Advanced SQL Techniques for Scalable Data Analysis and Reporting

Purpose: Demonstrate mastery of complex SQL queries using recursive CTEs, window functions,

JSON manipulation, dynamic pivot, lateral joins, and optimization hints.

Keywords: Advanced SQL, Recursive CTE, Window Functions, JSON_SQL, Dynamic Pivot, Query Optimization,

Lateral Join, APPLY Operator, Complex Joins, Data Analysis, Technical Writing, SEO Content

```
-- Step 1: Define a Recursive CTE for Hierarchical Data Traversal (e.g., Organization Hierarchy)

WITH RECURSIVE OrgHierarchy AS (

   -- Anchor member: Select top-level managers (no manager)

   SELECT

      EmployeeID,

      ManagerID,

      EmployeeName,

      Position,

      Department,

      1 AS Level,

      CAST(EmployeeID AS VARCHAR(1000)) AS HierarchyPath

   FROM

      Employees

   WHERE ManagerID IS NULL
```

```
    UNION ALL

    -- Recursive member: Get subordinates
    SELECT
        e.EmployeeID,
        e.ManagerID,
        e.EmployeeName,
        e.Position,
        e.Department,
        oh.Level + 1,
        CONCAT(oh.HierarchyPath, '->', e.EmployeeID)
    FROM
        Employees e
        INNER JOIN OrgHierarchy oh ON e.ManagerID = oh.EmployeeID
),

-- Step 2: Aggregate Sales Data with Advanced Window Functions and Filtering
SalesWindow AS (
    SELECT
        s.SalesID,
        s.EmployeeID,
        s.SaleDate,
        s.Amount,
        e.Department,
        -- Calculate running total sales per employee, ordered by date with sliding window of last 30 days
        SUM(s.Amount) OVER (
            PARTITION BY s.EmployeeID
            ORDER BY s.SaleDate
            RANGE BETWEEN INTERVAL '29' DAY PRECEDING AND CURRENT ROW
        ) AS Running30DaySales,

        -- Rank employees within each department by total sales descending
        RANK() OVER (
            PARTITION BY e.Department
            ORDER BY s.Amount DESC
        ) AS SalesRank,

        -- Calculate moving average of sales for each employee over 7 days
        AVG(s.Amount) OVER (
            PARTITION BY s.EmployeeID
            ORDER BY s.SaleDate
            ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
        ) AS MovingAvg7Day

    FROM
        Sales s
        INNER JOIN Employees e ON s.EmployeeID = e.EmployeeID
    WHERE
```

```
        s.SaleDate >= CURRENT_DATE - INTERVAL '90' DAY -- last 90 days for analysis
),

-- Step 3: Parse and Aggregate Complex JSON Data from a JSONB column (PostgreSQL syntax
example)
JsonExtract AS (
    SELECT
        p.ProductID,
        p.ProductName,

        -- Extract nested JSON attribute 'specs' -> 'dimensions' -> 'width' as numeric
        (p.ProductDetails->'specs'->>'width')::NUMERIC AS Width,

        -- Extract and aggregate array elements inside JSON 'tags'
        jsonb_array_elements_text(p.ProductDetails->'tags') AS Tag,

        -- Calculate JSON array length for 'features'
        jsonb_array_length(p.ProductDetails->'features') AS FeatureCount
    FROM
        Products p
    WHERE
        p.ProductDetails IS NOT NULL
),

-- Step 4: Dynamic Pivot Query using FILTER clause (PostgreSQL specific) or CASE WHEN
```

```
PivotSalesByMonth AS (
    SELECT
        EmployeeID,
        Department,
        EXTRACT(YEAR FROM SaleDate) AS SaleYear,
        EXTRACT(MONTH FROM SaleDate) AS SaleMonth,

        -- Dynamic monthly sales aggregation pivoted by months for last 6 months
        SUM(CASE WHEN EXTRACT(MONTH FROM SaleDate) = 1 THEN Amount ELSE 0
END) AS JanSales,
        SUM(CASE WHEN EXTRACT(MONTH FROM SaleDate) = 2 THEN Amount ELSE 0
END) AS FebSales,
        SUM(CASE WHEN EXTRACT(MONTH FROM SaleDate) = 3 THEN Amount ELSE 0
END) AS MarSales,
        SUM(CASE WHEN EXTRACT(MONTH FROM SaleDate) = 4 THEN Amount ELSE 0
END) AS AprSales,
        SUM(CASE WHEN EXTRACT(MONTH FROM SaleDate) = 5 THEN Amount ELSE 0
END) AS MaySales,
        SUM(CASE WHEN EXTRACT(MONTH FROM SaleDate) = 6 THEN Amount ELSE 0
END) AS JunSales

    FROM
        Sales
    WHERE
        SaleDate >= DATE_TRUNC('month', CURRENT_DATE) - INTERVAL '6 months'
    GROUP BY
        EmployeeID, Department, SaleYear, SaleMonth
```

```
),

-- Step 5: Using LATERAL JOIN / APPLY operator to get Top N customers per employee
TopCustomers AS (
    SELECT
        e.EmployeeID,
        e.EmployeeName,
        c.CustomerID,
        c.CustomerName,
        cs.TotalSpent
    FROM
        Employees e
        CROSS JOIN LATERAL (
            SELECT
                s.CustomerID,
                cu.CustomerName,
                SUM(s.Amount) AS TotalSpent
            FROM
                Sales s
                INNER JOIN Customers cu ON s.CustomerID = cu.CustomerID
            WHERE
                s.EmployeeID = e.EmployeeID
            GROUP BY
                s.CustomerID, cu.CustomerName
            ORDER BY
            TotalSpent DESC
        LIMIT 3 -- Top 3 customers per employee
    ) cs
    INNER JOIN Customers c ON cs.CustomerID = c.CustomerID
),

-- Step 6: Advanced Indexing Hint for Query Optimizer to Improve Join Performance
-- Note: syntax varies between RDBMS, example shown for SQL Server
IndexOptimizedQuery AS (
  SELECT /*+ INDEX(e idx_emp_department), INDEX(s idx_sales_employee) */
    e.EmployeeID,
    e.EmployeeName,
    SUM(s.Amount) AS TotalSales
  FROM
    Employees e WITH (INDEX(idx_emp_department))
    INNER JOIN Sales s WITH (INDEX(idx_sales_employee)) ON e.EmployeeID =
s.EmployeeID
  WHERE
    s.SaleDate BETWEEN DATEADD(month, -6, GETDATE()) AND GETDATE()
  GROUP BY
    e.EmployeeID, e.EmployeeName
),

-- Step 7: Final Aggregated Reporting Query Joining All Above CTEs
FinalReport AS (
  SELECT
    oh.EmployeeID,
    oh.EmployeeName,
    oh.Level AS OrgLevel,
    oh.HierarchyPath,
    sw.Running30DaySales,
    sw.SalesRank,
    sw.MovingAvg7Day,
    p.JanSales, p.FebSales, p.MarSales, p.AprSales, p.MaySales, p.JunSales,
    tc.CustomerName AS TopCustomer1,
    js.Width,
    js.FeatureCount
  FROM
    OrgHierarchy oh
    LEFT JOIN SalesWindow sw ON oh.EmployeeID = sw.EmployeeID
    LEFT JOIN PivotSalesByMonth p ON oh.EmployeeID = p.EmployeeID
    LEFT JOIN TopCustomers tc ON oh.EmployeeID = tc.EmployeeID
    LEFT JOIN JsonExtract js ON js.ProductID = (
        SELECT TOP 1 ProductID FROM Sales s WHERE s.EmployeeID = oh.EmployeeID
ORDER BY s.SaleDate DESC
    )
)

-- Select from the final CTE
SELECT
```

```
    EmployeeID,
    EmployeeName,
    OrgLevel,
    HierarchyPath,
    COALESCE(Running30DaySales, 0) AS Running30DaySales,
    COALESCE(SalesRank, 0) AS SalesRank,
    COALESCE(MovingAvg7Day, 0) AS MovingAvg7Day,
    COALESCE(JanSales, 0) AS JanSales,
    COALESCE(FebSales, 0) AS FebSales,
    COALESCE(MarSales, 0) AS MarSales,
    COALESCE(AprSales, 0) AS AprSales,
    COALESCE(MaySales, 0) AS MaySales,
    COALESCE(JunSales, 0) AS JunSales,
    TopCustomer1,
    Width,
    FeatureCount
FROM
    FinalReport
ORDER BY
    OrgLevel, EmployeeName;


/*******************************************************************************************
-- Explanation:
-- 1. Recursive CTE (OrgHierarchy) traverses an employee-manager hierarchy with a path string.
```

-- 2. Window functions compute running totals, ranks, and moving averages on sales data.

-- 3. JSON functions extract nested JSON data from ProductDetails column.

-- 4. Pivoting technique summarizes sales per month for visualization-ready format.

-- 5. Lateral join fetches top N customers per employee using CROSS APPLY.

-- 6. Index hints guide query optimizer for better performance on large datasets.

-- 7. FinalReport aggregates all pieces into a comprehensive report for business insights.

--

Keywords included: advanced sql, recursive cte, window functions, json_sql, lateral join, apply,

-- dynamic pivot, query optimization, indexing hints, complex joins, sales analysis, hierarchical data.