Library Final Report

Submitted to:

Leon Madrid

Written by:

Landen Master

Robert Lawrence

David Tochtermann

CSE 3241

The Ohio State University

Columbus, Ohio

30 November 2021

Table of Contents

Introduction

The following report pertains to a proposed database for a local library, who asked to manage their music and video collections to support their inventory and circulation operations. Given a large set of requirements and deliverables, including creating a Java application to interface with the database, the following information firstly details the high level database information in the Database Description section. In the User Manual section, more in-depth analysis of the database and its contents are explored, including its description and queries.

I. Database Description

1. Data Models and Schemas

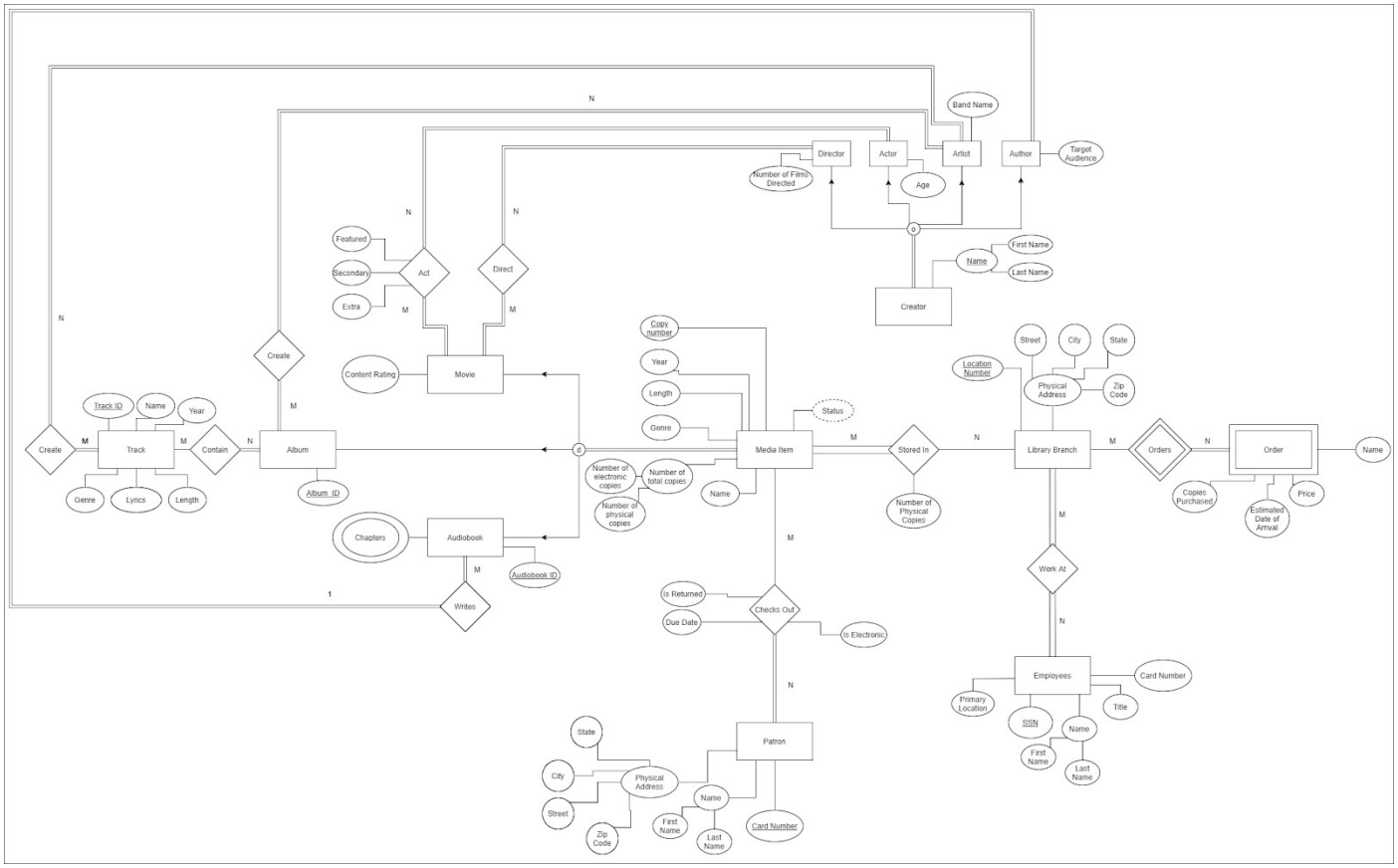

Figure 1: Entity-Relationship Model Diagram of Library Database

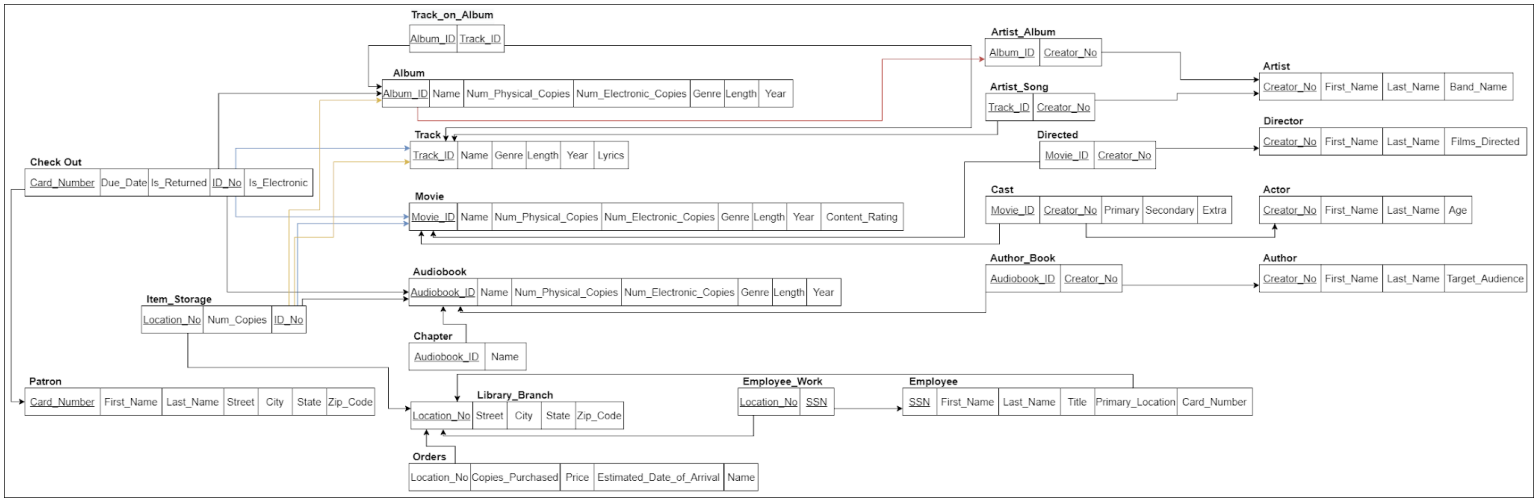

Figure 2: Relational Schema of Library Database with foreign keys

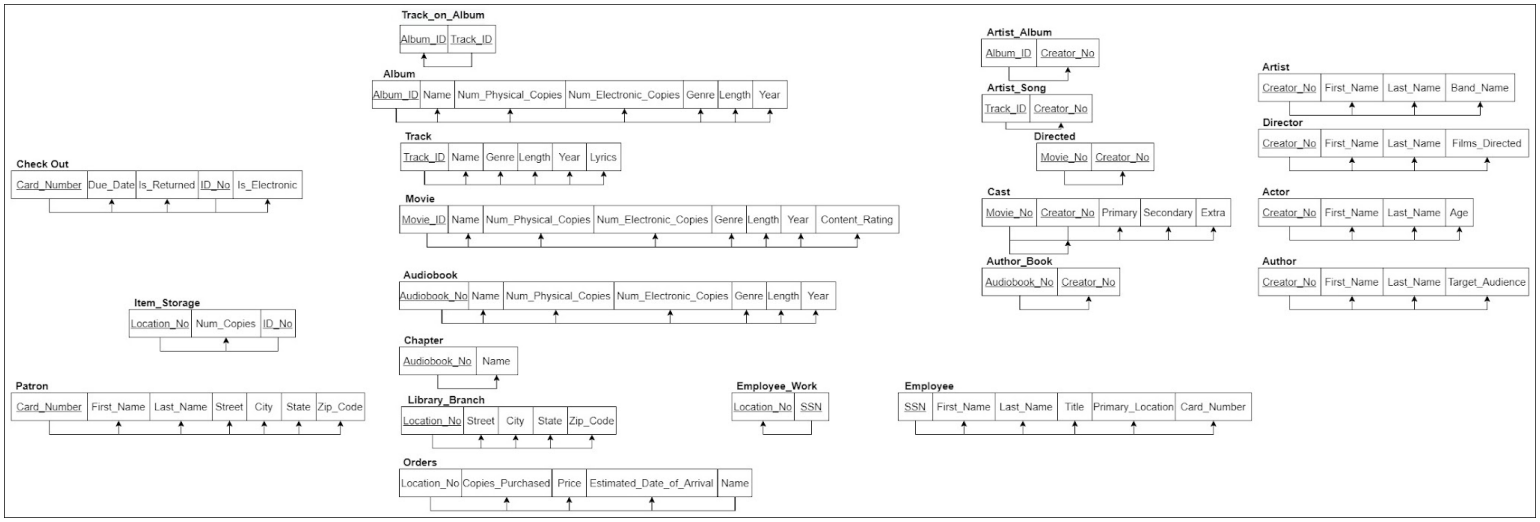

Figure 3: Relational Schema of Library Database with functional dependencies

The figures listed above will also be provided separately for easier viewing.

2. Normal Forms of Tables

To be in the First Normal Form, all attributes on a table must be atomic and all attributes are dependent on the primary key. To be in the Second Normal Form, a table must be in the First Normal Form and all non-key attributes must be fully dependent on the key. To be in the Third Normal Form, a table must be in the Second Normal Form and no attributes may be dependent on a non-key. Finally, to be in Boyce-Codd Normal Form a table must be in the Third Normal Form and all determinants must be a part of the candidate key. All of the functional dependencies below can be seen above in Figure 3.

Cast is in Third Normal Form as it is atomic and all attributes are dependent on Movie_ID and Creator_No, the primary keys. In addition, all non-key attributes are fully dependent on the Movie_ID and Creator_No, and no attributes are dependent on a non-key. However, it is not in Boyce-Codd Normal Form as Creator_No, one of the keys, is determined by Movie_ID. This can

not be decomposed as both Movie_ID and Creator_No are needed to determine Primary, Secondary, and Extra, and thus this relation's highest normalization is Third Normal Form.

Patron is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Card_Number and ID_No, the primary keys. In addition, all non-key attributes are fully dependent on the Card_Number and ID_No, and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Library_Branch is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Location_Number, the primary key. In addition, all non-key attributes are fully dependent on the Location_Number and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Artist is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Creator_No, the primary key. In addition, all non-key attributes are fully dependent on the Creator_No and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Director is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Creator_No, the primary key. In addition, all non-key attributes are fully dependent on the Creator_No and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Actor is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Creator_No, the primary key. In addition, all non-key attributes are fully dependent on the Creator_No and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Author is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Creator_No, the primary key. In addition, all non-key attributes are fully dependent on the Creator_No and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Album is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Album_ID, the primary key. In addition, all non-key attributes are fully dependent on the Album_ID and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Movie is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Movie_ID, the primary key. In addition, all non-key attributes are fully dependent on the Movie_ID and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Audiobook is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Audiobook_ID, the primary key. In addition, all non-key attributes are fully dependent on the Audiobook_ID and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Track is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Track_ID, the primary key. In addition, all non-key attributes are fully dependent on the Track_ID and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Employee is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on SSN, the primary key. In addition, all non-key attributes are fully dependent on the SSN and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Order is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Location_No and SSN, the primary keys. In addition, all non-key attributes are fully dependent on the Location_No and SSN, and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Artist_Song is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Track_ID, the primary key. In addition, all non-key attributes are fully dependent on the Track_ID and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Artist_Album is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Album_ID, the primary key. In addition, all non-key attributes are fully dependent on the Album_ID and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Author_Book is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Audiobook_ID, the primary key. In addition, all non-key attributes are fully dependent on the Audiobook_ID and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Directed is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Movie_ID, the primary key. In addition, all non-key attributes are fully dependent on the Movie_ID and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Track_on_Album is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Track_ID, the primary key. In addition, all non-key attributes are fully dependent

on the Track_ID and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Chapter is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Audiobook_ID, the primary key. In addition, all non-key attributes are fully dependent on the Audiobook_ID and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Item_Storage is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Location_No and ID_No, the primary keys. In addition, all non-key attributes are fully dependent on the Location_No and ID_No, and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Check_Out is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on Card_Number and ID_No, the primary keys. In addition, all non-key attributes are fully dependent on the Card_Number and ID_No, and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

Employee_Work is in Boyce-Codd Normal Form as it is atomic and all attributes are dependent on SSN, the primary key. In addition, all non-key attributes are fully dependent on the SSN and no attributes are dependent on a non-key. Finally, all determinants are a part of the candidate key.

3. Implemented Indexes

The first index implemented in the database relates to the inventory of each library branch. The SQL code for this index is 'CREATE INDEX inventory ON ITEM_STORAGE(ID_No);' This index will provide the ability to quickly search through and locate all items in each specific library location. Since every media item at a location has its own ID_No, creating this index will

greatly reduce the amount of time that it takes to search through the database to find a specific ID_No at any given location.

Another index implemented in the database relates to the card number of each patron. The SQL code for this index is 'CREATE UNIQUE INDEX card_no_index ON PATRON(Card_Number);' This index will prove useful as it is often that a library may lookup a customer's card number to evaluate things such as whether the customer has a card, the address of a customer, and can also use the card number using an equivalence test to see what items they have checked out or items they might have ordered.

The final index implemented in this database regards the types of items checked out by patrons. The SQL Code for this index is 'CREATE INDEX type_of_item ON CHECK_OUT(Media_Type);' This index sorts all media items checked out based on what kind of media item it is. This will be useful as finding a checked out item is based on its media type, as different media items ID numbers are not unique, and thus media type must be identified. This will be useful for many queries, such as the runtime of all movies checked out, patrons with an above average check out, and the most popular actors or bands in the database.

4. Implemented Views

The first view implemented regards the total inventory of each library branch. This view produces an overview of how many items are checked out at all locations. This view would be useful for employees, likely managers, to see how many items are checked out at any given time at a specific location. The view can be produced with the following relational algebra:

$$\rho_L(ITEM\_STORAGE \bowtie_{ITEM\_STORAGE.Location\_No=LIBRARY\_BRANCH.Location\_No} LIBRARY\_BRANCH)$$

$$Checked\_Out \leftarrow {}_{L.Location\_No}\mathcal{F}_{COUNT\ ID\_No}(CHECK\_OUT)$$

$$\pi_{L.Location\_Number,Checked\_Out}(\sigma_{CHECK\_OUT.ID\_No=L.ID\_No}(\sigma_{CHECK\_OUT.Is\_Returned=false}(L)))$$

Alternatively, the following SQL statements can create the same view for the database:

```
CREATE VIEW CHECKED_OUT_AT_LOCATION
AS
SELECT L.Location_Number,count(C.ID_No) AS Checked_Out
FROM CHECK_OUT AS C, (ITEM_STORAGE AS I JOIN LIBRARY_BRANCH AS
B ON I.Location_No=B.Location_Number) AS L
WHERE C.ID_No=L.ID_No
AND C.Is_Returned=false
GROUP BY L.Location_Number;
```

A sample of the view using the sample database can be seen below in figure 4.

Location_Number	**Checked_Out**
2	8
3	1
5	12
7	1
9	1
14	1
17	1
19	2

Figure 4: Sample of Checked_Out_At_Location View

The second view implemented regards actors and their movies. This view shows all movies that an actor has starred in. This view may be useful to patrons who are searching for movies that

have their favorite actor starring in it. The view can be produced with the following relational algebra:

$$Acts \leftarrow ACTOR \bowtie_{ACTOR.Creator\_No=CAST.Creator\_No} CAST$$

$$\pi_{MOVIE.Name}(\sigma_{Acts.Creator\_No=1}(\sigma_{MOVIE.Movie\_ID=Acts.Movie\_ID}$$

$$(\sigma_{Acts.Primary\_Actor=true\ OR\ Acts.Secondary=true\ OR\ Acts.Extra=true}(Acts))))$$

Alternatively, the following SQL statements can create the same view for the database:

```
CREATE VIEW ACTOR_PAGE
AS
SELECT Acts.First_Name, Acts.Last_Name, MOVIE.Name as Movie_Name
FROM (ACTOR JOIN [CAST] ON ACTOR.Creator_No=[CAST].Creator_No) AS Acts,
MOVIE
WHERE Acts.Creator_No > 0 AND MOVIE.Movie_ID = Acts.Movie_ID
ORDER BY Last_Name ASC;
```

Where SPECIFY is the creator number of the actor to make a page for.

A sample of the view using the sample database can be seen below in figure 5 using creator numbers 1, 3, and 10.

First_Name	Last_Name	Movie_Name
Jack	Nicholson	Views
Jack	Nicholson	One Flew Over the Cuckoo's Nest
Al	Pachino	The Godfather
Al	Pachino	Goodfellas
Landen	Master	Raiders of the Lost Ark
Landen	Master	Landens First Movie

Figure 5: Sample of ACTOR_PAGE View

5. Sample Transactions

One sample transaction that may happen on the database is the following:

```
BEGIN TRANSACTION;
INSERT OR ROLLBACK INTO ALBUM
VALUES (30, 'Yeezus', 20, 20, 'Rap', 2401, 2013);
INSERT OR ROLLBACK INTO ARTIST_ALBUM VALUES (30, 7);
COMMIT;
```

This sample transaction is useful to the database since it adds a new album to the to the inventory of the library, and updates the database to include the creator of the new album. This is a useful transaction because it takes the new album and relates it to an existing artist. Now, all information relating to the album artist can be traced back, and the album artist can include it in their discography.

Another sample transaction that may happen on the database is the following:

```
BEGIN TRANSACTION;
INSERT OR ROLLBACK INTO ITEM_STORAGE
VALUES (1, 17, 4, 'Movie');
UPDATE OR ROLLBACK ORDERS
    SET Estimated_Date_of_Arrival = 'now'
    WHERE Location_Number = 1 AND Name = 'Blade Runner';
COMMIT;
```

This sample transaction is useful to the database since it adds to the inventory of the library from a previous order, then updates the order to show that it has arrived. This is a useful transaction

because it takes the information from the Orders table, which contained a new item that was previously not held at the library, and places it in the Item_Storage table to account for its arrival. The Orders table is updated to replace the Estimated_Time_of_Arrival to the current date and time to show that it has arrived.

One final sample transaction that may happen on the database is the following:

```
BEGIN TRANSACTION;
INSERT OR ROLLBACK INTO LIBRARY_BRANCH
    VALUES ('21', '800 Vine Street', 'Cincinnati',
    'Ohio', '45202');
UPDATE OR ROLLBACK EMPLOYEE
    SET Primary_Location = '21'
    WHERE Title = 'Librarian' AND Primary_Location = '3';
COMMIT;
```

This transaction is to update an employee who will be transferred to a new library location. If a new location has just opened, and an experienced librarian from another branch was called to transfer to help open up the store and get new employees up to speed, this would be useful. This transaction will insert values into the Library_Branch table to write the new values of details of the new location to the table, and then will update the Employee information to reflect that their new location is branch 21, if they are currently a librarian at branch 3.

II. User Manual

1. Table Descriptions

PATRON: This table represents all information pertaining to a patron of the library system. They are identified by *Card_Number*, which is a 9 digit string of numbers. Patrons also all have their names in the database, stored by *First_Name* and *Last_Name*, both strings of up to 15 letters. Their emails are also stored by *Email*, a string of up to 100 letters. Patrons may also have their addresses stored, but this may be inputted later. Their address includes *Street*, a string of up to 25 letters, *City*, a string of up to 25 letters, *State*, a string of up to 13 letters, and *Zip_Code*, a 5 letter string.

LIBRARY_BRANCH: This table represents all information pertaining to a branch in the library system. They are identified by their *Location_Number*, an integer assigned to each location at their opening. Their addresses are also stored on their opening. Addresses include *Street*, a string of up to 25 letters, *City*, a string of up to 25 letters, *State*, a string of up to 13 letters, and *Zip_Code*, a 5 letter string.

ARTIST: This table represents all information pertaining to a musical artist in the database. They are identified by their *Creator_No*, an integer assigned to them when being inputted in the database. They have their names stored as *First_Name*, a string of up to 25 letters. Their last name, stored by *Last_Name*, may be null for artists with one name, such as Drake or Cher. When stored, it is held by a string of up to 15 letters. If part of a band, their band name is stored as *Band_Name*, a string of up to 30 letters. If not part of a band, this may also be left as null.

DIRECTOR: This table represents all information pertaining to a movie director in the database. They are identified by their *Creator_No*, an integer assigned to them when being

inputted in the database. Directors also all have their names in the database, stored by *First_Name* and *Last_Name*, both strings of up to 15 letters. Optionally, the number of films they have directed may be stored under *Films_Directed* as an integer.

ACTOR: This table represents all information pertaining to an actor in the database. They are identified by their *Creator_No*, an integer assigned to them when being inputted in the database. Actors also all have their names in the database, stored by *First_Name* and *Last_Name*, both strings of up to 15 letters. Optionally, their age may be stored under *Age* as an integer.

AUTHOR: This table represents all information pertaining to an author in the database. They are identified by their *Creator_No*, an integer assigned to them when being inputted in the database. Actors also all have their names in the database, stored by *First_Name* and *Last_Name*, both strings of up to 15 letters. Optionally, their target audience may be stored under *Target_Audience* as a string of up to 20 letters.

ALBUM: This table represents all information pertaining to albums in the database. They are identified by their *Album_ID*, an integer assigned to them when being inputted in the database. Their name is stored under *Name*, a string of up to 100 letters. The number of physical and digital copies in the library system are stored under *Num_Physical_Copies* and *Num_Electronic_Copies* as integers. The genre of the album may be stored under *Genre* as a string of up to 20 letters, but may be inputted later if the genre is unknown. Similarly, the length of the album in seconds and year of creation may be stored under *Length* and *Year* as integers, but may be inputted later.

MOVIE: This table represents all information pertaining to movies in the database. They are identified by their *Movie_ID*, an integer assigned to them when being inputted in the database. Their name is stored under *Name*, a string of up to 100 letters. The number of physical and

digital copies in the library system are stored under *Num_Physical_Copies* and *Num_Electronic_Copies* as integers. The rating of the movie is stored by *Content_Rating*, a string of up to 15 letters. The genre of the movie may be stored under *Genre* as a string of up to 20 letters, but may be inputted later if the genre is unknown. Similarly, the length of the movie in seconds and year of creation may be stored under *Length* and *Year* as integers, but may be inputted later.

AUDIOBOOK: This table represents all information pertaining to audiobooks in the database. They are identified by their *Audiobook_ID*, an integer assigned to them when being inputted in the database. Their name is stored under *Name*, a string of up to 100 letters. The number of physical and digital copies in the library system are stored under *Num_Physical_Copies* and *Num_Electronic_Copies* as integers. The genre of the audiobook may be stored under *Genre* as a string of up to 20 letters, but may be inputted later if the genre is unknown. Similarly, the length of the audiobook in seconds and year of creation may be stored under *Length* and *Year* as integers, but may be inputted later.

TRACK: This table represents all information pertaining to songs in the database. They are identified by their *Track_ID*, an integer assigned to them when being inputted in the database. Their name is stored under *Name*, a string of up to 100 letters. The genre of the track may be stored under *Genre* as a string of up to 20 letters, but may be inputted later if the genre is unknown. Similarly, the length of the track in seconds and year of creation may be stored under *Length* and *Year* as integers, but may be inputted later. Optionally, the lyrics of the song may be stored in *Lyrics*, a string of up to 10000 letters.

EMPLOYEE: This table represents all information pertaining to employees working for the library. They are identified by their Social Security Number, stored under *SSN* as a 9 digit string.

Employee's names in the database, stored by *First_Name* and *Last_Name*, both string up to 15 letters. Their job title is stored under *Title* as a string of up to 30 letters. The location they work at most often is stored under *Primary_Location* as an integer, which references *Location_Number* from the **LIBRARY_BRANCH** table. Optionally, employees may obtain a library card, and their card number is stored under *Card_Number*, a unique 9 digit string.

ORDER: This table represents all information on media items ordered by the library. Each Order contains which location it is being shipped to as *Location_No*, which references *Location_Number* from the **LIBRARY_BRANCH** table. The name of the media item being ordered is stored as *Name*, a string of up to 100 letters. In addition, each order has the number of copies purchased and price as *Copies_Purchased* and *Price*, which are integers. The date the order is expected to arrive is stored in *Estimated_Date_of_Arrival*, a date. Finally, the type of media item is stored under *Type*, which is a string of up to 10 letters.

ARTIST_SONG: This table represents the relation between artists and tracks, and shows what songs an artist has recorded. They are identified by their *Track_ID*, an integer which references *Track_ID* from the **TRACK** table which identifies what track is being referenced. They are also identified by their *Creator_No*, an integer which references *Creator_No* from the **ARTIST** table which identifies what artist created the referenced song.

ARTIST_ALBUM: This table represents the relation between artists and albums, and shows what albums an artist has recorded. They are identified by their *Album_ID*, an integer which references *Album_ID* from the **ALBUM** table which identifies what album is being referenced. They are also identified by their *Creator_No*, an integer which references *Creator_No* from the **ARTIST** table which identifies what artist created the referenced album.

AUTHOR_BOOK: This table represents the relation between authors and audiobooks, and shows what books an author has written. They are identified by their *Audiobook_ID*, an integer which references *Audiobook_ID* from the **AUDIOBOOK** table which identifies what audiobook is being referenced. They are also identified by their *Creator_No*, an integer which references *Creator_No* from the **AUTHOR** table which identifies what author created the referenced book.

DIRECTED: This table represents the relation between movies and directors, and shows what movies a director has directed. They are identified by their *Movie_ID*, an integer which references *Movie_ID* from the **MOVIE** table which identifies what movie is being referenced. They are also identified by their *Creator_No*, an integer which references *Creator_No* from the **DIRECTOR** table which identifies what director directed the referenced movie.

CAST: This table represents the relation between actors and directors, and shows what movies an actor has acted in. They are identified by their *Movie_ID*, an integer which references *Movie_ID* from the **MOVIE** table which identifies what movie is being referenced. They are also identified by their *Creator_No*, an integer which references *Creator_No* from the **ACTOR** table which identifies what actor is in the referenced movie. Optionally, the boolean values of *Primary_Actor*, *Secondary*, and *Extra* reference if the actor in the referenced movie is a lead actor, a secondary actor, or an extra.

TRACK_ON_ALBUM: This table represents the relation between tracks and albums, and shows what tracks are on what albums. They are identified by their *Track_ID*, an integer which references *Track_ID* from the **TRACK** table which identifies what track is being referenced. They are also identified by their *Album_ID*, an integer which references *Album_ID* from the **ALBUM** table which identifies what album the referenced track is on.

CHAPTER: This table represents the names of chapters in audiobooks. They contain an *Audiobook_ID*, an integer which references *Audiobook_ID* from the **AUDIOBOOK** table which identifies what audiobook is being referenced. In addition, the name of a chapter is stored under *Name* as a string of up to 30 letters.

ITEM_STORAGE: This table represents information on a media item at a specific library branch. They are identified by their *Location_No*, an integer which references *Location_No* from the **LIBRARY_BRANCH** table which identifies what location the media item is stored at. They are also identified by their *ID_No*, an integer which references the unique ID number for each type of media item, those being Movies, Audiobooks, Albums, and Tracks. The type of media item being held is specified by *Type*, a string of up to 10 letters. Finally, the number of copies of the media item at the specified branch is stored as an integer in *Num_Copies*.

CHECK_OUT: This table represents information on a checked out media item. *Card_Number* is a 9 digit string which references *Card_Number* from the **PATRON** table which identifies what patron is checking out the current media item. *ID_No*, an integer, references the unique ID number for each type of media item, those being Movies, Audiobooks, Albums, and Tracks. The type of media item being held is specified by *Type*, a string of up to 10 letters. Finding if the item checked out is physical or electronic is stored in the boolean value *Is_Electronic*. The due date of the media item is stored in *Due_Date* as a date. Finally, finding if the current item has been returned is stored as the boolean value *Is_returned*.

EMPLOYEE_WORK: This table represents the working locations of different employees. They are identified by their *SSN*, a 9 digit string referencing the *SSN* from the **EMPLOYEE** table which identifies the current employee. They are also identified by their *Location_No*, an

integer which references *Location_No* from the **LIBRARY_BRANCH** table which identifies what location the employee is working at.

2. Example Queries

a. Media Items Queries

One quality media item query is one which finds the titles of all tracks by a specified artist before a specified year, which can be seen in the SQL code below:

```
SELECT TRACK.Name
FROM TRACK, ARTIST, ARTIST_SONG
WHERE TRACK.Track_ID = ARTIST_SONG.Track_ID AND
ARTIST_SONG.Creator_No = ARTIST.Creator_No AND Year < SPYEAR AND
ARTIST.Creator_No = SPART
GROUP BY Name;
```

Where SPYEAR is the specified year and SPART is the Creator number of the specified artist. Alternatively, the query can be produced with the following relational algebra:

$$\pi_{Track.Name}\sigma_{Year = SPYEAR,\ Creator\_No = SPART}(TRACK \bowtie_{Track.Track\_ID = Arist\_Song.Track\_ID} ARTIST \text{ AND } ARTIST\_SONG \bowtie_{Creator\_No = Creator\_No} ARTIST))$$

Another good media item query is one which lists all the albums and their unique identifiers with less than 2 copies held by the library, which can be seen in the SQL code below:

```
SELECT ALBUM_ID, Name, Genre, Length, Year
FROM ALBUM
WHERE Num_Electronic_Copies + Num_Physical_Copies < 2;
```

Alternatively, the query can be produced with the following relational algebra:

$$\pi_{Album\_ID,\, Name,\, Genre,\, Length,\, Year}\,(\sigma Num\_Electronic\_Copies\ +\ Num\_Physical\_Copies\ <\ 2\,(ALBUM))$$

b. Patrons Queries

One quality query regarding patrons is one which finds the names of employees who are also patrons, which can be seen in the SQL code below:

```
SELECT First_Name, Last_Name
FROM (EMPLOYEE AS E JOIN PATRON AS P ON E.Card_Number =
P.Card_Number);
```

Alternatively, the query can be produced with the following relational algebra:

$$\pi_{First\_Name,\, Last\_Name}(\sigma\ EMPLOYEE \bowtie_{Card\_Number\,=\,Card\_Number} PATRON)$$

Another quality query regarding patrons is one which finds the names of all patrons in a specified zip code, which can be seen in the SQL code below:

```
SELECT First_Name, Last_Name
FROM Patron
WHERE Zip_Code=SPZIP;
```

Where SPZIP is the specified zip code. Alternatively, the query can be produced with the following relational algebra:

$$\pi_{First\_Name,Last\_Name}(\sigma_{Zip\_Code=ZIP}(PATRON))$$

c. Checked Out Movie Queries

One quality query regarding checked out movies is one which finds all movies and the date of their checkout from a single patron, which can be seen in the SQL code below:

```
SELECT MOVIE.Name, CHECK_OUT.Due_Date
FROM CHECK_OUT, MOVIE
WHERE CHECK_OUT.Media_Type = 'Movie' AND CHECK_OUT.ID_No =
MOVIE.Movie_ID AND CHECK_OUT.Card_Number = SPATRON;
```

Alternatively, the query can be produced with the following relational algebra:

$$\pi_{MOVIE.Name,\ CHECK\_OUT.Due\_Date}\ (\sigma_{\ Media\_Type\ =\ "Movie"}\ (CHECK\_OUT))\ AND$$

$$(CHECK\_OUT \bowtie_{\ Id\_No\ =\ Movie\_ID}\ MOVIE\ AND\ CHECK\_OUT \bowtie_{\ Card\_Number\ =\ Card\_Number}$$

$$PATRON)$$

Another quality query regarding checked out movies is one which finds the patron who has checked out the most movies and the total number of movies they have checked out, which can be seen with the SQL code below:

```
SELECT First_Name, Last_Name, COUNT(ID_No) as Num_Movies_Checked_Out
FROM Patron, Check_Out
WHERE Patron.Card_Number=Check_Out.Card_Number
AND Check_Out.Media_Type="Movie"
GROUP BY First_Name
ORDER BY COUNT(ID_No) DESC
LIMIT 1;
```

Alternatively, the query can be produced with the following relational algebra:

$$\pi_{First\_Name,Last\_Name}(\sigma_{Patron.Card\_Number=Check\_Out.Card\_Number}$$

$$(\sigma_{Check\_Out.Media\_Type="Movie"}(PATRON \cup CHECK\_OUT)))$$

$$_{First\_Name}\mathcal{F}_{MAX\ COUNT\ ID\_No}(PATRON \cup CHECK\_OUT)$$

Another good query that relates to checked out movies is one that provides a list of patron names, along with the total combined running time of all the movies they have checked out, which can be seen in the SQL code below:

```
SELECT Patron.First_Name, Patron.Last_Name, SUM(Length) as Total_Runtime
FROM (
SELECT Check_Out.Card_Number, Movie.Length
FROM Movie, Check_Out
WHERE Check_Out.Media_Type = 'Movie' AND Check_Out.ID_No =
Movie.Movie_ID) as M, Patron
WHERE Patron.Card_Number = M.Card_Number
GROUP BY Patron.First_Name;
```

Alternatively, the query can be produced with the following relational algebra:

$$R1 \leftarrow \pi_{Check\_Out.Card\_Number,Movie.Length}(\sigma_{Check\_Out.Media\_Type="Movie"}$$

$$(\sigma_{Check\_Out.ID\_No=Movie.Movie\_ID}(MOVIE \cup CHECK\_OUT)))$$

$$\pi_{Patron.First\_Name,Patron.Last\_Name}(\sigma_{Patron.Card\_Number=R1.Card\_Number}(R1 \cup PATRON))$$

$$_{Patron.First\_Name}\mathcal{F}_{SUM\ Length}(R1)$$

Finally, a great query regarding checked out movies is one that provides a list of the movies in the database and associated total copies lent to patrons, sorted from the movie that has been lent the most to the movies that has been lent the least, which can be seen in the SQL code below:

```
SELECT mov.Name AS Name, COUNT(co.Id_No) as Count
FROM MOVIE mov INNER JOIN CHECK_OUT co ON mov.Movie_Id = co.Id_No
WHERE co.Media_Type = 'Movie'
GROUP BY co.Id_No, desc;
```

Alternatively, the query can be produced with the following relational algebra:

$$\pi_{Name}\, \Im_{COUNT\ ID\_No\,(CHECK\_OUT)}\,((\sigma\ MOVIE \bowtie_{Movie\_ID\ =\ ID\_No}\ CHECK\_OUT)\ AND$$

$$Media\_Type\ =\ 'Movie'\,(CHECK\_OUT))$$

Finally, another great query regarding checked out movies would be one which provides a list of customer information for patrons who have checked out anything by the most watched actors in the database, which can be seen in the SQL code below:

```
SELECT PATRON.*
FROM PATRON, CHECK_OUT, "CAST"
WHERE CHECK_OUT.Card_Number = PATRON.Card_Number AND
CHECK_OUT.Media_Type = "Movie" AND CHECK_OUT.ID_No =
"CAST".Movie_ID AND "CAST".Creator_No =
(SELECT hold FROM
(SELECT hold, MAX(holder) AS "Movies checked out"
FROM
(SELECT Actor.Creator_No AS hold, Count(Actor.Creator_No) as holder
FROM CHECK_OUT, "CAST", ACTOR
WHERE CHECK_OUT.Media_Type = "Movie" AND CHECK_OUT.ID_No =
"CAST".Movie_ID AND "CAST".Creator_No = ACTOR.Creator_No
GROUP BY Actor.Creator_No)))
GROUP BY Patron.Card_Number;
```

Alternatively, the query can be produced with the following relational algebra:

$$\rho Result1\ (\sigma_{Creator\_No,\ \Im COUNT\ Creator\_No(ACTOR)}(\sigma Media\_Type\ =\ 'Movie'$$

$$(CHECK\_OUT)\ AND\ CHECK\_OUT \bowtie_{ID\_No\ =\ Movie\_ID} MOVIE\_CAST)))$$

$$\rho Result2\ (\sigma_{Album.Creator\_No,\ \Im MAX\ Creator\_No(ACTOR)}(Result1))$$

$$\rho Result\ \pi_{Card\_Number,\ First\_Name,\ Last\_Name,\ Street,\ City,\ State,\ Zip\_Code,\ Email}$$

$$(\sigma Media\_Type\ =\ 'Movie'(CHECK\_OUT)\ AND\ CHECK\_OUT \bowtie_{ID\_No\ =\ Movie\_ID}$$

$$MOVIE\_CAST\ AND\ Creator\_No\ (MOVIE)\ =\ Result2)$$

d. Checked Out Actor Queries

One quality query regarding checked out actors is one which finds all patrons who checked out a movie by a specified actor and the movies they checked out, which can be seen in the SQL code below:

```
SELECT First_Name, Last_Name, MOV.Name
FROM (
    SELECT First_Name, Last_Name, Movie.Name, Movie.Movie_ID
    FROM (
        SELECT First_Name, Last_Name, ID_No
        FROM Patron, Check_Out
        WHERE Patron.Card_Number=Check_Out.Card_Number
        AND Media_Type="Movie"
        ) AS COMov, Movie
    WHERE COMov.ID_No=Movie.Movie_ID
    ) AS MOV, [Cast] AS C
WHERE MOV.Movie_ID=C.Movie_ID AND C.Creator_No= SPACTOR;
```

Where SPACTOR is the Creator Number of the specified actor. Alternatively, this query can be produced with the following relational algebra:

$$\rho_{COMov}(\sigma_{Patron.Card\_Number=Check\_Out.Card\_Number}(\sigma_{Media\_Type="Movie"}(PATRON \cup CHECK\_OUT)))$$

$$R1 \leftarrow \pi_{First\_Name,Last\_Name,ID\_No}(\sigma_{COMov.ID\_No=Movie.Movie\_ID}(COMov \cup MOVIE))$$

$$\pi_{First\_Name,Last\_Name,MOV.Name}(\sigma_{MOV.Movie\_ID=C.Movie\_ID}(\sigma_{C.Creator\_No=SPACTOR}(R1 \cup CAST)))$$

Another quality query regarding checked out actors is one that finds the most popular actor in the database, which can be seen in the SQL code below:

```
SELECT Name, MAX(holder) as "Movies Checked Out"
FROM (
SELECT ACTOR.First_Name || ' ' || ACTOR.Last_Name AS Name,
Count(Actor.Creator_No) as holder
FROM CHECK_OUT, CAST, ACTOR
WHERE CHECK_OUT.Media_Type = "Movie" AND CHECK_OUT.ID_No =
"CAST".Movie_ID AND "CAST".Creator_No = ACTOR.Creator_No
GROUP BY Actor.Creator_No);
```

Alternatively, this query can be produced with the following relational algebra:

$$R1 \leftarrow \pi_{Actor.First\_Name,Name,\ \rho_{holder}(\mathcal{F}COUNT\ Creator\_No(ACTOR))}$$

$$(\sigma_{Check\_Out.Media\_Type="Movie"}(\sigma_{Check\_Out.ID\_No="CAST".Movie\_ID}$$

$$(\sigma_{"CAST".Creator\_No=ACTOR.Creator\_No}(CHECK\_OUT \cup CAST \cup ACTOR))))$$

$$\pi_{Name,\ \mathcal{F}MAX\ holder\ (R1)}(R1)$$

Finally,

e. Checked Out Album Queries

One quality query regarding checked out albums is one which finds the total number of albums checked out by a single patron, which can be seen in the SQL code below:

```
SELECT First_Name, Last_Name, COUNT(ID_No) as Checked_Out_Albums
FROM Patron, Check_Out
WHERE Patron.Card_Number = SPATRON AND Patron.Card_Number =
Check_Out.Card_Number AND Check_Out.Media_Type = "Album";
```

Where SPATRON is the library card number of the specified patron. Alternatively, this query can be produced with the following relational algebra:

$$R1 \leftarrow \sigma_{Patron.Card\_Numer=SPATRON}(\sigma_{Patron.Card\_Number=Check\_Out.Card\_Number}(\sigma_{Check\_Out.Media\_Type="Album"}(PATRON \cup CHECK\_OUT))))$$

$$_{First\_Name,Last\_Name}\mathcal{F}_{COUNT\ ID\_No}(R1)$$

Another quality query relating to checked out albums is one that provides a list of the albums in the database and associated totals for copies checked out to customers, sorted from the ones that have been checked out the highest amount to the ones checked out the lowest, which can be seen in the SQL code below:

```
SELECT Album.Name, COUNT(Album.Name) as Total_Copies_Checked_Out
FROM (SELECT Album.Name as Name
FROM Check_Out, Album
WHERE Media_Type = 'Album' AND Album_ID = ID_No) as M, Album
WHERE Album.Name = M.Name
GROUP BY Album.Name
ORDER BY TotalCheckout DESC;
```

Alternatively, this query can be produced with the following relational algebra:

$$\rho Result1\ (\sigma Media\_Type\ =\ 'Album'\ AND\ ALBUM \bowtie_{Album\_ID = ID\_No} CHECK\_OUT\ )$$

$$\pi_{Name,\ \Im COUNT\ Name\ (Album)}(\sigma\ ALBUM \bowtie_{Name = Name}\ RESULT1)$$

One final quality query regarding checked out albums is one that provides a list of patron names and email addresses for patrons who have checked out more albums than the average patron, which can be seen in the SQL code below:

```
SELECT Patron.First_Name, Patron.Last_name, Patron.Email
FROM Patron INNER JOIN Check_Out c ON patron.card_number = c.card_number
GROUP BY Last_Name
HAVING COUNT(c.Card_Number) >
(SELECT AVG(result) FROM (SELECT COUNT(Card_Number) as result FROM
CHECK_OUT WHERE CHECK_OUT.Media_Type = 'Album' GROUP BY
Card_Number));
```

Alternatively, this query can be produced with the following relational algebra:

$$\rho Result1\ (\sigma_{\Im COUNT\ Card\_Number\ (CHECK\_OUT)}(\sigma Media\_Type\ =\ 'Album'))$$

$$\rho Result2\ (\sigma_{\Im AVG\ Result1\ (RESULT1)}(Result1)$$

$$\rho Result\ \pi_{First\_Name,\ Last\_Name,\ Email\ (PATRON)}(\sigma PATRON \bowtie_{Card\_Number\ =\ Card\_Number} CHECK\_OUT)$$

$$(Result2)$$

f. Checked Out Artist Queries

One quality sample query relating to artists would be one which finds the most listened to artist in the database, which can be seen in the SQL code below:

```
SELECT Creator, MAX(find)/60 AS 'Hours listened to'
FROM
(SELECT Creator, CASE WHEN seconds NOT NULL THEN seconds * checkedOut
ELSE 1 END as find
```

```
FROM
    (SELECT CASE WHEN ARTIST.Band_Name IS NOT NULL THEN
    ARTIST.Band_Name ELSE CASE WHEN ARTIST.Last_Name IS NOT NULL
    THEN ARTIST.First_Name || ' ' || ARTIST.Last_Name ELSE
    ARTIST.First_Name END END AS Creator, CASE WHEN ARTIST.Band_Name
    IS NULL THEN Count(ALBUM.Name) ELSE
    Count(ALBUM.Name)/Count(ARTIST.Band_Name) END AS checkedOut,
    ALBUM.Length AS seconds
    FROM ARTIST, CHECK_OUT, ARTIST_ALBUM, ALBUM
    WHERE CHECK_OUT.Media_Type = 'Album' AND
    CHECK_OUT.ID_No = ARTIST_ALBUM.Album_ID AND
    ARTIST_ALBUM.Creator_No = ARTIST.Creator_No AND
    CHECK_OUT.ID_No = ALBUM.Album_ID
    GROUP BY Creator)
GROUP BY Creator);
```

Alternatively, this query can be produced with the following relational algebra:

$$R1 \leftarrow (\sigma_{Check\_Out.Media\_Type='Album'}(\sigma_{Check\_Out.ID\_No=Artist\_Album.Album\_ID}(\sigma_{Artist\_Album.Creator\_No=Artist.Creator\_No}$$

$$(\sigma_{Check\_Out.ID\_No=Album.Album\_ID}(ARTIST \cup CHECK\_OUT \cup ARTIST\_ALBUM \cup ALBUM)))))$$

$$\rho_{R2(Creator,checkedOut,seconds)}(\pi_{Artist.Band\_Name,\ \mathcal{F}_{COUNT\ Name}(ALBUM),\ Album.Length}(R1))$$

$$\rho_{R3(Creator,\ find)}(\pi_{Creator,seconds*checkedOut}(R2))$$

$$\rho_{Creator,"Hours\ listened\ to"}(\pi_{Creator,\ \mathcal{F}MAX\ find\ (R3)/60}(R3))$$

One final quality query relating to checked out artists would be one which provides a list of artists who authored the albums checked out by customers who have checked out more albums than the average customer, which can be seen in the SQL code below:

```
SELECT ARTIST.First_Name, ARTIST.Last_Name, ARTIST.Band_Name
FROM CHECK_OUT, ALBUM, ARTIST, ARTIST_ALBUM
WHERE CHECK_OUT.Media_Type = 'Album' AND CHECK_OUT.ID_No =
ALBUM.Album_ID AND ALBUM.Album_ID = ARTIST_ALBUM.Album_ID AND
ARTIST_ALBUM.Creator_No = ARTIST.Creator_No
GROUP BY First_Name, Last_Name, Band_Name
HAVING COUNT(CHECK_OUT.Card_Number) >
(SELECT AVG(result) FROM (SELECT COUNT(Card_Number) as result FROM
CHECK_OUT WHERE CHECK_OUT.Media_Type = 'Album' GROUP BY
Card_Number));
```

Alternatively, this query can be produced with the following relational algebra:

$$Q \leftarrow \pi_{Patron.Card\_Number}(\sigma_{Check\_Out.Media\_Type='Album'} (\sigma_{Check\_Out.Card\_Number=Patron.Card\_Number}(PATRON \cup CHECK\_OUT)))$$

$$A \leftarrow \pi_{Patron.Card\_Number,\ Patron.Card\_Number\ \mathcal{F}COUNT\ Patron.CardNumber\ (Q)} (\sigma_{Q.Card\_Number=Patron.Card\_Number}(PATRON \cup Q))$$

$$AvgAlbum \leftarrow \pi_{\mathcal{F}AVG\ NumAlbums\ (A)}(\sigma_{A.Card\_Number}(A))$$

$$AlbumsCO \leftarrow \pi_{Album.Name}(\sigma_{Check\_Out.Media\_Type='Album'}(\sigma_{ID\_No=Album\_ID}(CHECK\_OUT \cup ALBUM)))$$

$$\pi_{Artist.First\_Name,Artist.Last\_Name,Artist.Band\_Name} (\sigma_{AlbumsCO.Name > AlbumsAvg}(ARTIST \cup AlbumsCO \cup AvgAlbum))$$

3. Insert and Delete Syntax

a. Inserting and Deleting Tracks

To insert a new track into the database, the following syntax must be used:

INSERT INTO TRACK VALUES (A, 'B', 'C', D, E, 'F');

Where A is a unique ID for the track that must not already be in use, B is the title of the track, C is the Genre of the track which may be null, D is the length in seconds which may be null, E is the year created which may be null, and F is the lyrics of the track, and may be null. Take the following example:

INSERT INTO TRACK VALUES (21, 'Frank's Track', 'R&B', 38, 2016, 'The rings all, ring out, burn out, cave in, blackened, to dark out, I"m mixed now, fleshed out. There"s light with, no heat, we cooled out, it"s cool out. Life is, precious. We found out. We found out. We found out');

To delete a track in the database, the following syntax must be used:

DELETE FROM TRACK WHERE Track_ID = X;

Where X is the unique ID of the track to delete. For example, this is the statement to delete the inserted track above:

DELETE FROM TRACK WHERE Track_ID = 21;

b. Inserting and Deleting Albums

To insert a new album into the database, the following syntax must be used:

INSERT INTO ALBUM VALUES (A, 'B', C, D, 'E', F, G);

Where A is a unique ID for the album that must not already be in use, B is the title of the album, C is the Genre of the album which may be null, D is the length in seconds which may be null, and E is the year created which may be null. Take the following example:

INSERT INTO ALBUM VALUES (21, 'Led Zeppelin IV', 4, 19, 'Rock', 2540, 1971);

To delete an album in the database, the following syntax must be used:

DELETE FROM ALBUM WHERE Album_ID = X;

Where X is the unique ID of the movie to delete. For example, this is the statement to delete the inserted album above:

DELETE FROM ALBUM WHERE Album_ID = 21;

c. Inserting and Deleting Movies

To insert a new Movie into the database, the following syntax must be used:

INSERT INTO MOVIE VALUES (A, 'B', C, D, 'E', 'F', G, H);

Where A is a unique ID for the movie that must not already be in use, B is the title of the movie, C is the number of physical copies in the library's circulation, D is the number of electronic copies in the library's circulation, E is the content rating of the movie, F is the Genre of the movie which may be null, G is the length in seconds which may be null, and H is the year created which may be null. Take the following example:

INSERT INTO MOVIE VALUES (21, 'Citizen Kane', 3, 8, 'PG', 'Drama', 7140, 1941);

To delete a movie in the database, the following syntax must be used:

DELETE FROM MOVIE WHERE Movie_ID = X;

Where X is the unique ID of the movie to delete. For example, this is the statement to delete the inserted movie above:

DELETE FROM MOVIE WHERE Movie_ID = 21;

d. Inserting and Deleting Audiobooks

To insert a new Audiobook into the database, the following syntax must be used:

```
INSERT INTO AUDIOBOOK VALUES (A, 'B', C, D, 'E', F, G);
```

Where A is a unique ID for the audiobook that must not already be in use, B is the title of the audiobook, C is the number of physical copies in the library's circulation, D is the number of electronic copies in the library's circulation, E is the Genre of the audiobook which may be null, F is the length in seconds which may be null, and G is the year created which may be null. Take the following example:

```
INSERT INTO AUDIOBOOK VALUES (21, 'Don Quixote', 6, 10, 'Novel', 1000, 1605);
```

To delete an audiobook in the database, the following syntax must be used:

```
DELETE FROM AUDIOBOOK WHERE Audiobook_ID = X;
```

Where X is the unique ID of the audiobook to delete. For example, this is the statement to delete the inserted audiobook above:

```
DELETE FROM AUDIOBOOK WHERE Audiobook_ID = 21;
```

e. Inserting and Deleting Artists

To insert a new Artist into the database, the following syntax must be used:

```
INSERT INTO ARTIST VALUES (A, 'B', 'C', 'D');
```

Where A is a unique ID for the artist that must not already be in use, B is the first name of the artist, C is the last name of the artist which may be null, and D is their band name which may be null. Take the following example:

```
INSERT INTO ARTIST VALUES (21, 'Cher', NULL, NULL);
```

To delete an artist in the database, the following syntax must be used:

```
DELETE FROM ARTIST WHERE Creator_No = X;
```

Where X is the unique ID of the artist to delete. For example, this is the statement to delete the inserted artist above:

```
DELETE FROM ARTIST WHERE Creator_No = 21;
```

f. Inserting and Deleting Patrons

To insert a new patron into the database, the following syntax must be used:

```
INSERT INTO PATRON VALUES ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H');
```

Where A is a 9 digit library card number that must not be in use, B is the first name of the patron, C is the last name of the patron, D is the email of the patron, E is the street the address of the patrons home which may be null, F is the home city of the patron which may be null, G is the home state of the patron which may be null, and H is the home zip code of the patron. Take the following example:

```
INSERT INTO PATRON VALUES ('058427636', 'Leon', 'Madrid', 'madrid.1@osu.edu',
null, null, null, null);
```

To delete a patron in the database, the following syntax must be used:

```
DELETE FROM PATRON WHERE Card_Number = X;
```

Where X is the unique card number of the patron to delete. For example, this is the statement to delete the inserted patron above:

```
DELETE FROM PATRON WHERE Card_Number = '058427636';
```