Assignment Day-7

**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.**

To create an ER diagram that includes entities, relationships, attributes, and cardinality, and ensures proper normalization up to the third normal form (3NF), follow these steps:

**Step 1: Understand the Business Scenario**

Carefully read the given business scenario. Identify key components such as entities (e.g., customers, orders, products), relationships (e.g., customers place orders), and attributes (e.g., customer name, order date).

**Step 2: Identify Entities**

Entities are objects or concepts that can have data stored about them. For example:

Customer: Represents people who place orders.

Order: Represents orders placed by customers.

Product: Represents items available for sale.

OrderItem: Represents products within an order (if an order can contain multiple products).

**Step 3: Define Relationships**

Determine how entities are related:

A Customer places an Order (one-to-many).

An Order contains multiple OrderItems (one-to-many).

An OrderItem refers to a Product (many-to-one).

**Step 4: Identify Attributes**

Attributes are details about entities:

Customer: CustomerID (PK), Name, Email, Phone.

Order: OrderID (PK), OrderDate, CustomerID (FK).

Product: ProductID (PK), Name, Price.

OrderItem: OrderItemID (PK), OrderID (FK), ProductID (FK), Quantity.

**Step 5: Determine Cardinality**

Cardinality specifies the number of instances of one entity that can or must be associated with each instance of another entity:

One Customer can place many Orders (1:N).

One Order can contain many OrderItems (1:N).

Many OrderItems can refer to one Product (N:1).

**Step 6: Normalize the Data**

Ensure your design adheres to normalization rules:

1NF: Eliminate repeating groups, ensure each field contains only atomic values.

2NF: Ensure all non-key attributes are fully functionally dependent on the primary key.

3NF: Ensure no transitive dependencies (non-key attributes depend only on the primary key).

**Example ER Diagram**

**Customer (CustomerID, Name, Email, Phone)**

**1:M**

**Order (OrderID, OrderDate, CustomerID)**

**1:M**

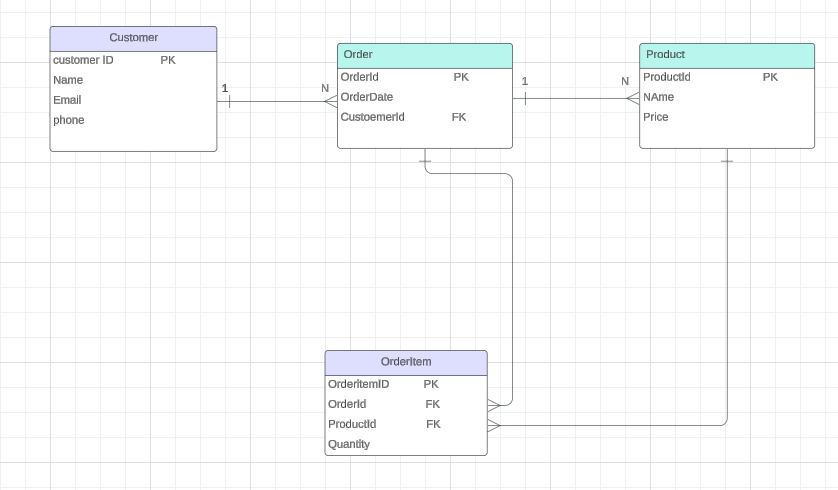
**OrderItem (OrderItemID, OrderID, ProductID, Quantity)**

**M:1**

**Product (ProductID, Name, Price)**

**Step 7: Create the ER Diagram**

ER diagram :---visualize the entities, relationships, and attributes.



**Example Diagram Steps**

Customer: Add CustomerID (PK), Name, Email, Phone.

Order: Add OrderID (PK), OrderDate, CustomerID (FK).

Product: Add ProductID (PK), Name, Price.

OrderItem: Add OrderItemID (PK), OrderID (FK), ProductID (FK), Quantity.

**Step 8: Validate Against Normal Forms**

Ensure each entity is in 1NF by having unique rows and atomic columns.

Check 2NF by verifying that each non-key attribute is fully dependent on the entire primary key.

Confirm 3NF by ensuring that all non-key attributes are dependent only on the primary key and not on other non-key attributes.

Final Steps

Review the ER diagram for completeness and accuracy.

Ensure all entities, relationships, attributes, and cardinalities are correctly represented.

Double-check normalization to confirm no redundancy or dependency issues.

By following these steps, you can create a comprehensive and normalized ER diagram for your business scenario.

**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. Authors**

author\_id (Primary Key, NOT NULL, UNIQUE)

first\_name (VARCHAR, NOT NULL)

last\_name (VARCHAR, NOT NULL)

**2. Books**

book\_id (Primary Key, NOT NULL, UNIQUE)

title (VARCHAR, NOT NULL)

isbn (VARCHAR, NOT NULL, UNIQUE)

publication\_year (YEAR, NOT NULL, CHECK(publication\_year >= 1000 AND publication\_year <= 9999))

author\_id (Foreign Key, NOT NULL, references Authors(author\_id))

**3. Members**

member\_id (Primary Key, NOT NULL, UNIQUE)

first\_name (VARCHAR, NOT NULL)

last\_name (VARCHAR, NOT NULL)

email (VARCHAR, NOT NULL, UNIQUE, CHECK(email LIKE '%@%'))

phone\_number (VARCHAR, UNIQUE)

**4. Loans**

loan\_id (Primary Key, NOT NULL, UNIQUE)

book\_id (Foreign Key, NOT NULL, references Books(book\_id))

member\_id (Foreign Key, NOT NULL, references Members(member\_id))

loan\_date (DATE, NOT NULL)

return\_date (DATE, CHECK(return\_date >= loan\_date))

CREATE TABLE Authors (

author\_id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(100) NOT NULL,

last\_name VARCHAR(100) NOT NULL

);

CREATE TABLE Books (

book\_id INT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(255) NOT NULL,

isbn VARCHAR(13) NOT NULL UNIQUE,

publication\_year YEAR NOT NULL CHECK (publication\_year >= 1000 AND publication\_year <= 9999),

author\_id INT NOT NULL,

FOREIGN KEY (author\_id) REFERENCES Authors(author\_id)

);

CREATE TABLE Members (

member\_id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(100) NOT NULL,

last\_name VARCHAR(100) NOT NULL,

email VARCHAR(255) NOT NULL UNIQUE CHECK (email LIKE '%@%'),

phone\_number VARCHAR(15) UNIQUE

);

CREATE TABLE Loans (

loan\_id INT AUTO\_INCREMENT PRIMARY KEY,

book\_id INT NOT NULL,

member\_id INT NOT NULL,

loan\_date DATE NOT NULL,

return\_date DATE CHECK (return\_date >= loan\_date),

FOREIGN KEY (book\_id) REFERENCES Books(book\_id),

FOREIGN KEY (member\_id) REFERENCES Members(member\_id)

);

**Explanation of Constraints**

**Primary Key:** Ensures each row in a table is unique.

**NOT NULL**: Ensures a column cannot have NULL values.

**UNIQUE:** Ensures all values in a column are different.

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

**Foreign Key:** Establishes a relationship between tables, ensuring referential integrity.

This schema provides a structured way to manage a library system, with clear relationships between books, authors, members, and loans.

**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 a Transaction**

The ACID properties ensure that database transactions are processed reliably and help maintain the integrity of the data. Here is a brief explanation of each property:

**Atomicity:** This ensures that all the operations within a transaction are completed successfully. If any part of the transaction fails, the entire transaction is rolled back, leaving the database in its previous state.

**Consistency:** This ensures that a transaction brings the database from one valid state to another valid state. It means the data must meet all validation rules, such as constraints, cascades, and triggers.

**Isolation:** This ensures that transactions are processed independently and transparently. Changes made by one transaction are not visible to other transactions until the transaction is committed. This property helps in maintaining data integrity in concurrent access.

**Durability:** This ensures that once a transaction is committed, the changes made are permanent and stored in the database, even in the event of a system failure.

**SQL Statements for Simulating a Transaction with Locking**

Here's how to simulate a transaction with locking and demonstrate different isolation levels:

Simulating a Transaction

-- Start a transaction

START TRANSACTION;

-- Update a record

UPDATE Accounts SET balance = balance - 100 WHERE account\_id = 1;

-- Introduce a delay to simulate long processing

-- DO SLEEP(5); -- This is pseudo-code to represent a delay

-- Update another record

UPDATE Accounts SET balance = balance + 100 WHERE account\_id = 2;

-- Commit the transaction

COMMIT;

Demonstrating Isolation Levels for Concurrency Control

**1. Read Uncommitted**

In this isolation level, transactions can read data that has been modified by other transactions but not yet committed. This can lead to dirty reads.

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

START TRANSACTION;

-- Read data (possible dirty read)

SELECT \* FROM Accounts WHERE account\_id = 1;

-- Commit or rollback transaction

COMMIT;

**2. Read Committed**

In this isolation level, transactions can only read data that has been committed by other transactions, preventing dirty reads.

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

START TRANSACTION;

-- Read data (only committed data)

SELECT \* FROM Accounts WHERE account\_id = 1;

-- Commit or rollback transaction

COMMIT;

**3. Repeatable Read**

In this isolation level, all reads within the transaction will return the same data, preventing non-repeatable reads. However, phantom reads can still occur.

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

-- Read data (same data within the transaction)

SELECT \* FROM Accounts WHERE account\_id = 1;

-- Commit or rollback transaction

COMMIT;

**4. Serializable**

This is the highest isolation level, ensuring complete isolation. It prevents dirty reads, non-repeatable reads, and phantom reads by locking the data set.

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

START TRANSACTION;

-- Read data (fully isolated)

SELECT \* FROM Accounts WHERE account\_id = 1;

-- Commit or rollback transaction

COMMIT;

**Explanation of Isolation Levels and Concurrency Control**

**Read Uncommitted:** Allows dirty reads, offering the least isolation but the highest performance.

**Read Committed:** Prevents dirty reads, ensuring that only committed data is read.

**Repeatable Read:** Ensures that if a transaction reads a row, it will read the same value for that row if read again, preventing non-repeatable reads.

**Serializable:** Ensures complete isolation by locking the entire data set being read, which can lead to reduced performance but guarantees consistency.

These isolation levels help manage concurrency control, balancing between data integrity and system performance.

**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 for the library system and then create the tables reflecting the schema designed earlier.

**1. Create the Database**

CREATE DATABASE LibraryDB;

USE LibraryDB;

**2. Create the Tables**

CREATE TABLE Authors (

author\_id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(100) NOT NULL,

last\_name VARCHAR(100) NOT NULL

);

CREATE TABLE Books (

book\_id INT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(255) NOT NULL,

isbn VARCHAR(13) NOT NULL UNIQUE,

publication\_year YEAR NOT NULL CHECK (publication\_year >= 1000 AND publication\_year <= 9999),

author\_id INT NOT NULL,

FOREIGN KEY (author\_id) REFERENCES Authors(author\_id)

);

CREATE TABLE Members (

member\_id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(100) NOT NULL,

last\_name VARCHAR(100) NOT NULL,

email VARCHAR(255) NOT NULL UNIQUE CHECK (email LIKE '%@%'),

phone\_number VARCHAR(15) UNIQUE

);

CREATE TABLE Loans (

loan\_id INT AUTO\_INCREMENT PRIMARY KEY,

book\_id INT NOT NULL,

member\_id INT NOT NULL,

loan\_date DATE NOT NULL,

return\_date DATE CHECK (return\_date >= loan\_date),

FOREIGN KEY (book\_id) REFERENCES Books(book\_id),

FOREIGN KEY (member\_id) REFERENCES Members(member\_id)

);

Using ALTER Statements to Modify Table Structures

Here, we will add a new column to the Books table and modify an existing column in the Members table.

**1. Add a New Column to the Books Table**

ALTER TABLE Books

ADD COLUMN genre VARCHAR(50);

**2. Modify an Existing Column in the Members Table**

ALTER TABLE Members

MODIFY COLUMN phone\_number VARCHAR(20);

Using DROP Statements to Remove a Redundant Table

Suppose we have a redundant table named Publishers that needs to be removed.

**1. Drop the Publishers Table**

DROP TABLE IF EXISTS Publishers;

Complete SQL Script

Combining all the above SQL statements, the complete script will look like this:

-- Create a new database

CREATE DATABASE LibraryDB;

USE LibraryDB;

-- Create the Authors table

CREATE TABLE Authors (

author\_id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(100) NOT NULL,

last\_name VARCHAR(100) NOT NULL

);

-- Create the Books table

CREATE TABLE Books (

book\_id INT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(255) NOT NULL,

isbn VARCHAR(13) NOT NULL UNIQUE,

publication\_year YEAR NOT NULL CHECK (publication\_year >= 1000 AND publication\_year <= 9999),

author\_id INT NOT NULL,

FOREIGN KEY (author\_id) REFERENCES Authors(author\_id)

);

-- Create the Members table

CREATE TABLE Members (

member\_id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(100) NOT NULL,

last\_name VARCHAR(100) NOT NULL,

email VARCHAR(255) NOT NULL UNIQUE CHECK (email LIKE '%@%'),

phone\_number VARCHAR(15) UNIQUE

);

-- Create the Loans table

CREATE TABLE Loans (

loan\_id INT AUTO\_INCREMENT PRIMARY KEY,

book\_id INT NOT NULL,

member\_id INT NOT NULL,

loan\_date DATE NOT NULL,

return\_date DATE CHECK (return\_date >= loan\_date),

FOREIGN KEY (book\_id) REFERENCES Books(book\_id),

FOREIGN KEY (member\_id) REFERENCES Members(member\_id)

);

-- Add a new column to the Books table

ALTER TABLE Books

ADD COLUMN genre VARCHAR(50);

-- Modify an existing column in the Members table

ALTER TABLE Members

MODIFY COLUMN phone\_number VARCHAR(20);

-- Drop the redundant Publishers table if it exists

DROP TABLE IF EXISTS Publishers;

This script creates a new database, defines the tables and their relationships, modifies table structures, and removes a redundant table.

**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**

Indexes are used to improve the speed of data retrieval operations on a database table at the cost of additional writes and storage space. Let's demonstrate the creation of an index on the Books table's title column.

**1. Create an Index**

CREATE INDEX idx\_title ON Books(title);

**How an Index Improves Query Performance**

Indexes work similarly to an index in a book. Instead of scanning the entire table to find the rows that match a query, the database can use the index to locate the rows more quickly. This is especially useful for large tables. Let's compare query performance with and without the index.

**Query Without Index**

First, let's run a query to find books by title without using the index.

-- This is a query to find a book by title

SELECT \* FROM Books WHERE title = 'Effective Java';

Without an index, the database will perform a full table scan to find the matching rows, which can be slow if the table contains a large number of records.

Query With Index

Now, let's run the same query after creating the index.

-- This query will use the index to find the book by title

SELECT \* FROM Books WHERE title = 'Effective Java';

With the index in place, the database can quickly locate the row(s) that match the title 'Effective Java' using the idx\_title index, significantly reducing the query execution time.

**Dropping the Index**

To remove the index, we use the DROP INDEX statement. Let's see how the removal of the index affects query performance.

**2. Drop the Index**

DROP INDEX idx\_title ON Books;

Impact on Query Execution

After dropping the index, run the same query again:

-- Query to find a book by title after dropping the index

SELECT \* FROM Books WHERE title = 'Effective Java';

Without the index, the database will revert to performing a full table scan to locate the matching rows, leading to longer query execution times, particularly for large datasets.

**Summary**

**With Index**

**Query Execution:** Fast, as the database uses the index to locate rows quickly.

**Performance:** Improved for read operations involving indexed columns.

**Overhead:** Slightly increased storage and slower write operations due to maintaining the index.

Without Index

**Query Execution:** Slower, as the database performs a full table scan to locate rows.

**Performance:** Decreased for read operations on large tables without the index.

**Overhead:** Reduced storage and faster write operations as no index maintenance is required.

**Complete SQL Script**

Here’s a complete script demonstrating the creation and dropping of an index:

-- Create the Books table

CREATE TABLE Books (

book\_id INT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(255) NOT NULL,

isbn VARCHAR(13) NOT NULL UNIQUE,

publication\_year YEAR NOT NULL CHECK (publication\_year >= 1000 AND publication\_year <= 9999),

author\_id INT NOT NULL,

FOREIGN KEY (author\_id) REFERENCES Authors(author\_id)

);

-- Create an index on the title column

CREATE INDEX idx\_title ON Books(title);

-- Query using the index

SELECT \* FROM Books WHERE title = 'Effective Java';

-- Drop the index

DROP INDEX idx\_title ON Books;

-- Query without the index

SELECT \* FROM Books WHERE title = 'Effective Java';

This script demonstrates the creation of an index, its impact on query performance, and the effect of dropping the index.

**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**

Let's create a new database user and grant specific privileges to that user.

**1. Create a New User**

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

**2. Grant Specific Privileges**

We will grant the library\_user privileges to select, insert, update, and delete data from the LibraryDB database.

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

Script to Revoke Certain Privileges and Drop the User

Now, we will write a script to revoke certain privileges from the user and then drop the user.

**1. Revoke Privileges**

Let's revoke the DELETE privilege from library\_user.

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

**2. Drop the User**

Finally, drop the user.

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

**Complete SQL Script**

Here is the complete script demonstrating the creation of a new user, granting specific privileges, revoking certain privileges, and dropping the user.

-- Create a new user

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

-- Grant specific privileges to the user

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

-- Revoke the DELETE privilege from the user

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

-- Drop the user

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

**Explanation**

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

**GRANT:** This command grants specific privileges to the user on the specified database. In this case, we grant SELECT, INSERT, UPDATE, and DELETE privileges on all tables in the LibraryDB database.

**REVOKE:** This command revokes specific privileges from the user. Here, we revoke the DELETE privilege from library\_user.

**DROP USER:** This command deletes the user from the database.

Using these commands, you can manage database users and their privileges effectively.

**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.**

-- Insert into Authors

INSERT INTO Authors (first\_name, last\_name) VALUES ('Joshua', 'Bloch');

INSERT INTO Authors (first\_name, last\_name) VALUES ('Robert', 'Martin');

-- Insert into Books

INSERT INTO Books (title, isbn, publication\_year, author\_id, genre)

VALUES ('Effective Java', '9780134685991', 2018, 1, 'Programming');

INSERT INTO Books (title, isbn, publication\_year, author\_id, genre)

VALUES ('Clean Code', '9780132350884', 2008, 2, 'Programming');

-- Insert into Members

INSERT INTO Members (first\_name, last\_name, email, phone\_number)

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

INSERT INTO Members (first\_name, last\_name, email, phone\_number)

VALUES ('Jane', 'Smith', 'jane.smith@example.com', '098-765-4321');

-- Insert into Loans

INSERT INTO Loans (book\_id, member\_id, loan\_date, return\_date)

VALUES (1, 1, '2024-05-01', '2024-05-15');

INSERT INTO Loans (book\_id, member\_id, loan\_date, return\_date)

VALUES (2, 2, '2024-05-10', '2024-05-24');

-- Update Author Name

UPDATE Authors

SET last\_name = 'Uncle Bob'

WHERE author\_id = 2;

-- Update Book Genre

UPDATE Books

SET genre = 'Software Engineering'

WHERE book\_id = 1;

-- Delete a Member

DELETE FROM Members

WHERE email = 'john.doe@example.com';

-- Delete Old Loans

DELETE FROM Loans

WHERE return\_date < '2024-05-01';

-- Create Authors for Bulk Insert (if they don't already exist)

INSERT INTO Authors (author\_id, first\_name, last\_name)

VALUES (3, 'Andy', 'Hunt'), (4, 'Martin', 'Fowler');

-- Bulk Insert into Books

LOAD DATA INFILE '/path/to/books.csv'

INTO TABLE Books

FIELDS TERMINATED BY ','

LINES TERMINATED BY '\n'

IGNORE 1 LINES

(title, isbn, publication\_year, author\_id, genre);

This script demonstrates how to insert new records, update existing records, delete records based on specific criteria, and perform bulk insert operations.