**How to Take Your SQL from Zero to Data Scientist Level — Part 3/3**

**Elevate your SQL coding skills to the next level with these three simple techniques**

**Intro**

Welcome to the last installment of the series. In the previous two parts, we set up a solid foundation in basic and advanced SQL queries and executed them using **SSMS** and the demo database **AdventureWorks**. In this final part, we will cover the set of skills that will take your SQL to the next level. These are equally important, if not more so, than simply knowing how to write in SQL, as they are the ones that will turn your queries into an accurate, easy-to-read reproducible asset!

**Part 3**

* QA testing your SQL queries
* Optimization of SQL queries
* Data science coding standards and best practices
* Plan for training

**👇 Links to Parts 1 & 2**

**1. QA Testing Your SQL Queries**

A common misconception that I also held strongly at the beginning of my career is that good data scientists have such high proficiency in SQL coding that they never make mistakes. In reality, everyone makes mistakes. Even surgeons and pilots are prone to making errors. The error rate will be different depending on experience, but in the end, everyone has an off day or days.

Fortunately, there are countermeasures to minimize such errors. They are called **checklists**! This is not a new idea nor unique to data scientists. In airlines, each pilot performs the same set of essential tests before the plane departs or when it lands. Similarly, there are steps that you can take to test that your SQL query does not return incorrect results (i.e., is safe for departure). I have highlighted the three main ones below.

**1.1 Uniqueness of Results**

The first item in the SQL checklist is the uniqueness of your results. In most of your queries, as a data scientist, you will have to join multiple tables to get the desired result. When the relationship between two tables is one-to-many, the joined table will have duplicates. As such, in every query you run, ensure you have checked the level (i.e., column) at which your resulting joined table is unique.

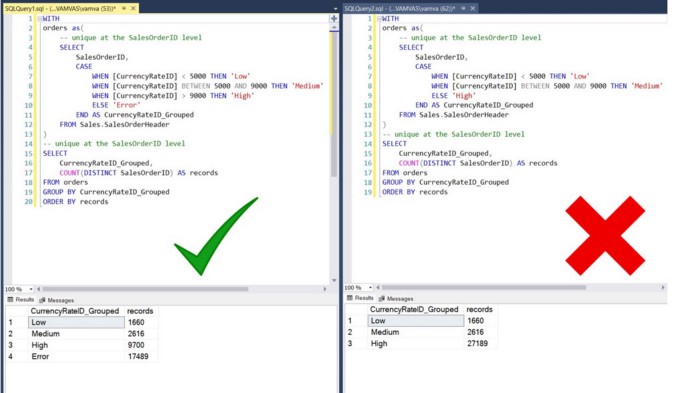
WITH  
orders as(  
 -- unique at the SalesOrderID level  
 SELECT \*  
 FROM Sales.SalesOrderHeader  
),  
orders\_detail as (  
 -- unique at the SalesOrderDetailID level  
 SELECT \*  
 FROM Sales.SalesOrderDetail  
)  
-- unique at the SalesOrderDetailID level  
SELECT \*  
/\* SELECT COUNT(\*), COUNT(DISTINCT SalesOrderDetailID)\*/  
FROM orders o  
INNER JOIN orders\_detail ol ON o.SalesOrderID = ol.SalesOrderID

In the example above you can see I commented out row 14, which I used to check the level of my results. This simple query compares the result of COUNT(\*) with the COUNT(DISTINCT *column*), and if they match, it verifies that the table is unique at that level. As you can see, I added a comment indicating the uniqueness of the tables in the result and the CTEs. That practice will also make it easier to join the tables at the right level!

**1.2 Calculated Columns**

One of the trickiest parts of your queries is calculated columns. These are prone to both logical and computational errors. Sometimes the issue might be that the column has NULL values or that the ranking function you selected is not working as intended. Other times, some of your conditions might overlap in a CASE statement, leading to incorrect results.

* Spot-check the results (especially for tricky cases) using a couple of randomly picked records
* Always use the ELSE clause in the CASE statement to flag unmatched records



Side-by-side queries using the ELSE clause correctly and incorrectly [Image by the author]



Example of NULL records for CurrencyRateID column [Image by the author]

In the example above, the issue is that there were NULL values in the data for the CurrencyRateID column. These were incorrectly added to the ‘High’ group in the right panel query but correctly flagged using the ELSE statement for unmatched conditions in the left panel.

**1.3 Comparison with Baseline Numbers**

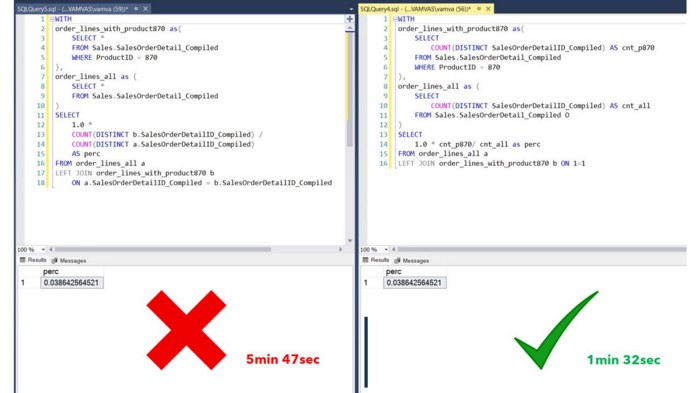
Finally, it is always helpful to compare the top-level results from your query with a company dashboard. This can be as simple as filtering your results for a specific date and comparing them with those in a dashboard (i.e., number of orders, total spend, etc.). This part is especially useful to double-check that you have not omitted a needed sanity filter, like excluding tester accounts or activity from bots.

**2. Optimization of SQL Queries**

For the purposes of this section, I have created a new table by unioning multiple times the SalesOrderHeader table. The new table is called SalesOrderDetail\_Compiled and has 65 million rows.

**2.1 Limit the Number of Records in CTEs**

Even though this seems very simple and straightforward, in my experience training analysts, this is the most frequently omitted step. Looking at the example below, you can see that we had a considerable decrease in runtime by aggregating results within the CTEs and then joining them (right panel) rather than getting all rows, joining, and then aggregating (left panel).

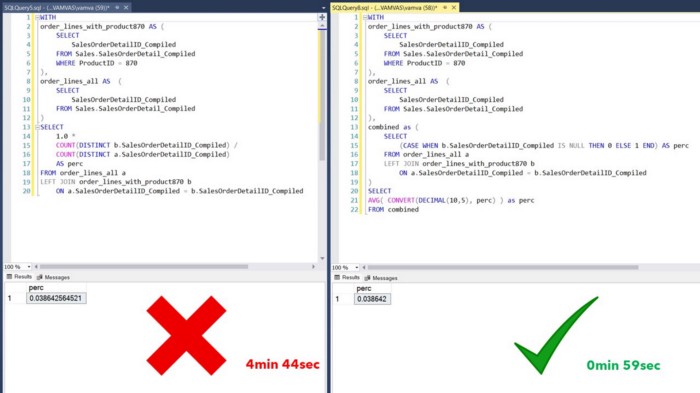


Improving Performance by aggregating first within the CTEs [Image by the author]

**2.2 Avoid Using COUNT DISTINCT**

The DISTINCT keyword is one of the most expensive in SQL. This makes sense as to get distinct results, SQL must scan each row of the column. There are of course cases where COUNT(*column*) and COUNT(DISTINCT *column*) will give you the same result (i.e., when the column is the primary key). In such cases, the obvious recommendation is to omit the DISTINCT keyword.

But what can be done about columns that have duplicates? In the example below, we have created a new binary column in the right panel. The records that did not match in the left join will have NULLs and be labeled as 0, with the non NULL labeled as 1. Then we can take the average of that column. Since we marked as 1 the matching records and 0 the rest, their average will give us the same result as the ratio of the distinct counts in the left panel. This will hopefully become more clear, looking at the snapshot below.

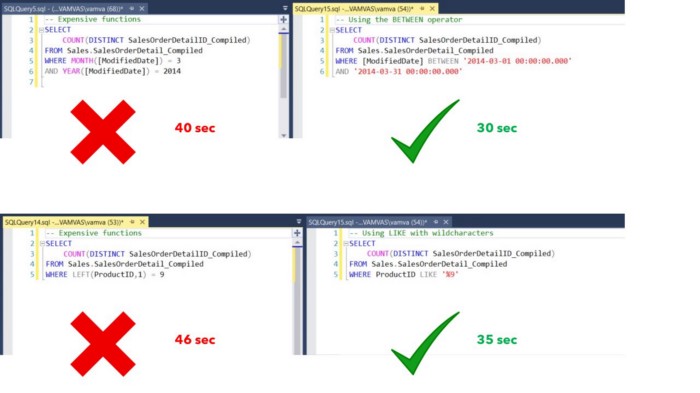


Alternatives to using COUNT DISTINCT [Image by the author]

In SQL Server, you can also use indexes to have certain columns pre-scanned, thus speeding up performance when using the DISTINCT keyword.

**2.3 Functions Wrapped Around JOINs and WHERE Clauses**

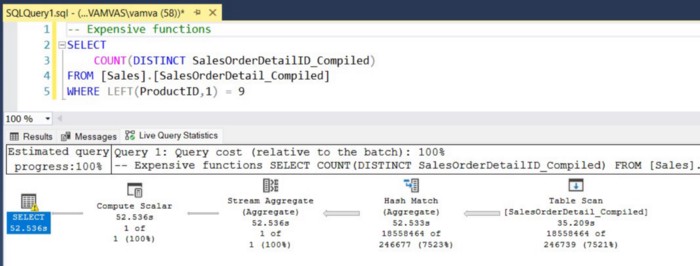
When faced with functions wrapped around columns in the WHERE clause or a JOIN, consider rewriting the query so that columns can be left clean. The reason why this increases processing time is that SQL has to compute the function for every value of the column and then filter the results.



Comparing performance with functions wrapped and not wrapped in columns [Image by the author]

**2.4 Use the Live Query Statistics in SSMS**

SSMS also provides a very nice tool, the “Live Query Statistics”. You can find it on the toolbar, and if selected, it will create a new tab in your results that will showcase your query live while it is executing and will give you statistics for each node to identify the bottlenecks in your code.



SSMS Live Query Statistics [Image by the author]

**3. Data Science Coding Standards and Best Practices**

Every time you write a piece of SQL code that is saved in your codebase (along with other SQL scripts), you are either creating an asset or a liability. An asset is a script that can be easily understood and rerun. In contrast, a liability will be a script that takes too much time to understand; you can’t wait to finish it and move to something else and sometimes you might even throw it in the bin and start from scratch (as this will be easier than working on the existing script).

This is what reproducible code is all about. It is not a nice to have add-on that we can use whenever there is spare time. It is about writing code in a way that makes it a reproducible asset. An asset that will increase the value of your work both in the present and future; for yourself and other team members as well! So what can we do to create assets and not liabilities?

**3.1 Comments**

Comments are there to help your present and future self (and anyone else that will work on your code) to quickly understand it:

* You can use the double dash to start a single-line comment in most DBMSs
* Comment the level that your CTEs and the main result are unique at
* Add comments to explain complicated calculated fields
* Add comments to explain any logic in your query that might need clarification in the future (filter conditions, etc.)

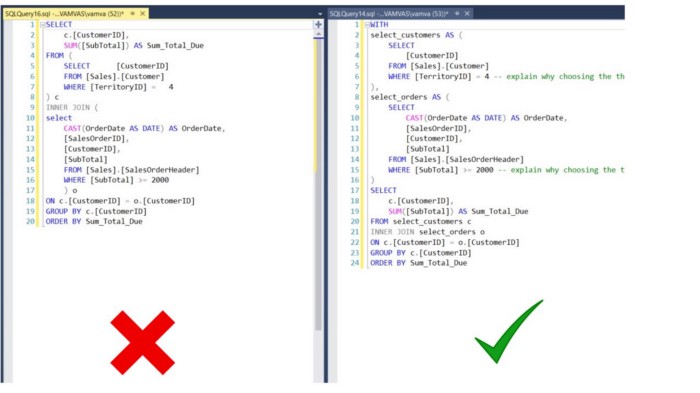
**3.2 Formatting**

In contrast to testing your code, in formatting, there is no set of necessary steps that everyone has to take. The goal is to simplify your code and make it easily read and understood. Any formatting that achieves that purpose is acceptable as long as it is consistent. Below you can see the format I use.

* Use capital for keywords and clauses (WHERE, DISTINCT, etc.)
* Use single tab indentation for columns and SELECT within the CTE
* Use CTEs instead of subqueries
* Use whitespace to separate elements such as clauses, predicates, and expressions in your queries

**3.3 Modularity (CTEs)**

Using CTEs instead of subqueries is also a good step toward improving the reproducibility of your code. The example below shows the code on the left panel (using subqueries, no comments, and no formatting) and its reproducible counterpart on the right panel.



Example of writing reproducible SQL code [Image by the author]

**4. Plan for Training**

And with that, we reached the final section of our three-part series! 🚀🚀

The biggest struggle in my SQL journey was finding what I needed to know and practicing it. I have kept the series self-sustained, as in you will have all the ammunition you need to start practicing afterward. This is not to say that we covered all functions or all clauses and functionalities in SQL but instead that we covered all the needed ones to start your journey. If, on your path, you need to find a different function that we have not covered, I trust that by now, you have a solid foundation to catch that fish on your own. There are also many free resources available online for your ad hoc needs, but my personal favorites are [W3schools](https://www.w3schools.com/sql/default.asp) and [SQLServerTutorial.Net](https://www.sqlservertutorial.net/).

**For your practice sessions, I also recommend the following:**

* Search online for SQL interview questions and try solving them using AdventureWorks2019 and SSMS
* Practice your SQL at least a couple of hours each week for a month or two so that you start thinking about writing in SQL subconsciously

I hope you found this article helpful and have fun querying!

If you have any questions or need further help, please feel free to comment below, and I will answer promptly.