CS4416 Group 13 Project Report

Contributions breakdown

James Connolly 23388102	25%
Enda Buckley 23358156	25%
Andrei Birsan 23304847	25%
Adam Urban 23381752	25%

Platform used

XAMPP on win11

Task 1: Description of changes

Artists

In the original schema, the albums, songs, and concerts tables each have an artist_id column, indicating that each album, song, and concert is associated with only a single artist. This will not work for our new goal of eliminating data redundancy, and adding support for multiple artists per album, song, and concert, multiple favourite artists per fan, and eventually a system to implement shared tickets.

Our first modification to the schema was to add the PRIMARY KEY Characteristic to the Artists Table. This is done to ensure that each artist has a unique identifier in the database, and to later promote a many-to-many relationship being common within our database.

This also guarantees that no two artists will have the same artist_id, and promotes non-nullability and uniqueness which will prevent further issues down the line.

Also to note is the implementation of the new field total_revenue, this will be directly relevant in the creation of our AFTER trigger but for now it shouldn't be worried about. Just know that it is assigned the value 0 by default.

Album

Next we modified the Albums table to also contain the PRIMARY KEY Characteristics for the same reasons mentioned above with the Artists table. However with this table we also removed the Artist_Id as the previous implementation which included it would allow each album to only be associated with one artist.

Artist_album

However the artist who wrote the Album is still important info and can't simply be omitted. For this reason we elected to create a new table entirely to accomplish this goal. We named this table artist_album signifying that it will contain the relationship between each artist and album. This allows an album to have multiple artists and an artist to appear on multiple albums. Next we create a composite primary key consisting of album_id and artist_id. This is done so that Each combination of album_id and artist_id is ensured to be unique, and that the same artist cannot be linked to the same album more than once as this would be redundant and go against our goal of elimination redundancy.

Next we established a foreign key constraint on the album_id column, which essentially just specifies that the values in album_id must correspond to a valid album_id value in the albums table. This ensures an artist cannot be linked to a non-existent album. We then implemented a foreign key constraint on the artist_id column as well, doing the same thing we did with album_id, ensuring that an album cannot be linked with a non-existent artist.

Songs

The only addition to the songs table is the addition of a PRIMARY KEY (for the same reasons as the previous tables) and a FOREIGN KEY constraint on the album_id. This is done to ensure that a song cannot be associated with an album that does not exist. We will link a song and artist through the creation of a new table described below.

Song_artists

This table does not require much description, as it is essentially the same thing that we did for the artist_album table described above. This table establishes a many-to-many relationship between songs and artists, enabling multiple artists to be associated with a single song and vice versa.

In this table, we create a composite PRIMARY KEY consisting of both song_id and artist_id. This ensures that the combination of song_id and artist_id is unique, preventing the same artist from being linked to the same song more than once, thus eliminating redundancy.

Additionally, FOREIGN KEY constraints are imposed on both the song_id and artist_id fields. The FOREIGN KEY constraint on song_id ensures that each song is linked to a valid song in the songs table, and the FOREIGN KEY constraint on artist_id ensures that each artist is linked to a valid artist in the artists table.

Concerts

Next we address the concerts table. We remove the artist_id field in order to satisfy our condition of having multiple artists per concert. This relation will instead be represented with the creation of a new table below. Similarly a PRIMARY KEY is added once again for the same reasons as described in the artist table.

Concerts_artists

This table, much like song_artists and artist_album, establishes a many-to-many relationship between concerts and artists, enabling multiple artists to be associated with a single concert and vice versa. It imposes foreign key constraints on both the concert_id and artist_id fields for the same reasons as the above tables.

Concert_songs

Next we address the concert_songs table. This table never had the artist_id field, so we do not have to remove anything. We instead add a PRIMARY KEY, This change is made to prevent data duplication and ensure that each song is uniquely associated with a specific concert, following the same rationale as the addition of a primary key in the artists table.

Concert_tickets

Next we address the concert_tickets table. This table previously contained fan specific data such as fan id, email, name, and age. These fields have all been removed under the new schema to meet the criteria of allowing ticket sharing. We then remove ticket_id, and add a PRIMARY KEY Constraint to ensure that each ticket has a unique identifier. We then add a FOREIGN KEY Constraint on concert_id. This is done to link each ticket to a specific concert in the

concerts table. This also ensures that only valid concert IDs can be inserted into the concert_tickets table.

Fans

We created a fans table in order to separate fan specific info from the ticket system. This simply contains the information that was cut from concert_tickets in order to make it compatible with the requirement of ticket sharing. It also includes a PRIMARY KEY Constraint on the fan_id field to ensure that no two fans have the same id.

Fans_ticket

We next created a table titled fans_ticket. This table addresses the need to track which fans have purchased tickets for a particular concert. This table supports the many-to-many relationship between fans and tickets. In other words, a single ticket can be shared by multiple fans, and a fan can have tickets for multiple concerts. To achieve this, the fans_ticket table includes a composite PRIMARY KEY consisting of ticket_id and fan_id. This ensures that the combination of ticket_id and fan_id is unique, preventing duplicate entries of the same fan for a given ticket. We then added a FOREIGN KEY Constraint on ticket_id and fan_id. This was done to ensure that each ticket_id in the fans_ticket table must already exist in the concert_tickets table, linking the fan to a valid ticket. Similarly, the fan_id must correspond to a valid fan in the fans table.

Fan_favourite_artists

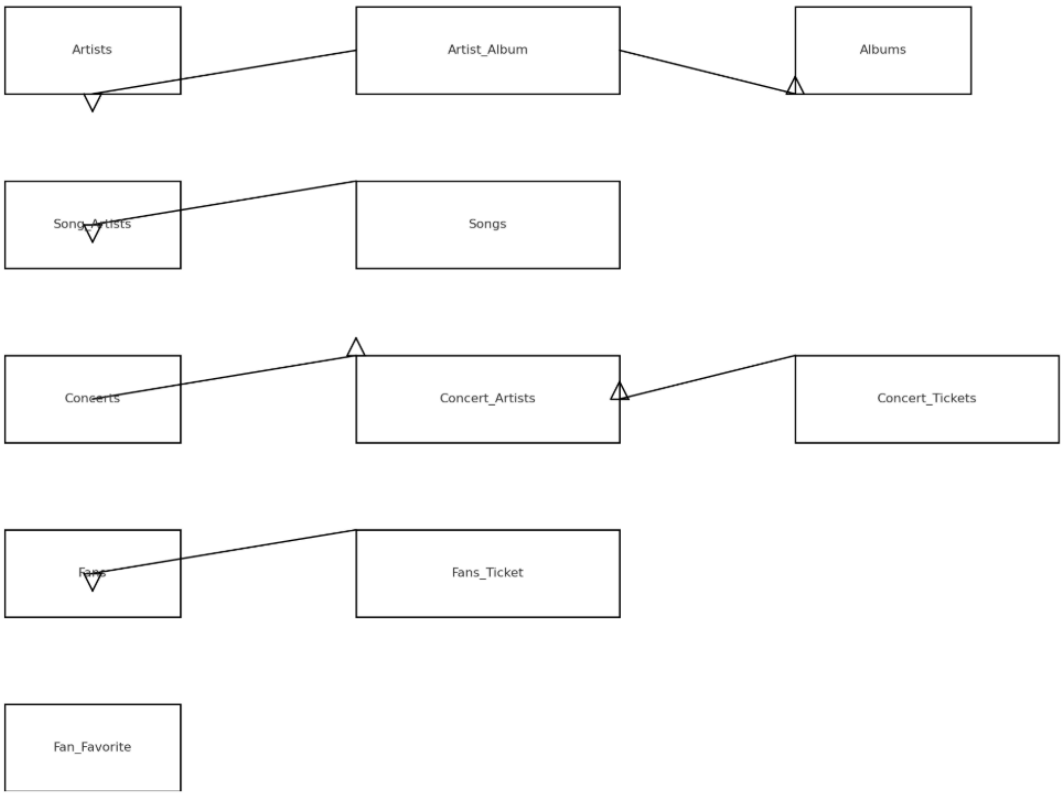
Finally, we created the table fan_favorite_artists. This table addresses the need to support multiple favorite artists per fan. In the original schema, there was a favourite_artist_id field in the concert_tickets table, which limited each fan to only one favorite artist. We created the fan_favorite_artists table to represent a many-to-many relationship between fans and their favorite artists in order to eliminate this limitation and allow fans to have multiple favorite artists. The table includes a composite PRIMARY KEY consisting of fan_id and artist_id. This ensures that each fan-artist combination is unique, preventing duplication of favorite artists for the same fan. Additionally, FOREIGN KEY constraints are added to both the fan_id and artist_id columns. These constraints ensure that the fan_id in the fan_favorite_artists table must correspond to a valid fan in the fans table, and that the artist_id must correspond to a valid artist in the artists table. This allows for the seamless association of fans with their favorite artists while maintaining referential integrity within the database.

That concludes our initial modification of the database. Next we wrote the insert into statements. We elected to go with The Beatles as one artist since they are timeless, and EricDOA (a more niche artist) as the second as James and Adam are big fans.

We then added some of their albums, songs, concerts, and their necessary relations with each other, as well as tickets and sample fans. We added two of each category in keeping with the project specification documents requirements.

Task 2: ERD Diagram

Below is the Entity Relation Diagram (ERD) in crow's foot notation. It should be noted that one to many is represented by a straight line at one end which indicates 1, and a crow's foot at the other end which indicates many. The reverse is true for a many to one relationship where the crow's foot is at the starting entity, and the straight line is at the related entity. And with a many to many relationship both ends of the line have crow's feet, representing that many instances of one entity can relate to many instances of another entity.



Entities in the ERD:

Artists - Represents the musical artists or bands.
Attributes: artist_id (Primary Key), artist_name, genre, debut_year, and total_revenue (new column added).

Primary Key: artist_id

Assumption: Each artist is uniquely identified by artist_id. The addition of the total_revenue field assumes that we need to track the cumulative revenue generated from concerts for each artist, which will be updated by a trigger after ticket sales.

Albums - Represents the albums released by artists.
Attributes: album_id (Primary Key), album_title, and release_date.

Primary Key: album_id

Assumption: An album belongs to a single artist, but the ERD also allows for the possibility of collaborative albums.

Artist_Album - Models the many-to-many relationship between artists and albums.
Attributes: album_id, artist_id.

Primary Key: Composite of album_id and artist_id

Assumption: An artist can have multiple albums, and an album can be associated with multiple artists (collaborations).

Songs - Represents songs in the system.
Attributes: song_id (Primary Key), album_id (Foreign Key), song_title, length, and release_date.

Primary Key: song_id

Assumption: Each song belongs to a specific album, but a song may be performed by multiple artists (via the song_artists table).

Song_Artists - Models the many-to-many relationship between songs and artists.
Attributes: song_id, artist_id.

Primary Key: Composite of song_id and artist_id

Assumption: Multiple artists can perform a song, and a song can involve multiple artists, such as collaborations or featured artists.

Concerts - Represents a concert event.
Attributes: concert_id (Primary Key), concert_title, location, and date_of_concert.

Primary Key: concert_id

Assumption: Each concert can feature multiple artists and may include several songs performed in a specific order.

Concert_Artists - Models the many-to-many relationship between concerts and artists.
Attributes: concert_id, artist_id.

Primary Key: Composite of concert_id and artist_id

Assumption: Multiple artists can perform in a single concert, and an artist may perform at multiple concerts.

Concert_Songs - Represents the many-to-many relationship between concerts and songs, including the order in which they are performed.
Attributes: concert_id, song_id, order_performance.

Primary Key: Composite of concert_id and song_id

Assumption: A concert can feature multiple songs, and each song may be performed at different concerts. The order_performance attribute helps determine the sequence of songs.

Concert_Tickets - Represents tickets sold for concerts.
Attributes: ticket_id (Primary Key), concert_id (Foreign Key), seat_zone, seat_number, purchase_date, and ticket_price.

Primary Key: ticket_id

Assumption: Each ticket corresponds to a specific concert, and the ticket_price can vary by seat zone. This table allows us to track individual tickets and the revenue generated from each sale.

Fans - Represents the fans who attend concerts or follow artists.
Attributes: fan_id (Primary Key), fan_name, fan_email, and age.

Primary Key: fan_id

Assumption: Each fan has a unique identifier (fan_id) and may purchase tickets or follow their favorite artists.

Fans_Ticket - Represents the many-to-many relationship between fans and the tickets they purchase.
Attributes: ticket_id, fan_id.

Primary Key: Composite of ticket_id and fan_id

Assumption: A fan can purchase multiple tickets, and a ticket can be purchased by more than one fan (group purchases, family, etc.).

Fan_Favorite_Artists - Models the many-to-many relationship between fans and their favorite artists.
Attributes: fan_id, artist_id.

Primary Key: Composite of fan_id and artist_id

Assumption: A fan can have multiple favorite artists, and an artist can have many fans following them.

Relationships in the ERD:

One-to-Many:

An artist can have multiple albums.
An album can have multiple songs.
A concert can have multiple concert tickets.
A fan can purchase multiple concert tickets.

Many-to-Many:

Artists can appear in multiple albums, and albums can have multiple artists (via the artist_album table).
Artists can perform in multiple concerts, and a concert can feature multiple artists (via the concert_artists table).
Songs can be performed in multiple concerts, and a concert can feature many songs (via the concert_songs table).
Fans can follow multiple artists, and artists can have many fans (via the fan_favorite_artists table).

Task 3: View of modified schema

Below is the view we have written. This view shows popular tours that the artist has performed based on total ticket revenue.

```
CREATE VIEW popular_artist_tours AS
SELECT
    a.artist_name,
    SUM(ct.ticket_price) AS total_revenue
FROM
    artists a
JOIN concert_artists ca ON a.artist_id = ca.artist_id
JOIN concerts c ON ca.concert_id = c.concert_id
JOIN concert_tickets ct ON c.concert_id = ct.concert_id
GROUP BY
    a.artist_name
HAVING
    SUM(ct.ticket_price) > 200;
```

This view calculates the total revenue generated by each artist from ticket sales across all the concerts they have performed in. This is done using the SUM(ct.ticket_price) function, which totals the ticket prices for all tickets sold at concerts associated with a given artist.

The view combines data from multiple tables using artists, concert_artists, concerts, and concerts_tickets. The Data is then grouped to ensure each row in the view corresponds to a unique artist and their total revenue. The Data is then filtered to include only those artists whose total revenue exceeds 200, identifying their tours as financially successful/popular. It should be noted that 200 has been chosen as an arbitrary number and is not tied to any currency, and would very likely be changed by the user of such a statement to better fit their situations. For example a huge record label may categorise a 500,000 result as popular/successful, however a smaller label or independent artist contractor may categorise 500 as popular/successful.

This view is useful as it presents critical information regarding an artist's financial success from live performances. It may allow management, record labels, etc to quickly assess which artists are generating the most revenue from their tours. Additionally, by using the GROUP BY and HAVING clauses,

the view is able to focus on the artists with significant tour earnings, making the information more relevant for aiding business decisions such as which artists should go on tour, when and where based off of prior concert earnings, and which artist tours may be most positively received and profitable.

Task 4: BEFORE And AFTER Triggers

BEFORE

```
DELIMITER $$

CREATE TRIGGER before_insert_concert_tickets
BEFORE INSERT ON concert_tickets
FOR EACH ROW
BEGIN
    -- Check if the ticket price is greater than the allowed maximum ($500)
    IF NEW.ticket_price > 500 THEN
        -- If it is, set it to the maximum allowed price
        SET NEW.ticket_price = 500;
    END IF;
END$$

DELIMITER ;
```

AFTER

```
DELIMITER $$

CREATE TRIGGER after_insert_concert_tickets
AFTER INSERT ON concert_tickets
FOR EACH ROW
BEGIN
    DECLARE artist_id INT;

table
    SELECT ca.artist_id
    INTO artist_id
    FROM concert_artists ca
    WHERE ca.concert_id = NEW.concert_id
    LIMIT 1;


    UPDATE artists
```

```
    SET total_revenue = IFNULL(total_revenue, 0) + NEW.ticket_price
    WHERE artist_id = artist_id;
END$$

DELIMITER ;
```

Before explained:

The before_insert_concert_tickets trigger makes sure that ticket prices for concerts remain within a reasonable limit by automatically enforcing a maximum price cap of $500. Before any new ticket is inserted into the concert_tickets table, the trigger checks the ticket_price of the new row being added. If the price exceeds $500, the trigger modifies the value to exactly $500 before the insertion occurs. A Delimiter is used in this function to prevent the possibility of SQL language from identifying the semicolon (;) as the end of a statement when it is not intended to be. In our testing purpose we temporarily set the delimiter to ## as it has a minimal chance of ever coming up in code.

This trigger is useful for maintaining consistency and enforcing business rules related to ticket pricing. Even in the real world some event organizers and platforms alike set price caps to prevent excessively high ticket prices. This is often done to prevent negative customer perception. By using this trigger, you ensure that no ticket is sold above a certain price, without requiring manual intervention or checks every time a new ticket record is inserted. This could be very useful to any big ticketing platform.

After explained:

The after_insert_concert_tickets trigger updates an artist's total revenue whenever a new ticket is added to the concert_tickets table. After the insertion of a ticket, the trigger identifies the artist_id associated with the concert by looking up the concert_id in the concert_artists table. It then updates the total_revenue column in the artists table by adding the price of the newly inserted ticket to the artist’s current total revenue. If the total_revenue is null, it initializes it to zero before performing the update. This ensures the artist's revenue stays accurate and up-to-date after each ticket sale. A Delimiter is also used in this function to prevent the possibility of SQL language from identifying the semicolon (;) as the end of a statement when it is not intended to be. In our testing purpose we temporarily set the delimiter to ## as it has a

minimal chance of ever coming up in code. It should also be noted that it is this trigger that necessitated the implementation of this line in the artists table “ total_revenue DECIMAL(10, 2) DEFAULT 0”, this was because we needed a field that could be updated with the response.

This trigger is a valuable asset in the process of revenue tracking. Instead of manually updating the total revenue for each artist after every ticket sale, the trigger performs this task automatically which ensures real-time updates to the artist's earnings. This helps in maintaining accurate financial records and ensures the data can always be used for accurate and sometimes split second decision-making such as deciding if a venue has made enough money to be considered profitable enough to proceed with the show. Without this trigger, the artist's total revenue would need to be manually adjusted each time a ticket is sold, which is both inefficient and prone to errors, as well as time consuming.

Task 5: Stored function

```
DELIMITER $$

CREATE FUNCTION total_occupied_seats(concert_id_input INT)
RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE seat_count INT;

    -- Initialize the count to 0
    SET seat_count = 0;

    -- Calculate the total number of tickets sold for the given concert_id
    SELECT COUNT(ticket_id)
    INTO seat_count
    FROM concert_tickets
    WHERE concert_id = concert_id_input;

    -- Return the total number of occupied seats
```

```
    RETURN seat_count;
END$$

DELIMITER ;
```

This function as required in the project specification calculates the total number of tickets sold for a specific concert based on its concert_id. It begins by declaring and initializing a variable, seat_count, to store the result. Our function then performs a SELECT COUNT(ticket_id) query on the concert_tickets table, filtering rows based on the input concert_id. The result of this query is stored in the seat_count variable. Finally, the function returns the value of seat_count as the output, showing the total number of occupied seats for the passed concert.A Delimiter is also used in this function to prevent the possibility of SQL language from identifying the semicolon (;) as the end of a statement when it is not intended to be. In our testing purpose we temporarily set the delimiter to ## as it has a minimal chance of ever coming up in code.

Task 6: Is song associated with album check

```
DELIMITER $$

CREATE PROCEDURE check_and_add_song_album(
    IN input_song_id INT,
    IN input_album_id INT
)
BEGIN
    DECLARE album_release_date DATE;
    DECLARE song_release_date DATE;
    DECLARE song_exists INT;

    SELECT COUNT(*)
    INTO song_exists
    FROM songs
    WHERE song_id = input_song_id AND album_id = input_album_id;

    IF song_exists = 0 THEN
        UPDATE songs
        SET album_id = input_album_id
```

```
        WHERE song_id = input_song_id;
    END IF;

    SELECT release_date INTO album_release_date FROM albums WHERE
album_id = input_album_id;
    SELECT release_date INTO song_release_date FROM songs WHERE
song_id = input_song_id;

    IF song_release_date > album_release_date THEN
        UPDATE songs
        SET release_date = album_release_date
        WHERE song_id = input_song_id;
    END IF;
END$$

DELIMITER ;
```

This stored procedure checks if a given song_id is associated with a given album_id, and can also adjust the songs release date if it was released after the album its in. First, it checks whether the given song (input_song_id) is already associated with the specified album (input_album_id) by querying the song_album mapping table we created in Task 1. If no existing association is found, the procedure proceeds to insert a new record into the table, linking the song and the album. Following this, the procedure checks if the song's release date is later than the album's release date. If this is the case, it updates the song's release date to match that of the album. This is done as you can not have a song released after the album it was supposed to be released in. Similarly to the above functions, a delimiter is used to prevent the possibility of SQL language from identifying the semicolon (;) as the end of a statement when it is not intended to be. In our testing purpose we temporarily set the delimiter to ## as it has a minimal chance of ever coming up in code.

Assumptions made during this project

Throughout the course of this project, some assumptions were made to make the creation of the database and functions easier. These assumptions are mostly reasonable and could be verified if the project called for more in depth error handling. When writing the stored function and procedure, several assumptions were made to ensure their functionality and simplicity. For the function total_occupied_seats, it was assumed that all records in the concert_tickets table are valid and represent legitimate ticket purchases, with no duplicates in seat assignments for a particular concert. Additionally, it was assumed that the concert_id provided as input would always correspond to an existing concert and that the function would not account for potential issues such as missing, invalid, or incorrectly formatted data. The procedure check_and_add_song_album assumes that the song_album table exists and enforces a one-to-one relationship between songs and albums, meaning each song can only be associated with a single album. It also presumed that both song_id and album_id provided as inputs are valid and exist within their respective tables. As mentioned above, error handling was minimal, with the expectation that any errors would be dealt with outside the functions or procedures, as they were designed to focus solely on the core features necessary for a concert. These assumptions were made to keep the operations simple and efficient, although in a real-world application, additional checks, validation, and error management would likely be necessary.

Note on Indexes:

To ensure the efficient execution of the view, triggers, function, and procedure in the modified schema, indexing could be rather useful. For the view “popular_artist_tours”, indexes could be created on columns frequently used in the joins, such as concert_id, artist_id, and ticket_price in their respective tables. This could greatly speed up the join operations and improve performance when calculating total revenue for each artist. For the before and after triggers, indexes on concert_id and artist_id could pretty effectively improve the lookup and update speeds, definitely so when checking and modifying related artist revenue. In the function “total_occupied_seats”, indexing concert_id in the concert_tickets table could expedite the count of occupied seats for a specific concert. Similarly, for the procedure

“check_and_add_song_album”, indexing song_id and album_id in the song_album relationship table could allow for more efficient checks and updates when associating songs with albums. However, it should be noted that while indexes improve read performance, they can slow down insertions and updates, especially in high-transaction environments, so indexing should be applied judiciously based on the most critical operations. A concert would be liable to having many insertions and updates in a short period of time, such as when new concert tickets are being inserted to the table directly after being released for presale or general sale. For this reason indexing may be a double edged sword and the pros and cons should be considered at each implementation. Furthermore they also consume additional storage space, which a smaller venue may not have to spare. This along with the fact that if tables experience many modifications then indexes might need to be constantly rebuilt or reorganized, which leads to longer maintenance times. For these reasons, we believe indexing to be generally advisable against in an industry liable to quick change such as the music concert industry. It would be different if the industry were more static and less prone to change but as it stands now indexing could just as likely be a determinant as a help in this specific database.

Conclusion

This project demonstrates the design and implementation of a comprehensive database system tailored to the music industry, specifically focusing on managing concerts, artist information, ticket sales, and revenue tracking. By minimizing redundancy and using efficient relationships between entities, the database is set up to handle complex queries and also has the added benefit of having decent future scalability. This modified Databases’s key features include the integration of multiple artists per concert, album, and song, as well as multiple favourite artists per user, and finally ticket sharing among fans. This system also has the impressive and useful dynamic calculation of artist revenue per concert feature. The addition of triggers and views ensures automated revenue updates and efficient data retrieval.