```
/* ================================================================================
   SQL Window Functions
--------------------------------------------------------------------------------
   SQL window functions enable advanced calculations across sets of rows
   related to the current row without resorting to complex subqueries or joins.
   This script demonstrates the fundamentals and key clauses of window functions,
   including the OVER, PARTITION, ORDER, and FRAME clauses, as well as common rules
   and a GROUP BY use case.

   Table of Contents:
     1. SQL Window Basics
     2. SQL Window OVER Clause
     3. SQL Window PARTITION Clause
     4. SQL Window ORDER Clause
     5. SQL Window FRAME Clause
     6. SQL Window Rules
     7. SQL Window with GROUP BY
================================================================================
*/

/* ================================================================================
   SQL WINDOW FUNCTIONS | BASICS
================================================================================*/

/* TASK 1:
   Calculate the Total Sales Across All Orders
*/
use SalesDB;  -- Use the appropriate database
SELECT
    SUM(Sales) AS Total_Sales
FROM Sales.Orders;

/* TASK 2:
   Calculate the Total Sales for Each Product
*/

SELECT
    ProductID,
    SUM(Sales) AS Total_Sales
FROM Sales.Orders
GROUP BY ProductID;

/* ================================================================================
   SQL WINDOW FUNCTIONS | OVER CLAUSE
================================================================================*/

/* TASK 3:
   Find the total sales across all orders,
   additionally providing details such as OrderID and OrderDate
*/
USE SalesDB;
SELECT
      OrderID,
      OrderDate,
```

```
      SUM(Sales) OVER (PARTITION BY OrderID ) AS Total_Sales
FROM Sales.Orders;




SELECT
    OrderID,
    OrderDate,
    ProductID,
    Sales,
    SUM(Sales) OVER () AS Total_Sales
FROM Sales.Orders;

/* ============================================================================
   SQL WINDOW FUNCTIONS | PARTITION CLAUSE
============================================================================*/

/* TASK 4:
   Find the total sales across all orders and for each product,
   additionally providing details such as OrderID and OrderDate
*/

SELECT
    OrderID,
    OrderDate,
    ProductID,
    Sales,
    SUM(Sales) OVER () AS Total_Sales,
    SUM(Sales) OVER (PARTITION BY ProductID) AS Sales_By_Product
FROM Sales.Orders;

/* TASK 5:
   Find the total sales across all orders, for each product,
   and for each combination of product and order status,
   additionally providing details such as OrderID and OrderDate
*/

SELECT
    OrderID,
    OrderDate,
    ProductID,
    OrderStatus,
    Sales,
    SUM(Sales) OVER () AS Total_Sales,
    SUM(Sales) OVER (PARTITION BY ProductID) AS Sales_By_Product,
    SUM(Sales) OVER (PARTITION BY ProductID, OrderStatus) AS Sales_By_Product_Status
FROM Sales.Orders;

/* ============================================================================
   SQL WINDOW FUNCTIONS | ORDER CLAUSE
============================================================================*/
```

```
/* TASK 6:
   Rank each order by Sales from highest to lowest */
SELECT
    OrderID,
    OrderDate,
    Sales,
    RANK() OVER (ORDER BY Sales DESC) AS Rank_Sales
FROM Sales.Orders;

/* ============================================================================
   SQL WINDOW FUNCTIONS | FRAME CLAUSE
============================================================================*/

/* TASK 7:
   Calculate Total Sales by Order Status for current and next two orders
*/

SELECT
    OrderID,
    OrderDate,
    ProductID,
    OrderStatus,
    Sales,
    SUM(Sales) OVER (
        PARTITION BY OrderStatus
        ORDER BY OrderDate
        ROWS BETWEEN CURRENT ROW AND 2 FOLLOWING
    ) AS Total_Sales
FROM Sales.Orders;

/* TASK 8:
   Calculate Total Sales by Order Status for current and previous two orders
*/
USE SalesDB;
SELECT
    OrderID,
    OrderDate,
    ProductID,
    OrderStatus,
    Sales,
    SUM(Sales) OVER (
        PARTITION BY OrderStatus
        ORDER BY OrderDate
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS Total_Sales
FROM Sales.Orders;

/* TASK 9:
   Calculate Total Sales by Order Status from previous two orders only
*/
SELECT
    OrderID,
    OrderDate,
    ProductID,
```

```
    OrderStatus,
    Sales,
    SUM(Sales) OVER (
        PARTITION BY OrderStatus
        ORDER BY OrderDate
        ROWS 2 PRECEDING
    ) AS Total_Sales
FROM Sales.Orders;

/* TASK 10:
   Calculate cumulative Total Sales by Order Status up to the current order
*/
USE SalesDB;
SELECT
    OrderID,
    OrderDate,
    ProductID,
    OrderStatus,
    Sales,
    SUM(Sales) OVER (
        PARTITION BY OrderStatus
        ORDER BY OrderDate
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS Total_Sales
FROM Sales.Orders;

/* TASK 11:
   Calculate cumulative Total Sales by Order Status from the start to the current row
*/
SELECT
    OrderID,
    OrderDate,
    ProductID,
    OrderStatus,
    Sales,
    SUM(Sales) OVER (
        PARTITION BY OrderStatus
        ORDER BY OrderDate
        ROWS UNBOUNDED PRECEDING
    ) AS Total_Sales
FROM Sales.Orders;

/* ============================================================================
   SQL WINDOW FUNCTIONS | RULES
============================================================================*/

/* RULE 1:
   Window functions can only be used in SELECT or ORDER BY clauses
*/
SELECT
    OrderID,
    OrderDate,
    ProductID,
    OrderStatus,
```

```
    Sales,
    SUM(Sales) OVER (PARTITION BY OrderStatus) AS Total_Sales
FROM Sales.Orders
WHERE SUM(Sales) OVER (PARTITION BY OrderStatus) > 100;  -- Invalid: window function in
WHERE clause

/* RULE 2:
   Window functions cannot be nested
*/
SELECT
    OrderID,
    OrderDate,
    ProductID,
    OrderStatus,
    Sales,
    SUM(SUM(Sales) OVER (PARTITION BY OrderStatus)) OVER (PARTITION BY OrderStatus) AS
Total_Sales  -- Invalid nesting
FROM Sales.Orders;

/* ============================================================================
   SQL WINDOW FUNCTIONS | GROUP BY
============================================================================*/

/* TASK 12:
   Rank customers by their total sales
*/
SELECT
    CustomerID,
    SUM(Sales) AS Total_Sales,
    RANK() OVER (ORDER BY SUM(Sales) DESC) AS Rank_Customers
FROM Sales.Orders
GROUP BY CustomerID;


/* TASK 13:
   Rank customers by their total sales, partitioned by OrderStatus
*/
SELECT
    CustomerID,
    SUM(Sales) AS Total_Sales,
    RANK() OVER (PARTITION BY OrderStatus ORDER BY SUM(Sales) DESC) AS Rank_Customers
FROM Sales.Orders
GROUP BY CustomerID, OrderStatus;


/* TASK 14:
   Rank customers by their total sales, partitioned by OrderStatus and ProductID
*/
SELECT
    CustomerID,
    ProductID,
    SUM(Sales) AS Total_Sales,
    RANK() OVER (PARTITION BY OrderStatus, ProductID ORDER BY SUM(Sales) DESC) AS
Rank_Customers
```

```
FROM Sales.Orders
GROUP BY CustomerID, OrderStatus, ProductID;
```

```
/* ================================================================================
   SQL Window Aggregate Functions
--------------------------------------------------------------------------------
   These functions allow you to perform aggregate calculations over a set
   of rows without the need for complex subqueries. They enable you to compute
   counts, sums, averages, minimums, and maximums while still retaining access
   to individual row details.

   Table of Contents:
    1. COUNT
    2. SUM
    3. AVG
    4. MAX / MIN
    5. ROLLING SUM & AVERAGE Use Case
================================================================================
*/

/* ============================================================
   SQL WINDOW AGGREGATION | COUNT
   ============================================================ */

/* TASK 1:
   Find the Total Number of Orders and the Total Number of Orders for Each Customer
*/

USE SalesDB;

SELECT
    OrderID,
    OrderDate,
    CustomerID,
    COUNT(*) OVER() AS TotalOrders,
    COUNT(*) OVER(PARTITION BY CustomerID) AS OrdersByCustomers
FROM Sales.Orders

/* TASK 2:
   - Find the Total Number of Customers
   - Find the Total Number of Scores for Customers
   - Find the Total Number of Countries
*/

SELECT TOP 1
      COUNT(CustomerID) OVER() AS TotalCustomers,
      SUM(Score) OVER() AS TotalScores,
      COUNT(Country) OVER() AS TotalCountries
```

```
FROM Sales.Customers


/* TASK 3:
   Check whether the table 'OrdersArchive' contains any duplicate rows
*/
SELECT
      *
FROM Sales.OrdersArchive
WHERE OrderID IN (
    SELECT OrderID
    FROM Sales.OrdersArchive
    GROUP BY OrderID
    HAVING COUNT(*) > 1
)


/* TASK 3.1:
   Check whether the table 'OrdersArchive' contains any duplicate rows
   using window function
*/
SELECT
    *
FROM (
    SELECT
        *,
        COUNT(*) OVER(PARTITION BY OrderID) AS CheckDuplicates
    FROM Sales.OrdersArchive
) t
WHERE CheckDuplicates > 1

/* ============================================================
   SQL WINDOW AGGREGATION | SUM
   ============================================================ */

/* TASK 4:
   - Find the Total Sales Across All Orders
   - Find the Total Sales for Each Product
*/


SELECT
    OrderID,
    OrderDate,
    Sales,
    ProductID,
    SUM(Sales) OVER () AS TotalSales,
    SUM(Sales) OVER (PARTITION BY ProductID) AS SalesByProduct
FROM Sales.Orders

/* TASK 5:
   Find the Percentage Contribution of Each Product's Sales to the Total Sales
*/
```

```sql
SELECT
    OrderID,
    ProductID,
    Sales,
    SUM(Sales) OVER () AS TotalSales,
    ROUND(CAST(Sales AS FLOAT) / SUM(Sales) OVER () * 100, 2) AS PercentageOfTotal
FROM Sales.Orders

/* ============================================================
   SQL WINDOW AGGREGATION | AVG
   ============================================================ */

/* TASK 6:
   - Find the Average Sales Across All Orders
   - Find the Average Sales for Each Product
*/

SELECT
    OrderID,
    OrderDate,
    Sales,
    ProductID,
    AVG(Sales) OVER () AS AvgSales,
    AVG(Sales) OVER (PARTITION BY ProductID) AS AvgSalesByProduct
FROM Sales.Orders

/* TASK 7:
   Find the Average Scores of Customers
*/
SELECT
    CustomerID,
    LastName,
    Score,
    COALESCE(Score, 0) AS CustomerScore,
    AVG(Score) OVER () AS AvgScore,
    AVG(COALESCE(Score, 0)) OVER () AS AvgScoreWithoutNull
FROM Sales.Customers

/* TASK 8:
   Find all orders where Sales exceed the average Sales across all orders
*/
USE SalesDB;

SELECT
    *
FROM (
    SELECT
        OrderID,
        ProductID,
        Sales,
        AVG(Sales) OVER () AS Avg_Sales
    FROM Sales.Orders
) t
WHERE Sales > Avg_Sales
```

```sql
/* ============================================================
   SQL WINDOW AGGREGATION | MAX / MIN
   ============================================================ */

/* TASK 9:
   Find the Highest and Lowest Sales across all orders
*/

SELECT
    MIN(Sales) AS MinSales,
    MAX(Sales) AS MaxSales
FROM Sales.Orders

/* TASK 10:
   Find the Lowest Sales across all orders and by Product
*/
SELECT
    OrderID,
    ProductID,
    OrderDate,
    Sales,
    MIN(Sales) OVER () AS LowestSales,
    MIN(Sales) OVER (PARTITION BY ProductID) AS LowestSalesByProduct
FROM Sales.Orders


/* TASK 11:
   Show the employees who have the highest salaries
*/
SELECT
      *
FROM Sales.Employees
WHERE Salary = (SELECT MAX(Salary) FROM Sales.Employees)




/* TASK 11.1:
   Show the employees who have the highest salaries using window function
*/


SELECT *
FROM (
 SELECT *,
     MAX(Salary) OVER() AS HighestSalary
 FROM Sales.Employees
) t
WHERE Salary = HighestSalary

/* TASK 12:
   Find the deviation of each Sale from the minimum and maximum Sales
```

```
*/
SELECT
      CustomerID,
      Sales,
      Sales - MIN(Sales) OVER () AS DeviationFromMin,
      MAX(Sales) OVER () - Sales AS DeviationFromMax
FROM Sales.Orders




SELECT
    OrderID,
    OrderDate,
    ProductID,
    Sales,
    MAX(Sales) OVER () AS HighestSales,
    MIN(Sales) OVER () AS LowestSales,
    Sales - MIN(Sales) OVER () AS DeviationFromMin,
    MAX(Sales) OVER () - Sales AS DeviationFromMax
FROM Sales.Orders

/* ============================================================
   Use Case | ROLLING SUM & AVERAGE
   ============================================================ */

/* TASK 13:
   Calculate the moving average of Sales for each Product over time
*/

SELECT
    OrderID,
    ProductID,
    OrderDate,
    Sales,
    AVG(Sales) OVER (PARTITION BY ProductID) AS AvgByProduct,
    AVG(Sales) OVER (PARTITION BY ProductID ORDER BY OrderDate) AS MovingAvg
FROM Sales.Orders

/* TASK 14:
   Calculate the moving average of Sales for each Product over time,
   including only the next order
*/

SELECT
    OrderID,
    ProductID,
    OrderDate,
    Sales,
    AVG(Sales) OVER (PARTITION BY ProductID ORDER BY OrderDate ROWS BETWEEN CURRENT ROW
AND 1 FOLLOWING) AS RollingAvg
FROM Sales.Orders
```

```
/* ================================================================================
   SQL Window Ranking Functions
--------------------------------------------------------------------------------
   These functions allow you to rank and order rows within a result set
   without the need for complex joins or subqueries. They enable you to assign
   unique or non-unique rankings, group rows into buckets, and analyze data
   distributions on ordered data.

   Table of Contents:
     1. ROW_NUMBER
     2. RANK
     3. DENSE_RANK
     4. NTILE
     5. CUME_DIST
================================================================================
*/

/* ============================================================
   SQL WINDOW RANKING | ROW_NUMBER, RANK, DENSE_RANK
   ============================================================ */

/* TASK 1:
   Rank Orders Based on Sales from Highest to Lowest
*/
USE SalesDB;
SELECT
    OrderID,
    ProductID,
    Sales,
    ROW_NUMBER() OVER (ORDER BY Sales DESC) AS SalesRank_Row,
    RANK() OVER (ORDER BY Sales DESC) AS SalesRank_Rank,
    DENSE_RANK() OVER (ORDER BY Sales DESC) AS SalesRank_Dense
FROM Sales.Orders;

/* TASK 2:
   Use Case | Top-N Analysis: Find the Highest Sale for Each Product
*/

SELECT *
FROM (
    SELECT
        OrderID,
        ProductID,
        Sales,
        ROW_NUMBER() OVER (PARTITION BY ProductID ORDER BY Sales DESC) AS RankByProduct
    FROM Sales.Orders
) AS TopProductSales
WHERE RankByProduct = 1;

/* TASK 3:
```

```
   Use Case | Bottom-N Analysis: Find the Lowest 2 Customers Based on Their Total Sales
*/

SELECT Top 2
    CustomerID,
    SUM(Sales) AS TotalSales
FROM Sales.Orders
GROUP BY CustomerID
Order BY TotalSales ASC;


/* Alternative using Window Functions */

SELECT *
FROM (
    SELECT
        CustomerID,
        SUM(Sales) AS TotalSales,
        ROW_NUMBER() OVER (ORDER BY SUM(Sales)) AS RankCustomers
    FROM Sales.Orders
    GROUP BY CustomerID
) AS BottomCustomerSales
WHERE RankCustomers <= 2;

/* TASK 4:
   Use Case | Assign Unique IDs to the Rows of the 'Order Archive'
*/
SELECT
    ROW_NUMBER() OVER (ORDER BY OrderID, OrderDate) AS UniqueID,
    *
FROM Sales.OrdersArchive;

/* TASK 5:
   Use Case | Identify Duplicates:
   Identify Duplicate Rows in 'Order Archive' and return a clean result without any
duplicates
*/

SELECT *
FROM (
    SELECT
        ROW_NUMBER() OVER (PARTITION BY OrderID ORDER BY CreationTime DESC) AS rn,
        *
    FROM Sales.OrdersArchive
) AS UniqueOrdersArchive
WHERE rn = 1;

/* ============================================================
   SQL WINDOW RANKING | NTILE
   ============================================================ */

/* TASK 6:
   Divide Orders into Groups Based on Sales
*/
```

```sql
USE SalesDB;

SELECT
    OrderID,
    Sales,
    ProductID,
    NTILE(1) OVER (ORDER BY Sales) AS OneBucket,
    NTILE(2) OVER (ORDER BY Sales) AS TwoBuckets,
    NTILE(3) OVER (ORDER BY Sales) AS ThreeBuckets,
    NTILE(4) OVER (ORDER BY Sales) AS FourBuckets,
    NTILE(2) OVER (PARTITION BY ProductID ORDER BY Sales) AS TwoBucketByProducts
FROM Sales.Orders;

/* TASK 7:
   Segment all Orders into 3 Categories: High, Medium, and Low Sales.
*/

SELECT
    OrderID,
    Sales,
    Buckets,
    CASE
        WHEN Buckets = 1 THEN 'High'
        WHEN Buckets = 2 THEN 'Medium'
        WHEN Buckets = 3 THEN 'Low'
    END AS SalesSegmentations
FROM (
    SELECT
        OrderID,
        Sales,
        NTILE(3) OVER (ORDER BY Sales DESC) AS Buckets
    FROM Sales.Orders
) AS SalesBuckets;

/* TASK 8:
   Divide Orders into Groups for Processing
*/
SELECT
    NTILE(5) OVER (ORDER BY OrderID) AS Buckets,
    *
FROM Sales.Orders;

/* ============================================================
   SQL WINDOW RANKING | CUME_DIST
   ============================================================ */

/* TASK 9:
   Find Products that Fall Within the Highest 40% of the Prices
*/
USE SalesDB;
SELECT
    Product,
    Price,
    DistRank,
```

```
    CONCAT(DistRank * 100, '%') AS DistRankPerc
FROM (
    SELECT
        Product,
        Price,
        CUME_DIST() OVER (ORDER BY Price DESC) AS DistRank
    FROM Sales.Products
) AS PriceDistribution
WHERE DistRank <= 0.4;




/* TASK 10:
   Use Case | Find the Top 10% of Sales
*/

SELECT
    OrderID,
    Sales,
    SalesRank,
    CONCAT(SalesRank * 100, '%') AS SalesRankPerc
FROM (
    SELECT
        OrderID,
        Sales,
        CUME_DIST() OVER (ORDER BY Sales DESC) AS SalesRank
    FROM Sales.Orders
) AS SalesDistribution
WHERE SalesRank <= 0.1;




/* TASK 11:
   Use Case | Find the Top 50% of Sales for Each Product
*/
SELECT
    OrderID,
    Sales,
    ProductID,
    SalesRank,
    CONCAT(SalesRank * 100, '%') AS SalesRankPerc
FROM (
    SELECT
        OrderID,
        Sales,
        ProductID,
        CUME_DIST() OVER (PARTITION BY ProductID ORDER BY Sales DESC) AS SalesRank
    FROM Sales.Orders
) AS ProductSalesDistribution
WHERE SalesRank <= 0.5;
```

```sql
/* TASK 12:
   Use Case | Find the Top 50% of Sales for Each Product and Each Customer
*/
SELECT
    OrderID,
    Sales,
    ProductID,
    CustomerID,
    SalesRank,
    CONCAT(SalesRank * 100, '%') AS SalesRankPerc
FROM (
    SELECT
        OrderID,
        Sales,
        ProductID,
        CustomerID,
        CUME_DIST() OVER (PARTITION BY ProductID, CustomerID ORDER BY Sales DESC) AS
SalesRank
    FROM Sales.Orders
) AS ProductCustomerSalesDistribution
WHERE SalesRank <= 0.5;


/* TASK 13:
   Use Case | Find the Top 10% of Sales for Each Product and Each Customer
   with DENSE_RANK and CUME_DIST
*/

SELECT
    OrderID,
    Sales,
    ProductID,
    CustomerID,
    SalesRank,
    CONCAT(SalesRank * 100, '%') AS SalesRankPerc
FROM (
    SELECT
        OrderID,
        Sales,
        ProductID,
        CustomerID,
        DENSE_RANK() OVER (PARTITION BY ProductID, CustomerID ORDER BY Sales DESC) AS
SalesRank,
        CUME_DIST() OVER (PARTITION BY ProductID, CustomerID ORDER BY Sales DESC) AS
SalesCumeDist
    FROM Sales.Orders
) AS ProductCustomerSalesDistribution
WHERE SalesRank <= 0.1;




/* ============================================================================
```

```
   SQL Window Value Functions
-------------------------------------------------------------------------------
   These functions let you reference and compare values from other rows
   in a result set without complex joins or subqueries, enabling advanced
   analysis on ordered data.

   Table of Contents:
     1. LEAD
     2. LAG
     3. FIRST_VALUE
     4. LAST_VALUE
===============================================================================
*/

/* ============================================================
   SQL WINDOW VALUE | LEAD, LAG
   ============================================================ */

/* TASK 1:
   Analyze the Month-over-Month Performance by Finding the Percentage Change in Sales
   Between the Current and Previous Months
*/
USE SalesDB;
SELECT
    *,
    CurrentMonthSales - PreviousMonthSales AS MoM_Change,
    ROUND(
        CAST((CurrentMonthSales - PreviousMonthSales) AS FLOAT)
        / PreviousMonthSales * 100, 1
    ) AS MoM_Perc
FROM (
    SELECT
        MONTH(OrderDate) AS OrderMonth,
        SUM(Sales) AS CurrentMonthSales,
        LAG(SUM(Sales)) OVER (ORDER BY MONTH(OrderDate)) AS PreviousMonthSales
    FROM Sales.Orders
    GROUP BY MONTH(OrderDate)
) AS MonthlySales;


/* TASK 2:
   Customer Loyalty Analysis - Rank Customers Based on the Average Days Between Their
Orders
*/
SELECT
    CustomerID,
    AVG(DaysUntilNextOrder) AS AvgDays,
    RANK() OVER (ORDER BY COALESCE(AVG(DaysUntilNextOrder), 999999)) AS RankAvg
FROM (
    SELECT
        OrderID,
        CustomerID,
        OrderDate AS CurrentOrder,
        LEAD(OrderDate) OVER (PARTITION BY CustomerID ORDER BY OrderDate) AS NextOrder,
```

```
        DATEDIFF(
            day,
            OrderDate,
            LEAD(OrderDate) OVER (PARTITION BY CustomerID ORDER BY OrderDate)
        ) AS DaysUntilNextOrder
    FROM Sales.Orders
) AS CustomerOrdersWithNext
GROUP BY CustomerID;

/* ============================================================
   SQL WINDOW VALUE | FIRST & LAST VALUE
   ============================================================ */

/* TASK 3:
   Find the Lowest and Highest Sales for Each Product,
   and determine the difference between the current Sales and the lowest Sales for each
Product.
*/
SELECT
    OrderID,
    ProductID,
    Sales,
    FIRST_VALUE(Sales) OVER (PARTITION BY ProductID ORDER BY Sales) AS LowestSales,
    LAST_VALUE(Sales) OVER (
        PARTITION BY ProductID
        ORDER BY Sales
        ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
    ) AS HighestSales,
    Sales - FIRST_VALUE(Sales) OVER (PARTITION BY ProductID ORDER BY Sales) AS
SalesDifference
FROM Sales.Orders;
```