SQL Assignment 2

Question 1 For an online purchasing database, create entity relationship diagrams. Create a database object from your entity diagram.

Answer:- Here's a breakdown for creating entity relationship diagrams and a database object for an online purchasing database:

Entity-Relationship Diagram (ERD) for an Online Purchasing Database

An ERD visually represents the relationships between different entities in a database. Here are the core entities and their relationships for an online purchasing database:

- **Entities:**
  - **Customer:**
    - `customer_id` (Primary Key)
    - `first_name`
    - `last_name`
    - `email`
    - `password_hash`
    - `address`
    - `city`
    - `state`
    - `zip_code`
    - `phone_number`
  - **Product:**
    - `product_id` (Primary Key)
    - `name`
    - `description`
    - `price`
    - `stock_quantity`
    - `category_id` (Foreign Key to Category)
  - **Category:**
    - `category_id` (Primary Key)
    - `name`
    - `description`
  - **Order:**
    - `order_id` (Primary Key)
    - `customer_id` (Foreign Key to Customer)

        - `order_date`
        - `total_amount`
        - `status` (e.g., Pending, Shipped, Delivered, Cancelled)
    - **Order_Item:** (Junction table for many-to-many relationship between Order and Product)
        - `order_item_id` (Primary Key)
        - `order_id` (Foreign Key to Order)
        - `product_id` (Foreign Key to Product)
        - `quantity`
        - `unit_price`
    - **Payment:**
        - `payment_id` (Primary Key)
        - `order_id` (Foreign Key to Order)
        - `payment_date`
        - `amount`
        - `payment_method` (e.g., Credit Card, PayPal)
        - `transaction_id` (if applicable)
- **Relationships:**
    - **Customer** to **Order:** One-to-Many (One customer can place many orders)
    - **Order** to **Order_Item:** One-to-Many (One order can have many order items)
    - **Product** to **Order_Item:** One-to-Many (One product can be in many order items)
    - **Category** to **Product:** One-to-Many (One category can have many products)
    - **Order** to **Payment:** One-to-One (One order typically has one payment, though sometimes multiple if partially paid or refunded)

Creating a Database Object from Your Entity Diagram

Once you have your ERD, you can translate it into a database schema using SQL Data Definition Language (DDL). Here's an example of how to create the tables (database objects) in SQL based on the ERD above. This example uses a generic SQL syntax, which might need slight adjustments depending on your specific database system (e.g., MySQL, PostgreSQL, SQL Server, Oracle).

```
-- Create the Category table
CREATE TABLE Category (
    category_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    description TEXT
);
```

```
-- Create the Product table
CREATE TABLE Product (
    product_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    price DECIMAL(10, 2) NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    category_id INT,
    FOREIGN KEY (category_id) REFERENCES Category(category_id)
);

-- Create the Customer table
CREATE TABLE Customer (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    address VARCHAR(255),
    city VARCHAR(100),
    state VARCHAR(100),
    zip_code VARCHAR(20),
    phone_number VARCHAR(20)
);

-- Create the Order table
CREATE TABLE "Order" ( -- "Order" is often a reserved keyword, so it's good
practice to quote it or choose a different name like 'CustomerOrder'
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT NOT NULL,
    order_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10, 2) NOT NULL,
    status VARCHAR(50) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES Customer(customer_id)
);

-- Create the Order_Item table (for many-to-many relationship)
CREATE TABLE Order_Item (
    order_item_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    unit_price DECIMAL(10, 2) NOT NULL,
```

```
    FOREIGN KEY (order_id) REFERENCES "Order"(order_id),
    FOREIGN KEY (product_id) REFERENCES Product(product_id),
    UNIQUE (order_id, product_id) -- Ensures a product is only listed once per
order item
);

-- Create the Payment table
CREATE TABLE Payment (
    payment_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT NOT NULL,
    payment_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    amount DECIMAL(10, 2) NOT NULL,
    payment_method VARCHAR(50) NOT NULL,
    transaction_id VARCHAR(255),
    FOREIGN KEY (order_id) REFERENCES "Order"(order_id)
);
```

Key Considerations:

- **Data Types:** The data types (e.g., `INT`, `VARCHAR`, `DECIMAL`, `DATETIME`) should be chosen based on the type of data they will store and the specific database system you are using.
- **Primary Keys (PK):** Uniquely identify each record in a table. `AUTO_INCREMENT` (or equivalent like `SERIAL` in PostgreSQL) is common for generating unique IDs automatically.
- **Foreign Keys (FK):** Establish relationships between tables, ensuring referential integrity. They link a column in one table to the primary key of another table.
- **Constraints:**
    - `NOT NULL`: Ensures a column cannot have a `NULL` value.
    - `UNIQUE`: Ensures all values in a column are different.
    - `DEFAULT`: Assigns a default value if none is provided.
- **Reserved Keywords:** Be mindful of reserved keywords in your specific SQL dialect (e.g., "Order" is often a reserved word, so it's best to enclose it in quotes or rename the table).

This provides a solid foundation for your online purchasing database. You can further refine it by adding more specific attributes, indexes for performance, and other constraints as needed for your application.

Question 2 Create a SQL store process to register the use of the database, complete it with proper validation and transaction rollback and commit.

Answer:- Here's a SQL stored procedure to register a user in the database, including validation, transaction rollback, and commit. This example assumes a `Users` table exists with at least `username`, `password_hash`, and `email` columns.

```
DELIMITER //

CREATE PROCEDURE RegisterUser(
    IN p_username VARCHAR(100),
    IN p_password VARCHAR(255),
    IN p_email VARCHAR(255)
)
BEGIN
    -- Declare a variable to hold the count of existing users
    DECLARE user_count INT;
    -- Declare a variable to store the hashed password
    DECLARE v_password_hash VARCHAR(255);

    -- Start a transaction
    START TRANSACTION;

    -- Basic validation: Check if username or email already exists
    SELECT COUNT(*) INTO user_count
    FROM Users
    WHERE username = p_username OR email = p_email;

    IF user_count > 0 THEN
        -- If user or email exists, rollback the transaction and signal an error
        ROLLBACK;
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Username or email already
exists.';
    ELSE
        -- Hash the password (you should use a strong hashing algorithm in a real
application)
        -- For demonstration, we'll use a simple SHA2, but consider bcrypt or Argon2 for
production.
        SET v_password_hash = SHA2(p_password, 256);

        -- Insert the new user into the Users table
        INSERT INTO Users (username, password_hash, email, registration_date)
        VALUES (p_username, v_password_hash, p_email, NOW());

        -- Check if the insert was successful
        IF ROW_COUNT() = 1 THEN
```

```
        -- Commit the transaction if successful
        COMMIT;
        SELECT 'User registered successfully.' AS Message;
    ELSE
        -- Rollback if the insert failed for some reason
        ROLLBACK;
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'User registration failed.';
    END IF;
END IF;

END //

DELIMITER ;
```

Explanation:

- **`DELIMITER // ... DELIMITER ;`**: This changes the statement delimiter from `;` to `//` temporarily, allowing you to use semicolons within the stored procedure body.
- **`CREATE PROCEDURE RegisterUser(...)`**: Defines the stored procedure named `RegisterUser` with three input parameters: `p_username`, `p_password`, and `p_email`.
- **`DECLARE user_count INT;`**: Declares a local variable `user_count` to store the number of matching users.
- **`DECLARE v_password_hash VARCHAR(255);`**: Declares a local variable `v_password_hash` to store the hashed password.
- **`START TRANSACTION;`**: Initiates a transaction. All subsequent SQL statements within the transaction will either all succeed or all fail together.
- **Validation Check**:
  - `SELECT COUNT(*) INTO user_count FROM Users WHERE username = p_username OR email = p_email;`: This query checks if a user with the provided `p_username` or `p_email` already exists in the `Users` table.
  - `IF user_count > 0 THEN ... ELSE ... END IF;`: If a duplicate is found, the transaction is `ROLLBACK`ed, and a `SIGNAL SQLSTATE '45000'` is raised with a custom error message. This effectively stops the procedure and indicates an error.
- **Password Hashing**:
  - `SET v_password_hash = SHA2(p_password, 256);`: This line demonstrates password hashing using `SHA2`. **In a real-world application, you should use more robust hashing algorithms like `bcrypt` or `Argon2` for security.**
- **Insert User**:

    - `INSERT INTO Users (username, password_hash, email, registration_date) VALUES (p_username, v_password_hash, p_email, NOW());`: Inserts the new user's details into the `Users` table. `NOW()` sets the `registration_date` to the current timestamp.
- **Transaction Commit/Rollback**:
    - `IF ROW_COUNT() = 1 THEN ... ELSE ... END IF;`: `ROW_COUNT()` returns the number of rows affected by the previous SQL statement. If exactly one row was inserted, the `COMMIT` statement makes the changes permanent in the database. Otherwise, the `ROLLBACK` statement undoes all changes made since `START TRANSACTION;`.
    - `SELECT 'User registered successfully.' AS Message;`: A success message is returned if the user is registered.
    - `SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'User registration failed.';`: An error message is returned if the registration fails.

To use this stored procedure:

1. Make sure you have a `Users` table created with appropriate columns (e.g., `user_id` (PK), `username`, `password_hash`, `email`, `registration_date`).
2. Execute the `CREATE PROCEDURE` statement in your SQL client.
3. Call the procedure with the desired user details:
   CALL RegisterUser('newuser', 'securepassword123', 'newuser@example.com');

Question 3 List the SQL aggregate function and demonstrate how to utilize it.

Answer:- Here are some common SQL aggregate functions and demonstrations of how to utilize them:

SQL Aggregate Functions

Aggregate functions perform a calculation on a set of rows and return a single summary value. They are often used with the `GROUP BY` clause to perform calculations on subsets of rows.

Here are some of the most commonly used SQL aggregate functions:

- `COUNT()`: Returns the number of rows that match a specified criterion.
- `SUM()`: Calculates the sum of a set of values.
- `AVG()`: Calculates the average of a set of values.
- `MIN()`: Returns the minimum value in a set.
- `MAX()`: Returns the maximum value in a set.

Demonstration of Utilization

Let's assume we have a table named `Orders` with the following columns:

- `order_id` (Primary Key)
- `customer_id`
- `order_date`
- `total_amount`
- `status`

Here's how you can use these aggregate functions:
`COUNT()` - Counting the total number of orders:

To find out how many orders are in the `Orders` table:
SELECT COUNT(order_id) AS TotalOrders
FROM Orders;
To count the number of orders for a specific customer (e.g., `customer_id = 101`):
SELECT COUNT(order_id) AS CustomerOrders
FROM Orders

1. WHERE customer_id = 101;

`SUM()` - Calculating the total revenue:

To find the total amount of all orders:
SELECT SUM(total_amount) AS TotalRevenue
FROM Orders;
To find the total revenue generated by each customer:
SELECT customer_id, SUM(total_amount) AS CustomerRevenue
FROM Orders

2. GROUP BY customer_id;

`AVG()` - Calculating the average order amount:

To find the average amount of all orders:
SELECT AVG(total_amount) AS AverageOrderAmount

FROM Orders;
To find the average order amount for orders with a 'Delivered' status:
SELECT AVG(total_amount) AS AverageDeliveredOrderAmount
FROM Orders

3. WHERE status = 'Delivered';

`MIN()` - Finding the smallest order amount:

To find the minimum `total_amount` among all orders:
SELECT MIN(total_amount) AS SmallestOrderAmount
FROM Orders;
To find the smallest order amount placed by each customer:
SELECT customer_id, MIN(total_amount) AS SmallestOrderAmount
FROM Orders

4. GROUP BY customer_id;

`MAX()` - Finding the largest order amount:

To find the maximum `total_amount` among all orders:
SELECT MAX(total_amount) AS LargestOrderAmount
FROM Orders;
To find the largest order amount placed on a specific date:
SELECT MAX(total_amount) AS LargestOrderOnDate
FROM Orders

5. WHERE order_date = ' 2025-09-29 ';

These examples demonstrate the basic usage of SQL aggregate functions. You can combine them with `WHERE` clauses for filtering, and `GROUP BY` clauses for grouping results, to perform powerful data analysis.

Question 4 In SQL, create a pivot query.

Answer:- Here's how to create a pivot query in SQL, along with an example

SQL Pivot Query

A pivot query in SQL transforms rows into columns, allowing you to rotate a table-valued expression by turning the unique values from one column into multiple columns and performing aggregations on other columns. This is particularly useful for summarizing data and presenting it in a more readable, cross-tabulated format.
While some database systems (like SQL Server and Oracle) have a dedicated `PIVOT` clause, you can achieve similar results in other databases (like MySQL and PostgreSQL) using conditional aggregation with `CASE` statements and `GROUP BY`.

Demonstration with Conditional Aggregation (for broader compatibility)

Let's assume you have a table named `Sales` with the following data:

Sales Table

SaleID	Region	Product	Amount
1	East	A	100
2	West	B	150
3	East	B	200
4	North	A	120
5	West	A	180
6	East	C	90

You want to see the total sales amount for each region, broken down by product.

Pivot Query using Conditional Aggregation:

```
SELECT
   Region,
   SUM(CASE WHEN Product = 'A' THEN Amount ELSE 0 END) AS Product_A_Sales,
   SUM(CASE WHEN Product = 'B' THEN Amount ELSE 0 END) AS Product_B_Sales,
   SUM(CASE WHEN Product = 'C' THEN Amount ELSE 0 END) AS Product_C_Sales
FROM
   Sales
GROUP BY
   Region
ORDER BY
   Region;
```

Explanation:

- **`SELECT Region, ...`**: We select the `Region` column, as this will be our grouping column.
- **`SUM(CASE WHEN Product = 'A' THEN Amount ELSE 0 END) AS Product_A_Sales`**:
    - For each `Region`, we are conditionally summing the `Amount`.
    - `CASE WHEN Product = 'A' THEN Amount ELSE 0 END`: If the `Product` for a given row is 'A', its `Amount` is included in the sum; otherwise, 0 is added.
    - `AS Product_A_Sales`: This assigns an alias to the new column,

making it descriptive.

- `FROM Sales`: Specifies the table from which to retrieve data.
- `GROUP BY Region`: This is crucial. It groups the rows by `Region`, so the `SUM` functions calculate the total for each product within each region.
- `ORDER BY Region`: Sorts the results by region for better readability.

Result of the Pivot Query:

Region	Product_A_Sales	Product_B_Sales	Product_C_Sales
East	100	200	90
North	120	0	0
West	180	150	0

Demonstration with `PIVOT` Clause (SQL Server/Oracle Syntax)

If you are using a database system that supports the `PIVOT` clause (like SQL Server), the query would look like this:

```
SELECT Region, [A], [B], [C]
FROM
(
   SELECT Region, Product, Amount
   FROM Sales
) AS SourceTable
PIVOT
(
   SUM(Amount)
   FOR Product IN ([A], [B], [C])
) AS PivotTable;
```

Explanation of `PIVOT` Clause:

- `FROM (SELECT Region, Product, Amount FROM Sales) AS SourceTable`: This subquery provides the base data for the pivot operation.
- `PIVOT (SUM(Amount) FOR Product IN ([A], [B], [C])) AS PivotTable`:
  - `SUM(Amount)`: This is the aggregate function applied to the `Amount` column.
  - `FOR Product IN ([A], [B], [C])`: This specifies that the unique values from the `Product` column ('A', 'B', 'C') will become the new column headers.

The `PIVOT` clause often provides a more concise and readable way to perform pivoting, but conditional aggregation is a widely supported alternative across different SQL databases.

Question 5 With an example, describe how to join in SQL.

Answer:- Here's a description of how to join in SQL, along with an example:

SQL Joins

In SQL, a `JOIN` clause is used to combine rows from two or more tables based on a related column between them. This is a fundamental operation for retrieving data that is spread across multiple tables in a relational database.
There are several types of joins:

- **`INNER JOIN`**: Returns only the rows where there is a match in *both* tables. This is the most common type of join.
- **`LEFT JOIN` (or `LEFT OUTER JOIN`)**: Returns all rows from the left table, and the matching rows from the right table. If there is no match in the right table, `NULL` values are returned for the right table's columns.
- **`RIGHT JOIN` (or `RIGHT OUTER JOIN`)**: Returns all rows from the right table, and the matching rows from the left table. If there is no match in the left table, `NULL` values are returned for the left table's columns.
- **`FULL JOIN` (or `FULL OUTER JOIN`)**: Returns all rows when there is a match in one of the tables. It returns rows from both tables, with `NULL` values for the columns where there is no match.
- **`CROSS JOIN`**: Returns the Cartesian product of the two tables, meaning it combines each row from the first table with every row from the second table. This type of join does not require a join condition.

Demonstration of Utilization (using `INNER JOIN`)

Let's assume we have two tables:

1. **`Customers` Table:**
    - `customer_id` (Primary Key)
    - `first_name`
    - `last_name`
    - `email`
2. **`Orders` Table:**
    - `order_id` (Primary Key)
    - `customer_id` (Foreign Key referencing `Customers.customer_id`)
    - `order_date`

    - `total_amount`

We want to retrieve a list of all orders along with the name of the customer who placed each order.

`INNER JOIN` Example:

```
SELECT
    Orders.order_id,
    Customers.first_name,
    Customers.last_name,
    Orders.order_date,
    Orders.total_amount
FROM
    Orders
INNER JOIN
    Customers ON Orders.customer_id = Customers.customer_id;
```

Explanation:

- **`SELECT Orders.order_id, Customers.first_name, ...`**: We specify the columns we want to retrieve from both the `Orders` and `Customers` tables. It's good practice to prefix column names with the table name (e.g., `Orders.order_id`) to avoid ambiguity if both tables have columns with the same name.
- **`FROM Orders`**: This indicates that `Orders` is our primary (left) table.
- **`INNER JOIN Customers ON Orders.customer_id = Customers.customer_id`**:
    - `INNER JOIN Customers`: We are joining the `Orders` table with the `Customers` table.
    - `ON Orders.customer_id = Customers.customer_id`: This is the join condition. It tells SQL to match rows from `Orders` with rows from `Customers` where the `customer_id` in the `Orders` table is equal to the `customer_id` in the `Customers` table.

Result of the `INNER JOIN` Query (Example Data):

If you had the following data

Customers Table

customer_id	first_name	last_name	email
1	Alice	Smith	alice.s@example.com
2	Bob	Johnson	bob.j@exampl

			e.com

Orders Table

order_id	customer_id	order_date	total_amount
101	1	2025-09-20	50.00
102	2	2025-09-21	75.50
103	1	2025-09-22	120.00

The `INNER JOIN` would produce a result similar to this:

order_id	first_name	last_name	order_date	total_amount
101	Alice	Sm…	2025…	50.00
102	Bob	Jo…	2025…	75.50
103	Alice	Sm…	2025…	120.00

This example demonstrates how `INNER JOIN` effectively combines related data from two tables into a single, more comprehensive result set.

Question 6 How to locate the 4th highest value in a column in a row. Create your table.

Answer:- Here's how to locate the 4th highest value in a column in SQL, along with a sample table:

Sample Table: `Scores`

StudentID	Score
101	95
102	88
103	92
104	78
105	95
106	85

107	90
108	78
109	88
110	91

SQL Query to Locate the 4th Highest Value

To find the 4th highest value, you can use window functions like `DENSE_RANK()` or `ROW_NUMBER()` in combination with a subquery. `DENSE_RANK()` is often preferred when you want to treat ties as a single rank (e.g., if two students have the same 3rd highest score, they both get rank 3, and the next unique score gets rank 4).

Here's an example using `DENSE_RANK()`:

```
SELECT Score
FROM (
   SELECT
      Score,
      DENSE_RANK() OVER (ORDER BY Score DESC) as rnk
   FROM
      Scores
) AS RankedScores
WHERE rnk = 4;
```

Explanation:

1. **Inner Query (`SELECT Score, DENSE_RANK() OVER (ORDER BY Score DESC) as rnk FROM Scores`)**:
   - `DENSE_RANK() OVER (ORDER BY Score DESC)`: This is a window function that assigns a rank to each score.
     - `ORDER BY Score DESC`: It orders the scores in descending order, so the highest score gets rank 1.
     - `DENSE_RANK()`: If there are ties (multiple students with the same score), they will receive the same rank, and the next distinct score will receive the next consecutive rank. For example, if the top two scores are 95, they both get rank 1, and the next score (e.g., 92) gets rank 2.
   - `as rnk`: This assigns an alias `rnk` to the calculated rank.
2. **Outer Query (`SELECT Score FROM ... WHERE rnk = 4`)**:
   - This query selects the `Score` from the results of the inner query.
   - `WHERE rnk = 4`: It filters these results to only include the row(s) where the calculated rank (`rnk`) is 4.

Result for the `Scores` table:

In the example `Scores` table, the scores in descending order with their `DENSE_RANK` would be:

- 95 (Rank 1)
- 95 (Rank 1)
- 92 (Rank 2)
- 91 (Rank 3)
- 90 (Rank 4)
- 88 (Rank 5)
- 88 (Rank 5)
- 85 (Rank 6)
- 78 (Rank 7)
- 78 (Rank 7)

Therefore, the query would return:

Score
90

If you wanted to find the 4th *distinct* highest value, you could modify the inner query to use `DISTINCT Score` within the `DENSE_RANK` or use `ROW_NUMBER()` after a `DISTINCT` selection. However, `DENSE_RANK()` as shown above is a common and robust way to handle "Nth highest" value scenarios, especially when ties are a possibility.

Gourav Rathi
rathi2k25@gmail.com