ANJALI KUMARI

DAY-8 ,9   SQL

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.

Sure, let's create a hypothetical business scenario for this assignment:

Business Scenario: Online Bookstore

An online bookstore wants to create a database to manage its inventory, customers, and orders. The store sells various categories of books, such as fiction, non-fiction, self-help, etc. Customers can browse the books, add them to their shopping carts, and place orders. Each order can contain multiple books, and each book can be part of multiple orders. The database should also keep track of customer information and store details of authors and publishers.

Based on this scenario, we can identify the following entities:

Book: Represents each book available in the inventory.

Attributes: BookID (Primary Key), Title, Author, Publisher, Price, Genre, etc.

Customer: Represents each customer of the online bookstore.

Attributes: CustomerID (Primary Key), Name, Email, Address, Phone, etc.

Order: Represents each order placed by a customer.

Attributes: OrderID (Primary Key), OrderDate, TotalPrice, etc.

OrderItem: Represents each item within an order.

Attributes: OrderItemID (Primary Key), OrderID (Foreign Key), BookID (Foreign Key), Quantity, Price, etc.

Author: Represents each author of the books.

Attributes: AuthorID (Primary Key), AuthorName, Bio, etc.

Publisher: Represents each publisher of the books.

Attributes: PublisherID (Primary Key), PublisherName, Location, etc.

Now, let's define the relationships between these entities:

A Book can be written by one or more Authors.

A Book can have only one Publisher.

A Customer can place one or more Orders.

An Order is placed by only one Customer.

An Order can contain one or more OrderItems.

An OrderItem corresponds to exactly one Book.

An Author can write one or more Books.

A Publisher can publish one or more Books.

With these entities, attributes, and relationships identified, we can now create the ER diagram:

```
+-------------+          +-------------+
|    Book     |          |    Author    |
+-------------+          +-------------+
| BookID (PK) |<------| AuthorID (PK)|
| Title        |    |     | AuthorName    |
| Author       |---|     | Bio            |
```

```
| Publisher     |<--|        +-------------+
| Price          |                  |
| Genre          |                  |
+-------------+                  |
         |                              |
         |                              |
         |                              |
         |                              |
         v                              |
+-------------+                  |
|     Order        |                  |
+-------------+                  |
| OrderID (PK)|----------+
| CustomerID    |-----+
| OrderDate     |         |
| TotalPrice    |         |
+-------------+         |
         |                     |
         |                     |
         |                     |
         |                     |
         v                     |
+-------------+         |
| OrderItem     |         |
```

```
+-------------+          |
| OrderItemID |<----+
| OrderID (FK)|
| BookID (FK) |
| Quantity       |
| Price             |
+-------------+
          |
          |
          v
+-------------+
|     Customer    |
+-------------+
| CustomerID    |
| Name               |
| Email              |
| Address          |
| Phone              |
+-------------+
          |
          |
          v
+-------------+
|     Publisher |
```

```
+-------------+
| PublisherID |
| PublisherName|
| Location      |
+-------------+
```

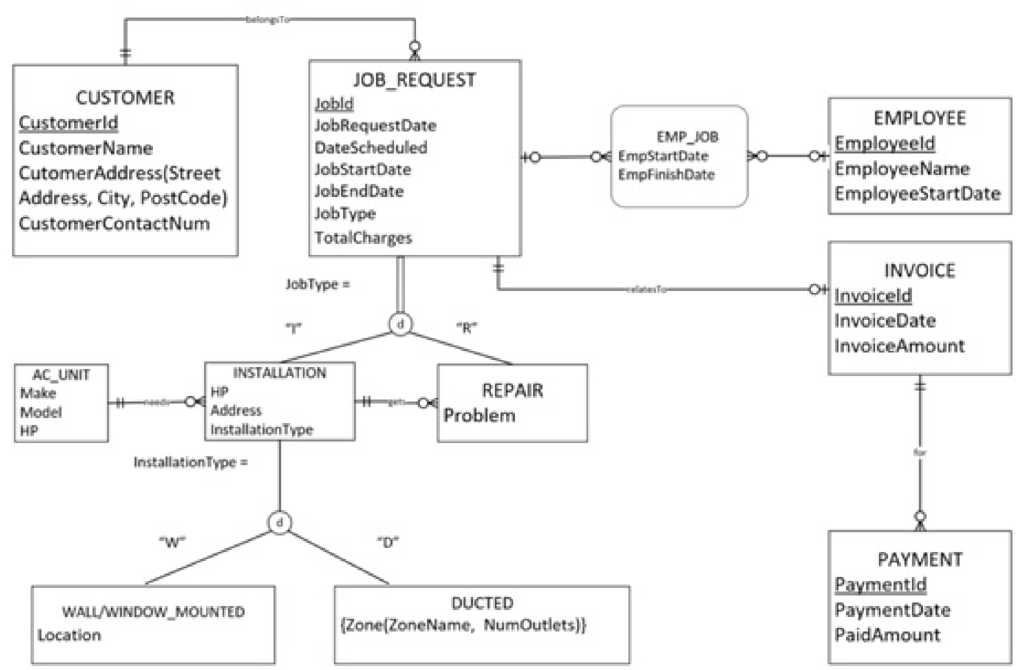
```sql
CREATE TABLE Book (
    BookID INT PRIMARY KEY,
    Title VARCHAR(255),
    AuthorID INT,
    PublisherID INT,
    Price DECIMAL(10, 2),
    Genre VARCHAR(50),
    FOREIGN KEY (AuthorID) REFERENCES Author(AuthorID),
    FOREIGN KEY (PublisherID) REFERENCES Publisher(PublisherID)
);

CREATE TABLE Customer (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(255),
    Email VARCHAR(255),
    Address VARCHAR(255),
    Phone VARCHAR(15)
    -- Additional columns as needed
```

);

-- Other tables (Order, OrderItem, Author, Publisher) would be created similarly.

Updated Sample solution for Assignment1 of COIT20247: ERD



==

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.

Sure, let's write SQL statements to create the database and tables for the library schema we designed earlier. We'll also include ALTER statements to modify the table structures and DROP statements to remove any redundant tables.

First, we'll create the database:

CREATE DATABASE LibraryDatabase;

USE LibraryDatabase;

Now, let's create the tables:

-- Create the Book table

```
CREATE TABLE Book (
    BookID INT PRIMARY KEY,
    Title VARCHAR(255),
    AuthorID INT,
    PublisherID INT,
    Price DECIMAL(10, 2),
    Genre VARCHAR(50),
    FOREIGN KEY (AuthorID) REFERENCES Author(AuthorID),
    FOREIGN KEY (PublisherID) REFERENCES Publisher(PublisherID)
);
```

-- Create the Customer table

```
CREATE TABLE Customer (
```

```sql
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(255),
    Email VARCHAR(255),
    Address VARCHAR(255),
    Phone VARCHAR(15)
    -- Additional columns as needed
);
```

-- Create the Order table

```sql
CREATE TABLE Order (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    TotalPrice DECIMAL(10, 2),
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)
    -- Additional columns as needed
);
```

-- Create the OrderItem table

```sql
CREATE TABLE OrderItem (
    OrderItemID INT PRIMARY KEY,
    OrderID INT,
    BookID INT,
    Quantity INT,
```

```
    Price DECIMAL(10, 2),
    FOREIGN KEY (OrderID) REFERENCES Order(OrderID),
    FOREIGN KEY (BookID) REFERENCES Book(BookID)
    -- Additional columns as needed
);
```

-- Create the Author table

```
CREATE TABLE Author (
    AuthorID INT PRIMARY KEY,
    AuthorName VARCHAR(255),
    Bio TEXT
    -- Additional columns as needed
);
```

-- Create the Publisher table

```
CREATE TABLE Publisher (
    PublisherID INT PRIMARY KEY,
    PublisherName VARCHAR(255),
    Location VARCHAR(255)
    -- Additional columns as needed
);
```

Now, let's say we realize we need to add an attribute to the Customer table to store membership status. We can do this using ALTER statement:

ALTER TABLE Customer

ADD MembershipStatus VARCHAR(50);

If, for example, we've identified a redundant table (let's say a table named Books), we can drop it using the DROP statement:

DROP TABLE Books;

These SQL statements should help you set up the database and tables according to the library schema we designed earlier, and make necessary modifications using ALTER and DROP statements.

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.

-- Create Authors table

CREATE TABLE Authors (

author_id INT PRIMARY KEY,

author_name VARCHAR(100) NOT NULL

);

-- Create Books table

CREATE TABLE Books (

book_id INT PRIMARY KEY,

title VARCHAR(255) NOT NULL,

author_id INT,

genre VARCHAR(100),

publication_year INT,

```sql
    isbn VARCHAR(20) UNIQUE NOT NULL,

    FOREIGN KEY (author_id) REFERENCES Authors(author_id)

);

-- Create Members table

CREATE TABLE Members (

    member_id INT PRIMARY KEY,

    member_name VARCHAR(100) NOT NULL,

    email VARCHAR(255) NOT NULL,

    phone VARCHAR(20) NOT NULL

);

-- Create Loans table

CREATE TABLE Loans (

    loan_id INT PRIMARY KEY,

    book_id INT,

    member_id INT,

    loan_date DATE NOT NULL,

    return_date DATE,

    returned BOOLEAN DEFAULT FALSE,

    FOREIGN KEY (book_id) REFERENCES Books(book_id),

    FOREIGN KEY (member_id) REFERENCES Members(member_id),

    CHECK (loan_date < return_date OR return_date IS NULL)

);
```

In this SQL representation:

Primary keys are defined using the PRIMARY KEY constraint.

Foreign keys are defined using the FOREIGN KEY constraint, referencing the primary key of another table.

NOT NULL constraints ensure that certain fields cannot be empty.

UNIQUE constraint ensures that the ISBN of books is unique.

CHECK constraint ensures that the loan date is before the return date, or the return date is null (indicating the book hasn't been returned yet).

The DEFAULT constraint sets a default value for the returned field in the Loans table.

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.

The ACID properties are a set of four characteristics that ensure the reliability and consistency of transactions in a database system:

Atomicity: Atomicity ensures that a transaction is treated as a single unit of work. This means that either all the operations within the transaction are successfully completed and committed to the database, or none of them are. There's no partial execution or incomplete state left in the database.

Consistency: Consistency ensures that the database remains in a valid state before and after the transaction. This means that all constraints, such as foreign key constraints or unique constraints, are enforced, and the integrity of the data is maintained. Transactions should transform the database from one valid state to another valid state.

Isolation: Isolation ensures that the concurrent execution of transactions does not result in any data inconsistency. Each transaction should appear to be executing in isolation, without being affected by other concurrently executing transactions. Isolation levels determine how transactions interact with each other, and they range from allowing maximum concurrency (but potentially introducing anomalies) to strictly serializing transactions.

Durability: Durability ensures that once a transaction is committed, its changes are permanent and survive system failures. Even in the event of a crash or power failure, the changes made by committed transactions should not be lost.

Now, let's simulate a transaction in SQL using locking and demonstrate different isolation levels:

```
-- Simulate a transaction with locking

BEGIN TRANSACTION;

-- Example SQL statements within the transaction

UPDATE Accounts SET balance = balance - 100 WHERE account_id = 123;

UPDATE Accounts SET balance = balance + 100 WHERE account_id = 456;

COMMIT TRANSACTION;
```

This SQL code initiates a transaction that transfers $100 from account 123 to account 456. The BEGIN TRANSACTION statement marks the beginning of the transaction, and the COMMIT TRANSACTION statement commits the transaction, making the changes permanent.

To demonstrate different isolation levels, you can set the isolation level before starting the transaction:

```
-- Set isolation level to SERIALIZABLE

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN TRANSACTION;

-- Transaction operations...

COMMIT TRANSACTION;
```

Replace SERIALIZABLE with other isolation levels like READ COMMITTED, REPEATABLE READ, or READ UNCOMMITTED to observe their effects on concurrency control. These isolation levels determine the degree to which transactions can see each other's intermediate states and changes.

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.

Certainly! Indexes play a crucial role in database performance by improving the speed of data retrieval operations such as SELECT queries. Let's demonstrate the creation of an index on a table and discuss its impact on query performance:

Suppose we have a table named Employees with columns employee_id, first_name, last_name, department, and salary.

-- Create Employees table

```
CREATE TABLE Employees (

    employee_id INT PRIMARY KEY,

    first_name VARCHAR(50),
```

```
    last_name VARCHAR(50),
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

Now, let's say we frequently run queries to retrieve employees based on their department:

```
-- Query without index
SELECT * FROM Employees WHERE department = 'IT';
```

To improve the performance of this query, we can create an index on the department column:

```
-- Create index on department column
CREATE INDEX idx_department ON Employees(department);
```

After creating the index, the query execution time for retrieving employees based on their department should be faster. This is because the index allows the database engine to quickly locate the rows that match the specified department value without scanning the entire table.

However, indexes come with a cost in terms of storage space and performance overhead during data modification operations (such as INSERT, UPDATE, and DELETE), as the index needs to be updated to reflect the changes. Therefore, it's essential to consider the trade-offs and create indexes judiciously based on the queries frequently executed against the table.

Now, let's drop the index and observe the impact on query execution:

```
-- Drop the index
DROP INDEX idx_department ON Employees;
```

After dropping the index, the query performance for retrieving employees based on their department may degrade, especially if the table size is significant and the query involves a large number of rows. Without the index, the database engine needs to perform a full table scan to find matching rows, which can be slower compared to using an index.

In summary, while indexes can significantly improve query performance, they should be used strategically, considering factors such as the frequency of queries and the impact on data modification 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.

Certainly! Let's create a new database user with specific privileges using the CREATE USER and GRANT commands, then revoke certain privileges using the REVOKE command, and finally drop the user.

Assuming we're working with a PostgreSQL database:

-- Create a new user

CREATE USER new_user WITH PASSWORD 'password';

-- Grant privileges to the user

GRANT SELECT, INSERT, UPDATE ON table_name TO new_user;

GRANT USAGE ON SCHEMA schema_name TO new_user;

-- Revoke certain privileges from the user

REVOKE UPDATE ON table_name FROM new_user;

-- Drop the user

DROP USER new_user;

Explanation:

CREATE USER is used to create a new database user.

GRANT is used to grant specific privileges to the user. In this example, the user is granted SELECT, INSERT, and UPDATE privileges on a specific table (table_name), and USAGE privilege on a schema (schema_name).

REVOKE is used to revoke specific privileges from the user. In this example, the UPDATE privilege on the table is revoked from the user.

DROP USER is used to drop the user from the database once their privileges are no longer needed.

Remember to replace new_user, table_name, and schema_name with the appropriate values specific to your database environment.

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.

Certainly! Here's a series of SQL statements to perform INSERT, UPDATE, and DELETE operations on library tables, along with a BULK INSERT operation to load data from an external source:

Assuming we have the following tables: Books, Authors, Members, and Loans.

INSERT new records:

-- Insert a new book

INSERT INTO Books (title, author_id, genre, publication_year, isbn)

VALUES ('The Great Gatsby', 1, 'Fiction', 1925, '9780743273565');

-- Insert a new author

INSERT INTO Authors (author_name)

VALUES ('F. Scott Fitzgerald');

-- Insert a new member

INSERT INTO Members (member_name, email, phone)

VALUES ('John Doe', 'john@example.com', '123-456-7890');

-- Insert a new loan

INSERT INTO Loans (book_id, member_id, loan_date, return_date, returned)

VALUES (1, 1, '2024-05-10', NULL, FALSE);

UPDATE existing records:

-- Update book information

UPDATE Books

SET genre = 'Classic'

WHERE title = 'The Great Gatsby';

-- Update member information

UPDATE Members

SET phone = '987-654-3210'

WHERE member_name = 'John Doe';

DELETE records based on specific criteria:

-- Delete a book

DELETE FROM Books

WHERE title = 'The Great Gatsby';

-- Delete a member

DELETE FROM Members

WHERE member_name = 'John Doe';

BULK INSERT from an external source:

Suppose you have a CSV file named books.csv containing book data with columns title, author_id, genre, publication_year, and isbn. You can use a BULK INSERT operation to load this data into the Books table:

-- Bulk insert from external source (assuming PostgreSQL)

```
COPY Books (title, author_id, genre, publication_year, isbn)

FROM '/path/to/books.csv'

DELIMITER ','

CSV HEADER;
```

Make sure to replace /path/to/books.csv with the actual path to your CSV file.

These SQL statements demonstrate how to perform basic data manipulation operations in a library database, including inserting new records, updating existing records, deleting records based on specific criteria, and bulk inserting data from an external source.
