Varun Sagar
Theegala

TOP 10 PRACTICAL TIPS TO ENHANCE SQL JOINS & SUB-QUERIES

Varun Sagar Theegala

Efficient use of JOINs and subqueries is key to **writing high-performance SQL.**

Mastering the **right techniques** reduces query time and complexity, while knowing when to use INNER, LEFT, or RIGHT JOINs and **manage subqueries optimizes query speed and accuracy.**

Varun Sagar
Theegala

LET’S VISIT 10 TIPS TO NOTE WHEN USING JOINS & SUB-QUERIES

#1

Use INNER JOIN for Matching Data

WHAT IT IS ?

Retrieve rows that match in both tables.

WHY IT MATTERS ?

INNER JOIN is the most efficient for filtering matching data, ensuring only relevant rows are returned.

HOW TO IMPLEMENT ?

Use INNER JOIN to retrieve data where both tables have matching rows.

LET'S WORK WITH AN EXAMPLE

BEFORE IMPLEMENTATION

```sql
SELECT customers.name, orders.order_id
FROM customers, orders
WHERE customers.customer_id = orders.customer_id;
```

AFTER IMPLEMENTATION

```sql
SELECT customers.name, orders.order_id
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id;
```

Varun Sagar Theegala

NEXT

#2

Use LEFT JOIN for Non-Matching Data

WHAT IT IS ?

Return all rows from the left table, even if there's no match.

WHY IT MATTERS ?

LEFT JOIN helps you find missing or unmatched data, providing a full dataset even when no match exists.

HOW TO IMPLEMENT ?

Use LEFT JOIN when you need all records from the left table, regardless of matches.

Varun Sagar Theegala

NEXT

LET'S WORK WITH AN EXAMPLE

BEFORE IMPLEMENTATION

```sql
SELECT customers.name, orders.order_id
FROM customers, orders
WHERE customers.customer_id = orders.customer_id;
```

AFTER IMPLEMENTATION

```sql
SELECT customers.name, orders.order_id
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id;
```

#3

Avoid Subqueries When a JOIN Will Do

WHAT IT IS ?

Replace subqueries with JOINs whenever possible.

WHY IT MATTERS ?

JOINs are generally faster than subqueries, reducing the query’s complexity and improving performance.

HOW TO IMPLEMENT ?

Convert subqueries to JOIN when data retrieval can be done with a simple match.

LET'S WORK WITH AN EXAMPLE

BEFORE IMPLEMENTATION

```sql
SELECT name
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders WHERE order_total > 100);
```

AFTER IMPLEMENTATION

```sql
SELECT customers.name
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id
WHERE orders.order_total > 100;
```

Varun Sagar
Theegala

NEXT

#4

Use LATERAL for Row-by-Row Joins

WHAT IT IS ?

Use LATERAL for table-valued functions or row-wise complex joins.

WHY IT MATTERS ?

LATERAL executes row-by-row, making it ideal when each row depends on a subquery or function, giving more granular results.

HOW TO IMPLEMENT ?

Implement LATERAL to join the result of a row-wise subquery or table function.

LET'S WORK WITH AN EXAMPLE

BEFORE IMPLEMENTATION

```sql
SELECT p.product_id, o.order_id
FROM products p
JOIN orders o ON p.product_id = o.product_id
WHERE o.order_date = (
    SELECT MAX(order_date)
    FROM orders
    WHERE product_id = p.product_id
);
```

AFTER IMPLEMENTATION

```sql
SELECT p.product_id, o.order_id
FROM products p
LATERAL (
    SELECT order_id, order_date
    FROM orders o
    WHERE o.product_id = p.product_id
    ORDER BY order_date DESC
    LIMIT 3
) o;
```

Varun Sagar
Theegala

NEXT

#5

Use RIGHT JOIN Sparingly

WHAT IT IS ?

Return all rows from the right table, with matching rows from the left.

WHY IT MATTERS ?

RIGHT JOIN can often be less intuitive, and LEFT JOIN provides the same result with better readability, making queries easier to maintain.

HOW TO OPTIMISE ?

Default to LEFT JOIN unless there’s a clear use case for RIGHT JOIN.

LET'S WORK WITH AN EXAMPLE

BEFORE IMPLEMENTATION

```
SELECT customers.name, orders.order_id
FROM orders
RIGHT JOIN customers ON customers.customer_id = orders.customer_id;
```

AFTER IMPLEMENTATION

```
SELECT customers.name, orders.order_id
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id;
```

Varun Sagar Theegala

NEXT

#6
Avoid Cartesian Products in JOINs

WHAT IT IS ?

Prevent accidental cross joins that result in all possible row combinations.

WHY IT MATTERS ?

Cross joins without conditions can return huge datasets, significantly slowing performance and yielding irrelevant data.

HOW TO OPTIMISE ?

Include an ON condition with your joins to avoid unintentional Cartesian products.

LET'S WORK WITH AN EXAMPLE

BEFORE IMPLEMENTATION

```
SELECT customers.name, orders.order_id
FROM customers, orders;
```

AFTER IMPLEMENTATION

```
SELECT customers.name, orders.order_id
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id;
```

Varun Sagar Theegala

NEXT

#7

Use Aliases in JOINs for Clarity

WHAT IT IS ?

Use clear aliases for table names in joins.

WHY IT MATTERS ?

Meaningful aliases improve readability, especially in complex queries with multiple tables, making it easier to understand and maintain.

HOW TO OPTIMISE ?

Assign meaningful aliases to your table names, avoiding single-letter aliases.

LET'S WORK WITH AN EXAMPLE

BEFORE IMPLEMENTATION

```sql
SELECT c.name, o.order_id, p.product_name
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
JOIN products p ON o.product_id = p.product_id;
```

AFTER IMPLEMENTATION

```sql
SELECT customer.name, order.order_id, product.product_name
FROM customers AS customer
JOIN orders AS order ON customer.customer_id = order.customer_id
JOIN products AS product ON order.product_id = product.product_id;
```

Varun Sagar Theegala

NEXT

#8

Avoid Subqueries in SELECT Clauses

WHAT IT IS ?

Avoid use subqueries for column or case-whens inside the SELECT statement.

WHY IT MATTERS ?

Subqueries in SELECT can increase complexity and execution time, making the query harder to read and less efficient.

HOW TO OPTIMISE ?

Use joins or Common Table Expressions (CTEs) to refactor subqueries in the SELECT clause.

Varun Sagar Theegala

NEXT

LET'S WORK WITH AN EXAMPLE

BEFORE IMPLEMENTATION

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

AFTER IMPLEMENTATION

```
WITH OrderCounts AS (
    SELECT customer_id,
           COUNT(*) AS order_count
    FROM orders
    GROUP BY customer_id
)
SELECT customers.customer_name,
       OrderCounts.order_count
FROM customers
LEFT JOIN OrderCounts
ON customers.customer_id = OrderCounts.customer_id;
```

Varun Sagar
Theegala

NEXT

#9

Be Mindful of NULLs in Joins

WHAT IT IS ?

Handle NULL values carefully when joining to pul fields from other tables

WHY IT MATTERS ?

NULL values can lead to unexpected results in joins, causing issues in filtering, calculations, and data interpretation.

HOW TO OPTIMISE ?

Use COALESCE() or ISNULL() to replace NULLs in your result set.

LET'S WORK WITH AN EXAMPLE

BEFORE IMPLEMENTATION

```sql
SELECT customers.name, orders.order_total
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id;
```

AFTER IMPLEMENTATION

```sql
SELECT customers.name, COALESCE(orders.order_total, 0) AS order_total
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id;
```

Varun Sagar Theegala

#10
Optimize Performance by Indexed JOIN fields

WHAT IT IS ?

Index columns that are frequently used in join conditions.

WHY IT MATTERS ?

Indexes significantly improve join performance by speeding up the retrieval of matched rows, especially in large datasets.

HOW TO OPTIMISE ?

Add indexes to commonly joined columns, ensuring faster query execution.

LET'S WORK WITH AN EXAMPLE

BEFORE IMPLEMENTATION

```sql
SELECT customers.name, orders.order_id
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id;
```

AFTER IMPLEMENTATION

```sql
CREATE INDEX idx_customer_id ON customers(customer_id);

SELECT customers.name, orders.order_id
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id;
```

TL;DR:

1. Use INNER JOIN for matching rows.
2. Use LEFT JOIN to preserve unmatched rows.
3. Avoid subqueries when JOIN suffices.
4. Use LATERAL for row-wise operations.
5. Use RIGHT JOIN sparingly; prefer LEFT.
6. Avoid Cartesian products with ON conditions.
7. Use meaningful aliases for table names.
8. Minimize subqueries in SELECT clauses.
9. Handle NULLs carefully in JOINs.
10. Index JOIN columns for performance.

REMEMBER

Efficiently handling JOINs and subqueries ensures your SQL queries **remain fast, scalable, and easy to read**.

By **using the correct type of JOIN and avoiding subquery pitfalls**, you'll write better SQL code that **enhances performance and reduces resource consumption.**

Varun Sagar Theegala

SHARE THIS

If you think **your network** would find this **valuable**

FOLLOW ME

I help you **GROW & SUSTAIN** as a **Data Analyst**