Ideal Design, Issues Faced and Subsequent Implementation

We will start off with discussing what we had hoped to accomplish initially when we were assigned this data, then discuss what measures we took to implement this, or issues we faced that prevented us from implementing this the way we had envisioned.

Checkout Records and Checking Out/In a Book

Our vision with these records was to use them to track the availability of books across the library. This way if a user wants to check out a book in the current day we can easily execute it if the book was indeed available. We would simply iterate over the entries with the book's BibNumber from the records, and then decrement the item count by 1 every time it was checked out. That way we can determine how many copies we had of this book at the end of the month.

There are a few major issues we faced with this that practically forced us to change the role these records would play in our implementation. For starters, to check the current item count, we would have to cross reference that with the library's inventory - which is great in theory. The issue is that the inventory was stocked at only 2 dates; September and October and ONLY on the first of those months, while the checkout records end abruptly on October 18th. This poses some uncertainty in the execution, since the checkout records end in the middle of the month. We only know at the start of the month that we have a certain number of copies; but then what happens at the start of the next month (November) when the records end on the 18th of October? What happens when someone checks it out on the 18th; do we just mark the book as unavailable? How do we know it was ever returned?

To illustrate another issue, consider the case where the records state that Book A was checked out on August 10th, we can not check the item count of this book in the inventory since we only know the number of copies available on October 1st; with no clue how many existed in August. Another issue we had was the fact that there were several entries per BibNumber for each (location, type, collection) tuple. This means that two Books can have the exact same BibNumber, Type, and Collection - with only the Location to separate the two entries. The issue is that the records do not show us the location this book was checked out at. What this means is that every time someone checks out a book, we would have to query the 2 million+ CSV, choose a location at random (since it was never specified, meaning we can potentially have multiple of the same entry), and then decrement its count by 1. The fact that location was absent also restricts the client application's potential; Users should be able to check in and checkout a book to/form a location they please. Not tracking the location in the records means that we don't give the User that choice.

What we realized was that we practically can not consider the checkouts before October as relevant - we will just assume that everything in the October inventory is available and check in/out based on that item's count specifically; the checkout records for the earlier part of the year will be kept for data mining or for querying purposes. Any time a User wants to check out a book, we will simply ask for the itemType, itemCollection, branch and BibNumber of the book; which gives us a singular entry in the inventory and allows us to easily change the itemCount as we please.

You are probably thinking “how would a User know all this information about a book?” Not to worry; we have a function that takes ANY part of a title and outputs a CSV with all the matching titles and their subsequent information, so you will always have a way of knowing the identifying information of any book in the library. For any new checkouts, we will store that in a table containing the libraryCard of the User who checked it out, and the location it was checked out from in addition to the aforementioned identifiers. Checking in a book just simply edits the check in date, and updates the status of the book.

To summarize, there are currently 2 versions of the checkout records - new and old. The old ones are not too helpful in the sense that they give us only book-specific information; regardless of the branch the book was checked out at and the User that checked it out. There are limitations to this, since a Branch can not see which books were checked out from its branch, and a User can not see which books they checked out; nor can an Admin track which books are overdue for a User, etc. As a result we have 1 table for old records; just for the sake of book keeping and preserving records; and then we have a new table for all Users after October 1st, with more comprehensive data stored in this table.

ISBN and GoodReads Ratings

This is by far the most frustrating aspect of this data; one that was most detrimental to our ideal execution of this project. In theory, the ISBN of a book in 1 database should be able to match it to another database such as GoodReads and return the rating etc. Except not exactly; since for some reason nearly every book in Seattle’s library has more than 1 ISBN. Which of these do we choose? Do we take the first ISBN or try and loop through all ISBNS to check for a match in GoodReads? Do we discard the ISBN’s that do not match? You can imagine how ridiculously inefficient it is to iterate over 1 million+ books and cross reference them with the GoodReads database that’s nearly 4 times larger. Another issue is the fact that there are just so many different ISBN for the same book title, and many others do not even have an ISBN, and many other just simply do not exist in the GoodReads data set to the point where it is useless if you want to use it as an identifier across both datasets. What we opted to do instead was just drop all null ISBN from GoodReads, and make the (Title, ISBN) tuple a primary key in GoodReads - to allow us to have entries that have duplicate Titles, just different ISBNs. We then tracked every single BibNumber with multiple ISBN, and wrote a script that creates a new entry for each (BibNumber, ISBN) pair - and then we inner join on the ISBN column with GoodReads. The result is a Foriegn Key that relates the Book BibNumber to this Temp table’s BibNumber; and a Foriegn Key from Temp’s BibNumber to GoodReads ISBN. This allows us to relate the Books in our Seattle Library to the GoodReads Reviews.

Titles in Seattle/GoodReads and GoodReads Ratings

Another idea we had was to match the titles across both datasets; but that proved futile. The way titles are formatted in Seattle right now is as follows: “Title of Book / authors and all contributors”, in contrast to GoodReads lack of consistency. Take these 2 entries for example:

```
110938,The Awakening and Other Stories,0613708458,Kate
Chopin,3.86,2000,14,11,Turtleback,5:239,4:255,3:196,2:57,1:13,total:760,2,,375,,
```

110939,The Awakening & Selected Stories (Modern Library),0679424695,Kate Chopin,3.92,2000,1,11,Modern Library,5:3793,4:3276,3:2139,2:743,1:342,total:10293,3,,354,,

If we filter out the database to remove the brackets, then we get 2 books with the exam title - which of these ratings do we choose? GoodReads has multiple titles of the same book but added extra identifiers to the point where it is practically impossible to get just single unique entries in the dataset.

Filtering Data

Checkout Records

There were millions of entries in the 2017 csv alone; let alone the ones from 2005-2016. In fact, there are nearly 5 million just from January to September; which is just pointless data to store since we had no Data Mining component. We opted to just use the records from August to October, which has 500K entries on its (as mentioned in piazza) and allow the User to query those for Data Mining if need be. We took all the attributes from these records.

Seattle Inventory

We chose to only keep the latest version of the inventory, which are the entries with the report date “01/10/2017”; anything else was dropped since we need the most recent inventory to ensure we are executing check in/out queries correctly. All entries with empty BibNumbers/ItemType/Title/Collection/Location were also dropped. Each book has a branch, a type and a collection. The issue with these values is that they are basic string values; what this means is that any time we want to change the identifier of collection/branch/type, we will have to query the entire dataset and implement that change. Instead, what we did was map an integer to each identifier and store that into a table (id, value) to prevent such a case from happening.

GoodReads

We took 2 million + entries from GoodReads, dropping only anything that doesn’t have an ISBN. We also noticed some csv’s had more/less columns then the others, so we normalized that data by adding those rows.

Summary Of Implementation:

Rating and Reviewing within a Library

We idealized that Users in a library should be able to rate and review Books they have in their own library - to inspire a sense of community amongst the Users. In the end, we implemented this in a way such that there is a foriegn key from User to Review, and a Foriegn key from Review to Book; so that all things are connected; a User can Review a Book, and Check in/out a book. This implementation allows the User to edit their old reviews as well; and now multiple Users can rate and review a Book as well.

Finding Ratings from GoodReads

Our final implementation just asks for the title the User is curious about, and then we will “select where” using regexp on the Title in the Seattle Library, and then finally search for that ISBN in

GoodReads. This means that all a user really needs is just the Title of a Book; which is a realistic scenario.

Checking In and Checking Out a Book:
When a User checks out a Book, we create an entry in the Book table (BibNumber, CheckOutDate, CheckInDate, DueDate, Status, Type, Collection, Branch) where the status is 0 if the User has just checked it out. The due date is just 2 weeks ahead of the check out date. When a User returns the book, we check if the return date was > 2 weeks from the check in date; if so then we mark it as a 2 (overdue) or a 1(on time) A User can also not checkout the same book more than once in the same 24 hours.

Searching for Book
As it stands, we have allowed the User to search for a book by Title, or search for it by either ItemType, ItemCollection or both. The output of any of these returns a CSV with the information (Bib, Author, Title, etc) into a CSV, to avoid congesting the client console. We also have non-client facing functions, such as searching for the ISBN of a book by its BibNumber to help us with our queries.

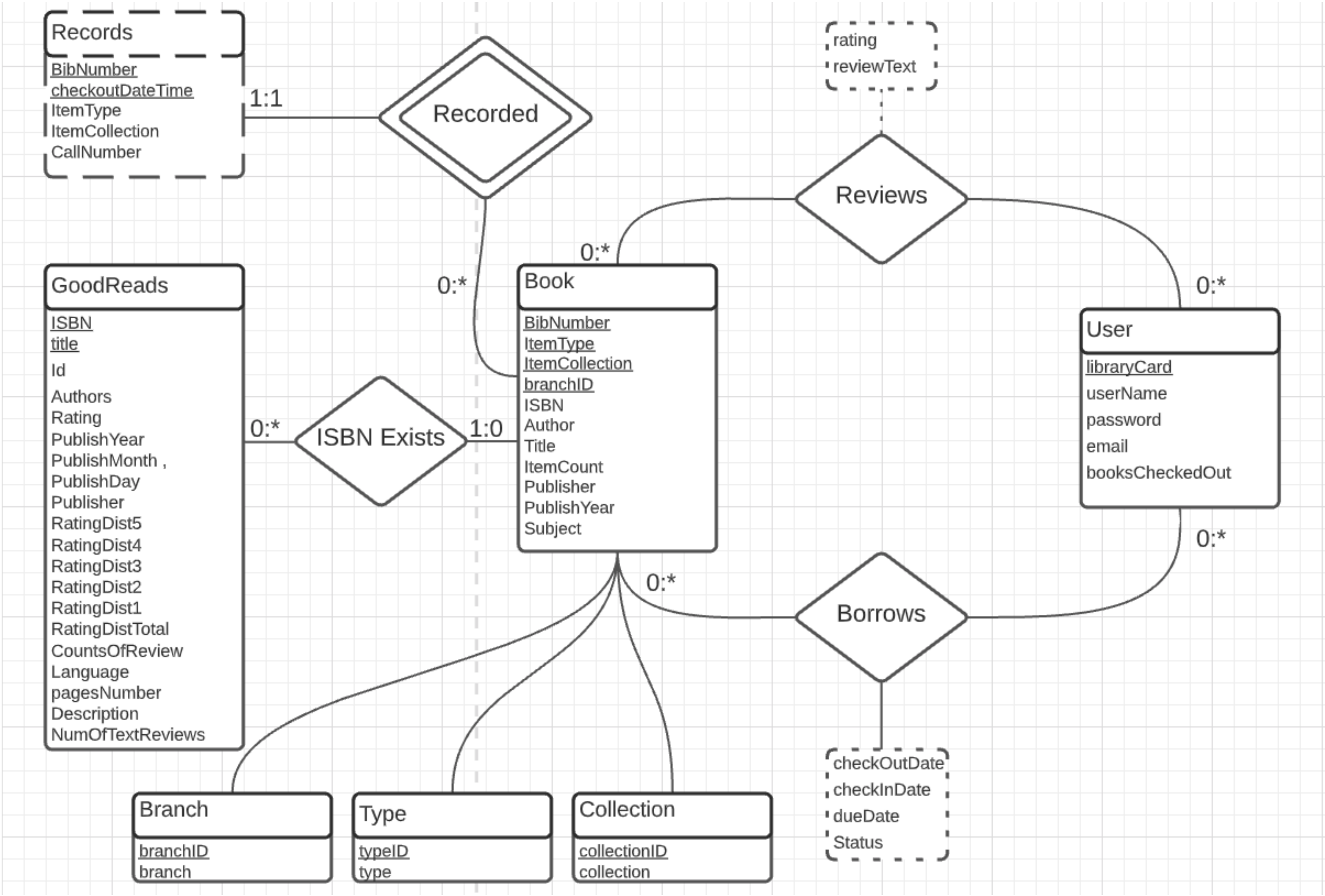
Checkout Records
The old ones are not too helpful in the sense that they give us only book-specific information; regardless of the branch the book was checked out at and the User that checked it out. There are limitations to this, since a Branch can not see which books were checked out from its branch, and a User can not see which books they checked out; nor can an Admin track which books are overdue for a User, etc. As a result we have 1 table for old records; just for the sake of book keeping and preserving records; and then we have a new table for all Users after October 1st, with more comprehensive data stored in this table. A User can also see how many active checkouts they have currently.

Admin
We have allowed administrators a host of functionality; adding/removing a Book/User/Review, as well as seeing which/how many books a User has checked out.

Functionality

1. Create a User
2. Add and Remove a Book
3. Add, Edit and Remove Seattle Library Reviews
4. Find GoodReads Rating for Book
5. Find Book Info by Title
6. Find Books based on Category
7. Check In and Checkout a Book
8. Find all Records for a Books by BibNumber
9. Find all Records for User by libraryCard
10. Find Number of Books Currently Checked out by User

ER Diagram



The diagram above illustrates the relationships in our project. A User can Borrow as many books as he wants (0 : *) and a book can be Borrowed by as many Users as exists, or even 0 Users. Likewise, a User can Review a book several times, or however many books he wants, while a Book can have 0 reviews about it, or as many reviews as there are in the database. Each book has a Branch, Type and Collection as specified. Lastly, if the ISBN exists in the GoodReads database, the Book will have a singular entry for it, while it may also not exist (0:1). At the same time, GoodReads data can have 0 books from Seattle Library in its database, or as many Books as exists in the Library. Records is a weak entity relying on Book, represented by the relationship Recorded. It represented grandfathered data from the original Seattle Library that seemed unrealistic to process and was not helpful for a user’s ability to check out books. However, we kept the records for general bookkeeping.

DataBase Design

Each book has a branch, a type and a collection. The issue with these values is that they are basic string values; what this means is that any time we want to change the identifier of collection/branch/type, we will have to query the entire dataset and implement that change. Instead, what we did was map an integer to each identifier and store that into a table (id, value)to prevent such a case from happening.

```
create table Branches (
   branchID INTEGER PRIMARY KEY,
   branch VARCHAR(5) NOT NULL
);

create table ItemCollection(
   collectionID INTEGER PRIMARY KEY,
   collection TEXT NOT NULL
);

create table ItemType(
   typeID INTEGER PRIMARY KEY,
   type TEXT NOT NULL
);
```

The User table is quite simple; each User has a password, email, User name you would expect of any client that is relational. In addition each User has alibraryCard since that is what we will use to track the checkouts and check ins. The last attribute the User has is the booksChecked out, which is just a count of all active checkouts.

```
create table User (
   userName TEXT NOT NULL,
   password TEXT NOT NULL,
   email TEXT NOT NULL,
   libraryCard INTEGER PRIMARY KEY,
   booksCheckedOut INTEGER NOT NULL
);
```

Each Book in our database will have a 4-tuple as its unique identifier (BibNumber, branchId, ItemCollection, ItemType) Additional attributes have been added such as the Author and year the Book was published, or what subject it is related to. The most important of these additional attributes is the ItemCount, which allows us to implement our logic for checking in/out a Book. Given that a book has a Branch, Type and Collection, each of these is a Foreign Key to their respective tables.

```
create table Book (
   BibNumber INTEGER NOT NULL,
   Title TEXT NOT NULL,
   Author TEXT,
   ISBN TEXT NOT NULL,
   Publisher TEXT,
   PublishYear TEXT,
   ItemType INTEGER NOT NULL,
   Subject TEXT,
   ItemCollection INTEGER NOT NULL,
   branchID INTEGER NOT NULL,
   ItemCount INTEGER NOT NULL,
   PRIMARY KEY (BibNumber, branchID, ItemCollection, ItemType),
   FOREIGN KEY (branchID) references Branches(branchID),
   FOREIGN KEY (ItemType) references ItemType(typeID),
   FOREIGN KEY (ItemCollection) references ItemCollection(collectionID)
);
```

We mentioned earlier that we allow a User to Review a Book. This here is the table that will embody this dynamic. The review is for a specific Book, by a specific User which is why it must contain the BibNumber of a Book and the libraryCard of the User; this tuple will be the primary key. Each review has a rating (decimal) and a review of how the User felt, ex: (“great book, loved the plot!”). The BibNumber is a foriegn key to Book, while libraryCard is a Forigen Key to User

```
create table LibraryReview (
   BibNumber INTEGER NOT NULL,
   rating DECIMAL(2,1) NOT NULL,
   reviewText TEXT NOT NULL,
   libraryCard INTEGER,
   PRIMARY KEY (BibNumber, libraryCard),
   FOREIGN KEY (libraryCard) references User(libraryCard)
   FOREIGN KEY (BibNumber references Book(BibNumber)
);
```

Since this is a Table that consists of Books that have been checked out, it makes sense for the BibNumber to be included in the Primary Key; in addition to the checkoutDateTime. Each entry has a Type and Collection, so those Foreign Keys have also been added. The only additional attribute is just the CallNumber.

```
create table Records(
   BibNumber INTEGER NOT NULL,
   ItemType INTEGER,
   ItemCollection INTEGER,
   CallNumber TEXT,
   CheckoutDateTime datetime NOT NULL,
   PRIMARY KEY (BibNumber, CheckoutDateTime),
   FOREIGN KEY (ItemType) references ItemType(typeID),
   FOREIGN KEY (ItemCollection) references ItemCollection(collectionID)
);
```

Each Checkout must have 3 dates (DueDate, CheckinDate, CheckOutDate). Since these are also Books that are being checked out, we also need the BibNumber, Collection, Type and branchID (and those will be foriegn keys to a Book) the Primary Key will be the combination of that 4-tuple and the day it was checked out, as well as the library card since multiple Users can checkout the same Book several times - even in the same day (if the Users are different) The status as mentioned earlier is just to signify if the book is currently checked out (0), returned on time (1), or overdue (2)

```
create table CheckOuts(
  libraryCard INTEGER NOT NULL,
  BibNumber INTEGER NOT NULL,
  checkoutDate datetime NOT NULL,
  checkInDate datetime,
  dueDate datetime NOT NULL,
  branchID INTEGER NOT NULL,
  status INTEGER NOT NULL,
  itemType INTEGER NOT NULL,
  itemCollection INTEGER NOT NULL,
  primary key (libraryCard, BibNumber, checkoutDate, branchID, itemType, itemCollection),
  FOREIGN key (libraryCard) references User(libraryCard),
  FOREIGN key (branchID) references Branches(branchID),
  FOREIGN key (BibNumber, branchID, itemType, itemCollection) references Book(BibNumber, branchID,
ItemCollection, ItemType)
);
```

In addition to these tables, we also have to filter our data:

The GoodReads table simply has all the data that we took from the original data set (Rating, Publisher-related information, ID, Description, pagesCount) as well as the Primary Keys - ISBN and Title.

```
create table newGoodReads (
   Id TEXT NOT NULL,
   title VARCHAR(500),
   ISBN VARCHAR(20),
   Authors TEXT NOT NULL,
   Rating TEXT NOT NULL,
   PublishYear TEXT,
   PublishMonth TEXT,
   PublishDay TEXT,
   Publisher TEXT,
   RatingDist5 TEXT NOT NULL,
   RatingDist4 TEXT NOT NULL,
   RatingDist3 TEXT NOT NULL,
   RatingDist2 TEXT NOT NULL,
   RatingDist1 TEXT NOT NULL,
   RatingDistTotal TEXT NOT NULL,
   CountsOfReview TEXT NOT NULL,
   Language TEXT NOT NULL,
   pagesNumber TEXT,
   Description TEXT,
   NumOfTextReviews TEXT,
   primary key (ISBN)
);
```

We created an Index on the Primary Key pair as well, to improve the query run times when searching for a rating.

```
CREATE INDEX indexTitle
ON newGoodReads (title, ISBN);
```

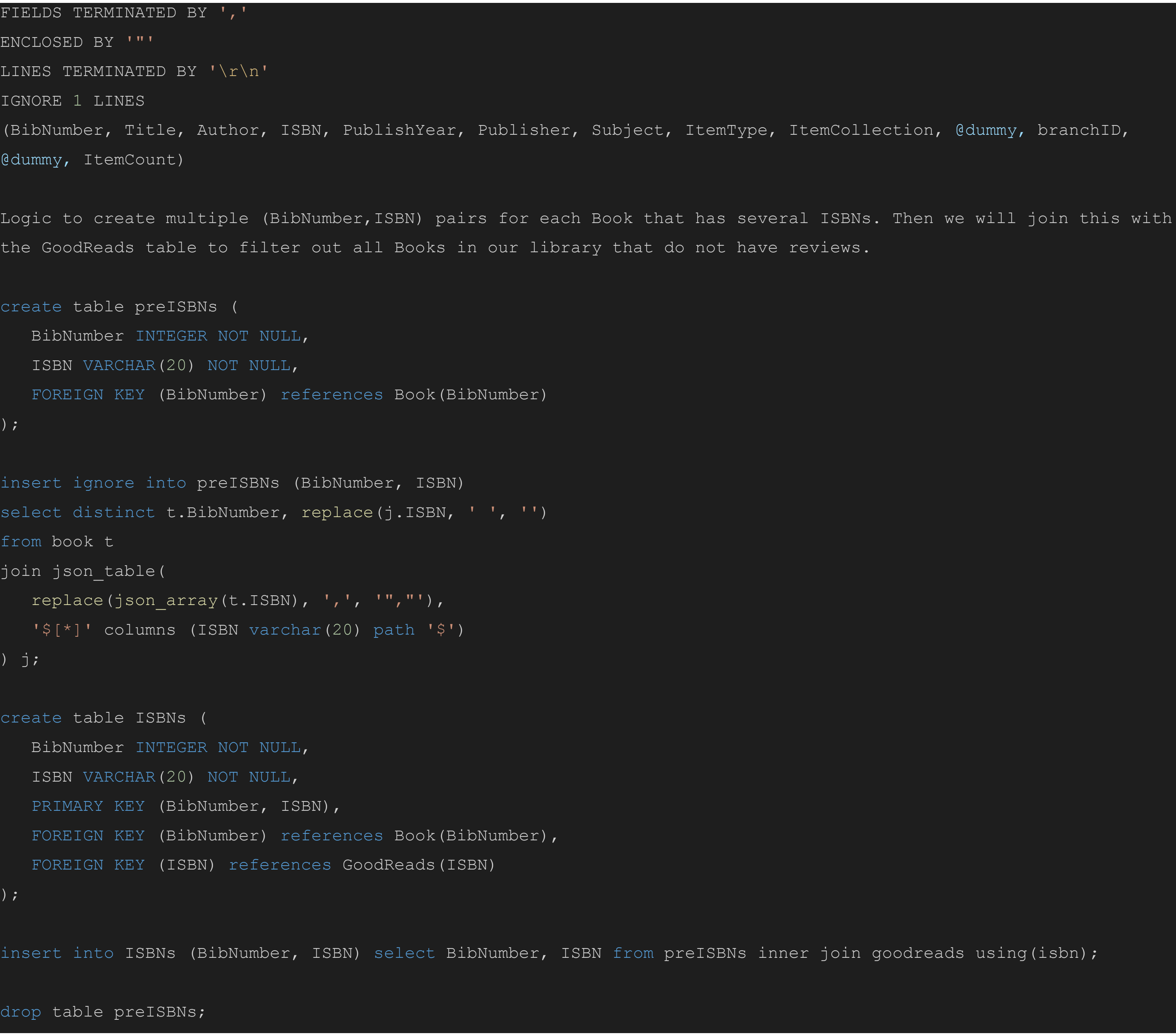
Basic loads to load the data from the CSV’s

```
LOAD DATA LOCAL INFILE 'C:/Users/russe/Documents/ECE356/Project/books1500003.csv' INTO TABLE GoodReads
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES;

LOAD DATA LOCAL INFILE 'C:/Users/russe/Documents/ECE356/Project/books1500003.csv' INTO TABLE GoodReads
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES;

LOAD DATA LOCAL INFILE 'C:/Users/russe/Documents/ECE356/Project/books1500003.csv' INTO TABLE GoodReads
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES;

LOAD DATA LOCAL INFILE 'C:/Users/russe/Documents/ECE356/Project/seattle.csv' INTO TABLE Book
```



```
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\r\n'
IGNORE 1 LINES
(BibNumber, Title, Author, ISBN, PublishYear, Publisher, Subject, ItemType, ItemCollection, @dummy, branchID,
@dummy, ItemCount)
```

Logic to create multiple (BibNumber,ISBN) pairs for each Book that has several ISBNs. Then we will join this with the GoodReads table to filter out all Books in our library that do not have reviews.

```
create table preISBNs (
   BibNumber INTEGER NOT NULL,
   ISBN VARCHAR(20) NOT NULL,
   FOREIGN KEY (BibNumber) references Book(BibNumber)
);

insert ignore into preISBNs (BibNumber, ISBN)
select distinct t.BibNumber, replace(j.ISBN, ' ', '')
from book t
join json_table(
   replace(json_array(t.ISBN), ',', '","'),
   '$[*]' columns (ISBN varchar(20) path '$')
) j;

create table ISBNs (
   BibNumber INTEGER NOT NULL,
   ISBN VARCHAR(20) NOT NULL,
   PRIMARY KEY (BibNumber, ISBN),
   FOREIGN KEY (BibNumber) references Book(BibNumber),
   FOREIGN KEY (ISBN) references GoodReads(ISBN)
);

insert into ISBNs (BibNumber, ISBN) select BibNumber, ISBN from preISBNs inner join goodreads using(isbn);

drop table preISBNs;
```

Testing Flow

Create User

Creating a user that does not already exist in the database:
Input “user” → input “add” → (username, password, userName) → “User has been added. Your library card number is: "+str(libraryCard)”
Creating a user that already exists in the database:

```
{"Error" : "Unable to execute query","Status": False}
```

Get Number Of Existing Checkouts:

Input(“user”) → input (“check”) → (libraryCard) → `"You have "+str(numBooks)+" book(s) checked out currently."`
If the libraryCard is invalid:

```
{"Error" : "Unable to execute query","Status": False}
```

Add/Remove Book:

Input “book” → “add” → (BibNumber, Title, Author, ISBN, Publisher, PublishYear, ItemType, Subject, ItemCollection, branchID, ItemCount) → `“Book has been added”`
If book with same BibNumber, ItemType, ItemCollection, branchID already exists:

```
“This book already exists. Please delete the record if you wish to add a new one.”
```

If args are incorrect:

```
“Incorrect/missing information. Please ensure the data type is accurate."
```

Else:
Query failed

Input “book” → “remove” → (BibNumber, Title, Author, ISBN, Publisher, PublishYear, ItemType, Subject, ItemCollection, branchID, ItemCount) → “Book has been deleted”
If password incorrect:

```
“Password is incorrect, please try again later.”
```

If no book with BibNumber, ItemType, ItemCollection, branchID exists:

```
“This book does not exist or has already been deleted”
```

If args are incorrect:

```
“Incorrect/missing information. Please ensure the data type is accurate."
```

Else:
Query failed

Checkout Book:

Input “checkout → (libraryCard, bibNumber, itemType, itemCollection, branchID) → “{book title} checked out”
Checking out an item with bookCount == 0:
“This book is already checked out”

Else:
Query failed

Check in Book
Checking in a book that is already checked out:
Input “checkin” → (libraryCard, bibNumber, itemType, itemCollection, branchID) → `"The results are in the result.csv file."`
Checking in a book that is not checked out or has been checked out by the same user in the last 24 hours:
Input “checkin” → (libraryCard, bibNumber, itemType, itemCollection, branchID) → `"The book was checked in."`
Checking in a book that was not checked out by the user →
Input “checkin” → (libraryCard, bibNumber, itemType, itemCollection, branchID) → `{"Error" : "Unable to execute query","Status": False}`

Add/Edit a Review of a Book:
Adding a review that does not already exist, or editing a review that already exists for this user:
Input (“review”) →Input (“add” or “edit”) → (BibNumber, Rating, ReviewText, LibraryCard, Password) → `“Review has been added/updated”`
If the book does not exist:
`“This book does not exist”`
If the password is incorrect:
`"Password or User is incorrect"`
If the review does not exist:
`"You have no review to edit."`
If query fails:
`"Review was not added/updated. Ensure you have entered the correct book information or do not already have a review for this book."`

Removing review:
Input (“review”) →Input (“delete”) → (BibNumber, LibraryCard, Password) → `“Review has been deleted”`
If query fails:
`"Review was not removed. Ensure you have entered the correct book information or that you have a review for this book."`
If the password is incorrect:
`"Password or User is incorrect"`
If the book does not exist:
`“This book does not exist”`
If the review does not exist:
`"You have no review to delete."`

View Review
Input (“review”) →Input (“view”) → (BibNumber) →
`“Reviews:`

```
{BibNumber, Rating, ReviewText}
Average rating: {rating/count} out of {count} reviews"
```

If the book does not exist or the query fails:

```
{"Error" : "Unable to execute query","Status": False}
```

Search for Items:

Input (“search”) → Input(“title”) → (searchString)→`"The results are in the result.csv file."`

Else:

Query failed

Get BibNumber from result.csv

Input(“search”) → Input(“goodreads”) → (BibNumber) → “{book information}”

If BibNumber does not exist in ISBNs table (no ISBN matching to goodreads):

```
{"Error" : "Unable to execute query","Status": False}
```

Else:

Query failed

Check csv to verify results

Input (“search”) → Input(“bibnumber”) → (BibNumber)→`"The results are in the result.csv file."`

Else:

Query failed

Check csv to verify results

Input(“search”) → Input(“category”) → Input(“2”) → “{itemType, itemCollection}”→`"The results are in the result.csv file."`

Else:

Query failed

Check csv to verify results

Input (“search”) → Input(“records”) → (BibNumber)→`"The results are in the records.csv file."`

Else:

Query failed

Check csv to verify results

User Checkout Records :

Input(“Checkouts”) → (libraryCard) → `The results are in the result.csv file.`

If the libraryCard is invalid:

```
{"Error" : "Unable to execute query","Status": False}
```

Possible Improvements

There are possible organizational improvements we could have made to be more “correct” in our ER approach. For example, we could have stored each Publisher, Publisher Day/Month/Year entry for a Book that exists in Good reads into a separate table (the same way we did with the Seattle Library with Branch, Type and Collection) While this will not improve the run time of our queries, it is just cleaner to have each chunk of data that relevant to itself given a specific context to be its own data. The same can also be said for the Rating, RatingDist 5/4/3/2/1 information; as well as the Publisher and Publisher Year in the Seattle Library.

It is important to keep in mind that unlike(Branch, Type, Collection), this information was not subject to change in any way - a Book with a certain BibNumber will never change its Publish Year etc; but a Book Title may have different ISBNs as well as different Publishers etc; so in that sense it would be more professional to separate the data. These changes would help from an organizational perspective as well as help us search the Database more objectively. We could also have added more comprehensive - for example, allowing the User to search for Books by a given Author; instead of just searching by the Title, Bib, Type or the Collection.