by music lovers who would like to keep track of songs they like and search for them based on the lyrics, artist name and even the album name.

Thanks to this program they could add, remove and search their favourite songs, artists and albums. The target audience for this program are the people who enjoy being able to store their music and classify it. A note-worthy function is the ability to search the song by the lyrics where the user will manage to find the song name when they happen to have an earworm. If multiple users make use of the program, with a large enough database other users would be able to search for songs even if they have not added them personally.

If they remember even a word from the lyrics, they can initiate a search using that word and find the song they are looking for as well as details about the singer that sang it and details about the album the song belongs to.

2 Logical Model

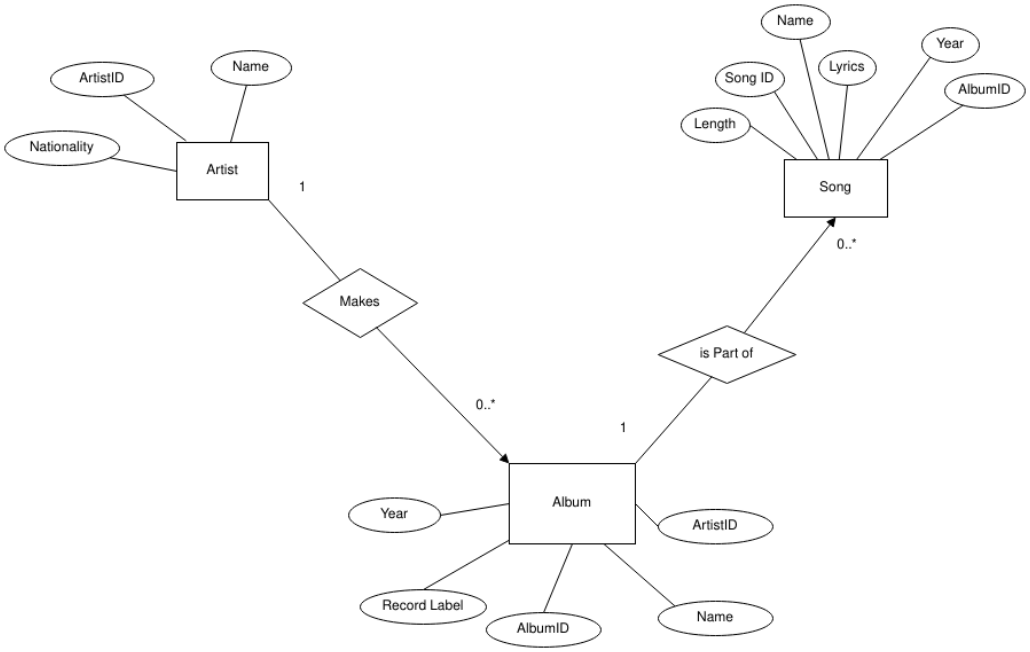

Figure 1: ER diagram of the database

Figure 1 shows the ER diagram that was used in the database. Beginning from the top left the Artist entity set can be found with an ID, so that each row could be easily identifiable. Nationalities and names were fields that would be quite reasonable information to store about an artist.

In the middle, the Album entity set can be found. This has some reasonable fields such as the year, the record label and the name of the album. The ArtistID field will allow to relate this entity set to the Artist entity set.

Finally in the top right there is a Song entity set that contains other reasonable fields, notably an album ID to relate it to the album, and a song ID that makes each row easily identifiable. The Song entity set and the Album entity set both have a "year" property, which might seem slightly redundant. This is justified due to the fact that an album might have been released in a certain year, yet the song belonging to that album might have been released several years after the album

As can be seen in the diagram the database has some sort of inverse tree structure, in the sense that one artist can have many albums, and each individual album can have many songs belonging to it.

3 Design in SQL

The ER diagram translated into the following structure in SQL.

There are three tables created in the SQL database. The first table that was created is the Artists table, since it has no other requirements. Following that the Albums table is created since it depends on the Artists table, and finally the Songs table which depends on the Albums table.

The Artists table will contain all the artists in the database. There are three attributes. Firstly an ArtistID that is the unique key, and auto increments on every addition. Secondly a country that will be represented with two digits according to ISO 3166-1 alpha-2. This field will be a varchar in the database. Another field will be the name of the artist, which will also be a varchar with a length of 50, since it was deemed unnecessary to have longer strings for the artist names.

The next table created was called “Albums” which will contain all the albums in our database. Each of them have to be belong to an artist in the database. The Albums table has 5 attributes: AlbumID, ArtistID, Year, Name and Record Label. AlbumID attribute is a unique individual ID number given to each album in the database. It is the primary key auto-incrementing attribute for the Albums table. The ArtistID attribute is the ID number of the artist that created the album, set as a foreign key. The Year attribute is the year the album was initially released. The Name attribute is the name of the album. Record Label attribute is the name of the record label that marketed the album. All the attributes except the IDs or the year, which are integers, are varchars with the length of 50.

Finally the last table created was the “Songs” table which will contain all the songs in our database. Each song has to be released in an album created by the artist that made the song. In the case of "single" releases, the album will have the same name as the release of the song.

The Songs table has 6 attributes: SongID, AlbumID, Year, Name, Length, Lyrics. SongID attribute is a unique individual id number given to each song in the database, it is the primary key auto-incrementing attribute for the Songs table. AlbumID attribute is the id number of the album that the song belongs to, configured as a foreign key. Year attribute is the year the song was released. The Name attribute is the name of the song. The Length attribute is the length of the song in seconds. Finally the Lyrics attribute contains the lyrics of the song.

All the ID attributes whether own or foreign are configured to be integers. Years and length will also be configured as integers. The rest of the variables will be configured as varchars of length 50 except the Lyrics attribute, which will be configured as a text field that can store much longer strings, as would be needed for the lyrics of a song.

4 SQL Queries

Our program consists of a number of different SQL queries, but only a few were the necessary queries needed for the assignment.

Two of our queries that implement data from more than one table are the search song by name and the search song by lyrics functionalities. Note that the black text in the queries are variables.

1. The first query, implementing the search by song name feature, will perform an inner join on all the tables, since in order to get the name of the artist that released a specific song, a link between the albums and the artists is required. This was done by linking the unique IDs that appear in each of the tables together. After this the name column is switched into lowercase, then compared with the user inputs, which are also switched to lowercase. Wildcards are added to the query so matches containing that name are also returned.

```
"SELECT Songs.SongID,`Songs`.`Name`,`Artists`.`Name` as
    `Artist name`,`Songs`.`Year`,Songs.Length FROM Songs
    inner join Albums" +
" on Songs.Album = Albums.AlbumID inner join Artists on
    Albums.Artist = Artists.ArtistID " +
" WHERE lower(`Songs`.`Name`) like " + "'%" +
    name.toLowerCase() + "%'"
```

2. The second query respectively is used to perform a case insensitive search on the lyrics of a song. This function will also perform an inner join in the three tables of the database based on the ID. Once again, the column containing the lyrics will be turned into lowercase, afterwards the input lyrics of the will be inserted into lowercase, adding wildcards to ensure the highest possibility of a match.

```
    "SELECT Songs.SongID,`Songs`.`Name`,`Artists`.`Name`
       as `Artist name`,`Songs`.`Year`,Songs.Length FROM
       Songs inner join Albums" +
    " on Songs.Album = Albums.AlbumID inner join Artists
       on Albums.Artist = Artists.ArtistID " +
    " WHERE lower(`Songs`.`Lyrics`) like " + "'%" +
       lyricsText.toLowerCase() + "%'"
```

3. Another query that uses SQL join to a lesser extent is the query that will order songs by length, and similar. In this case the database will join only the songs and the albums table, performing an inner join. This is used to show the name of the song as well as the album that specific song belongs to. Then, the ASC or DESC keywords will be used in the query to represent the order the songs should be displayed, based on the song length. Note that the "order" variable will indicate whether they should be ordered in an ascending or descending manner.

```
    "SELECT
       Songs.SongID,`Songs`.`Length`,`Songs`.`Name`,`Albums`.`Name`
       as `Album Name` FROM Songs inner join Albums on
       Songs.Album = Albums.AlbumID ORDER By Length " +
       order;
```

4. The next query included can count all the songs that each individual artist has. The query will output a table with two columns. One of the columns will have the name of the artist, and the other column will have the amount of songs that artist has. As previously mentioned, all three tables need to be joined for this to work, due to the design of the database. In this case, a right outer join is being used. The reasoning for this being used is that in the case an artist does not have albums or songs related to them, they will still be added to the result of the query. The group keyword is included in order to group the output table by the name of the artist, ensuring that they will appear only once. Utilising the keyword count on the Songs.Name column can allow us to count the amount of songs per artist. This query will print all the artists in the database, which might lead to a long or unstable output, therefore, last query would be recommended.

```
"SELECT `Artists`.`Name` as `Artist name`,
   COUNT(`Songs`.`Name`) AS `Song Amount` FROM Songs
   RIGHT OUTER join Albums on Songs.Album =
   Albums.AlbumID RIGHT OUTER join Artists on
   Albums.Artist = Artists.ArtistID GROUP By
   `Artists`.`Name`"
```

5. Finally another query where grouping was implemented would be on the query that allows the user to count the songs of a specific artist. The implementation for this query is essentially the same as for the query above, except in this scenario a "having" clause is implemented in order to match with a specific artist ID.

```
"SELECT `Artists`.`Name` as `Artist name`,
   COUNT(`Songs`.`Name`) AS `Song Amount` FROM Songs
   RIGHT OUTER join Albums on Songs.Album =
   Albums.AlbumID RIGHT OUTER join Artists on
   Albums.Artist = Artists.ArtistID GROUP By
   `Artists`.`ArtistID` HAVING Artists.ArtistID = " +
   artistID;
```

5 Supplemental video

We created a video showcasing some of the functions that we implemented in the database. The video can be found in the following link:

`https://www.youtube.com/watch?v=83ckmYkIA00`

In the submission there will also be a small database dump so that the functionality can be tried. It is to note that one small setback of adding songs is that due to the nature of command-line interfaces, pasting any input that contains a line break will trigger the "enter" function of the input.