Relational Database Project
CS4416 Final Report

Igor Kochanski - 23358459
Ciaran Whelan - 23370211
Luke Scanlon - 23390573
Emily Domini - 23362235

1 CONTRIBUTION BREAKDOWN
Work was divided evenly between the group.

2 PLATFORM

The platform used was XAMPP running on Windows 11.

3 MODIFIED SCHEMA (task 1)

3.1 Primary Keys
We made the ID values of each table the primary keys as they are unique.

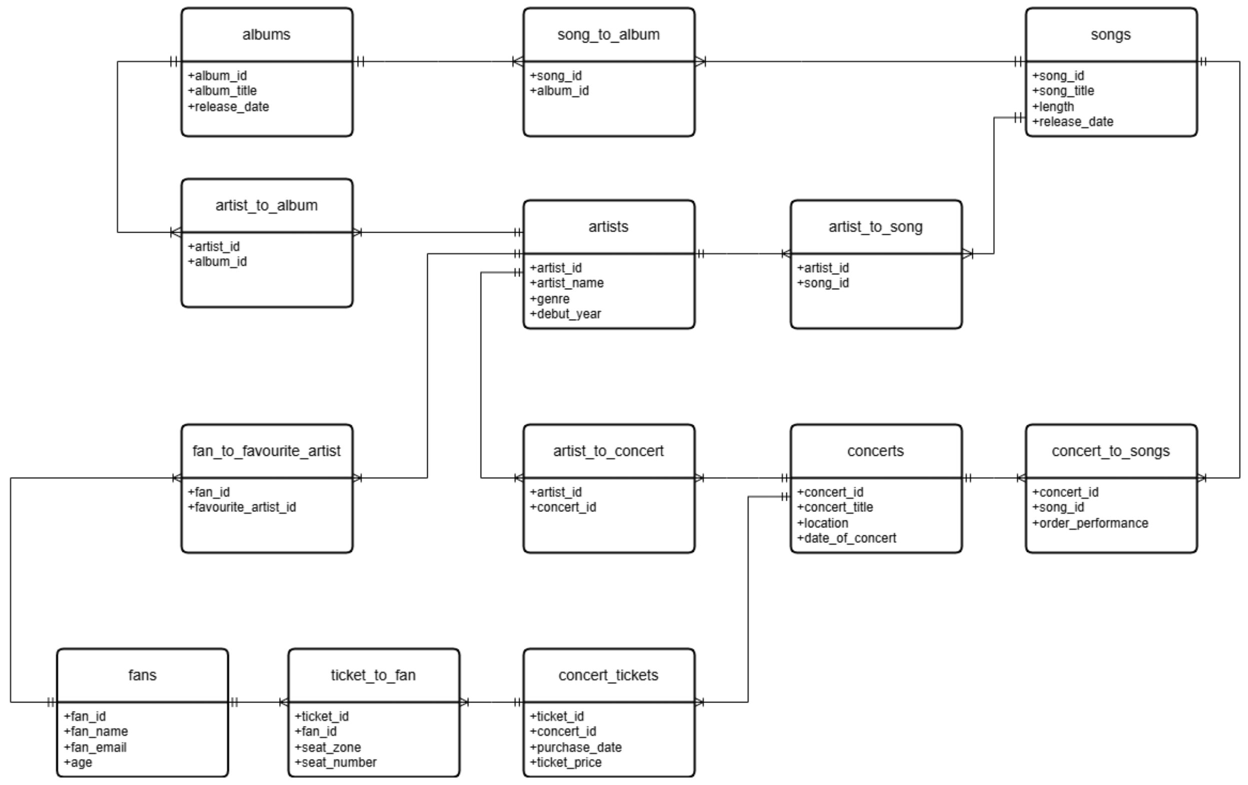
3.2 Join Tables
We created linking tables for:
- artists to (albums, songs, concerts) to allow multiple artists per each
- songs to albums to allow the same song in multiple albums
- fans to artist to allow fans to have numerous favourite artists (see 3.3)
- tickets to fans to enable multiple fans per ticket (see 3.3)

We modified the linking table:
- concerts to songs that allow multiple songs per concert and have an order (renamed, primary keys, added references)

The linking tables reference the primary keys from the two tables they link. Some values were removed from the reference tables to allow linking tables. Such as:
- artist_id from (albums, concerts)
- album_id from songs
- (favourite_artist_id, seat_zone, seat_number) from concert tickets

3.3 Concert Tickets and Fans

The concert tickets table was heavily modified. We moved the values for (fan_id, fan_name, fan_email, age) into a new table called fans. The value (favorite_artist_id) was moved to a new table that allows multiple favourite artists. The values (seat_zone, seat_number) were moved to a new table that links each ticket to one or multiple fans and each fan has their seat.



4 ENTITY-RELATIONSHIP DIAGRAM (task 2)

We created this Entity-Relationship Diagram to display our modified schema in a simple and clear manner. After we heavily modified the original schema given to us, we connected everything together with join tables and One Mandatory to Mandatory Many arrows. Almost everything connects to the artist table, so we decided to put it in the centre and have the other views stem out from it.

5 VIEW & TRIGGERS (task 3, 4)

5.1 Create View

We created a view that displays information about an artist's total time performing at concerts, along with the revenue they have earned from each concert.
The view takes the artist's name and ID from the artist table, then gets the length of songs from songs, finds which have been performed at a concert, gets the artists associated with each song, and finally sums all of them together to get the total time played for their entire career.

The last column takes the concert ticket prices, sums them together, distributes them evenly between each artist, and then sums them all up to get the total revenue an artist has made across their entire career.
The view finishes with a group-by clause so it can distinctively differentiate between each artist and provide their overall career information.

5.2 Triggers

We added a before trigger that checks the age of the fans to ensure that the fan added is above the age of 16. If the fan is underage, an error is thrown, and the data is not added to the table. This trigger is useful to prevent small children from going to concerts and being recorded in the database.

We also added an after trigger which, upon ticket deletion, will remove all related data from the ticket_to_fan table. This is very practical because it will ensure that the details of a deleted ticket aren't left behind in another table.

6 ASSUMPTIONS (tasks 5, 6)

6.1 Function

We added a function to count the seats occupied at any single concert. This function is based on counting the amount of seat number values in the ticket_to_fan that have the same concert ID.

We chose to count the seat numbers as, in our schema, one ticket can be linked to multiple seats, and we could guarantee that no two fans would be sharing the same seat.

This function is very useful to the artist and others involved in scheduling the concerts to see the artist's popularity for when they are scheduling more concerts to gauge how many people will attend to book a big enough location.

6.2 Procedure

We assumed that the brief asked for us to insert a new row into the table containing the song_id and new album_id, rather than editing the album_id of the given song.

A table must exist, containing both song_id and album_id, neither of which are primary keys, so that they can be duplicated when we add the new association.

We took it that the given song_id exists in the database, so when the procedure returns that no association was found, it was not because that song didn't exist.

We assumed that the given album_id also exists in the database so that we could retrieve its release date. If the song_id or album_id doesn't exist, an error is thrown by the DBMS.

7 DISCUSSION

We believe that indexes on artist_id and concert_id in tables such as artists, artist_to_concert, songs, concert_to_songs, and artist_to_song could enhance join efficiency and improve aggregation performance. Additionally, a composite index on (concert_id, ticket_price) in concert_tickets would speed up revenue calculations.

The CheckAge trigger could use the index of the age column of the fans table to make quick checks. The DeleteTicket trigger would benefit from using an index on the ticket_id in the ticket_to_fan table for faster deletion of associated records.

Indexes increase the performance of functions like Count(). When an index is present, the database can compute totals directly from the index, eliminating the need to scan the actual data rows. This can result in time savings, particularly when analyzing concert data.

The AddSongAlbumAssoc procedure would perform more efficiently if we added indexes on song_id and album_id in the song_to_album table to check if the given association already exists. This would eliminate the need for that same check when the procedure is called. We could also use indexes on release_date in both the songs and albums tables to find the release dates quickly within the procedure.