WolfMedia

media streaming service

CSC 540 Database Management Systems
Project 3

Aastha Singh: asingh59
Kalyan Karnati: kkarnat
Kritika Javali: ksjavali
Nagaraj Madamshetti: nmadams

Date: April 13, 2023

Assumptions:

1. Media streaming services handle all the payments of the entire system.
2. Each song has a track number associated with the parent album so that it can be used to listen in a particular order.
3. Each song is associated with only one main artist and multiple collaborations are possible.
4. Each podcast is owned by only one podcast host and a podcast episode belongs to only one podcast.
5. Each artist is contracted with only one record label at any given point in time.
6. An artist belongs to only one country at any given time, and also a podcast has only one origin country at a time.
7. A song can only be in one language and can be in multiple genres.
8. An artist has only one primary genre.
9. A song can only be part of only one album and an album can have multiple songs.
10. A podcast can belong to multiple genres.

Q1. Submit revised versions of the previous reports - you will get credit for the improvements, scaled by 50%. You will need to submit both the relevant *numbered* pages of the original reports and your revisions; the revisions should (1) mention the item and page numbers in the original report and (2) have the improved parts highlighted.

Answer:

Corrections Report 1

Q2. Intended Users: Page 3

> **Artists:** They are one of the users of the database, they are responsible for publishing songs under the record label. They also take part in collaborating with other artists in singing songs.
>
> **Podcast Host:** They are a class of database users who create podcasts and podcast episodes. They also allow guests to collaborate with them in podcast episodes.
>
> **Record Label:** make contracts with artists. Assign artists to albums and songs. Also responsible for the collaborations of artists in different albums.

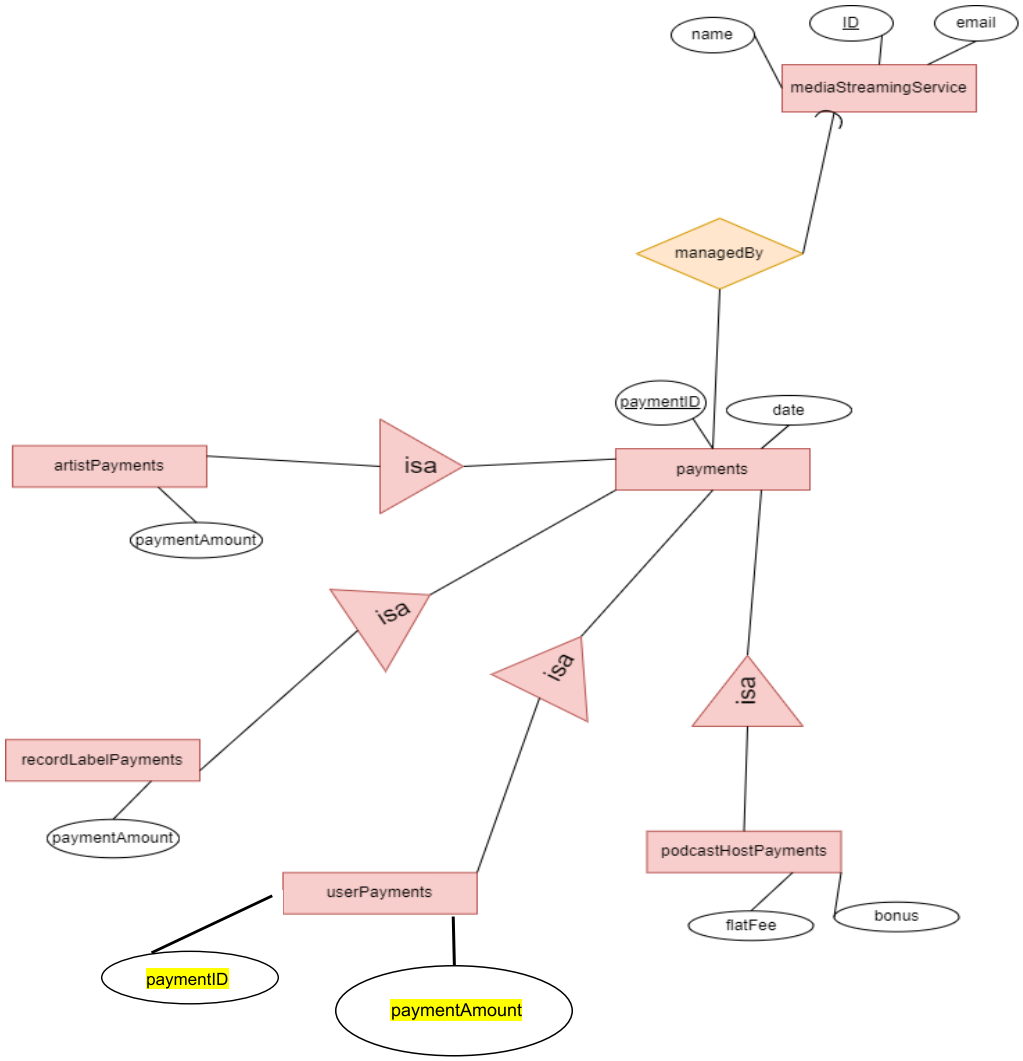
Payment Team: are responsible for managing payments for the artists, record labels and podcast hosts and also the payments received by the users.

Added Administrator as an intended user.
Administrators: They are responsible for entering, updating and deleting information about songs, artists, podcasts, podcast hosts and episodes. They will also use the database system to maintain and analyze the information about songs, artists, record labels, podcasts, podcast hosts, episodes, and payments, and generate reports on the performance and revenue of the streaming service.

Q7. Local E/R diagrams:

Answer: Added attributes paymentID and paymentAmount to userPayments.

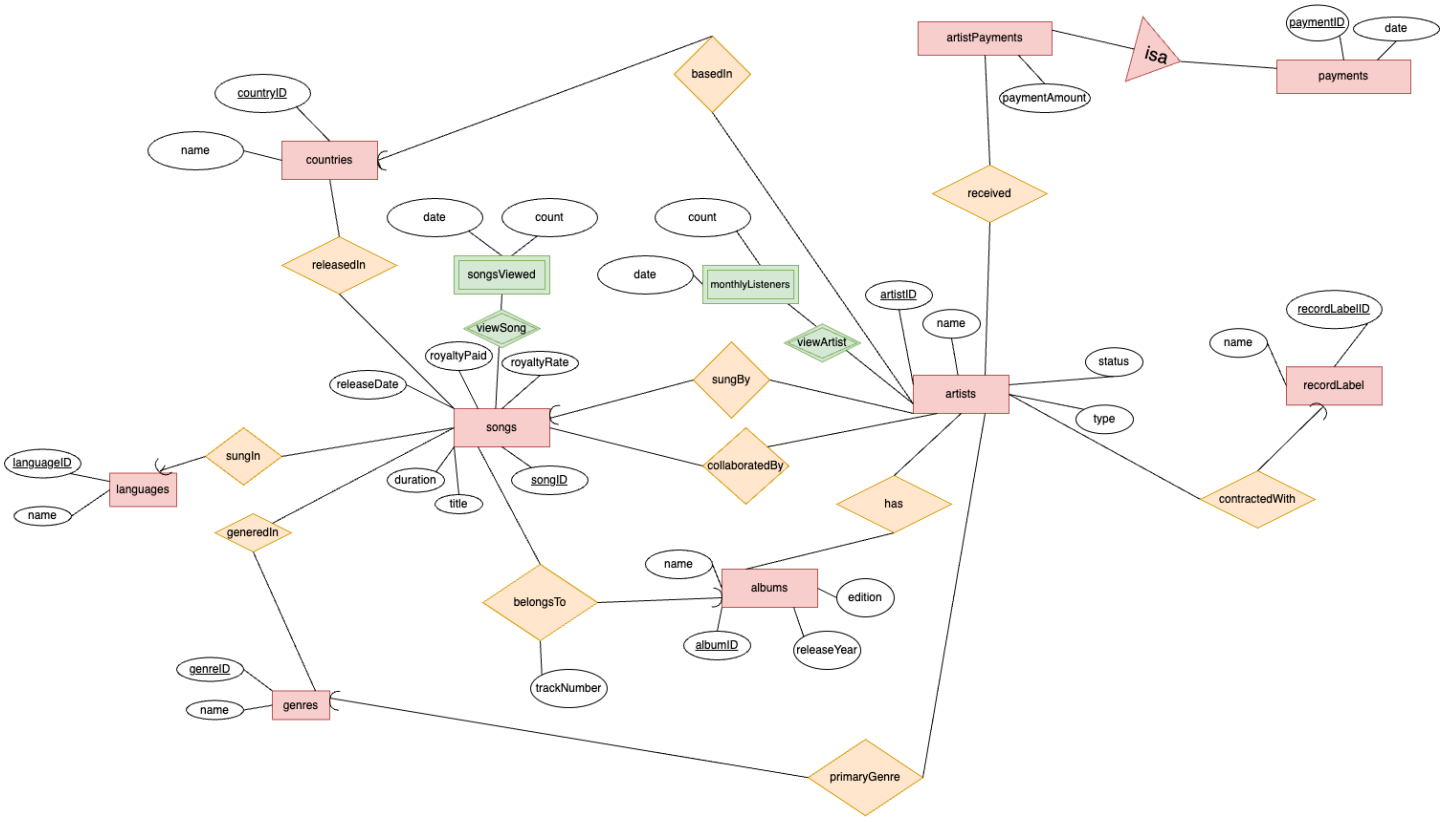
Payments Team View:



Q9. _Local Relational Schema: Pages 13, 14 and 15

Artist/record label view: supporting relationships “songsViewed” and “monthly Listeners” are redundant and should not be converted to relations. missing “viewersCount”

Answer:



We removed viewersCount weak entity from our previous diagram, and made songsViewed and monthlyListeners into weak entities of songs and artists respectively joined through the weak relations viewSong and viewArtist respectively (Earlier they were weak relations for the weak entity songsViewed). The changed relations and entities are highlighted in green.

Artist View:

songs(songID, title, duration, releaseDate, royaltyPaid, royaltyRate)
artists(artistID, name, status, type)
albums(albumID, name, edition, releaseYear)
payments(paymentID, date)
artistPayments(paymentID, paymentAmount)

countries(countryID, name)
recordLabel(recordLabelID, name)
languages(languageID, name)
genres(genreID, name)
songsViewed(songID, date, count)
monthlyListeners(artistID, date, count)
basedIn(artistID, countryID)
received(artistID, paymentID)
contractedWith(artistID, recordLabelID)
sungBy(artistID, songID)
collaboratedBy(artistID, songID)
belongsTo(albumID, songID, trackNumber)
primaryGenre(artistID, genreID)
sungIn(songID, languageID)
generedIn(songID, genreID)
releasedIn(songID, countryID)

Corrections Report 2:

Q2. Design for Global Schema: Pages 10, 11, 12, 13 and 14

Added discussion of all referential integrities

The entity sets in our diagram were made into relations, with the attributes the same for song, artists and album.The entity sets in our diagram were made into relations, with the attributes the same for podcast, podcastHost and podcastEpisode.The entity sets in our diagram were made into relations, with the attributes the same for recordLabel.The entity sets in our diagram were made into relation based on the types of payments being made.The Entity sets that are subsets of Payments were made into relations based on the E/R viewpoint to avoid redundancy and divide payment among payees.

The weak entity set "viewersCount" was made in relation with all its own attributes, plus the artists attribute through monthlyListeners relationship as well as from songs attribute through the songsViewed relationship. It is a weak entity set because it is a many-one relationship that we need to know the artistID in combination with the songID to get the count of the particular song and to know the number of monthly listeners of an artist.

Relationships basedIn, received, contractedWith, sungBy, collaboratedBy, belongsTo, primaryGenre, sungIn, generedIn, releasedIn, originCountry, podcastGeneredIn, sponsoredBy, ownedBy, podcastPayments, partOf, guestsFeatured, paymentsReceived and managedBy from the E/R diagrams have each been turned into relations in our schema. Their attributes in the schema are the keys of the entities they represent. viewersCount also has the attribute artistID of artists and songID of songs which they get from songsViewed and monthlyListeners relationships because it is a weak entity set.

1) **songs(<u>songID</u>, title, duration, releaseDate, royaltyPaid, royaltyRate)**
   songID is the primary key
   title, duration, releaseDate, royaltyPaid, royaltyRate are not allowed to be null

2) **artists(<u>artistID</u>, name, status, type)**
   artistID is the primary key
   name, status, type are not allowed to be null

3) **albums(<u>albumID</u>, name, edition, releaseYear)**
   albumID is the primary key
   name, edition, releaseYear are not allowed to be null

4) **podcasts(<u>podcastID</u>, podcastName, language, episodeCount, totalSubscribers, rating)**
   podcastID is primary key
   podcastName, language, episodeCount, totalSubscribers rating are not allowed to be null

5) **podcastHosts(<u>podcastHostID</u>, firstName, lastName, phone, email, city)**
   podcastHostID is primary key. Phone and email are the candidate keys.
   podcastHostID, firstName, lastName, phone, email, city are not allowed to be null

6) **podcastEpisodes(<u>podcastEpisodeID</u>, episodeTitle, duration, releaseDate, listeningCount, advertisementCount)**
   podcastEpisodeID is primary key
   episodeTitle, duration, releaseDate, listeningCount, advertisementCount are not allowed to be null

7) **songsViewed(<u>songID</u>, date, count)**
   songID and date together form the primary key
   count is not allowed to be null
   songID references songs(songID)

8) **monthlyListeners(<u>artistID</u>, date, count)**
   artistID, date together form the primary key
   count are not allowed to be null
   artistID references artists(artistID)

9) **belongsTo(<u>albumID</u>, <u>songID</u>, trackNumber)**
   albumID, songID together form the primary key
   trackNumber is not allowed to be null
   albumID references album(albumID) and songID references songs(songID)

10) **recordLabel(<u>recordLabelID</u>, name)**
    recordLabelID is primary key
    name is not allowed to be null

11) **countries(<u>countryID</u>, name)**
    countryID is primary key, name is the candidate key.
    name is not allowed to be null

12) **languages(<u>languageID</u>, name)**

languageID is primary key, name is the candidate key.
name is not allowed to be null

13) **genres(genreID, name)**
genreID is primary key, name is the candidate key.
name is not allowed to be null

14) **sponsors(sponsorID, sponsorName)**
sponsorID is primary key
sponsorName is not allowed to be null

15) **guests(guestID, name)**
guestID is primary key
name is not allowed to be null

16) **users(userID, phone, email, registrationDate, monthlySubscriptionFee, statusOfSubscription, firstName, lastName)**
userID is the primary key
phone, email, registrationDate, monthlySubscriptionFee, statusOfSubscription,
firstName, last Name are not allowed to be null

17) **payments(paymentID, date)**
paymentID is primary key
date is not allowed to be null

18) **podcastHostPayments(paymentID, flatFee, bonus)**
paymentID is primary key
flatFee, bonus is not allowed to be null
paymentID references payments(ID)

19) **recordLabelPayments(paymentID, paymentAmount)**
paymentID is primary key
paymentAmount is not allowed to be null
paymentID is referenced from payments(paymentID)

20) **artistPayments(paymentID, paymentAmount)**
paymentID is primary key
paymentAmount is not allowed to be null
paymentID is referenced from payments(paymentID)

21) mediaStreamingService(ID, name, email)

ID is the primary key and email is the candidate key

name is not allowed to be null

22) basedIn(artistID, countryID)

artistID and countryID together make the primary key

artistID is referenced from artists(artistID) and countryID is referenced from countries(countryID)

23) received(paymentID,artistID)

artistID and paymentID together make the primary key

paymentID is referenced from payments(paymentID) and artistID is referenced from artists(artistID)

24) has(artistID, albumID)

artistID and albumID together make the primary key

artistID is referenced from artists(artistID) and albumID is referenced from albums(albumID)

25) contractedWith(artistID, recordLabelID)

artistID and recordLabelID together make the primary key

artistID is referenced from artists(artistID) and recordLabelID is referenced from recordLabel(recordLabelID)

26) sungBy(artistID, songID)

songID is the primary key

artistID is not allowed to be null

artistID is referenced from artists(artistID) and songID is referenced from songs(songID)

27) collaboratedBy(artistID, songID)

artistID and songID together make the primary key

artistID is referenced from artists(artistID) and songID is referenced from songs(songID)

28) primaryGenre(artistID, genreID)

artistID is the primary key

genreID is not allowed to be null

artistID is referenced from artists(artistID) and genreID is referenced from genres(genreID)

29) **userPayments(paymentID,paymentAmount)**
paymentID is the primary key
paymentAmount is not allowed to be null
paymentID is referenced from payments(paymentID)

30) **paymentsReceived(recordLabelID, paymentID)**
paymentID and recordLabelID together make the primary key
paymentID is referenced from payments(paymentID) and recordLabelID is referenced from recordLabel(recordLabelID)

31) **sungIn(songID, languageID)**
languageID is not allowed to be null
songID is the primary key
songID is referenced from songs(songID) and languageID is referenced from languages(languageID)

32) **subscribeArtist(userID, artistID)**
userID and artistID together make the primary key
userID is referenced from users(userID) and artistID is referenced from artists(artistID)

33) **paymentsMade(userID, paymentID)**
userID and paymentID together make the primary key
userID is referenced from users(userID) and paymentID is referenced from payments(paymentID)

34) **podcastPayments(paymentID, podcastHostID)**
podcastHostID and paymentID together make the primary key
paymentID is referenced from payments(paymentID) and podcastHostID is referenced from podcastHosts(podcastHostID)

35) **generedIn(songID, genreID)**
songID and genreID together make the primary key
songID is referenced from songs(songID) and genreID is referenced from genres(genreID)

36) **releasedIn(songID, countryID)**

songID and countryID together make the primary key
songID is referenced from songs(songID) and countryID is referenced from countries(countryID)

37) originCountry(<u>podcastID</u>, countryID)
podcastID is the primary key
countryID is not allowed to be null
podcastID is referenced from podcasts(podcastID) and countryID is referenced from countries(countryID)

38) podcastGeneredIn(<u>podcastID</u>, <u>genreID</u>)
podcastID and genreID together make the primary key
podcastID is referenced from podcasts(podcastID) and genreID is referenced from genres(genreID)

39) sponsoredBy(<u>podcastID</u>, <u>sponsorID</u>)
podcastID and sponsorID together make the primary key
podcastID is referenced from podcasts(podcastID) and sponsorID is referenced by sponsors(sponsorID)

40) ownedBy(<u>podcastID</u>, <u>podcastHostID</u>)
podcastID and podcastHostID together make the primary key
podcastID is referenced from podcasts(podcastID) and podcastHostID is referenced from podcastHosts(podcastHostID)

41) subscribePodcast(<u>userID</u>, <u>podcastID</u>)
podcastID and userID together make the primary key
userID is referenced from users(userID) and podcastID is referenced from podcasts(podcastID)

42) partOf(<u>podcastID</u>, <u>podcastEpisodeID</u>)
podcastID and podEpisodeID together make the primary key
podcastID is referenced from podcasts(podcastID) and podcastEpisodeID is referenced from podcastEpisodes(podcastEpisodeID)

43) guestsFeatured(<u>guestID</u>, <u>podcastEpisodeID</u>)
guestID and podcastEpisodeID together make the primary key
guestID is referenced from guests(guestID) and podcastEpisodeID is referenced from podcastEpisodes(podcastEpisodeID)

44) **managedBy(<u>paymentID</u>, <u>ID</u>)**

ID and paymentID together make the primary key

paymentID is referenced from payments(paymentID) and ID is referenced from mediaStreamingService(ID)

Q2. (60 points) Write all the required applications and test them appropriately. (A lot of points will be taken off if your code does not close DBMS connections.) Submit all source code to the submit board. Your application-code language must be different from SQL and must include functionalities for sending SQL commands to a database-management system and for receiving responses from the DBMS. (Java with its JDBC functionalities is one example of permitted application-code language.) Your application programs in such a permitted language must use SQL for all database interactions.

Answer:

We have coded the application in Java and submitted the source code to the board. All the DB connections have been closed.

Q3. (120 points) In at least two of the applications (see item 2 above), use transactions to ensure that the applications work correctly even if they encounter unexpected events. (Example: a credit-card authorization fails because a staff member has entered an invalid credit-card number.) Document the program logic of the transactions in the applications. Submit, as parts of your project report 3 paper, (1) the parts of the code that contain the transactions (points will be taken off by the grader if these parts of the code are not easy to locate), and (2) your documentation. Make sure that your transactions use the COMMIT and ROLLBACK statements.

Answer:

Transactions have been used in the following applications:

1. AlbumInformationProcessing:
   Source File: src/wolfMedia/AlbumInformationProcessing.java
   Method: createAlbum(), updateAlbum(), deleteAlbum(), readAlbum()

   Algorithm Used:
   The createAlbum method starts a new transaction by calling conn.setAutoCommit(false) and creates an Album object based on user input. It then calls the createAlbum method of the Album class to insert the album data into the database. If this operation is successful,

the user is prompted to add artists and songs to the album. If the user chooses to do so, the createHasArtists and createBelongsTo methods are called to insert the corresponding data into the database. If any of these operations fail, the transaction is rolled back by calling conn.rollback(), which cancels all previous database changes.

The updateAlbum method reads an Album object from the database based on user input and starts a new transaction by calling conn.setAutoCommit(false). The user is then prompted to enter new album data, and the updateAlbum method of the Album class is called to update the database. If this operation is successful, the transaction is committed by calling conn.commit(), which makes all changes permanent. If any exception is thrown during the execution of this method, the transaction is rolled back by calling conn.rollback(), which cancels all previous database changes.

The deleteAlbum method deletes an album from the database based on user input and starts a new transaction by calling conn.setAutoCommit(false). If the album does not exist in the database, the transaction is rolled back. Otherwise, the deleteAlbum method of the Album class is called to delete the album data from the database. If this operation is successful, the transaction is committed by calling conn.commit(), which makes all changes permanent. If any exception is thrown during the execution of this method, the transaction is rolled back by calling conn.rollback(), which cancels all previous database changes.

The readAlbum method reads an Album object from the database based on user input and starts a new transaction by calling conn.setAutoCommit(false). If the album does not exist in the database, the transaction is rolled back. Otherwise, the method prints the album information and lists the artists and songs associated with it by calling the getArtists and getSongsByAlbumID methods of the Album class. If any exception is thrown during the execution of this method, the transaction is rolled back by calling conn.rollback(), which cancels all previous database changes.
In summary, the algorithm used for transactions in this code is to group a set of related database operations into a single transaction, and either commit all changes if they are successful, or roll back the transaction if any error occurs during the transaction.

Program Source Code:

```
package wolfMedia;

import java.sql.*;
import java.util.*;

/**
 * The type Album information processing.
 */
public class AlbumInformationProcessing {
    /**
     * The constant input.
     */
    public static Scanner input = new Scanner(System.in);

    /**
     * Process album.
     *
     * @throws SQLException the sql exception
     */
    public static void processAlbum() throws SQLException {
        System.out.println(x: "Enter/update/delete basic information:");
        System.out.println(x: "1. Create album");
        System.out.println(x: "2. Update album");
        System.out.println(x: "3. Delete album");
        System.out.println(x: "4. Read album information");
        System.out.print(s: "Choice: ");

        int subChoice3 = input.nextInt();

        input.nextLine(); // consume newline character

        switch (subChoice3) {
            case 1:
                createAlbum();
                break;
            case 2:
                updateAlbum();
                break;
            case 3:
                deleteAlbum();
                break;
```

```java
            case 4:
                readAlbum();
                break;
            default:
                System.out.println(x: "Invalid choice. Please enter a valid option.");
                break;
        }
    }

    /**
     * Create Album
     */
    private static void createAlbum() throws SQLException {
        System.out.println(x: "Enter album information:");
        System.out.print(s: "Album ID: ");
        String albumID = input.nextLine();
        System.out.print(s: "Name: ");
        String name = input.nextLine();
        System.out.print(s: "Edition: ");
        String edition = input.nextLine();
        System.out.print(s: "Release year: ");
        int releaseYear = Integer.parseInt(input.nextLine());

        Connection conn = Connections.open();
        conn.setAutoCommit(autoCommit: false); // start transaction

        Album a = new Album(albumID, name, edition, releaseYear);
        int isAlbumCreated = Album.createAlbum(a, conn);
        int isHasArtistCreated = 0;
        int isBelongsToCreated = 0;

        if (isAlbumCreated == 0) {
            System.out.println(x: "Album not created");
        } else {
            try {
                System.out.println(x: "Add artists to album? Enter yes/no");
                String response = input.nextLine();
                if (response.equals(anObject: "yes")) {
                    isHasArtistCreated = createHasArtists(albumID, conn);
                }

                System.out.println(x: "Add song IDs to album? Enter yes/no");
                response = input.nextLine();
                if (response.equals(anObject: "yes")) {
                    isBelongsToCreated = createBelongsTo(albumID, conn);
                }
```

```
            }

            if (isHasArtistCreated == 0 || isBelongsToCreated == 0) {
                conn.rollback(); // rollback if any failures
                System.out.println(x: "Transaction rolled back.");
            } else {
                conn.commit(); // commit if everything is successful
                System.out.println(x: "Transaction committed.");
            }
        } catch (Exception e) {
            conn.rollback(); // rollback on exception
            System.out.println("Transaction rolled back due to exception: " + e.getMessage())
        }
    }

    Connections.close(conn);
}

/**
 * Create belongs to int.
 *
 * @param albumID the album id
 * @param conn    the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int createBelongsTo(String albumID, Connection conn) throws SQLException {
    int isCreated = 0;
    while (true) {
        System.out.println(x: "Add a song to album? Enter yes/no");
        String response = input.nextLine();
        if (response.equals(anObject: "no")) {
            break;
        }
        System.out.println(x: "Song ID: ");
        String songID = input.nextLine();
        System.out.println(x: "Track Number: ");
        int trackNumber = Integer.parseInt(input.nextLine());
        if (!songID.isEmpty()) {
            BelongsTo bT = new BelongsTo(albumID, songID, trackNumber);
            isCreated = BelongsTo.createBelongsTo(bT, conn);
        }

    }
    return isCreated;
```

```java
}

/**
 * Create has artists int.
 *
 * @param albumID the album id
 * @param conn    the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int createHasArtists(String albumID, Connection conn) throws SQLException {
    System.out.println(x: "Enter album artist IDs by space: ");
    String[] albumArtistIDs = input.nextLine().split(regex: " ");
    int isCreated = 0;
    for (int i = 0; i < albumArtistIDs.length; i++) {
        Has h = new Has(albumArtistIDs[i], albumID);
        isCreated = Has.createHas(h, conn);
    }
    return isCreated;
}

/**
 * Delete has artists int.
 *
 * @param artistID the artist id
 * @param songID   the song id
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int deleteHasArtists(String artistID, String songID, Connection conn) throws SQLException {
    int isDeleted = 0;
    isDeleted = CollaboratedBy.deleteCollaboration(artistID, songID, conn);
    return isDeleted;
}
```

```java
/**
 * Update Album
 *
 * @throws SQLException the sql exception
 */
public static void updateAlbum() throws SQLException {
    System.out.println(x: "Enter album ID to update:");
    String updateID = input.nextLine();
    Connection conn = Connections.open();
    Album aToUpdate = Album.readAlbum(updateID, conn);

    if (aToUpdate == null) {
        System.out.println("Album with ID " + updateID + " does not exist");
        Connections.close(conn);
        return;
    }

    conn.setAutoCommit(autoCommit: false); // Begin transaction

    try {
        System.out.println(x: "Enter new album information:");
        System.out.print(s: "Name: ");
        aToUpdate.setName(input.nextLine());
        System.out.print(s: "Edition: ");
        aToUpdate.setEdition(input.nextLine());
        System.out.print(s: "Release year: ");
        aToUpdate.setReleaseYear(Integer.parseInt(input.nextLine()));
        int isUpdated = Album.updateAlbum(aToUpdate, conn);

        if (isUpdated == 0) {
            System.out.println(x: "Failed to update album");
            conn.rollback(); // Rollback transaction
        } else {
            System.out.println(x: "Album updated successfully");
            conn.commit(); // Commit transaction
        }

    } catch (SQLException ex) {
        System.out.println("Error updating album: " + ex.getMessage());
        conn.rollback(); // Rollback transaction
    } finally {
        Connections.close(conn);
    }
}
```

```java
public static void deleteAlbum() throws SQLException {
    System.out.println(x: "Enter album ID to delete:");
    String deleteID = input.nextLine();
    Connection conn = Connections.open();

    conn.setAutoCommit(autoCommit: false); // Begin transaction

    try {
        int s = Album.deleteAlbum(deleteID, conn);
        if (s == 0) {
            System.out.println("Album with ID " + deleteID + " does not exist");
            conn.rollback(); // Rollback transaction
        } else {
            System.out.println("Album with ID " + deleteID + " is deleted");
            conn.commit(); // Commit transaction
        }

    } catch (SQLException ex) {
        System.out.println("Error deleting album: " + ex.getMessage());
        conn.rollback(); // Rollback transaction
    } finally {
        Connections.close(conn);
    }
}


private static void readAlbum() throws SQLException {
    System.out.println(x: "Enter album ID to read:");
    String readID = input.nextLine();
    Connection conn = Connections.open();
    try {
        conn.setAutoCommit(autoCommit: false);
        Album a = Album.readAlbum(readID, conn);
        if (a == null) {
            System.out.println("Album with ID " + readID + " does not exist");
        } else {
            System.out.println("Album ID: " + a.albumID);
            System.out.println("Name: " + a.name);
            System.out.println("Edition: " + a.edition);
            System.out.println("Release Year: " + a.releaseYear);
            System.out.println(x: "Artists:");
            List<Artist> artists = Album.getArtists(a.albumID, conn);
            if (artists.isEmpty()) {
                System.out.println(x: "  (none)");
```

```
                } else {
                    for (Artist artist : artists) {
                        System.out.println("  " + artist.getArtistID() + " - " + artist.getName());
                    }
                }
                System.out.println(x: "Songs:");
                List<Song> songs = Album.getSongsByAlbumID(a.albumID, conn);
                if (songs.isEmpty()) {
                    System.out.println(x: "  (none)");
                } else {
                    for (Song song : songs) {
                        System.out.println("  " + song.getSongID() + " - " + song.getTitle());
                    }
                }
            }
            conn.commit();
        } catch (SQLException ex) {
            System.out.println("Failed to read album with ID " + readID);
            conn.rollback();
        } finally {
            Connections.close(conn);
        }
    }


}
```

2. ArtistInformationProcessing:

   Source File: src/wolfMedia/ArtistInformationProcessing.java
   Method: createArtist(), updateArtist(), deleteArtist()
   Algorithm Used:
   Each method begins by setting the connection's auto-commit mode to false, which indicates that the changes made to the database should not be committed until the transaction is completed. Then, a try-catch block is used to handle any exceptions that may occur during the transaction.

   The createArtist() method creates a new Artist record in the database, along with its related information such as songs, collaborations, albums, and contracts. The method checks whether each related record has been successfully created or not, and if any related record fails to be created, the entire transaction is rolled back. If all related records are successfully created, the changes are committed.

   The updateArtist() method updates an existing Artist record in the database. The method first reads the Artist record from the database, then prompts the user to enter the new information for the Artist. If the update is successful, the changes are committed, otherwise, the transaction is rolled back.
   The deleteArtist() method deletes an existing Artist record from the database, along with its related records. If the delete operation is successful, the changes are committed, otherwise, the transaction is rolled back.

   In summary, the algorithm used for transactions in this code is to group a set of related database operations into a single transaction, and either commit all changes if they are successful, or roll back the transaction if any error occurs during the transaction.

Program Source Code:

```
package wolfMedia;

import java.sql.*;
import java.util.*;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

/**
 * The type Artist information processing.
 */
public class ArtistInformationProcessing {
    /**
     * The constant input.
     */
    public static Scanner input = new Scanner(System.in);

    /**
     * Process artist.
     *
     * @throws SQLException the sql exception
     */
    public static void processArtist() throws SQLException {
        System.out.println(x: "Enter/update/delete basic information:");
        System.out.println(x: "1. Create artist");
        System.out.println(x: "2. Update artist");
        System.out.println(x: "3. Delete artist");
        System.out.println(x: "4. Read artist information");
        System.out.println(x: "5. Assign Artist To Album");
        System.out.println(x: "6. Assign Artist To RecordLabel");
        System.out.print(s: "Choice: ");

        int subChoice3 = input.nextInt();

        input.nextLine(); // consume newline character

        switch (subChoice3) {
            case 1:
                createArtist();
                break;
            case 2:
                updateArtist();
                break;
            case 3:
                deleteArtist();
                break;
```

```
            case 4:
                readArtist();
                break;
            case 5:
                assignArtistToAlbum();
                break;
            case 6:
                assignArtistToRecordLabel();
                break;
            default:
                System.out.println(x: "Invalid choice. Please enter a valid option.");
                break;
        }
    }

    /**
     * Create Artist
     */
    private static void createArtist() throws SQLException {
        System.out.println(x: "Enter artist information:");
        System.out.print(s: "Artist ID: ");
        String artistID = input.nextLine();
        System.out.print(s: "Name: ");
        String name = input.nextLine();
        System.out.print(s: "Status active/inactive: ");
        String status = input.nextLine();
        System.out.print(s: "Type solo/band: ");
        String type = input.nextLine();
        Connection conn = null;

        Artist a = new Artist(artistID, name, status, type);
        int isCreated = 0;
        int isBasedInCreated = 0;
        int isSongCreated = 0;
        int isSungByCreated = 0;
        int areCollaboratorsCreated = 0;
        int areAlbumsLinked = 0;
        int isPrimaryGenreCreated = 0;
        int isArtistMonthlyListenersCreated = 0;
        int isArtistContractsAdded = 0;
        try {
            conn = Connections.open();
            conn.setAutoCommit(autoCommit: false);

            isCreated = Artist.createArtist(a, conn);
```

```
            isCreated = Artist.createArtist(a, conn);
            if (isCreated == 0) {
                System.out.println(x: "Artist not created");
                conn.rollback();
            } else {
                isBasedInCreated = createBasedIn(artistID, conn);

                System.out.println(x: "Want to add a song? Enter yes/no");
                String response = input.nextLine();
                if (response.equals(anObject: "yes")) {
                    isSongCreated = SongInformationProcessing.createSong(from: "artist");
                } else {
                    System.out.println(x: "Insted just Want to add song IDs? Enter yes/no");
                    response = input.nextLine();
                    if (response.equals(anObject: "yes")) {
                        isSungByCreated = createSungBy(artistID, conn);
                    }
                }

                System.out.println(x: "Want to add any collaborations with other artist's songs? Enter yes/no");
                response = input.nextLine();
                if (response.equals(anObject: "yes")) {
                    areCollaboratorsCreated = createCollaborations(artistID, conn);
                }

                System.out.println(x: "Want to add artist to any existing albums? Enter yes/no");
                response = input.nextLine();
                if (response.equals(anObject: "yes")) {
                    areAlbumsLinked = createHasAlbums(artistID, conn);
                }

                isPrimaryGenreCreated = createArtistPrimaryGeneredIn(artistID, conn);

                isArtistMonthlyListenersCreated = createArtistMonthlyListeners(artistID, conn);

                isArtistContractsAdded = addArtistRecordLabelcontracts(artistID, conn);

                if (isBasedInCreated == 0 || isSongCreated == 0 || isSungByCreated == 0 ||
                    areCollaboratorsCreated == 0 || areAlbumsLinked == 0 ||
                    isPrimaryGenreCreated == 0 || isArtistMonthlyListenersCreated == 0 ||
                    isArtistContractsAdded == 0) {
                    conn.rollback();
                    System.out.println(x: "Changes rolled back.");
                } else {
                    conn.commit();
                }
            }
        } catch (SQLException e) {
            conn.rollback();
            e.printStackTrace();
        } finally {
            Connections.close(conn);
        }
    }

    /**
     * Assign Artist To Album.
     *
     * @return nothing
     * @throws SQLException the sql exception
     */
    public static void assignArtistToAlbum() throws SQLException {
        Connection conn = Connections.open();
        System.out.println(x: "EnterArtist ID\n");
        String artistID = input.nextLine();
        int isCreated = 0;
        if (!artistID.isEmpty()) {
            isCreated = createHasAlbums(artistID, conn);
        }
        if (isCreated == 0) {
            System.out.println(x: "Artist not assigned to Albums mentioned");
        } else {
            System.out.println("Artist with ID: " + artistID + "has been assigned to respective Albums");
        }
        Connections.close(conn);
    }
```

```java
/**
 * Assign Artist To RecordLabel.
 *
 * @return nothing
 * @throws SQLException the sql exception
 */
public static void assignArtistToRecordLabel() throws SQLException {
    Connection conn = Connections.open();
    System.out.println(x: "EnterArtist ID\n");
    String artistID = input.nextLine();
    int isCreated = 0;
    if (!artistID.isEmpty()) {
        isCreated = addArtistRecordLabelcontracts(artistID, conn);
    }
    if (isCreated == 0) {
        System.out.println(x: "Artist not assigned to Record Label");
    } else {
        System.out.println("Artist with ID: " + artistID + "has been assigned to respective Record Label");
    }
    Connections.close(conn);
}

/**
 * Add artist record labelcontracts int.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int addArtistRecordLabelcontracts(String artistID, Connection conn) throws SQLException {
    System.out.println(x: "Enter  Record Label ID\n");
    String recordLabelID = input.nextLine();
    int isCreated = 0;
    if (!recordLabelID.isEmpty()) {
        ContractedWith cW = new ContractedWith(artistID, recordLabelID);
        isCreated = ContractedWith.createContractedWith(cW, conn);
    }
    return isCreated;
}
```

```java
/**
 * Create artist primary genered in int.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int createArtistPrimaryGeneredIn(String artistID, Connection conn) throws SQLException {
    System.out.println(
            x: "Enter Artist Primary Genre: \n 1: Pop\n 2: Rock\n 3: Hip hop\n 4: Electronic\n 5: Classical\n 6: Country\n 7: Jazz\n 8: B
    String genreID = input.nextLine();
    int isCreated = 0;
    if (!genreID.isEmpty()) {
        PrimaryGenre pG = new PrimaryGenre(artistID, genreID);
        isCreated = PrimaryGenre.createPrimaryGenre(pG, conn);
    }
    return isCreated;
}

/**
 * Delete artist primary genered in int.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int deleteArtistPrimaryGeneredIn(String artistID, Connection conn) throws SQLException {
    int isDeleted = 0;
    isDeleted = PrimaryGenre.deletePrimaryGenre(artistID, conn);
    return isDeleted;
}

/**
 * Create has albums int.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
```

```java
public static int createHasAlbums(String artistID, Connection conn) throws SQLException {
    System.out.println(x: "Enter album IDs by space: ");
    String[] artistAlbumIDs = input.nextLine().split(regex: " ");
    int isCreated = 0;
    for (int i = 0; i < artistAlbumIDs.length; i++) {
        Has h = new Has(artistID, artistAlbumIDs[i]);
        isCreated = Has.createHas(h, conn);
    }
    return isCreated;
}

/**
 * Create collaborations int.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int createCollaborations(String artistID, Connection conn) throws SQLException {
    System.out.println(x: "Collaborated song IDs by space: ");
    String[] collaboratedSongIDs = input.nextLine().split(regex: " ");
    int isCreated = 0;
    for (int i = 0; i < collaboratedSongIDs.length; i++) {
        CollaboratedBy cB = new CollaboratedBy(artistID, collaboratedSongIDs[i]);
        isCreated = CollaboratedBy.createCollaboration(cB, conn);
    }
    return isCreated;
}

/**
 * Delete collaboration int.
 *
 * @param artistID the artist id
 * @param songID   the song id
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int deleteCollaboration(String artistID, String songID, Connection conn) throws SQLException {
    int isDeleted = 0;
    isDeleted = CollaboratedBy.deleteCollaboration(artistID, songID, conn);
    return isDeleted;
}
```



```java
public static int addPaymentReceived(String artistID, String paymentID, Connection conn) throws SQLException {
    Received r = new Received(paymentID, artistID);
    int isCreated = Received.createReceived(r, conn);
    return isCreated;
}

/**
 * Create based in int.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int createBasedIn(String artistID, Connection conn) throws SQLException {
    System.out.println(
            x: "Artist is Based In: \n 1: United States\n 2: United Kingdom\n 3: Canada\n 4: Australia\n 5: Japan\n 6: Germany\n 7: Fran
    String countryID = input.nextLine();
    int isCreated = 0;
    if (!countryID.isEmpty()) {
        BasedIn bI = new BasedIn(artistID, countryID);
        isCreated = BasedIn.createBasedIn(bI, conn);
    }
    return isCreated;
}

/**
 * Delete based in int.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int deleteBasedIn(String artistID, Connection conn) throws SQLException {
    int isDeleted = 0;
    isDeleted = BasedIn.deleteBasedIn(artistID, conn);
    return isDeleted;
}
```

```java
public static int increaseArtistMonthlyListeners(String artistID, Connection conn) throws SQLException {
    LocalDate currentDate = LocalDate.now();
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern(pattern: "yyyy-MM-01");
    String date = currentDate.format(formatter);

    MonthlyListeners mL = MonthlyListeners.readMonthlyListeners(artistID, date, conn);
    if (mL == null) {
        mL = new MonthlyListeners(artistID, date, count: 1);
        MonthlyListeners.createMonthlyListeners(mL, conn);
        return 1;
    } else {
        int count = mL.getCount();
        mL.setCount(count + 1);
        MonthlyListeners.updateMonthlyListeners(mL, conn);
        return count + 1;
    }
}

/**
 * Create artist monthly listeners int.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int createArtistMonthlyListeners(String artistID, Connection conn) throws SQLException {
    LocalDate currentDate = LocalDate.now();
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern(pattern: "yyyy-MM-01");
    String date = currentDate.format(formatter);

    MonthlyListeners mL = new MonthlyListeners(artistID, date, count: 0);
    int isCreated = MonthlyListeners.createMonthlyListeners(mL, conn);
    return isCreated;
}
```

```java
public static int createSungBy(String artistID, Connection conn) throws SQLException {
    int isCreated = 0;
    while (true) {
        System.out.println(x: "Want to add a songID? Enter yes/no: ");
        String res = input.nextLine();
        if (res == "no") {
            break;
        }
        System.out.println(x: "Enter songID: ");
        String songID = input.nextLine();
        if (!songID.isEmpty()) {
            SungBy sB = new SungBy(artistID, songID);
            isCreated = SungBy.createSungBy(sB, conn);
        }
    }
    return isCreated;
}

/**
 * Delete sung by int.
 *
 * @param artistID the artist id
 * @param songID   the song id
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int deleteSungBy(String artistID, String songID, Connection conn) throws SQLException {
    int isDeleted = 0;
    isDeleted = SungBy.deleteSungBy(artistID, songID, conn);
    return isDeleted;
}
```

```java
/**
 * Update Artist
 */
private static void updateArtist() throws SQLException {
    System.out.println(x: "Enter artist ID to update:");
    String updateID = input.nextLine();
    Connection conn = Connections.open();
    Artist aToUpdate = Artist.readArtist(updateID, conn);

    if (aToUpdate == null) {
        System.out.println("Artist with ID " + updateID + " does not exist");
        Connections.close(conn);
        return;
    }

    conn.setAutoCommit(autoCommit: false); // Begin transaction

    try {
        System.out.println(x: "Enter new artist information:");
        System.out.print(s: "Name: ");
        aToUpdate.setName(input.nextLine());
        System.out.print(s: "Status active/inactive: ");
        aToUpdate.setStatus(input.nextLine());
        System.out.print(s: "Type solo/band: ");
        aToUpdate.setType(input.nextLine());
        int isUpdated = Artist.updateArtist(aToUpdate, conn);

        if (isUpdated == 0) {
            System.out.println(x: "Failed to update artist");
            conn.rollback(); // Rollback transaction
        } else {
            System.out.println(x: "Artist updated successfully");
            conn.commit(); // Commit transaction
        }

    } catch (SQLException ex) {
        System.out.println("Error updating artist: " + ex.getMessage());
        conn.rollback(); // Rollback transaction
    } finally {
        Connections.close(conn);
    }
}
```

```java
private static void deleteArtist() throws SQLException {
    System.out.println(x: "Enter artist ID to delete:");
    String deleteID = input.nextLine();
    Connection conn = Connections.open();
    Artist artistToDelete = Artist.readArtist(deleteID, conn);
    if (artistToDelete == null) {
        System.out.println("Artist with ID " + deleteID + " does not exist");
        Connections.close(conn);
        return;
    }
    conn.setAutoCommit(autoCommit: false); // Begin transaction
    try {
        int isDeleted = Artist.deleteArtist(deleteID, conn);
        if (isDeleted == 0) {
            System.out.println(x: "Failed to delete artist");
            conn.rollback(); // Rollback transaction
        } else {
            System.out.println("Artist with ID " + deleteID + " is deleted");
            // TODO: delete artist relationship tables - done with cascase delete
            conn.commit(); // Commit transaction
        }
    } catch (SQLException ex) {
        System.out.println("Error deleting artist: " + ex.getMessage());
        conn.rollback(); // Rollback transaction
    } finally {
        Connections.close(conn);
    }
}
```

```java
public static List<ArtistPayments> getArtistPayments(String artistID, Connection conn) throws SQLException {
    List<ArtistPayments> artistPayments = Received.getArtistPayments(artistID, conn);
    if (artistPayments.isEmpty()) {
        System.out.println(x: "  (none)");
    } else {
        for (ArtistPayments artistPayment : artistPayments) {
            System.out.println("  " + artistPayment.getPaymentID() + " - " + artistPayment.getPaymentAmount());
        }
    }
    return artistPayments;
}

/**
 * Artist based in string.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return the string
 * @throws SQLException the sql exception
 */
public static String artistBasedIn(String artistID, Connection conn) throws SQLException {
    BasedIn rI = BasedIn.readBasedIn(artistID, conn);
    if (rI == null) {
        return "(none)";
    }
    Country c = Country.readCountry(rI.getCountryID(), conn);
    return c.getName();
}

/**
 * Artist monthly viewed int.
 *
 * @param artistID the artist id
 * @param date     the date
 * @param conn     the conn
 * @return int value=> operation success(1)/failure(0)
 * @throws SQLException the sql exception
 */
public static int artistMonthlyViewed(String artistID, String date, Connection conn) throws SQLException {
    MonthlyListeners mL = MonthlyListeners.readMonthlyListeners(artistID, date, conn);
    if (mL == null) {
        return 0;
    }
    return mL.getCount();
}
```

```java
/**
 * Gets artist songs.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return the artist songs
 * @throws SQLException the sql exception
 */
public static List<Song> getArtistSongs(String artistID, Connection conn) throws SQLException {
    List<Song> artistSongs = SungBy.getSongsByArtistID(artistID, conn);
    if (artistSongs.isEmpty()) {
        System.out.println(x: "  (none)");
    } else {
        for (Song artistSong : artistSongs) {
            System.out.println("  " + artistSong.getSongID() + " - " + artistSong.getTitle());
        }
    }
    return artistSongs;
}

/**
 * Gets artist collaborated songs.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return the artist collaborated songs
 * @throws SQLException the sql exception
 */
public static List<Song> getArtistCollaboratedSongs(String artistID, Connection conn) throws SQLException {
    List<Song> collabSongs = CollaboratedBy.getSongsByArtistID(artistID, conn);
    if (collabSongs.isEmpty()) {
        System.out.println(x: "  (none)");
    } else {
        for (Song artistSong : collabSongs) {
            System.out.println("  " + artistSong.getSongID() + " - " + artistSong.getTitle());
        }
    }
    return collabSongs;
}
```

```java
/**
 * Gets artist albums.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return the artist albums
 * @throws SQLException the sql exception
 */
public static List<Album> getArtistAlbums(String artistID, Connection conn) throws SQLException {
    List<Album> albums = Has.getAlbumsByArtistID(artistID, conn);
    if (albums.isEmpty()) {
        System.out.println(x: "  (none)");
    } else {
        for (Album album : albums) {
            System.out.println("  " + album.getAlbumID() + " - " + album.getName());
        }
    }
    return albums;
}

/**
 * Gets artist record label contracts.
 *
 * @param artistID the artist id
 * @param conn     the conn
 * @return the artist record label contracts
 * @throws SQLException the sql exception
 */
public static List<RecordLabel> getArtistRecordLabelContracts(String artistID, Connection conn)
        throws SQLException {
    List<RecordLabel> recordLabels = ContractedWith.getRecordLabelContractsByArtistID(artistID, conn);
    if (recordLabels.isEmpty()) {
        System.out.println(x: "  (none)");
    } else {
        for (RecordLabel recordLabel : recordLabels) {
            System.out.println("  " + recordLabel.getRecordLabelID() + " - " + recordLabel.getName());
        }
    }
    return recordLabels;
}
```

```java
private static void readArtist() throws SQLException {
    // add code to prompt for artist ID and display artist information from data
    // store
    LocalDate currentDate = LocalDate.now();
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern(pattern: "yyyy-MM-01");
    String date = currentDate.format(formatter);

    System.out.println(x: "Enter artist ID to read:");
    String readID = input.nextLine();
    Connection conn = Connections.open();
    Artist a = Artist.readArtist(readID, conn);
    if (a == null) {
        System.out.println("Artist with ID " + readID + " does not exist");
    } else {
        System.out.println("Artist ID: " + a.artistID);
        System.out.println("Name: " + a.name);
        System.out.println("Status: " + a.status);
        System.out.println("Type: " + a.type);

        System.out.println(x: "Payments Received: ");

        getArtistPayments(readID, conn);

        System.out.println("Artist is based in: " + artistBasedIn(readID, conn));

        System.out.println("Artist Monthy Views: " + artistMonthlyViewed(readID, date, conn));

        System.out.println(x: "Artist Songs: ");

        getArtistSongs(readID, conn);

        System.out.println(x: "Artist collaborations: ");

        getArtistCollaboratedSongs(readID, conn);

        System.out.println(x: "Artist Albums: ");

        getArtistAlbums(readID, conn);

        System.out.println(x: "Artist Contracted Record Labels: ");

        getArtistRecordLabelContracts(readID, conn);

    }
```

```
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern(pattern: "yyyy-MM-01");
        String date = currentDate.format(formatter);

        System.out.println(x: "Enter artist ID to read:");
        String readID = input.nextLine();
        Connection conn = Connections.open();
        Artist a = Artist.readArtist(readID, conn);
        if (a == null) {
            System.out.println("Artist with ID " + readID + " does not exist");
        } else {
            System.out.println("Artist ID: " + a.artistID);
            System.out.println("Name: " + a.name);
            System.out.println("Status: " + a.status);
            System.out.println("Type: " + a.type);

            System.out.println(x: "Payments Received: ");

            getArtistPayments(readID, conn);

            System.out.println("Artist is based in: " + artistBasedIn(readID, conn));

            System.out.println("Artist Monthy Views: " + artistMonthlyViewed(readID, date, conn));

            System.out.println(x: "Artist Songs: ");

            getArtistSongs(readID, conn);

            System.out.println(x: "Artist collaborations: ");

            getArtistCollaboratedSongs(readID, conn);

            System.out.println(x: "Artist Albums: ");

            getArtistAlbums(readID, conn);

            System.out.println(x: "Artist Contracted Record Labels: ");

            getArtistRecordLabelContracts(readID, conn);

        }
        Connections.close(conn);
    }
}
```

Q4. (60 points) Document your programs. The documentation should be part of the code (submitted via the submit board), but will be graded in addition. In a separate submission that is a part of your project 3 report paper, highlight *high-level* design decisions, which are any choices/decisions that you had to make when designing your database and applications. As part of the documentation submitted as part of your project 3 report paper, explain who in your team played what functional role (e.g., software engineer, database designer/administrator, etc, see above under Organization) in each part (1 through 3) of the project. Submit the documentation.

Answer:
Documentation: src/wolfMedia/package-tree.html
This file is the documentation file for all the classes in our application.

Design Decisions:

Our main menu shows five options as follows:

Please choose an option:

1. Enter/update/delete basic Information Processing
2. Maintain metadata and records
3. Maintain payments
4. Generate reports
5. Exit

    If Choice=1:
    Enter/update/delete basic information:
    1. Song
    2. Artist
    3. Podcast host
    4. Podcast episode
    5. Album
    6. Record label
    7. Podcast

        If Choice=1: (Song)
        1. Create song
        2. Update song
        3. Delete song
        4. Read song information
        5. Assign Song To Album

        If Choice=2: (Artist)

1. Create artist
2. Update artist
3. Delete artist
4. Read artist information
5. Assign Artist To Album
6. Assign Artist To RecordLabel

If Choice=3: (Podcast Host)
1. Create PodcastHost
2. Update PodcastHost
3. Delete PodcastHost
4. Read PodcastHost information
5. Assign Podcast Host To Podcast

If Choice=4: (Podcast Episode)
1. Create Podcast Episode
2. Update Podcast Episode
3. Delete Podcast Episode
4. Read Podcast Episode information
5. Assign Podcast Episode To Podcast

If Choice=5: (Album)
1. Create album
2. Update album
3. Delete album
4. Read album information

If Choice=6: (Record Label)
1. Create Record Label
2. Update Record Label
3. Delete Record Label
4. Read Record Label information

If Choice=7: (Podcast)
1. Create podcast
2. Update podcast
3. Delete podcast
4. Read podcast information

If Choice=2:
Update/Maintain metadata and records of:

1. Update Play count for songs
2. Update Count of monthly listeners for artists
3. Update Total count of subscribers and ratings for podcasts
4. Update Listening count for podcast episodes
5. Find songs and podcast episodes given artist, album, and/or podcast

If Choice=3:
Maintain payments:

a) Make royalty payments for a given song
b) Make payment to podcast hosts
c) Receive payment from subscribers

If Choice=4:
Generate reports:

a) Monthly play count per song/album/artist
b) Total payments made out to host/artist/record labels per a given time period
c) Total revenue of the streaming service per month, per year
d) Report all songs/podcast episodes given an artist, album, and/or podcast

The main menu comprises all the possible operations on the database. As the name suggests, information processing further has a submenu that enlists the basic operations for adding, updating and deleting for songs, albums, podcasts, podcast episodes, record labels, artists, podcast hosts; a submenu for maintaining metadata and records, which involves updating play count for songs, monthly listener count for artists, total count of subscribers and rating for podcasts, listening count for podcast episodes and finding songs and podcast episodes given artist, album and/or podcast; a submenu for maintaining payments such as making payments for a song, to podcast hosts and to receive payments from subscribers; and a submenu for generating reports.

We have made a separate file ‘Connections.java’ which uses Driver Manager, and takes the url, username and password for establishing the jdbc connection.

Functional Roles:

Part 1:
Software Engineer: Aastha Singh (Prime), Nagaraj Madamshetti (Backup)
Database Designer/Administrator: Kalyan Karnati (Prime), Kritika Javali (Backup)
Application Programmer: Kritika Javali (Prime), Aastha Singh(Backup)
Test Plan Engineer: Nagaraj Madamshetti (Prime), Kalyan Karnati (Backup)

Part 2:
Software Engineer: Nagaraj Madamshetti (Prime), Kalyan Karnati (Backup)
Database Designer/Administrator: Kritika Javali (Prime), Nagaraj Madamshetti(Backup)
Application Programmer: Aastha Singh (Prime), Kritika Javali (Backup)
Test Plan Engineer:Kalyan Karnati (Prime), Aastha Singh (Backup)

Part 3:
Software Engineer: Kalyan Karnati(Prime), Kritika Javali (Backup)
Database Designer/Administrator: Kritika Javali (Prime), Nagaraj Madamshetti (Backup)
Application Programmer: Nagaraj Madamshetti (Prime), Aastha Singh (Backup)
Test Plan Engineer: Aastha Singh (Prime), Kalyan Karnati (Backup)

Q5. (320 points) Demo. Graded on the functionality & robustness of your programs. (You are *not* required to implement a graphical user interface or a web-based interface, and will *not* get extra credit for doing so.)

Answer:
Demo is scheduled on 14th April, 2023.

Q6. Peer evaluations (via submit board). For each project report, if you (individually) wish to get a grade for the report, you must submit your own peer evaluation of each member of your team. Please submit a single plain-text file *per student* via the submit board, with your own evaluation of all members of your team including yourself, using the values (such as "excellent", "ordinary", or "superficial", to grade your team members' *participation in the teamwork*) listed in the peer-evaluation form. No signature is needed in your submit-board submission.

Answer:
Peer evaluations have been submitted by each team member.