Library Management System

Database Management System MSCS_542L_256

Team Amigos

Marist College
School of Computer Science and Mathematics

Submitted To:
Dr. Reza Sadeghi

11-29-2023

Table of Contents

Table of Figures

Project Report of Library Management System

Team Name

Team Amigos

Team Members

1. Meenakshi Miryala Meenakshi.Miryala1@marist.edu (Team Head)
2. Snithika Reddy Gaddam SnithikaReddy.Gaddam1@ marist.edu (Team Member)
3. Rajesh Onteru Rajesh.Onteru1@ marits.edu (Team Member)

Description of Team Members

1. Meenakshi Miryala

I am Meenakshi Miryala. I completed my bachelor's degree in electronics and communications. Even though I don't have the supporting background for my master's, I still managed to learn the basics of Python and SQL to stay engaged in this enduring field. I got placed in one of the multinational companies "Cognizant" as a programmer analyst. However, I turned my route to the USA for a master's at MARIST College to gain and train myself in computer science. Because I believe that global exposure tremendously helps advance my skills and career.

2. Snithika Reddy Gaddam

I Completed my undergrad in Information Technology. Where I gained a lot of knowledge About Programming in Undergrad. Projects based on Python and machine learning algorithms. In which we incorporated a database to retrieve the data. Seeking knowledge into the depths of databases I heard cloud which is virtual data. This led to completing my masters in MSCS Cloud as a Major stream.

3. Rajesh Onteru

I have done my graduation in the field of Computer Science Engineering from JNTUH. I have worked in various companies like Amazon, Synchrony, and Coinbase as a Support Analyst. My zeal to learn new technologies and interest in cloud computing led me to pursue a Master's. With this project, I'm sure I will gain in-depth knowledge of Data Base Management Systems which will help me in my career.

Section 1: Introduction

The library has a vast number of books and resources which are specifically arranged in a proper order to make the readers hassle-free. It is a difficult and time-consuming process to manage the data manually. We need a system to handle the records and data expertly to overcome such complexities. This benefits both the readers and the service providers. Library management is a system that is designed to handle data virtually.

We must be able to manage all the data that has been created already and the current data. The old data must be preserved securely and adjusted by the newly created records. The ultimate moto of the library management system is to record the necessary details such as the name of the book, specified identification number, and name of the author followed by the borrower's details which incorporate name, phone number, email ID, student ID, borrowed date, and expected date to return the book.

However, it’s quite difficult to maintain the records on paper so the library management helps to doodle the data virtually by using SQL to create the database, programming language, and graphical user interface (GUI) to interact with the users.

Section 2: Project Objectives

- Project objective is to leverage these models to create a Library Management System that optimizes operations and enhances the overall library experience for our academic community.
- Our project’s main goal is to create a ER diagram and EER diagram, which is the main step in creating SQL database.
- It consists of multiple entities with its desired attributes where entities maintain various cardinalities.
- A GUI is built for user interaction, connected to the database to retrieve the data.
- Application is used in advanced search capabilities including keyword, title, author, and subject with filters and sorting options for easy resource discovery.
- The application will allow students to place holds or reservations on books or items that are currently checked out and notify them when the items become available.
- Students could renew borrowed items online, subject to library policies and restrictions.
- Easy interface to collect feedback from library users through surveys and suggestion forms to improve services.
- The application provides the capability for the students or faculty to modify their contact information and profile changes with proper authentication.

- Built using some tools for managing interlibrary loan requests and tracking borrowed and lent materials.

Section 3: Related work

- One of the best examples of LMS found on the internet is Evergreen[1] which is open-source library system.
  REVIEW
  Evergreen is a powerful and flexible library management system that offers numerous benefits, open-source solutions. It has flexibility and scalability where it has user friendly interface.

- A Comparative Analysis of Entity-Relationship Diagrams.[2]
  REVIEW
  The Entity-Relationship diagram has been widely used in structured analysis and conceptual modeling. The ER approach is easy to understand, powerful to model real-world problems, and readily translated into a database schema. The ERD views that the real world consists of a collection of business entities, the relationship between them, and the attributes used to describe them. Other ER modeling semantics used by most methodologies include cardinality, participation, and generalization.
- Vufind[3] is the LMS that is present on the Internet.
  REVIEW
  To work with Vufind we need to have a minimum knowledge of the internal Library System i.e. about the books and the genres where finding something would be difficult if we lack knowledge.

Section 4: Merits

- User Authentication and Authorization methods and inadequate user role-based access controls leading to unauthorized actions.
- LMS helps libraries efficiently organize their collections. It allows librarians to input and manage information about books, journals, multimedia materials, and other resources, making it easier for users to search for and locate items.
- LMS provides a user-friendly interface for students to search for materials.
- Features like keyword searching, filters, and advanced search options help users find what they need quickly.

- LMS often enables remote access to library resources, allowing students to access books, databases, and other materials from anywhere with an internet connection.

Section 5: GitHub Address

https://github.com/Meenakshi1402/TeamAmigos

Section 6: Entity Relationship Model (ER Model)

External ER for admin

Admins have different roles within the system, granting them specific privileges and access levels. Can view transaction history, including checkouts and returns. Can manage the inventory of books, add new books, update book details, etc. Can oversee and manage book reservations made by users. Admins have access to administrative tools and settings for managing the library system. Admins can process and manage loan requests made by users. Admins can manage member accounts, issue library cards, update member information, etc. Please look at the feedback provided by users and take necessary actions. Access logs related to user activities, transactions, and system usage.

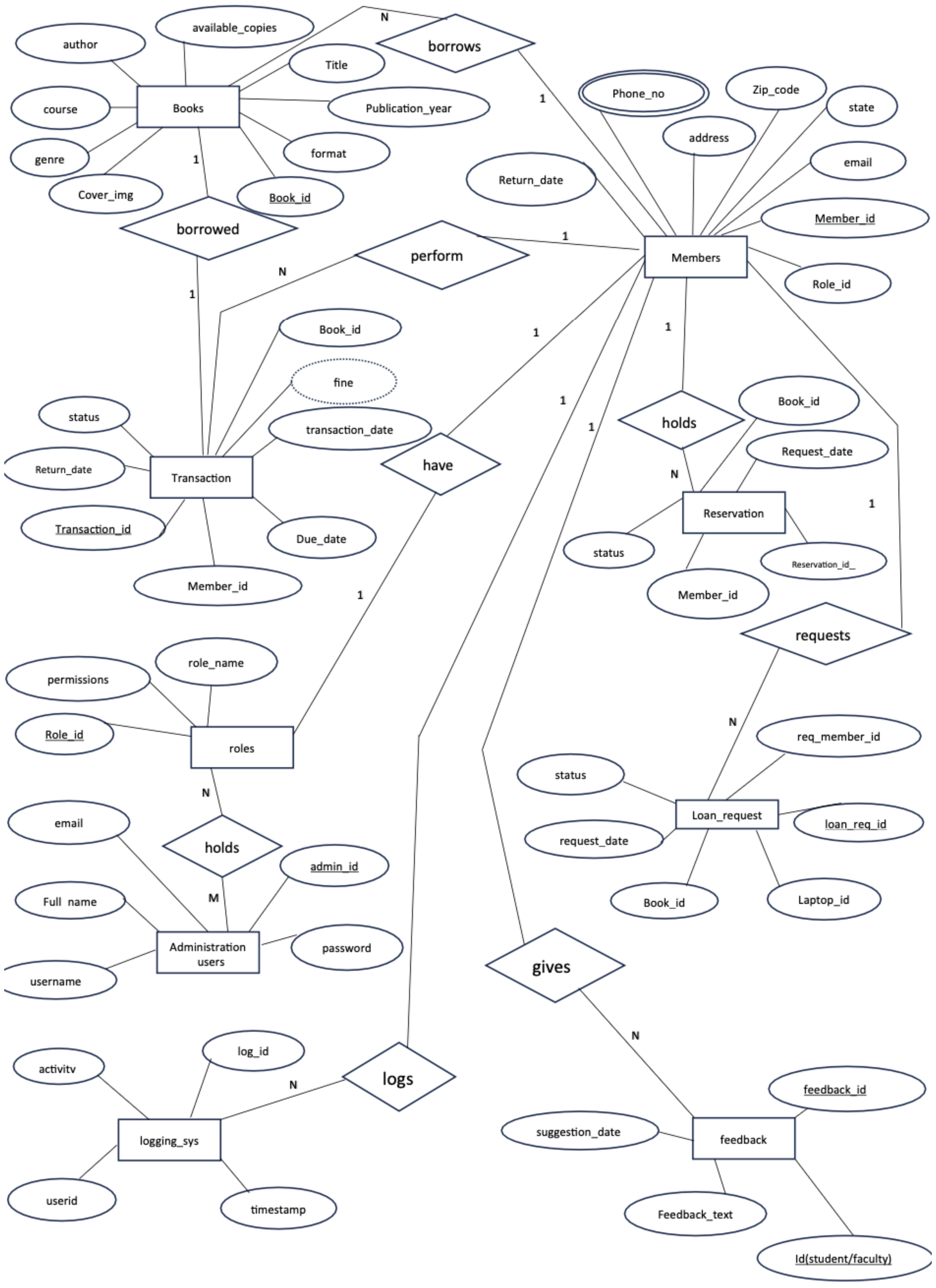


Figure 1:ER diagram in the admin point of view

External ER for user

Users have different roles within the system, determining their privileges and access levels, though typically users have limited roles compared to admins, viewing their transaction history, including books borrowed and returned. Search for books, check availability, and borrow or return books, make reservations for books they want to borrow in the future. Users can submit requests for loans, specifying the books they wish to borrow, provide feedback on their library experience, specific books, or any issues they encounter, do not directly interact with the logging system. Users can manage their user profiles, update contact information, and change account settings.

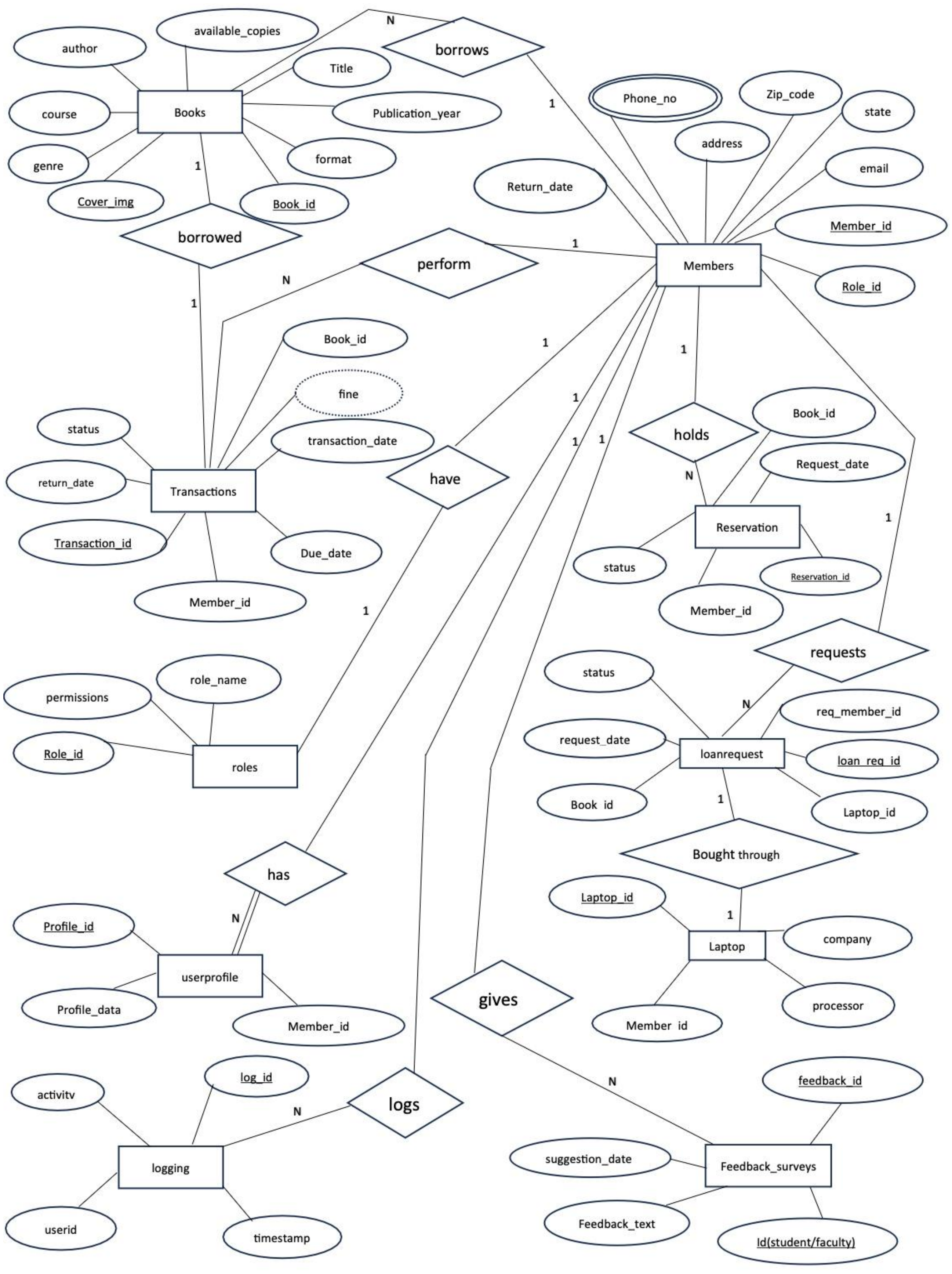


Figure 2:ER diagram in the user point of view

1. **Books**

- This table represents information about the library's collection of books. It includes details such as the book's unique identifier `book_id`, title, author, course, available copies, publication year, cover image, genre, and format.

- Relationships:

-Books to Members have many-to-one relationships.
-Books to reservations have one-to-one relationships.
-Books to transactions have one-to-one relationships.

2. **Members**

- This table stores information about both students and faculty members who use the library's services. It includes a unique member identifier (`member_id`), name, email, phone number, and address.

- Relationships:

-Has a one-to-many relationship with Books, reservations, loan requests, feedback, and user profile tables.
-Has a one-to-one relationship with the roles table.

In the relation between members and user profiles, the user profile has total participation as every profile has a member associated with it.

3. **Transactions**

- This table records the transactions related to the borrowing and returning of books. It includes a transaction identifier (`transaction_id`), book identifier (`book_id`), member identifier (`member_id`), transaction date, due date, return date, and a status indicator (e.g., checked out, returned).

- Relationships:

- Relates to the "Books" with one-to-one cardinality and the "Members" table with many-to-one cardinality.

4. **Holds/Reservations**

- This table manages book reservations made by library members. It includes a hold identifier (`hold_id`), book identifier (`book_id`), member identifier (`member_id`), request date, and a status indicator (e.g., pending, fulfilled).

- Relationships:

- Relates to the "Books" with one-to-one cardinality and the "Members" table with many-to-one cardinality.

5. **Feedback/Surveys**

- This table collects feedback and suggestions from library users. It includes a feedback identifier (`feedback_id`), a member identifier (`member_id`) to link feedback to specific users (optional if anonymous), feedback text, and the suggestion date.
- **Relationships:**
  - Relates to the "Members" table with many-to-one cardinality.

6. **User Profile**

- This table stores additional profile-related data for library members. It includes a profile identifier (`profile_id`), a member identifier (`member_id`) to link profiles to specific members, and a field for storing profile data.
- **Relationships:**
  - Relates to the "Members" table with many-to-one cardinality.

7. **Interlibrary Loan Requests**

- This table tracks requests for books that are not available in the library's collection. It includes a loan request identifier (`loan_request_id`), requester member identifier (`requester_member_id`), book identifier (`book_id`), request date, and a status indicator (e.g., pending, approved, denied).
- **Relationships:**
  -Relates to the "Laptop" with one-to-one and "Members" with many-to-one cardinality respectively.

8. **Administration Users**

- This table stores information about administrative users of the library management system. It includes an admin identifier (`admin_id`), username, password, full name, and email.
- **Relationships:**

Typically, this table would have a many-to-many relationship with the "Roles and Permissions" table to manage access control.

9. **Roles**

- This table defines user roles and their associated permissions within the library management system. It includes a role identifier (`role_id`), role name (e.g., student, faculty, admin), and a field for specifying permissions.

-**Relationships**:

-Typically, this table would have one-to-one related to the "members" table and many-to-many with the “administration” table.

10. **Logging and Audit Trail (for tracking system activities)**

- This table records system activities and user actions for auditing and tracking purposes. It includes a log identifier (`log_id`), and a user identifier (`user_id`) to link logs to specific users, activity descriptions, and timestamps.

- **Relationships**

- Relates to the "Members" with many-to-one cardinality.

11. **Laptop**

-This table consists of the laptop identifier(laptop_id), company name, member_id, and the type of processor. It is used to track the data of the members who borrow the laptops.

-**Relationship**

- Relates to the “loan request” table with a one-to-one relationship.

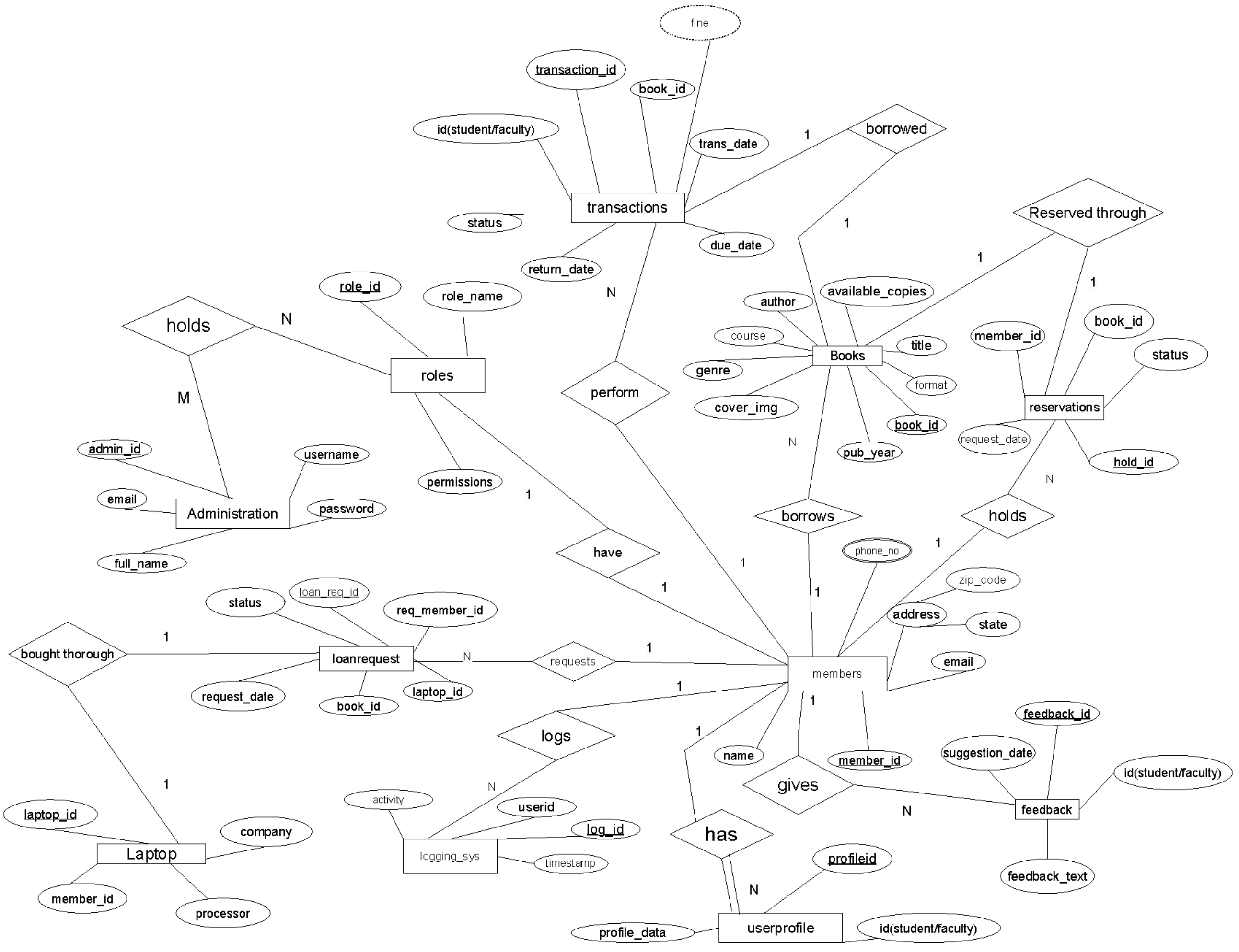


Figure 3:ER diagram

Section 7: Enhanced Entity Relationship Model (EER Model)

The Enhanced Entity-Relationship (EER) diagram takes the foundational structure of the Entity-Relationship (ER) diagram and extends it with advanced modeling concepts to provide a more nuanced and comprehensive representation of the Library Management System (LMS). The EER diagram introduces the following enhancements:

- **Attributes and Relationships:**

- The EER diagram retains the core entities and attributes found in the ER diagram, including "Books" with attributes such as book_id, title, author, and more. Relationships between entities, such as the association between "Books" and "Members" through the "Transactions" and "Holds/Reservations" tables, remain intact. These relationships facilitate book transactions, reservations, and member interactions.

- **Additional Attributes for Enhanced Profiling:**

- To capture a more comprehensive profile of library members, the EER diagram introduces an extended "User Profile" entity. This entity stores additional profile-related data beyond basic member information, allowing for a richer user experience and personalized services within the LMS.

- **Access Control with Roles and Permissions:**

- Building upon the ER diagram's basic user management, the EER diagram refines the "Roles and Permissions" entity. It enables granular access control by specifying what each role can access and do within the system. This enhancement ensures that administrators, students, and faculty members have appropriate privileges aligned with their roles.

- **Audit and Logging for System Transparency:**

- The EER diagram retains the "Logging and Audit Trail" entity to record system activities and user actions. This feature ensures transparency and accountability by tracking changes made by members and administrators, enhancing security and data integrity.

- **Interlibrary Loan Requests Refinement:**

- The "Interlibrary Loan Requests" entity is refined to accommodate the generalized member hierarchy, allowing both students and faculty to make requests for books not available in the library's collection. This enhancement ensures a seamless interlibrary loan process.

- **Laptop details for eBooks purchase:**

-The laptop entity allows the members to lend laptops from the library same as books where eBooks are provided for them through a loan request.

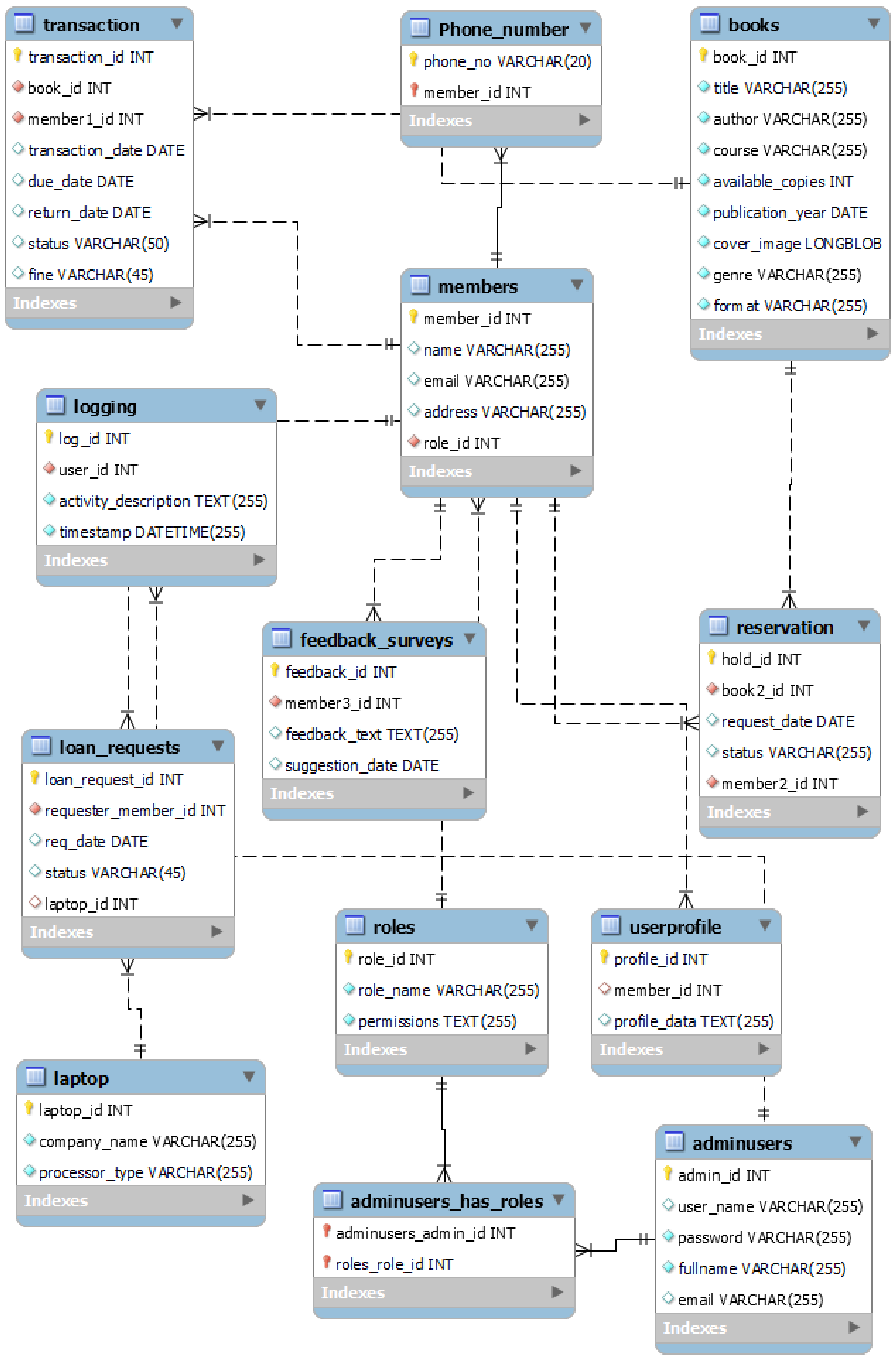


Figure 4:EER diagram

Section 8: Database Development

Firstly to create a Database we need to create a Schema as our project is based on library management we created the schema using the below syntax.

Syntax: CREATE SCHEMA `lib` ;

Secondly, we are creating tables according to the above ER diagram using the entities and attributes.

ADMINISTRATION_USERS TABLE

```
CREATE TABLE Administration_Users (
    admin_id INT PRIMARY KEY,
    username VARCHAR(255),
    password VARCHAR(255),
    full_name VARCHAR(255),
    email VARCHAR(255)
);
```

CODE DESCRIPTION: The "Administration_Users" table is designed to store information about administration users, including a unique identifier (admin_id), username, password (considered insecure in practice, should use hashed and salted versions), full name, and email address.

admin_id (Integer): This column serves as the primary key, providing a unique identifier for each administration user.

username (Varchar): A variable character field with a maximum length of 255 characters, likely intended to store the username of each administration user.

password (Varchar): Another variable character field with a maximum length of 255 characters, probably meant to store the password of each administration user. It's important to note that storing passwords directly in a database is not recommended for security reasons. Instead, a hashed and salted version of the password should be stored.

full_name (Varchar): This column is likely meant to store the full name of each administration user. It's a variable character field with a maximum length of 255 characters.

email (Varchar): A variable character field with a maximum length of 255 characters, likely intended to store the email address of each administration user.

BOOKS TABLE

```
CREATE TABLE Books (
    book_id INT PRIMARY KEY,
    title VARCHAR(255),
    author VARCHAR(255),
    course VARCHAR(255),
    available_copies INT,
    publication_year INT,
    cover_image BLOB,
    genre VARCHAR(255),
    format VARCHAR(255)
);
```

CODE DESCRIPTION:The table is designed to store information about books, including a unique identifier (book_id), title, author, course association, available copies, publication year, cover image (as binary data), genre, and format.

book_id (Integer): This column is set as the primary key, meaning it will contain unique values for each row and is used to uniquely identify each book.

title (Varchar): A variable character field with a maximum length of 255 characters, likely intended to store the title of each book.

author (Varchar): Another variable character field with a maximum length of 255 characters, probably meant to store the author's name for each book.

course (Varchar): This column, with a maximum length of 255 characters, may be used to store information about the course or subject associated with the book.

available_copies (Integer): An integer field representing the number of available copies of the book.

publication_year (Integer): An integer field intended to store the publication year of the book.

cover_image (BLOB - Binary Large Object): This column is likely intended to store binary data representing the cover image of the book. BLOBs are often used for storing large amounts of binary data, such as images.

genre (Varchar): A variable character field with a maximum length of 255 characters, possibly used to store the genre of the book.

format (Varchar): Another variable character field with a maximum length of 255 characters, possibly used to store the format of the book (e.g., paperback, hardcover, ebook).

MEMBERS TABLE

```
CREATE TABLE Members (

   member_id INT PRIMARY KEY,

   name VARCHAR(255),

   email VARCHAR(255),

   address VARCHAR(255),

   role_id INT,

   FOREIGN KEY (role_id) REFERENCES Roles(role_id)

);
```

CODE DESCRIPTION: The "Members" table is designed to store information about members, including a unique identifier (member_id), name, email, phone number, address, and a reference to their role through the role_id foreign key. The role information is linked to another table named "Roles.

member_id (Integer): This column is set as the primary key, meaning it will contain unique values for each row and is used to uniquely identify each member.

name (Varchar): A variable character field with a maximum length of 255 characters, likely intended to store the name of each member.

email (Varchar): Another variable character field with a maximum length of 255 characters, probably meant to store the email address of each member.

address (Varchar): A variable character field with a maximum length of 255 characters, probably used to store the address of each member.

role_id (Integer): This column represents a foreign key referencing the role_id column in another table named "Roles." This foreign key establishes a relationship between the "Members" table and the "Roles" table, indicating the role or position of each member.

The FOREIGN KEY (role_id) REFERENCES Roles(role_id) constraint ensures that the values in the role_id column of the "Members" table must match the values in the role_id column of the "Roles" table.

PHONE NUMBER TABLE

```
CREATE TABLE phone_number (
  Phone_no VARCHAR(20) NOT NULL,
  member_id INT NOT NULL,
  PRIMARY KEY (Phone_no, member_id),
    FOREIGN KEY (member_id) REFERENCES members (member_id)
);
```

CODE DESCRIPTION: The combination of `Phone_no` and `member_id` columns together form the primary key for this table, ensuring each combination is unique. This means that each phone number can be associated with multiple member IDs, but each pairing of phone number and member ID must be unique in the table.

Phone_no - A column of type VARCHAR(20) that stores phone numbers. It's marked as NOT NULL, meaning it must have a value for each row.

member_id - An integer column that stores member IDs. It's also marked as NOT NULL.

FOREIGN KEY constraint is added on the `member_id` column, referencing the `member_id` column in the `members` table. This constraint ensures that the values inserted into the `member_id` column in the `phone_number` table must exist in the `member_id` column of the `members` table, maintaining referential integrity.

FEEDBACK_SURVEYS TABLE

```
CREATE TABLE Feedback_Surveys (
    feedback_id INT PRIMARY KEY,
    member_id INT,
    feedback_text TEXT,
    suggestion_date DATE,
    FOREIGN KEY (member_id) REFERENCES Members(member_id)
);
```

CODE DESCRIPTION: The "Feedback_Surveys" table is designed to store information about feedback surveys, including a unique identifier (feedback_id), the member who provided the feedback (member_id), the text content of the feedback, and the date when the feedback was submitted. The member information is linked to another table named "Members" through the member_id foreign key.

feedback_id (Integer): This column is set as the primary key, meaning it will contain unique values for each row and is used to uniquely identify each feedback survey.

member_id (Integer): This column represents a foreign key referencing the member_id column in another table named "Members." This foreign key establishes a relationship between the "Feedback_Surveys" table and the "Members" table, indicating the member who provided the feedback.

The FOREIGN KEY (member_id) REFERENCES Members(member_id) constraint ensures that the values in the member_id column of the "Feedback_Surveys" table must match the values in the member_id column of the "Members" table.

feedback_text (TEXT): This column is likely intended to store the text content of the feedback provided by the member. The TEXT data type is suitable for storing large amounts of text.

suggestion_date (DATE): This column is intended to store the date when the feedback or suggestion was submitted. The DATE data type is used for storing dates.

LAPTOP TABLE

```
CREATE TABLE Laptop (
    laptop_id INT PRIMARY KEY,
    company_name VARCHAR(255),
    processor_type VARCHAR(255)
);
```

CODE DESCRIPTION: The "Laptop" table is designed to store information about laptops, including a unique identifier (laptop_id), the name of the company producing the laptop, and details about the processor type used in the laptop.

laptop_id (Integer): This column is set as the primary key, meaning it will contain unique values for each row and is used to uniquely identify each laptop.

company_name (Varchar): A variable character field with a maximum length of 255 characters, likely intended to store the name of the company that manufactured or produced the laptop.

processor_type (Varchar): Another variable character field with a maximum length of 255 characters, probably meant to store information about the type of processor used in the laptop.

LOAN_REQUESTS TABLE

```
CREATE TABLE Loan_Requests (
    loan_request_id INT PRIMARY KEY,
    requester_member_id INT,
    laptop_id INT,
    request_date DATE,
    status VARCHAR(50)
);
```

CODE DESCRIPTION:The "Loan_Requests" table is designed to manage information about loan requests for laptops, including a unique identifier for each request (loan_request_id), the member making the request (requester_member_id), the specific laptop requested (laptop_id), the date of the request (request_date), and the status of the request (status).

loan_request_id (Integer): This column serves as the primary key, providing a unique identifier for each loan request.

requester_member_id (Integer): This column likely represents the member (user) who made the loan request. It could be a foreign key referencing the member_id column in a "Members" table.

laptop_id (Integer): This column likely represents the specific laptop that is requested for a loan. It could be a foreign key referencing the laptop_id column in a "Laptop" table.

request_date (DATE): This column is intended to store the date when the loan request was made. The DATE data type is used to represent dates.

status (Varchar): This column is likely meant to store the status of the loan request, indicating whether it's pending, approved, denied, or some other status. The VARCHAR(50) data type allows for a variable character field with a maximum length of 50 characters

USER_PROFILE TABLE

```
CREATE TABLE User_Profile (
    profile_id INT PRIMARY KEY,
    member_id INT,
    profile_data TEXT,
    FOREIGN KEY (member_id) REFERENCES Members(member_id)
);
```

CODE DESCRIPTION: The "User_Profile" table is designed to store information about user profiles, including a unique identifier for each profile (profile_id), a reference to the associated member (member_id), and textual data representing the user's profile details (profile_data). The member information is linked to another table named "Members" through the member_id foreign key.

profile_id (Integer): This column serves as the primary key, providing a unique identifier for each user profile.

member_id (Integer): This column represents a foreign key referencing the member_id column in another table, likely a "Members" table. It establishes a relationship between the "User_Profile" table and the "Members" table, indicating the user associated with the profile.

The FOREIGN KEY (member_id) REFERENCES Members(member_id) constraint ensures that the values in the member_id column of the "User_Profile" table must match the values in the member_id column of the "Members" table.

profile_data (TEXT): This column is likely intended to store textual data related to the user's profile. The TEXT data type allows for the storage of large amounts of text.

LOGGING TABLE

```
CREATE TABLE Logging (
    log_id INT PRIMARY KEY,
    user_id INT,
    activity_description TEXT,
    timestamp DATETIME,
    FOREIGN KEY (user_id) REFERENCES Members(member_id)
);
```

CODE DESCRIPTION:The "Logging" table is designed to log activities, including a unique identifier for each log entry (log_id), a reference to the user associated with the activity (user_id), a description of the activity (activity_description), and a timestamp indicating when the activity occurred (timestamp). The user information is linked to another table named "Members" through the user_id foreign key.

log_id (Integer): This column serves as the primary key, providing a unique identifier for each log entry.

user_id (Integer): This column represents a foreign key referencing the member_id column in another table, likely a "Members" table. It establishes a relationship between the "Logging" table and the "Members" table, indicating the user associated with the logged activity.

The FOREIGN KEY (user_id) REFERENCES Members(member_id) constraint ensures that the values in the user_id column of the "Logging" table must match the values in the member_id column of the "Members" table.

activity_description (TEXT): This column is likely intended to store textual data describing the activity or event that is being logged. The TEXT data type allows for the storage of large amounts of text.

timestamp (DATETIME): This column is intended to store the date and time when the activity occurred. The DATETIME data type is used to represent both date and time information.

RESERVATION TABLE

```
CREATE TABLE Reservation (
    reservation_id INT PRIMARY KEY,
    book_id INT,
    member_id INT,
    request_date DATE,
    status VARCHAR(50),
    FOREIGN KEY (book_id) REFERENCES Books(book_id),
    FOREIGN KEY (member_id) REFERENCES Members(member_id)
);
```

CODE DESCRIPTION: The "Reservation" table is designed to manage information about reservations, including a unique identifier for each reservation (reservation_id), the specific book being reserved (book_id), the member making the reservation (member_id), the date of the reservation request (request_date), and the status of the reservation (status). The book and member information are linked to the "Books" and "Members" tables through foreign key constraints.

reservation_id (Integer): This column serves as the primary key, providing a unique identifier for each reservation.

book_id (Integer): This column represents a foreign key referencing the book_id column in another table, likely a "Books" table. It establishes a relationship between the "Reservation" table and the "Books" table, indicating the specific book that has been reserved.

The FOREIGN KEY (book_id) REFERENCES Books(book_id) constraint ensures that the values in the book_id column of the "Reservation" table must match the values in the book_id column of the "Books" table.

member_id (Integer): This column represents a foreign key referencing the member_id column in another table, likely a "Members" table. It establishes a relationship between the "Reservation" table and the "Members" table, indicating the member who made the reservation.

The FOREIGN KEY (member_id) REFERENCES Members(member_id) constraint ensures that the values in the member_id column of the "Reservation" table must match the values in the member_id column of the "Members" table.

request_date (DATE): This column is intended to store the date when the reservation request was made. The DATE data type is used to represent dates.

status (Varchar): This column is likely meant to store the status of the reservation, indicating whether it's pending, approved, denied, or some other status. The VARCHAR(50) data type allows for a variable character field with a maximum length of 50 characters.

ROLES TABLE

```
CREATE TABLE Roles (
    role_id INT PRIMARY KEY,
    role_name VARCHAR(50),
    permissions TEXT
);
```

CODE DESCRIPTION: The "Roles" table is designed to define different roles within a system, including a unique identifier for each role (role_id), a name or title for the role (role_name), and information about the permissions associated with that role (permissions).

This table can be used to manage and assign roles to users, determining the level of access and functionality they have within the system.

role_id (Integer): This column serves as the primary key, providing a unique identifier for each role.

role_name (Varchar): A variable character field with a maximum length of 50 characters, likely intended to store the name or title of each role.

permissions (TEXT): This column is likely intended to store information about the permissions associated with each role. The TEXT data type allows for the storage of large amounts of text, which could represent a list of permissions or specific access rights granted to users with that role.

ADMIN_ROLES TABLE

```
CREATE TABLE Admin_Roles (
    admin_id INT,
    role_id INT,
    PRIMARY KEY (admin_id, role_id),
    FOREIGN KEY (admin_id) REFERENCES Administration_Users(admin_id),
    FOREIGN KEY (role_id) REFERENCES Roles(role_id)
);
```

CODE DESCRIPTION: The "Adminusers_has_Roles" table is used to model the many-to-many relationship between administration users and roles. It tracks which roles are assigned to which administration users. The composite primary key ensures the uniqueness of each association, and the foreign key constraints link the table to the "Administration_Users" and "Roles" tables, respectively.

admin_id (Integer): This column is part of a composite primary key and represents a foreign key referencing the admin_id column in the "Administration_Users" table. It establishes a relationship between the "Adminusers_has_Roles" table and the "Administration_Users" table, indicating the administration user associated with a particular role.

The FOREIGN KEY (admin_id) REFERENCES Administration_Users(admin_id) constraint ensures that the values in the admin_id column of the "Adminusers_has_Roles" table must match the values in the admin_id column of the "Administration_Users" table.

role_id (Integer): This column is the second part of the composite primary key and represents a foreign key referencing the role_id column in the "Roles" table. It establishes a relationship between the "Adminusers_has_Roles" table and the "Roles" table, indicating the role associated with a particular administration user.

The FOREIGN KEY (role_id) REFERENCES Roles(role_id) constraint ensures that the values in the role_id column of the "Adminusers_has_Roles" table must match the values in the role_id column of the "Roles" table.

PRIMARY KEY (admin_id, role_id): This line specifies that the combination of admin_id and role_id together forms the primary key for the "Adminusers_has_Roles" table. This ensures that each combination of administration user and role is unique in the table.

TRANSACTION TABLE

```
CREATE TABLE Transactions (
    transaction_id INT PRIMARY KEY,
    book_id INT,
    member_id INT,
    transaction_date DATE,
    due_date DATE,
    return_date DATE,
    status VARCHAR(50),
    FOREIGN KEY (book_id) REFERENCES Books(book_id),
    FOREIGN KEY (member_id) REFERENCES Members(member_id)
);
```

CODE DESCRIPTION: The "Transactions" table is designed to manage information about book transactions, including a unique identifier for each transaction (transaction_id), the specific book involved (book_id), the member involved (member_id), transaction date (transaction_date), due date for return (due_date), return date (return_date), and the status of the transaction (status). The book and member information are linked to the "Books" and "Members" tables through foreign key constraints.

transaction_id (Integer): This column serves as the primary key, providing a unique identifier for each transaction.

book_id (Integer): This column represents a foreign key referencing the book_id column in the "Books" table. It establishes a relationship between the "Transactions" table and the "Books" table, indicating the specific book involved in the transaction.

The FOREIGN KEY (book_id) REFERENCES Books(book_id) constraint ensures that the values in the book_id column of the "Transactions" table must match the values in the book_id column of the "Books" table.

member_id (Integer): This column represents a foreign key referencing the member_id column in the "Members" table. It establishes a relationship between the "Transactions" table and the "Members" table, indicating the member involved in the transaction.

The FOREIGN KEY (member_id) REFERENCES Members(member_id) constraint ensures that the values in the member_id column of the "Transactions" table must match the values in the member_id column of the "Members" table.

transaction_date (DATE): This column is intended to store the date when the transaction (borrowing or return) occurred. The DATE data type is used to represent dates.

due_date (DATE): This column is likely intended to store the due date for returning the borrowed book.

return_date (DATE): This column is likely intended to store the date when the borrowed book was returned.

status (Varchar): This column is likely meant to store the status of the transaction, indicating whether it's pending, completed, overdue, or some other status. The VARCHAR(50) data type allows for a variable character field with a maximum length of 50 characters.

Section 9: Loading Data and Performance Enhancements

1.Importing Data

BOOKS TABLE

INSERT INTO Books (book_id, title, author, course, available_copies, publication_year, cover_image, genre, format) VALUES

(1, 'Book1', 'Author1', 'Course1', 5, 2020, NULL, 'Fiction', 'Paperback'),

(2, 'Book2', 'Author2', 'Course2', 3, 2019, NULL, 'Science', 'Hardcover'),

(3, 'Book3', 'Author3', 'Course3', 8, 2021, NULL, 'History', 'E-book'),

(4, 'Book4', 'Author4', 'Course4', 2, 2018, NULL, 'Biography', 'Paperback'),

(5, 'Book5', 'Author5', 'Course5', 6, 2022, NULL, 'Mystery', 'Hardcover'),

(6, 'Book6', 'Author6', 'Course6', 4, 2023, NULL, 'Romance', 'E-book'),

(7, 'Book7', 'Author7', 'Course7', 7, 2024, NULL, 'Thriller', 'Paperback'),

(8, 'Book8', 'Author8', 'Course8', 9, 2025, NULL, 'Self-help', 'Hardcover'),

(9, 'Book9', 'Author9', 'Course9', 1, 2026, NULL, 'Poetry', 'E-book'),

(10, 'Book10', 'Author10', 'Course10', 10, 2027, NULL, 'Drama', 'Paperback');

Failed to insert because of foreign key constraint violation.

MEMBERS TABLE

INSERT INTO Members (member_id, name, email, address, role_id,password) VALUES

(1, 'Meenakshi Miryala', 'Meenakshi.Miryala1@marist.edu', 'Address 1', 1,’Meena@2002),

(2, 'Snithika Reddy Gaddam', 'SnithikaReddy.Gaddam1@marist.edu','Address 2', 2,’Sni@2002’),

(3, 'Rajesh Onteru', 'Rajesh.Onteru1@marist.edu', 'Address 3', 2,’Rajesh@1993’),

(11, 'Member1', 'member1@example.com', 'Address1', 1,’mem@11’),

(22, 'Member2', 'member2@example.com', 'Address2', 2),

(33, 'Member3', 'member3@example.com', 'Address3', 1),

(4, 'Member4', 'member4@example.com', 'Address4', 2),

```
(5, 'Member5', 'member5@example.com', 'Address5', 1),
(6, 'Member6', 'member6@example.com', 'Address6', 2),
(7, 'Member7', 'member7@example.com', 'Address7', 1),
(8, 'Member8', 'member8@example.com', 'Address8', 2),
(9, 'Member9', 'member9@example.com', 'Address9', 1),
(10, 'Member10', 'member10@example.com', 'Address10', 2);
```

PHONE_NUMBER TABLE

```
INSERT INTO Phone_number (phone_no, member_id)
VALUES
    ('Phone Number1', 1),
    ('Phone Number2', 2),
    ('Phone Number3', 3),
    ('1234567890', 11),
    ('2345678901', 22),
    ('3456789012', 33),
    ('4567890123', 4),
    ('5678901234', 5),
    ('6789012345', 6),
    ('7890123456', 7),
    ('8901234567', 8),
    ('9012345678', 9),
    ('0123456789', 10);
```

TRANSACTIONS TABLE

```
INSERT INTO Transactions (transaction_id, book_id, member_id, transaction_date, due_date, return_date, status) VALUES
(1, 1, 1, '2023-10-01', '2023-10-15', '2023-10-13', 'Returned'),
```

(2, 2, 2, '2023-10-02', '2023-10-16', '2023-10-14', 'Returned'),

(3, 3, 3, '2023-10-03', '2023-10-17', '2023-10-15', 'Returned'),

(4, 4, 4, '2023-10-04', '2023-10-18', '2023-10-16', 'Returned'),

(5, 5, 5, '2023-10-05', '2023-10-19', '2023-10-17', 'Returned'),

(6, 6, 6, '2023-10-06', '2023-10-20', '2023-10-18', 'Returned'),

(7, 7, 7, '2023-10-07', '2023-10-21', '2023-10-19', 'Returned'),

(8, 8, 8, '2023-10-08', '2023-10-22', '2023-10-20', 'Returned'),

(9, 9, 9, '2023-10-09', '2023-10-23', '2023-10-21', 'Returned'),

(10, 10, 10, '2023-10-10', '2023-10-24', '2023-10-22', 'Returned');

RESERVATION TABLE

INSERT INTO Reservation (reservation_id, book_id, member_id, request_date, status) VALUES

(1, 1, 1, '2023-10-01', 'Pending'),

(2, 2, 2, '2023-10-02', 'Pending'),

(3, 3, 3, '2023-10-03', 'Pending'),

(4, 4, 4, '2023-10-04', 'Pending'),

(5, 5, 5, '2023-10-05', 'Pending'),

(6, 6, 6, '2023-10-06', 'Pending'),

(7, 7, 7, '2023-10-07', 'Pending'),

(8, 8, 8, '2023-10-08', 'Pending'),

(9, 9, 9, '2023-10-09', 'Pending'),

(10, 10, 10, '2023-10-10', 'Pending');

FEEDBACK_SURVEYS TABLE

```
INSERT INTO Feedback_Surveys (feedback_id, member_id, feedback_text, suggestion_date) VALUES

(1, 1, 'Feedback1', '2023-10-01'),

(2, 2, 'Feedback2', '2023-10-02'),

(3, 3, 'Feedback3', '2023-10-03'),

(4, 4, 'Feedback4', '2023-10-04'),

(5, 5, 'Feedback5', '2023-10-05'),

(6, 6, 'Feedback6', '2023-10-06'),

(7, 7, 'Feedback7', '2023-10-07'),

(8, 8, 'Feedback8', '2023-10-08'),

(9, 9, 'Feedback9', '2023-10-09'),

(10, 10, 'Feedback10', '2023-10-10');
```

USER_PROFILE TABLE

```
INSERT INTO User_Profile (profile_id, member_id, profile_data) VALUES

(1, 1, 'ProfileData1'),

(2, 2, 'ProfileData2'),

(3, 3, 'ProfileData3'),

(4, 4, 'ProfileData4'),

(5, 5, 'ProfileData5'),

(6, 6, 'ProfileData6'),

(7, 7, 'ProfileData7'),

(8, 8, 'ProfileData8'),

(9, 9, 'ProfileData9'),

(10, 10, 'ProfileData10');
```

LOAN_REQUESTS TABLE

INSERT INTO Loan_Requests (loan_request_id, requester_member_id, laptop_id, request_date, status) VALUES

(1, 1, 1, '2023-10-01', 'Approved'),

(2, 2, 2, '2023-10-02', 'Approved'),

(3, 3, 3, '2023-10-03', 'Approved'),

(4, 4, 4, '2023-10-04', 'Approved'),

(5, 5, 5, '2023-10-05', 'Approved'),

(6, 6, 6, '2023-10-06', 'Approved'),

(7, 7, 7, '2023-10-07', 'Approved'),

(8, 8, 8, '2023-10-08', 'Approved'),

(9, 9, 9, '2023-10-09', 'Approved'),

(10, 10, 10, '2023-10-10', 'Approved');

ADMINISTRATION_USERS TABLE

INSERT INTO Administration_Users (admin_id, username, password, full_name, email) VALUES

(1, 'admin1', 'password1', 'Admin User1', 'admin1@example.com'),

(2, 'admin2', 'password2', 'Admin User2', 'admin2@example.com'),

(3, 'admin3', 'password3', 'Admin User3', 'admin3@example.com'),

(4, 'admin4', 'password4', 'Admin User4', 'admin4@example.com'),

(5, 'admin5', 'password5', 'Admin User5', 'admin5@example.com'),

(6, 'admin6', 'password6', 'Admin User6', 'admin6@example.com'),

(7, 'admin7', 'password7', 'Admin User7', 'admin7@example.com'),

(8, 'admin8', 'password8', 'Admin User8', 'admin8@example.com'), (9, 'admin9', 'password9', 'Admin User9', 'admin9@example.com'),

(10, 'admin10', 'password10', 'Admin User10', 'admin10@example.com');

Unique key constrain error because when duplicate value was entered.

ROLES TABLE

INSERT INTO Roles (role_id, role_name, permissions) VALUES

(1, 'Role1', 'Permission1'),

(2, 'Role2', 'Permission2'),

(3, 'Role3', 'Permission3'),

(4, 'Role4', 'Permission4'),

(5, 'Role5', 'Permission5'),

(6, 'Role6', 'Permission6'),

(7, 'Role7', 'Permission7'),

(8, 'Role8', 'Permission8'),

(9, 'Role9', 'Permission9'),

(10, 'Role10', 'Permission10');

Remove the Foreign key and then insert

SET FOREIGN_KEY_CHECKS=0 to disable foreign key check

ADMIN_ROLES TABLE

INSERT INTO Admin_Roles (admin_id, role_id) VALUES

(1, 1),

(2, 2),

(3, 3),

(4, 4),

```
(5, 5),
(6, 6),
(7, 7),
(8, 8),
(9, 9),
(10, 10);
```

LOGGING TABLE

```
INSERT INTO Logging (log_id, user_id, activity_description, timestamp) VALUES
(1, 1, 'Activity1', '2023-10-01 10:00:00'),
(2, 2, 'Activity2', '2023-10-02 11:00:00'),
(3, 3, 'Activity3', '2023-10-03 12:00:00'),
(4, 4, 'Activity4', '2023-10-04 13:00:00'),
(5, 5, 'Activity5', '2023-10-05 14:00:00'),
(6, 6, 'Activity6', '2023-10-06 15:00:00'),
(7, 7, 'Activity7', '2023-10-07 16:00:00'),
(8, 8, 'Activity8', '2023-10-08 17:00:00'),
(9, 9, 'Activity9', '2023-10-09 18:00:00'),
(10, 10, 'Activity10', '2023-10-10 19:00:00');
```

LAPTOP TABLE

```
INSERT INTO Laptop (laptop_id, company_name, processor_type) VALUES
(1, 'Company1', 'ProcessorType1'),
(2, 'Company2', 'ProcessorType2'),
(3, 'Company3', 'ProcessorType3'),
(4, 'Company4', 'ProcessorType4'),
(5, 'Company5', 'ProcessorType5'),
(6, 'Company6', 'ProcessorType6'),
(7, 'Company7', 'ProcessorType7'),
(8, 'Company8', 'ProcessorType8'),
(9, 'Company9', 'ProcessorType9'),
(10, 'Company10', 'ProcessorType10');
```

Selecting Data from tables

INNER JOIN

```
SELECT Books.title, Books.author, Members.name
FROM Books
INNER JOIN Transactions ON Books.book_id = Transactions.book_id
INNER JOIN Members ON Transactions.member_id = Members.member_id;
```

LEFT JOIN

```
SELECT Books.title, Books.author, Transactions.transaction_date
FROM Books
LEFT JOIN Transactions ON Books.book_id = Transactions.book_id;
```

RIGHT JOIN

```
SELECT Transactions.transaction_id, Transactions.transaction_date, Books.title
FROM Transactions
RIGHT JOIN Books ON Transactions.book_id = Books.book_id;
```

ORDER BY

```
SELECT Members.name, Books.title, Transactions.transaction_date
FROM Members
```

```
LEFT JOIN Transactions ON Members.member_id = Transactions.member_id
LEFT JOIN Books ON Transactions.book_id = Books.book_id
ORDER BY Members.name;
```

GROUPBY Aggregation

```
SELECT Members.name, COUNT(Transactions.transaction_id) AS total_transactions
FROM Members
INNER JOIN Transactions ON Members.member_id = Transactions.member_id
GROUP BY Members.name;
```

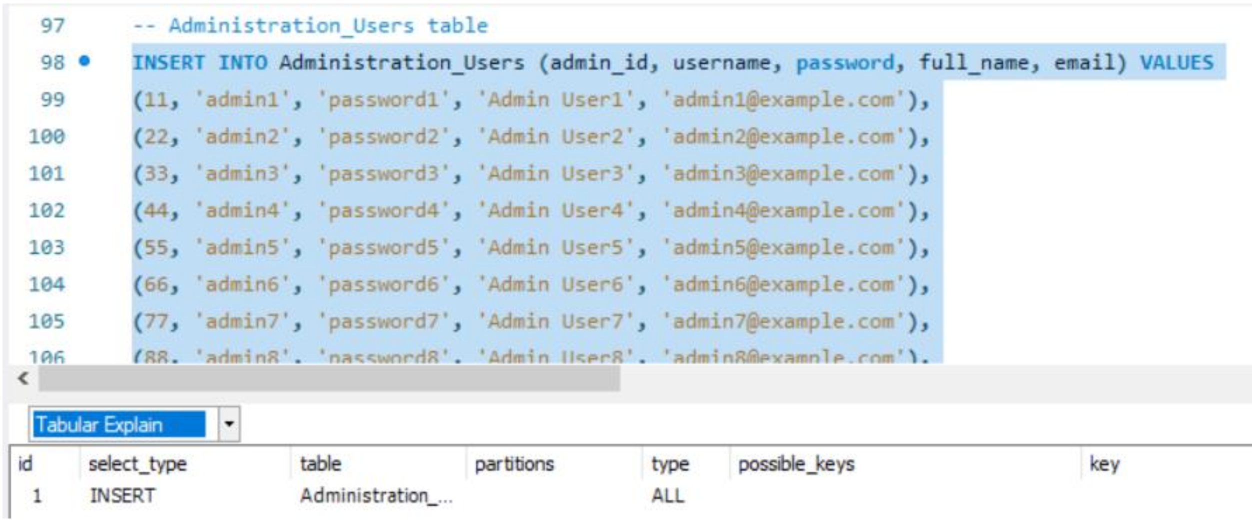
--Select queries for all tables

We are selecting the tables to display all tables

```
-- Books table
SELECT * FROM Books;

-- Members table
SELECT * FROM Members;

-- Transactions table
SELECT * FROM Transactions;

-- Reservation table
SELECT * FROM Reservation;
```

```
-- Feedback_Surveys table

SELECT * FROM Feedback_Surveys;


-- User_Profile table

SELECT * FROM User_Profile;


-- Loan_Requests table

SELECT * FROM Loan_Requests;


-- Administration_Users table

SELECT * FROM Administration_Users;


-- Roles table

SELECT * FROM Roles;


-- Admin_Roles table

SELECT * FROM Admin_Roles;


-- Logging table

SELECT * FROM Logging;


-- Laptop table

SELECT * FROM Laptop;
```

Bulk Insert



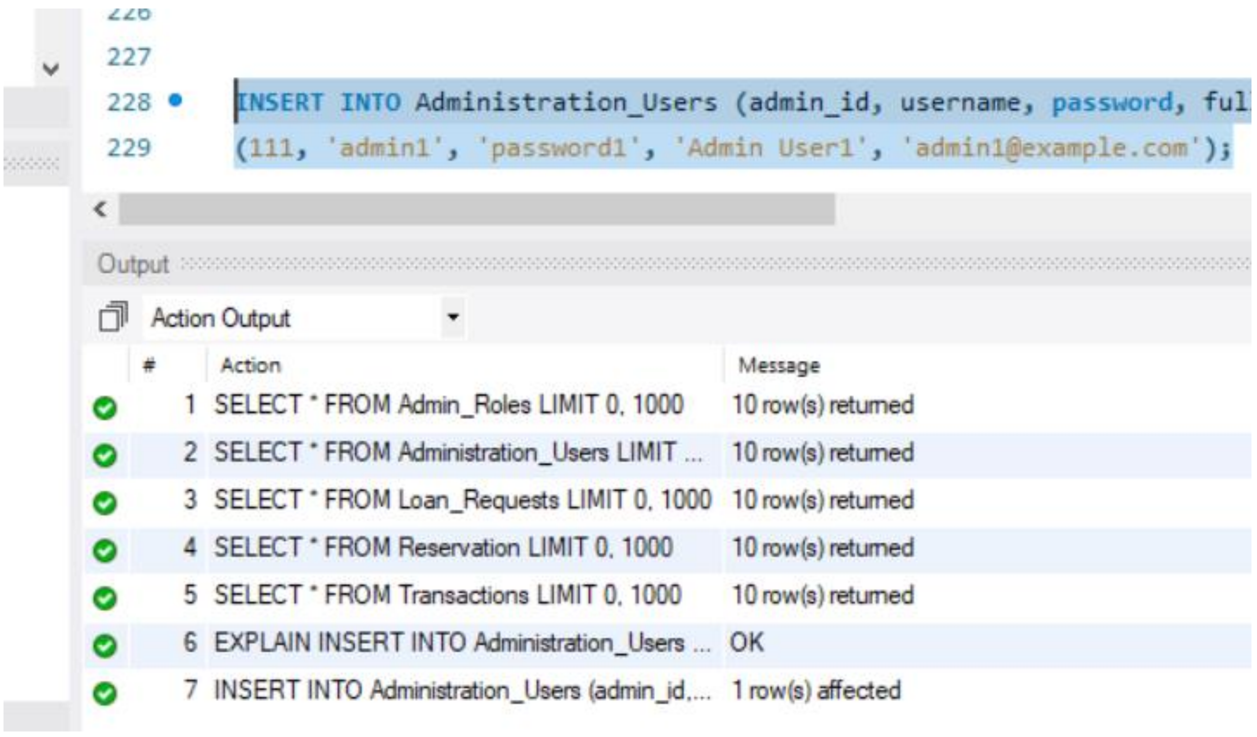
```sql
-- Administration_Users table
INSERT INTO Administration_Users (admin_id, username, password, full_name, email) VALUES
(11, 'admin1', 'password1', 'Admin User1', 'admin1@example.com'),
(22, 'admin2', 'password2', 'Admin User2', 'admin2@example.com'),
(33, 'admin3', 'password3', 'Admin User3', 'admin3@example.com'),
(44, 'admin4', 'password4', 'Admin User4', 'admin4@example.com'),
(55, 'admin5', 'password5', 'Admin User5', 'admin5@example.com'),
(66, 'admin6', 'password6', 'Admin User6', 'admin6@example.com'),
(77, 'admin7', 'password7', 'Admin User7', 'admin7@example.com'),
(88, 'admin8', 'password8', 'Admin User8', 'admin8@example.com'),
```

Tabular Explain

id	select_type	table	partitions	type	possible_keys	key
1	INSERT	Administration_...		ALL		

Figure 5:Bulk insert

Single Insert



```sql
INSERT INTO Administration_Users (admin_id, username, password, ful
(111, 'admin1', 'password1', 'Admin User1', 'admin1@example.com');
```

Output

Action Output

#	Action	Message
1	SELECT * FROM Admin_Roles LIMIT 0, 1000	10 row(s) returned
2	SELECT * FROM Administration_Users LIMIT ...	10 row(s) returned
3	SELECT * FROM Loan_Requests LIMIT 0, 1000	10 row(s) returned
4	SELECT * FROM Reservation LIMIT 0, 1000	10 row(s) returned
5	SELECT * FROM Transactions LIMIT 0, 1000	10 row(s) returned
6	EXPLAIN INSERT INTO Administration_Users ...	OK
7	INSERT INTO Administration_Users (admin_id,...	1 row(s) affected

Figure 6:Single insert

The process of entering several rows of data into a database table in a single go is bulk insert. It has various advantages over single-row inserts, including better speed, lower transaction overhead, less network traffic, simpler error handling, and better resource efficiency. However, bulk inserts may not be appropriate for many cases, such as when data must be put in one row at a time.

2.Implement Foreign Key Constraints

When we initially create a table, we may not anticipate all the foreign key relationships that are needed. Later, as our database evolves, you might identify the need to establish relationships between tables.

Using **ALTER TABLE** you can modify the existing table structure to add foreign key constraints, enforcing referential integrity between tables.

Table Name	Foreign Key	Referenced Table	Referenced Column	Comments
Members	role_id	Roles	role_id	Connects the role of a member to the Roles table
Transactions	book_id	Books	book_id	Links the book in a transaction to the Books table
Transactions	member_id	Members	member_id	Associates the member in a transaction to the Members table
Reservation	book_id	Books	book_id	Links the reserved book to the Books table
Reservation	member_id	Members	member_id	Associates the reserving member to the Members table
Feedback_Surveys	member_id	Members	member_id	Associates the feedback to a member in the Members table
User_Profile	member_id	Members	member_id	Associates the user profile to a member in the Members table

Figure 7:Foreign Key

-- Members Table

ALTER TABLE Members

ADD CONSTRAINT fk_role_id

FOREIGN KEY (role_id) REFERENCES Roles(role_id);

The implementation of the foreign key constraint involves the database system checking and ensuring that every role_id value in the Members table corresponds to a valid role_id in the Roles table. If a record is inserted or updated in the Members table, the foreign key constraint ensures that the associated role_id is valid, and if it's deleted from the Roles table, the constraint dictates how the database system should handle related records in the Members table

* Data Integrity: A foreign key constraint ensures that the values in the role_id column of the Members table correspond to valid values in the Roles table. This helps maintain data integrity by preventing the insertion of records with non-existent role_id values.

* Relationship Definition: It defines a relationship between the Members and Roles tables. In this case, it signifies that the Members table is associated with the Roles table through the role_id column.

* Enforcement of Referential Integrity: The foreign key constraint enforces referential integrity, meaning that you cannot have "orphaned" records in the Members table with role_id values that don't exist in the Roles table

-- Transactions Table

ALTER TABLE Transactions

ADD CONSTRAINT fk_book_id

FOREIGN KEY (book_id) REFERENCES Books(book_id);

ALTER TABLE Transactions

ADD CONSTRAINT fk_member_id

FOREIGN KEY (member_id) REFERENCES Members(member_id);

The foreign key implementation involves the database system checking and ensuring that every book_id and member_id value in the Transactions table corresponds to a valid book_id in the Books table and a valid member_id in the Members table, respectively. If a record is inserted or updated in the Transactions table, the foreign key constraints ensure that the associated book_id and member_id values are valid. If a referenced record is deleted from the Books or Members table, the constraints dictate how the database system should handle related records in the Transactions table.

* Data Integrity: These foreign key constraints ensure that the values in the book_id column of the Transactions table correspond to valid values in the Books table, and the values in the member_id column correspond to valid values in the Members table. This helps maintain data integrity by preventing the insertion of records with non-existent book_id or member_id values.

* Relationship Definition: These constraints define relationships between the Transactions table and the Books table (via book_id) and the Members table (via member_id). This is crucial for modeling the associations between entities in the database.

* Referential Integrity: The foreign key constraints enforce referential integrity, ensuring that every book_id and member_id in the Transactions table refers to an existing book_id in the Books table and an existing member_id in the Members table, respectively.

-- Reservation Table

ALTER TABLE Reservation

ADD CONSTRAINT fk_book1_id

FOREIGN KEY (book_id) REFERENCES Books(book_id);

ALTER TABLE Reservation

ADD CONSTRAINT fk_member_id1

FOREIGN KEY (member_id) REFERENCES Members(member_id);

The foreign key implementation involves the database system checking and ensuring that every book_id and member_id value in the Reservation table corresponds to a valid book_id in the Books table and a valid member_id in the Members table, respectively. If a record is inserted or updated in the Reservation table, the foreign key constraints ensure that the associated book_id and member_id values are valid. If a referenced record is deleted from the Books or Members table, the constraints dictate how the database system should handle related records in the Reservation table (e.g., cascade the deletion, set to NULL, etc., depending on the specified action).

* Data Integrity: These foreign key constraints ensure that the values in the book_id column of the Reservation table correspond to valid values in the Books table, and the values in the member_id column correspond to valid values in the Members table. This helps maintain data integrity by preventing the insertion of records with non-existent book_id or member_id values.

* Relationship Definition: These constraints define relationships between the Reservation table and the Books table (via book_id) and the Members table (via member_id). This is crucial for modeling the associations between entities in the database.

* Referential Integrity: The foreign key constraints enforce referential integrity, ensuring that every book_id and member_id in the Reservation table refers to an existing book_id in the Books table and an existing member_id in the Members table, respectively.

-- Feedback_Surveys Table

```
ALTER TABLE Feedback_Surveys

ADD CONSTRAINT fk_member_id2

FOREIGN KEY (member_id) REFERENCES Members(member_id);
```

The foreign key implementation involves the database system checking and ensuring that every member_id value in the Feedback_Surveys table corresponds to a valid member_id in the Members table. If a record is inserted or updated in the Feedback_Surveys table, the foreign key constraint ensures that the associated member_id value is valid. If a referenced record is deleted from the Members table, the constraint dictates how the database system should handle related records in the Feedback_Surveys table Data Integrity: The foreign key constraint ensures that the values in the member_id column of the

Feedback_Surveys table correspond to valid values in the Members table. This helps maintain data integrity by preventing the insertion of records with non-existent member_id values.

* Relationship Definition: The constraint defines a relationship between the Feedback_Surveys table and the Members table, indicating that the Feedback_Surveys table is associated with the Members table through the member_id column.

* Referential Integrity: The foreign key constraint enforces referential integrity, ensuring that every member_id in the Feedback_Surveys table refers to an existing member_id in the Members table.

-- User_Profile Table

```
ALTER TABLE User_Profile

ADD CONSTRAINT fk_member_id3

FOREIGN KEY (member_id) REFERENCES Members(member_id);
```

The foreign key implementation involves the database system checking and ensuring that every member_id value in the User_Profile table corresponds to a valid member_id in the Members table. If a record is inserted or updated in the User_Profile table, the foreign key constraint ensures that the associated member_id value is valid. If a referenced record is deleted from the Members table, the constraint dictates how the database system should handle related records in the User_Profile table

* Data Integrity: The foreign key constraint ensures that the values in the member_id column of the User_Profile table correspond to valid values in the Members table. This helps maintain data integrity by preventing the insertion of records with non-existent member_id values.

* Relationship Definition: The constraint defines a relationship between the User_Profile table and the Members table, indicating that the User_Profile table is associated with the Members table through the member_id column.

* Referential Integrity: The foreign key constraint enforces referential integrity, ensuring that every member_id in the User_Profile table refers to an existing member_id in the Members table.

-- Loan_Requests Table

```
ALTER TABLE Loan_Requests

ADD CONSTRAINT fk_requester_member_id

FOREIGN KEY (requester_member_id) REFERENCES Members(member_id);


ALTER TABLE Loan_Requests

ADD CONSTRAINT fk_laptop_id1

FOREIGN KEY (laptop_id) REFERENCES Laptop(laptop_id);
```

The foreign key implementation involves the database system checking and ensuring that every requester_member_id and laptop_id value in the Loan_Requests table corresponds to a valid member_id in the Members table and a valid laptop_id in the Laptop table, respectively. If a record is inserted or updated in the Loan_Requests table, the foreign key constraints ensure that the associated requester_member_id and laptop_id values are valid. If a referenced record is deleted from the Members or Laptop table, the constraints dictate how the database system should handle related records in the Loan_Requests table

* Data Integrity: These foreign key constraints ensure that the values in the requester_member_id column of the Loan_Requests table correspond to valid values in the Members table, and the values in the laptop_id column correspond to valid values in the

Laptop table. This helps maintain data integrity by preventing the insertion of records with non-existent requester_member_id or laptop_id values.

* Relationship Definition: These constraints define relationships between the Loan_Requests table and the Members table (via requester_member_id) and the Laptop table (via laptop_id). This is crucial for modeling the associations between entities in the database.

* Referential Integrity: The foreign key constraints enforce referential integrity, ensuring that every requester_member_id and laptop_id in the Loan_Requests table refers to an existing member_id in the Members table and an existing laptop_id in the Laptop table, respectively.

-- Admin_Roles Table

```
ALTER TABLE Admin_Roles

ADD CONSTRAINT fk_admin_id

FOREIGN KEY (admin_id) REFERENCES Administration_Users(admin_id);


ALTER TABLE Admin_Roles

ADD CONSTRAINT fk_role_id1

FOREIGN KEY (role_id) REFERENCES Roles(role_id);
```

The foreign key implementation involves the database system checking and ensuring that every admin_id and role_id value in the Admin_Roles table corresponds to a valid admin_id in the Administration_Users table and a valid role_id in the Roles table, respectively. If a record is inserted or updated in the Admin_Roles table, the foreign key constraints ensure that the associated admin_id and role_id values are valid. If a referenced record is deleted from the Administration_Users or Roles table, the constraints dictate how the database system should handle related records in the Admin_Roles table

* Data Integrity: These foreign key constraints ensure that the values in the admin_id column of the Admin_Roles table correspond to valid values in the Administration_Users table, and the values in the role_id column correspond to valid values in the Roles table. This helps maintain data integrity by preventing the insertion of records with non-existent admin_id or role_id values.

* Relationship Definition: These constraints define relationships between the Admin_Roles table and the Administration_Users table (via admin_id) and the Roles table (via role_id). This is crucial for modeling the associations between entities in the database.

* Referential Integrity: The foreign key constraints enforce referential integrity, ensuring that every admin_id and role_id in the Admin_Roles table refers to an existing admin_id in the Administration_Users table and an existing role_id in the Roles table, respectively.

-- Logging Table

```
ALTER TABLE Logging

ADD CONSTRAINT fk_user_id

FOREIGN KEY (user_id) REFERENCES Members(member_id);
```

The foreign key implementation involves the database system checking and ensuring that every user_id value in the Logging table corresponds to a valid member_id in the Members table. If a record is inserted or updated in the Logging table, the foreign key constraint ensures that the associated user_id value is valid. If a referenced record is deleted from the Members table, the constraint dictates how the database system should handle related records in the Logging table

* Data Integrity: The foreign key constraint ensures that the values in the user_id column of the Logging table correspond to valid values in the Members table. This helps

maintain data integrity by preventing the insertion of records with non-existent user_id values.

* Relationship Definition: The constraint defines a relationship between the Logging table and the Members table, indicating that the Logging table is associated with the Members table through the user_id column.

* Referential Integrity: The foreign key constraint enforces referential integrity, ensuring that every user_id in the Logging table refers to an existing member_id in the Members table.

3.Normalization check

Books Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: The primary key for this table is book_id. All other attributes are fully functionally dependent on the entire primary key.
3NF: The Books table is in 3NF as it contains no transitive dependencies and each non-key attribute is dependent on the key.

Members Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: The primary key for this table is member_id. All other attributes are fully functionally dependent on the entire primary key.
3NF: In this table, there is a potential transitive dependency between role_id and name, email, phone_number, address. The role might be dependent on the member_id, and other attributes are dependent on member_id. To remove this transitive dependency, we can create a separate table for roles.

Phone Number Table

1NF:The table meets the requirements of 1NF because it doesn't contain repeating groups, and each column holds atomic (indivisible) values.It has a primary key (composed of `Phone_no` and `member_id`), and each column contains single values.

2NF:In this case, if `Phone_no` determines `member_id` (assuming one phone number belongs to only one member), it satisfies 2NF as both columns are part of the primary key.
3NF:If `Phone_no` and `member_id` are the only relevant attributes in this context (where `member_id` doesn't depend on any other non-key attributes), then it meets 3NF as well.

Transactions Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: In this table, the primary key is transaction_id. All other attributes (book_id, member_id, transaction_date, due_date, return_date, status) are fully functionally dependent on the entire primary key.
3NF:The Transactions is in 3NF as it contains no transitive dependencies and each non-key attribute is dependent on the key.

Reservation Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: In this table, the primary key is reservation_id. All other attributes (book_id, member_id, request_date, status) are fully functionally dependent on the entire primary key.
3NF: The Reservation table is in 3NF as it contains no transitive dependencies and each non-key attribute is dependent on the key.

Feedback_Surveys Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: The primary key for this table is feedback_id. All other attributes (member_id, feedback_text, suggestion_date) are fully functionally dependent on the entire primary key.
3NF: In this table, there is a transitive dependency between member_id and feedback_text, suggestion_date. The member might be dependent on the feedback_id, and other attributes are dependent on feedback_id. To remove this transitive dependency, we can create a separate table for User_Profile to store the user profile data.

User_Profile Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: The primary key for this table is profile_id. All other attributes (member_id, profile_data) are fully functionally dependent on the entire primary key.
3NF: The User_Profile table is in 3NF as it contains no transitive dependencies and each non-key attribute is dependent on the key.

Loan_Requests Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: The primary key for this table is loan_request_id. All other attributes (requester_member_id, laptop_id, request_date, status) are fully functionally dependent on the entire primary key.
3NF: The Loan_Requests table is in 3NF as it contains no transitive dependencies and each non-key attribute is dependent on the key.

Administration_Users Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: Admin_id is the primary key. All other attributes (username, password, full_name, and email) are dependent on the admin_id
3NF: The Administration_Users table is in 3NF as it contains no transitive dependencies and each non-key attribute is dependent on the key.

Roles Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: In this table, the primary key is role_id. All other attributes (role_name, permissions) are fully functionally dependent on the entire primary key.
3NF: The Roles table is in 3NF as it contains no transitive dependencies and each non-key attribute is dependent on the key.

Admin_Roles Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: In this table, the primary key is composed of both admin_id and role_id. All other attributes (admin_id and role_id) are fully functionally dependent on the entire primary key.
3NF: The Admin_Roles table is in 3NF as it contains no transitive dependencies and each non-key attribute is dependent on the key.

Logging Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: In this table, the primary key is log_id. All other attributes (user_id, activity_description, timestamp) are fully functionally dependent on the entire primary key.
3NF: In this table, there is a transitive dependency between user_id and activity_description, timestamp. The user might be dependent on the log_id, and other attributes are dependent on log_id. To remove this transitive dependency, we can create a separate table for Members to store user information.

Laptop Table

1NF: A table is in 1NF if it contains only atomic values and there are no repeating groups or arrays.
2NF: In this table, the primary key is laptop_id. All other attributes (company_name, processor_type) are fully functionally dependent on the entire primary key.
3NF: In this table, there are no transitive dependencies. All non-prime attributes are directly dependent on the primary key.

Section 10: Application development

1.Graphical user experience design

Book Management User Interface Design

- Add, edit, or delete books
- Search for books by title, author, genre, or publication year

Member Management User Interface Design

- Add, edit, or delete members
- Search for members by name, email address, or role
- View member details, including email addresses, phone numbers, and addresses

Transaction Management User Interface Design

- Manage book transactions, including borrowing and returning books
- View transaction details, such as the transaction date, due date, and return date
- Filter transactions to view active or overdue transactions

Reservation Management User Interface Design

- Allow members to reserve books and librarians to manage these reservations
- View reservation details, such as the request date and reservation status

Feedback and Survey User Interface Design

- Allow members to provide feedback and suggestions
- Display feedback details along with the member's information

User Profile User Interface Design

- Allow members to manage their profiles, update personal information, and view their profile

Loan Request User Interface Design

- Request specific items, like laptops
- See an overview of all pending and approved loan requests

Administration User Interface Design

- Manage administrators, their roles, and permissions
- Add, edit, or delete administration users
- View administration user details, such as usernames, full names, and email addresses

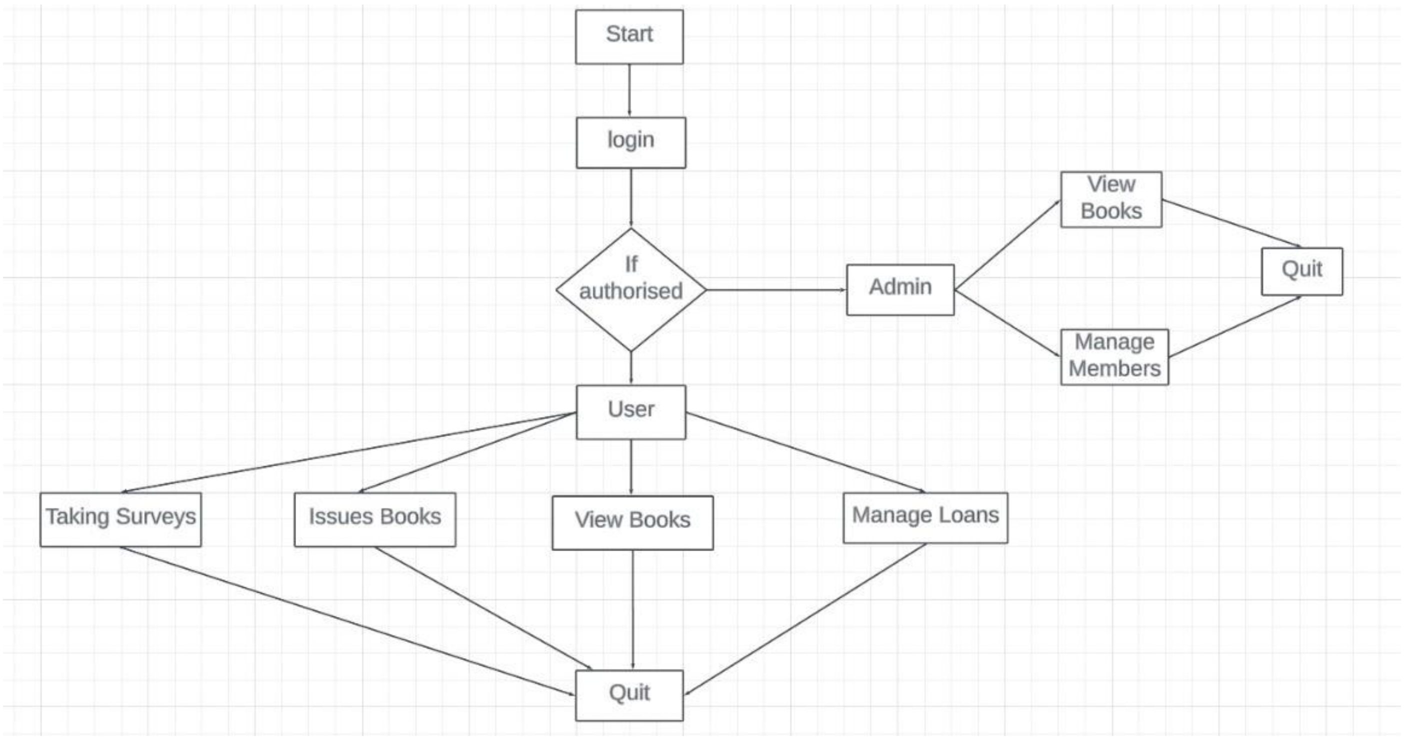


Figure 8:GUI flow chart

2.Database Views

In SQL, a view is a virtual table based on the result-set of an SQL statement. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database. You can add SQL statements and functions to a view and present the data as if the data were coming from one single table. A view is created with the CREATE VIEW statement

View for managing books, including transaction details and associated members.

```
CREATE VIEW BookManagementView AS

SELECT

   b.title as book_title,

   b.author as book_author,

   m.name as member_name,

   t.transaction_date as transaction_date
```

```
FROM Books b

INNER JOIN Transactions t ON b.book_id = t.book_id

INNER JOIN Members m ON t.member_id = m.member_id;
```

CODE DESCRIPTION: The view is designed to provide a comprehensive snapshot of book management by combining information from the "Books," "Transactions," and "Members" tables through appropriate joins. The INNER JOIN clauses connect records in the "Books" table with those in the "Transactions" table based on the book_id, and then further connect them with records in the "Members" table based on the member_id.

Users can query this view to obtain information about book titles, authors, associated members, and transaction dates in a single result set, simplifying book management tasks.

View for managing members, providing details such as contact information and roles.

```
CREATE VIEW MemberManagementView AS

SELECT

   m.name as member_name,

   m.email,

   p.phone_number,

   m.address,

   r.role_name

FROM Members m

INNER JOIN Roles r ON m.role_id = r.role_id;
```

CODE DESCRIPTION: The view is designed to provide a comprehensive snapshot of member management by combining information from the "Members" and "Roles" tables through an INNER JOIN. The INNER JOIN clause connects records in the "Members" table with those in the "Roles" table based on the role_id.

Users can query this view to obtain information about member names, contact details, and associated roles in a single result set, simplifying member management tasks. It's a useful way to consolidate relevant data for a more holistic view of members and their roles.

View for managing book transactions, displaying details such as due dates and return status.

```
CREATE VIEW TransactionManagementView AS
SELECT
    t.transaction_id,
    b.title as book_title,
    m.name as member_name,
    t.transaction_date,
    t.due_date,
    t.return_date,
    t.status
FROM Transactions t
INNER JOIN Books b ON t.book_id = b.book_id
INNER JOIN Members m ON t.member_id = m.member_id;
```

CODE DESCRIPTION: The view is designed to provide a comprehensive snapshot of book transactions by combining information from the "Transactions," "Books," and "Members" tables through appropriate joins. The INNER JOIN clauses connect records in the "Transactions" table with those in the "Books" and "Members" tables based on the book_id and member_id, respectively.

Users can query this view to obtain information about transaction details, including book titles, member names, transaction dates, due dates, return dates, and transaction status in a single result set. It's a useful way to consolidate relevant data for managing book transactions.

View for managing book reservations, including request details and member information.

```
CREATE VIEW ReservationManagementView AS
```

```
SELECT
    r.reservation_id,
    b.title as book_title,
    m.name as member_name,
    r.request_date,
    r.status
FROM Reservation r
INNER JOIN Books b ON r.book_id = b.book_id
INNER JOIN Members m ON r.member_id = m.member_id;
```

CODE DESCRIPTION: The view is designed to provide a comprehensive snapshot of book reservations by combining information from the "Reservation," "Books," and "Members" tables through appropriate joins. The INNER JOIN clauses connect records in the "Reservation" table with those in the "Books" and "Members" tables based on the book_id and member_id, respectively.

Users can query this view to obtain information about reservation details, including book titles, member names, reservation request dates, and reservation status in a single result set. It's a useful way to consolidate relevant data for managing book reservations.

View for managing feedback and surveys, displaying member feedback and suggestions.

```
CREATE VIEW FeedbackSurveyView AS
SELECT
    f.feedback_id,
    m.name as member_name,
    f.feedback_text,
    f.suggestion_date
FROM Feedback_Surveys f
```

```
INNER JOIN Members m ON f.member_id = m.member_id;
```

CODE DESCRIPTION: The view is designed to provide a comprehensive snapshot of feedback and surveys by combining information from the "Feedback_Surveys" and "Members" tables through an INNER JOIN. The INNER JOIN clause connects records in the "Feedback_Surveys" table with those in the "Members" table based on the member_id.

Users can query this view to obtain information about member feedback and suggestions, including member names, feedback text, and submission dates in a single result set. It's a useful way to consolidate relevant data for managing member feedback.

View for managing administration users, including roles and associated permissions.

```
CREATE VIEW AdminUserView AS
SELECT
    a.admin_id,
    a.username,
    a.full_name,
    a.email,
    r.role_name
FROM Administration_Users a
INNER JOIN Admin_Roles ar ON a.admin_id = ar.admin_id
INNER JOIN Roles r ON ar.role_id = r.role_id;
```

CODE DESCRIPTION: The view is designed to provide a comprehensive snapshot of administration users, including details about usernames, full names, emails, and associated roles. This information is obtained by combining data from the "Administration_Users," "Admin_Roles," and "Roles" tables through appropriate joins. The INNER JOIN clauses connect records in the "Administration_Users" table with those in the "Admin_Roles" and "Roles" tables based on the admin_id and role_id, respectively.

Users can query this view to obtain information about administration users, including usernames, full names, emails, and associated roles in a single result set. It's a useful way to consolidate relevant data for managing administration user roles.

3.Graphical user interface design

a.Connection to database

Connecting to a database involves establishing a connection between your application and the database system.

Import the Relevant Module:

Use import to bring in the necessary Python module for your chosen database system (mysql.connector).

Establish a Connection:

Use the module's connect function to establish a connection to the database. Provide connection details such as host, user, password, and database name.

Create a Cursor:

Create a cursor object using the cursor() method on the connection. The cursor is used to execute SQL queries.

Execute SQL Queries:

Use the cursor's execute method to run SQL queries against the database. This includes operations like creating tables, inserting data, updating records, etc.

Commit Changes and Close Connection:

After executing queries, commit the changes using commit() (for write operations). Close the connection when done using close().

Db.py code

```
import mysql.connector
import uuid
db_config = {
        'host': '127.0.0.1',
        'user': 'root',
        'password': 'root',
        'database': 'lib'
    }

def admin_login(username, password):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        query = "SELECT * FROM Administration_Users WHERE username = %s AND
password = %s"
        cursor.execute(query, (username, password))
        result = cursor.fetchone()
        cursor.close()
        connection.close()
        return result is not None
    except mysql.connector.Error as e:
        print("Error: {}".format(e))
        raise e
def member_login(username, password):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        query = "SELECT * FROM Members WHERE name = %s AND password = %s"
        cursor.execute(query, (username, password))
        result = cursor.fetchone()
        cursor.close()
        connection.close()
        return result is not None
    except mysql.connector.Error as e:
        print("Error: {}".format(e))
        raise e
def get_member_info(username, password):
  try:
    connection = mysql.connector.connect(**db_config)
    cursor = connection.cursor()
    query = "SELECT member_id as id, name, email, phone_number FROM Members
WHERE name = %s AND password = %s"
    cursor.execute(query, (username, password))
```

```
        result = cursor.fetchone()

        cursor.close()
        connection.close()

        return result

    except mysql.connector.Error as e:
        print("Error: {}".format(e))
        raise e
def get_member_info_by_name(name):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor(dictionary=True)
        query = "SELECT * FROM Members WHERE name = %s"
        cursor.execute(query, (name,))
        result = cursor.fetchone()

        cursor.close()
        connection.close()

        return result

    except mysql.connector.Error as e:
        print("Error: {}".format(e))
        raise e
def get_members():
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor(dictionary=True)
        query = "SELECT * FROM Members"
        cursor.execute(query)
        result = cursor.fetchall()
        cursor.close()
        connection.close()

        return result

    except mysql.connector.Error as e:
        print("Error: {}".format(e))
        raise e
def get_admin_info(username, password):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        query = "SELECT admin_id, username, password, full_name, email FROM Administration_Users WHERE username = %s AND password = %s"
        cursor.execute(query, (username, password))
        result = cursor.fetchone()

        cursor.close()
        connection.close()

        return result

    except mysql.connector.Error as e:
        print("Error: {}".format(e))
        raise e

def create_book(title, author, course, available_copies, publication_year, genre, book_format):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        cursor.execute("SELECT MAX(book_id) FROM Books")
        max_book_id = cursor.fetchone()[0]
        book_id = 1 if max_book_id is None else max_book_id + 1
        query = "INSERT INTO Books (book_id, title, author, course, available_copies, publication_year, genre, format) " \
                "VALUES (%s, %s, %s, %s, %s, %s, %s, %s)"
        values = (book_id, title, author, course, available_copies, publication_year, genre, book_format)
        cursor.execute(query, values)
        connection.commit()
        print("Book created successfully.")
    except mysql.connector.Error as e:
        print("Error: Failed to create book.")
        connection.rollback()
        raise e

def get_books():
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        query = "SELECT * FROM Books"
        cursor.execute(query)
        books = []
        for row in cursor.fetchall():
            book = {
                "book_id": row[0],
                "title": row[1],
                "author": row[2],
                "course": row[3],
                "available_copies": row[4],
                "publication_year": row[5],
                "genre": row[7],
                "format": row[8]
            }
            books.append(book)
        return books
    except mysql.connector.Error as e:
        print("Error: Failed to fetch books.")
        raise e

def get_book(book_id):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        query = "SELECT * FROM Books WHERE book_id = %s"
        cursor.execute(query, (book_id,))
        row = cursor.fetchone()
        if row:
            book = {
                "book_id": row[0],
                "title": row[1],
                "author": row[2],
                "course": row[3],
                "available_copies": row[4],
                "publication_year": row[5],
                "genre": row[6],
                "format": row[7]
            }
            return book
        else:
            return None
    except mysql.connector.Error as e:
        print("Error: Failed to fetch book.")
        raise e

def update_book(book_id, title, author, course, available_copies, publication_year, genre,
book_format):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        query = "UPDATE Books SET title = %s, author = %s, course = %s, available_copies =
%s, " \
                "publication_year = %s, genre = %s, format = %s WHERE book_id = %s"
        values = (title, author, course, available_copies, publication_year, genre, book_format,
book_id)
        cursor.execute(query, values)
        connection.commit()
        print("Book updated successfully.")
    except mysql.connector.Error as e:
        print("Error: Failed to update book.")
        connection.rollback()
        raise e

def delete_book(book_id):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        query = "DELETE FROM Books WHERE book_id = %s"
        cursor.execute(query, (book_id,))
        connection.commit()
        print("Book deleted successfully.")
    except mysql.connector.Error as e:
        print("Error: Failed to delete book.")
        connection.rollback()
        raise e

def get_reservations_for_member(member_id):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor(dictionary=True)
        query = "SELECT * FROM Reservation WHERE member_id = %s"
        cursor.execute(query, (member_id,))
        reservations = cursor.fetchall()
        return reservations
    except mysql.connector.Error as e:
        print(f"Error retrieving reservations: {e}")
        raise
    finally:
        if connection.is_connected():
            connection.close()

def create_reservation(book_id, member_id, request_date, status):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        cursor.execute("SELECT MAX(reservation_id) FROM Reservation")
        r_id = cursor.fetchone()[0]
        r_id = 1 if r_id is None else r_id + 1
        query = "INSERT INTO Reservation (reservation_id, book_id, member_id, 
request_date, status) VALUES (%s, %s, %s, %s, %s)"
        values = (r_id, book_id, member_id, request_date, status)
        cursor.execute(query, values)
        connection.commit()
        print("Reservation created successfully.")
    except mysql.connector.Error as e:
        print(f"Error creating reservation: {e}")
        raise
    finally:
        if connection.is_connected():
            connection.close()
def get_book_options():
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor(dictionary=True)
        query = "SELECT book_id, title FROM Books"
        cursor.execute(query)
        books = cursor.fetchall()
        return books
    except mysql.connector.Error as e:
        print(f"Error retrieving book options: {e}")
        raise
    finally:
        if connection.is_connected():
            connection.close()
def get_book_id_by_name(book_name):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor(dictionary=True)
        query = "SELECT book_id FROM Books WHERE title = %s"
        cursor.execute(query, (book_name,))
        result = cursor.fetchall()
        if result:
            return result[0]['book_id']
        else:
            return None
    except mysql.connector.Error as e:
        print(f"Error retrieving book ID by name: {e}")
        raise
    finally:
        if connection.is_connected():
            connection.close()
def get_loan_requests_for_member(member_id):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor(dictionary=True)
        query = "SELECT * FROM Loan_Requests WHERE requester_member_id = %s"
        cursor.execute(query, (member_id,))
        loan_requests = cursor.fetchall()
        return loan_requests
    except mysql.connector.Error as e:
        print(f"Error retrieving loan requests: {e}")
        raise
    finally:
        if connection.is_connected():
            connection.close()

def create_loan_request(requester_member_id, laptop_id, request_date, status):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        cursor.execute("SELECT MAX(loan_request_id) FROM Loan_Requests")
        request_id = cursor.fetchone()[0]
        request_id = 1 if request_id is None else request_id + 1
        query = "INSERT INTO Loan_Requests (loan_request_id, requester_member_id,
laptop_id, request_date, status) VALUES (%s, %s, %s, %s, %s)"
        values = (request_id, requester_member_id, laptop_id, request_date, status)
        cursor.execute(query, values)
        connection.commit()
        print("Loan request created successfully.")
    except mysql.connector.Error as e:
        print(f"Error creating loan request: {e}")
        raise
    finally:
        if connection.is_connected():
            connection.close()

def get_laptop_options():
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor(dictionary=True)
        query = "SELECT laptop_id, company_name, processor_type FROM Laptop"
        cursor.execute(query)
        laptops = cursor.fetchall()
        return laptops
    except mysql.connector.Error as e:
        print(f"Error retrieving laptop options: {e}")
        raise
    finally:
        if connection.is_connected():
            connection.close()

def get_laptop_id_by_name(laptop_name):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor(dictionary=True)
        company, processor = laptop_name.split("-")
        company = company.strip()
        processor = processor.strip()
        query = "SELECT laptop_id FROM Laptop WHERE company_name = %s AND
processor_type = %s"
        cursor.execute(query, (company, processor))
        result = cursor.fetchall()
        if result:
            return result[0]['laptop_id']
        else:
            return None
    except mysql.connector.Error as e:
        print(f"Error retrieving laptop ID by name: {e}")
        raise
    finally:
        if connection.is_connected():
            connection.close()

def create_feedback(member_id, feedback_text, suggestion_date):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        cursor.execute("SELECT MAX(feedback_id) FROM Feedback_Surveys")
        max_feedback_id = cursor.fetchone()[0]
        feedback_id = 1 if max_feedback_id is None else max_feedback_id + 1

        query = "INSERT INTO Feedback_Surveys (feedback_id, member_id, feedback_text,
suggestion_date) VALUES (%s, %s, %s, %s)"
        values = (feedback_id, member_id, feedback_text, suggestion_date)
        cursor.execute(query, values)

        connection.commit()
        print("Feedback created successfully.")
    except mysql.connector.Error as e:
        print("Error: Failed to create feedback.")
        connection.rollback()
        raise e
    finally:
        if connection.is_connected():
            connection.close()

def get_feedback_for_member(member_id):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor(dictionary=True)

        query = "SELECT * FROM Feedback_Surveys WHERE member_id = %s"
        cursor.execute(query, (member_id,))
        feedback_list = cursor.fetchall()

        cursor.close()
        connection.close()

        return feedback_list

    except mysql.connector.Error as e:
        print("Error: {}".format(e))
        raise e
def create_member(name, email, phone_number, address, password, role_id):
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor()
        cursor.execute("SELECT MAX(member_id) FROM Members")
        max_member_id = cursor.fetchone()[0]
        member_id = 1 if max_member_id is None else max_member_id + 1

        query = "INSERT INTO Members (member_id, name, email, phone_number, address,
password, role_id) VALUES (%s, %s, %s, %s, %s, %s, %s)"
        values = (member_id, name, email, phone_number, address, password, role_id)
        cursor.execute(query, values)

        connection.commit()
        print("Member created successfully.")
    except mysql.connector.Error as e:
        print("Error: Failed to create member.")
        connection.rollback()
        raise e
    finally:
        if connection.is_connected():
            connection.close()

def get_roles():
    try:
        connection = mysql.connector.connect(**db_config)
        cursor = connection.cursor(dictionary=True)
        query = "SELECT * FROM Roles"
        cursor.execute(query)
        roles = cursor.fetchall()
        return roles
    except mysql.connector.Error as e:
        print(f"Error retrieving roles: {e}")
        raise
    finally:
        if connection.is_connected():
            connection.close()
```

b.Login Page

- **Login**

Users access the system via the login page. It has spaces to input the username and password, a checkbox to choose between admin and member login, and a login button. In response to a successful or unsuccessful login, a message box is displayed by the on-login button click. Certain rights are opened when the Main Page is opened, depending on the kind of user member or admin.

The login page is used by library members to access their account and manage their library activities, such as checking out and returning books, renewing books, and viewing their due dates.

- Two text fields, one for the username and one for the password.

- If we check the member login box then it will direct us as a member if not it will direct as an admin.
- A button with the text "Login" will allow you to login to your login.

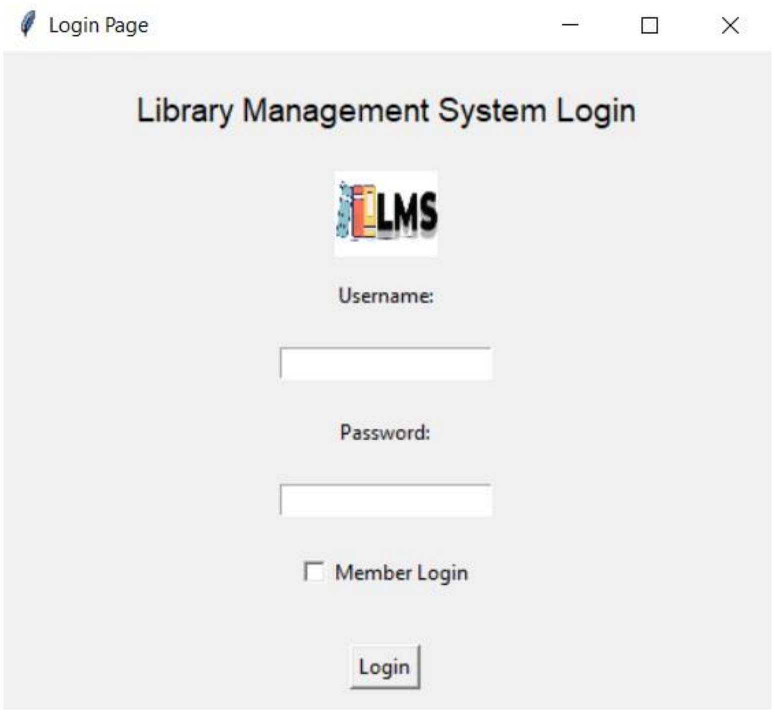


Figure 9:Login page

Importing Libraries:

- `tkinter` is imported to create the GUI.
- `messagebox` from `tkinter` for displaying messages.
- `member_login`, `get_member_info`, `admin_login`, `get_admin_info` are imported from separate modules (`db` and `main`) for handling member and admin login-related functions.
- `Image` and `ImageTk` from `PIL` for handling images.

2. **LoginPage Class**:

- `LoginPage` is a class inheriting from `tk.Tk`.
- The constructor (`__init__`) sets up the window properties like title, size, and creates various widgets for the login page.

3. **Widgets**:

- Labels (`label_welcome`, `label_username`, `label_password`) for displaying text.
- Entry widgets (`entry_username`, `entry_password`) to get user inputs.
- Checkbutton (`check_member_login`) to differentiate between member and admin login.
- Button (`btn_login`) that triggers the `on_login_click` function when clicked.

4. **Functions**:

- `validate_login`: Checks the validity of the login credentials by calling either `member_login` or `admin_login` functions based on the login type (member or admin).
- `get_member_info`: Retrieves member information by calling `get_member_info` function from the database.
- `on_login_click`: Triggered when the login button is clicked. It gets the entered username, password, and the type of login (member or admin) and validates the login. If successful, it retrieves user information and opens the `MainPage`.

Login.py Code

```
import tkinter as tk
from tkinter import messagebox
from db import member_login, get_member_info, admin_login, get_admin_info
from main import MainPage
from PIL import Image, ImageTk

class LoginPage(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Login Page")
        self.geometry("450x400")
        logo_path = "logo.png"
        logo_image = Image.open(logo_path)
        logo_image = logo_image.resize((60, 50), Image.LANCZOS)
        self.logo = ImageTk.PhotoImage(logo_image)
        self.label_welcome = tk.Label(self, text="Library Management System Login", 
font=("Helvetica", 14))
        self.label_welcome.pack(pady=20)
        self.logo_label = tk.Label(self, image=self.logo)
        self.logo_label.pack()

        self.label_username = tk.Label(self, text="Username:")
        self.label_username.pack(pady=10)
        self.entry_username = tk.Entry(self)
        self.entry_username.pack(pady=10)

        self.label_password = tk.Label(self, text="Password:")
        self.label_password.pack(pady=10)
        self.entry_password = tk.Entry(self, show="*")
        self.entry_password.pack(pady=10)
        self.is_member_login_var = tk.BooleanVar()
        self.check_member_login = tk.Checkbutton(self, text="Member Login", 
variable=self.is_member_login_var)
        self.check_member_login.pack(pady=10)

        self.btn_login = tk.Button(self, text="Login", command=self.on_login_click)
        self.btn_login.pack(pady=20)

    def validate_login(self, username, password, is_member_login):
        try:
            if is_member_login:
                return member_login(username, password)
            else:
                return admin_login(username, password)
        except Exception as e:
            print("Error: {}".format(e))
            return False

    def get_member_info(self, username, password):
        return get_member_info(username, password)

    def on_login_click(self):
        username = self.entry_username.get()
        password = self.entry_password.get()
        is_member_login = self.is_member_login_var.get()

        if self.validate_login(username, password, is_member_login):
            if is_member_login:
                member_info = self.get_member_info(username, password)
                messagebox.showinfo("Login Successful", f"Welcome, {member_info[1]}!")
                main_page = MainPage(member_info)

            else:
                admin_info = get_admin_info(username, password)
                messagebox.showinfo("Login Successful", f"Welcome Admin, {admin_info[1]}!")
                main_page = MainPage(admin_info, True)
            self.withdraw()
            main_page.mainloop()
            self.deiconify()
        else:
            messagebox.showerror("Login Failed", "Invalid username or password")

if __name__ == "__main__":
    app = LoginPage()
    app.mainloop()
```

c.Main Menu Page

Following login, the MainPage serves as the primary interface and offers options according to the kind of user (admin or member). It has buttons to examine books, issuebooks, request loans, take surveys, and, for administrators, manage members and books.
The MainPage class is implemented as a tk.Toplevel window.Along with buttons for other activities. It shows a welcome message with the user's name on it.Clicking on different buttons leads to different pages, including the MemberManagementPage, BookPage, ReservationPage, LoanRequestPage, and FeedbackPage.

The admin dashboard is a page that allows the administrator of the library to manage the library's collection, members, and other settings.

The "View Books" button would allow the administrator to view a list of all the books in the library's collection. The administrator could then add, edit, or delete books from the collection, as needed.

The "Manage Members" button would allow the administrator to view a list of all the library's members. The administrator could then add, edit, or delete members from the system, as needed. The administrator could also use this page to assign members to groups, such as students, faculty, or staff. This would allow the administrator to grant different permissions to different groups of members.

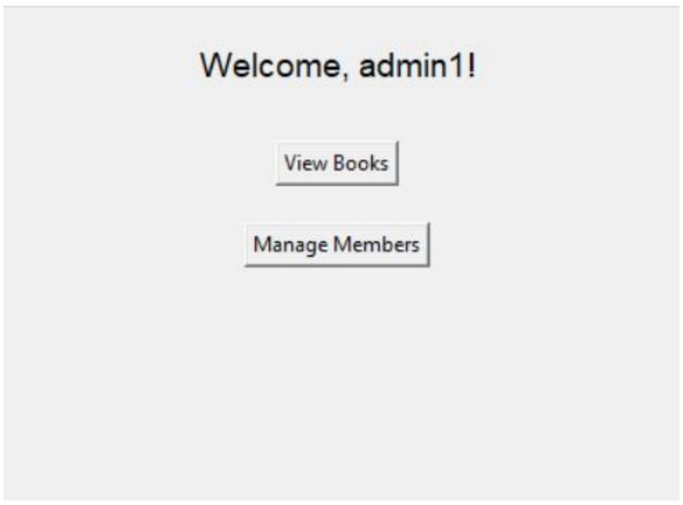


Figure 10:Admin main page

The member page is used by library members to manage their library accounts and activities.

- View Books: This button takes the member to a page where they can browse the library's catalog of books. The member can search for books by title, author, genre, or other criteria.

- Issue Book: This button allows the member to check out books from the library.
- Manage Loans: This button takes the member to a page where they can view their current loans, renew books, and pay fines.
- Take Survey: This button takes the member to a survey where they can provide feedback on the library's services and resources.

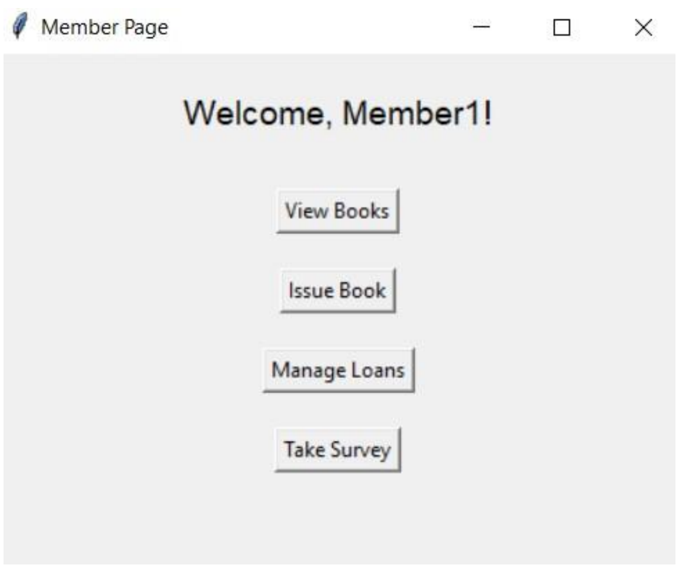


Figure 11:Member main page

1. **Class `MainPage`**:
- Inherits from `tk.Toplevel`, representing a window that pops up as a child window.
- Initializes the main window based on whether the logged-in user is an admin or a regular member.
2. **Constructor (`__init__`)**:
- Sets up the window properties such as title, size, and stores user information passed during initialization.
- Displays a welcome message using `tk.Label`.
3. **Buttons**:
- Creates different buttons based on user privileges:
- `"View Books"`: Allows any user (admin or member) to view available books.
- `"Issue Book"`: For regular members only, allowing them to reserve books for loaning.
- `"Manage Loans"`: For regular members, enables them to manage their current loans or loan requests.
- `"Take Survey"`: For regular members, facilitates feedback submission/survey.
- `"Manage Members"`: Specifically for admins to manage members' accounts.
4. **Button Functions**:
- `view_books()`: Redirects to the BookPage interface to view available books.
- `issue_book()`: Opens the ReservationPage for members to request a book for loaning.
- `manage_loans()`: Initiates the LoanRequestPage for members to manage their loans.
- `take_survey()`: Redirects to the FeedbackPage for members to provide feedback.
- `manage_member()`: Opens the MemberManagementPage, specifically designed for admins to manage members' accounts.
5. **Pages**:
- Each button click instantiates different pages:
- `BookPage`: For viewing available books.
- `ReservationPage`: For reserving a book for loaning.
- `LoanRequestPage`: For managing loan requests.
- `FeedbackPage`: For submitting feedback.
- `MemberManagementPage`: For admin access to manage member accounts

Main.py code

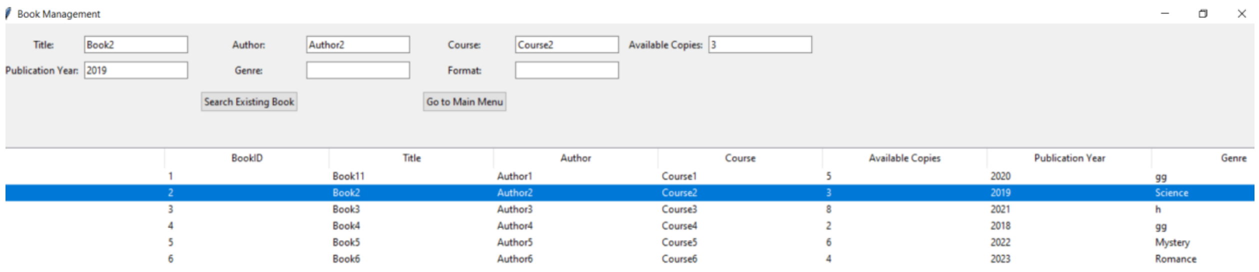
```
import tkinter as tk
from tkinter import messagebox
from book import BookPage
from reservation import ReservationPage
from loan import LoanRequestPage
from feedback import FeedbackPage
from member import MemberManagementPage

class MainPage(tk.Toplevel):
    def __init__(self, user, is_admin=False):
        super().__init__()
        self.is_admin = is_admin
        self.title("Member Page")
        self.geometry("400x300")

        self.user = user

        self.label_welcome = tk.Label(self, text=f"Welcome, {self.user[1]}!",
font=("Helvetica", 14))
        self.label_welcome.pack(pady=20)

        btn_view_books = tk.Button(self, text="View Books", command=self.view_books)
        btn_view_books.pack(pady=10)
        if not is_admin:
            btn_issue_book = tk.Button(self, text="Issue Book", command=self.issue_book)
            btn_issue_book.pack(pady=10)

            btn_manage_loans = tk.Button(self, text="Manage Loans",
command=self.manage_loans)
            btn_manage_loans.pack(pady=10)

            btn_take_survey = tk.Button(self, text="Take Survey", command=self.take_survey)
            btn_take_survey.pack(pady=10)
        if is_admin:
            btn_manage_ember = tk.Button(self, text="Manage Members",
command=self.manage_member)
            btn_manage_ember.pack(pady=11)

    def view_books(self):
        self.withdraw()

        book_page = BookPage(None, None, self, self.is_admin)
        # book_page.mainloop()



    def issue_book(self):
        reserve_page = ReservationPage(None, (self.user[0], ), self)

    def manage_loans(self):
        LoanRequestPage(None, (self.user[0], ), self)


    def take_survey(self):
        FeedbackPage(None, (self.user[0], ), self)

    def manage_member(self):
        MemberManagementPage(None, (self.user[0], ), self)
```

d.Action Page

1. Book Page

Users can view the books that are accessible on the BookPage. It comes with a treeview widget that shows details about the books and links to return to the main menu members can also search books based on different search criteria.The BookPage class initializes a window with a Treeview widget to show book information. There are buttons on it for searching and going back to the main menu. Admin user can also add and update books information based on user type the create and update actions are enabled or disabled.

Member Users:

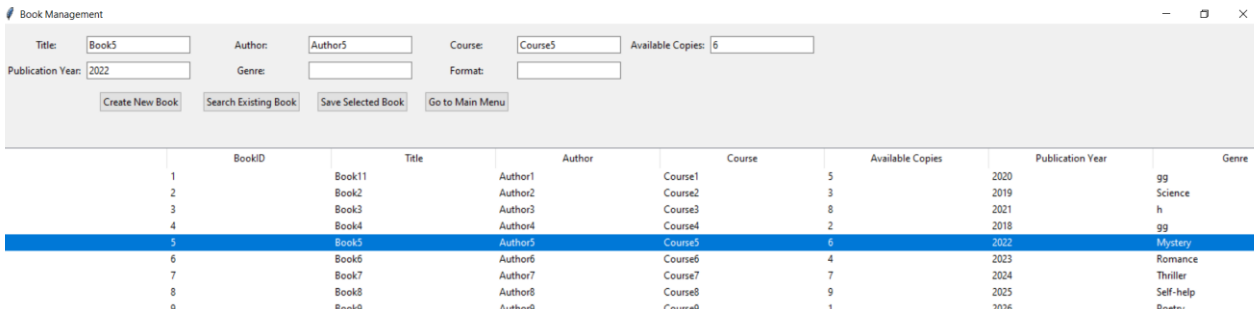


Title:	Book2	Author:	Author2	Course:	Course2	Available Copies:	3
Publication Year:	2019	Genre:		Format:			

BookID	Title	Author	Course	Available Copies	Publication Year	Genre
1	Book11	Author1	Course1	5	2020	gg
2	Book2	Author2	Course2	3	2019	Science
3	Book3	Author3	Course3	8	2021	h
4	Book4	Author4	Course4	2	2018	gg
5	Book5	Author5	Course5	6	2022	Mystery
6	Book6	Author6	Course6	4	2023	Romance

Figure 12:Member page

Admin users:



Title:	Book5	Author:	Author5	Course:	Course5	Available Copies:	6
Publication Year:	2022	Genre:		Format:			

BookID	Title	Author	Course	Available Copies	Publication Year	Genre
1	Book11	Author1	Course1	5	2020	gg
2	Book2	Author2	Course2	3	2019	Science
3	Book3	Author3	Course3	8	2021	h
4	Book4	Author4	Course4	2	2018	gg
5	Book5	Author5	Course5	6	2022	Mystery
6	Book6	Author6	Course6	4	2023	Romance
7	Book7	Author7	Course7	7	2024	Thriller
8	Book8	Author8	Course8	9	2025	Self-help
9	Book9	Author9	Course9	1	2026	Poetry

Figure 13:Admin Page

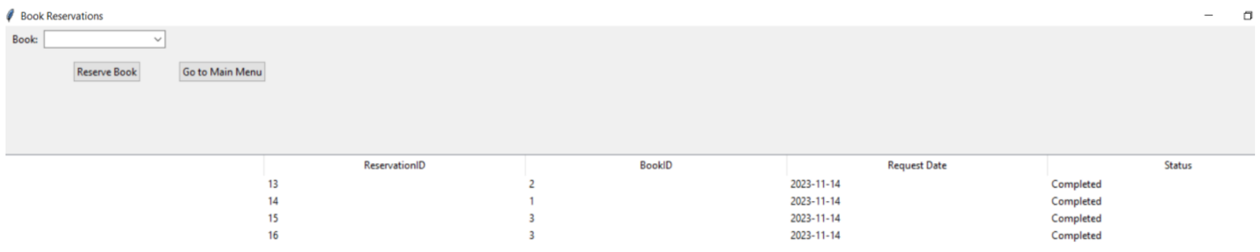
1. **Class `BookPage`**:
- Initializes a graphical interface (`tkinter`) to manage books in the system.
- It allows users (admin or regular user) to view, create, update, search, and delete book records.
2. **Constructor (`__init__`)**:
- Sets up the window, including the title and layout for displaying book details.
- Initializes a Treeview widget (`book_tree`) to display book records in a table-like format.
- Creates labels and entry fields for entering book details.
- Defines buttons for various book management operations such as creating, searching, deleting, and saving book details.
3. **Functionality**:
- `save_book`: Saves changes made to an existing book entry.
- `refresh_book_table`: Refreshes the book table with the updated records from the database.
- `create_book`: Adds a new book entry to the system.
- `search_book`: Searches for books based on specified criteria.
- `filter_book_table`: Filters the book table based on search criteria.
- `delete_book`: Deletes a selected book entry.
- `get_book_details`: Retrieves details of a selected book for editing.
- `main_page`: Navigates back to the main menu.
4. **Buttons & Actions**:
- The buttons trigger corresponding actions, such as creating, saving, searching, and deleting book entries.
- Some buttons (`button_delete_book`) are commented out, possibly because they are under development or not intended for immediate use.
5. **Integration**:
- The class integrates with a database module (`db`) that includes functions for fetching, updating, creating, and deleting book records.
6. **Usage**:
- The `if __name__ == "__main__":` block creates an instance of the `BookPage` class within a `tkinter.Tk()` window, initiating the Book Management interface when the script is run independently.

Book.py code:

```
import tkinter as tk
from tkinter import ttk, messagebox
from db import get_book, get_books, update_book, create_book, delete_book

class BookPage:
    def __init__(self, root, user,main_menu, is_admin=False):
        if not root:
            root = tk.Tk()
        self.main_menu = main_menu
        self.root = root
        self.is_admin = is_admin
        self.root.title("Book Management")
        self.root.state("zoomed")
        self.book_tree = ttk.Treeview(root, columns=("BookID", "Title", "Author", "Course",
"AvailableCopies", "PublicationYear", "Genre", "Format"))
        self.book_tree.heading("BookID", text="BookID")
        self.book_tree.heading("Title", text="Title")
        self.book_tree.heading("Author", text="Author")
        self.book_tree.heading("Course", text="Course")
        self.book_tree.heading("AvailableCopies", text="Available Copies")
        self.book_tree.heading("PublicationYear", text="Publication Year")
        self.book_tree.heading("Genre", text="Genre")
        self.book_tree.heading("Format", text="Format")
        self.book_tree.pack(pady=150, fill=tk.BOTH, expand=True)
        self.refresh_book_table()
        self.book_tree.bind("<<TreeviewSelect>>", self.get_book_details)
        self.form_container = ttk.Frame(root)
        self.form_container.pack(pady=10)
        self.label_title = ttk.Label(self.form_container, text="Title:")
        self.label_author = ttk.Label(self.form_container, text="Author:")
        self.label_course = ttk.Label(self.form_container, text="Course:")
        self.label_available_copies = ttk.Label(self.form_container, text="Available Copies:")
        self.label_publication_year = ttk.Label(self.form_container, text="Publication Year:")
        self.label_genre = ttk.Label(self.form_container, text="Genre:")
        self.label_format = ttk.Label(self.form_container, text="Format:")
        self.entry_title = ttk.Entry(self.form_container)
        self.entry_author = ttk.Entry(self.form_container)
        self.entry_course = ttk.Entry(self.form_container)
        self.entry_available_copies = ttk.Entry(self.form_container)
        self.entry_publication_year = ttk.Entry(self.form_container)
        self.entry_genre = ttk.Entry(self.form_container)
        self.entry_format = ttk.Entry(self.form_container)
        self.button_create_book = ttk.Button(self.form_container, text="Create New Book",
command=self.create_book)
        self.button_search = ttk.Button(self.form_container, text="Search Existing Book",
command=self.search_book)
        self.button_delete_book = ttk.Button(self.form_container, text="Delete Selected Book",
command=self.delete_book)
        self.button_save_book = ttk.Button(self.form_container, text="Save Selected Book",
command=self.save_book)
        self.button_back = ttk.Button(self.form_container, text="Go to Main Menu",
command=self.main_page)
        self.label_title.grid(row=0, column=0, padx=(10, 5), pady=5)
        self.entry_title.grid(row=0, column=1, padx=(0, 5), pady=5)
        self.label_author.grid(row=0, column=2, padx=5, pady=5)
        self.entry_author.grid(row=0, column=3, padx=(0, 5), pady=5)
        self.label_course.grid(row=0, column=4, padx=5, pady=5)
        self.entry_course.grid(row=0, column=5, padx=(0, 5), pady=5)
        self.label_available_copies.grid(row=0, column=6, padx=5, pady=5)
        self.entry_available_copies.grid(row=0, column=7, padx=(0, 5), pady=5)
        self.label_publication_year.grid(row=1, column=0, padx=5, pady=5)
        self.entry_publication_year.grid(row=1, column=1, padx=(0, 5), pady=5)
        self.label_genre.grid(row=1, column=2, padx=5, pady=5)
        self.entry_genre.grid(row=1, column=3, padx=(0, 5), pady=5)
        self.label_format.grid(row=1, column=4, padx=5, pady=5)
        self.entry_format.grid(row=1, column=5, padx=(0, 5), pady=5)
        if self.is_admin:
            self.button_create_book.grid(row=2, column=1, padx=10, pady=10)
            self.button_save_book.grid(row=2, column=3, padx=10, pady=10)
        self.button_search.grid(row=2, column=2, padx=10, pady=10)
        # self.button_delete_book.grid(row=2, column=3, padx=10, pady=10)

        self.button_back.grid(row=2, column=4, padx=10, pady=10)
        self.root.update_idletasks()
        self.form_container.place(x=0, y=10)

    def save_book(self):
        selected_item = self.book_tree.selection()
        if not selected_item:
            messagebox.showerror("Error", "Select a book to edit.")
            return
        book_id = self.book_tree.item(selected_item, "values")[0]
        title = self.entry_title.get()
        author = self.entry_author.get()
        course = self.entry_course.get()
        available_copies = self.entry_available_copies.get()
        publication_year = self.entry_publication_year.get()
        genre = self.entry_genre.get()
        book_format = self.entry_format.get()
        if not title or not author or not course or not available_copies or not publication_year or
not genre or not book_format:
            messagebox.showerror("Error", "Enter values in all the form fields.")
            return

        try:
            update_book(book_id, title, author, course, available_copies, publication_year, genre,
book_format)
            messagebox.showinfo("Success", "Book updated successfully")
            self.refresh_book_table()
        except Exception:
            messagebox.showerror("Error", "Failed to update book.")
    def refresh_book_table(self):
        for item in self.book_tree.get_children():
            self.book_tree.delete(item)
        books = get_books()
        for book in books:
            values = [book.get(column, "") for column in ("book_id", "title", "author", "course",
"available_copies", "publication_year", "genre", "format")]
            self.book_tree.insert("", "end", values=values)

    def create_book(self):
        title = self.entry_title.get()
        author = self.entry_author.get()
        course = self.entry_course.get()
        available_copies = self.entry_available_copies.get()
        publication_year = self.entry_publication_year.get()
        genre = self.entry_genre.get()
        book_format = self.entry_format.get()
        if not title or not author or not course or not available_copies or not publication_year or not genre or not book_format:
            messagebox.showerror("Error", "Enter all form fields.")
            return
        try:
            create_book(title, author, course, available_copies, publication_year, genre, book_format)
            messagebox.showinfo("Success", "Book created successfully")
            self.refresh_book_table()
        except Exception:
            messagebox.showerror("Error", "Failed to create book.")

    def search_book(self):
        search_criteria = {
            "title": self.entry_title.get(),
            "author": self.entry_author.get(),
            "course": self.entry_course.get(),
            "available_copies": self.entry_available_copies.get(),
            "publication_year": int(self.entry_publication_year.get()),
            "genre": self.entry_genre.get(),
            "format": self.entry_format.get()
        }
        search_criteria = {key: value for key, value in search_criteria.items() if value}
        self.filter_book_table(search_criteria)
    def filter_book_table(self, search_params):
        for item in self.book_tree.get_children():
            self.book_tree.delete(item)
        books = get_books()
        for book in books:
            show = all(book.get(key) == value for key, value in search_params.items())
            if show:
                values = [book.get(column, "") for column in ("book_id", "title", "author", "course", "available_copies", "publication_year", "genre", "format")]
                self.book_tree.insert("", "end", values=values)

    def delete_book(self):
        selected_item = self.book_tree.selection()
        if not selected_item:
            messagebox.showerror("Error", "Select a book to delete.")
            return

        book_id = int(self.book_tree.item(selected_item, "values")[0])
        title = self.book_tree.item(selected_item, "values")[1]

        confirmation = messagebox.askyesno("Confirmation", f"Are you sure you want to
delete the book '{title}'?")
        if not confirmation:
            return
        try:
            delete_book(book_id)
            messagebox.showinfo("Success", "Book deleted successfully")
            self.refresh_book_table()
        except Exception:
            messagebox.showerror("Error", "Failed to delete book.")

    def get_book_details(self, event):
        selected_item = self.book_tree.selection()
        if not selected_item:
            return
        book_id = self.book_tree.item(selected_item, "values")[0]
        book = get_book(book_id)
        self.entry_title.delete(0, tk.END)
        self.entry_title.insert(0, book["title"])
        self.entry_author.delete(0, tk.END)
        self.entry_author.insert(0, book["author"])
        self.entry_course.delete(0, tk.END)
        self.entry_course.insert(0, book["course"])
        self.entry_available_copies.delete(0, tk.END)
        self.entry_available_copies.insert(0, book["available_copies"])
        self.entry_publication_year.delete(0, tk.END)
        self.entry_publication_year.insert(0, book["publication_year"])
        self.entry_genre.delete(0, tk.END)
        self.entry_genre.insert(0, book["genre"])
        self.entry_format.delete(0, tk.END)
        self.entry_format.insert(0, book["format"])
    def main_page(self):
        self.root.withdraw()
        self.main_menu.deiconify()

if __name__ == "__main__":
    root = tk.Tk()
    book_management_window = BookPage(root, (1,), None)
    root.mainloop()
```

2. Reservation Page

Users can make book reservations on the ReservationPage. It has buttons for making reservations and returning to the main menu, as well as a treeview displaying reservation details and a dropdown for choosing a book. To show reservation information, a window is initialized with a Treeview widget by the ReservationPage class. It has buttons for making reservations and going back to the main menu in addition to a dropdown menu for choosing a book to reserve. The reservation details that are shown are updated via the refresh_reservation_table method.



ReservationID	BookID	Request Date	Status
13	2	2023-11-14	Completed
14	1	2023-11-14	Completed
15	3	2023-11-14	Completed
16	3	2023-11-14	Completed

Figure 14:Reservation Page

1. **Class `ReservationPage`**:
- Initializes an interface using `tkinter` for users to view their book reservations and make new reservations.
- Allows users to see their existing reservations in a Treeview widget (`reservation_tree`).
2. **Constructor (`__init__`)**:
- Sets up the window title and layout for displaying reservation details.
- Creates a Treeview widget to show reservation records in a tabular format.
- Defines a frame (`form_container`) for placing labels, dropdowns, and buttons related to reservations.
3. **Functionality**:

- `get_book_options`: Fetches available book options from the database for the user to choose from.
- `reserve_book`: Handles the process of making a reservation. It fetches the selected book's ID and creates a reservation record for the user.
- `refresh_reservation_table`: Refreshes the reservation table with updated records from the database.
- `main_page`: Navigates back to the main menu from the reservation page.

4. **Buttons & Actions**:

- The `button_reserve_book` triggers the `reserve_book` function to make a new reservation.
- The `button_back` navigates the user back to the main menu.

5. **Integration with Database**:

- The class integrates with a database module (`db`) that includes functions for fetching reservations, book options, getting book IDs by names, and creating new reservations.

6. **Usage**:

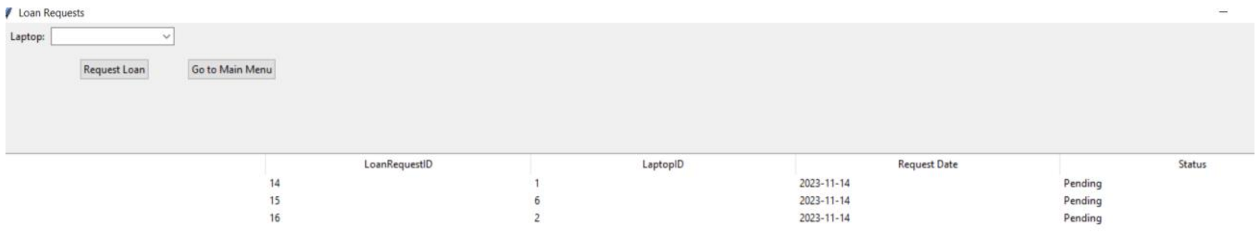
- The `if __name__ == "__main__":` block creates an instance of the `ReservationPage` class within a `tkinter.Tk()` window, initiating the Book Reservation interface when the script is run independently.

Reservation.py code:

```
 import tkinter as tk
from tkinter import ttk, messagebox
from db import get_reservations_for_member, create_reservation, get_book_options,
get_book_id_by_name
from datetime import date

class ReservationPage:
    def __init__(self, root, user, main_menu):
        if not root:
            root = tk.Tk()
        self.main_menu = main_menu
        self.root = root
        self.user = user
        self.root.title("Book Reservations")
        self.root.state("zoomed")
        self.reservation_tree = ttk.Treeview(root, columns=("ReservationID", "BookID",
"RequestDate", "Status"))
        self.reservation_tree.heading("ReservationID", text="ReservationID")
        self.reservation_tree.heading("BookID", text="BookID")
        self.reservation_tree.heading("RequestDate", text="Request Date")
        self.reservation_tree.heading("Status", text="Status")
        self.reservation_tree.pack(pady=150, fill=tk.BOTH, expand=True)
        self.refresh_reservation_table()
        self.form_container = ttk.Frame(root)
        self.form_container.pack(pady=10)
        self.label_book = ttk.Label(self.form_container, text="Book:")
        self.book_options = self.get_book_options()
        self.book_var = tk.StringVar(self.form_container)
        self.book_dropdown = ttk.Combobox(self.form_container, textvariable=self.book_var,
values=self.book_options)

        self.button_reserve_book = ttk.Button(self.form_container, text="Reserve Book",
command=self.reserve_book)
        self.button_back = ttk.Button(self.form_container, text="Go to Main Menu",
command=self.main_page)
        self.label_book.grid(row=0, column=0, padx=(10, 5), pady=5)
        self.book_dropdown.grid(row=0, column=1, padx=(0, 5), pady=5)
        self.button_reserve_book.grid(row=1, column=1, padx=10, pady=10)
        self.button_back.grid(row=1, column=2, padx=10, pady=10)

        self.root.update_idletasks()
        self.form_container.place(x=0, y=0)

    def get_book_options(self):
        books = get_book_options()
        books_list = []
        for book in books:
            books_list.append(f"{book['title']}")
        return books_list

    def reserve_book(self):
        book_name = self.book_var.get()
        if not book_name:
            messagebox.showerror("Error", "Please select a book to reserve.")
            return

        try:
            book_id = get_book_id_by_name(book_name)
            create_reservation(book_id, self.user[0], date.today(), "Completed")

            messagebox.showinfo("Success", "Book reserved successfully!")
            self.refresh_reservation_table()

        except Exception as e:
            messagebox.showerror("Error", f"Failed to reserve book: {str(e)}")

    def refresh_reservation_table(self):
        for item in self.reservation_tree.get_children():
            self.reservation_tree.delete(item)

        reservations = get_reservations_for_member(self.user[0])
        for reservation in reservations:
            values = [reservation.get(column, "") for column in ("reservation_id", "book_id", "request_date", "status")]
            self.reservation_tree.insert("", "end", values=values)

    def main_page(self):
        self.root.withdraw()
        self.main_menu.deiconify()

if __name__ == "__main__":
    root = tk.Tk()
    reservation_window = ReservationPage(root, (1,), None)
    root.mainloop()
```

3. Loan Request Page

Users can apply for laptop loans on the LoanRequestPage. It has buttons to seek a loan and return to the main menu, as well as a treeview displaying information on loan requests. To show loan request details, a window with a Treeview widget is initialized by the LoanRequestPage class. It has buttons to request a loan and go back to the main menu, as well as a dropdown menu to choose which laptop to loan. The loan request data that is shown is updated via the refresh_loan_request_table method.



LoanRequestID	LaptopID	Request Date	Status
14	1	2023-11-14	Pending
15	6	2023-11-14	Pending
16	2	2023-11-14	Pending

Figure 15:Loan request page

1. **Class `LoanRequestPage`**:
- Initializes a graphical interface (`tkinter`) for users to view their loan requests and make new requests for laptops.
- Allows users to see their existing loan requests in a Treeview widget (`loan_request_tree`).
2. **Constructor (`__init__`)**:
- Sets up the window title and layout for displaying loan request details.
- Creates a Treeview widget to show loan request records in a tabular format.
- Defines a frame (`form_container`) for placing labels, dropdowns, and buttons related to loan requests.
3. **Functionality**:
- `get_laptop_options`: Fetches available laptop options from the database for the user to choose from.
- `request_loan`: Handles the process of making a loan request. It fetches the selected laptop's ID and creates a loan request record for the user.
- `refresh_loan_request_table`: Refreshes the loan request table with updated records from the database.
- `main_page`: Navigates back to the main menu from the loan request page.
4. **Buttons & Actions**:
- The `button_request_loan` triggers the `request_loan` function to make a new loan request.
- The `button_back` navigates the user back to the main menu.
5. **Integration with Database**:
- The class integrates with a database module (`db`) that includes functions for fetching loan requests, laptop options, getting laptop IDs by names, and creating new loan requests.
6. **Usage**:
- The `if __name__ == "__main__":` block creates an instance of the `LoanRequestPage` class within a `tkinter.Tk()` window, initiating the Loan Request interface when the script is run independently.

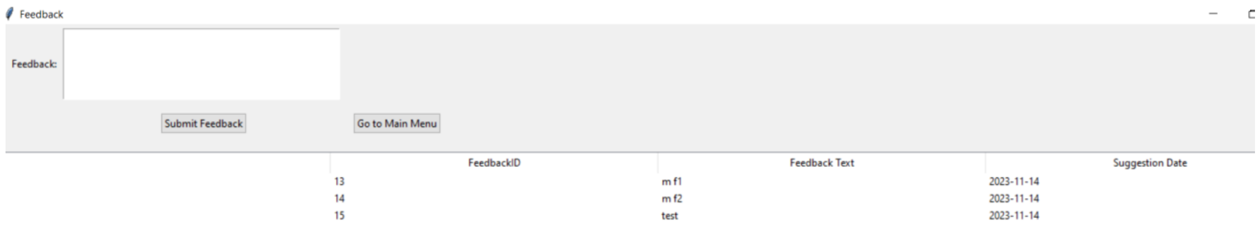
Loan.py code:

```
import tkinter as tk
from tkinter import ttk, messagebox
from db import get_loan_requests_for_member, create_loan_request, get_laptop_options,
get_laptop_id_by_name
from datetime import date

class LoanRequestPage:
    def __init__(self, root, user, main_menu):
        if not root:
            root = tk.Tk()
        self.main_menu = main_menu
        self.root = root
        self.user = user
        self.root.title("Loan Requests")
        self.root.state("zoomed")
        self.loan_request_tree = ttk.Treeview(root, columns=("LoanRequestID", "LaptopID",
"RequestDate", "Status"))
        self.loan_request_tree.heading("LoanRequestID", text="LoanRequestID")
        self.loan_request_tree.heading("LaptopID", text="LaptopID")
        self.loan_request_tree.heading("RequestDate", text="Request Date")
        self.loan_request_tree.heading("Status", text="Status")
        self.loan_request_tree.pack(pady=150, fill=tk.BOTH, expand=True)

        self.refresh_loan_request_table()

        self.form_container = ttk.Frame(root)
        self.form_container.pack(pady=10)

        self.label_laptop = ttk.Label(self.form_container, text="Laptop:")
        self.laptop_options = self.get_laptop_options()
        self.laptop_var = tk.StringVar(self.form_container)
        self.laptop_dropdown = ttk.Combobox(self.form_container,
textvariable=self.laptop_var, values=self.laptop_options)

        self.button_request_loan = ttk.Button(self.form_container, text="Request Loan",
command=self.request_loan)
        self.button_back = ttk.Button(self.form_container, text="Go to Main Menu",
command=self.main_page)
```

```
        self.label_laptop.grid(row=0, column=0, padx=(10, 5), pady=5)
        self.laptop_dropdown.grid(row=0, column=1, padx=(0, 5), pady=5)
        self.button_request_loan.grid(row=1, column=1, padx=10, pady=10)
        self.button_back.grid(row=1, column=2, padx=10, pady=10)

        self.root.update_idletasks()
        self.form_container.place(x=0, y=0)

    def get_laptop_options(self):
        laptops = get_laptop_options()
        laptops_list = []
        for laptop in laptops:
            laptops_list.append(f"{laptop['company_name']} - {laptop['processor_type']}")
        return laptops_list

    def request_loan(self):
        laptop_name = self.laptop_var.get()
        if not laptop_name:
            messagebox.showerror("Error", "Please select a laptop to request a loan.")
            return

        try:
            laptop_id = get_laptop_id_by_name(laptop_name)
            create_loan_request(self.user[0], laptop_id, date.today(), "Pending")

            messagebox.showinfo("Success", "Loan request submitted successfully!")
            self.refresh_loan_request_table()

        except Exception as e:
            messagebox.showerror("Error", f"Failed to submit loan request: {str(e)}")

    def refresh_loan_request_table(self):
        for item in self.loan_request_tree.get_children():
            self.loan_request_tree.delete(item)

        loan_requests = get_loan_requests_for_member(self.user[0])
        for loan_request in loan_requests:
            values = [loan_request.get(column, "") for column in ("loan_request_id", "laptop_id",
"request_date", "status")]
            self.loan_request_tree.insert("", "end", values=values)

    def main_page(self):
        self.root.withdraw()
        self.main_menu.deiconify()

if __name__ == "__main__":
    root = tk.Tk()
    loan_request_window = LoanRequestPage(root, (1,), None)
    root.mainloop()
```

4. Feedback Page

Users can provide feedback on the FeedbackPage. Along with a text entry area, a submit button, and a button to go back to the main menu, it shows the feedback that has already been left. In order to display feedback data, a window is initialized with a Treeview widget by the FeedbackPage class. It has three buttons one to submit, one to go back to the main menu, and one text field for comments. The feedback data that is presented is updated using the refresh_feedback_table method.



FeedbackID	Feedback Text	Suggestion Date
13	m f1	2023-11-14
14	m f2	2023-11-14
15	test	2023-11-14

Figure 16:Feedback Page

1. **Class `FeedbackPage`**:
- Initializes a graphical interface (`tkinter`) allowing users to submit feedback and view their previously submitted feedback.
- Displays existing feedback records in a Treeview widget (`feedback_tree`).
2. **Constructor (`__init__`)**:
- Sets up the window title and layout for displaying feedback details.
- Creates a Treeview widget to show feedback records in a tabular format.

- Defines a frame (`form_container`) for placing labels, text input, and buttons related to feedback submission and display.

3. **Functionality**:
- `submit_feedback`: Handles the process of submitting feedback. It retrieves the entered feedback text and creates a new feedback record for the user.
- `refresh_feedback_table`: Refreshes the feedback table with updated records from the database.
- `main_page`: Navigates back to the main menu from the feedback page.

4. **Buttons & Actions**:
- The `button_submit_feedback` triggers the `submit_feedback` function to create a new feedback entry.
- The `button_back` navigates the user back to the main menu.

5. **Integration with Database**:
- The class integrates with a database module (`db`) that includes functions for creating feedback records and fetching feedback for a specific member.

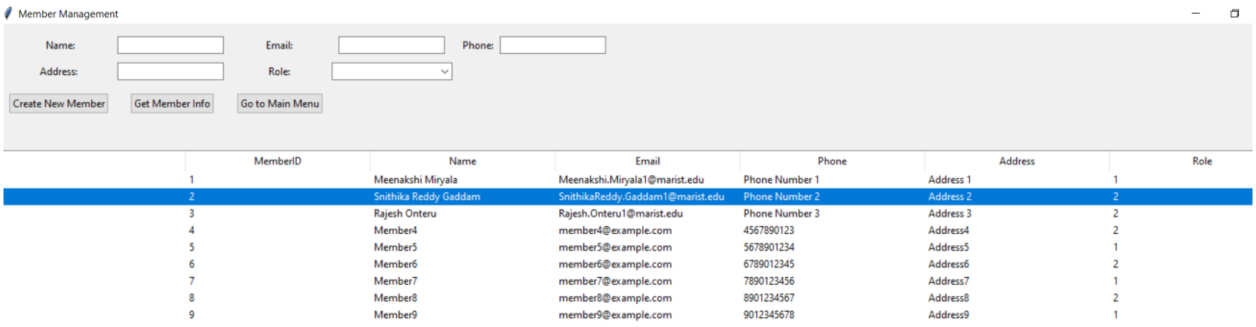
6. **Usage**:
- The `if __name__ == "__main__":` block creates an instance of the `FeedbackPage` class within a `tkinter.Tk()` window, initiating the Feedback interface when the script is run independently.

Feedback.py code:

```
import tkinter as tk
from tkinter import ttk, messagebox
from datetime import date
from db import create_feedback, get_member_info, get_feedback_for_member

class FeedbackPage:
    def __init__(self, root, user, main_menu):
        if not root:
            root = tk.Tk()
        self.main_menu = main_menu
        self.root = root
        self.user = user
        self.root.title("Feedback")
        self.root.state("zoomed")

        self.feedback_tree = ttk.Treeview(root, columns=("FeedbackID", "FeedbackText",
"SuggestionDate"))
        self.feedback_tree.heading("FeedbackID", text="FeedbackID")
        self.feedback_tree.heading("FeedbackText", text="Feedback Text")
        self.feedback_tree.heading("SuggestionDate", text="Suggestion Date")
        self.feedback_tree.pack(pady=150, fill=tk.BOTH, expand=True)

        self.refresh_feedback_table()
        self.form_container = ttk.Frame(root)
        self.form_container.pack(pady=10)
        self.label_feedback = ttk.Label(self.form_container, text="Feedback:")
        self.text_feedback = tk.Text(self.form_container, height=5, width=40)
        self.button_submit_feedback = ttk.Button(self.form_container, text="Submit Feedback",
command=self.submit_feedback)
        self.button_back = ttk.Button(self.form_container, text="Go to Main Menu",
command=self.main_page)
        self.label_feedback.grid(row=0, column=0, padx=(10, 5), pady=5)
        self.text_feedback.grid(row=0, column=1, padx=(0, 5), pady=5)
        self.button_submit_feedback.grid(row=1, column=1, padx=10, pady=10)
        self.button_back.grid(row=1, column=2, padx=10, pady=10)

        self.root.update_idletasks()
        self.form_container.place(x=0, y=0)

    def submit_feedback(self):
        feedback_text = self.text_feedback.get("1.0", tk.END).strip()
        if not feedback_text:
            messagebox.showerror("Error", "Please enter your feedback.")
            return

        try:
            create_feedback(self.user[0], feedback_text, date.today())

            messagebox.showinfo("Success", "Feedback submitted successfully!")
            self.refresh_feedback_table()

        except Exception as e:
            messagebox.showerror("Error", f"Failed to submit feedback: {str(e)}")

    def refresh_feedback_table(self):
        for item in self.feedback_tree.get_children():
            self.feedback_tree.delete(item)
        feedbacks = get_feedback_for_member(self.user[0])
        for feedback in feedbacks:
            values = [feedback.get(column, "") for column in ("feedback_id", "feedback_text",
"suggestion_date")]
            self.feedback_tree.insert("", "end", values=values)

    def main_page(self):
        self.root.withdraw()
        self.main_menu.deiconify()

if __name__ == "__main__":
    root = tk.Tk()
    feedback_window = FeedbackPage(root, (1,), None)
    root.mainloop()
```

5. Member Management Page

Admin users manage members on the MemberManagementPage. It has form fields for adding new members, search a meber with a search criteria a treeview showing member details, action buttons, and a button to go back to the main menu. To show member information, a window is initialized with a Treeview widget by the MemberManagementPage class. It has buttons for activities, a button to go back to the main menu, and form fields for adding new members. The member data that is presented is updated using the refresh_member_table method.



MemberID	Name	Email	Phone	Address	Role
1	Meenakshi Miryala	Meenakshi.Miryala1@marist.edu	Phone Number 1	Address 1	1
2	Snithika Reddy Gaddam	SnithikaReddy.Gaddam1@marist.edu	Phone Number 2	Address 2	2
3	Rajesh Onteru	Rajesh.Onteru1@marist.edu	Phone Number 3	Address 3	2
4	Member4	member4@example.com	4567890123	Address4	2
5	Member5	member5@example.com	5678901234	Address5	1
6	Member6	member6@example.com	6789012345	Address6	2
7	Member7	member7@example.com	7890123456	Address7	1
8	Member8	member8@example.com	8901234567	Address8	2
9	Member9	member9@example.com	9012345678	Address9	1

Figure 17:Member Management Page

1. **Class `MemberManagementPage`**:
- Initializes a window for managing library members, displaying member information in a table format.
- Provides functionalities to create new members, retrieve member information by name, and navigate back to the main menu.
2. **Constructor (`__init__`)**:
- Sets up the window title and layout to display member details in a `Treeview` widget (`member_tree`).
- Creates a frame (`form_container`) to house labels, entry fields, dropdowns, and buttons related to member management.
3. **Functionality**:
- `create_member`: Creates a new member with provided details (name, email, phone, address, role) and displays a success message upon successful creation.
- `get_member_info`: Retrieves member information based on the entered member name and displays it in the member table.
- `display_member_info`: Displays member information in the `member_tree` based on the provided member details.
- `refresh_member_table`: Refreshes the member table with updated member records fetched from the database.
- `get_role_names` and `get_role_id_by_name`: Helper functions to fetch role-related information for member creation.
4. **Buttons & Actions**:
- The `button_create_member` triggers the `create_member` function to add a new member.
- The `button_get_member` calls the `get_member_info` function to fetch member information.
- The `button_back` navigates the user back to the main menu.
5. **Integration with Database**:
- Integrates with a database module (`db`) that includes functions for member management, such as creating members, fetching member information, and retrieving roles.
6. **Usage**:
- The `if __name__ == "__main__":` block creates an instance of the `MemberManagementPage` class within a `tkinter.Tk()` window, initializing the member management interface when the script is run independently.

Member.py code:

```
import tkinter as tk
from tkinter import ttk, messagebox
from db import create_member, get_member_info, get_roles, get_member_info_by_name,
get_members
```

```
class MemberManagementPage:
    def __init__(self, root, user, main_menu):
        if not root:
            root = tk.Tk()
        self.main_menu = main_menu
        self.root = root
        self.root.title("Member Management")
        self.root.state("zoomed")
        self.member_tree = ttk.Treeview(root, columns=("MemberID", "Name", "Email",
"Phone", "Address", "Role"))
        self.member_tree.heading("MemberID", text="MemberID")
        self.member_tree.heading("Name", text="Name")
        self.member_tree.heading("Email", text="Email")
        self.member_tree.heading("Phone", text="Phone")
        self.member_tree.heading("Address", text="Address")
        self.member_tree.heading("Role", text="Role")
        self.member_tree.pack(pady=150, fill=tk.BOTH, expand=True)
        self.refresh_member_table()
        self.form_container = ttk.Frame(root)
        self.form_container.pack(pady=10)
        self.label_name = ttk.Label(self.form_container, text="Name:")
        self.label_email = ttk.Label(self.form_container, text="Email:")
        self.label_phone = ttk.Label(self.form_container, text="Phone:")
        self.label_address = ttk.Label(self.form_container, text="Address:")
        self.label_role = ttk.Label(self.form_container, text="Role:")
        self.entry_name = ttk.Entry(self.form_container)
        self.entry_email = ttk.Entry(self.form_container)
        self.entry_phone = ttk.Entry(self.form_container)
        self.entry_address = ttk.Entry(self.form_container)
        self.role_var = tk.StringVar()
        self.role_dropdown = ttk.Combobox(self.form_container, textvariable=self.role_var,
values=self.get_role_names())
        self.button_create_member = ttk.Button(self.form_container, text="Create New
Member", command=self.create_member)
        self.button_get_member = ttk.Button(self.form_container, text="Get Member Info",
command=self.get_member_info)
        self.button_back = ttk.Button(self.form_container, text="Go to Main Menu",
command=self.main_page)
```

```
        self.label_name.grid(row=0, column=0, padx=(10, 5), pady=5)
        self.entry_name.grid(row=0, column=1, padx=(0, 5), pady=5)
        self.label_email.grid(row=0, column=2, padx=5, pady=5)
        self.entry_email.grid(row=0, column=3, padx=(0, 5), pady=5)
        self.label_phone.grid(row=0, column=4, padx=5, pady=5)
        self.entry_phone.grid(row=0, column=5, padx=(0, 5), pady=5)
        self.label_address.grid(row=1, column=0, padx=5, pady=5)
        self.entry_address.grid(row=1, column=1, padx=(0, 5), pady=5)
        self.label_role.grid(row=1, column=2, padx=5, pady=5)
        self.role_dropdown.grid(row=1, column=3, padx=(0, 5), pady=5)
        self.button_create_member.grid(row=2, column=0, padx=10, pady=10)
        self.button_get_member.grid(row=2, column=1, padx=10, pady=10)
        self.button_back.grid(row=2, column=2, padx=10, pady=10)
        self.root.update_idletasks()
        self.form_container.place(x=0, y=10)

    def create_member(self):
        name = self.entry_name.get()
        email = self.entry_email.get()
        phone = self.entry_phone.get()
        address = self.entry_address.get()
        role_name = self.role_var.get()
        if not name or not email or not phone or not address:
            messagebox.showerror("Error", "Enter all form fields.")
            return
        try:
            role_id = self.get_role_id_by_name(role_name)
            if not role_id:
                role_id = 1
            create_member(name, email, phone, address,"password", role_id)
            messagebox.showinfo("Success", "Member created successfully")
            self.refresh_member_table()
        except Exception:
            messagebox.showerror("Error", "Failed to create member.")

    def get_member_info(self):
        name = self.entry_name.get()
        if not name:
            messagebox.showerror("Error", "Enter member name to get information.")
            return
        try:
            member_info = get_member_info_by_name(name)
            self.display_member_info(member_info)
        except Exception:
            messagebox.showerror("Error", "Failed to get member information.")

    def display_member_info(self, member_info):
        for item in self.member_tree.get_children():
            self.member_tree.delete(item)
        values = [member_info.get(column, "") for column in ("member_id", "name", "email", 
"phone_number", "address", "role_id")]
        self.member_tree.insert("", "end", values=values)

    def refresh_member_table(self):
        for item in self.member_tree.get_children():
            self.member_tree.delete(item)
        members = get_members()
        for member in members:
            values = [member.get(column, "") for column in ("member_id", "name", "email", 
"phone_number", "address", "role_id")]
            self.member_tree.insert("", "end", values=values)

    def get_role_names(self):
        roles = get_roles()
        return [role["role_name"] for role in roles]

    def get_role_id_by_name(self, role_name):
        roles = get_roles()
        for role in roles:
            if role["role_name"] == role_name:
                return role["role_id"]
        return None

    def main_page(self):
        self.root.withdraw()
        self.main_menu.deiconify()
```

```
if __name__ == "__main__":
    root = tk.Tk()
    member_management_window = MemberManagementPage(root, (1,), None)
    root.mainloop()
```

Section 11: Conclusion And Future Work

1.Conclusion

The Library Management System (LMS) project marks a significant achievement in enhancing library operations through automation and improved efficiency. This initiative, characterized by meticulous planning and execution, successfully replaces manual tasks with automated processes, minimizing errors and accelerating task completion. These enhancements contribute to a seamless and convenient library experience.

Moreover, the LMS integrates a robust reporting system that empowers administrators with valuable insights into library usage patterns and resource popularity. This data-driven approach facilitates informed decision-making, allowing the library to adapt to user needs effectively. Security and data integrity are prioritized, with features such as user authentication and regular data backups ensuring the protection of sensitive information.

In summary, the LMS has modernized and optimized library operations, positively impacting both staff and students. The system's efficiency, accessibility, and data-driven decision-making contribute to a dynamic and vital library resource for the community. As technology evolves, the LMS serves as a foundation for ongoing improvements, ensuring the library remains a responsive and indispensable hub for its users.

2.Future Work

In the future, enhancing our application with PostgreSQL and PostGIS for spatial data and Geopandas enables precise book location tracking. To optimize performance, adopting ReactJS or JavaScript for the GUI ensures dynamic and responsive interfaces. MongoDB's flexible schema and GeoJSON support make it an ideal database choice. This integrated approach, utilizing PostgreSQL, Geopandas, ReactJS, and MongoDB, promises a high-performance, scalable, and adaptable solution. This strategic amalgamation of technologies ensures efficient library resource management with advanced spatial analytics, offering users a seamless and engaging experience while accommodating evolving industry standards.

References

[1] Referred from http://evergreen.lib.in.us/eg/opac/home
[2] Song, Il-Yeol & Evans, Mary & Park, Eui Kyun. (1995). A Comparative Analysis of Entity-Relationship Diagrams. Journal of Computer and Software Engineering.
[3] https://vufind.org/vufind/