1) Create a Sales Table and Use Aggregate Functions a) Create a Sales table with columns: SaleID, ProductID, Quantity, SaleAmount, and SaleDate. b) Insert at least 10 sales records with different products and quantities. c) Write a query to calculate the total revenue generated using the SUM function. d) Find the product with the highest sale amount using the MAX function. e) Retrieve the average sale amount per transaction using the AVG function.

Sure! Here's how you can do each of the steps in SQL:

---

a) Create a Sales table

```
CREATE TABLE Sales (
    SaleID INT PRIMARY KEY,
    ProductID INT,
    Quantity INT,
    SaleAmount DECIMAL(10, 2),
    SaleDate DATE
);
```

---

b) Insert at least 10 sales records

```
INSERT INTO Sales (SaleID, ProductID, Quantity, SaleAmount, SaleDate) VALUES
(1, 101, 2, 40.00, '2025-04-01'),
(2, 102, 1, 25.00, '2025-04-01'),
(3, 103, 3, 75.00, '2025-04-02'),
(4, 101, 1, 20.00, '2025-04-02'),
(5, 104, 5, 125.00, '2025-04-03'),
(6, 105, 2, 60.00, '2025-04-03'),
(7, 102, 4, 100.00, '2025-04-04'),
(8, 106, 1, 30.00, '2025-04-04'),
(9, 103, 2, 50.00, '2025-04-05'),
(10, 104, 3, 75.00, '2025-04-05');
```

---

c) Query to calculate total revenue using SUM

```
SELECT SUM(SaleAmount) AS TotalRevenue FROM Sales;
```

---

d) Find the product with the highest sale amount using MAX

```
SELECT ProductID, SaleAmount
FROM Sales
WHERE SaleAmount = (SELECT MAX(SaleAmount) FROM Sales);
```

---

e) Retrieve the average sale amount per transaction using AVG

SELECT AVG(SaleAmount) AS AverageSaleAmount FROM Sales;

Let me know if you'd like to add product details in a separate table or visualize the data too!

2) Use DDL and DML Commands a) Create a Products table with columns for ProductID, ProductName, Price, and StockQuantity using DDL commands. b) Insert five product records and display all products using a SELECT query. c) Update the price of a product with ProductID = 3 and check the changes using a SELECT statement. d) Delete a product from the table and verify whether the changes are reflected. e) Alter the table to add a new column Discount and set a default value of 5%.

Sure! Here's a simple version of each part using **DDL (Data Definition Language)** and **DML (Data Manipulation Language)** commands:

---

a) Create Products Table (DDL)

```
CREATE TABLE Products (
  ProductID INT,
  ProductName VARCHAR(50),
  Price DECIMAL(10,2),
  StockQuantity INT
);
```

---

b) Insert 5 Records and Display All

```
-- Insert records
INSERT INTO Products VALUES
(1, 'Laptop', 800.00, 10),
(2, 'Mouse', 20.00, 50),
(3, 'Keyboard', 30.00, 40),
(4, 'Monitor', 150.00, 15),
(5, 'Printer', 120.00, 8);

-- Display all products
SELECT * FROM Products;
```

---

c) Update Price for ProductID = 3 and Check

```
-- Update
UPDATE Products SET Price = 35.00 WHERE ProductID = 3;

-- Check updated record
SELECT * FROM Products WHERE ProductID = 3;
```

---

d) Delete a Product and Verify

```
-- Delete product with ProductID = 2
DELETE FROM Products WHERE ProductID = 2;

-- Check remaining records
SELECT * FROM Products;
```

---

e) Alter Table to Add Discount Column

```
ALTER TABLE Products ADD Discount DECIMAL(5,2) DEFAULT 5.00;

-- Check structure and data
SELECT * FROM Products;
```

Let me know if you'd like to connect the Products table with the Sales table using foreign keys or do more advanced queries!

3) Create a Customer Table with Integrity Constraints a) Create a Customers table with constraints: CustomerID (PRIMARY KEY), Email (UNIQUE), Age (CHECK Age > 18). b) Insert a valid customer record and verify that the default country is assigned if not explicitly provided. c) Attempt to insert a customer with an age of 16 and observe the CHECK constraint violation. d) Try inserting two customers with the same email ID and observe the UNIQUE constraint violation. e) Retrieve all customers who are older than 25 and belong to a country other than 'India'.

a) Create Customers Table with Constraints

```
CREATE TABLE Customers (
  CustomerID INT PRIMARY KEY,
  Name VARCHAR(50),
  Email VARCHAR(100) UNIQUE,
  Age INT CHECK (Age > 18),
  Country VARCHAR(50) DEFAULT 'India'
);
```

b) Insert a Valid Customer (No Country Provided)

```
-- Insert valid record (Country will default to 'India')
INSERT INTO Customers (CustomerID, Name, Email, Age)
VALUES (1, 'John Doe', 'john@example.com', 30);

-- View inserted record
SELECT * FROM Customers;
```

c) Try Inserting a Customer with Age = 16 (Should Fail)

```
sql
CopyEdit
-- This will give an error due to CHECK constraint (Age must be > 18)
INSERT INTO Customers (CustomerID, Name, Email, Age)
VALUES (2, 'Teen User', 'teen@example.com', 16);
```

d) Try Inserting Two Customers with Same Email (Should Fail)

```
sql
CopyEdit
-- First one is valid
INSERT INTO Customers (CustomerID, Name, Email, Age, Country)
VALUES (3, 'Alice', 'alice@example.com', 28, 'USA');

-- This will fail because email must be UNIQUE
INSERT INTO Customers (CustomerID, Name, Email, Age, Country)
VALUES (4, 'Bob', 'alice@example.com', 35, 'UK');
```

e) Retrieve Customers Older Than 25 and Not from India

```
sql
CopyEdit
SELECT * FROM Customers
WHERE Age > 25 AND Country <> 'India';
```

4) Create a Table with Constraints a) Create an EmployeeDetails table with EmployeeID as the PRIMARY KEY and DepartmentID as a FOREIGN KEY referencing a Department table. b) Insert a valid employee record with an existing DepartmentID, then attempt to insert an employee with a non-existent DepartmentID and observe the constraint violation. c) Insert an employee with a duplicate EmployeeID and check how the primary key constraint prevents duplicate entries. d) Modify the Salary column to have a UNIQUE constraint and attempt to insert two employees with the same salary to test the constraint. e) Write a query to delete an employee from EmployeeDetails and ensure that the deletion does not violate any referential integrity constraints.

Absolutely! Here's a step-by-step SQL guide to create an EmployeeDetails table with constraints and test them.

---

a) Create Department and EmployeeDetails Tables with Constraints

```
-- Create Department table
CREATE TABLE Department (
  DepartmentID INT PRIMARY KEY,
  DepartmentName VARCHAR(50)
);

-- Create EmployeeDetails table with constraints
CREATE TABLE EmployeeDetails (
  EmployeeID INT PRIMARY KEY,
  Name VARCHAR(50),
  DepartmentID INT,
  Salary DECIMAL(10, 2),
  FOREIGN KEY (DepartmentID) REFERENCES Department(DepartmentID)
);
```

---

b) Insert Valid and Invalid Employee Records

```
-- Insert a department
INSERT INTO Department VALUES (1, 'HR'), (2, 'IT');

-- Insert valid employee (valid DepartmentID)
INSERT INTO EmployeeDetails VALUES (101, 'John', 1, 50000.00);

-- Try inserting an employee with a non-existent DepartmentID (should fail)
INSERT INTO EmployeeDetails VALUES (102, 'Jane', 5, 60000.00);
```

❌ The second insert will fail due to **foreign key constraint violation** (DepartmentID 5 does not exist).

---

c) Insert Duplicate EmployeeID (Should Fail)

```
-- Try inserting with duplicate EmployeeID
INSERT INTO EmployeeDetails VALUES (101, 'Mike', 2, 55000.00);
```

❌ This will fail due to **primary key constraint** on EmployeeID.

---

d) Add UNIQUE Constraint on Salary and Test It

```
-- Add UNIQUE constraint to Salary column
ALTER TABLE EmployeeDetails ADD CONSTRAINT unique_salary UNIQUE (Salary);

-- Insert employee with a new unique salary (will succeed)
INSERT INTO EmployeeDetails VALUES (103, 'Anna', 2, 70000.00);

-- Try inserting another employee with the same salary (should fail)
INSERT INTO EmployeeDetails VALUES (104, 'Ella', 1, 70000.00);
```

❌ The second insert with the same salary will fail due to the **UNIQUE constraint**.

---

e) Delete an Employee Safely

```
-- Delete an employee (only if no other tables reference EmployeeDetails via FK)
DELETE FROM EmployeeDetails WHERE EmployeeID = 103;
```

✅ This will succeed **as long as no other table is using EmployeeID as a foreign key**.

5) Create an Employee Table with Various Columns a) Create a table Employee with attributes: EmployeeID (INT, PRIMARY KEY), Name (VARCHAR), Salary (DECIMAL), JoiningDate (DATE), and ActiveStatus (BOOLEAN). b) Insert five sample employee records and ensure each employee has a unique EmployeeID. c) Write a query to find all employees who joined before January 1, 2023. d) Update the salary of an employee named ‘Amit Sharma’ by 10% and display the updated record. e) Retrieve all employees who are currently active (ActiveStatus = TRUE).
Here’s a simple step-by-step SQL guide to create and work with an Employee table as described:

---

a) Create Employee Table

```
CREATE TABLE Employee (
  EmployeeID INT PRIMARY KEY,
  Name VARCHAR(100),
  Salary DECIMAL(10, 2),
  JoiningDate DATE,
  ActiveStatus BOOLEAN
);
```

---

b) Insert 5 Sample Employee Records

```
INSERT INTO Employee VALUES
(1, 'Amit Sharma', 50000.00, '2022-06-15', TRUE),
(2, 'Priya Verma', 60000.00, '2023-03-10', TRUE),
(3, 'Rahul Mehta', 55000.00, '2021-11-25', FALSE),
(4, 'Sneha Kapoor', 70000.00, '2022-12-01', TRUE),
(5, 'Manoj Yadav', 65000.00, '2024-01-20', FALSE);
```

---

c) Query: Employees Who Joined Before January 1, 2023

```
SELECT * FROM Employee
WHERE JoiningDate < '2023-01-01';
```

---

d) Update Salary of ‘Amit Sharma’ by 10% and Show Updated Record

```
-- Update salary
UPDATE Employee
SET Salary = Salary * 1.10
WHERE Name = 'Amit Sharma';

-- Display updated record
SELECT * FROM Employee
WHERE Name = 'Amit Sharma';
```

---

e) Retrieve All Active Employees (ActiveStatus = TRUE)

```
SELECT * FROM Employee
WHERE ActiveStatus = TRUE;
```

Let me know if you'd like to sort, group by joining year, or add more fields like department or email!

Sure! Here's a complete and **simple SQL guide** to perform all the **aggregate functions** using the Sales table.
For this example, we assume the Sales table has these columns:
SaleID, ProductID, ProductName, Quantity, SaleAmount, Discount, SaleDate

---

✅ **a) Total Sales Amount in February 2025**

```
SELECT SUM(SaleAmount) AS TotalSalesFeb2025
FROM Sales
WHERE SaleDate >= '2025-02-01' AND SaleDate < '2025-03-01';
```

✅ **b) Average Billing Amount**

```
SELECT AVG(SaleAmount) AS AverageBillingAmount
```

FROM Sales;

✅ **c) Minimum Quantity Sold in Any Transaction**
SELECT MIN(Quantity) AS MinQuantitySold
FROM Sales;

✅ **d) Highest Discount Applied**
SELECT MAX(Discount) AS MaxDiscountApplied
FROM Sales;

✅ **e) Count of Transactions for Product "Laptop"**
SELECT COUNT(*) AS LaptopSalesCount
FROM Sales
WHERE ProductName = 'Laptop';

Let me know if you want to group results by product or month, or visualize sales trends!

7) Constraints (on a single table: Employees) a) Create the Employees table with EmployeeID as PRIMARY KEY, Email as UNIQUE, and Salary with a CHECK (Salary > 10000) constraint. b) Add a NOT NULL constraint on the Name column in the Employees table and try inserting a record without the name. c) Add a DEFAULT value 'Active' to the Status column in Employees, and insert a record without specifying the status to verify the default. d) Insert a record into Employees where Salary is less than 10000 to test the CHECK constraint. e) Try inserting two employees with the same Email ID to verify the enforcement of the UNIQUE constraint.

Here's a clear and simple SQL guide to work with **constraints on the Employees table**, step by step:

✅ **a) Create Employees Table with Constraints**
```
CREATE TABLE Employees (
  EmployeeID INT PRIMARY KEY,
  Name VARCHAR(100),
  Email VARCHAR(100) UNIQUE,
  Salary DECIMAL(10, 2) CHECK (Salary > 10000),
  Status VARCHAR(20) DEFAULT 'Active'
);
```

✅ **b) Add NOT NULL Constraint on Name and Try Inserting Without Name**
```
-- Alter the column to add NOT NULL
ALTER TABLE Employees
MODIFY Name VARCHAR(100) NOT NULL;

-- Try inserting without a name (will FAIL)
INSERT INTO Employees (EmployeeID, Email, Salary)
VALUES (1, 'test1@example.com', 15000.00);
```
❌ This will fail due to the NOT NULL constraint on Name.

✅ **c) Insert a Record Without Specifying Status (Should Use Default)**
```
INSERT INTO Employees (EmployeeID, Name, Email, Salary)
VALUES (2, 'Anita Roy', 'anita@example.com', 20000.00);

-- View inserted record
SELECT * FROM Employees WHERE EmployeeID = 2;
```
✅ Status will be automatically set to 'Active'.

✅ **d) Insert a Record with Salary Less Than 10000 (Should Fail)**
```
-- This will FAIL due to CHECK constraint on Salary
INSERT INTO Employees (EmployeeID, Name, Email, Salary)
```

VALUES (3, 'Ravi Kumar', 'ravi@example.com', 9000.00);
❌ This will fail due to CHECK (Salary > 10000).

✅ **e) Try Inserting Two Employees with the Same Email (Should Fail)**
-- First insert (OK)
INSERT INTO Employees (EmployeeID, Name, Email, Salary)
VALUES (4, 'Priya Mehta', 'priya@example.com', 25000.00);

-- Duplicate Email (will FAIL)
INSERT INTO Employees (EmployeeID, Name, Email, Salary)
VALUES (5, 'Karan Sinha', 'priya@example.com', 26000.00);
❌ The second insert will fail due to the UNIQUE constraint on Email.

---

8) DDL and DML Commands a) Use DDL commands to create a Library database and define a Books table with fields: BookID, Title, Author, Genre, and Price. b) Insert at least five sample records into the Books table using INSERT (DML) and verify them using a SELECT query. c) A new column PublicationYear needs to be added. Use ALTER TABLE to modify the existing table structure. d) Update the price of all books published before 2020 by increasing 10% using the UPDATE statement. e) Use DELETE to remove all books where the genre is 'Outdated Technology' and validate the change with a SELECT query.

Here's a simple and complete SQL guide for working with **DDL and DML commands** in a Library database scenario:

✅ **a) Create Library Database and Books Table (DDL)**
-- Create Library database (optional, depends on SQL system)
CREATE DATABASE Library;

-- Use the Library database (if required)
USE Library;

-- Create Books table
CREATE TABLE Books (
  BookID INT PRIMARY KEY,
  Title VARCHAR(100),
  Author VARCHAR(100),
  Genre VARCHAR(50),
  Price DECIMAL(10, 2)
);

✅ **b) Insert 5 Sample Book Records and Verify**
-- Insert sample books
INSERT INTO Books VALUES
(1, 'Digital Fortress', 'Dan Brown', 'Thriller', 350.00),
(2, 'Clean Code', 'Robert C. Martin', 'Programming', 550.00),
(3, 'Harry Potter', 'J.K. Rowling', 'Fantasy', 400.00),
(4, 'The Pragmatic Programmer', 'Andrew Hunt', 'Programming', 600.00),
(5, 'Old Tech Guide', 'John Doe', 'Outdated Technology', 200.00);

-- View all records
SELECT * FROM Books;

✅ **c) Add PublicationYear Column Using ALTER TABLE**
ALTER TABLE Books ADD PublicationYear INT;

✅ **d) Update Price by 10% for Books Published Before 2020**

```
-- First, update publication years for testing
UPDATE Books SET PublicationYear = 2005 WHERE BookID = 1;
UPDATE Books SET PublicationYear = 2009 WHERE BookID = 2;
UPDATE Books SET PublicationYear = 2021 WHERE BookID = 3;
UPDATE Books SET PublicationYear = 2015 WHERE BookID = 4;
UPDATE Books SET PublicationYear = 2010 WHERE BookID = 5;

-- Now increase price by 10% for books before 2020
UPDATE Books
SET Price = Price * 1.10
WHERE PublicationYear < 2020;

-- View updated prices
SELECT * FROM Books;
```

✅ **e) Delete Books with Genre = 'Outdated Technology' and Verify**

```
DELETE FROM Books
WHERE Genre = 'Outdated Technology';

-- Verify changes
SELECT * FROM Books;
```

Let me know if you'd like to add more advanced filtering, relationships with authors, or borrowing history!

9) DDL and DML Commands (on a single table: Books) a) Create a table Books using DDL with fields: BookID, Title, Author, Price, and StockAvailable. b) Insert 5 book records into the Books table using the INSERT command. c) Modify the structure of Books table by adding a new column Genre using the ALTER TABLE command. d) Use the UPDATE command to increase the price of all books by RS 50 in the Books table. e) Delete all records from the Books table where StockAvailable is 0 using the DELETE command

Here's a simple step-by-step SQL guide using **DDL and DML commands** on a single table Books:

✅ **a) Create Books Table (DDL)**

```
CREATE TABLE Books (
  BookID INT PRIMARY KEY,
  Title VARCHAR(100),
  Author VARCHAR(100),
  Price DECIMAL(10, 2),
  StockAvailable INT
);
```

✅ **b) Insert 5 Book Records (DML)**

```
INSERT INTO Books VALUES
(1, 'The Alchemist', 'Paulo Coelho', 300.00, 10),
(2, 'Wings of Fire', 'A.P.J. Abdul Kalam', 250.00, 5),
(3, 'Zero to One', 'Peter Thiel', 450.00, 0),
(4, 'Atomic Habits', 'James Clear', 500.00, 3),
(5, 'Think and Grow Rich', 'Napoleon Hill', 350.00, 0);
```

✅ **c) Add Genre Column (DDL - ALTER TABLE)**

```
ALTER TABLE Books ADD Genre VARCHAR(50);
```

✅ **d) Increase Price of All Books by ₹50 (DML - UPDATE)**

```
UPDATE Books
SET Price = Price + 50;
```

✅ **e) Delete Books Where StockAvailable = 0 (DML - DELETE)**

```
DELETE FROM Books
WHERE StockAvailable = 0;

-- Verify changes
SELECT * FROM Books;
```

---

10) Analyze Sales Performance Using Aggregate Functions a) Calculate the total quantity of products sold across all transactions in the Sales table. b) Find the average sale amount for transactions made in March 2025. c) Identify the product with the minimum sale quantity from the Sales table. d) Determine the maximum discount offered in February 2025. e) Count how many sales were made by each salesperson using GROUP BY SalesPerson.

Here's a straightforward SQL guide to **analyze sales performance using aggregate functions**, assuming your Sales table has the following columns:

SaleID, ProductID, ProductName, Quantity, SaleAmount, Discount, SaleDate, SalesPerson

---

✅ **a) Total Quantity of Products Sold**

```
SELECT SUM(Quantity) AS TotalQuantitySold
FROM Sales;
```

✅ **b) Average Sale Amount for March 2025**

```
SELECT AVG(SaleAmount) AS AvgSaleMarch2025
FROM Sales
WHERE SaleDate >= '2025-03-01' AND SaleDate < '2025-04-01';
```

✅ **c) Product with Minimum Sale Quantity**

```
SELECT ProductName, MIN(Quantity) AS MinQuantitySold
FROM Sales
GROUP BY ProductName
ORDER BY MinQuantitySold ASC
LIMIT 1;
```

Or if you want only **one row** with the absolute lowest quantity:

```
SELECT ProductName, Quantity
FROM Sales
```

```
ORDER BY Quantity ASC
LIMIT 1;
```

✅ d) Maximum Discount Offered in February 2025

```
SELECT MAX(Discount) AS MaxDiscountFeb2025
FROM Sales
WHERE SaleDate >= '2025-02-01' AND SaleDate < '2025-03-01';
```

✅ e) Number of Sales Made by Each SalesPerson

```
SELECT SalesPerson, COUNT(*) AS TotalSales
FROM Sales
GROUP BY SalesPerson;
```
