CSA0593

DATABASE MANAGEMENT SYSTEM.

NAME : S.DEEPIKA

REG NO. : 192324269

ASSIGNMENT - 2

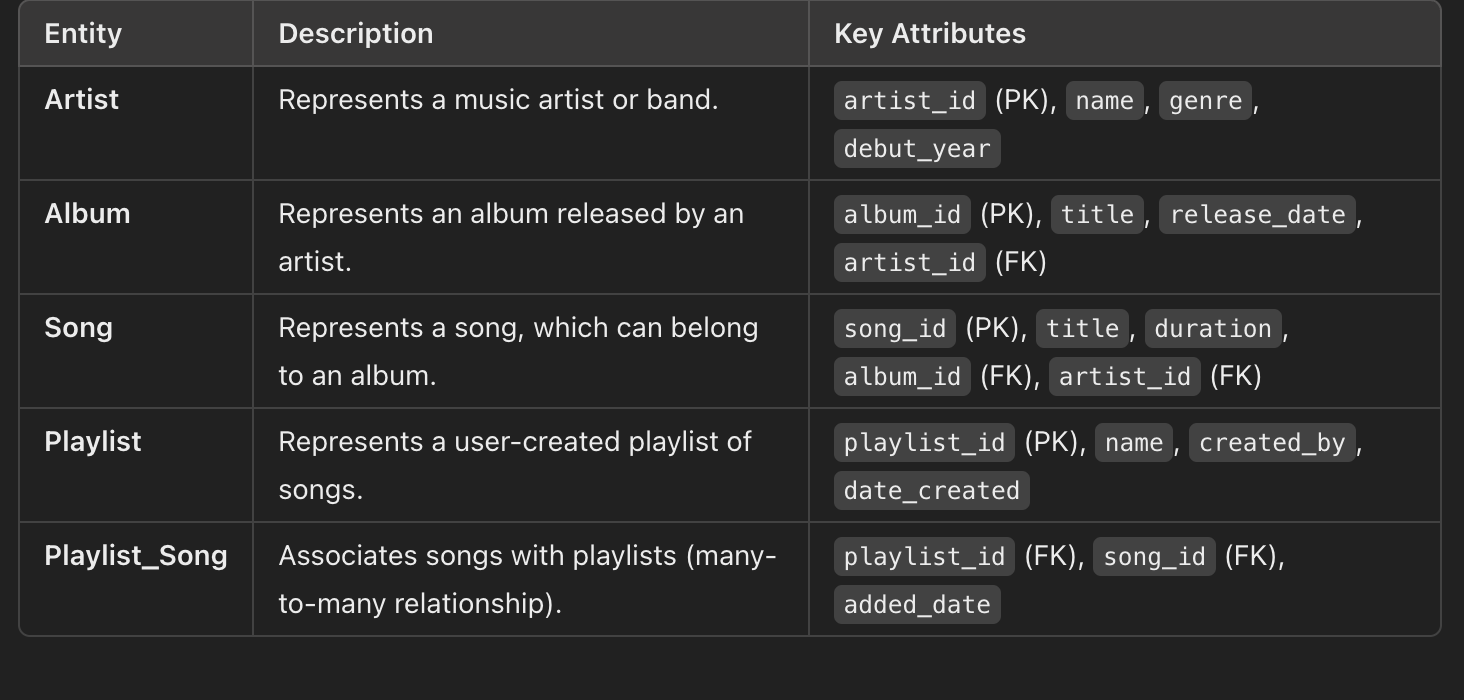
**2) Create a database for managing songs, albums, artists, and playlists.  
a.)Model tables for songs, albums, artists, and playlists.  
b.)Write stored procedures for adding songs to playlists and albums.  
c.)Implement triggers to update play counts and song rankings.  
d.)Write SQL queries to analyse song popularity and user activity.**

1. **Create a database for managing songs, albums, artists, and playlists**

To create a database for managing songs, albums, artists, and playlists, here’s a detailed relational design in table format that includes entities, attributes, and relationships:

1. **Entity Definitions**: Defining songs, albums, artists, and playlists as entities.
2. **Relationships**: Modelling relationships between these entities.
3. **Database Tables**: Outlining the structure of the tables with attributes, primary keys (PK), and foreign keys (FK).

**DATABASE DESIGN:-**

****

**b.)Write stored procedures for adding songs to playlists and albums.**

**Stored Procedure 1: Adding a Song to an Album**

This stored procedure adds a song to a specific album. It takes the song details and the album ID as inputs. If the album exists, it inserts the song into the Song table with the specified album ID.

**SQL CODE:**

DELIMITER //

CREATE PROCEDURE AddSongToAlbum(

IN song\_title VARCHAR(255),

IN song\_duration TIME,

IN album\_id INT,

IN artist\_id INT

)

BEGIN

-- Check if the album exists

IF EXISTS (SELECT 1 FROM Album WHERE album\_id = album\_id) THEN

-- Insert the song into the Song table

INSERT INTO Song (title, duration, album\_id, artist\_id)

VALUES (song\_title, song\_duration, album\_id, artist\_id);

ELSE

-- If album does not exist, raise an error

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Album does not exist';

END IF;

END //

DELIMITER ;

**Stored Procedure 2: Adding a Song to a Playlist**

This stored procedure adds an existing song to a specified playlist. It takes the song ID and playlist ID as inputs. If both the playlist and song exist, it creates a record in the Playlist\_Song table to link the song to the playlist.

**SQL CODE:**

DELIMITER //

CREATE PROCEDURE AddSongToPlaylist(

IN song\_id INT,

IN playlist\_id INT

)

BEGIN

IF EXISTS (SELECT 1 FROM Song WHERE song\_id = song\_id) THEN

IF EXISTS (SELECT 1 FROM Playlist WHERE playlist\_id = playlist\_id) THEN

INSERT INTO Playlist\_Song (playlist\_id, song\_id, added\_date)

VALUES (playlist\_id, song\_id, NOW());

ELSE

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Playlist does not exist';

END IF;

ELSE

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Song does not exist';

END IF;

END //

DELIMITER ;

**c.)Implement triggers to update play counts and song rankings.**

To track play counts and dynamically adjust song rankings based on play counts, we can implement triggers in the database:

1. \*\*Trigger to Update Play Counts\*\*: This trigger increments the play count every time a song is played.

2. \*\*Trigger to Update Song Rankings\*\*: This trigger updates the ranking of a song based on its play count.

**Step 1: Create the `PlayHistory` Table**

The `PlayHistory` table will store records every time a song is played, which allows us to use this data to trigger the play count and ranking updates.

**SQL CODE:**

CREATE TABLE PlayHistory (

play\_id INT AUTO\_INCREMENT PRIMARY KEY,

song\_id INT,

played\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (song\_id) REFERENCES Song(song\_id)

);

**Step 2: Trigger to Update Play Counts**

This trigger increments the `play\_count` in the `Song` table every time a new record is added to the `PlayHistory` table.

**SQL CODE:**

DELIMITER //

CREATE TRIGGER UpdatePlayCount

AFTER INSERT ON PlayHistory

FOR EACH ROW

BEGIN

-- Increment the play count in the Song table for the played song

UPDATE Song

SET play\_count = play\_count + 1

WHERE song\_id = NEW.song\_id;

END //

DELIMITER ;

**Step 3: Trigger to Update Song Rankings**

This trigger updates the ranking of a song based on its play count. The ranking criteria can be set based on specific thresholds. For simplicity, let's say:

- \*\*Rank 1\*\*: `play\_count` >= 1000

- \*\*Rank 2\*\*: `play\_count` >= 500

- \*\*Rank 3\*\*: `play\_count` >= 100

- \*\*Rank 4\*\*: `play\_count` < 100

**SQL CODE:**

DELIMITER //

CREATE TRIGGER UpdateSongRanking

AFTER UPDATE ON Song

FOR EACH ROW

BEGIN

DECLARE new\_rank INT;

IF NEW.play\_count >= 1000 THEN

SET new\_rank = 1;

ELSEIF NEW.play\_count >= 500 THEN

SET new\_rank = 2;

ELSEIF NEW.play\_count >= 100 THEN

SET new\_rank = 3;

ELSE

SET new\_rank = 4;

END IF;

IF NEW.ranking <> new\_rank THEN

UPDATE Song

SET ranking = new\_rank

WHERE song\_id = NEW.song\_id;

END IF;

END //

DELIMITER ;

**d.)Write SQL queries to analyse song popularity and user activity.**

**Query to Get Top 10 Most Popular Song**

SELECT

s.song\_id,

s.title AS song\_title,

a.name AS artist\_name,

s.play\_count

FROM

Song s

JOIN

Artist a ON s.artist\_id = a.artist\_id

ORDER BY

s.play\_count DESC

LIMIT 10;

### Query to Get Most Active Users

SELECT

u.user\_id,

u.username,

COUNT(ph.play\_id) AS play\_count

FROM

User u

JOIN

PlayHistory ph ON u.user\_id = ph.user\_id

GROUP BY

u.user\_id, u.username

ORDER BY

play\_count

LIMIT 10;

**CONCLUSION :**

In conclusion, the database design and SQL queries outlined for a music streaming platform provide a robust framework for analyzing song popularity and user activity. Through carefully designed tables, triggers, and indexes, this system can efficiently manage and query large datasets, enabling real-time tracking of play counts, rankings, and activity.