Published in Towards Data Science

Aashish Nair Follow

Aug 15, 2022 · 5 min read · ✨ · Listen

Save

A Look Into SQL's Order Of Execution

This is how SELECT queries are carried out

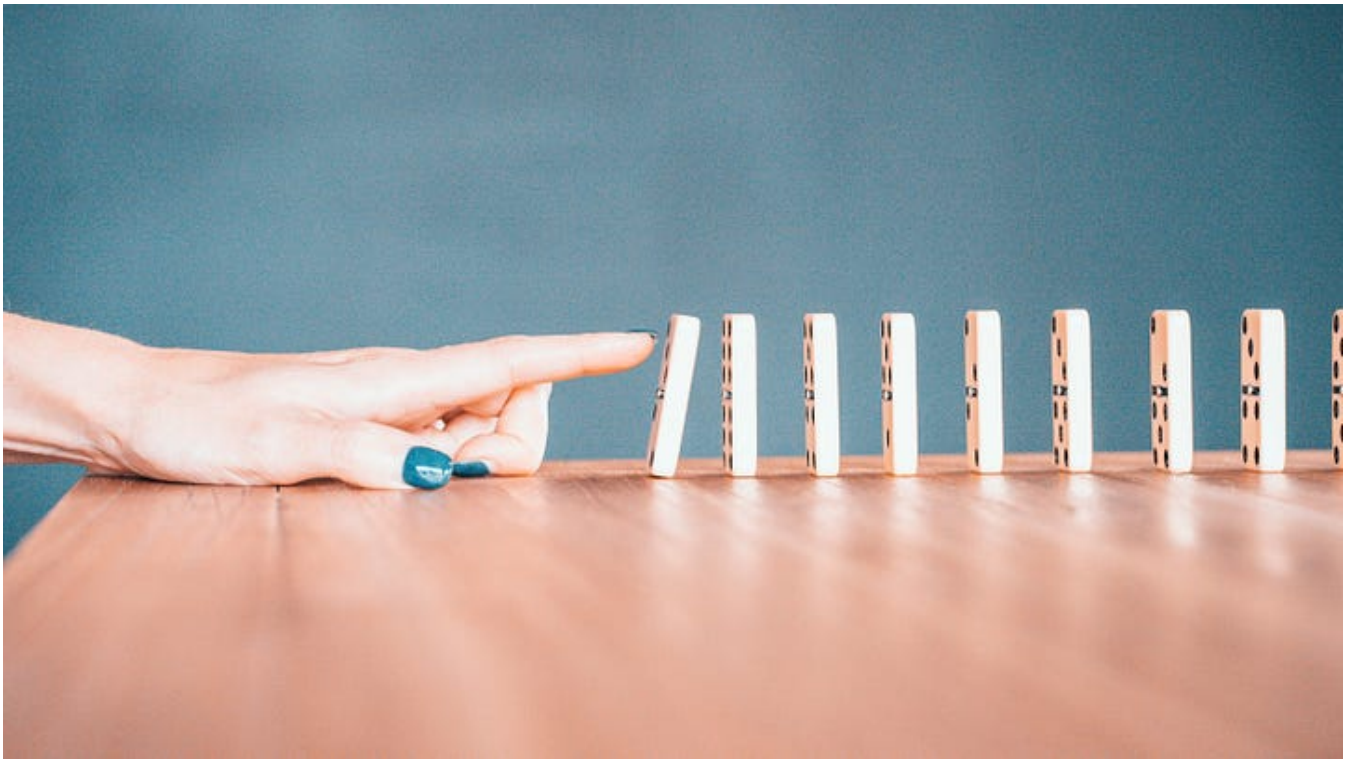

Photo by Bradyn Trollip on Unsplash

After writing a considerable number of SQL scripts, you're likely to reach some form of a plateau in terms of performance. You extract insights using the same strategies and run into the same types of errors.

Fortunately, you can improve your experience with writing queries by taking the time to understand how the clauses in SQL are evaluated.

Here, we discuss the order of execution in SQL and explain why it matters.

Order of Execution

SQL queries adhere to a specific order when evaluating clauses, similar to how mathematical operations adhere to PEMDAS or BIDMAS.

From the eyes of the user, queries begin from the first clause and end at the last clause. However, queries aren't actually read from top to bottom when carried out.

The order in which the clauses in queries are executed is as follows:

1. FROM/JOIN: The FROM and/or JOIN clauses are executed first to determine the data of interest.

2. WHERE: The WHERE clause is executed to filter out records that do not meet the constraints.

3. GROUP BY: The GROUP BY clause is executed to group the data based on the values in one or more columns.

4. HAVING: The HAVING clause is executed to remove the created grouped records that don't meet the constraints.

5. SELECT: The SELECT clause is executed to derive all desired columns and expressions.

6. ORDER BY: The ORDER BY clause is executed to sort the derived values in ascending or descending order.

7. LIMIT/OFFSET: Finally, the LIMIT and/or OFFSET clauses are executed to keep or skip a specified number of rows.

Case Study

To illustrate the order of execution in SQL, it's best to use an example.

We have two tables named Orders and Products, which detail purchases made on stationery items.

	order_id [PK] integer	name character varying (50)	product_id integer	quantity integer
1	1	Timmy	1	2
2	2	Tommy	1	1
3	3	Lonnie	5	3
4	4	Tobbi	4	3
5	5	Bonnie	1	3

Orders Table (Created By Author)

	id [PK] integer	name character varying (50)	price integer
1	1	Pencil	10
2	2	Pen	40
3	3	Ruler	20
4	4	Erasor	30
5	5	Paper	20

Products Table (Created By Author)

Using these tables, we want to find out:

Who has spent the second most amount of money on pens?

We can obtain the answer with the following query.

```sql
-- Find the person that spent the second most amount on pens
SELECT O.name,
        SUM(O.quantity*P.price) AS total_spent
FROM Orders O
JOIN Products P
        ON O.product_id = P.id
WHERE P.name = 'Pen'
GROUP BY O.name
ORDER BY total_spent DESC
LIMIT 1 OFFSET 1;
```

query.sql hosted with ❤ by GitHub view raw

	name character varying (50)	total_spent bigint
1	Lonnie	2120

Code Output (Created By Author)

The answer is successfully outputted, but how is it derived in the first place?

As previously stated, a user might read the query starting from the SELECT clause and ending with the OFFSET clause, but SQL doesn't read clauses from top to bottom.

Instead, it derives the answer with the following steps:

1. The FROM and JOIN clauses merge the Products and Orders tables to obtain the total data of interest.

2. The WHERE clause removes records where pens aren't purchased.

3. The GROUP BY clause groups together records by name.

4. The HAVING clause removes groups that spend more than 2000 on pens.

5. The SELECT clause derives the buyers and their corresponding amount spent on pens.

6. The ORDER BY clause sorts the results based on the total amount spent on pens in descending order.

7. The LIMIT and OFFSET clauses skip the first row and keep only the next record, which contains the person that spent the second most on pens.

Why This Matters

The order of execution in SQL might seem unimportant at first glance. After all, if queries are delivering the desired outputs, who cares how the clauses are evaluated?

Unfortunately, users unfamiliar with the order of execution will struggle with writing queries with greater complexity since any issues that emerge will be harder to debug. Those looking to troubleshoot errors with greater ease will benefit from understanding the order in which the clauses in SQL are read.

For example, a common mistake in SQL is incorrectly referencing column aliases.

We can highlight this mistake by using the Orders and Products tables to answer another question:

Out of those who have spent less than 50, who has spent the most amount of money on an order?

We can calculate the total money spent on an order by multiplying the price variable in the Products table by the quantity variable in the Orders table and naming this expression with the alias "total_spent".

Let's see what answer the following query outputs.

```
ERROR:  column "total_spent" does not exist
LINE 7: WHERE total_spent > 50
```

Code Output (Created By Author)

An error? What's going on?

A user unfamiliar with the order of execution might not see the issue of using the column alias "total_spent" in the WHERE clause, so the query might seem valid at face value.

However, knowing the order of execution in SQL, it is clear that since the column alias is created in the SELECT clause, which is evaluated *after* the WHERE clause,

SQL will not contain that alias when evaluating the WHERE clause, hence the error.

Now that we can identify the source of the error, we can fix it by not using the column alias in the WHERE clause's expression.

	name character varying (50)	total_spent integer
1	Tammy	40
2	Bobbi	40
3	Bonnie	40
4	Bonnie	40
5	Tobbi	40

Preview of Code Output (Created By Author)

The query now runs successfully.

Note that the ORDER BY clause still uses the column alias "total_spent", which is fine since this clause is evaluated *after* the SELECT clause.

In general, issues like these are easy to address for those that understand the order of execution in SQL. However, those oblivious to it will have a hard time fixing errors that seem benign at face value.

Conclusion

Photo by Prateek Katyal on Unsplash

All in all, knowing how queries are read will benefit users looking to improve their scripts and have a more pain-free experience with troubleshooting any errors.

It might seem daunting to remember this order at first, but the order of execution will be easy to learn by heart after just a little experience (like PEMDAS or BIDMAS).

I wish you the best of luck in your data science endeavors!

Data Science Sql