**Assignment on 05-08-2024**

**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.

**Business Scenario**

**Scenario: Project Management System**

A company wants to develop a system to manage projects. The system should track:

* **Employees** who work for the company, including their details and the department they belong to.
* **Projects** that the company undertakes, including project details and the departments responsible for them.
* **Tasks** associated with each project, including details and assigned employees.
* **Departments** within the company, which have multiple employees and can be responsible for multiple projects.

**Key Requirements**

1. Each employee belongs to one department but can work on multiple tasks across different projects.
2. Each project is managed by one department and can have multiple tasks.
3. Each task can be assigned to one or more employees.

**Identifying Entities, Attributes, and Relationships**

**Entities and Attributes:**

1. **Employee**
   * EmployeeID (PK)
   * Name
   * Position
   * Email
   * DepartmentID (FK)
2. **Department**
   * DepartmentID (PK)
   * DepartmentName
3. **Project**
   * ProjectID (PK)
   * ProjectName
   * StartDate
   * EndDate
   * DepartmentID (FK)
4. **Task**
   * TaskID (PK)
   * TaskName
   * DueDate
   * ProjectID (FK)
5. **EmployeeTask** (Association entity for many-to-many relationship between Employee and Task)
   * EmployeeID (FK)
   * TaskID (FK)

**Relationships and Cardinality:**

1. **Department to Employee**: One-to-Many (1

)

* + A department has many employees, but an employee belongs to one department.

1. **Department to Project**: One-to-Many (1

)

* + A department can manage multiple projects, but a project is managed by one department.

1. **Project to Task**: One-to-Many (1

)

* + A project can have multiple tasks, but each task belongs to one project.

1. **Task to Employee**: Many-to-Many (M

)

* + A task can be assigned to multiple employees, and an employee can work on multiple tasks.

**ER Diagram**

Let's create an ER diagram based on the identified entities, relationships, and cardinalities. I'll represent this diagram in text form, and then we can draw it graphically.

**Diagram Representation (Text Form)**

* **Employee** (EmployeeID, Name, Position, Email, DepartmentID)
* **Department** (DepartmentID, DepartmentName)
* **Project** (ProjectID, ProjectName, StartDate, EndDate, DepartmentID)
* **Task** (TaskID, TaskName, DueDate, ProjectID)
* **EmployeeTask** (EmployeeID, TaskID)

**Relationships:**

1. **Department to Employee**: (1

)

1. **Department to Project**: (1

)

1. **Project to Task**: (1

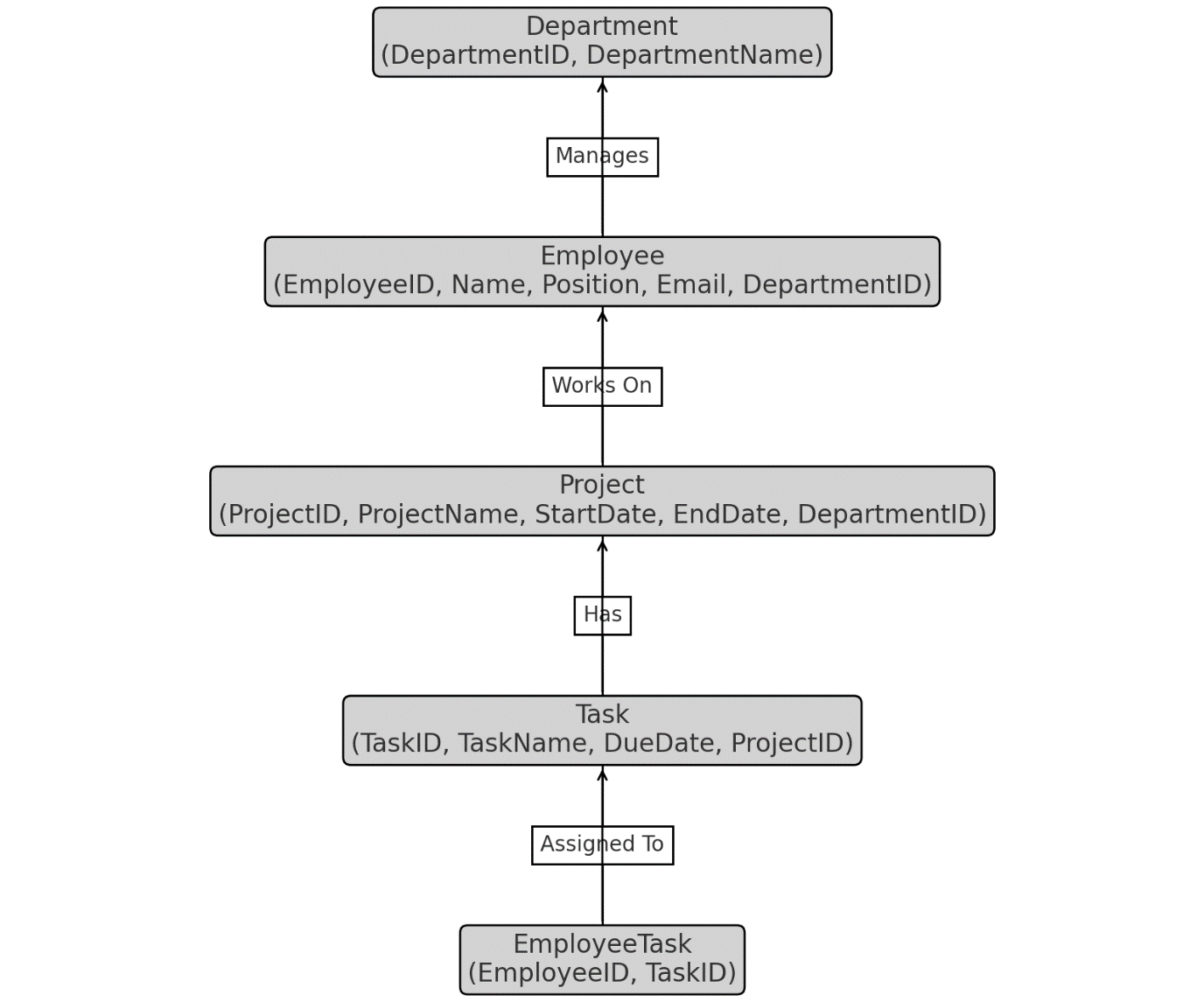
)

1. **Task to Employee** (via EmployeeTask): (M

)

**Graphical ER Diagram**

I will now create the ER diagram graphically to illustrate the scenario.



Here's the ER diagram for the Project Management System scenario. The diagram includes the entities, their attributes, relationships, and cardinalities, ensuring normalization up to the third normal form (3NF).

* **Entities**:
  + **Department**: Manages many employees and projects.
  + **Employee**: Belongs to one department and works on many tasks.
  + **Project**: Managed by one department and has many tasks.
  + **Task**: Assigned to many employees.
  + **EmployeeTask**: Represents the many-to-many relationship between employees and tasks.

This diagram ensures that the database design is normalized, with all the attributes properly distributed among the entities to avoid redundancy and ensure data integrity

**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.

**Tables and Fields**

1. **Book**
   * **BookID**: INT (Primary Key)
   * **Title**: VARCHAR(255) (NOT NULL)
   * **AuthorID**: INT (Foreign Key to Author.AuthorID)
   * **PublisherID**: INT (Foreign Key to Publisher.PublisherID)
   * **ISBN**: VARCHAR(13) (NOT NULL, UNIQUE)
   * **PublishedYear**: INT (CHECK (PublishedYear >= 1000 AND PublishedYear <= 9999))
   * **Genre**: VARCHAR(100)
   * **CopiesAvailable**: INT (NOT NULL, CHECK (CopiesAvailable >= 0))
2. **Author**
   * **AuthorID**: INT (Primary Key)
   * **FirstName**: VARCHAR(100) (NOT NULL)
   * **LastName**: VARCHAR(100) (NOT NULL)
   * **DateOfBirth**: DATE
   * **Nationality**: VARCHAR(100)
3. **Publisher**
   * **PublisherID**: INT (Primary Key)
   * **Name**: VARCHAR(255) (NOT NULL)
   * **Address**: VARCHAR(255)
   * **Phone**: VARCHAR(20)
4. **Member**
   * **MemberID**: INT (Primary Key)
   * **FirstName**: VARCHAR(100) (NOT NULL)
   * **LastName**: VARCHAR(100) (NOT NULL)
   * **Email**: VARCHAR(255) (NOT NULL, UNIQUE)
   * **Phone**: VARCHAR(20)
   * **DateOfMembership**: DATE (NOT NULL)
   * **MembershipType**: VARCHAR(50) (CHECK (MembershipType IN ('Regular', 'Premium')))
5. **Loan**
   * **LoanID**: INT (Primary Key)
   * **BookID**: INT (Foreign Key to Book.BookID)
   * **MemberID**: INT (Foreign Key to Member.MemberID)
   * **IssueDate**: DATE (NOT NULL)
   * **DueDate**: DATE (NOT NULL)
   * **ReturnDate**: DATE
6. **Reservation**
   * **ReservationID**: INT (Primary Key)
   * **BookID**: INT (Foreign Key to Book.BookID)
   * **MemberID**: INT (Foreign Key to Member.MemberID)
   * **ReservationDate**: DATE (NOT NULL)

**Constraints**

* **Primary Keys**: Each table has a unique identifier as the primary key (e.g., BookID, AuthorID, etc.).
* **Foreign Keys**: Establish relationships between tables (e.g., Book.AuthorID references Author.AuthorID).
* **NOT NULL**: Ensures that certain fields cannot be left empty (e.g., Title, ISBN).
* **UNIQUE**: Ensures unique values for fields (e.g., ISBN, Email).
* **CHECK**: Ensures certain conditions are met (e.g., PublishedYear within a valid range, MembershipType values).

**Example Schema Definition (SQL)**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

CREATE TABLE Author (

AuthorID INT PRIMARY KEY,

FirstName VARCHAR(100) NOT NULL,

LastName VARCHAR(100) NOT NULL,

DateOfBirth DATE,

Nationality VARCHAR(100)

);

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

CREATE TABLE Publisher (

PublisherID INT PRIMARY KEY,

Name VARCHAR(255) NOT NULL,

Address VARCHAR(255),

Phone VARCHAR(20)

);

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

CREATE TABLE Book (

BookID INT PRIMARY KEY,

Title VARCHAR(255) NOT NULL,

AuthorID INT,

PublisherID INT,

ISBN VARCHAR(13) NOT NULL UNIQUE,

PublishedYear INT CHECK (PublishedYear >= 1000 AND PublishedYear <= 9999),

Genre VARCHAR(100),

CopiesAvailable INT NOT NULL CHECK (CopiesAvailable >= 0),

FOREIGN KEY (AuthorID) REFERENCES Author(AuthorID),

FOREIGN KEY (PublisherID) REFERENCES Publisher(PublisherID)

);

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

CREATE TABLE Member (

MemberID INT PRIMARY KEY,

FirstName VARCHAR(100) NOT NULL,

LastName VARCHAR(100) NOT NULL,

Email VARCHAR(255) NOT NULL UNIQUE,

Phone VARCHAR(20),

DateOfMembership DATE NOT NULL,

MembershipType VARCHAR(50) CHECK (MembershipType IN ('Regular', 'Premium'))

);

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

CREATE TABLE Loan (

LoanID INT PRIMARY KEY,

BookID INT,

MemberID INT,

IssueDate DATE NOT NULL,

DueDate DATE NOT NULL,

ReturnDate DATE,

FOREIGN KEY (BookID) REFERENCES Book(BookID),

FOREIGN KEY (MemberID) REFERENCES Member(MemberID)

);

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

CREATE TABLE Reservation (

ReservationID INT PRIMARY KEY,

BookID INT,

MemberID INT,

ReservationDate DATE NOT NULL,

FOREIGN KEY (BookID) REFERENCES Book(BookID),

FOREIGN KEY (MemberID) REFERENCES Member(MemberID)

);

**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.has context menu

The ACID properties define the essential characteristics of a reliable database transaction, ensuring data integrity and consistency. Let's briefly describe each property:

1. **Atomicity**: A transaction is an all-or-nothing operation. Either all the operations within a transaction are completed successfully, or none are applied. If any part of the transaction fails, the entire transaction is rolled back, leaving the database unchanged.
2. **Consistency**: A transaction ensures that the database transitions from one valid state to another. This means that any data written to the database must be valid according to all defined rules, including constraints, cascades, and triggers.
3. **Isolation**: Transactions are isolated from each other, meaning that the operations of one transaction are invisible to other transactions until the transaction is committed. This prevents the "dirty read" problem and ensures that concurrent transactions do not interfere with each other.
4. **Durability**: Once a transaction is committed, its changes are permanent and must survive any subsequent system failures. The data changes are written to persistent storage, ensuring that they are not lost.

**Simulating a Transaction with Locking and Isolation Levels**

To demonstrate ACID properties, particularly **Isolation**, we'll simulate a transaction involving updating an account balance in a banking system. We'll also use different isolation levels to illustrate how concurrency control is managed in SQL.

**Setup: Creating a Table and Inserting Initial Data**

-- Create a table for bank accounts

CREATE TABLE BankAccount (

AccountID INT PRIMARY KEY,

Balance DECIMAL(10, 2) NOT NULL CHECK (Balance >= 0)

);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-- Insert initial data

INSERT INTO BankAccount (AccountID, Balance) VALUES (1, 1000.00);

INSERT INTO BankAccount (AccountID, Balance) VALUES (2, 1500.00);

**Transaction Simulation**

Suppose we want to transfer $200 from Account 1 to Account 2. We'll wrap this operation in a transaction to ensure atomicity and consistency.

-- Start a transaction

BEGIN TRANSACTION;

-- Lock the rows to prevent other transactions from modifying them

SELECT Balance FROM BankAccount WHERE AccountID = 1 FOR UPDATE;

SELECT Balance FROM BankAccount WHERE AccountID = 2 FOR UPDATE;

-- Deduct $200 from Account 1

UPDATE BankAccount

SET Balance = Balance - 200

WHERE AccountID = 1;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-- Add $200 to Account 2

UPDATE BankAccount

SET Balance = Balance + 200

WHERE AccountID = 2;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-- Commit the transaction

COMMIT;

**Isolation Levels Demonstration**

Isolation levels control the visibility of changes made in one transaction to other transactions. We'll demonstrate four standard isolation levels:

1. **READ UNCOMMITTED**: Allows dirty reads. A transaction can see changes made by other uncommitted transactions.
2. **READ COMMITTED**: Prevents dirty reads. A transaction can only see changes made by other transactions once they are committed.
3. **REPEATABLE READ**: Prevents non-repeatable reads. Ensures that if a transaction reads the same data multiple times, it will see the same values each time.
4. **SERIALIZABLE**: Ensures full isolation. Transactions are executed in a way that results are the same as if the transactions were executed sequentially.

**Example with SERIALIZABLE Isolation Level:**

-- Set isolation level to SERIALIZABLE

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;

-- This transaction will have a high level of isolation.

-- Any read here will be consistent throughout the transaction.

-- Lock the rows

SELECT Balance FROM BankAccount WHERE AccountID = 1 FOR UPDATE;

-- Perform the operation

UPDATE BankAccount

SET Balance = Balance - 200

WHERE AccountID = 1;

-- Simultaneous transactions attempting to access Account 1 will be blocked until this transaction is completed.

-- Commit the transaction

COMMIT;

In the example above, using SERIALIZABLE ensures that no other transactions can interfere with the current transaction's operations on Account 1 until it is complete, thus maintaining full isolation and preventing phenomena like dirty reads, non-repeatable reads, and phantom reads.

These concepts are crucial for maintaining data integrity and consistency in multi-user database environments, where concurrent transactions can potentially conflict.

**Assignment on 06-08-2024**

**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.

**Creating a New Database and Tables**

First, we'll create a new database, then define the tables according to the library schema described earlier.

-- Create a new database

CREATE DATABASE LibraryDB;

-- Use the new database

USE LibraryDB;

-- Create Author table

CREATE TABLE Author (

AuthorID INT PRIMARY KEY,

FirstName VARCHAR(100) NOT NULL,

LastName VARCHAR(100) NOT NULL,

DateOfBirth DATE,

Nationality VARCHAR(100)

);

-- Create Publisher table

CREATE TABLE Publisher (

PublisherID INT PRIMARY KEY,

Name VARCHAR(255) NOT NULL,

Address VARCHAR(255),

Phone VARCHAR(20)

);

-- Create Book table

CREATE TABLE Book (

BookID INT PRIMARY KEY,

Title VARCHAR(255) NOT NULL,

AuthorID INT,

PublisherID INT,

ISBN VARCHAR(13) NOT NULL UNIQUE,

PublishedYear INT CHECK (PublishedYear >= 1000 AND PublishedYear <= 9999),

Genre VARCHAR(100),

CopiesAvailable INT NOT NULL CHECK (CopiesAvailable >= 0),

FOREIGN KEY (AuthorID) REFERENCES Author(AuthorID),

FOREIGN KEY (PublisherID) REFERENCES Publisher(PublisherID)

);

-- Create Member table

CREATE TABLE Member (

MemberID INT PRIMARY KEY,

FirstName VARCHAR(100) NOT NULL,

LastName VARCHAR(100) NOT NULL,

Email VARCHAR(255) NOT NULL UNIQUE,

Phone VARCHAR(20),

DateOfMembership DATE NOT NULL,

MembershipType VARCHAR(50) CHECK (MembershipType IN ('Regular', 'Premium'))

);

-- Create Loan table

CREATE TABLE Loan (

LoanID INT PRIMARY KEY,

BookID INT,

MemberID INT,

IssueDate DATE NOT NULL,

DueDate DATE NOT NULL,

ReturnDate DATE,

FOREIGN KEY (BookID) REFERENCES Book(BookID),

FOREIGN KEY (MemberID) REFERENCES Member(MemberID)

);

-- Create Reservation table

CREATE TABLE Reservation (

ReservationID INT PRIMARY KEY,

BookID INT,

MemberID INT,

ReservationDate DATE NOT NULL,

FOREIGN KEY (BookID) REFERENCES Book(BookID),

FOREIGN KEY (MemberID) REFERENCES Member(MemberID)

);

**ALTER Statements to Modify Table Structures**

Let's say we need to add a new column to the Member table and update the Book table's PublishedYear column to set a default value.

-- Add a new column 'Address' to the Member table

ALTER TABLE Member

ADD Address VARCHAR(255);

-- Set a default value for the 'PublishedYear' column in the Book table

ALTER TABLE Book

ALTER COLUMN PublishedYear SET DEFAULT 2000;

-- Add a new column 'Address' to the Member table

ALTER TABLE Member

ADD Address VARCHAR(255);

-- Set a default value for the 'PublishedYear' column in the Book table

ALTER TABLE Book

ALTER COLUMN PublishedYear SET DEFAULT 2000;

**DROP Statements to Remove a Redundant Table**

Suppose the Reservation table is deemed redundant and needs to be removed from the database

-- Drop the Reservation table

DROP TABLE Reservation;

**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**

An index is a database object that improves the speed of data retrieval operations on a table at the cost of additional storage space and write operation overhead. Indexes can be created on one or more columns of a table to facilitate quick searches, sorts, and joins.

**Example: Creating an Index on the Book Table**

Let's create an index on the ISBN column of the Book table to improve the performance of queries that filter by ISBN.

-- Create an index on the ISBN column in the Book table

CREATE INDEX idx\_ISBN ON Book(ISBN);

**How Indexes Improve Query Performance**

Indexes improve query performance by allowing the database management system (DBMS) to find rows more quickly without scanning the entire table. When a query uses a column with an index, the DBMS can utilize the index to quickly locate the rows that match the query criteria. This is particularly useful for large tables, where scanning every row would be inefficient.

**Example: Query Performance Improvement**

Suppose we have a large number of books in the Book table, and we want to find a specific book by its ISBN.

**Without Index:**

-- Query without index

SELECT \* FROM Book WHERE ISBN = '1234567890123';

In this case, without an index, the DBMS would perform a full table scan, checking each row to find the matching ISBN. This can be slow, especially with a large number of rows.

**With Index:**

-- Query with index

SELECT \* FROM Book WHERE ISBN = '1234567890123';

With the index idx\_ISBN on the ISBN column, the DBMS can quickly locate the specific ISBN in the index and then retrieve the corresponding row(s) from the table. This significantly reduces the time taken to execute the query.

**Dropping the Index**

If we decide that the index is no longer needed or if it's negatively impacting the performance of write operations (inserts, updates, deletes), we can drop the index.

-- Drop the index on the ISBN column in the Book table

DROP INDEX idx\_ISBN ON Book;

**Impact of Dropping the Index on Query Execution**

When the index is dropped, any subsequent queries that filter by the ISBN column will no longer benefit from the index. This means the DBMS will revert to performing a full table scan for such queries, which can significantly increase the query execution time, especially for large tables.

**Without the index:**

* Query execution time may increase, especially for SELECT operations involving large data sets.
* Write operations (INSERT, UPDATE, DELETE) may become faster because the overhead of maintaining the index is removed.

**With the index:**

* Query execution time is reduced for SELECT operations, particularly those involving filtering, sorting, or joining on the indexed columns.
* Write operations may be slower due to the need to update the index whenever the table data changes.

**Conclusion**

Indexes are a powerful tool for optimizing query performance in databases. However, they come with trade-offs in terms of storage space and the impact on write operations. The decision to create or drop an index should be based on careful consideration of query patterns and performance requirements.

**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 and Granting Privileges**

To create a new database user and assign specific privileges, we'll use the CREATE USER and GRANT commands.

**Example: Creating a New User and Granting Privileges**

-- Create a new user 'library\_user' with a password

CREATE USER 'library\_user'@'localhost' IDENTIFIED BY 'secure\_password';

-- Grant specific privileges to the user on the 'LibraryDB' database

GRANT SELECT, INSERT, UPDATE, DELETE ON LibraryDB.\* TO 'library\_user'@'localhost';

In this example:

* The CREATE USER command creates a new user named 'library\_user' with access from the 'localhost' host and a specified password ('secure\_password').
* The GRANT command gives the user privileges to perform SELECT, INSERT, UPDATE, and DELETE operations on all tables in the LibraryDB database.

**Revoking Specific Privileges and Dropping the User**

If we need to revoke certain privileges or drop the user, we can use the REVOKE and DROP USER commands.

**Example: Revoking Privileges and Dropping the User**

-- Revoke UPDATE and DELETE privileges from the user on the 'LibraryDB' database

REVOKE UPDATE, DELETE ON LibraryDB.\* FROM 'library\_user'@'localhost';

-- Drop the user 'library\_user'

DROP USER 'library\_user'@'localhost';

**Script Example**

Below is a complete script combining the above operations:

-- Create a new user

CREATE USER 'library\_user'@'localhost' IDENTIFIED BY 'secure\_password';

-- Grant privileges

GRANT SELECT, INSERT, UPDATE, DELETE ON LibraryDB.\* TO 'library\_user'@'localhost';

-- Revoke specific privileges

REVOKE UPDATE, DELETE ON LibraryDB.\* FROM 'library\_user'@'localhost';

-- Drop the user

DROP USER 'library\_user'@'localhost';

**Considerations**

* **Security**: Ensure that passwords are strong and stored securely.
* **Privileges**: Assign the minimum necessary privileges to users to adhere to the principle of least privilege, reducing the risk of unintended data modifications or access.
* **Audit**: Regularly review and audit user privileges to maintain security and compliance.

**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.

Here is a series of SQL statements demonstrating how to perform INSERT, UPDATE, and DELETE operations in a library database. Additionally, a BULK INSERT operation is included to load data from an external source.

**1. Inserting New Records into the Library Tables**

Assume we have the following tables in the library database:

* Books (BookID, Title, Author, PublishedYear, Genre)
* Members (MemberID, FirstName, LastName, DateOfBirth, MembershipDate)
* BorrowedBooks (BorrowID, BookID, MemberID, BorrowDate, ReturnDate)

**Example: Inserting New Records**

-- Inserting new books into the Books table

INSERT INTO Books (BookID, Title, Author, PublishedYear, Genre)

VALUES

(101, 'The Great Gatsby', 'F. Scott Fitzgerald', 1925, 'Fiction'),

(102, '1984', 'George Orwell', 1949, 'Dystopian'),

(103, 'To Kill a Mockingbird', 'Harper Lee', 1960, 'Fiction');

-- Inserting new members into the Members table

INSERT INTO Members (MemberID, FirstName, LastName, DateOfBirth, MembershipDate)

VALUES

(201, 'John', 'Doe', '1985-06-15', '2024-08-20'),

(202, 'Jane', 'Smith', '1990-04-23', '2024-08-21'),

(203, 'Alice', 'Johnson', '1975-12-05', '2024-08-22');

-- Inserting new records into the BorrowedBooks table

INSERT INTO BorrowedBooks (BorrowID, BookID, MemberID, BorrowDate, ReturnDate)

VALUES

(301, 101, 201, '2024-08-20', NULL),

(302, 102, 202, '2024-08-21', NULL),

(303, 103, 203, '2024-08-22', '2024-09-05');

**2. Updating Existing Records**

**Example: Updating Existing Records**

-- Updating the title and genre of a book in the Books table

UPDATE Books

SET Title = 'The Great Gatsby - Revised Edition', Genre = 'Classic Fiction'

WHERE BookID = 101;

-- Updating a member's last name and membership date in the Members table

UPDATE Members

SET LastName = 'Doe-Smith', MembershipDate = '2024-08-25'

WHERE MemberID = 201;

-- Updating the return date for a borrowed book in the BorrowedBooks table

UPDATE BorrowedBooks

SET ReturnDate = '2024-09-01'

WHERE BorrowID = 301;

**3. Deleting Records**

**Example: Deleting Records Based on Specific Criteria**

-- Deleting books published before the year 1950 from the Books table

DELETE FROM Books

WHERE PublishedYear < 1950;

-- Deleting members who joined the library before a specific date from the Members table

DELETE FROM Members

WHERE MembershipDate < '2024-01-01';

-- Deleting borrowed books records that were never returned (NULL ReturnDate) and borrowed more than 30 days ago

DELETE FROM BorrowedBooks

WHERE ReturnDate IS NULL AND BorrowDate < DATEADD(DAY, -30, GETDATE());

**4. Bulk Insert Operation**

**Example: Bulk Insert Operation to Load Data from an External Source**

Assume you have a CSV file named new\_books.csv located at C:\Data\new\_books.csv with the following structure:

* BookID, Title, Author, PublishedYear, Genre

-- Bulk Insert statement to load data from an external CSV file into the Books table

BULK INSERT Books

FROM 'C:\Data\new\_books.csv'

WITH (

FIELDTERMINATOR = ',',

ROWTERMINATOR = '\n',

FIRSTROW = 2 -- Skips the header row

);

**Explanation:**

1. **INSERT Statements**: Used to add new records to the Books, Members, and BorrowedBooks tables.
2. **UPDATE Statements**: Modify existing records based on specific conditions.
3. **DELETE Statements**: Remove records that meet particular criteria from the tables.
4. **BULK INSERT**: A fast method to load large amounts of data from an external file into the database.

This series of SQL operations covers a range of typical database maintenance tasks that might be required in a library management system.