2021-05-07T17:00:00+02:00
Zahin Rahman

What Is the WITH Clause in SQL?

In this article, you will learn about the SQL WITH clause, also known as common table expression (CTE). We will go over some examples to demonstrate some of their use cases in light of their benefits.

Introduction to the SQL WITH Clause

The WITH clause in SQL was introduced in standard SQL to simplify complex long queries, especially those with `JOINs` and subqueries. Often interchangeably called CTE or subquery refactoring, a `WITH` clause defines a temporary data set whose output is available to be referenced in subsequent queries.

The best way to learn the `WITH` clause in SQL is through practice. I recommend LearnSQL.com's interactive Recursive Queries course. It contains over 100 exercises that teach the `WITH` clause starting with the basics and progressing to advanced topics like recursive `WITH` queries.

The `WITH` clause is considered "temporary" because the result is not permanently stored anywhere in the database schema. It acts as a temporary view that only exists for the duration of the query, that is, it is only available during the execution scope of `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `MERGE` statements. It is only valid in the query to which it belongs, making it possible to improve the structure of a statement without polluting the global namespace.

The `WITH` clause is used in queries in which a derived table is unsuitable. Therefore, it is considered a neater alternative to temp tables. Put simply, the key advantage of the `WITH` clause is that it helps organize and simplify long and complex hierarchical queries by breaking them down into smaller, more readable chunks.

Learn how to process trees and graphs in SQL with our Recursive Queries course. The ultimate SQL challenge!

The `WITH` clause was introduced in the SQL standard first in 1999 and is now available in all major RDBMS. Some common applications of SQL CTE include:

- Referencing a temporary table multiple times in a single query.
- Performing multi-level aggregations, such as finding the average of maximums.
- Performing an identical calculation multiple times over within the context of a larger query.
- Using it as an alternative to creating a view in the database.

OrderDetailID OrderID ProductID Quantity

1	10248	11	12
2	10248	42	10
3	10248	72	5
4	10249	14	9
5	10249	51	40
...	...	...	...
518	10443	28	12

Let's see a quick and simple example of the WITH clause below using the **OrderDetails** table from the well-known Northwind database. The objective is to return the average quantity ordered per ProductID:

QUERY:

```
WITH cte_quantity
AS
(SELECT
    SUM(Quantity) as Total
FROM OrderDetails
GROUP BY ProductID)

SELECT
    AVG(Total) average_product_quantity
FROM cte_quantity;
```

RESULT:

Number of Records: 1

average_product_quantity
165.493

If you were to execute it without the WITH clause and use a subquery instead, the query would look something like this:

QUERY:

```
SELECT
    AVG(Total) average_product_quantity
FROM
(SELECT
SUM(Quantity) as Total
FROM OrderDetails
GROUP BY ProductID)
```

Although you may not see a lot of tangible differences between the two, a broken-down structure that a WITH clause facilitates will be invaluable as your queries scale up in size and hierarchy. We will see an example of this below in the form of a nested WITH clause. You can find more examples in one of our previous articles on the topic – CTEs Explained with Examples.

The WITH Clause Syntax

The general sequence of steps to execute a `WITH` clause is:

1. Initiate the `WITH`
2. Specify the expression name for the to-be-defined query.
3. *Optional*: Specify column names separated by commas.
4. After assigning the name of the expression, enter the `AS` command. The expressions, in this case, are the named result sets that you will use later in the main query to refer to the CTE.
5. Write the query required to produce the desired temporary data set.
6. If working with more than one CTEs or WITH clauses, initiate each subsequent one separated by a comma and repeat steps 2-4. Such an arrangement is also called a nested `WITH` clause.
7. Reference the expressions defined above in a subsequent query using `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `MERGE`

The syntax for implementing a `WITH` clause is shown in the pseudo-code below:

```
WITH expression_name_1 (column_1, column_2,…,column_n)
AS
(CTE query definition 1),
expression_name_2 (column_1, column_2,…,column_n)
AS
(CTE query definition 2)

SELECT expression_A, expression_B, ...
FROM expression_name_2
```

The `WITH` clause is a drop-in replacement to normal subqueries. The only difference is that you can re-use the same derived result set multiple times in your code when you use the `WITH` clause to generate a CTE. You cannot do the same with subqueries.

As we see above, the key execution parameters for a `WITH` clause are:

> `WITH`: Used to create a CTE, or the temporary data set(s).
>
> **`expression_name`** `(column_1, …, column_n)`: The name of the virtual temporary data set which will be used in the main query, and `column_1` to `column_n` are the column names that can be used in subsequent query steps.
>
> `AS` (….): This section defines the query that will populate the CTE **`expression_name`**. If implementing a nested CTE, the query within the second AS will likely refer to the first CTE.
>
> `SELECT expression_A, expression_B FROM expression_name`: This section specifies the main outer query where the `SELECT` statement (or `INSERT`, `UPDATE`, `DELETE`, or `MERGE` statements) is used on one or more of the generated CTEs to subsequently output the intended result.

All of the parameters mentioned above are mandatory. You may choose to use `WHERE`, `GROUP BY`, `ORDER BY`, and/or `HAVING` clauses as required.

When a query with a `WITH` clause is executed, first, the query mentioned within the clause is evaluated and the output of this evaluation is stored within a temporary relation. Then, the main query associated with the `WITH` clause is finally executed using the temporary relation produced.

This example will demonstrate a nested `WITH` clause using the same **`OrderDetails`** table as above. A nested `WITH` clause, or nested CTEs, involve two CTEs within the same query, the second one referencing the first.

OBJECTIVE: To return the average number of orders, or sales made, by `EmployeeID` for `ShipperID` 2 and `ShipperID` 3.

QUERY:

```
WITH cte_sales
AS
(SELECT
    EmployeeID,
    COUNT(OrderID) as Orders,
    ShipperID
FROM Orders
GROUP BY EmployeeID, ShipperID),

shipper_cte
AS
(SELECT *
FROM cte_sales
WHERE ShipperID=2 or ShipperID=3)

SELECT
    ShipperID, AVG(Orders) average_order_per_employee
FROM
shipper_cte
GROUP BY ShipperID;
```

RESULT:

Number of Records: 2

ShipperID	average_order_per_employee
2	9.25
3	7.555555555555555

Here, we calculate the average number of orders per employee but only for `ShipperID` 2 and `ShipperID` 3. In the first CTE, **`cte_sales`**, the number of orders are counted and grouped by `EmployeeID` and `ShipperID`. In the second CTE, **`shipper_cte`**, we refer to the first CTE and define the `ShipperID` conditions using a `WHERE` clause. Then in the main query, we only refer to the second CTE, **`shipper_cte`**, to calculate the average orders per employee by `ShipperID`.

Further nuances of the syntax associated with SQL `WITH` clauses and CTEs are detailed out in Module #2 of the Recursive Queries course, which also contains a collection of more advanced walkthrough examples.

Use Cases of the SQL WITH Clause

So, when do you really need to use a WITH Clause? Well, there are a few unique use cases. Most of them are geared towards convenience and ease of query development and maintenance.

The standout applications and associated benefits of SQL CTEs can be summarized as:

Improves Code Readability– Literate programming is an approach introduced by Donald Kuth, which aims to arrange source code in the order of human logic such that it can be understood with minimal effort by reading it like a novel in a sequential manner. The SQL `WITH` clause helps do just that by creating virtual named tables and breaking large computations into smaller parts. They can then be combined later in the query in the final `SELECT`, or another statement, instead of lumping it all into one large chunk.

Improves Code Maintainability – Going hand in hand with readability is maintainability. As your queries and databases scale up with time, there will always be the need for debugging and troubleshooting – an easier to read code is easier to maintain!

Alternative to a View– CTEs can substitute for views and can `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `MERGE` This can be particularly useful if you do not have the system rights to create a view object or if you don't want to create a view just to be used in a single query.

Overcome Statement Limitations– CTEs help overcome constraints such as `SELECT` statement limitations, for example, performing a `GROUP BY` using non-deterministic functions.

Processing Hierarchical Structures– This is one of the more advanced applications of the CTE and is accomplished through what is known as recursive CTEs. Recursive queries can call on themselves, allowing you to traverse complex hierarchical models. More on this below.