# 5 Practical SQL CTE Demos

*Common table expressions (CTEs) were introduced into SQL to improve the readability and the structure of SQL queries, especially those requiring multiple steps to get the necessary output. In this article, we will go through several Demos to show how SQL CTEs can help you with complex calculations and hierarchical data structures.*

*Do note: Demos of interest are [1-3] … Demos [4-5] are still incomplete and not required for YR2.*

# Common Table Expressions in SQL

Common table expressions (CTEs), also called WITH clauses, allow **creating named subqueries that are further referenced in the main query**. CTEs were introduced in SQL to improve the readability and the structure of an SQL statement.

The basic CTE syntax is as follows:

|  |
| --- |
| WITH subquery\_name AS  (SELECT … subquery ...)  SELECT … main query ... |

We start with the WITH keyword followed by the name we assign to the CTE (subquery). Then, we put the AS keyword and include the subquery in parentheses. After the CTE is defined, we move on to the main query, where we can reference this CTE by its name.

It is possible to have multiple CTEs in one query, reference one CTE within another (i.e., nested CTEs), or even reference a CTE within itself (recursive CTEs). This gives us a whole bunch of tools and opportunities.

# SQL CTE Demos

To show how CTEs can assist you with various analytical tasks, I’ll go through five practical Demos.

We’ll start with the table orders, with some basic information like the order date, the customer ID, the store name, the ID of the employee who registered the order, and the total amount of the order.

In SSMS right Click on Database and create a Database called [CTE-Orders]. You can run the following code and replicate the activities.

USE [CTE-Orders]

GO

/\*\*\*\*\*\* Object: Table [dbo].[orders] Script Date: 09/10/2022 00:01:07 \*\*\*\*\*\*/

CREATE TABLE [dbo].[orders](

[id] [int] NOT NULL,

[date] [date] NULL,

[customer\_id] [real] NULL,

[store] [nchar](10) NULL,

[employee\_id] [int] NULL,

[amount] [smallmoney] NULL,

CONSTRAINT [PK\_orders] PRIMARY KEY CLUSTERED

(

[id] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON, OPTIMIZE\_FOR\_SEQUENTIAL\_KEY = OFF) ON [PRIMARY]

) ON [PRIMARY]

GO

-- Insert test data

USE [CTE-Orders]

GO

INSERT [dbo].[orders] ([id], [date], [customer\_id], [store], [employee\_id], [amount]) VALUES (101, CAST(N'2021-07-01' AS Date), 234, N'Pheonix ', 11, 198.0000)

GO

INSERT [dbo].[orders] ([id], [date], [customer\_id], [store], [employee\_id], [amount]) VALUES (102, CAST(N'2021-07-01' AS Date), 675, N'Pegasus ', 13, 799.0000)

GO

INSERT [dbo].[orders] ([id], [date], [customer\_id], [store], [employee\_id], [amount]) VALUES (103, CAST(N'2021-07-01' AS Date), 456, N'Pegasus ', 14, 698.0000)

GO

INSERT [dbo].[orders] ([id], [date], [customer\_id], [store], [employee\_id], [amount]) VALUES (104, CAST(N'2021-07-01' AS Date), 980, N'Orion ', 15, 99.0000)

GO

INSERT [dbo].[orders] ([id], [date], [customer\_id], [store], [employee\_id], [amount]) VALUES (105, CAST(N'2021-07-02' AS Date), 594, N'Orion ', 16, 1045.4500)

GO

INSERT [dbo].[orders] ([id], [date], [customer\_id], [store], [employee\_id], [amount]) VALUES (106, CAST(N'2021-07-02' AS Date), 435, N'Pegasus ', 11, 599.0000)

GO

INSERT [dbo].[orders] ([id], [date], [customer\_id], [store], [employee\_id], [amount]) VALUES (107, CAST(N'2021-07-02' AS Date), 246, N'Rigel ', 14, 678.8900)

GO

INSERT [dbo].[orders] ([id], [date], [customer\_id], [store], [employee\_id], [amount]) VALUES (108, CAST(N'2021-07-03' AS Date), 256, N'Pheonix ', 12, 458.8000)

GO

INSERT [dbo].[orders] ([id], [date], [customer\_id], [store], [employee\_id], [amount]) VALUES (109, CAST(N'2021-07-03' AS Date), 785, N'Pheonix ', 12, 99.0000)

GO

INSERT [dbo].[orders] ([id], [date], [customer\_id], [store], [employee\_id], [amount]) VALUES (111, CAST(N'2021-07-03' AS Date), 443, N'Orion ', 16, 325.5000)

GO

Now, let’s write a couple of SQL queries!

# CTE Demo 1

In our first Demo, we want to compare the total amount of each order with the average order amount at the corresponding store.

We can start by calculating the average order amount for each store using a CTE and adding this column to the output of the main query:

|  |
| --- |
| use [CTE-Orders]  -- calc average store amount  SELECT store, AVG(amount) AS average\_order  FROM orders  GROUP BY store  -- CTE avg\_per\_store presented on each order.  WITH avg\_per\_store AS  (SELECT store, AVG(amount) AS average\_order  FROM orders  GROUP BY store)  SELECT o.id, o.store, o.amount, avg.average\_order AS avg\_for\_store  FROM orders o  JOIN avg\_per\_store avg  ON o.store = avg.store; |

As you see, our query begins with a CTE called avg\_per\_store. Using this CTE, we create a table that lists all stores and the average order amount by store. Then, in the main query, we select to display the order ID, the store name, the order amount from the original orders table, and the average order amount for each store (avg\_for\_store) from the CTE defined earlier.

Here’s the results:

Table

Description automatically generated

With this table, we can see how each order compares to the average order amount at the corresponding store.

Now, let’s move on to a more complex Demo.

# CTE Demo 2

Here, we’ll compare different stores. Specifically, we want to see how the average order amount for each store compares to the minimum and the maximum of the average order amount among all stores.

As in our first Demo, we’ll start by calculating the average order amount for each store using a CTE. Then, we’ll define two more CTEs:

* To calculate the minimum of the average order amount among all stores.
* To calculate the maximum of the average order amount among all stores.

Note that these two CTEs will use the result of the first CTE.

Finally, in the main query, we’ll join all three CTEs to get the information we need:

|  |
| --- |
|  |

As you see, even with multiple nested CTEs, the SQL query remains clean and easy to follow. If you were to use subqueries, you would need to nest one subquery within the other two and

-- each stores average order and compare to all stores min avg order and all stores max avg order

WITH avg\_per\_store AS (

SELECT store, AVG(amount) AS average\_order

FROM orders

GROUP BY store),

min\_order\_store AS (

SELECT MIN (average\_order) AS min\_avg\_order\_store

FROM avg\_per\_store),

max\_order\_store AS (

SELECT MAX (average\_order) AS max\_avg\_order\_store

FROM avg\_per\_store)

SELECT avg.store, avg.average\_order, min.min\_avg\_order\_store,

max.max\_avg\_order\_store

FROM avg\_per\_store avg

CROSS JOIN min\_order\_store min

CROSS JOIN max\_order\_store max;

repeat it several times within the same query. Here, with CTEs, we simply define all three CTEs at the beginning then reference them when needed.

Here’s the output of this query:

Table

Description automatically generated

You can easily see how each store compares with others in terms of the average order amount.

Of course, when you have only three stores, we could just compare them without adding the min\_avg\_order\_store and max\_avg\_order\_store columns. However, when you need to analyze the performance of many stores by different metrics, this approach might be very helpful.

# CTE Demo 3

In our next Demo, we’ll continue with comparing the performance of our stores but with a few different metrics. Let’s say our company considers orders below $200 to be small and orders equal or above $200 to be big. Now, we want to calculate how many big orders and small orders each store had.

To address this task using WITH clauses, we need two common table expressions:

* To get the number of big orders for each store.
* To get the number of small orders for each store.

Some stores may not have any big orders or any small orders, leading to NULL values. We need to make sure we don’t lose any stores during JOINs. For this reason, I prefer to have yet another CTE that simply outputs a list of all stores. Then, in the main query, we’ll join this CTE with the two CTEs containing the metrics on big and small orders:

|  |
| --- |
| WITH stores AS  (SELECT store  FROM orders  GROUP BY store),  big AS  (SELECT store, COUNT(\*) AS big\_orders  FROM orders  WHERE amount >= 200.00  GROUP BY store),  small AS  (SELECT store, COUNT(\*) AS small\_orders  FROM orders  WHERE amount < 200.00  GROUP BY store)  SELECT s.store, b.big\_orders, sm.small\_orders  FROM stores s  FULL JOIN big b  ON s.store = b.store  FULL JOIN small sm  ON s.store = sm.store; |

So, in this query, we:

* Define the CTE stores to get a full list of stores.
* Define the CTE big to calculate, for each store, the number of orders with the total amount equal to or above $200.
* Define the CTE small to calculate, for each store, the number of orders below $200.
* Join all three CTEs.

Here’s the output:

Graphical user interface, application, table

Description automatically generated

We can now see the West store performs really well; all of its orders are above $200. The Center store is also good, with two orders above $200 and one order below $200. Only half of the orders at the East store are big, with two orders above $200 and two orders below $200.

Optional and still working on demos:

# CTE Demo 4

For the next two Demos, we’ll use the table below with some basic information about the employees of our company. Specifically, we have the employee ID, the first name, the last name, the ID of the employee’s superior, the department, and the last bonus amount.

-- Test table for Demo 4

USE [HumanResources]

GO;

CREATE TABLE [dbo].[MyEmployees](

[id] [int] NOT NULL,

[first\_name] [nvarchar](50) NOT NULL,

[last\_name] [nvarchar](50) NOT NULL,

[superior\_id] [int] NULL,

[department] [nvarchar](50) NOT NULL,

[bonus] [int] NOT NULL

) ON [PRIMARY]

GO

-- test data for CTE Demo

USE [HumanResources]

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (1, N'John', N'Davies', 0, N'CEO', 2545)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (2, N'Mark', N'Taylor', 1, N'Finance', 1100)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (3, N'Kate', N'Wilson', 1, N'Operations', 900)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (4, N'Olivia', N'Watson', 3, N'Operations', 450)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (5, N'James', N'Addington', 1, N'Sales', 1900)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (6, N'Rachael', N'White', 1, N'Marketing', 1250)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (7, N'Sara', N'Clinton', 6, N'Marketing', 1000)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (11, N'John', N'Smith', 5, N'Sales', 800)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (12, N'Noah', N'Jones', 11, N'Sales', 500)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (13, N'Steven', N'Brown', 5, N'Sales', 900)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (14, N'Liam', N'Williams', 13, N'Sales', 700)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (15, N'Paul', N'Lee', 5, N'Sales', 500)

GO

INSERT [dbo].[MyEmployees] ([id], [first\_name], [last\_name], [superior\_id], [department], [bonus]) VALUES (16, N'Patrick', N'Evans', 15, N'Sales', 500)

GO

Now, let’s calculate the average bonus by department, then count how many employees had bonuses above their respective department average and how many had below.

Common table expressions can be very handy with such complex calculations. We’ll have three CTEs in this SQL query:

* To calculate the average bonus amount for each department.
* To calculate, by department, the number of employees whose bonuses were **above their respective department average**.
* To calculate, by department, the number of employees whose bonuses were **below their respective department average**.

In the main query, we’ll join all three CTEs.

|  |
| --- |
| ----  --Now, let’s calculate the average bonus by department, then count how many employees had bonuses above their  -- respective department average and how many had below. Common table expressions can be very handy with such  -- complex calculations.  -- We have three CTEs in this SQL query:  --• To calculate the average bonus amount for each department.  --• To calculate, by department, the number of employees whose bonuses were above their respective department average.  --• To calculate, by department, the number of employees whose bonuses were below their respective department average.  --In the main query, we’ll join all three CTEs.  USE [HumanResources]  GO  WITH avg\_bonus\_department AS  -- 1. department and average bonus  (SELECT department, AVG(bonus) AS average\_bonus  FROM MyEmployees  GROUP BY department),  above\_average AS  (-- 2. employees with bonus greater than the department average  SELECT e.department, count(\*) AS employees\_above\_average  FROM MyEmployees e  JOIN avg\_bonus\_department avg  ON e.department = avg.department  WHERE bonus > average\_bonus  GROUP BY e.department),  -- 3. employees with bonus below the department average  below\_average AS  (SELECT e.department, count(\*) AS employees\_below\_average  FROM MyEmployees e  JOIN avg\_bonus\_department avg  ON e.department = avg.department  WHERE bonus < average\_bonus  GROUP BY e.department)  SELECT avg.department, avg.average\_bonus, aa.employees\_above\_average, ba.employees\_below\_average  FROM avg\_bonus\_department avg  LEFT JOIN above\_average aa  ON avg.department = aa.department  LEFT JOIN below\_average ba  ON avg.department = ba.department; |

Here’s the result of the query:

Graphical user interface, application, Teams

Description automatically generated

Since there is only one person in Finance, the average bonus for the department is exactly equal to the bonus of this person. As a result, we have nobody in the Finance department whose bonus was either above or below average (reflected as NULL values in the result). The same applies to the CEO.

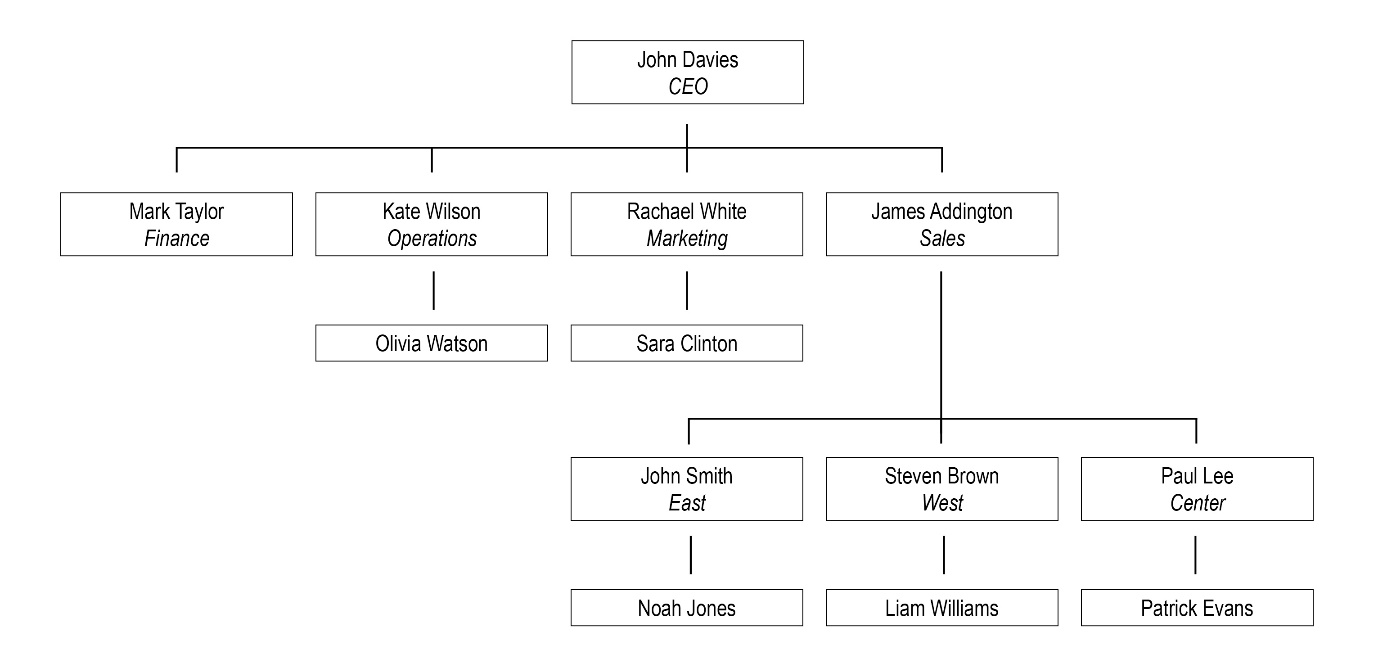
For the Sales department, we can see that the average bonus was $828.57, and only two out of seven people had bonuses above the department average.

We will leave you to interpret the results for the Marketing and Operations departments in the same way, and we will move on to an even more complex Demo with a recursive query.

# CTE Demo 5 – incomplete

**Common table expressions can reference themselves**, making them a perfect tool for analyzing hierarchical structures. Let’s see with an Demo.

Using the information from the employees table and the orders table, we can draw the following organizational structure of our company. The store personnel are considered part of the Sales team. In addition, in the orders table, we can see which employees have orders in which stores, so we can derive the store to which each salesperson belongs.



Now, let’s say we need to find out the level of each employee in the organizational structure (i.e., level 1 is the CEO, level 2 is for his direct reports, etc.). We can add a column that shows this with a recursive query:

|  |
| --- |
| -- MN: could not get this working and will come back to it!  WITH RECURSIVE mylevels AS (  SELECT  id,  first\_name,  last\_name,  superior\_id,  1 AS level  FROM MyEmployees  WHERE superior\_id IS NULL  UNION ALL  SELECT  MyEmployees.id,  MyEmployees.first\_name,  MyEmployees.last\_name,  MyEmployees.superior\_id,  mylevels.level + 1  FROM MyEmployees, mylevels  WHERE MyEmployees.superior\_id = mylevels.id  )    SELECT \*  FROM mylevels; |

As you see, the CTE levels in this query references itself. It starts with selecting the record corresponding to the big boss, the one who doesn’t have a superior (i.e., superior\_id IS NULL). We assign 1 to the level of this person, then use UNION ALL to add other records, adding one to it for each level of management in the organizational structure.

Here’s the output we are supposed to to obtain:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **first\_name** | **last\_name** | **superior\_id** | **level** |
| 1 | John | Davies | NULL | 1 |
| 2 | Mark | Taylor | 1 | 2 |
| 3 | Kate | Wilson | 1 | 2 |
| 5 | James | Addington | 1 | 2 |
| 6 | Rachael | White | 1 | 2 |
| 4 | Olivia | Watson | 3 | 3 |
| 7 | Sara | Clinton | 6 | 3 |
| 11 | John | Smith | 5 | 3 |
| 13 | Steven | Brown | 5 | 3 |
| 15 | Paul | Lee | 5 | 3 |
| 12 | Noah | Jones | 11 | 4 |
| 14 | Liam | Williams | 13 | 4 |
| 16 | Patrick | Evans | 15 | 4 |