University Library Database

By Alexandria Brown and Ryan Mueller

Introduction:

This project includes the university library database schema which has been created with SQLite. It also includes a command line user interface that has been written in python utilizing the splite3 library to enable interactions with the database including adding, removing, changing, and viewing data. Three user roles have been created for this system with students and professors being able to search for books, borrow and return books (Students can borrow for up to 90 days whole professors can borrow for up to a year), and view their own borrowing history, while librarians have further administrative privileges allowing them to generate reports, add and remove books, and add potential new users.

Entity-Relationship Modeling (ERM)

The user tables and their corresponding Login tables all have one-to-one relationships because each potential user can only make one account. Each Book can have multiple copies. Each transaction can have only one book copy and only one user associated with it but every book copy and every user can have multiple transactions. Each user can have one token in the Active table while they are interacting with the database, but the table itself contains no foreign keys. The ID attribute would be a foreign key if it wasn't split between the two tables. We use cred to point to the table where the ID came from.

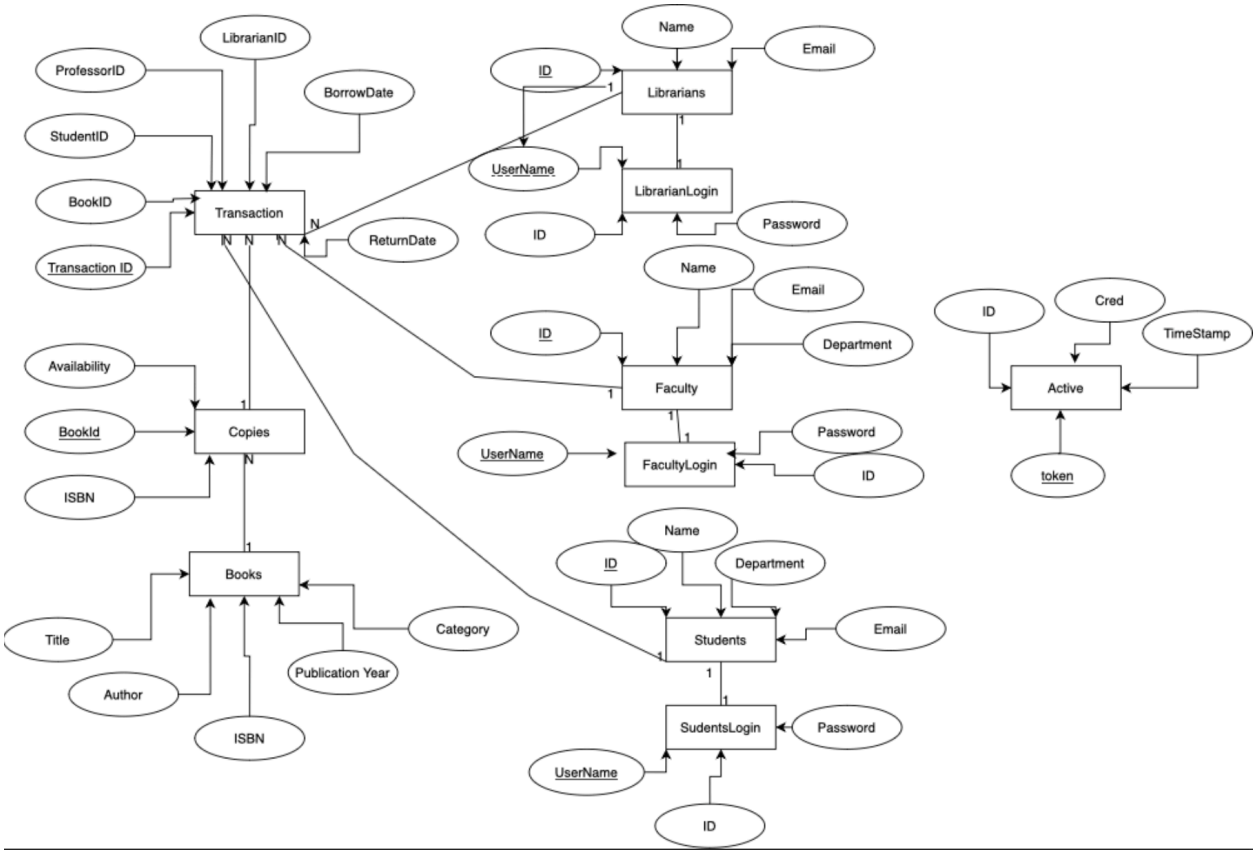

Fig. 1

Database Schema:

The STUDENTS, FACULTY, and LIBRARIANS tables are tables within the database that store the information of members of the university who are eligible to create a Library account. Once an account is made and a user has been authorised to use the library, a users login information is stored in the respective Login table, which include the primary key of the users tables as well as the user's user name and password. Any user can create their own transactions in order to borrow books. Each transaction is also linked to the specific copy that it is borrowing. Any book in BOOKS can have between 0 and many copies.

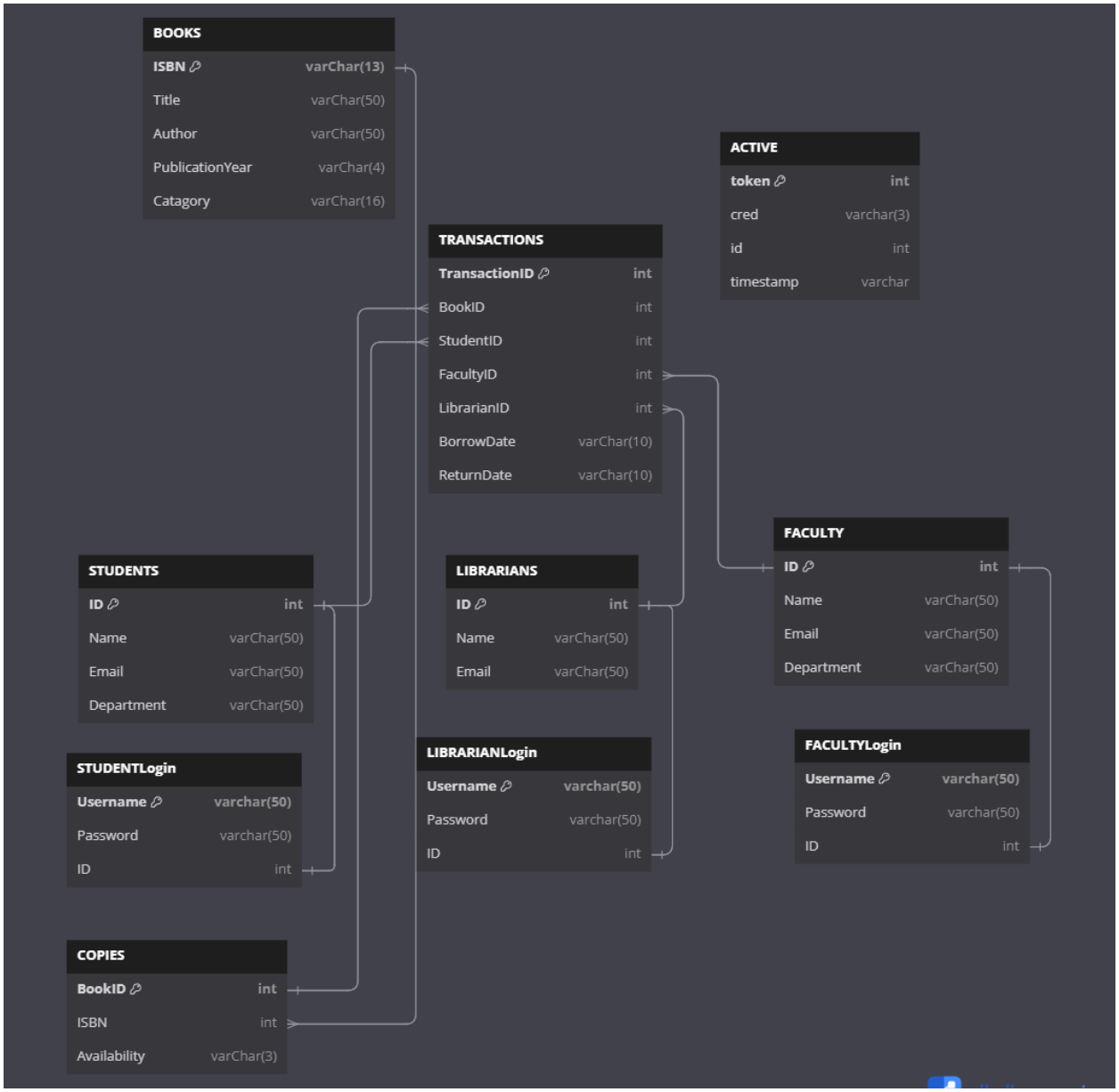

Fig. 2

SQL Queries:

While in the code the queries are all formatted with the python sqlite library, here are some of the queries formatted in sqlite for easier reading. Figure 3 is a query used to retrieve the three most popular books for the `generateTrendsReport()` action. Similar queries are also used in the `generateTrendsReport()` function for the most popular authors and categories. Figure 4 is a query from the `AddFaculty()` function used by a librarian to input a new faculty potential user into the FACULTY table. Similar queries are used in the `AddStudent()` and the

`AddLibrarian()` functions for inserting new potential users in their respective roles. Additional queries can be viewed in the functions shown in figures 6-10.

```
SELECT Title, count(*)
FROM TRANSACTIONS NATURAL JOIN COPIES NATURAL JOIN BOOKS
Group by ISBN
ORDER by count(*)
DESC LIMIT 3
```

Fig. 3

```
insert into FACULTY (ID, name, email, department) VALUES(<id>, <name>, <email>, <department>)
```

Fig. 4

Python Code:

We start with the `start()` finction shown in figure 5, which prompts the user to either sign in if they do not have a library account or login if they do. The `signIn()` function first verifies that the user is a member of the university by passing their name and university ID into the `verifySignIn()` function and then takes their user name and password to create a new account by storing their information in the appropriate Login table in the database. The `login()` function checks the user imputed username and password and checks them against the usernames and passwords stored in the appropriate login table as determined by the role argument. If their is a username and password entry that matches the user imputed ones, the user is logged in to the database.

```
def start():
    print("Are you a STUDENT, FACULTY, or LIBRARIAN?")
    role = input()
    role = role.upper()#To make sure the input is uppercase
    #if the user has inputted a legitimate role continue
    if(role=="STUDENT" or role=="FACULTY" or role=="LIBRARIAN"):
        print("Do you already have an account? (yes or no)")
        account = input()
        if(account == "yes"):
            login(role)#To access the database
        elif(account == "no"):
            signIn(role)#To create an account
        else: start()
    else:
        start() #if the role is not legitimate then start interaction over
```

Fig. 5

Once logged in the progrma uses a variety of functions that run quries to interact with the database. This program uses the `sqlite3` library already built into python to interact with the Library Database, Library.db, which was built with SQLite. As can be seen in the highlighted portions of figure 6 below, `sqlite3` works by first opening the database with the `connect()` function, then calling the `cursor()` function, and then calling the `execute()` function which takes the query to be run as an argument. After a query that alters the database the `commit()` function to be called so that the changes will be persistent. At the end of each function that opens the database the database should be closed with the `close()` function.

```
def AddLibrarian():
    db=sqlite3.connect('Library.db')
    cursor=db.cursor()
    print("Enter the ID")
    id = input()
    print("Enter the name")
    name = input()
    print("Enter the email")
    email = input()
    cursor.execute("insert into LIBRARIANS (ID, name, email) VALUES(?, ?,
?)",(id,name,email))
    db.commit()
    db.close()
```

Fig. 6

When a query returns a result, that result is then returned as an array of arrays that can be accessed by the `fetchall()` function as shown in figure 7.

```
#Retrieve the title, bookID, and transaction ID from the transactions, coppies,
and books tables
    cursor.execute("SELECT Title, TRANSACTIONS.BookID, TransactionID FROM
TRANSACTIONS NATURAL JOIN COPIES NATURAL JOIN BOOKS")
    result = cursor.fetchall()
    i = 0
    #For each tuple, write the results to the file
    for tuple in result:
        title = result[i][0]
        bookId = result[i][1]
        transactionId = result[i][2]
        if(overDue(transactionId)):
          date = getDueDate(transactionId)
```

```
            row = title+","+str(bookId)+","+date.strftime("%x")+"\n"
            file.write(row)
        i = i+1
```

Fig. 7

Another important package is the dateTime package which allows the program to both record the date on which a transaction takes place (figure 8), compare two date objects to see which one comes first (figure 8), and calculate a date a certain number of days from a given date (figure 9).

```
def overDue(transactionId):
    dueDate = (getDueDate(transactionId))
    curDate = date.today()

    connection = sqlite3.connect('Library.db')
    cursor = connection.cursor()
    cursor.execute("SELECT ReturnDate FROM TRANSACTIONS Where TransactionID
=?", (transactionId,))
    result = cursor.fetchall()
    Rdate = result[0][0]

    if(dueDate<curDate and Rdate != None):
        return True
    else:
        return False
```

Fig. 8

```
def getDueDate(transactionId):
    connection = sqlite3.connect('Library.db')
    cursor = connection.cursor()
    cursor.execute("SELECT BorrowDate, StudentID FROM TRANSACTIONS Where
TransactionID =?", (transactionId,))
    result= cursor.fetchall()
    theDate = date.fromisoformat(result[0][0])
    if(result[0][1] == None):
        dueDate = theDate + timedelta(365)
    else:
        dueDate = theDate + timedelta(90)
    cursor.close()
    return dueDate
```

Fig. 9

After login, authorization is handled with tokens using the `verify()` function (figure 10). This function takes as parameters a token and the function name. The token is used in a sql query to retrieve the credentials of the user. We use the credentials of the user to decide whether or not they have access to the requested function. `verify()` is called at the beginning of any function in our database that require authorization. This authorization and authentication system would be useful in a server-client relationship becasue it would reduce the amount of information the client needs to send to the server. As this is not a server-client service, these tokens are slightly less impactful. Some improvements to the token functionality could be made in the `CreateToken()` function (figure 11). Currently, the tokens that are generated are not very strong. We could make better tokens by taking the hash of random numbers.

```
def Verify(token,transaction):

  db=sqlite3.connect("Library.db")
  cursor=db.cursor()
  cursor.execute('Select cred from ACTIVE where token=?',(token,))
  cred=str(cursor.fetchall())[3:7]
  cursor.close()
  db.close()
  privileges=[]
  if cred=="stud":
      privileges=["CheckOutCoppie","ReturnCopies","Search","viewBorrowingHistory"] ##stud
privileges
  if cred == "fac'":
      privileges=["CheckOutCoppie","ReturnCopies","Search","viewBorrowingHistory"] ##fac'
privileges
  if cred =="lib'":

privledges=["LibrarianAddBook","LibrarianRemoveBook","CheckOutCoppie","ReturnCopies","Search
","viewBorrowingHistory","GenerateReport","addUser"] ##lib' privileges
  if transaction in privileges:
      return True
  else:
      return False
```

Fig. 10

```
#Creates a token to be used to identify a what actions a user can take
def CreateToken(): #Future improvment: More secure tokens
  db=sqlite3.connect("Library.db")
  cursor=db.cursor()
  cursor.execute("Select max(token) from active")
  token=int(cursor.fetchone()[0])+1
  cursor.close()
  db.close()
  return token
```

Fig. 11