**When Should I Use a Common Table Expression (CTE)?**

CTEs can be very useful, especially if you have already mastered the basics of SQL, such as selecting, ordering, filtering data, and joining tables.

## What Is a Common Table Expression?

A common table expression (CTE) is a relatively new SQL feature. It was introduced in SQL:1999, the fourth SQL revision, with ISO standards issued from 1999 to 2002 for this version of SQL.

CTEs were first introduced in SQL Server in 2005, then PostgreSQL made them available starting with Version 8.4 in 2009. MySQL waited a little bit longer and made them available in 2018, starting with Version 8.0.  Simply put, it’s a temporary data set returned by a query, which is then used by another query. It’s temporary because the result is not stored anywhere; it exists only when the query is run.

There are two types of CTEs:

* non-recursive
* recursive

I’ll discuss only the non-recursive CTE in this article, and only mention the recursive CTE briefly at the end.

The basic syntax of the (non-recursive) CTE is as follows:

|  |
| --- |
| WITH expression\_name AS (CTE definition) |

As you can see, it is done using a WITH statement. For this reason, CTEs are also called WITH queries. After the WITH, you define a CTE in parenthesis. Defining CTE simply means writing a SELECT query which will give you a result you want to use within another query.

SELECT ...

FROM expression\_name

You define your SELECT query and then reference your CTE, using it as you would any other table after the FROM clause.

If you want to read more about CTEs before going on to the examples, here is [an article](https://www.essentialsql.com/introduction-common-table-expressions-ctes/) that explains them nicely.

## The CTE Syntax

Now, let’s see how the CTE syntax works in practice. Suppose there is a database containing various data from the university with the following three tables:

* **students**
* **subjects**
* **exams**

The table **students** has the following columns:

* id: the ID of the student, a primary key
* first\_name: the student’s first name
* last\_name: the student’s last name

The next table is the **subjects** table containing the data:

* id: the ID of the subject, a primary key
* subject\_name: the name of the subject

The third table is the **exams** table that stores the following data:

* id: the ID of the exam given
* exam\_date: the date when the exam was given
* subject\_id: the ID of the subject, a foreign key from the table **subjects**
* student\_id: the ID of the student who took the exam, a foreign key from the table **students**

Your task is to calculate the average grade for the students. Then for every student with the average grade above 8.5, you need to show their first name, last name, and their average grade, and label them as “exceptional” students. How would you do this using a CTE?

The code that will give you the desired result can be written like this:

|  |
| --- |
| WITH grade\_average AS (  SELECT  s.id,          s.first\_name,          s.last\_name,          AVG (e.grade) AS average\_grade  FROM students s JOIN exams e ON s.id = e.student\_id  GROUP BY s.id, s.first\_name, s.last\_name  )    SELECT  first\_name,          last\_name,          average\_grade,          'exceptional' AS tag  FROM grade\_average  WHERE average\_grade>8.5; |

First, you need to define your CTE. As you’ve already learned, this is done using a WITH statement. It is followed by the name of the CTE, which is **grade\_average** in this case. A CTE query is defined in between the parentheses. When you look at it on its own, it’s not complicated; it is a rather regular looking SELECT query. It selects id, first\_name, and the last\_name from the table **students**. It also calculates the average grade, using the column grade from the table **exams**. The result is shown in the new column average\_grade. The tables **students** and **exams** are joined on the appropriate student ID column from each table. The result is grouped by the columns id, first\_name, and last\_name from the table **students**. The records are grouped, since you need to obtain the result by student.

After the CTE is defined, there’s another SELECT query that uses the CTE. This query selects the columns first\_name, last\_name, and average\_grade from the CTE, **grade\_average**. It also assigns the value “exceptional.” There is a WHERE clause at the end to show only those students with an average grade above 8.5.

Running the query will result in the names of three exceptional students.

| **first\_name** | **last\_name** | **average\_grade** | **tag** |
| --- | --- | --- | --- |
| John | Cheese | 9.00 | exceptional |
| Rowan | Chatkinson | 9.50 | exceptional |
| Petunia | Opportunia | 8.67 | exceptional |

## Using More Than one CTEs in a Query

It is possible to define and use more than one CTEs in a query. You do so by separating every CTE with a comma, and you use a WITH statement only when defining the first CTE.

Let me show you an example. With the same tables from the previous example, you have the following task: show the name of the subjects and their respective average and minimum grades, but only for those subjects in which everybody passed the exam, i.e. their mark is 5 or above.

To get the desired result, your query should look like this:

|  |
| --- |
| WITH subject\_average AS (  SELECT  su.id,  su.subject\_name,          AVG (e.grade) AS subject\_average\_grade  FROM subjects su JOIN exams e ON su.id = e.subject\_id  GROUP BY su.id, su.subject\_name  ),    min\_grade AS (  SELECT  su.id,          su.subject\_name,          MIN (e.grade) AS subject\_min\_grade  FROM subjects su JOIN exams e ON su.id = e.subject\_id  GROUP BY su.id, su.subject\_name  HAVING MIN (e.grade) > 5  )    SELECT  sa.id,          sa.subject\_name,          sa.subject\_average\_grade  FROM subject\_average sa JOIN min\_grade m ON sa.id =m.id; |

First, a CTE called **subject\_average** is defined. It selects columns id and subject\_name from the table **subjects**. Next, it calculates the average grades using data from the table **exams** and assigns the results in the new column subject\_average\_grade. Then, it groups the data to get the result by subject.

Now, you define the second CTE. Remember what I said earlier—you separate CTEs with commas and write the second CTE omitting the WITH statement. The second CTE here is named **min\_grade**. It too selects id and subject\_name from the table **subjects** then calculates the minimum grades, showing the result in the new column subject\_min\_grade. It groups the data as was done in the first CTE. Since you need the result only for the subjects in which everybody passed, you use a HAVING clause to select only the subjects in which the minimum grade is 5 or above.

Finally, you write the SELECT query which will show the subject ID, the subject name, and the average grade for each subject that meets the criteria. There are only two such subjects:

| **id** | **subject\_name** | **subject\_average\_grade** |
| --- | --- | --- |
| 5 | Monetary Policy | 7.40 |
| 6 | Tax | 8.00 |

Once you learn the basics of CTEs, there’s [the Recursive Queries course](https://learnsql.com/course/common-table-expressions/) with plenty more examples where you can practice writing the syntax.

## When to Use CTEs

CTEs allow you to perform **multi-level aggregations**. What are they?

Let’s go back to the tables we used in the previous examples. The task now is to calculate the average minimum grade and the average maximum grade by subject.

Where would you start? If you think logically, you should first find the minimum and maximum grades per subject then find the average of the results by subject. It’s straightforward—the code looks like this:

|  |
| --- |
| SELECT      su.id,          MIN (e.grade) AS min\_grade,          MAX (e.grade) AS max\_grade,          AVG (MIN (e.grade)) AS avg\_min\_grade,          AVG (MAX (e.grade)) AS avg\_max\_grade  FROM subjects su JOIN exams e ON su.id = e.subject\_id  GROUP BY su.id, su.subject\_name; |

Logically speaking, this tries to calculate the minimum grade and the maximum grade by subject first, then the average of those values. Voila! You now run the code, and get a message that looks like this:

Msg 130, Level 15, State 1, Line 16

Cannot perform an aggregate function on an expression containing an aggregate or a subquery.

Hm, not the outcome you hoped for? That’s because SQL doesn’t allow constructions such as AVG (MIN (e.grade)). Your thoughts were correct, but you have to use a CTE to translate them into a SQL code. Here’s how to do it:

|  |
| --- |
| WITH min\_max\_grade AS (  SELECT      su.id,          MIN (e.grade) AS min\_grade,          MAX (e.grade) AS max\_grade  FROM subjects su JOIN exams e ON su.id = e.subject\_id  GROUP BY su.id, su.subject\_name  )    SELECT      AVG (min\_grade) AS avg\_min\_grade,          AVG (max\_grade) AS avg\_max\_grade  FROM min\_max\_grade; |

The CTE is named **min\_max\_grade**. In it, there is a SELECT statement that calculates the minimum and the maximum grades by subject, as I intended to do in the query that returned the error message. The result is shown in the new columns min\_grade and max\_grade. A CTE now helps translate your logic into code.

After defining the CTE, you write a SELECT statement that calculates the average of min\_grade and max\_grade from the CTE. The result will be shown in the new columns avg\_min\_grade and avg\_max\_grade. Now that you see it, it’s easy, right?

| **avg\_min\_grade** | **avg\_max\_grade** |
| --- | --- |
| 4.166666 | 9.833333 |

CTEs are also very helpful when you need to organize long and complex queries. Using CTEs will improve the readability of your code, since it breaks down the code nicely into separate steps. It becomes easier to change the code or correct errors. If you were to insist on not using CTEs, your code could look like this:

|  |
| --- |
| SELECT      AVG (min\_grade) AS avg\_min\_grade,          AVG (max\_grade) AS avg\_max\_grade  FROM (  SELECT  su.id,          su.subject\_name,          MIN (e.grade) AS min\_grade,          MAX (e.grade) AS max\_grade      FROM subjects su JOIN exams e ON su.id = e.subject\_id      GROUP BY su.id, su.subject\_name  ) AS min\_max; |

Compared to the solution using a CTE, this seems a bit all over the place and harder to read. Reading subqueries can be difficult, because you have to first think about what each subquery does, then go back to the main query, and somehow connect them all in your head. Besides, using subqueries like this goes counter to how your mind works logically and how you think about the steps that would get you to the solution. Recall that you broke down the problem into two steps: first, calculate the minimum and the maximum grades for every subject, then calculate the average of the minimums and the maximums. The CTE code reflects this order exactly.

The logic in the code with a subquery is the opposite of how you thought about the solution. Here, we first write that you want some average of the grades, then we specify in the subquery that you want the averages to be of the minimum and the maximum grades. When you use a subquery, how you write the code generally goes counter to how you think of the logic.

And if the code with a subquery is less readable and harder to understand than the code with a CTE in this simple example, imagine what it would be like if you had to write more complex queries! You would be scratching your head, trying really hard just to understand what every part of the code does. Having difficulties understanding a code can be very frustrating. This is where CTEs can help you.

You’ve probably noticed that CTEs are a lot like subqueries. Maybe you were wondering why I am using CTEs when everything I did could have been done with subqueries. That’s true, but aside from being more readable, CTEs have one big advantage over subqueries: the results from a CTE can be used more than once in a query. If you’re interested in more on this topic, I recommend [reading about more differences between CTEs and subqueries](https://learnsql.com/blog/sql-subquery-cte-difference/).

I mentioned earlier that CTEs may be non-recursive or recursive. So far, we’ve looked at only non-recursive CTEs. Recursive CTEs are CTEs that reference themselves; by doing so, they return the sub-result and repeat the process until they return the final result. Using recursive CTEs really unlocks [the power of CTEs](https://learnsql.com/blog/get-to-know-the-power-of-sql-recursive-queries/); they are useful when processing hierarchical structures, such as [trees](https://learnsql.com/blog/do-it-in-sql-recursive-tree-traversal/) and graphs.

## Getting the Hang of how to Use CTEs?

We’ve covered some of the basics of the CTE in this article. You’ve learned what a CTE is, understood its syntax, and reviewed some simple examples to give you a feel for what CTEs can do. I also pointed out some common uses of CTEs to help you find a way to use them in your study or work. I hope to have given you some good directions; now it’s your turn to put what you’ve learned into practice.

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 examples to show how SQL CTEs can help you with complex calculations and hierarchical data structures.

## 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.

If you are new to CTEs, you may need to check out [this article that explains in more detail how CTEs work](https://learnsql.com/blog/how-cte-works/).

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 Examples

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

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.

| **orders** | | | | | |
| --- | --- | --- | --- | --- | --- |
| id | date | customer\_id | store | employee\_id | amount |
| 101 | 2021-07-01 | 234 | East | 11 | 198.00 |
| 102 | 2021-07-01 | 675 | West | 13 | 799.00 |
| 103 | 2021-07-01 | 456 | West | 14 | 698.00 |
| 104 | 2021-07-01 | 980 | Center | 15 | 99.00 |
| 105 | 2021-07-02 | 594 | Center | 16 | 1045.45 |
| 106 | 2021-07-02 | 435 | East | 11 | 599.00 |
| 107 | 2021-07-02 | 246 | West | 14 | 678.89 |
| 108 | 2021-07-03 | 256 | East | 12 | 458.80 |
| 109 | 2021-07-03 | 785 | East | 12 | 99.00 |
| 110 | 2021-07-03 | 443 | Center | 16 | 325.50 |

Now, let’s write a couple of SQL queries! You may also practice SQL CTEs in this interactive [Recursive Queries](https://learnsql.com/course/common-table-expressions) course that covers all kinds of CTEs.

### Example 1

In our first example, 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:

|  |
| --- |
| 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 output:

| **id** | **store** | **amount** | **avg\_for\_store** |
| --- | --- | --- | --- |
| 101 | East | 198.00 | 338.70 |
| 102 | West | 799.00 | 725.30 |
| 103 | West | 698.00 | 725.30 |
| 104 | Center | 99.00 | 489.98 |
| 105 | Center | 1045.45 | 489.98 |
| 106 | East | 599.00 | 338.70 |
| 107 | West | 678.89 | 725.30 |
| 108 | East | 458.80 | 338.70 |
| 109 | East | 99.00 | 338.70 |
| 110 | Center | 325.50 | 489.98 |

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 example.

### Example 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 example, 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:

|  |
| --- |
| 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; |

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 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:

| **store** | **average\_order** | **min\_avg\_order\_store** | **max\_avg\_order\_store** |
| --- | --- | --- | --- |
| Center | 489.98 | 338.70 | 725.30 |
| East | 338.70 | 338.70 | 725.30 |
| West | 725.30 | 338.70 | 725.30 |

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.

Read [this guide](https://learnsql.com/blog/sql-cte-best-practices/) to learn the SQL CTE best practices.

### Example 3

In our next example, 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:

| **store** | **big\_orders** | **small\_orders** |
| --- | --- | --- |
| Center | 2 | 1 |
| East | 2 | 2 |
| West | 3 | NULL |

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.

### Example 4

For the next two examples, 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.

| **employees** | | | | | |
| --- | --- | --- | --- | --- | --- |
| id | first\_name | last\_name | superior\_id | department | bonus |
| 1 | John | Davies | NULL | CEO | 2545.00 |
| 2 | Mark | Taylor | 1 | Finance | 1100.00 |
| 3 | Kate | Wilson | 1 | Operations | 900.00 |
| 4 | Olivia | Watson | 3 | Operations | 450.00 |
| 5 | James | Addington | 1 | Sales | 1900.00 |
| 6 | Rachael | White | 1 | Marketing | 1250.00 |
| 7 | Sara | Clinton | 6 | Marketing | 1000.00 |
| 11 | John | Smith | 5 | Sales | 800.00 |
| 12 | Noah | Jones | 11 | Sales | 500.00 |
| 13 | Steven | Brown | 5 | Sales | 900.00 |
| 14 | Liam | Williams | 13 | Sales | 700.00 |
| 15 | Paul | Lee | 5 | Sales | 500.00 |
| 16 | Patrick | Evans | 15 | Sales | 500.00 |

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.

|  |
| --- |
| WITH avg\_bonus\_department AS      (SELECT department, AVG(bonus) AS average\_bonus      FROM employees      GROUP BY department),      above\_average AS      (SELECT e.department, count(\*) AS employees\_above\_average       FROM employees e       JOIN avg\_bonus\_department avg       ON e.department = avg.department       WHERE bonus > average\_bonus       GROUP BY e.department),       below\_average AS       (SELECT e.department, count(\*) AS employees\_below\_average       FROM employees 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:

| **department** | **average\_bonus** | **employees\_above\_average** | **employees\_below\_average** |
| --- | --- | --- | --- |
| CEO | 2545.00 | NULL | NULL |
| Marketing | 1125.00 | 1 | 1 |
| Finance | 1100.00 | NULL | NULL |
| Operations | 675.00 | 1 | 1 |
| Sales | 828.57 | 2 | 5 |

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.

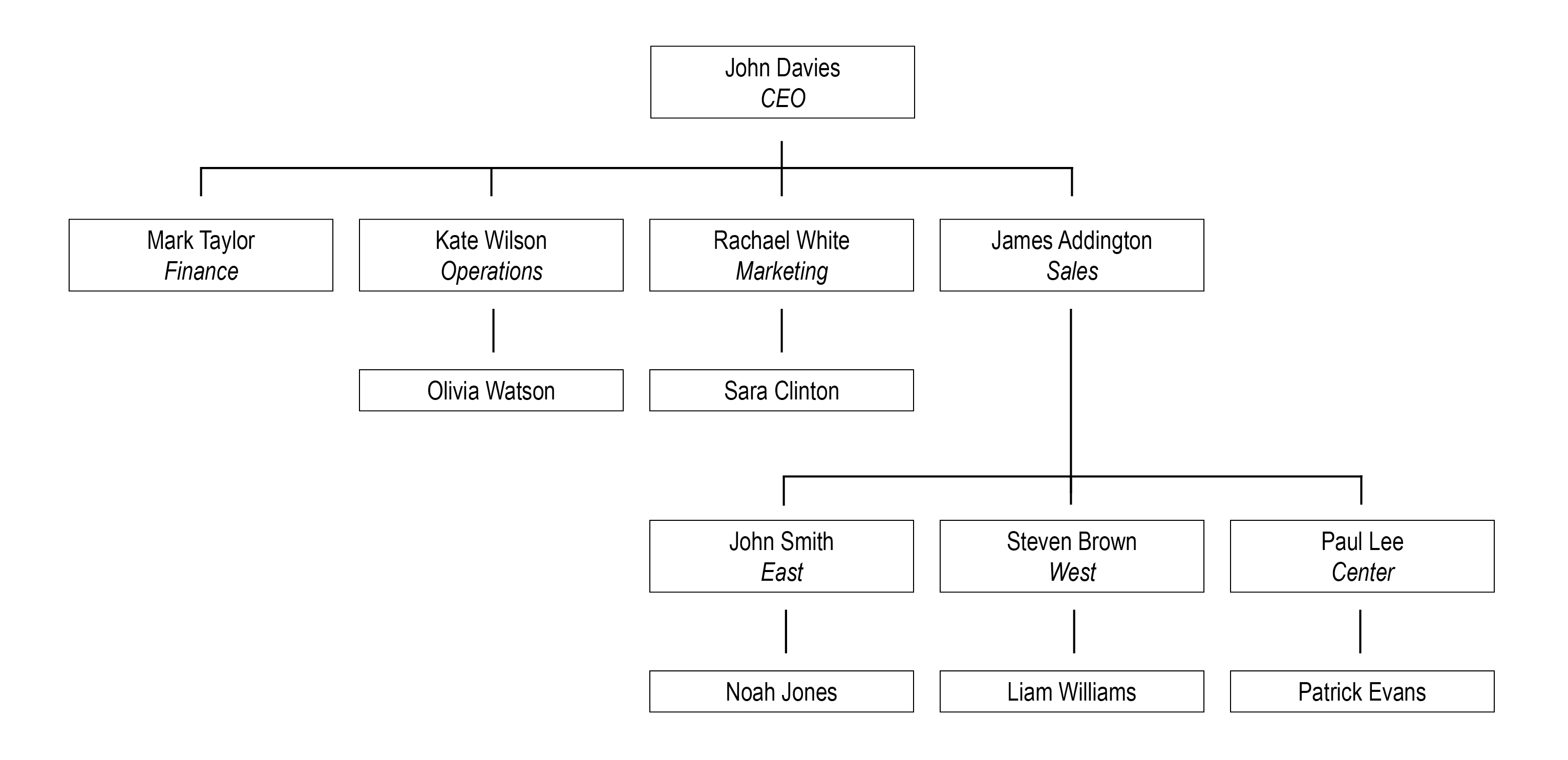
For the Sales department, we can see that the average bonus was $828.57, and only two out of seven people had bonuses above the department average.

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 example with a recursive query.

### Example 5

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

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:

|  |
| --- |
| WITH RECURSIVE levels AS (    SELECT      id,      first\_name,      last\_name,      superior\_id,      1 AS level    FROM employees    WHERE superior\_id IS NULL    UNION ALL    SELECT      employees.id,      employees.first\_name,      employees.last\_name,      employees.superior\_id,      levels.level + 1    FROM employees, levels    WHERE employees.superior\_id = levels.id  )    SELECT \*  FROM levels; |

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:

| **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 |

The topic of recursive queries is quite challenging, so I will not go into more details here. But please make sure to check out [this article explaining recursive CTEs with examples](https://learnsql.com/blog/do-it-in-sql-recursive-tree-traversal/), especially if you work with hierarchical data.