Assignment 1:

Analyze a given business scenario and create an ER diagram that includes entities, relationships, attributes, and cardinality. Ensure that the diagram reflects proper normalization up to the third normal form.

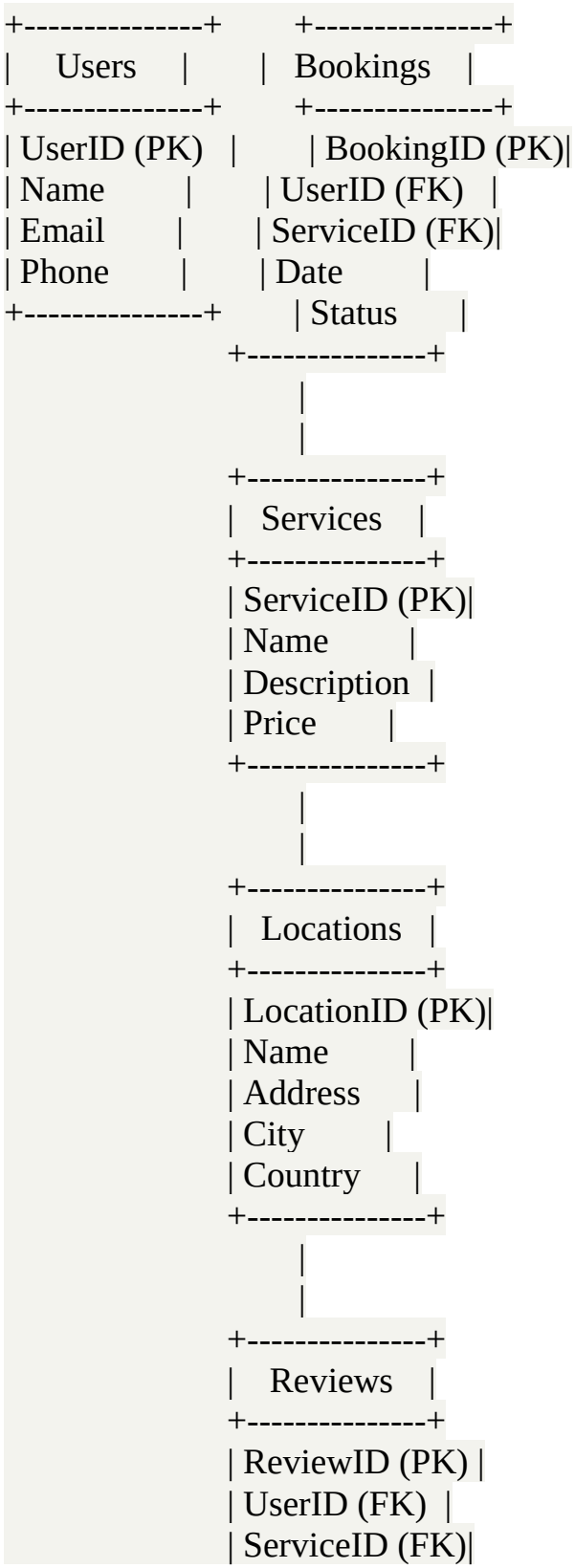
The key entities in a travel booking system are:

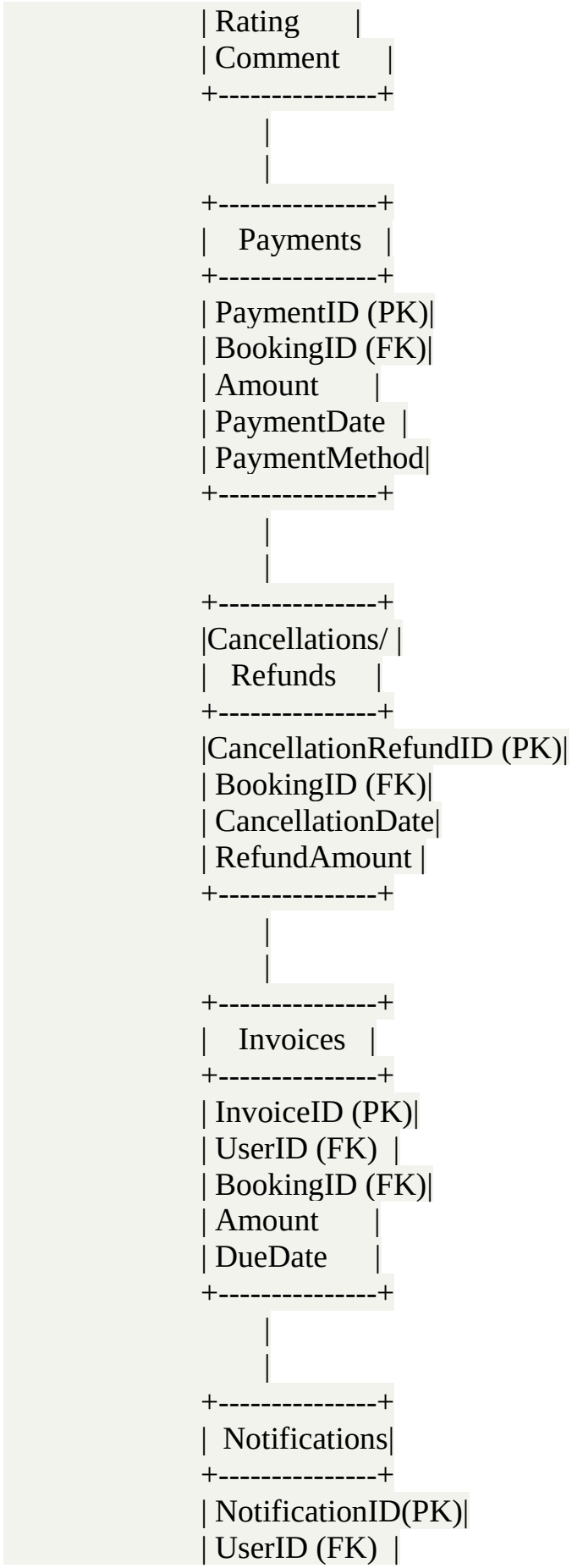
1. Users: Represents individuals who interact with the system. Attributes include UserID, Name, Email, Phone.
2. Bookings: Represents reservations made by users. Attributes include BookingID, UserID, ServiceID, Date, Status.
3. Services: Represents the travel-related services offered, such as flights, accommodations, activities. Attributes include ServiceID, Name, Description, Price.
4. Locations: Represents the physical locations of services, such as hotels and airports. Attributes include LocationID, Name, Address, City, Country.
5. Reviews: Represents user feedback on services. Attributes include ReviewID, UserID, ServiceID, Rating, Comment.
6. Payments: Represents the financial transactions for bookings. Attributes include PaymentID, BookingID, Amount, PaymentDate, PaymentMethod.
7. Cancellations/Refunds: Represents the process of cancelling bookings and issuing refunds. Attributes include CancellationRefundID, BookingID, CancellationDate, RefundAmount.
8. Invoices: Represents the billing information for bookings. Attributes include InvoiceID, UserID, BookingID, Amount, DueDate.
9. Notifications: Represents the communication between the system and users. Attributes include NotificationID, UserID, Message.

The key relationships between these entities are:

- Users can make many Bookings (Many-to-One)
- Bookings can be associated with many Services (Many-to-One)
- Users can book many Activities (Many-to-Many)
- Bookings can include many Accommodations (Many-to-One)
- Bookings can include many Flights (Many-to-One)

The ER diagram reflecting these entities, relationships, and attributes is as follows:




| Rating |
| Comment |
+----------+
|
|
+----------+
| Payments |
+----------+
| PaymentID (PK)|
| BookingID (FK)|
| Amount |
| PaymentDate |
| PaymentMethod|
+----------+
|
|
+----------+
|Cancellations/ |
| Refunds |
+----------+
|CancellationRefundID (PK)|
| BookingID (FK)|
| CancellationDate|
| RefundAmount |
+----------+
|
|
+----------+
| Invoices |
+----------+
| InvoiceID (PK)|
| UserID (FK) |
| BookingID (FK)|
| Amount |
| DueDate |
+----------+
|
|
+----------+
| Notifications|
+----------+
| NotificationID(PK)|
| UserID (FK) |

```
| Message      |
+--------------+
```

Assignment 2:

Design a database schema for a library system, including tables, fields, and constraints like NOT NULL, UNIQUE, and CHECK. Include primary and foreign keys to establish relationships between tables.

Here is a database schema design for a library system, including tables, fields, constraints, primary keys, and foreign keys to establish relationships between tables:

Library System Database Schema

Tables:

1. **Books**

   - book_id (INT, PRIMARY KEY)

   - title (VARCHAR(100), NOT NULL)

   - author (VARCHAR(100), NOT NULL)

   - publication_year (INT, CHECK(publication_year >= 1900 AND publication_year <= YEAR(CURRENT_DATE)))

   - genre (VARCHAR(50), NOT NULL)

   - available (BOOLEAN, NOT NULL, DEFAULT TRUE)

2. **Members**

- member_id (INT, PRIMARY KEY)
- first_name (VARCHAR(50), NOT NULL)
- last_name (VARCHAR(50), NOT NULL)
- email (VARCHAR(100), NOT NULL, UNIQUE)
- phone (VARCHAR(20), NOT NULL)
- registration_date (DATE, NOT NULL, DEFAULT CURRENT_DATE)

3. **Loans**
- loan_id (INT, PRIMARY KEY)
- book_id (INT, NOT NULL, FOREIGN KEY REFERENCES Books(book_id))
- member_id (INT, NOT NULL, FOREIGN KEY REFERENCES Members(member_id))
- loan_date (DATE, NOT NULL, DEFAULT CURRENT_DATE)
- due_date (DATE, NOT NULL)
- return_date (DATE)

4. **Fines**
- fine_id (INT, PRIMARY KEY)
- loan_id (INT, NOT NULL, FOREIGN KEY REFERENCES Loans(loan_id))
- amount (DECIMAL(10,2), NOT NULL, CHECK(amount > 0))
- payment_date (DATE)

Relationships:

1. **One-to-Many Relationship (Books to Loans):**
- One book can have multiple loans, but each loan is associated with only one book.

- The book_id field in the Loans table is a foreign key referencing the book_id primary key in the Books table.

2. **One-to-Many Relationship (Members to Loans):**

- One member can have multiple loans, but each loan is associated with only one member.

- The member_id field in the Loans table is a foreign key referencing the member_id primary key in the Members table.

3. **One-to-One Relationship (Loans to Fines):**

- Each loan can have a maximum of one fine associated with it, and each fine is associated with only one loan.

- The loan_id field in the Fines table is a foreign key referencing the loan_id primary key in the Loans table.

Constraints:

1. **NOT NULL Constraints:**

- Ensures that the specified fields cannot be left empty.

- Applied to title, author, genre, and available fields in the Books table.

- Applied to first_name, last_name, email, and phone fields in the Members table.

- Applied to book_id, member_id, and due_date fields in the Loans table.

- Applied to loan_id and amount fields in the Fines table.

2. **UNIQUE Constraint:**

- Ensures that the specified field has a unique value for each record.

- Applied to the email field in the Members table to enforce unique email addresses.

3. **CHECK Constraints:**

- Ensures that the specified condition is met for each record.

- Applied to the publication_year field in the Books table to ensure that the year is between 1900 and the current year.

- Applied to the amount field in the Fines table to ensure that the fine amount is greater than 0.

4. **Primary Keys:**

- Uniquely identifies each record in a table.

- book_id, member_id, loan_id, and fine_id fields are set as primary keys for their respective tables.

5. **Foreign Keys:**

- Establishes a relationship between two tables.

- The book_id and member_id fields in the Loans table are foreign keys referencing the book_id and member_id primary keys in the Books and Members tables, respectively.

- The loan_id field in the Fines table is a foreign key referencing the loan_id primary key in the Loans table.

This database schema provides a structured way to store and manage the data related to the library system, including books, members, loans, and fines. The relationships between tables and constraints ensure data integrity and consistency.

Assignment 3:

Explain the ACID properties of a transaction in your own words. Write SQL statements to simulate a transaction that includes locking and demonstrate different isolation levels to show concurrency control.

ACID Properties of Transactions

The ACID properties are a set of characteristics that ensure the reliability and consistency of database transactions. Here's an explanation of each property in my own words:

1. **Atomicity**: A transaction is treated as a single, indivisible unit of work. Either all the operations within the transaction are executed successfully, or none of them are. If any part of the transaction fails, the entire transaction is rolled back, and the database returns to its previous state before the transaction began.

2. **Consistency**: A transaction must maintain the database's integrity constraints and move it from one valid state to another. This means that if a transaction is executed alone, it should not violate any rules or constraints defined in the database schema.

3. **Isolation**: Transactions are executed independently without interference from other transactions. Isolation ensures that the intermediate results of a transaction are not visible to other transactions until the transaction is complete. This property allows multiple transactions to occur concurrently without compromising data integrity.

4. **Durability**: Once a transaction is committed, its effects are permanent and survive system failures, crashes, or restarts. The changes made by the transaction are stored in a way that ensures they will not be lost, even in the event of a system failure.

SQL Statements for Transaction Simulation

Here's an example of SQL statements that simulate a transaction, including locking and demonstrating different isolation levels for concurrency control:

```sql
-- Create a table for demonstration
CREATE TABLE accounts (
  id INT PRIMARY KEY,
  balance DECIMAL(10,2)
);

-- Insert some initial data
INSERT INTO accounts VALUES (1, 1000.00), (2, 500.00);

-- Begin a transaction with READ COMMITTED isolation level
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;

-- Lock the table for exclusive access
LOCK TABLES accounts WRITE;

-- Transfer funds between accounts
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;

-- Commit the transaction
```

```
COMMIT;
UNLOCK TABLES;

-- Demonstrate different isolation levels

-- Transaction 1 (READ UNCOMMITTED)
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT * FROM accounts;
ROLLBACK;

-- Transaction 2 (REPEATABLE READ)
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
SELECT * FROM accounts;
COMMIT;

-- Transaction 3 (SERIALIZABLE)
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
SELECT * FROM accounts;
COMMIT;
```

In this example:

1. A table named `accounts` is created with `id` as the primary key and `balance` as the balance amount.

2. Initial data is inserted into the `accounts` table.

3. A transaction is started with the `READ COMMITTED` isolation level.

4. The `accounts` table is locked for exclusive access using `LOCK TABLES`.

5. Funds are transferred between accounts using `UPDATE` statements.

6. The transaction is committed using `COMMIT`, and the table lock is released with `UNLOCK TABLES`.

7. Different isolation levels are demonstrated:

- `READ UNCOMMITTED`: Allows reading uncommitted data from other transactions.

- `REPEATABLE READ`: Guarantees that rows read during a transaction will be the same in subsequent reads within the same transaction.

- `SERIALIZABLE`: Ensures that transactions are executed in a serial manner, one after the other, to avoid any concurrency issues.

These SQL statements showcase how transactions work, including locking and different isolation levels, to maintain data consistency and concurrency control in a database.

Assignment 4:

Write SQL statements to CREATE a new database and tables that reflect the library schema you designed earlier. Use ALTER statements to modify the table structures and DROP statements to remove a redundant table.

Here are the SQL statements to create a new database, tables reflecting the library schema, alter table structures, and drop a redundant table:

Create a New Database and Tables:

```sql
-- Create a new database for the library system
CREATE DATABASE LibrarySystem;
USE LibrarySystem;

-- Create the Books table
CREATE TABLE Books (
  book_id INT PRIMARY KEY,
  title VARCHAR(100) NOT NULL,
  author VARCHAR(100) NOT NULL,
  publication_year INT CHECK(publication_year >= 1900 AND publication_year <= YEAR(CURRENT_DATE)),
  genre VARCHAR(50) NOT NULL,
  available BOOLEAN NOT NULL DEFAULT TRUE
);

-- Create the Members table
CREATE TABLE Members (
  member_id INT PRIMARY KEY,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  email VARCHAR(100) NOT NULL UNIQUE,
  phone VARCHAR(20) NOT NULL,
  registration_date DATE NOT NULL DEFAULT CURRENT_DATE
);

-- Create the Loans table
```

```
CREATE TABLE Loans (
  loan_id INT PRIMARY KEY,
  book_id INT NOT NULL,
  member_id INT NOT NULL,
  loan_date DATE NOT NULL DEFAULT CURRENT_DATE,
  due_date DATE NOT NULL,
  return_date DATE,
  FOREIGN KEY (book_id) REFERENCES Books(book_id),
  FOREIGN KEY (member_id) REFERENCES Members(member_id)
);

-- Create the Fines table
CREATE TABLE Fines (
  fine_id INT PRIMARY KEY,
  loan_id INT NOT NULL,
  amount DECIMAL(10,2) NOT NULL CHECK(amount > 0),
  payment_date DATE,
  FOREIGN KEY (loan_id) REFERENCES Loans(loan_id)
);
```

Alter Table Structures:

```sql
-- Add a new column 'isbn' to the Books table
ALTER TABLE Books
```

ADD COLUMN isbn VARCHAR(20);

-- Modify the data type of 'phone' column in Members table

ALTER TABLE Members

MODIFY COLUMN phone VARCHAR(15);

-- Rename the 'genre' column in Books table to 'category'

ALTER TABLE Books

CHANGE COLUMN genre category VARCHAR(50);

```

### Drop a Redundant Table:

```sql

-- Drop the redundant table 'OldBooks'

DROP TABLE OldBooks;

```

These SQL statements demonstrate creating a new database, defining tables based on the library schema, altering table structures by adding columns, modifying data types, and renaming columns, as well as dropping a redundant table from the database.

Assignment 5:

Demonstrate the creation of an index on a table and discuss how it improves query performance. Use a DROP INDEX statement to remove the index and analyze the impact on query execution.
```

Creating an Index on a Table:

To demonstrate the creation of an index on a table and discuss how it improves query performance, let's consider the `Books` table from the library system:

```sql
-- Create an index on the 'author' column of the Books table
CREATE INDEX idx_author ON Books(author);
```

How Index Improves Query Performance:

- **Faster Data Retrieval**: When a query filters or sorts data based on the indexed column (in this case, `author`), the database engine can quickly locate the relevant rows using the index. This results in faster data retrieval as the database doesn't have to scan the entire table.

- **Reduced Disk I/O**: Indexes store a sorted copy of the indexed column's values, making it easier for the database to locate specific data without reading the entire table. This reduces disk I/O operations and speeds up query processing.

- **Improved Query Execution Plans**: The presence of an index allows the database optimizer to choose more efficient query execution plans. It can utilize the index to perform index scans or seeks, leading to optimized query performance.

Dropping the Index and Analyzing Impact on Query Execution:

To demonstrate the impact of dropping the index on query execution:

```sql
-- Drop the index on the 'author' column of the Books table
DROP INDEX idx_author ON Books;
```

Impact Analysis:

- **Slower Query Performance**: Without the index on the `author` column, queries that rely on filtering or sorting by author may experience slower performance. The database will need to perform full table scans to locate the relevant data, leading to increased query execution times.

- **Increased Disk I/O**: Dropping the index means the database has to read the entire table to fulfill queries, resulting in higher disk I/O operations. This can impact overall system performance, especially in scenarios with large datasets.

- **Suboptimal Query Execution Plans**: The absence of an index may lead to suboptimal query execution plans, as the database optimizer has limited options for optimizing query performance. This can result in inefficient query processing and slower response times.

In conclusion, creating an index on frequently queried columns can significantly improve query performance by facilitating faster data retrieval, reducing disk I/O, and enabling optimized query execution plans. However, dropping an index can lead to slower query performance, increased disk I/O, and suboptimal query execution plans, highlighting the importance of index maintenance for efficient database operations.

Assignment 6:

Create a new database user with specific privileges using the CREATE USER and GRANT commands. Then, write a script to REVOKE certain privileges and DROP the user.

Creating a New Database User with Specific Privileges:

To create a new database user with specific privileges and grant them access to the library system database, you can use the following SQL commands:

```sql
-- Create a new user 'library_user' with a password
CREATE USER 'library_user'@'localhost' IDENTIFIED BY 'password';

-- Grant SELECT, INSERT, UPDATE privileges on the LibrarySystem database to the user
GRANT SELECT, INSERT, UPDATE ON LibrarySystem.* TO 'library_user'@'localhost';
```

Revoking Certain Privileges and Dropping the User:

To revoke specific privileges from the user and then drop the user from the database, you can use the following SQL commands:

```sql
-- Revoke UPDATE privilege on the LibrarySystem database from the user
REVOKE UPDATE ON LibrarySystem.* FROM 'library_user'@'localhost';

-- Drop the user 'library_user' from the database
DROP USER 'library_user'@'localhost';
```

Explanation:

- The `CREATE USER` command is used to create a new database user with a specified username and password.
- The `GRANT` command is used to assign specific privileges (e.g., SELECT, INSERT, UPDATE) on the specified database to the user.
- The `REVOKE` command is used to revoke specific privileges from the user, in this case, revoking the UPDATE privilege.
- The `DROP USER` command is used to remove the user from the database, effectively deleting their account and revoking all associated privileges.

By following these steps, you can create a new database user, grant them specific privileges, revoke certain privileges if needed, and ultimately drop the user from the database when their access is no longer required.

Assignment 7:

Prepare a series of SQL statements to INSERT new records into the library tables, UPDATE existing records with new information, and DELETE records based on specific criteria. Include BULK INSERT operations to load data from an external source.

SQL Statements for Data Manipulation in Library Tables:

1. INSERT New Records into Library Tables:

```sql
-- Insert a new book record into the Books table
INSERT INTO Books (book_id, title, author, publication_year, genre, available)
VALUES (101, 'The Great Gatsby', 'F. Scott Fitzgerald', 1925, 'Classic', TRUE);

-- Insert a new member record into the Members table
INSERT INTO Members (member_id, first_name, last_name, email, phone, registration_date)
VALUES (201, 'Alice', 'Smith', 'alice@example.com', '123-456-7890', '2024-06-15');
```

2. UPDATE Existing Records in Library Tables:

```sql
-- Update the genre of a book in the Books table
UPDATE Books
SET genre = 'Historical Fiction'
WHERE book_id = 101;

-- Update the phone number of a member in the Members table
UPDATE Members
SET phone = '987-654-3210'
```

WHERE member_id = 201;

```

#### 3. DELETE Records Based on Specific Criteria:

```sql

-- Delete a book record from the Books table

DELETE FROM Books

WHERE book_id = 101;

-- Delete a member record from the Members table

DELETE FROM Members

WHERE member_id = 201;

```

4. BULK INSERT Operations to Load Data from an External Source:

Assuming you have a CSV file named `new_books.csv` with book data to be loaded into the `Books` table:

```sql

-- Create a temporary table to stage the data

CREATE TEMPORARY TABLE temp_books (

  book_id INT,

  title VARCHAR(100),

  author VARCHAR(100),

  publication_year INT,

```
  genre VARCHAR(50),
  available BOOLEAN
);

-- Load data from the CSV file into the temporary table
LOAD DATA INFILE '/path/to/new_books.csv'
INTO TABLE temp_books
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 LINES;

-- Insert data from the temporary table into the Books table
INSERT INTO Books (book_id, title, author, publication_year, genre, available)
SELECT * FROM temp_books;

-- Drop the temporary table
DROP TEMPORARY TABLE temp_books;
```

Explanation:

- The `INSERT` statements add new records to the tables.
- The `UPDATE` statements modify existing records based on specified conditions.
- The `DELETE` statements remove records that meet specific criteria.

- The `BULK INSERT` operation loads data from an external CSV file into a temporary table before inserting it into the target table, allowing for efficient bulk data loading.

By using these SQL statements, you can effectively manipulate data in the library tables by inserting new records, updating existing information, deleting records based on criteria, and performing bulk data loading operations from external sources.