CSC 540(001) Database management concepts and  Systems

# WOLFMEDIA

## A MEDIA STREAMING SERVICE

## Team - G

- Arun Srinivasan Parthasarathy, apartha4
- Kiron Jayesh, kjayesh
- Adittya Soukarjya Saha, asaha4
- Arun Kumar Ramesh, arames25

**Project Report - 3**

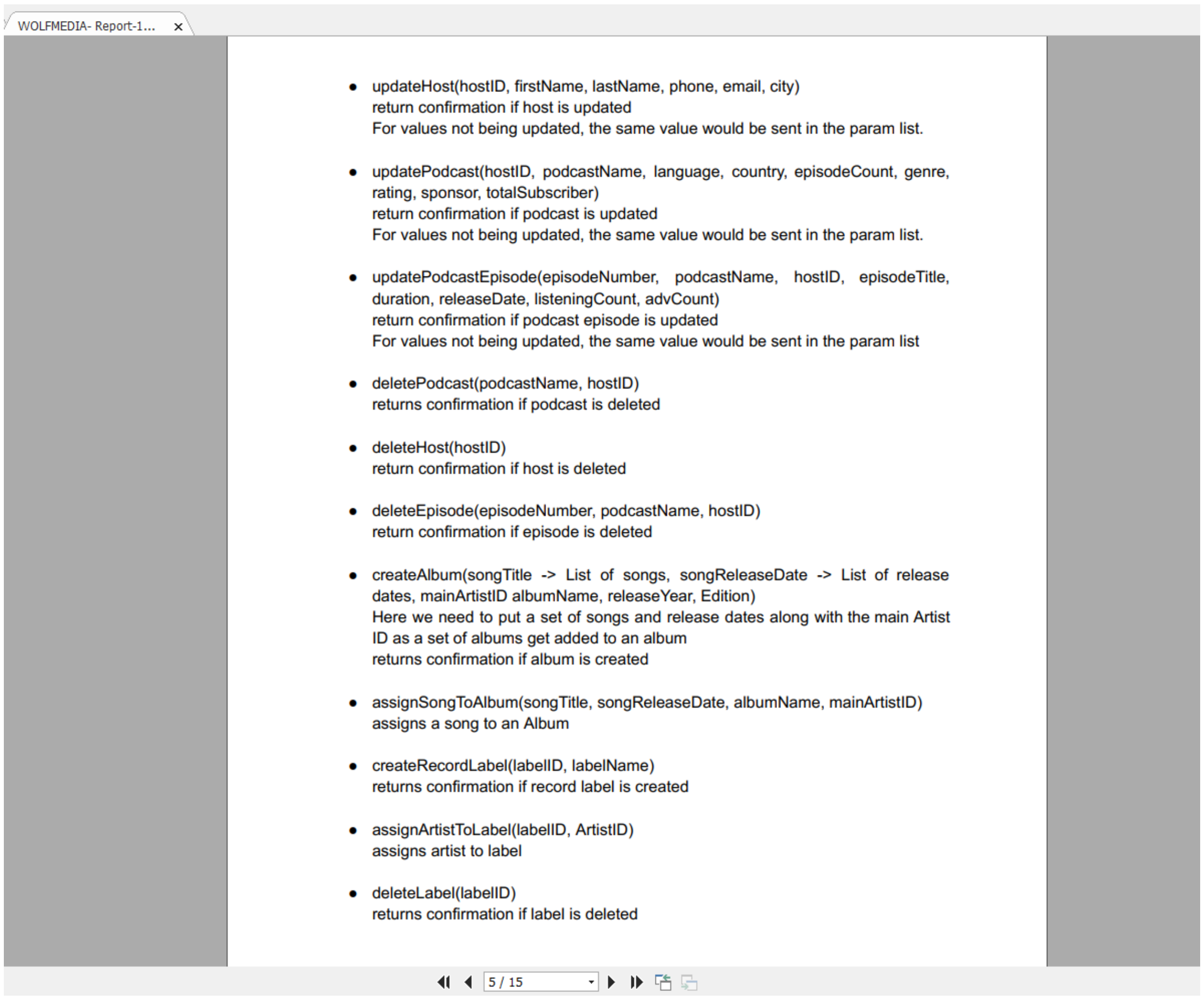
## Assumptions

1. There can only be one main artist for one song. The other artists are added in the CollaboratedBy table.
2. An artist can belong to only one record label.
3. All songs in an album have the same main artist.
4. There cannot be two songs with the same title, released by the same artist on the same date. In other words, song title, artist ID, and release date of the song are primary keys of Songs.
5. Royalty payment for songs will be made on the first of every month.
6. No two albums can have the same name if they have the same artist.
7. Album is considered to be a weak entity to songs since an album cannot exist without at least one song. Therefore, when you create an album, you have to add atleast one song to it.
8. Play count in the Songs table contains the number of times the song was played during the current month.
9. The podcast payment is made in lump sum to the podcast hosts. The responsibility of splitting the payment lies among the podcast hosts.
10. Podcasts cannot have the same name if they are hosted by the same set of hosts.
11. If an episode is deleted from a podcast, the episode number of the episodes that follow the deleted episode are not decreased by 1 since we want to retain the identity of the other episodes as such. Although, Episode Count in Podcast table updates after a deletion/addition of an episode.
12. The podcast payments are designed in such a way that payment to an episode can be done by the user right after the episode is released.
13. The fee per advertisement for podcast episodes is not stored in any of the tables and those values have to be entered by the user while making the payment for an episode.
14. The flat fee is not fixed for a podcast and hence, it can vary for different episodes in the same podcast. So, we have not included a ‘Flat Fee’ attribute in the Podcast table and the user has to enter the flat fee while they make a payment for an episode. The flat fee corresponding to that payment is stored in the Podcast Payment table.
15. The revenue for the media streaming service is solely coming only from users. There are no additional sources of revenue like advertisements or sponsorships to the company directly. For example, if there are 2 users and their monthly subscription fee is $10 each, then the company’s revenue for the current month is $20 ($10 + $10).
16. Users' payments are done for the entire month. For instance, if a user registers/subscribes at the end of May, the user will have to pay the subscription for the entire month of May.
17. Since all the users pay their monthly subscription fee at the end of the month, we record one whole payment in the UserPayment table which is a summation of all the individual user payments for that month. But, those individual payments are recorded in the PayTo table.

## **Modifications according to the feedback given in Report-1:**

**Point - 5** of the feedback given in Report - 1

Pg-6 of the original document and Pg-5 of the revision edition. Since the generation of APIs for User and Guests was not asked for, we removed those APIs from the report which were initially present after the deleteLabel() in the original report. Now after the deleteLabel(), there are no more APIs for User and Guests. We are attaching the portion of the updated version below to show that:



- updateHost(hostID, firstName, lastName, phone, email, city)
  return confirmation if host is updated
  For values not being updated, the same value would be sent in the param list.

- updatePodcast(hostID, podcastName, language, country, episodeCount, genre, rating, sponsor, totalSubscriber)
  return confirmation if podcast is updated
  For values not being updated, the same value would be sent in the param list.

- updatePodcastEpisode(episodeNumber, podcastName, hostID, episodeTitle, duration, releaseDate, listeningCount, advCount)
  return confirmation if podcast episode is updated
  For values not being updated, the same value would be sent in the param list

- deletePodcast(podcastName, hostID)
  returns confirmation if podcast is deleted

- deleteHost(hostID)
  return confirmation if host is deleted

- deleteEpisode(episodeNumber, podcastName, hostID)
  return confirmation if episode is deleted

- createAlbum(songTitle -> List of songs, songReleaseDate -> List of release dates, mainArtistID albumName, releaseYear, Edition)
  Here we need to put a set of songs and release dates along with the main Artist ID as a set of albums get added to an album
  returns confirmation if album is created

- assignSongToAlbum(songTitle, songReleaseDate, albumName, mainArtistID)
  assigns a song to an Album

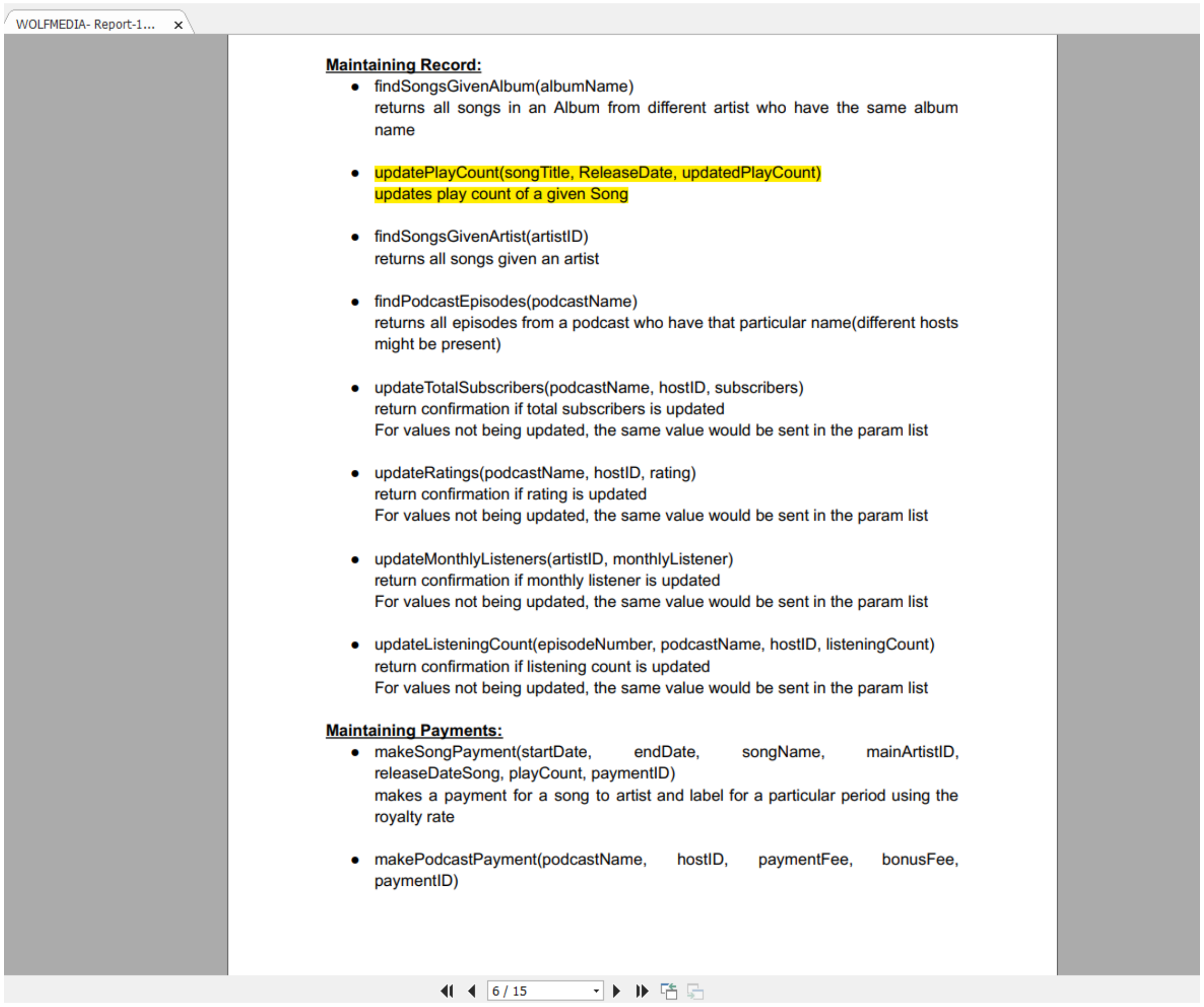
- createRecordLabel(labelID, labelName)
  returns confirmation if record label is created

- assignArtistToLabel(labelID, ArtistID)
  assigns artist to label

- deleteLabel(labelID)
  returns confirmation if label is deleted

Page: 5

The API for the play count for Song has been added in Pg-6 of the revised edition and has been highlighted in yellow as well. The portion of the revised version has been attached below, so that it's easier to find the change.



**Maintaining Record:**

- findSongsGivenAlbum(albumName)
  returns all songs in an Album from different artist who have the same album name

- updatePlayCount(songTitle, ReleaseDate, updatedPlayCount)
  updates play count of a given Song

- findSongsGivenArtist(artistID)
  returns all songs given an artist

- findPodcastEpisodes(podcastName)
  returns all episodes from a podcast who have that particular name(different hosts might be present)

- updateTotalSubscribers(podcastName, hostID, subscribers)
  return confirmation if total subscribers is updated
  For values not being updated, the same value would be sent in the param list

- updateRatings(podcastName, hostID, rating)
  return confirmation if rating is updated
  For values not being updated, the same value would be sent in the param list

- updateMonthlyListeners(artistID, monthlyListener)
  return confirmation if monthly listener is updated
  For values not being updated, the same value would be sent in the param list

- updateListeningCount(episodeNumber, podcastName, hostID, listeningCount)
  return confirmation if listening count is updated
  For values not being updated, the same value would be sent in the param list

**Maintaining Payments:**

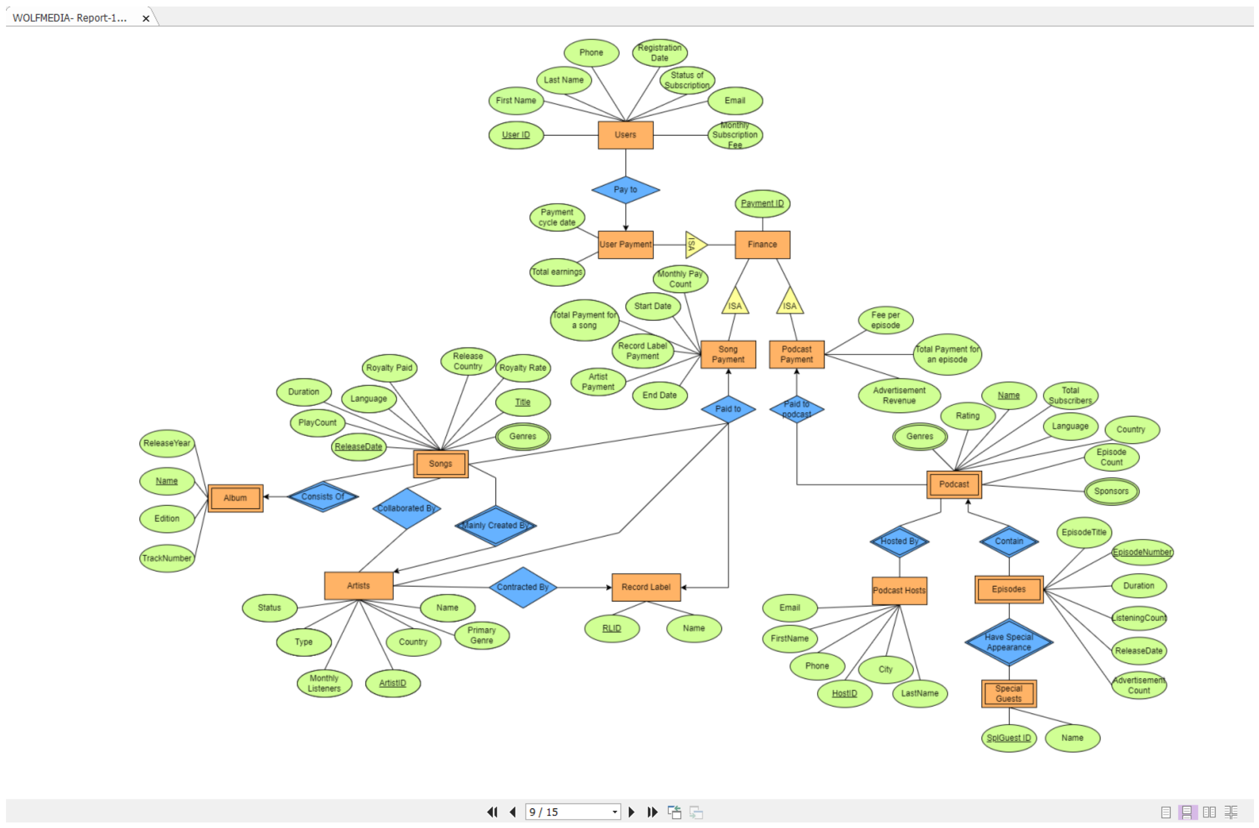
- makeSongPayment(startDate, endDate, songName, mainArtistID, releaseDateSong, playCount, paymentID)
  makes a payment for a song to artist and label for a particular period using the royalty rate

- makePodcastPayment(podcastName, hostID, paymentFee, bonusFee, paymentID)

Page: 6

**Point - 9** of the feedback given in Report - 1

Pg-9 of the original document and Pg-9 of the revised version. ArtistID is considered as one of the primary keys of Album entity. But since the album entity is a weak entity to the song entity and song entity itself is a weak entity to the artist entity, therefore the ArtistID was considered as the primary keys of the Album entity. It does not directly link to the artist entity. Therefore, no relationships were shown in the ER diagram. As a result there is no change in the revised edition.

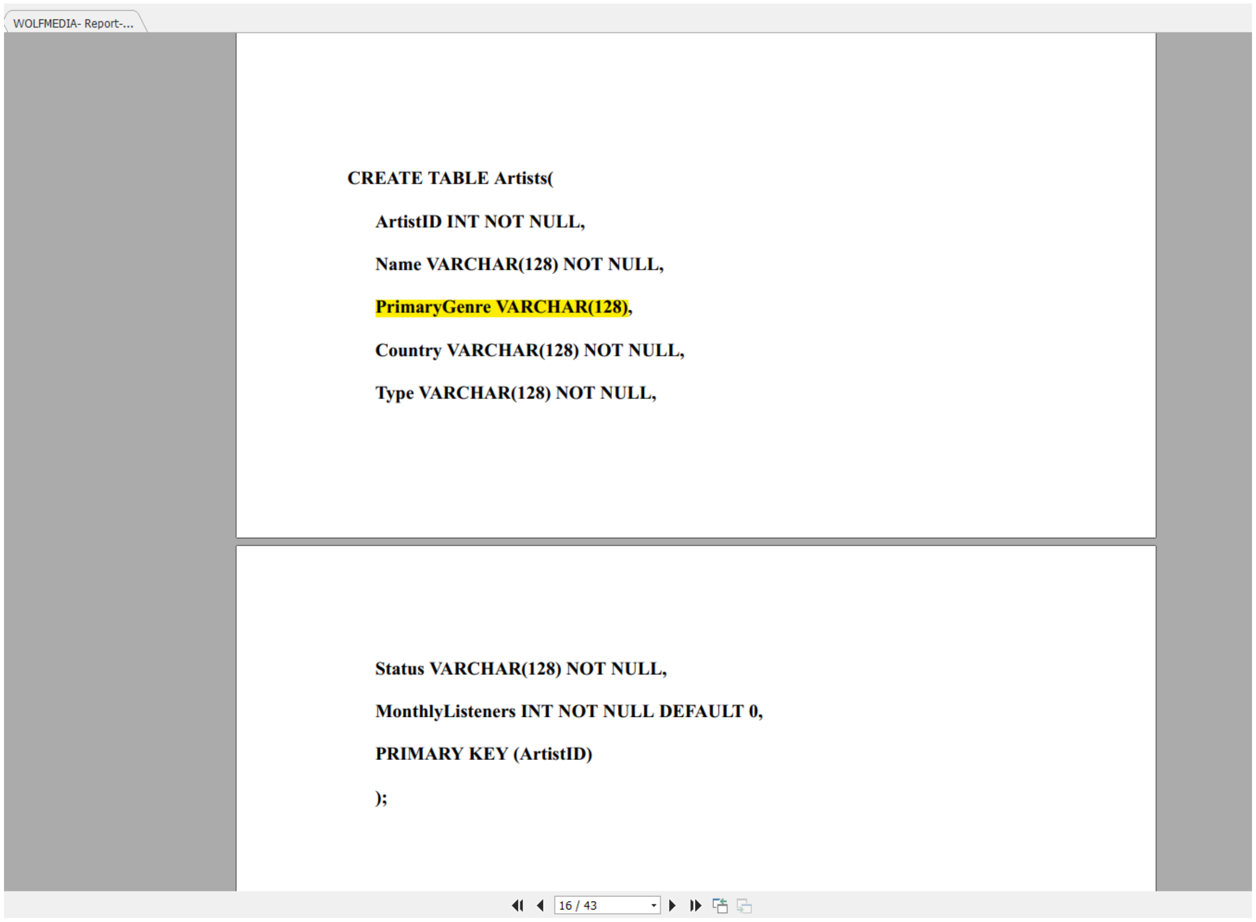


Page: 9

## **Modifications according to the feedback given in Report-2:**

### **Point - 1** of the feedback given in Report - 2:

The point mentioned that there is unclear relation between the table “Genre” and the table “Artists”. We don’t have the table “Genre” in the first place, so not sure where the unclear relation is coming from. The “Artists” table has the attribute primaryGenre which is the primary genre of the song (as shown in Pg - 16 of the revised version) that the artist has sung when the song might fall in the category of multiple genres.

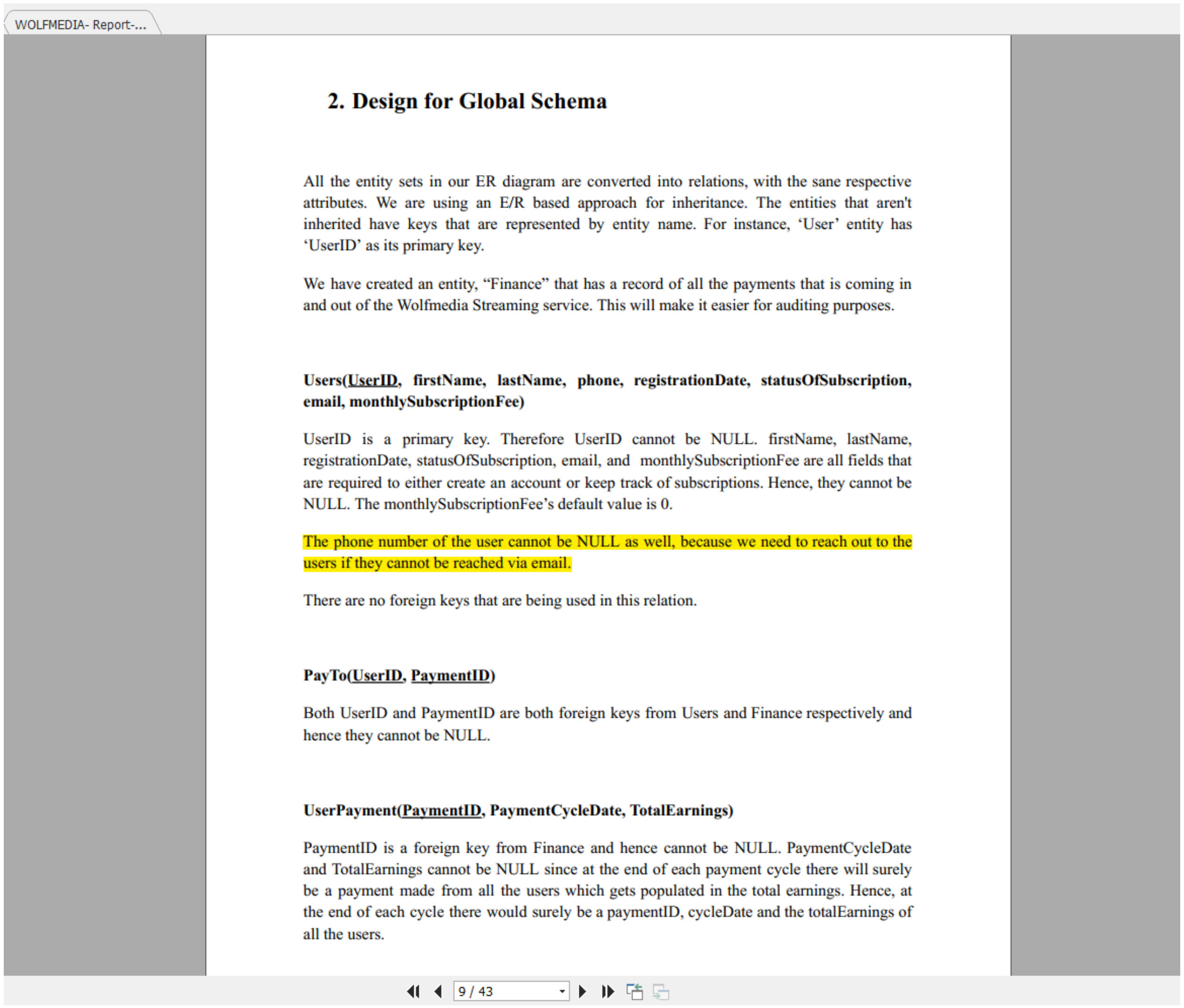


```
CREATE TABLE Artists(
    ArtistID INT NOT NULL,
    Name VARCHAR(128) NOT NULL,
    PrimaryGenre VARCHAR(128),
    Country VARCHAR(128) NOT NULL,
    Type VARCHAR(128) NOT NULL,
    Status VARCHAR(128) NOT NULL,
    MonthlyListeners INT NOT NULL DEFAULT 0,
    PRIMARY KEY (ArtistID)
    );
```

Page: 16

<u>**Point - 3** of the feedback given in Report - 2:</u>

Pg-9 of the original document and Pg-9,13 of the revised version. The 'phone' in the table named "Users" can not be NULL due to the fact that if the users cannot be reached via email, then they have to be contacted in some other way, which is by calling them. The reasons have been given and highlighted in Pg-9 of the revised edition and the highlighted segment of the table have been shown in Pg-13.


WOLFMEDIA- Report-...

## 2. Design for Global Schema

All the entity sets in our ER diagram are converted into relations, with the sane respective attributes. We are using an E/R based approach for inheritance. The entities that aren't inherited have keys that are represented by entity name. For instance, 'User' entity has 'UserID' as its primary key.

We have created an entity, "Finance" that has a record of all the payments that is coming in and out of the Wolfmedia Streaming service. This will make it easier for auditing purposes.

**Users(UserID, firstName, lastName, phone, registrationDate, statusOfSubscription, email, monthlySubscriptionFee)**

UserID is a primary key. Therefore UserID cannot be NULL. firstName, lastName, registrationDate, statusOfSubscription, email, and monthlySubscriptionFee are all fields that are required to either create an account or keep track of subscriptions. Hence, they cannot be NULL. The monthlySubscriptionFee's default value is 0.

The phone number of the user cannot be NULL as well, because we need to reach out to the users if they cannot be reached via email.

There are no foreign keys that are being used in this relation.

**PayTo(UserID, PaymentID)**

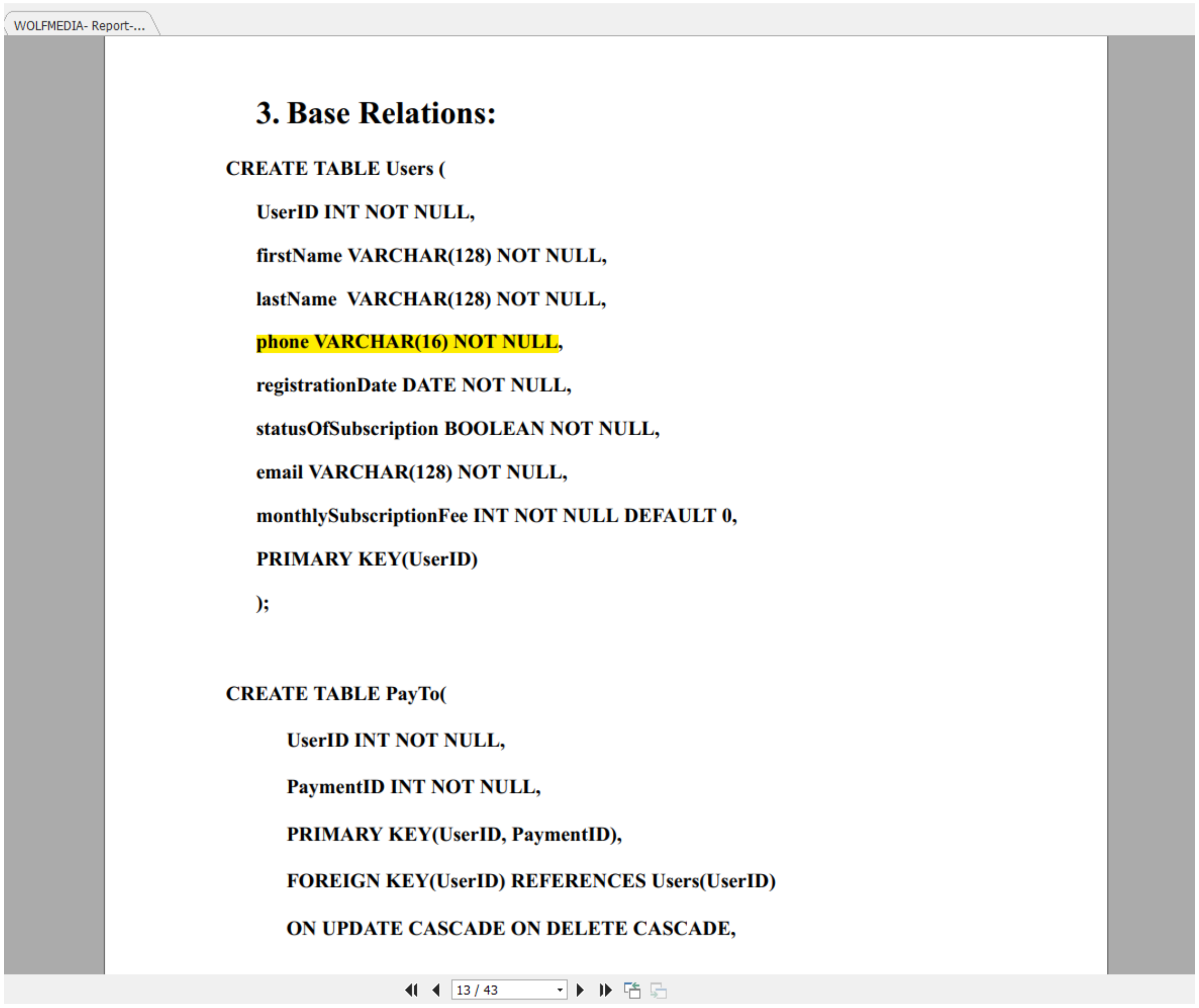
Both UserID and PaymentID are both foreign keys from Users and Finance respectively and hence they cannot be NULL.

**UserPayment(PaymentID, PaymentCycleDate, TotalEarnings)**

PaymentID is a foreign key from Finance and hence cannot be NULL. PaymentCycleDate and TotalEarnings cannot be NULL since at the end of each payment cycle there will surely be a payment made from all the users which gets populated in the total earnings. Hence, at the end of each cycle there would surely be a paymentID, cycleDate and the totalEarnings of all the users.

9 / 43

Page: 9


WOLFMEDIA- Report-...

## 3. Base Relations:

```
CREATE TABLE Users (
    UserID INT NOT NULL,
    firstName VARCHAR(128) NOT NULL,
    lastName  VARCHAR(128) NOT NULL,
    phone VARCHAR(16) NOT NULL,
    registrationDate DATE NOT NULL,
    statusOfSubscription BOOLEAN NOT NULL,
    email VARCHAR(128) NOT NULL,
    monthlySubscriptionFee INT NOT NULL DEFAULT 0,
    PRIMARY KEY(UserID)
    );

CREATE TABLE PayTo(
        UserID INT NOT NULL,
        PaymentID INT NOT NULL,
        PRIMARY KEY(UserID, PaymentID),
        FOREIGN KEY(UserID) REFERENCES Users(UserID)
        ON UPDATE CASCADE ON DELETE CASCADE,
```

13 / 43

Page:13

## Transactions:

### Transaction 1: Entering song into the Song table:

While entering a song we have to also enter the collaborated artists if there are multiple artists present. If there is a scenario where if the artists we entered in the collaborated artists table is not present in the artist table, it could cause the song to be entered into the songs table but the collaborated artists would not be present in the database. Since this is an inconsistent state, we need to rollback to the previous stable state of the database which is the state where the song was not inserted.

The line numbers for this transaction in the submitted codebase are 72 - 116.
Code snippet for the transaction:

```
def enterSong():
   try:
      cnx.start_transaction()
      selectArtist()

      # Open the cursor
      cursor = cnx.cursor()

      # Enter the new song details
      print("Enter your song details:")
      title = input("Enter the song Title: ")
      releaseDate = input("Enter the song release date (YYYY-MM-DD): ")
       artistID = input("Enter the ArtistID from the above mentioned Artists: 
")
      genres = input("Enter the Genre:")
      royaltyRate = float(input("Enter the Royalty Rate(Enter True/False):"))
      royaltyPaid = bool(input("Enter whether the royalty is paid or not:"))
      releaseCountry = input("Enter the release country:")
      language = input("Enter the language:")
      duration  = float(input("Enter the duration:"))
      playcount = 0

      sql_query = """INSERT INTO Songs
                          (Title, ReleaseDate, ArtistID, Genres, RoyaltyRate, 
RoyaltyPaid, ReleaseCountry, Language, Duration, Playcount)
                 VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)"""

            values = (title, releaseDate, artistID, genres, royaltyRate, 
royaltyPaid, releaseCountry, language, duration, playcount)

      # Execute the query to insert a song into the database
      cursor.execute(sql_query, values)

      # Input the collaborated hosts of the song
```

```
        flag = int(input("Please enter 1 if there are any collaborated artists
else enter 0: "))
      while(flag):
         collab_artistID = int(input("Enter the collaborated artist ID: "))
         sql_query = """INSERT INTO CollaboratedBy VALUES(%s, %s, %s, %s)"""

         values = (collab_artistID, title, releaseDate, artistID)
         cursor.execute(sql_query, values)

             flag = int(input("Please enter 1 if there are any collaborated
artists else enter 0: "))
      cnx.commit()

   except:
      print("There was an error so the changes are rolled back")
      cnx.rollback()

   # Cursor Close
   cursor.close()
```

**Transaction 2: Entering song into the Album table:**

While entering a song into an album, we first check whether the album exists or not. There can be two cases in this scenario.

1. A case where the album exists and we want to enter some songs.
2. The album does not exist and we want to enter some songs.

Since an album usually contains multiple songs, we need to add multiple songs into the album. Since this would require multiple inserts based on the user input, there could be a failure while entering a song into the album, for example:

Let us enter one correct song into the album, but the next song we enter into the album does not exist. This would cause an inconsistent state where we have one song and don't have the other song. So, we can't keep an inconsistent state in the database and hence we have to rollback the transaction so that we get back to the previous stable state of the database, i.e. having no songs entered into the album.

The line numbers for this transaction in the submitted codebase are 787 - 857.
Code snippet for the transaction:

```
def enterAlbum():

   # Open the cursor
   cursor = cnx.cursor()
   try:
      cnx.start_transaction()
      # Check if the album exists
      album_name = input("Enter album name: ")
      cursor = cnx.cursor()
      query = "SELECT * FROM Album WHERE Name=%s"
      cursor.execute(query, (album_name,))
      result = cursor.fetchall()

      # If Album exists
      if len(result)!= 0:

         # Find the current maximum tracknumber
         query1 = ("""
                SELECT MAX(TrackNumber) FROM Album WHERE Name = %s
            """)

         cursor.execute(query1, (album_name,))
         result1 = cursor.fetchone()
         maxTrackNumber = result1[0]

         # Keep adding the new songs 1 by 1 until not neccessary
          flag = int(input("Enter 1 if you want a song to be added else enter
0: "))
         while(flag):
            maxTrackNumber+=1

            # Take user input for song values
            title = input("Enter the song Title: ")
            releaseDate = input("Enter the song release date (YYYY-MM-DD): ")
            artistID = int(input("Enter the ArtistID: "))

              query = "INSERT INTO Album (Name, Title, ReleaseDate, ArtistID,
ReleaseYear, Edition, TrackNumber) VALUES (%s, %s, %s, %s, %s, %s, %s)"
               params = (album_name, title, releaseDate, artistID, result[4],
result[5], maxTrackNumber)

            cursor.execute(query, params)
               flag = int(input("Enter 1 if you want a song to be added else
enter 0: "))
         cnx.commit()


      # If album does not exist, create a new album
      else:
         print("Lets enter a new album")

         # Take user input for the album specific information
         release_year = int(input("Release year: "))
```

```
        edition = input("Edition: ")
        maxTrackNumber = 0

        # Keep adding the new songs 1 by 1 until not neccessary
         flag = int(input("Enter 1 if you want a song to be added else enter
0: "))

        while(flag):
           maxTrackNumber+=1

           # Take user input for song values
           title = input("Enter the song Title: ")
           releaseDate = input("Enter the song release date(YYYY-MM-DD): ")
           artistID = int(input("Enter the ArtistID: "))

             query = "INSERT INTO Album (Name, Title, ReleaseDate, ArtistID,
ReleaseYear, Edition, TrackNumber) VALUES (%s, %s, %s, %s, %s, %s, %s)"
            params = (album_name, title, releaseDate, artistID, release_year,
edition, maxTrackNumber)
           cursor.execute(query, params)
              flag = int(input("Enter 1 if you want a song to be added else
enter 0: "))

        cnx.commit()

   except:
      print("There was an error so the changes are rolled back")
      cnx.rollback()

   cursor.close()
```

## Design Decisions:

The system has a main menu which prompts the user to enter a number from 1-27 corresponding with the kind of action they would like to take such as “Enter Song”, “Delete Artist”, etc. Once an option is selected by the user by entering the corresponding number, the user might be prompted to make further selections which would further lead to more specific operations. In our system we created a cursor object to execute SQL queries. In our code we make use of the MySQL connector, which is a driver for connecting Python to MariaDB databases. In order to do the commits and rollbacks, the helper methods of the MySQLConnection class have been used that commits the current transaction to the database as well as rolls back the transaction to the previous stable state of the database, respectively. We also made use of try and except statements to catch any errors that might occur when taking inputs from the users.

**Functional Roles:**

**Part 1:**
Software Engineer: Arun Ramesh(Prime), Adittya Soukarjya Saha(Backup)
Database Designer/Administrator: Adittya Soukarjya Saha(Prime), Kiron Jayesh(Backup)
Application Programmer: Arun Srinivasan Parthasarathy(Prime), Arun Ramesh(Backup)
Test Plan Engineer: Kiron Jayesh(Prime), Arun Srinivasan Parthasarathy(Backup)
**Part 2:**
Software Engineer: Kiron Jayesh(Prime), Adittya Soukarjya Saha(Backup)
Database Designer/Administrator: Arun Srinivasan Parthasarathy(Prime), Arun Ramesh (Backup)
Application Programmer: Adittya Soukarjya Saha(Prime), Arun Srinivasan Parthasarathy (Backup)
Test Plan Engineer: Arun Ramesh(Prime), Kiron Jayesh(Backup)
**Part 3:**
Software Engineer: Arun Srinivasan Parthasarathy(Prime), Kiron Jayesh(Backup)
Database Designer/Administrator: Arun Ramesh(Prime), Adittya Soukarjya Saha(Backup)
Application Programmer: Kiron Jayesh(Prime), Arun Ramesh(Backup)
Test Plan Engineer: Adittya Soukarjya Saha(Prime), Arun Srinivasan Parthasarathy(Backup)

How to Run the Code:

```
python apis.py
```