Exercise 1:Ranking and Window Functions

-- Exercise 1: Ranking and Window Functions

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

-- First, create and populate the products table

CREATE TABLE products (

category VARCHAR(50),

product\_name VARCHAR(100),

price DECIMAL(10,2)

);

-- Insert sample data

INSERT INTO products (category, product\_name, price) VALUES

('Electronics', 'iPhone 14', 999.99),

('Electronics', 'Samsung Galaxy S23', 899.99),

('Electronics', 'iPad Pro', 899.99), -- Same price as Samsung (tie)

('Electronics', 'MacBook Air', 1199.99),

('Electronics', 'Dell Laptop', 799.99),

('Electronics', 'AirPods Pro', 249.99),

('Clothing', 'Designer Jacket', 299.99),

('Clothing', 'Premium Jeans', 129.99),

('Clothing', 'Luxury Shirt', 129.99), -- Same price as jeans (tie)

('Clothing', 'Cashmere Sweater', 199.99),

('Clothing', 'Running Shoes', 159.99),

('Clothing', 'Basic T-Shirt', 29.99),

('Home & Garden', 'Coffee Machine', 599.99),

('Home & Garden', 'Vacuum Cleaner', 399.99),

('Home & Garden', 'Air Purifier', 299.99),

('Home & Garden', 'Blender', 299.99), -- Same price as air purifier (tie)

('Home & Garden', 'Toaster', 79.99);

-- Step 1: Using ROW\_NUMBER() - Assigns unique sequential numbers

SELECT

category,

product\_name,

price,

ROW\_NUMBER() OVER (PARTITION BY category ORDER BY price DESC) as row\_num

FROM products

ORDER BY category, row\_num;

-- Step 2: Using RANK() - Leaves gaps after ties

SELECT

category,

product\_name,

price,

RANK() OVER (PARTITION BY category ORDER BY price DESC) as rank\_position

FROM products

ORDER BY category, rank\_position;

-- Step 3: Using DENSE\_RANK() - No gaps after ties

SELECT

category,

product\_name,

price,

DENSE\_RANK() OVER (PARTITION BY category ORDER BY price DESC) as dense\_rank\_position

FROM products

ORDER BY category, dense\_rank\_position;

-- Step 4: Comparing all three ranking functions side by side

SELECT

category,

product\_name,

price,

ROW\_NUMBER() OVER (PARTITION BY category ORDER BY price DESC) as row\_num,

RANK() OVER (PARTITION BY category ORDER BY price DESC) as rank\_pos,

DENSE\_RANK() OVER (PARTITION BY category ORDER BY price DESC) as dense\_rank\_pos

FROM products

ORDER BY category, price DESC;

-- Step 5: Finding TOP 3 most expensive products in each category using different methods

-- Method 1: Using ROW\_NUMBER() (always gets exactly 3 per category)

WITH ranked\_products\_row AS (

SELECT

category,

product\_name,

price,

ROW\_NUMBER() OVER (PARTITION BY category ORDER BY price DESC) as row\_num

FROM products

)

SELECT

category,

product\_name,

price,

row\_num

FROM ranked\_products\_row

WHERE row\_num <= 3

ORDER BY category, row\_num;

-- Method 2: Using RANK() (may get more than 3 if there are ties in 3rd position)

WITH ranked\_products\_rank AS (

SELECT

category,

product\_name,

price,

RANK() OVER (PARTITION BY category ORDER BY price DESC) as rank\_pos

FROM products

)

SELECT

category,

product\_name,

price,

rank\_pos

FROM ranked\_products\_rank

WHERE rank\_pos <= 3

ORDER BY category, rank\_pos;

-- Method 3: Using DENSE\_RANK() (includes all products with top 3 distinct prices)

WITH ranked\_products\_dense AS (

SELECT

category,

product\_name,

price,

DENSE\_RANK() OVER (PARTITION BY category ORDER BY price DESC) as dense\_rank\_pos

FROM products

)

SELECT

category,

product\_name,

price,

dense\_rank\_pos

FROM ranked\_products\_dense

WHERE dense\_rank\_pos <= 3

ORDER BY category, dense\_rank\_pos;

-- BONUS: Advanced example with additional window functions

SELECT

category,

product\_name,

price,

-- Ranking functions

ROW\_NUMBER() OVER (PARTITION BY category ORDER BY price DESC) as row\_num,

RANK() OVER (PARTITION BY category ORDER BY price DESC) as rank\_pos,

DENSE\_RANK() OVER (PARTITION BY category ORDER BY price DESC) as dense\_rank\_pos,

-- Additional window functions

COUNT(\*) OVER (PARTITION BY category) as total\_products\_in\_category,

AVG(price) OVER (PARTITION BY category) as avg\_price\_in\_category,

MAX(price) OVER (PARTITION BY category) as max\_price\_in\_category,

MIN(price) OVER (PARTITION BY category) as min\_price\_in\_category,

-- Percentage calculations

ROUND(price / MAX(price) OVER (PARTITION BY category) \* 100, 2) as pct\_of\_max\_price,

ROUND(price / AVG(price) OVER (PARTITION BY category) \* 100, 2) as pct\_of\_avg\_price

FROM products

ORDER BY category, price DESC;

/\*

KEY DIFFERENCES EXPLAINED:

1. ROW\_NUMBER():

- Always assigns unique sequential numbers (1, 2, 3, 4...)

- Even for tied values, each gets a different number

- Useful when you need exactly N records per group

2. RANK():

- Assigns same rank to tied values

- Leaves gaps after ties (1, 2, 2, 4, 5...)

- The next rank after a tie skips numbers equal to the tie count

3. DENSE\_RANK():

- Assigns same rank to tied values

- No gaps after ties (1, 2, 2, 3, 4...)

- The next rank after a tie continues sequentially

WHEN TO USE EACH:

- ROW\_NUMBER(): When you need exactly N items per group

- RANK(): When you want to show ties but preserve the "true" position

- DENSE\_RANK(): When you want to show ties and count distinct ranking levels

PARTITION BY: Divides the result set into partitions and applies the window function to each partition separately.

ORDER BY: Defines the order within each partition for ranking.

\*/

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a table

AI-generated content may be incorrect.

A table with numbers and a price list

AI-generated content may be incorrect.

A screenshot of a table

AI-generated content may be incorrect.

A screenshot of a table

AI-generated content may be incorrect.

A screenshot of a table

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a table

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

A table of numbers and letters

AI-generated content may be incorrect.



Exercise 1:Creating a Stored Procedure

-- STEP 1: Create Employees table

DROP TABLE IF EXISTS Employees;

CREATE TABLE Employees (

EmployeeID INT IDENTITY(1,1) PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

DepartmentID INT,

Salary DECIMAL(10,2),

JoinDate DATE

);

-- STEP 2: Insert sample data

INSERT INTO Employees (FirstName, LastName, DepartmentID, Salary, JoinDate)

VALUES

('Alice', 'Smith', 1, 50000.00, '2022-01-15'),

('Bob', 'Johnson', 2, 60000.00, '2021-11-10'),

('Charlie', 'Williams', 1, 55000.00, '2023-03-12'),

('Diana', 'Brown', 3, 70000.00, '2020-07-20'),

('Evan', 'Davis', 2, 58000.00, '2024-05-05');

GO

-- STEP 3: Drop and create the procedure to get employees by department

DROP PROCEDURE IF EXISTS sp\_GetEmployeesByDepartment;

GO

CREATE PROCEDURE sp\_GetEmployeesByDepartment

@DepartmentID INT

AS

BEGIN

SELECT EmployeeID, FirstName, LastName, DepartmentID, Salary, JoinDate

FROM Employees

WHERE DepartmentID = @DepartmentID;

END;

GO

-- STEP 4: Drop and create the procedure to insert employee

DROP PROCEDURE IF EXISTS sp\_InsertEmployee;

GO

CREATE PROCEDURE sp\_InsertEmployee

@FirstName VARCHAR(50),

@LastName VARCHAR(50),

@DepartmentID INT,

@Salary DECIMAL(10,2),

@JoinDate DATE

AS

BEGIN

INSERT INTO Employees (FirstName, LastName, DepartmentID, Salary, JoinDate)

VALUES (@FirstName, @LastName, @DepartmentID, @Salary, @JoinDate);

END;

GO

A white background with black text

AI-generated content may be incorrect.

Exercise 5:Return Data from a Stored Procedure

-- Drop the procedure if it already exists

DROP PROCEDURE IF EXISTS sp\_GetEmployeeCountByDepartment;

GO

-- Create the procedure

CREATE PROCEDURE sp\_GetEmployeeCountByDepartment

@DepartmentID INT

AS

BEGIN

SELECT COUNT(\*) AS TotalEmployees

FROM Employees

WHERE DepartmentID = @DepartmentID;

END;

GO

EXEC sp\_GetEmployeeCountByDepartment @DepartmentID = 1;

A screenshot of a computer

AI-generated content may be incorrect.