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.

Sol:

Designing a database schema for a library system involves creating several tables to handle different entities such as books, authors, members, loans, and so on. Here's a detailed schema with tables, fields, and constraints:

Tables and Fields

1. Authors

- `AuthorID` (Primary Key, Integer, NOT NULL)
- `FirstName` (Varchar (50), NOT NULL)
- `LastName` (Varchar (50), NOT NULL)

2. Books

- `BookID` (Primary Key, Integer, NOT NULL)
- `Title` (Varchar (255), NOT NULL)
- `ISBN` (Varchar (13), UNIQUE, NOT NULL)
- `PublicationYear` (Year, CHECK (`PublicationYear` > 0))
- `AuthorID` (Foreign Key, Integer, NOT NULL, REFERENCES `Authors`(`AuthorID`))

3. Members

- `MemberID` (Primary Key, Integer, NOT NULL)
- `FirstName` (Varchar (50), NOT NULL)
- `LastName` (Varchar (50), NOT NULL)
- `Email` (Varchar (100), UNIQUE, NOT NULL)
- `Phone` (Varchar (15), UNIQUE, NOT NULL)
- `JoinDate` (Date, NOT NULL)

4. Loans

- `LoanID` (Primary Key, Integer, NOT NULL)
- `BookID` (Foreign Key, Integer, NOT NULL, REFERENCES `Books`(`BookID`))

- `MemberID` (Foreign Key, Integer, NOT NULL, REFERENCES `Members`(`MemberID`))
- `LoanDate` (Date, NOT NULL)
- `DueDate` (Date, NOT NULL, CHECK (`DueDate` > `LoanDate`))
- `ReturnDate` (Date, NULL)

5. Categories

- `CategoryID` (Primary Key, Integer, NOT NULL)
- `CategoryName` (Varchar (100), UNIQUE, NOT NULL)

6. Book Categories

- `BookID` (Foreign Key, Integer, NOT NULL, REFERENCES `Books`(`BookID`))
- `CategoryID` (Foreign Key, Integer, NOT NULL, REFERENCES`Categories`(`CategoryID`))
- PRIMARY KEY (`BookID`, `CategoryID`)

Detailed Schema

SQL:

```
CREATE TABLE Authors (
    AuthorID INT PRIMARY KEY NOT NULL,
    FirstName VARCHAR (50) NOT NULL,
    LastName VARCHAR (50) NOT NULL
);

CREATE TABLE Books (
    BookID INT PRIMARY KEY NOT NULL,
    Title VARCHAR (255) NOT NULL,
    ISBN VARCHAR (13) UNIQUE NOT NULL,
    PublicationYear YEAR CHECK (PublicationYear > 0),
    AuthorID INT NOT NULL,
    FOREIGN KEY (AuthorID) REFERENCES Authors (AuthorID)
);
```

```
CREATE TABLE Members (
    MemberID INT PRIMARY KEY NOT NULL,
    FirstName VARCHAR (50) NOT NULL,
    LastName VARCHAR (50) NOT NULL,
    Email VARCHAR (100) UNIQUE NOT NULL,
    Phone VARCHAR (15) UNIQUE NOT NULL,
    JoinDate DATE NOT NULL
);

CREATE TABLE Loans (
    LoanID INT PRIMARY KEY NOT NULL,
    BookID INT NOT NULL,
    MemberID INT NOT NULL,
    LoanDate DATE NOT NULL,
    DueDate DATE NOT NULL CHECK (DueDate > LoanDate),
    ReturnDate DATE NULL,
    FOREIGN KEY (BookID) REFERENCES Books (BookID),
    FOREIGN KEY (MemberID) REFERENCES Members (MemberID)
);

CREATE TABLE Categories (
    CategoryID INT PRIMARY KEY NOT NULL,
    CategoryName VARCHAR (100) UNIQUE NOT NULL
);

CREATE TABLE BookCategories (
    BookID INT NOT NULL,
    CategoryID INT NOT NULL,
```

PRIMARY KEY (BookID, CategoryID),

FOREIGN KEY (BookID) REFERENCES Books (BookID),

FOREIGN KEY (CategoryID) REFERENCES Categories (CategoryID)

);

Explanation of Constraints

Primary Key (PK): Ensures each record in the table is unique.

Foreign Key (FK): Establishes relationships between tables.

NOT NULL: Ensures that a field cannot have a NULL value.

UNIQUE: Ensures all values in a column are unique.

CHECK: Ensures that all values in a column satisfy a specific condition.

This schema provides a comprehensive structure for managing a library system, covering books, authors, members, loans, and categories, while enforcing data integrity and relationships between tables.

==========

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.

Sol:

Certainly! Below is a set of SQL statements to create a new database and tables for the library schema designed earlier. This includes the creation of the database, tables, insertion of data, altering table structures, and dropping a redundant table.

Step 1: Create the Database

CREATE DATABASE LibraryDB;

USE LibraryDB;

Step 2: Create Tables

Authors Table

```
CREATE TABLE Authors (
    AuthorID INT PRIMARY KEY AUTO_INCREMENT,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL
);
```

Books Table

```
CREATE TABLE Books (
    BookID INT PRIMARY KEY AUTO_INCREMENT,
    Title VARCHAR(255) NOT NULL,
    ISBN VARCHAR(13) UNIQUE NOT NULL,
    PublicationYear YEAR CHECK (PublicationYear > 0),
    AuthorID INT NOT NULL,
    FOREIGN KEY (AuthorID) REFERENCES Authors(AuthorID)
);
```

Members Table

```
CREATE TABLE Members (
    MemberID INT PRIMARY KEY AUTO_INCREMENT,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE NOT NULL,
    Phone VARCHAR(15) UNIQUE NOT NULL,
    JoinDate DATE NOT NULL
);
```

Loans Table

```
CREATE TABLE Loans (
    LoanID INT PRIMARY KEY AUTO_INCREMENT,
    BookID INT NOT NULL,
```

```
    MemberID INT NOT NULL,
    LoanDate DATE NOT NULL,
    DueDate DATE NOT NULL CHECK (DueDate > LoanDate),
    ReturnDate DATE,
    FOREIGN KEY (BookID) REFERENCES Books(BookID),
    FOREIGN KEY (MemberID) REFERENCES Members(MemberID)
);
```

Categories Table

```
CREATE TABLE Categories (
    CategoryID INT PRIMARY KEY AUTO_INCREMENT,
    CategoryName VARCHAR(100) UNIQUE NOT NULL
);
```

BookCategories Table

```
CREATE TABLE BookCategories (
    BookID INT NOT NULL,
    CategoryID INT NOT NULL,
    PRIMARY KEY (BookID, CategoryID),
    FOREIGN KEY (BookID) REFERENCES Books(BookID),
    FOREIGN KEY (CategoryID) REFERENCES Categories(CategoryID)
);
```

Step 3: Insert Sample Data

Insert Data into Authors Table

```
INSERT INTO Authors (FirstName, LastName) VALUES
('J.K.', 'Rowling'),
('George', 'Orwell'),
('J.R.R.', 'Tolkien');
```

Insert Data into Books Table

```
INSERT INTO Books (Title, ISBN, PublicationYear, AuthorID) VALUES
('Harry Potter and the Philosopher\'s Stone', '9780747532699', 1997, 1),
```

('1984', '9780451524935', 1949, 2),

('The Hobbit', '9780261102217', 1937, 3);

Insert Data into Members Table

INSERT INTO Members (FirstName, LastName, Email, Phone, JoinDate) VALUES

('Alice', 'Smith', 'alice.smith@example.com', '1234567890', '2023-01-15'),

('Bob', 'Brown', 'bob.brown@example.com', '0987654321', '2023-02-20');

Insert Data into Loans Table

INSERT INTO Loans (BookID, MemberID, LoanDate, DueDate, ReturnDate) VALUES

(1, 1, '2023-05-01', '2023-05-15', NULL),

(2, 2, '2023-05-03', '2023-05-17', '2023-05-10');

Insert Data into Categories Table

INSERT INTO Categories (CategoryName) VALUES

('Fantasy'),

('Dystopian'),

('Adventure');

Insert Data into BookCategories Table

INSERT INTO BookCategories (BookID, CategoryID) VALUES

(1, 1),

(2, 2),

(3, 3);

Step 4: Modify Table Structures

Alter the Authors Table to Add a MiddleName Column

ALTER TABLE Authors

ADD COLUMN MiddleName VARCHAR(50);

Alter the Books Table to Add a Publisher Column

ALTER TABLE Books

ADD COLUMN Publisher VARCHAR(100);

Step 5: Drop a Redundant Table

Assume we have a redundant table named `OldMembers`

DROP TABLE IF EXISTS OldMembers;

Complete Script

Here is the complete script to create the `LibraryDB` database, set up the tables, insert sample data, alter the table structures, and drop a redundant table:

```
-- Create Database
CREATE DATABASE LibraryDB;
USE LibraryDB;

-- Create Tables
CREATE TABLE Authors (
    AuthorID INT PRIMARY KEY AUTO_INCREMENT,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL
);

CREATE TABLE Books (
    BookID INT PRIMARY KEY AUTO_INCREMENT,
    Title VARCHAR(255) NOT NULL,
    ISBN VARCHAR(13) UNIQUE NOT NULL,
    PublicationYear YEAR CHECK (PublicationYear > 0),
    AuthorID INT NOT NULL,
    FOREIGN KEY (AuthorID) REFERENCES Authors(AuthorID)
);

CREATE TABLE Members (
    MemberID INT PRIMARY KEY AUTO_INCREMENT,
    FirstName VARCHAR(50) NOT NULL,
```

```
    LastName VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE NOT NULL,
    Phone VARCHAR(15) UNIQUE NOT NULL,
    JoinDate DATE NOT NULL
);

CREATE TABLE Loans (
    LoanID INT PRIMARY KEY AUTO_INCREMENT,
    BookID INT NOT NULL,
    MemberID INT NOT NULL,
    LoanDate DATE NOT NULL,
    DueDate DATE NOT NULL CHECK (DueDate > LoanDate),
    ReturnDate DATE,
    FOREIGN KEY (BookID) REFERENCES Books(BookID),
    FOREIGN KEY (MemberID) REFERENCES Members(MemberID)
);

CREATE TABLE Categories (
    CategoryID INT PRIMARY KEY AUTO_INCREMENT,
    CategoryName VARCHAR(100) UNIQUE NOT NULL
);

CREATE TABLE BookCategories (
    BookID INT NOT NULL,
    CategoryID INT NOT NULL,
    PRIMARY KEY (BookID, CategoryID),
    FOREIGN KEY (BookID) REFERENCES Books(BookID),
    FOREIGN KEY (CategoryID) REFERENCES Categories(CategoryID)
);
```

-- Insert Sample Data

```
INSERT INTO Authors (FirstName, LastName) VALUES
('J.K.', 'Rowling'),
('George', 'Orwell'),
('J.R.R.', 'Tolkien');

INSERT INTO Books (Title, ISBN, PublicationYear, AuthorID) VALUES
('Harry Potter and the Philosopher\'s Stone', '9780747532699', 1997, 1),
('1984', '9780451524935', 1949, 2),
('The Hobbit', '9780261102217', 1937, 3);

INSERT INTO Members (FirstName, LastName, Email, Phone, JoinDate) VALUES
('Alice', 'Smith', 'alice.smith@example.com', '1234567890', '2023-01-15'),
('Bob', 'Brown', 'bob.brown@example.com', '0987654321', '2023-02-20');

INSERT INTO Loans (BookID, MemberID, LoanDate, DueDate, ReturnDate) VALUES
(1, 1, '2023-05-01', '2023-05-15', NULL),
(2, 2, '2023-05-03', '2023-05-17', '2023-05-10');

INSERT INTO Categories (CategoryName) VALUES
('Fantasy'),
('Dystopian'),
('Adventure');

INSERT INTO BookCategories (BookID, CategoryID) VALUES
(1, 1),
(2, 2),
(3, 3);
```

-- Alter Table Structures

```
ALTER TABLE Authors
ADD COLUMN MiddleName VARCHAR(50);

ALTER TABLE Books
ADD COLUMN Publisher VARCHAR(100);
```

-- Drop Redundant Table

```
DROP TABLE IF EXISTS OldMembers;
```

==========

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 analyse the impact on query execution.

Sol:

1. Creating a Table

First, we'll create a sample table. Let's assume we have a table called `employees` with the following columns: `id`, `name`, `department`, and `salary`.

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

2. Inserting Sample Data

Next, let's insert some sample data into the table.

INSERT INTO employees (id, name, department, salary) VALUES

(1, 'Alice', 'HR', 60000),

(2, 'Bob', 'Engineering', 80000),

(3, 'Charlie', 'Marketing', 55000),

(4, 'David', 'Engineering', 75000),

(5, 'Eve', 'HR', 70000);

3. Creating an Index

We'll create an index on the `department` column to improve query performance when filtering by department.

CREATE INDEX index department ON employees(department);

4. Analysing Query Performance with Index

Consider the following query to fetch all employees in the Engineering department:

SELECT * FROM employees WHERE department = 'Engineering';

With the index `index department`, the database can quickly locate the rows where the `department` is 'Engineering' without scanning the entire table. This significantly improves query performance, especially for large tables.

5. Dropping the Index

To demonstrate the impact of removing the index, we'll drop it using the `DROP INDEX` statement.

DROP INDEX index department ON employees;

6. Analysing Query Performance without Index

Running the same query after dropping the index:

SELECT * FROM employees WHERE department = 'Engineering';

==========

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

Sol:

1. Creating a New User

First, let's create a new database user. Assume we are working with a MySQL database.

-- Create a new user 'test_user' with password 'password123'

```
CREATE USER 'test_user'@'localhost' IDENTIFIED BY 'password123';
```

2. Granting Privileges

Next, we'll grant specific privileges to this user. For instance, let's give the user privileges to select, insert, update, and delete on a database called `test_db`.

-- Grant SELECT, INSERT, UPDATE, DELETE privileges on the test_db database

```
GRANT SELECT, INSERT, UPDATE, DELETE ON test_db.* TO 'test_user'@'localhost';
```

3. Revoking Privileges

Now, let's write a script to revoke certain privileges from the user. Suppose we want to revoke the `DELETE` privilege.

-- Revoke DELETE privilege on the test_db database

```
REVOKE DELETE ON test_db.* FROM 'test_user'@'localhost';
```

4. Dropping the User

Finally, let's drop the user.

-- Drop the user 'test_user'

```
DROP USER 'test_user'@'localhost';
```

Combined Script

Here's a combined script that creates a user, grants privileges, revokes a privilege, and then drops the user:

-- Step 1: Create a new user

```
CREATE USER 'test_user'@'localhost' IDENTIFIED BY 'password123';
```

-- Step 2: Grant specific privileges to the user

GRANT SELECT, INSERT, UPDATE, DELETE ON test_db.* TO 'test_user'@'localhost';

-- Step 3: Revoke the DELETE privilege from the user

REVOKE DELETE ON test_db.* FROM 'test_user'@'localhost';

-- Step 4: Drop the user

DROP USER 'test_user'@'localhost';

Explanation

1**. CREATE USER:** This command creates a new user with a specified username and password.

2**. GRANT:** This command grants specified privileges (SELECT, INSERT, UPDATE, DELETE) on all tables in the `test_db` database to the user.

3. **REVOKE**: This command revokes the `DELETE` privilege from the user, removing their ability to delete records in the `test_db` database.

4. **DROP USER:** This command removes the user from the database, effectively revoking all their privileges and deleting their account.

==========

Assignment 7:

Prepare a series 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.

we have a library database with the following tables:

1. `books`

2. `authors`

3. `members`

4. `loans`

We'll go through the following steps:

1. Inserting New Records

2. Updating Existing Records

3. Deleting Records Based on Specific Criteria

4. Bulk Insert Operations

1. Inserting New Records

First, let's add new records to the `books`, `authors`, and `members` tables.

-- Insert new records into the authors table

INSERT INTO authors (author_id, name, birth_date)

VALUES

(1, 'J.K. Rowling', '1965-07-31'),

(2, 'George Orwell', '1903-06-25'),

(3, 'Jane Austen', '1775-12-16');

-- Insert new records into the books table

INSERT INTO books (book_id, title, author_id, genre, published_year)

VALUES

(1, 'Harry Potter and the Philosopher\'s Stone', 1, 'Fantasy', 1997),

(2, '1984', 2, 'Dystopian', 1949),

(3, 'Pride and Prejudice', 3, 'Romance', 1813);

-- Insert new records into the members table

INSERT INTO members (member_id, name, join_date)

VALUES

(1, 'Alice Johnson', '2023-01-15'),

(2, 'Bob Smith', '2023-03-22'),

(3, 'Carol White', '2023-05-11');

2. Updating Existing Records

Next, let's update existing records with new information.

-- Update an author's name

```
UPDATE authors
SET name = 'Joanne Rowling'
WHERE author_id = 1;
```

-- Update a book's genre

```
UPDATE books
SET genre = 'Classic'
WHERE book_id = 2;
```

-- Update a member's join date

```
UPDATE members
SET join_date = '2023-04-01'
WHERE member_id = 2;
```

3. Deleting Records Based on Specific Criteria

Now, let's delete records based on certain criteria.

-- Delete books published before the year 1900

```
DELETE FROM books
WHERE published_year < 1900;
```

-- Delete members who joined before 2023

```
DELETE FROM members
WHERE join_date < '2023-01-01';
```

-- Delete authors with no books

```
DELETE FROM authors
WHERE author_id NOT IN (SELECT author_id FROM books);
```

4. Bulk Insert Operations

For bulk insert operations, we'll assume you have a CSV file named `books_data.csv` that contains data for the `books` table. Here's how to load that data using a bulk insert.

Example CSV File (`books_data.csv`):

book_id,title,author_id, genre,published_year

4, Animal Farm,2,Political Satire,1945

5, Emma,3, Romance,1815

6, The Casual Vacancy,1, Drama,2012

Bulk Insert Command:

-- Load data from an external CSV file into the books table

```
LOAD DATA INFILE '/path/to/books_data.csv'
INTO TABLE books
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(book_id, title, author_id, genre, published_year);
```

==========

Assignment 1:

Write a SELECT query to retrieve all columns from a 'customers' table, and modify it to return only the customer name and email address for customers in a specific city.

Sol:

The basic SQL query to retrieve all columns from a `customers` table looks like this:

SELECT * FROM customers;

To modify this query to return only the customer name and email address for customers in a specific city, you'll need to add a `WHERE` clause to filter by city and specify the columns you want to retrieve. Assuming the columns for customer name and email address are named `customer_name` and `email`, and the column for the city is named `city`, the query would be:

SELECT customer_name, email

FROM customers

WHERE city = 'Specific City';

Replace `'Specific City'` with the actual city you are interested in "New York".

SELECT customer_name, email

FROM customers

WHERE city = 'New York';

Here is the general form of the query:

SELECT customer_name, email

FROM customers

WHERE city = 'YourCityName';

Just replace ` with the desired YourCityName`city's name.

==========

Assignment 2:

Craft a query using an INNER JOIN to combine 'orders' and 'customers' tables for customers in a specified region, and a LEFT JOIN to display all customers including those without orders.

Sol:

Here are the Two SQL queries:

1. INNER JOIN to combine `orders` and `customers` tables for customers in a specified region.

2. LEFT JOIN to display all customers, including those without orders.

Query 1: INNER JOIN for Customers in a Specified Region

Let's assume the `customers` table has columns like `customer_id`, `customer_name`, `email`, and `region`, and the `orders` table has columns like `order_id`, `order_date`, `customer_id`, and `order_amount`.

Here's how you can write an `INNER JOIN` query to get the orders for customers in a specific region:

```
SELECT c.customer_name, c.email, o.order_id, o.order_date, o.order_amount
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
WHERE c.region = 'Specified Region';
```

Replace `'Specified Region'` with the actual region you are interested in. For example, if the region is 'North', the query would be:

```
SELECT c.customer_name, c.email, o.order_id, o.order_date, o.order_amount
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
WHERE c.region = 'North';
```

Query 2: LEFT JOIN to Display All Customers Including Those Without Orders

Here's how you can write a `LEFT JOIN` query to get all customers, including those without any orders:

```
SELECT c.customer_name, c.email, o.order_id, o.order_date, o.order_amount
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

This query will return all customers, and for those without any orders, the `order_id`, `order_date`, and `order_amount` fields will contain `NULL`.

By combining these queries, you can retrieve the information as needed for different scenarios.

==========

Assignment 3:

Utilize a subquery to find customers who have placed orders above the average order value, and write a UNION query to combine two SELECT statements with the same number of columns.

Sol:

Subquery to Find Customers Who Have Placed Orders Above the Average Order Value

To find customers who have placed orders above the average order value, we first need to calculate the average order value. Then, we use a subquery to filter out customers whose orders exceed this average.

```
SELECT c.customer_name, c.email
FROM customers c
WHERE c.customer_id IN (
    SELECT o.customer_id
    FROM orders o
    WHERE o.order_amount > (
```

SELECT AVG(order_amount)

FROM orders));

UNION Query to Combine Two SELECT Statements with the Same Number of Columns

A UNION query combines the results of two or more SELECT statements. Each SELECT statement within the UNION must have the same number of columns, and the corresponding columns must have compatible data types.

Here are two example SELECT statements that can be combined using UNION:

1. The first SELECT statement retrieves customer names and emails from customers who are located in 'Region A'.

2. The second SELECT statement retrieves customer names and emails from customers who have placed orders above a certain amount.

-- First SELECT statement: Customers from 'Region A'

SELECT customer_name, email

FROM customers

WHERE region = 'Region A'

UNION

-- Second SELECT statement: Customers with orders above a certain amount

SELECT c.customer_name, c.email

FROM customers c

JOIN orders o ON c.customer_id = o.customer_id

WHERE o.order_amount > 100;

In this example, replace `'Region A'` with the desired region and `100` with the order amount threshold you are interested in.

Combined Explanation

-Subquery: Used within the `WHERE` clause to find customers who have placed orders above the average order value. The subquery calculates the average order value, and the main query filters customers based on this value.

-UNION: Combines the results of two SELECT statements with the same number of columns and compatible data types. The `UNION` operator removes duplicate rows by default. If you want to include duplicates, use `UNION ALL`.

These queries demonstrate how to perform complex data retrieval tasks using subqueries and UNION operations in SQL.