**Advanced SQL Tips and Tricks for Data Analysts**

Advanced tricks that will save you time and improve your code's performance

Basic SQL can be pretty straightforward. Doing SQL well and properly for advanced use cases is not that easy, though.

Here are a few more advanced SQL tips and tricks that will help you do more complex queries, or just do the basic ones better:

**Common Table Expressions (CTEs)**

CTEs allow you to break down complex queries into smaller, more manageable parts by creating intermediate results that can be reused multiple times within a single query.

You can use them to replace nested queries, so instead of this:

SELECT \*  
FROM  
(  
 SELECT \*  
 FROM users  
 WHERE sex = 'M'  
) men  
INNER JOIN  
(  
 SELECT \*  
 FROM orders  
 WHERE total>100  
) big\_orders  
ON men.id = big\_orders.user\_id

you can do this:

WITH men AS (  
 SELECT \*  
 FROM users  
 WHERE sex = 'M'  
),  
  
big\_orders AS (  
 SELECT \*  
 FROM orders  
 WHERE total>100  
)  
  
SELECT \*  
FROM men  
INNER JOIN big\_orders ON men.id = big\_orders.user\_id

This serves two purposes: making your query run faster and your code easier to read.

**Window Functions**

Window functions allow you to perform calculations across rows in a query result set, without the need for self-joins or subqueries.

SELECT customer\_id, amount,   
 SUM(amount) OVER (PARTITION BY customer\_id ORDER BY order\_date) AS running\_total  
FROM orders

In the above example, we have one row per order, and for each order we have the amount spent by the respective customer *so far*.

**Pivot Tables**

Pivot tables allow you to transform rows into columns, making it easier to analyze data in certain situations.

Imagine you have the following table:

+--------+----------+-------+  
| region | year | sales |  
+--------+----------+-------+  
| North | 2010 | 100 |  
| North | 2011 | 200 |  
| South | 2010 | 150 |  
| South | 2011 | 75 |  
| East | 2010 | 50 |  
+--------+----------+-------+

And you wanted to pivot it into the table below:

+--------+----------+----------+  
| region | 2010 | 2011 |  
+--------+----------+----------+  
| East | 50 | NULL |  
| North | 100 | 200 |  
| South | 150 | 75 |  
+--------+----------+----------+

You could do this with the following query:

WITH pivot\_data AS (  
 SELECT region, year, sales  
 FROM sales\_data  
)  
SELECT \*  
FROM pivot\_data  
PIVOT (SUM(sales) FOR year IN ([2010], [2011])) AS pivot\_table

They work kind of like Excel pivot tables but, as you can see, the process can be a bit manual if you have multiple different values in the columns ("year" in this case).

**Recursive Queries**

Recursive queries allow you to query hierarchical or tree-like data structures, such as organizational charts or file systems.

Imagine an employees table that shows a hierarchy:

+-----------+------------+  
| employee\_id | manager\_id |  
+-----------+------------+  
| 1 | NULL |  
| 2 | 1 |  
| 3 | 2 |  
| 4 | 2 |  
| 5 | 3 |  
+-----------+------------+

If you need to know the hierarchical level of each employee, you could use the following code:

WITH RECURSIVE employee\_hierarchy (employee\_id, manager\_id, level) AS (  
 SELECT employee\_id, manager\_id, 1  
 FROM employees  
 WHERE manager\_id IS NULL  
 UNION ALL  
 SELECT e.employee\_id, e.manager\_id, eh.level + 1  
 FROM employees e  
 JOIN employee\_hierarchy eh ON e.manager\_id = eh.employee\_id  
)  
SELECT \*  
FROM employee\_hierarchy

Which would return the following result:

+-----------+------------+-------+  
| employee\_id | manager\_id | level |  
+-----------+------------+-------+  
| 1 | NULL | 1 |  
| 2 | 1 | 2 |  
| 3 | 2 | 3 |  
| 4 | 2 | 3 |  
| 5 | 3 | 4 |  
+-----------+------------+-------+

From which you could build correspondences such as 1: CEO, 2: Director, 3: Manager, etc.

**Dynamic SQL**

Dynamic SQL allows you to generate and execute SQL statements dynamically at runtime, based on user inputs or other conditions.

DECLARE @sql NVARCHAR(MAX);  
DECLARE @table\_name NVARCHAR(100) = ‘employees’;  
DECLARE @column\_name NVARCHAR(100) = ‘salary’;  
  
SET @sql = 'SELECT AVG(' + @column\_name + ') FROM ' + @table\_name + ';'  
  
EXEC sp\_executesql @sql

This can be particularly useful if you need to try multiple values for a query and this value repeats itself multiple times in the query. The availability and syntax of this feature will depend on the SQL version you use.

**Materialized Views**

Materialized views allow you to pre-calculate and store the results of a query, so that subsequent queries can access the results more quickly. This can be useful for improving performance when working with large or slow-performing data sets.

CREATE MATERIALIZED VIEW sales\_summary  
AS  
SELECT product, SUM(sales) AS total\_sales  
FROM sales\_data  
GROUP BY product;  
  
SELECT product, total\_sales  
FROM sales\_summary

You might be wondering what is the difference between a materialized view and a "standard" view. They are two different ways to encapsulate and reuse complex query logic in a database. The key difference between them is that a materialized view is a pre-calculated and stored result set, while a standard view is simply a stored query definition.

When you query a standard view, the database management system (DBMS) will execute the underlying query each time you run the view, so the result set will always reflect the current state of the data.

When you query a materialized view, on the other hand, the DBMS will return the stored result set, rather than re-running the underlying query. Because the result set is pre-calculated and stored, querying a materialized view can be much faster than querying a standard view, especially if the underlying data is large or slow to query.

In general, you should use a materialized view if you have a large or slow-performing query that you need to execute frequently, and you can tolerate some level of stale data. On the other hand, if you need to have the most up-to-date data, and query performance is less of a concern, you may want to use a standard view.

**Unnest + Generate\_array**

Suppose you want to generate a list of all possible discounts between 10% and 50%, in steps of 5%. Here's how you could do it:

SELECT ROUND(discount / 100, 2) AS discount  
FROM UNNEST( GENERATE\_ARRAY(10, 50, 5) ) AS discount

This can be useful to insert into a cross join with a products table, to create all possible discounts for a given product.

I’m not sure if this method has a name, so I just call it “UNNEST + GENERATE\_ARRAY” (let me know in the comments if you happen to know its proper name). Also, I use it in Google BigQuery, and it might not work the same way in every environment, so you need to test if the tool you use accepts arrays.

If you like what you read, consider joining Medium and reading many more articles. A portion of your fee goes to support authors like me. [Click here](https://medium.com/@arthurmello_/membership) to join.

If you would like to discuss further, feel free to reach out to me on other platforms, it would be a pleasure (honestly):