Database Writeup
Danny Hong
Arthur Skok

Library Management System

Abstract:

Our database project aims to emulate a library system for book enthusiasts in an online format. There is a catalog that exhibits all the books in the collection, whether they are borrowed or not, and users would be able to get a book issued or returned to them via an administrator user. Administrators can also add new users and books to their respective tables, and users are able to keep track of their current reservations and accumulated late fees.

General Changes:

Some changes to the proposal include the removal (at least for now) of the capability to see what other books are borrowed and for how long. The latter was sidelined as arithmetic operations on datetime values proved hard (spent several hours trying to figure it out to no avail) to implement for lateness functionality (the books under the bookloan table would have a scheduler that checks every so often what the difference is between the current date time and the date time stored in the table and seeing if it was greater than a certain duration. While the scheduler was inevitably included to update late fees for a given book loan, the way we implemented determining whether or not a book was late was instead managed directly by the administrator, marking loans as late manually through a button), and instead users are only able to see when they took out the book initially and whether or not the administrator has deemed it late.

In the same vein, history of a user's previous transactions was scrapped as of the writing of this writeup, though it may be included eventually. Instead the transactions manifest in the users total accrued late fees, though if the admin deems that they returned the book on time, there will essentially be no record of those returned books stored per user (issue with borrowing the same book over and over again, bit of an edge case admittedly).

Books are able to be issued and returned, but in our current iteration only by the administrator. The logic is that the administrator knows whether or not the user borrowed a *physical* copy of the book and they log that transaction into the database themselves; the same for users returning their physical copy. (We thought it didn't make sense that a user be able to reserve a physical book via the database and have their start day of their loan be recorded when they may not even have access to it yet, the same with a user potentially lying about when they returned a book and evading potential late fees.)

Framework:

Our project utilizes a Flask framework with the backend code written in Python and the frontend code written in HTML. We decided to use Flask because it not only provides us with the necessary tools and functions for creating a web application, but is also very simple to understand and extremely organized in terms of how it connects together our backend to our frontend. Our database is created by implementing an ORM (Object Relational Mapping) with SQLAlchemy which is then connected to a SQLite database. As we learned in class, ORM's are helpful since they allow us to write queries and edit our database by writing code using a backend programming language. The SQLAlchemy library, which is an open-source SQL toolkit, was leveraged in our backend order to map our database to our code. The database itself is created using SQLite which emulates a (RDBMS) relational database management system. Finally, we installed the DB Browser for SQLite application so that we can make edits to our database schema and to also run a few SQL queries on it.

Backend Structure:

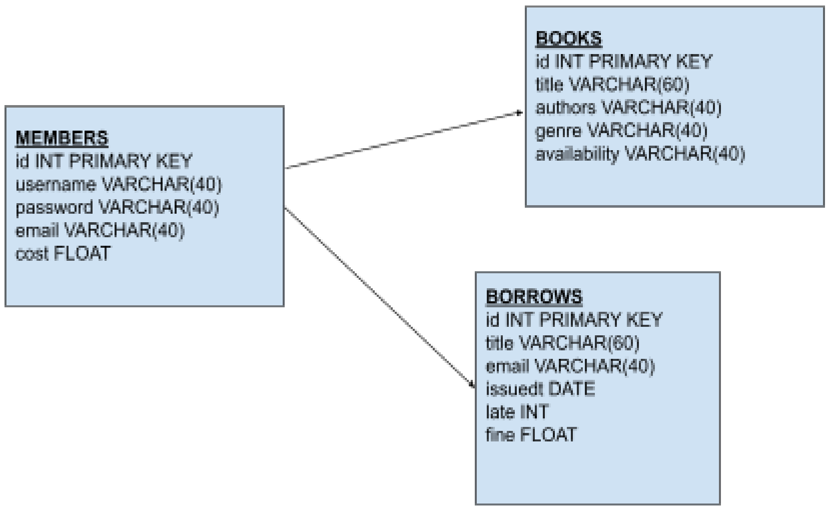
Our backend structure consists of 5 files: *models.py, forms.py, views.py, createdb.py,* and *app.py*. The *models.py* file consists of the 3 tables: Members, Books, and Borrows which are used to establish the ORM to the *library.db* database through the SQLAlchemy library. The *forms.py* contains all the necessary forms that retrieve values of form elements posted to HTTP (with a form using the 'POST' method) which are required by the functions in *views.py*. The *views.py* file contains all the functions that our library management web application will be capable of performing, and also contains the routes to the frontend html files. The *createdb.py* file simply initializes and configures the database to the program, and finally, *app.py* configures all the dependent backend files together and runs the program. Additionally, though it may not be ideal, our *app.py* file also contains a function in which we implement a scheduler that will help us make subsequent queries to the database in order to increment the late fines of a user who had returned a book late on a base rate of $0.01 per second late.

Schema Changes:

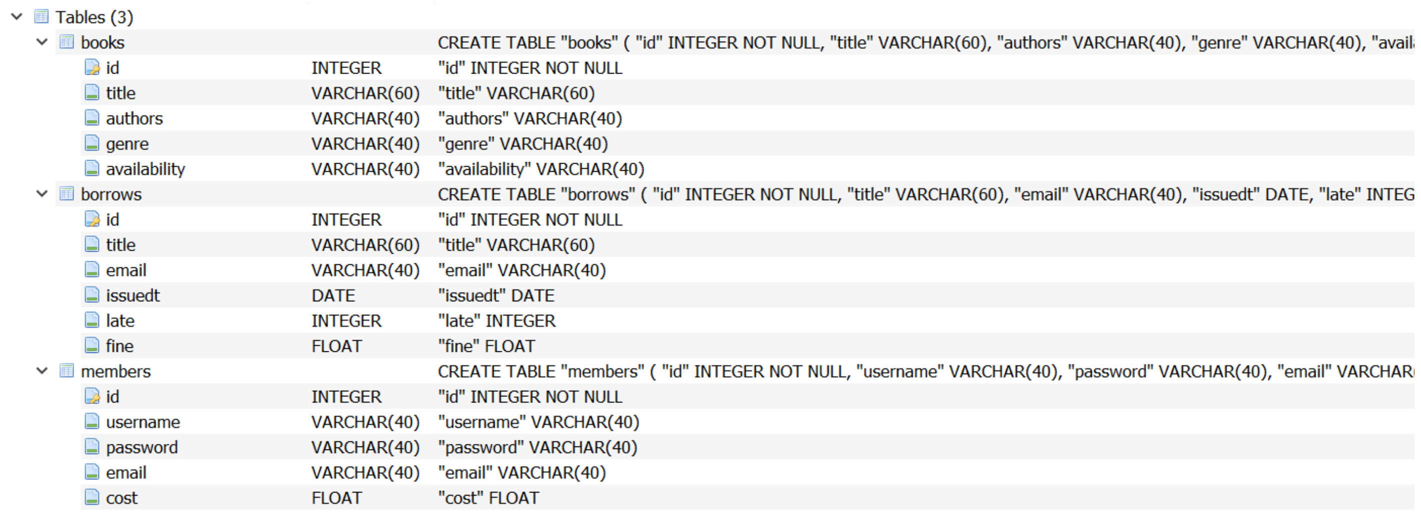
Shown below is our updated schema ER diagram. Names have been shortened to just usernames (first names) for now, small change. Rental price removed as the functionality of purchasing a book has not been implemented yet, essentially replaced by late fees only. Another difference is that we have decided to merge our Admin table with the Members table since the Admin attributes were exactly the same as the Member attributes. Since there will be certain functions that only an Admin can perform, a function called *checkmember()* will be written and implemented that will check for if a particular member using our library program is an Admin or a Non-Admin, which will determine what functions on the web page they will have access to.

Returns table not yet implemented, we thought it was important to first implement books, users, and current transactions alongside late fees. Finally, we had also forgotten to include a registration_date parameter for Members, but implementing this in our database would be very simple since it'll just result in our Members Table having one extra attribute that needs to be accounted for when running any Member related functions in our program.

ER Diagram (Updated):



Database Tables in SQLite:



Tables (3)		
books		CREATE TABLE "books" ("id" INTEGER NOT NULL, "title" VARCHAR(60), "authors" VARCHAR(40), "genre" VARCHAR(40), "avail
id	INTEGER	"id" INTEGER NOT NULL
title	VARCHAR(60)	"title" VARCHAR(60)
authors	VARCHAR(40)	"authors" VARCHAR(40)
genre	VARCHAR(40)	"genre" VARCHAR(40)
availability	VARCHAR(40)	"availability" VARCHAR(40)
borrows		CREATE TABLE "borrows" ("id" INTEGER NOT NULL, "title" VARCHAR(60), "email" VARCHAR(40), "issuedt" DATE, "late" INTEG
id	INTEGER	"id" INTEGER NOT NULL
title	VARCHAR(60)	"title" VARCHAR(60)
email	VARCHAR(40)	"email" VARCHAR(40)
issuedt	DATE	"issuedt" DATE
late	INTEGER	"late" INTEGER
fine	FLOAT	"fine" FLOAT
members		CREATE TABLE "members" ("id" INTEGER NOT NULL, "username" VARCHAR(40), "password" VARCHAR(40), "email" VARCHAR
id	INTEGER	"id" INTEGER NOT NULL
username	VARCHAR(40)	"username" VARCHAR(40)
password	VARCHAR(40)	"password" VARCHAR(40)
email	VARCHAR(40)	"email" VARCHAR(40)
cost	FLOAT	"cost" FLOAT

Data Description:

We used a data dump (.csv file) from kaggle called books.csv. The source link is shown below. The original dataset contained 6610 book entries. In our *filter.py* script, data for columns that weren't necessary for our model (such as publisher information and book rating) were cleaned out of the dataset and missing fields of required attributes were either filled with a n/a or were dropped rows from the dataset using dropna function in pandas. As a result, we decided to keep and use the title/subtitle, genre, and author attributes for our project. After cleaning out entries that contained missing fields and getting rid of unnecessary attribute columns, the data size shrunk from 6610 to 6640. The cleaned dataset was then stored and written to books1.csv. We then used this filtered data to populate the books table of our database in our *insert.py* script.

(Kaggle Link: **https://www.kaggle.com/dylanjcastillo/7k-books-with-metadata**)

Functionality Demonstration:

To run our program, first go to the Github link listed below and clone the repository. It could be helpful to create a folder and store the cloned files in that folder. Make sure that all the libraries/packages listed in the *requirements.txt* file have been pip installed. After that, open any command line interface that can be used to compile python and navigate to the folder. Run *filter.py* first and then *insert.py* right after. If there are write permission errors to books1.csv while running *filter.py* (because the program creates the books1.csv file and then writes data to it), then delete the book1.csv file, and then run *filter.py* again. After running both scripts, navigate to the src folder, and run *app.py*. Then navigate to http://127.0.0.1:5000/, and so our program will be running on there (Local IP: 127.0.0.1 and Port: 5000).

To demonstrate our functionality, several screenshots of our web interface have been provided in the link below. A user will first be directed to the login page. Given that the user cannot login because he/she is new, he/she would need to click the Sign Up option and create a new account. After creating the new account, the user will be redirected back to the login page where he/she can sign in with those newly created credentials. Once the user has signed in, he/she is directed to the main page, which shows the catalogue of books present in our dataset. A normal user is limited in functions they can perform: they can only search for a book (by title, genre, author, or availability) in the search bar, view their issued books and total fines (through the dropdown from the Members field in the navigation bar up top), and sign out (also through the dropdown from the Members field in the navigation bar up top).

Admins can do everything a normal user can do by accessing the Members dropdown options, but have added functionality as shown in the Admin dropdown. When an admin signs in (email: admin@gmail.com, password: admin), an admin is directed to the main page and can Issue Book, Return/Mark Late book, Manage Books (Add/Delete Books to and from the books list), Manage Members(Add/Delete Members to and from the Members list), and Show all members (See list of all members).

Lastly, for all the forms a member can fill out, corresponding error messages will be thrown if the program detects an error in the field inputs.

(Github Link**:** **https://github.com/dhong515/ECE-464-Library-Management-System**)

(Screenshots Link: **https://drive.google.com/drive/folders/1Bh5uIqeVUGEkMRCKiLr8fKU4Dq2RWsZm?usp=sharing**)

Work Division

Danny was responsible for collecting the books dataset from Kaggle and then writing the *filter.py* and *insert.py* scripts to filter out our data and insert it to our database. He was also responsible for writing the *forms.py* file that dealt with request forms in the backend. Danny wrote the checkmember() function and also wrote error checking code in member related functions in *views.py* to ensure that an error would be thrown if there exists missing information in requested form fields. In *views.py*, Danny wrote all member related functions: userpage(), index(), signup(), addmember(), deletemember(), showmember(). Lastly, Danny was responsible for writing the *index.html, addmember.html, background.html, login.html, showmembers.html*, and *layout.html* files for the frontend.

Arthur was responsible for setting up and initializing the database in SQLite and then writing the *model.py* function that sets up the ORM tables in the backend and then connecting it to the SQLite database in *createdb.py*. Arthur wrote all book and borrows related functions in *views.py*: searchbook(), issue_book(), add_book(), deletebook(), returnbook(), showbooks(), showmybooks(), and issue_late(). In regards to issue_late(), Arthur also set up and implemented the scheduler function in *app.py* that makes queries to the database and increases a user's fine by $0.01 for every second it's late. Lastly, Arthur was responsible for writing the *issuebook.html, addbook.html,* and *showbooks.html* files for the frontend.

Known Bugs/Missing Features

There aren't any known functional deficiencies within our code that we are aware of. However, as pointed out by our friend and classmate Jonathan, there is a chance that there are duplicate entries of books in our dataset. Although our addbook() function takes this into account and throws an error if the Admin tries to insert a title that already exists, the issue could lie within the Kaggle dataset that we have decided to use to populate our books table. For instance, if the dataset itself has more than one copy of the same book (more than one book with the same title), then we definitely should have implemented an error checking in our *filter.py* program where we filter out duplicate entries from the books data.

Given more time, we also could have added some extra functionality in our backend. For instance, perhaps we could've added a Change Password form for all Members, in which members would be able to change their password. This form would likely be present when logging in to our program since a common issue that users face is the fact that they forget what they set their password to be when they first signed up.

Additionally, we could’ve also added forms to the addmember() and addbooks() functions where the Admin would be able to edit information for a pre-existing user or book in the database. Our current code would just throw an error for trying to add a book/member whose title/email already exists in the database, but it’s also likely that the Admin was simply trying to edit the information of a particular book/member that is in the database instead.

Lastly, we would’ve experimented more with the frontend portion of the code and made it more complex and visually appealing. For example, we could’ve looked into the React frontend framework in order to accomplish this. Although we lack a bit of experience in this aspect, adopting a frontend framework should still be relatively simple for us to learn and apply.

Conclusion:

While the barebones functionality is implemented, the frontend does leave a lot to be desired when it comes to style. Given more time, we would’ve experienced more with CSS and Javascript in order to create a more interactive and decorative frontend. The database project was a rather holistic approach that while mirroring a bit of the project experience we had in software development, was more fine tuned towards the technicalities we are more interested in, and was rather enjoyable to learn while coding it.