Let's start with a table named `Orders`, then create **10 SQL Server views** where each is posed as a **question** (use case) with its **SQL answer**.

✅ Base Table: `Orders`

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY IDENTITY(1,1),
    CustomerName NVARCHAR(100),
    ProductName NVARCHAR(100),
    Quantity INT,
    Price DECIMAL(10, 2),
    OrderDate DATE
);
```

🚀 Sample Insert

```
INSERT INTO Orders (CustomerName, ProductName, Quantity, Price, OrderDate)
VALUES
('Alice', 'Laptop', 1, 1200.00, '2023-10-01'),
('Bob', 'Phone', 2, 800.00, '2023-11-15'),
('Charlie', 'Monitor', 1, 300.00, '2023-11-20'),
('Alice', 'Mouse', 3, 25.00, '2023-12-01'),
('Diana', 'Keyboard', 2, 45.00, '2024-01-05'),
('Eve', 'Laptop', 1, 1300.00, '2024-01-10'),
('Bob', 'Headphones', 2, 150.00, '2024-02-01'),
('Alice', 'Webcam', 1, 75.00, '2024-02-15'),
('Charlie', 'Mouse', 2, 25.00, '2024-03-01'),
('Diana', 'Monitor', 1, 310.00, '2024-03-15');
```

🔟 Views: Questions with SQL Answers

1. What are the total sales per customer?

```
CREATE VIEW vw_TotalSalesPerCustomer AS
SELECT
    CustomerName,
    SUM(Quantity * Price) AS TotalSpent
FROM Orders
GROUP BY CustomerName;
```

2. What are the orders made in 2024?

```
CREATE VIEW vw_OrdersIn2024 AS
SELECT *
FROM Orders
WHERE YEAR(OrderDate) = 2024;
```

3. Which products were ordered more than once (across all customers)?

```
CREATE VIEW vw_MultiOrderedProducts AS
```

```sql
SELECT ProductName, COUNT(*) AS OrderCount
FROM Orders
GROUP BY ProductName
HAVING COUNT(*) > 1;
```

4. What is the average order value per product?

```sql
CREATE VIEW vw_AvgOrderValuePerProduct AS
SELECT
    ProductName,
    AVG(Quantity * Price) AS AvgOrderValue
FROM Orders
GROUP BY ProductName;
```

5. Who are the top 3 customers by total spend?

```sql
CREATE VIEW vw_Top3CustomersBySpend AS
SELECT TOP 3
    CustomerName,
    SUM(Quantity * Price) AS TotalSpent
FROM Orders
GROUP BY CustomerName
ORDER BY TotalSpent DESC;
```

6. What were the orders placed in the last 30 days?

```
CREATE VIEW vw_RecentOrders AS
SELECT *
FROM Orders
WHERE OrderDate >= DATEADD(DAY, -30, GETDATE());
```

7. What are the total quantities ordered per product?

```
CREATE VIEW vw_TotalQuantityPerProduct AS
SELECT
    ProductName,
    SUM(Quantity) AS TotalQuantity
FROM Orders
GROUP BY ProductName;
```

8. Which customers have ordered more than 3 products in total (cumulative quantity)?

```
CREATE VIEW vw_HeavyBuyers AS
SELECT
    CustomerName,
    SUM(Quantity) AS TotalQuantity
FROM Orders
GROUP BY CustomerName
```

```
HAVING SUM(Quantity) > 3;
```

---

9. What is the monthly sales summary?

```
CREATE VIEW vw_MonthlySales AS
SELECT
    FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
    SUM(Quantity * Price) AS TotalSales
FROM Orders
GROUP BY FORMAT(OrderDate, 'yyyy-MM');
```

---

10. What is the total revenue per product per month?

```
CREATE VIEW vw_ProductRevenueByMonth AS
SELECT
    ProductName,
    FORMAT(OrderDate, 'yyyy-MM') AS OrderMonth,
    SUM(Quantity * Price) AS Revenue
FROM Orders
GROUP BY ProductName, FORMAT(OrderDate, 'yyyy-MM');
```

Would you like to materialize any of these into indexed views for performance optimization?

Follow GPT creator profile on LinkedIn https://linkedin.com/in/khanukov/

Here are **5 practical stored procedure examples** in SQL Server based on the `Orders` table. Each one demonstrates a different use case:

✅ 1. Add a New Order

```
CREATE PROCEDURE usp_AddOrder
    @CustomerName NVARCHAR(100),
    @ProductName NVARCHAR(100),
    @Quantity INT,
    @Price DECIMAL(10, 2),
    @OrderDate DATE
AS
BEGIN
    INSERT INTO Orders (CustomerName, ProductName, Quantity, Price, OrderDate)
    VALUES (@CustomerName, @ProductName, @Quantity, @Price, @OrderDate);
END;
```

✅ 2. Get Orders by Customer

```
CREATE PROCEDURE usp_GetOrdersByCustomer
    @CustomerName NVARCHAR(100)
AS
BEGIN
    SELECT *
    FROM Orders
    WHERE CustomerName = @CustomerName;
END;
```

✅ 3. Get Total Revenue Between Dates

```
CREATE PROCEDURE usp_GetRevenueBetweenDates
    @StartDate DATE,
    @EndDate DATE
AS
BEGIN
    SELECT
        SUM(Quantity * Price) AS TotalRevenue
    FROM Orders
    WHERE OrderDate BETWEEN @StartDate AND @EndDate;
END;
```

✅ 4. Update Order Quantity

```
CREATE PROCEDURE usp_UpdateOrderQuantity
    @OrderID INT,
    @NewQuantity INT
AS
BEGIN
    UPDATE Orders
    SET Quantity = @NewQuantity
    WHERE OrderID = @OrderID;
END;
```

✅ 5. Delete Order by ID

```
CREATE PROCEDURE usp_DeleteOrder
    @OrderID INT
AS
BEGIN
    DELETE FROM Orders
    WHERE OrderID = @OrderID;
END;
```

You can call any of these using the `EXEC` statement. For example:

```
EXEC usp_AddOrder
    @CustomerName = 'Frank',
    @ProductName = 'Tablet',
    @Quantity = 2,
    @Price = 450.00,
    @OrderDate = '2024-05-01';
```

Would you like examples with **output parameters** or **transactions with rollback** logic?

Follow GPT creator profile on LinkedIn https://linkedin.com/in/khanukov/

—--------------

Here’s an example using **3 related tables** with **foreign keys**, followed by **5 views** that demonstrate **JOINs** across these tables in SQL Server.

🗃️ Tables

1. `Customers`

```
CREATE TABLE Customers (

    CustomerID INT PRIMARY KEY IDENTITY(1,1),

    CustomerName NVARCHAR(100),

    Email NVARCHAR(100)

);
```

2. `Products`

```
CREATE TABLE Products (

    ProductID INT PRIMARY KEY IDENTITY(1,1),

    ProductName NVARCHAR(100),

    UnitPrice DECIMAL(10, 2)

);
```

3. `Orders`

```
CREATE TABLE Orders (

    OrderID INT PRIMARY KEY IDENTITY(1,1),

    CustomerID INT FOREIGN KEY REFERENCES Customers(CustomerID),

    ProductID INT FOREIGN KEY REFERENCES Products(ProductID),
    Quantity INT,
    OrderDate DATE
);
```

✅ Sample Inserts

```
-- Customers
INSERT INTO Customers (CustomerName, Email)
VALUES ('Alice', 'alice@example.com'),
       ('Bob', 'bob@example.com');

-- Products
INSERT INTO Products (ProductName, UnitPrice)
VALUES ('Laptop', 1200), ('Mouse', 25), ('Monitor', 300);

-- Orders
INSERT INTO Orders (CustomerID, ProductID, Quantity, OrderDate)
VALUES
(1, 1, 1, '2024-04-01'),
(1, 2, 3, '2024-04-05'),
(2, 3, 2, '2024-04-10');
```

👁️‍🗨️ Views with JOINs

1. Full Order Details (Customer + Product Info)

Which customers ordered which products and how much did it cost?

```
CREATE VIEW vw_FullOrderDetails AS
SELECT
   o.OrderID,
   c.CustomerName,
   p.ProductName,
   o.Quantity,
   p.UnitPrice,
   (o.Quantity * p.UnitPrice) AS TotalPrice,
   o.OrderDate
FROM Orders o
JOIN Customers c ON o.CustomerID = c.CustomerID
JOIN Products p ON o.ProductID = p.ProductID;
```

2. Customer Order Summary

How many orders and total spend per customer?

```
CREATE VIEW vw_CustomerOrderSummary AS
SELECT
   c.CustomerName,
   COUNT(o.OrderID) AS TotalOrders,
   SUM(o.Quantity * p.UnitPrice) AS TotalSpent
FROM Customers c
```

```
JOIN Orders o ON c.CustomerID = o.CustomerID
JOIN Products p ON o.ProductID = p.ProductID
GROUP BY c.CustomerName;
```

---

3. Product Sales Summary

How many units of each product were sold?

```
CREATE VIEW vw_ProductSalesSummary AS
SELECT
   p.ProductName,
   SUM(o.Quantity) AS TotalUnitsSold,
   SUM(o.Quantity * p.UnitPrice) AS TotalRevenue
FROM Products p
JOIN Orders o ON p.ProductID = o.ProductID
GROUP BY p.ProductName;
```

---

4. Orders by Date with Customer and Product Info

What are the full details of all orders by date?

```
CREATE VIEW vw_OrdersByDate AS
SELECT
   o.OrderDate,
   c.CustomerName,
   p.ProductName,
   o.Quantity,
    (o.Quantity * p.UnitPrice) AS OrderValue
FROM Orders o
JOIN Customers c ON o.CustomerID = c.CustomerID
JOIN Products p ON o.ProductID = p.ProductID
ORDER BY o.OrderDate;
```

5. High Value Orders (> $500)

Which orders exceed $500 in value?

```
CREATE VIEW vw_HighValueOrders AS
SELECT
    o.OrderID,
    c.CustomerName,
    p.ProductName,
    o.Quantity,
    (o.Quantity * p.UnitPrice) AS OrderTotal
FROM Orders o
JOIN Customers c ON o.CustomerID = c.CustomerID
JOIN Products p ON o.ProductID = p.ProductID
WHERE (o.Quantity * p.UnitPrice) > 500;
```

Would you like to see **aggregated views with window functions** or **filtered views for reports (e.g. monthly revenue)**?

Follow GPT creator profile on LinkedIn https://linkedin.com/in/khanukov/

Here are **5 stored procedures** that interact with the `Customers`, `Products`, and `Orders` tables using **joins, inserts, and aggregations**.

✅ 1. Add New Order

```
CREATE PROCEDURE usp_AddOrder
    @CustomerID INT,
    @ProductID INT,
    @Quantity INT,
    @OrderDate DATE
AS
BEGIN
    INSERT INTO Orders (CustomerID, ProductID, Quantity, OrderDate)
    VALUES (@CustomerID, @ProductID, @Quantity, @OrderDate);
END;
```

✅ 2. Get Full Order Details for a Specific Customer

```
CREATE PROCEDURE usp_GetCustomerOrderDetails
    @CustomerID INT
AS
BEGIN
    SELECT
        o.OrderID,
        c.CustomerName,
        p.ProductName,
        o.Quantity,
        p.UnitPrice,
        (o.Quantity * p.UnitPrice) AS TotalPrice,
        o.OrderDate
    FROM Orders o
    JOIN Customers c ON o.CustomerID = c.CustomerID
    JOIN Products p ON o.ProductID = p.ProductID
    WHERE o.CustomerID = @CustomerID;
END;
```

✅ 3. Get Total Sales for a Product

```
CREATE PROCEDURE usp_GetProductSales
   @ProductID INT
AS
BEGIN
   SELECT
      p.ProductName,
      SUM(o.Quantity) AS TotalUnitsSold,
      SUM(o.Quantity * p.UnitPrice) AS TotalRevenue
   FROM Orders o
   JOIN Products p ON o.ProductID = p.ProductID
   WHERE p.ProductID = @ProductID
   GROUP BY p.ProductName;
END;
```

✅ 4. Update Customer Email

```
CREATE PROCEDURE usp_UpdateCustomerEmail
   @CustomerID INT,
   @NewEmail NVARCHAR(100)
AS
BEGIN
   UPDATE Customers
   SET Email = @NewEmail
   WHERE CustomerID = @CustomerID;
END;
```

✅ 5. Delete All Orders for a Customer

```
CREATE PROCEDURE usp_DeleteOrdersByCustomer
   @CustomerID INT
AS
BEGIN
   DELETE FROM Orders
   WHERE CustomerID = @CustomerID;
END;
```

You can execute any procedure like this:

```
EXEC usp_GetCustomerOrderDetails @CustomerID = 1;
```

Would you like examples that include **transactions**, **error handling**, or **output parameters**?

Follow GPT creator profile on LinkedIn https://linkedin.com/in/khanukov/

Yes, in **SQL Server**, you can use the `ALTER` statement to **modify both stored procedures and views** without having to drop and recreate them. Here's how it works:

✅ `ALTER PROCEDURE`

Used to **modify** an existing stored procedure:

```
ALTER PROCEDURE usp_AddOrder
    @CustomerID INT,
    @ProductID INT,
    @Quantity INT,
    @OrderDate DATE
AS
BEGIN
    -- Modified logic here
    PRINT 'Inserting new order...';
    INSERT INTO Orders (CustomerID, ProductID, Quantity, OrderDate)
    VALUES (@CustomerID, @ProductID, @Quantity, @OrderDate);
END;
```

✅ `ALTER VIEW`

Used to **update** an existing view definition:

```
ALTER VIEW vw_CustomerOrderSummary AS
SELECT
    c.CustomerName,
    COUNT(o.OrderID) AS TotalOrders,
    SUM(o.Quantity * p.UnitPrice) AS TotalSpent
FROM Customers c
JOIN Orders o ON c.CustomerID = o.CustomerID
JOIN Products p ON o.ProductID = p.ProductID
GROUP BY c.CustomerName;
```

🔄 Why Use `ALTER`?

- Preserves **permissions** and **dependencies**.
- Avoids issues from dropping and recreating objects (especially in production environments).
- Clean way to version control SQL logic changes.

Would you like a script to check if a view or procedure exists before altering or creating it?

Follow GPT creator profile on LinkedIn https://linkedin.com/in/khanukov/

In SQL Server, a stored procedure is a set of SQL statements that can be executed on demand. Here’s an example of a stored procedure with input parameters and output results.

Example 1: Simple Stored Procedure with Input and Output

This example demonstrates a stored procedure that takes an input parameter (`@EmployeeID`) and returns details about that employee.

```
CREATE PROCEDURE GetEmployeeDetails
    @EmployeeID INT
AS
BEGIN
    SELECT FirstName, LastName, Department, Salary
    FROM Employees
    WHERE EmployeeID = @EmployeeID;
END;
```

Explanation:

- **Input Parameter:** `@EmployeeID` (an integer representing the employee's ID).
- **Output:** The procedure returns the employee's first name, last name, department, and salary.

To Execute the Procedure:

```
EXEC GetEmployeeDetails @EmployeeID = 101;
```

This will output the details of the employee with `EmployeeID = 101`.

Example 2: Stored Procedure with Output Parameters

This example calculates and returns the total salary of all employees in a specific department. It uses an output parameter.

```
CREATE PROCEDURE GetDepartmentSalaryTotal
   @DepartmentName VARCHAR(50),
   @TotalSalary DECIMAL(18,2) OUTPUT
AS
BEGIN
   SELECT @TotalSalary = SUM(Salary)
   FROM Employees
   WHERE Department = @DepartmentName;
END;
```

Explanation:

- **Input Parameter:** `@DepartmentName` (a string representing the department name).
- **Output Parameter:** `@TotalSalary` (a decimal number that will hold the sum of all salaries in the department).

To Execute the Procedure:

```
DECLARE @Total DECIMAL(18,2);

EXEC GetDepartmentSalaryTotal @DepartmentName = 'Sales', @TotalSalary = @Total OUTPUT;

SELECT @Total AS TotalSalary;
```

This will calculate the total salary for all employees in the 'Sales' department and output the result.

Example 3: Stored Procedure Returning a Table (Table-Valued Output)

In this example, the procedure returns a result set (table) of employees working in a specific department.

```
CREATE PROCEDURE GetEmployeesByDepartment
   @DepartmentName VARCHAR(50)
AS
BEGIN
   SELECT EmployeeID, FirstName, LastName, Salary
   FROM Employees
   WHERE Department = @DepartmentName;
END;
```

To Execute the Procedure:

```
EXEC GetEmployeesByDepartment @DepartmentName = 'Marketing';
```

This will return a table of employees in the 'Marketing' department, including their employee ID, first name, last name, and salary.

These are simple examples to demonstrate the concept of stored procedures with input and output in SQL Server. You can modify these to meet more specific needs, such as handling more complex logic or using different data types.

Sure! Here's a complete example of a **stored procedure with an output parameter** in **Microsoft SQL Server**, along with how to create and execute it.

✅ Stored Procedure with Output Parameter

🔹 Example: Return the full name of an employee given their ID

Step 1: Create the table (if needed for testing)

```
CREATE TABLE Employees (
   EmployeeID INT PRIMARY KEY,
   FirstName NVARCHAR(50),
    LastName NVARCHAR(50)
);

-- Insert sample data
INSERT INTO Employees (EmployeeID, FirstName, LastName)
VALUES (1, 'John', 'Doe'), (2, 'Jane', 'Smith');
```

Step 2: Create the procedure with an input and an output parameter

```
CREATE PROCEDURE GetEmployeeFullName
    @EmpID INT,                       -- Input
    @FullName NVARCHAR(101) OUTPUT    -- Output
AS
BEGIN
    SELECT @FullName = FirstName + ' ' + LastName
    FROM Employees
    WHERE EmployeeID = @EmpID;
END;
```

Step 3: Execute the procedure and get the output

```
DECLARE @Name NVARCHAR(101);  -- Variable to receive the output

EXEC GetEmployeeFullName
     @EmpID = 1,
     @FullName = @Name OUTPUT;

-- Display the output
SELECT @Name AS EmployeeFullName;
```

📝 Output:

For `@EmpID = 1`, the result will be:

```
EmployeeFullName
-----------------
John Doe
```

Would you like an example that **returns multiple values** or **a table result along with output** as well?

Here are **three more SQL Server stored procedure examples** using **output parameters**, showing different use cases. Each one includes how to execute it.

🔹 Example 1: Calculate Bonus Based on Salary

Procedure: `CalculateBonus`

```
CREATE PROCEDURE CalculateBonus
    @EmployeeID INT,
    @Bonus DECIMAL(10, 2) OUTPUT
AS
BEGIN
    DECLARE @Salary DECIMAL(10, 2);

    SELECT @Salary = Salary FROM Employees WHERE EmployeeID = @EmployeeID;

    -- Bonus is 10% of Salary
    SET @Bonus = @Salary * 0.10;
END;
```

Execute:

```
DECLARE @MyBonus DECIMAL(10,2);

EXEC CalculateBonus
    @EmployeeID = 2,
    @Bonus = @MyBonus OUTPUT;

SELECT @MyBonus AS BonusAmount;
```

🔹 Example 2: Get Count of Employees in a Department

Procedure: `CountEmployeesByDepartment`

```
CREATE PROCEDURE CountEmployeesByDepartment
    @DeptName NVARCHAR(50),
    @EmpCount INT OUTPUT
AS
BEGIN
    SELECT @EmpCount = COUNT(*)
    FROM Employees
    WHERE Department = @DeptName;
END;
```

Execute:

```
DECLARE @Count INT;

EXEC CountEmployeesByDepartment
    @DeptName = 'HR',
    @EmpCount = @Count OUTPUT;

SELECT @Count AS TotalEmployees;
```

🔹 Example 3: Return Highest Paid Employee in a Department

Procedure: `GetTopEarner`

```
CREATE PROCEDURE GetTopEarner
    @DeptName NVARCHAR(50),
    @TopEarnerName NVARCHAR(101) OUTPUT,
    @TopSalary DECIMAL(10, 2) OUTPUT
AS
BEGIN
    SELECT TOP 1
        @TopEarnerName = FirstName + ' ' + LastName,
        @TopSalary = Salary
    FROM Employees
    WHERE Department = @DeptName
    ORDER BY Salary DESC;
END;
```

Execute:

```
DECLARE @Name NVARCHAR(101), @Salary DECIMAL(10,2);
```

```
EXEC GetTopEarner
    @DeptName = 'Sales',
    @TopEarnerName = @Name OUTPUT,
    @TopSalary = @Salary OUTPUT;

SELECT @Name AS TopEarner, @Salary AS Salary;
```

These examples demonstrate common tasks like **computations**, **aggregates**, and **retrieving top values** using **output parameters** in SQL Server stored procedures.

Would you like an example involving **input/output together with a table-valued return** or **error handling** next?

Yes, in **SQL Server**, you can use the `ALTER` statement to **modify both stored procedures and views** without having to drop and recreate them. Here's how it works:

✅ `ALTER PROCEDURE`

Used to **modify** an existing stored procedure:

```
ALTER PROCEDURE usp_AddOrder
    @CustomerID INT,
    @ProductID INT,
    @Quantity INT,
    @OrderDate DATE
AS
BEGIN
    -- Modified logic here
    PRINT 'Inserting new order...';
    INSERT INTO Orders (CustomerID, ProductID, Quantity, OrderDate)
    VALUES (@CustomerID, @ProductID, @Quantity, @OrderDate);
END;
```

✅ `ALTER VIEW`

Used to **update** an existing view definition:

```
ALTER VIEW vw_CustomerOrderSummary AS
```

```
SELECT
   c.CustomerName,
   COUNT(o.OrderID) AS TotalOrders,
   SUM(o.Quantity * p.UnitPrice) AS TotalSpent
FROM Customers c
JOIN Orders o ON c.CustomerID = o.CustomerID
JOIN Products p ON o.ProductID = p.ProductID
GROUP BY c.CustomerName;
```

🔄 Why Use `ALTER`?

- Preserves **permissions** and **dependencies**.
- Avoids issues from dropping and recreating objects (especially in production environments).
- Clean way to version control SQL logic changes.

Would you like a script to check if a view or procedure exists before altering or creating it?

Follow GPT creator profile on LinkedIn https://linkedin.com/in/khanukov/

Great question!

In **SQL Server**, the `UPDATE` statement is used to **modify data (rows)** in a table—not the **structure** of objects like views or procedures. Here's how it compares and what each is used for:

✅ `UPDATE` – for data rows

Used to change existing data in **tables**.

```
-- Update quantity for a specific order
UPDATE Orders
SET Quantity = 5
WHERE OrderID = 1;
```

✅ You can also update via joins:

```
-- Increase unit price for 'Mouse' in the Products table
UPDATE p
SET p.UnitPrice = p.UnitPrice + 5
FROM Products p
JOIN Orders o ON p.ProductID = o.ProductID
WHERE p.ProductName = 'Mouse';
```

❌ `UPDATE` Cannot Be Used To:

- Modify stored procedure code → use `ALTER PROCEDURE`
- Modify view definitions → use `ALTER VIEW`
- Change table structure (columns, types) → use `ALTER TABLE`

Quick Summary:

Command	Purpose
`UPDATE`	Modify **data** in tables
`ALTER PROCEDURE`	Modify **stored procedure** definition
`ALTER VIEW`	Modify **view** definition
`ALTER TABLE`	Modify **table structure**

Want an example where you `UPDATE` data using a **view with `INSTEAD OF` trigger**?

Follow GPT creator profile on LinkedIn https://linkedin.com/in/khanukov/

The key difference between `ALTER` and `UPDATE` in SQL Server lies in **what they change**:

🔧 `ALTER` — Changes the Structure (Schema)

- Used to **modify database objects** like:
    - Tables (`ALTER TABLE`)
    - Views (`ALTER VIEW`)
    - Stored Procedures (`ALTER PROCEDURE`)
    - Functions, Triggers, Schemas, etc.
- **Does not affect data**, only the **structure** or logic.

Example:

```
ALTER TABLE Orders
ADD OrderStatus NVARCHAR(50);  -- Adds a new column
```

✏️ `UPDATE` — Changes the Data (Rows)

- Used to **modify existing data** in tables.
- Works on the **values inside rows**, not on table or object definitions.

Example:

```
UPDATE Orders
SET Quantity = 10
WHERE OrderID = 3;
```

🧠 Analogy

Think of a **table as a spreadsheet**:

Action	What it Changes	Analogy
`ALTER`	The structure (columns, logic)	Adding/removing columns

`UPDATE`	The data inside cells	Editing cell values

Would you like a diagram or cheat sheet summarizing all DDL vs. DML differences?

Follow GPT creator profile on LinkedIn https://linkedin.com/in/khanukov/