Advanced SQL Queries

Optimizing SQL for Analytics in the Cloud

NASHVILLE SOFTWARE SCHOOL

Objectives

1. Recall the core SQL building blocks (SELECT, JOIN, GROUP BY, ORDER BY)
2. Recognize when a subquery would be effective.
3. Write scalar subqueries, IN subqueries and EXISTS subqueries.
4. Use derived tables (subqueries in FROM) to simplify queries.
5. Construct Common Table Expressions (CTEs) to break down complex queries into readable steps.
6. Compare the tradeoffs of subqueries and CTEs in terms of clarity and reuse.

Review: What is SQL

SQL = Structured Query Language

- Standard language for interacting with relational databases
- As an analyst, you will use SQL primarily to retrieve data
- Can also:
    - Manipulate data: insert, update, and delete records
    - Define structures: create/alter tables, views, and indexes
- Lets you work directly with large datasets efficiently
- Often used to pull data into other tools (Excel, Python, Power BI) for further analysis
- Is made up of a collection of **keywords** (SELECT, FROM, WHERE, etc.)

The Foundation: SELECT + FROM

Purpose: Pull columns from one or more tables

SELECT chooses what columns to display in the output.

FROM specifies the source table or tables.

Example:

```
SELECT first_name, last_name
FROM customers;
```

Filtering with WHERE

Purpose: Restrict rows returned or included in a calculation

Requires a conditional statement of some kind:

- Simple conditions (=, <, >, < >)
- Pattern matching (LIKE, ILIKE)
- Combining conditions (AND, OR)

Example:

```
SELECT *
FROM orders
WHERE amount > 100
  AND status = 'Shipped';
```

JOINs and ON

Purpose: Combine related data stored in different tables.

ON specifies how to match up rows.
The type of join determines what to do with rows with no matches:

- INNER JOIN: only matching rows
- LEFT JOIN: all from the left table, matches from right
- OUTER JOIN: all rows from both tables

Example:

```
SELECT c.customer_name, o.order_id, o.amount
FROM customers c
JOIN orders o
  ON c.customer_id = o.customer_id;
```

Aggregates and GROUP BY

Purpose: Summarizing Data

SQL supports many types of aggregates:

- COUNT, SUM, AVG, MIN, MAX

Can perform aggregates at the group level by using GROUP BY.

Example:

```
SELECT customer_id, SUM(amount) AS total_spent
FROM orders
GROUP BY customer_id;
```

Filtering with HAVING

Purpose: Filters grouped results

Uses aggregate functions:

- SUM, COUNT, AVG, MAX, MIN

Syntax: follows the GROUP BY clause

Example:

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING AVG(salary) > 50000;
```

NASHVILLE SOFTWARE SCHOOL

Sorting Results

Purpose: Ordering Your Output

ORDER BY can order the output by one or more columns in ascending or descending order.
Can be combined with LIMIT to return the top N results.

Example:

```
SELECT product_name, price
FROM products
ORDER BY price DESC
LIMIT 5;
```

Subqueries

A **subquery** is a query nested inside another query.

Can be used in SELECT, FROM, WHERE, and HAVING.

Help break complex questions into smaller, manageable steps.

Three types: scalar, row, and table.

Scalar Subqueries

- Returns one value (one row and one column)
- Can be used in SELECT or WHERE.

Example in WHERE:

```
SELECT order_id, amount
FROM orders
WHERE amount > (SELECT AVG(amount) FROM orders);
```

Example in SELECT:

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

NASHVILLE SOFTWARE SCHOOL

Row Subqueries

- Returns one row with multiple columns
- Can be used in WHERE.
- Useful for comparing multiple attributes at once.

Example:

```
SELECT *
FROM employees
WHERE (department_id, salary) =
     (SELECT department_id, MAX(salary)
      FROM employees
      GROUP BY department_id
      ORDER BY MAX(salary) DESC
      LIMIT 1);
```

Table Subqueries

- Return a table (one or more rows and columns)
- Can be used in FROM (derived tables) or WHERE … IN / EXISTS.

Example in FROM:

```
SELECT customer_id, total_amount
FROM (SELECT customer_id, SUM(amount) AS total_amount
    FROM orders
    GROUP BY customer_id) AS customer_totals
WHERE total_amount > 1000;
```

Table Subqueries

Example in WHERE and IN:

```
SELECT *
FROM products
WHERE product_id IN (SELECT product_id FROM order_items);
```

Example using EXISTS:

```
SELECT customer_id, customer_name
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id
);
```

Common Table Expressions (CTEs)

A CTE is a temporary named result set you can reference within your main query (or a subsequent CTE).

Purpose:

- Improve readability of complex queries
- Break queries into manageable steps
- Avoid repeating the same subquery multiple times

Syntax:

```
WITH cte_name AS (
    -- your query here
)
SELECT …
FROM cte_name;
```

Common Table Expressions (CTEs)

Simple CTE Example:

```
WITH customer_totals AS (
    SELECT customer_id, SUM(amount) AS total_amount
    FROM orders
    GROUP BY customer_id
)
SELECT *
FROM customer_totals
WHERE total_amount > 1000;
```

NASHVILLE SOFTWARE SCHOOL

Common Table Expressions (CTEs)

Multi-step CTE:

```
WITH customer_totals AS (
    SELECT customer_id, region, SUM(amount) AS total_amount
    FROM orders
    GROUP BY customer_id, region
),
top_customers AS (
    SELECT *
    FROM customer_totals
    WHERE total_amount > 1000
)
SELECT region, COUNT(*) AS num_top_customers
FROM top_customers
GROUP BY region;
```

CTEs vs Subqueries

CTEs:
- Easier to read and debug
- Can be references multiple times
- Good for multi-step transformations

Subqueries:
- Inline, often shorter for one-off calculations
- Sometimes more efficient for small datasets