# ****Advanced SQL Exercises for Online Retail Store****

## ****Exercise 1: Ranking and Window Functions****

**Goal:** Use ROW\_NUMBER(), RANK(), DENSE\_RANK(), OVER(), and PARTITION BY.

**Scenario:**  
Find the top 3 most expensive products in each category using different ranking functions.

**Steps:** Use ROW\_NUMBER() to assign a unique rank within each category.

Use RANK() and DENSE\_RANK() to compare how ties are handled.

Use PARTItION BY Category and ORDER BY Price DESC.

**Sample Query:**

SELECT

ProductID,

ProductName,

Category,

Price,

ROW\_NUMBER() OVER (PARTITION BY Category ORDER BY Price DESC) AS RowNum,

RANK() OVER (PARTITION BY Category ORDER BY Price DESC) AS RankNum,

DENSE\_RANK() OVER (PARTITION BY Category ORDER BY Price DESC) AS DenseRankNum

FROM Products;

## ****Exercise 2: Aggregation with GROUPING SETS, CUBE, and ROLLUP****

**Goal:** Analyze sales data across multiple dimensions.

**Scenario:**  
Generate a report showing total quantity sold by Region and Category.

**Steps:**

Join Orders, OrderDetails, Customers, and Products.

Use GROUPING SETS to get totals by Region, Category, and both.

Use ROLLUP to get subtotals and grand totals.

use CUBE to get all combinations of Region and Category.

**Sample Query (GROUPING SETS):**

SELECT

c.Region,

p.Category,

SUM(od.Quantity) AS TotalQuantity

FROM Orders o

JOIN OrderDetails od ON o.OrderID = od.OrderID

JOIN Customers c ON o.CustomerID = c.CustomerID

JOIN Products p ON od.ProductID = p.ProductID

GROUP BY GROUPING SETS ((c.Region), (p.Category), (c.Region, p.Category));

## ****Exercise 3: CTEs and MERGE****

**Goal:** Use WITH, CTEs, Recursive CTEs, and MERGE.

**Scenario:**  
a) Create a recursive CTE to generate a calendar table.  
b) Use a MERGE statement to update or insert product prices from a staging table.

**Steps:**

Create a recursive CTE to generate dates from '2025-01-01' to '2025-01-31'.

Create a StagingProducts table with updated prices.

Use MERGE to update existing products or insert new ones.

**Sample Query (Recursive CTE):**

WITH Calendar AS (

SELECT CAST('2025-01-01' AS DATE) AS DateValue

UNION ALL

SELECT DATEADD(DAY, 1, DateValue)

FROM Calendar

WHERE DateValue < '2025-01-31'

)

SELECT \* FROM Calendar;

**Sample Query (MERGE):**

MERGE Products AS target

USING StagingProducts AS source

ON target.ProductID = source.ProductID

WHEN MATCHED THEN

UPDATE SET target.Price = source.Price

WHEN NOT MATCHED THEN

INSERT (ProductID, ProductName, Category, Price)

VALUES (source.ProductID, source.ProductName, source.Category, source.Price);

## ****Exercise 4: PIVOT and UNPIVOT****

**Goal:** Transform data for reporting.

**Scenario:**  
Show monthly sales quantity per product in a pivoted format, and then unpivot it back.

**Steps:**

Aggregate sales by Product and Month.

Use PIVOT to convert rows into columns (one column per month).

Use UNPIVOT to convert the pivoted data back into row format.

**Sample Query (PIVOT):**

SELECT \*

FROM (

SELECT ProductID, MONTH(OrderDate) AS OrderMonth, Quantity

FROM Orders o

JOIN OrderDetails od ON o.OrderID = od.OrderID

) AS SourceTable

PIVOT (

SUM(Quantity) FOR OrderMonth IN ([1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12])

) AS PivotTable;

**Sample Query (UNPIVOT):**

SELECT ProductID, OrderMonth, Quantity

FROM PivotTable

UNPIVOT (

Quantity FOR OrderMonth IN ([1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12])

) AS UnpivotTable;

## ****Exercise 5: Using CTE to Simplify a Query****

**Goal:** Use Common Table Expressions (CTEs) to simplify complex queries.

**Scenario:**  
Find all customers who have placed more than 3 orders in total.

**Steps:**

Create a CTE that counts the number of orders placed by each customer.

Select only those customers who have placed more than 3 orders.

**Sample Query:**

WITH CustomerOrderCounts AS (

SELECT

o.CustomerID,

COUNT(o.OrderID) AS OrderCount

FROM Orders o

GROUP BY o.CustomerID

)

SELECT

c.CustomerID,

c.Name,

coc.OrderCount

FROM CustomerOrderCounts coc

JOIN Customers c ON c.CustomerID = coc.CustomerID

WHERE coc.OrderCount > 3;

# 5.Employee Management System - SQL Exercises with Answers

## Exercise 1: Create a Scalar Function

**Goal: Create a scalar function to calculate the annual salary of an employee.**

**Answer:**

CREATE FUNCTION fn\_CalculateAnnualSalary (@MonthlySalary DECIMAL(10,2))  
RETURNS DECIMAL(10,2)  
AS  
BEGIN  
 RETURN @MonthlySalary \* 12  
END;

**Test:**

SELECT EmployeeID, FirstName, LastName, dbo.fn\_CalculateAnnualSalary(Salary) AS AnnualSalary FROM Employees;

## Exercise 2: Create a Table-Valued Function

**Goal: Create a table-valued function to return employees in a specific department.**

**Answer:**

CREATE FUNCTION fn\_GetEmployeesByDepartment (@DeptID INT)  
RETURNS TABLE  
AS  
RETURN  
 SELECT \* FROM Employees WHERE DepartmentID = @DeptID;

**Test:**

SELECT \* FROM dbo.fn\_GetEmployeesByDepartment(2);

## Exercise 3: Create a User-Defined Function

**Goal: Create a user-defined function to calculate the bonus for an employee.**

**Answer:**

CREATE FUNCTION fn\_CalculateBonus (@Salary DECIMAL(10,2))  
RETURNS DECIMAL(10,2)  
AS  
BEGIN  
 RETURN @Salary \* 0.10  
END;

**Test:**

SELECT EmployeeID, FirstName, LastName, dbo.fn\_CalculateBonus(Salary) AS Bonus FROM Employees;

## Exercise 4: Modify a User-Defined Function

**Goal: Modify the `fn\_CalculateBonus` function to return `Salary \* 0.15`.**

**Answer:**

ALTER FUNCTION fn\_CalculateBonus (@Salary DECIMAL(10,2))  
RETURNS DECIMAL(10,2)  
AS  
BEGIN  
 RETURN @Salary \* 0.15  
END;

**Test:**

SELECT EmployeeID, FirstName, LastName, dbo.fn\_CalculateBonus(Salary) AS Bonus FROM Employees;

## Exercise 5: Delete a User-Defined Function

**Goal: Delete the `fn\_CalculateBonus` function.**

**Answer:**

DROP FUNCTION fn\_CalculateBonus;

**Verification:**

**-- Try calling the function again to confirm deletion (should throw an error)**

## Exercise 6: Execute a User-Defined Function

**Goal: Execute the `fn\_CalculateAnnualSalary` function.**

**Test:**

SELECT EmployeeID, FirstName, LastName, dbo.fn\_CalculateAnnualSalary(Salary) AS AnnualSalary FROM Employees;

## Exercise 7: Return Data from a Scalar Function

**Goal: Return the annual salary for a specific employee using `fn\_CalculateAnnualSalary`.**

**Test:**

SELECT dbo.fn\_CalculateAnnualSalary(Salary) AS AnnualSalary FROM Employees WHERE EmployeeID = 1;

## Exercise 8: Return Data from a Table-Valued Function

**Goal: Return employees from the Finance department using `fn\_GetEmployeesByDepartment`.**

**Test:**

SELECT \* FROM dbo.fn\_GetEmployeesByDepartment(3);

## Exercise 9: Create a Nested User-Defined Function

**Goal: Create a nested user-defined function to calculate the total compensation for an employee.**

**Answer:**

CREATE FUNCTION fn\_CalculateTotalCompensation (@Salary DECIMAL(10,2))  
RETURNS DECIMAL(10,2)  
AS  
BEGIN  
 RETURN dbo.fn\_CalculateAnnualSalary(@Salary) + dbo.fn\_CalculateBonus(@Salary)  
END;

**Test:**

SELECT EmployeeID, FirstName, LastName, dbo.fn\_CalculateTotalCompensation(Salary) AS TotalCompensation FROM Employees;

## Exercise 10: Modify a Nested User-Defined Function

**Goal: Modify the `fn\_CalculateTotalCompensation` function to include a new bonus calculation.**

**Answer:**

ALTER FUNCTION fn\_CalculateTotalCompensation (@Salary DECIMAL(10,2))  
RETURNS DECIMAL(10,2)  
AS  
BEGIN  
 RETURN dbo.fn\_CalculateAnnualSalary(@Salary) + dbo.fn\_CalculateBonus(@Salary)  
END;

**Test:**

SELECT EmployeeID, FirstName, LastName, dbo.fn\_CalculateTotalCompensation(Salary) AS TotalCompensation FROM Employees;