# SQL for Data Engineers

## SQL Querying Techniques for Data Engineers

**Gerald Britton**
Pluralsight Author

@GeraldBritton www.linkedin.com/in/geraldbritton

# SQL Subqueries

Executing queries within queries

AKA Nested or Inner queries

```
SELECT employee_name
FROM employees
WHERE employee_id = (
    SELECT manager_id FROM employees
    WHERE employee_name = 'John'
    );
```

◀ **Single value subquery**

◀ **Subquery returns single value**

```
SELECT product_name
FROM products
WHERE category_id IN (
    SELECT category_id FROM
categories
    WHERE category_name =
'Electronics'
    );
```

◀ **Multiple row subquery**

```
SELECT customer_name
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id =
        c.customer_id
    AND o.order_date > '2023-01-01’
);

SELECT
    customer_name,
    (SELECT COUNT(*) FROM orders
    WHERE orders.customer_id =
        customers.customer_id)
            AS order_count
FROM
    customers;
```

◂ **Correlated subquery**

◂ **WHERE EXISTS is a best practice**

◂ **Subquery in column list**

# Aggregation

Aggregation in SQL can be used to summarize and analyze data, which can help to get important insights and statistics.

# SQL Aggregation Functions

```
SELECT SUM(sales_amount) FROM sales;

SELECT AVG(sales_amount) FROM sales;

SELECT COUNT(*) FROM customers;

SELECT MIN(order_date) FROM orders;

SELECT MAX(order_date) FROM orders;

STDEV(), VAR(), COVAR() …
```

```sql
SELECT department_id,
    AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id
ORDER BY department_id;


SELECT department_id,
    AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id
HAVING AVG(salary) <= 50000
ORDER BY department_id;

SELECT *
FROM employees
WHERE salary >
    (SELECT AVG(salary) FROM
        employees);
```

◀ **Aggregation by category**

◀ **Average per department**

◀ **Aggregation with filtering**

◀ **Filter results by average salary**

◀ **Aggregation as a filter in a subquery**

# SQL Window Functions

Window functions do aggregations over subsets of rows in a dataset

```
SELECT employee_id,
       department_id,
       salary,

       ROW_NUMBER()
       OVER (
           PARTITION BY department_id
           ORDER BY salary DESC)
       AS rn
FROM employees;
```

- Query using a window function
- Window function
- OVER Clause
- PARTITION BY defines the subset
- ORDER BY controls evaluation of function

```
SELECT employee_id,
       department_id,
       salary,

       AVG(salary)
       OVER (
           PARTITION BY department_id
           )
       AS avg_salary
FROM employees;
```

- No ORDER BY clause

# Common Table Expressions (CTEs)

Writing subqueries ahead of time

```
WITH
    -- First CTE
    cte1 [(col1, col2, …)]
    AS (
        SELECT …),

    -- Second CTE
    cte2 [(col1, col2, …)]
    AS (
        SELECT …),

    -- Other CTEs

-- Outer query using the CTEs as
tables
SELECT | INSERT | UPDATE | DELETE
```

- Keyword WITH starts things off
- Common Table Expression (CTE) with optional list of column names
- Query definition
- Optional second common table expression
- Additional CTEs, as required
- Final query, using the common table expressions as tables

```
WITH
-- CTEs
    emps AS (
        SELECT * FROM employee
        WHERE BirthDate > '1990-01-
01'
    ),
    peeps AS (
        SELECT * FROM Person AS p
        WHERE LastName LIKE 'K%'
    )

-- Main query
SELECT CONCAT(p.FirstName, ' ‘,
              p.LastName)
                 AS full_name
FROM emps e
JOIN peeps p
    ON p.emp_id = e.emp_id
```

◂ **Example using two CTEs**

◂ **Employees, filtered by date of birth**

◂ **People, filter by last name**

◂ **Main query getting full name**

◂ **JOIN the CTEs on employee id**

# Recursive Common Table Expressions

Use CTEs to navigate hierarchical or graphical data with recursion

```
WITH
    -- Recursive CTE
    recurs [(col1, col2, …)]
    AS (
        -- Anchor member: Base case
        SELECT c1, c2, c3
        FROM table1
        WHERE [condition]

        UNION ALL

        -- Recursive member:
        SELECT c1, c2, c3
        FROM recurs
        WHERE [condition]
        )

-- Outer query using the recursive
SELECT | INSERT | UPDATE | DELETE
FROM recurs
```

◀ **Beginning of recursive CTE**

◀ **Anchor query: Base case**

◀ **UNION ALL operator is required**

◀ **Recursive query: References itself**

◀ **Main query: uses recursive CTE**

```sql
WITH RECURSIVE EmployeeHierarchy AS (
    SELECT
        employee_id, employee_name, manager_id,
        1 AS level
    FROM
        employees
    WHERE
        manager_id IS NULL

    UNION ALL

    SELECT
        employee_id, employee_name, manager_id,
        eh.level + 1 AS level
    FROM
        employees e
    INNER JOIN
        EmployeeHierarchy eh
            ON e.manager_id = eh.employee_id
)
SELECT
    employee_id, employee_name, manager_id,
    level
FROM
    EmployeeHierarchy;
```

- ◂ **Recursive CTE for employees**
- ◂ **Anchor member**
- ◂ **Top-level managers have NULL as manager_id**
- ◂ **Recursive Member**
- ◂ **Self reference**
- ◂ **Recursion stops when JOIN returns no results**
- ◂ **Main query**

# SQL JOIN operations

Combining data from two or more tables

# SQL Join types

Table 1

Table 2

| JOIN type | Results returned | Row count |
|---|---|---|
| `INNER JOIN` | Matching rows from both tables | 0 - \| largest \| |
| `FULL OUTER JOIN` | All rows from both tables | \| largest \| |
| `LEFT OUTER JOIN` | All rows from left table, matching or not | \| Table 1 \| |
| `RIGHT OUTER JOIN` | All rows from right table, matching or not | \| Table 2 \| |
| `CROSS JOIN` | Cartesian product: all possible combinations | \|T1\| * \|T2\| |

# Dynamic SQL

Building SQL statements at runtime

# Dynamic SQL in T-SQL on SQL Server

```sql
DECLARE @ColumnName NVARCHAR(50)
DECLARE @SQLQuery NVARCHAR(MAX)

SET @ColumnName = 'FirstName' -- Example column name

SET @SQLQuery = '
    SELECT ' + ColumnName + '
    FROM Employees
'

-- Execute the dynamic SQL query
EXEC sp_executesql @SQLQuery
```

# Dynamic SQL in PL-SQL on Oracle

```
DECLARE
    ColumnName VARCHAR2(50);
    SQLQuery VARCHAR2(1000);
    result_value VARCHAR2(100);
BEGIN
    ColumnName := 'FirstName';

    -- Build the dynamic statement
    SQLQuery := 'SELECT "' || ColumnName || '" FROM Employees';

    -- Execute the dynamic SQL query
    EXECUTE IMMEDIATE SQLQuery INTO result_value;
END;
```

# Dynamic SQL in PL/pgSQL on PostgreSQL

```
DO $$
DECLARE
    ColumnName TEXT;
    SQLQuery TEXT;
    result_value TEXT;
BEGIN
    ColumnName := 'FirstName’;

    -- Build the dynamic statement
    SQLQuery := 'SELECT "' || ColumnName || '" FROM Employees';

    -- Execute the dynamic SQL query
    EXECUTE SQLQuery INTO result_value;
END;
$$;
```

# Dynamic SQL Gotchas

Maintenance

Performance

SQL Injection

'FirstName; DROP TABLE Employees;'