What is a Common Table Expression

The following is chapter 2 in its entirety from my Common Table Expressions book.

What is a Common Table Expression?

The first time I tried snowboarding after being a skier for many years it was very uncomfortable. Everything was difficult, riding the snowboard was strange and I ended up with my face in the snow more times than I can remember. There were times that I felt as though I should just give up and go back to skiing, but I did not. I just kept trying and trying and trying. After a several attempts at snowboarding I realized that I was starting to have fun, and that it was getting easier to get down the mountain without ending up with my face in the snow. That was several years ago, and today I still love snowboarding.

For a T-SQL programmer, learning Common Table Expressions (CTEs) will be like a skier learning to snowboard. The first few times you use CTEs it is going to feel strange, or different. After a while, the confidence level comes and the CTEs will start to be comfortable, and then you will reach a point where they become natural, and you will want to use them all the time.

To get started, just think of a Common Table Expression as a temporary named result set that can be used within a single query. The CTE is like a temporary table (temp table) or a temporary in-line view (if there was such a thing). The CTE has the ability to group a query into a named expression that can then be queried similarly to how we would query a view or a temp table.

Unlike a temp table, there is no storage associated with a CTE, other than that which is needed to run the current query. Also, unlike a view, the life-time of a CTE is constrained to the run time of a single query. Later in the book we will get into more details on the times when it is appropriate to use a view or a temp table rather than a CTE. For now let’s get started with building our first CTE.

READER NOTE: *Please run the CTEBookSetup.sql script in order to follow along with the examples in this chapter. The script mentioned in this chapter is available at http://SteveStedman.com.*

Overview of a CTE

Most of us have heard the story of Little Red Riding Hood. She is told to bring a basket full of goodies to grandma's house. What exactly is in that basket? Maybe the basket has bread and cheese for a snack. Perhaps grandma needs her medicine and that is in the basket too. Mom knew exactly what grandma needed Little Red Riding Hood to bring and packed the perfect basket.

Which tables should be packed into your basket of goodies that makes the next report needed by your company? At first, a Common Table Expression looks like one nice neat little package of query logic that can combine tables into one easy name. That is true, but if the story ended there then it would seem like a view or temp table. There are some clear advantages to CTEs well worth knowing.

When we have access to a table we can query on any field in the table we want. What about fields not in that table but derived or calculated from that field?

Sometimes we will need to create new objects from existing objects to get queries to run. We can use views or derived tables (explained later in this chapter) to materialize queries and expression fields. The derived table's code provides many options for our queries, but the code can become complex and make for messy reading.

In this chapter we will combine different techniques to show the power of temporary objects and also show how the CTE can make complex code easier to read.

The CTE was one of the standout debut features in SQL Server 2005 and was widely applauded by SQL Server developers. Readers who are learning SQL Server now can be grateful they don't need to experience the pain and pitfalls of temporary tables (a.k.a., "temp tables", which are different from derived tables) and other messy constructs whose work is now elegantly accomplished by the CTE. Common Table Expressions play an important role when writing recursive queries, so pay close attention to how the CTE works in these exercises.

Writing Your First CTE

To get started with writing our first CTE it is good to understand the derived table syntax. After seeing the old syntax we can compare it to a CTE.

The CTE shares advantages with the derived table, with a few extras to boot. We won't learn every advantage here, but we will learn some of the more useful ones to make our code more readable.

First we will start with a very basic derived table example (Figure 2.1).

```
SELECT *
  FROM (SELECT id, department, parent
        FROM Departments) as Dept;
```

Editor | Results | Messages

	id	department	parent
1	1	Camping	NULL
2	2	Cycle	NULL
3	3	Snowsports	NULL
4	4	Fitness	NULL
5	5	Tents	1
6	6	Backpacks	1
7	7	Sleeping Bags	1
8	8	Cooking	1
9	9	Bikes	2
10	10	Helmets	2
11	11	Locks	2

Figure 2.1 This shows a basic derived table result set.

To get started with CTEs we kept this first query very simple. In reality the derived table example in Figure 2.1 doesn't do much more than simply run the inner query. To help keep the lesson clear we will use a simple CTE example as well.

We will write a query that does the same thing as the derived table in Figure 2.2 but use a CTE instead.

```
WITH departmentsCTE AS
(
 SELECT id, department, parent
   FROM Departments
)
SELECT * FROM departmentsCTE;
```

Editor | Results | Messages

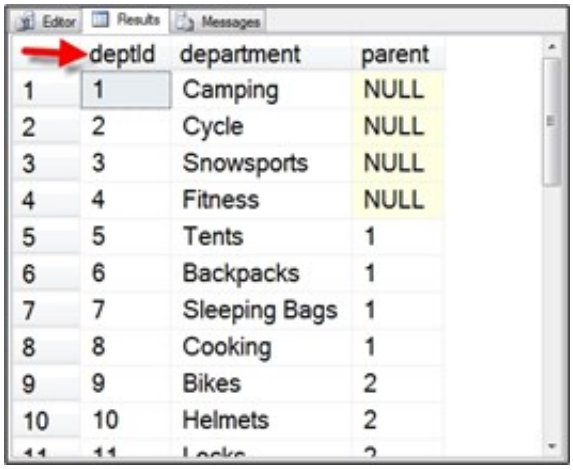
	id	department	parent
1	1	Camping	NULL
2	2	Cycle	NULL
3	3	Snowsports	NULL
4	4	Fitness	NULL
5	5	Tents	1
6	6	Backpacks	1
7	7	Sleeping Bags	1
8	8	Cooking	1
9	9	Bikes	2
10	10	Helmets	2
11	11	Locks	2

Figure 2.2 Output from a basic CTE query.

Running the CTE example or the derived table example will produce exactly the same results. At this point there is no clear advantage to the CTE syntax over the derived table syntax. However this is just the foundation to move on to the more interesting examples.

The column name id is not very descriptive and could be better understood if it was called deptId. After the CTE name there is an optional set of parentheses which acts as part of the declaration. In this example we show the id field with the alias of deptId shown in bold.

```
WITH departmentsCTE (deptId, department, parent) AS
(
  SELECT id, department, parent
    FROM Departments
)
SELECT *
  FROM departmentsCTE;
```



	deptId	department	parent
1	1	Camping	NULL
2	2	Cycle	NULL
3	3	Snowsports	NULL
4	4	Fitness	NULL
5	5	Tents	1
6	6	Backpacks	1
7	7	Sleeping Bags	1
8	8	Cooking	1
9	9	Bikes	2
10	10	Helmets	2

Figure 2.3 This shows the updated deptId column name as set in the CTE declaration.

We don’t need to use different names to have the column list as part of the definition. We could be explicit and still use the original names as seen in this code sample shown in bold:

```
WITH departmentsCTE(id, department, parent) AS
(
  SELECT id, department, parent
    FROM Departments
)


SELECT *
  FROM departmentsCTE;
```

Editor | Results | Messages

	id	department	parent
1	1	Camping	NULL
2	2	Cycle	NULL
3	3	Snowsports	NULL
4	4	Fitness	NULL
5	5	Tents	1
6	6	Backpacks	1
7	7	Sleeping Bags	1
8	8	Cooking	1
9	9	Bikes	2
10	10	Helmets	2
11	11	Locks	2

Figure 2.4 Column headings shown from the declaration without change.

WITH Syntax

Most queries that return data start with the SELECT statement, but this is one way the CTE is different. When we look at a query that uses a CTE, it will always start with the following keyword:

WITH …

Immediately after the WITH keyword will be the name of the CTE. When naming the CTE I usually name it in a way similar to how stored procedures, temp tables, or views may be named. Often times, when naming a CTE, the prefix or suffix of “CTE” is used. Here are two examples of how to start creating a CTE:

WITH **departmentsCTE**

or

WITH **cteDepartments**

Using the prefix or suffix is completely optional, but it can be helpful when writing the query to avoid conflicts or confusion between the CTE and actual table names or view names.

Optional Columns

Once the CTE has been started and given a name, the next part of the syntax is the optional column declarations. As the name suggests, these are optional, but they are useful in explicitly showing what the CTE is going to return. We take the same CTE declaration from the previous section and specify the three columns that will be returned. Think of this as the declaration of the CTE, and if the CTE returns a different number of columns as compared to the column declaration then an error will be thrown. In the following example the column definitions are shown in bold.

WITH departmentsCTE**(id, department, parent)**

The column declaration of the CTE statement is optional, and if written incorrectly can cause misleading results. If three columns are specified in the CTE query, as shown in the previous example, but the three columns are in a different order, there won't be an error. The resulting data will just end up with the wrong column name. In the following example the id field will show up as department:

```
WITH departmentsCTE(department, parent, id) AS
(
  SELECT id, department, parent
    FROM Departments
```

```
)

SELECT * FROM departmentsCTE;
```

Editor | Results | Messages

	department	parent	id
1	1	Camping	NULL
2	2	Cycle	NULL
3	3	Snowsports	NULL
4	4	Fitness	NULL
5	5	Tents	1
6	6	Backpacks	1
7	7	Sleeping Bags	1
8	8	Cooking	1
9	9	Bikes	2
10	10	Helmets	2
11	11	Locks	2

Figure 2.5 Column headings don’t match the data.

When creating a CTE we can leave the optional columns definition off while working on all of the details of the CTE. Once complete, we can then add the columns definition to improve the readability of the query. This makes it much easier for the next person looking at the query to understand what it is doing.

Query Definition

Now that we understand the WITH syntax, CTE name, and the use of the optional columns, we can move into the query definition of the CTE. The following code shows the changes in bold:

```
WITH departmentsCTE(id, department, parent) AS
(
  SELECT id, department, parent
    FROM Departments
)



SELECT * FROM departmentsCTE;
```

The query definition starts with the AS keyword and has a query enclosed in the parentheses.

If we choose to not use the optional columns declaration, our query may look like the following:

```
WITH departmentsCTE AS (
   SELECT id, department, parent
     FROM Departments
)



SELECT * FROM departmentsCTE;
```

Most any query could go inside of the CTE query definition. However there are some things that can’t go into the query. The query definition must be a SELECT statement. We can’t use INSERT, UPDATE or DELETE for example. In a later chapter we will cover using CTEs with INSERT, UPDATE, DELETE and other DML statements. However they will not be inside of the CTE query definition.

Calling or Running the CTE

Once the CTE has been created as outlined in the previous example, the next step is to run it. A CTE is executed, or run, by referencing it in a query. Following the closing parentheses of the CTE query definition, start another SELECT statement that references the CTE by name.

```
WITH departmentsCTE(id, department, parent) AS
(
  SELECT id, department, parent
    FROM Departments
)
```

```
SELECT *
  FROM departmentsCTE;
```

Editor | Results | Messages

	id	department	parent
1	1	Camping	NULL
2	2	Cycle	NULL
3	3	Snowsports	NULL
4	4	Fitness	NULL
5	5	Tents	1
6	6	Backpacks	1
7	7	Sleeping Bags	1
8	8	Cooking	1
9	9	Bikes	2
10	10	Helmets	2
11	11	Locks	2

Figure 2.6 Output from departmentsCTE.

The most basic execution of a CTE would be SELECT * FROM followed by the name of the CTE (Figure 2.6).

At this point the Common Table Expression that we have created is not much more interesting than the simple derived table example that we started with at the beginning of the chapter, but it has covered the basic syntax. This includes how it is built, how the CTE query is defined, and how it is executed. We now have the CTE foundation that is needed to move forward with some of the more interesting features.

Terminate Previous Statements

It is a common best practice in T-SQL to terminate queries with a semi-colon, but in most cases it is completely optional and will not cause any problems if the semi-colon is overlooked. When using Common Table Expressions in T-SQL batches with multiple statements, the preceding T-SQL statement MUST be terminated with a semi-colon or an error will be thrown.

An alternative to this would be to use the GO statement. Since SSMS uses the GO statement to break the query into different batches, it is similar to using a semi-colon. Any variables, temp tables, or other structures go out of the scope of the current query. Keep in mind that the GO statement can’t be used inside of a stored procedure or function. The following code uses a semi-colon at the end of the CTE but no semi-colon after the first query.

```
SELECT * FROM Departments
```

```
WITH departmentsCTE AS
(
  SELECT id, department, parent
    FROM Departments
)
```

```
SELECT *
  FROM departmentsCTE;
```

It looks like this query would run fine, but it throws an error message:

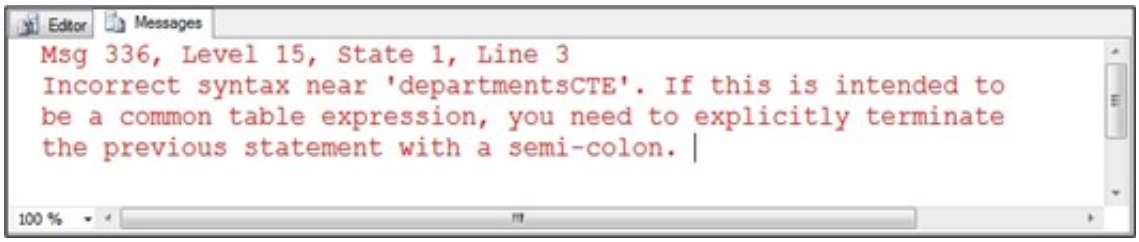


Figure 2.7 Error caused by the missing semi-colon in the first query.

In order to avoid this syntax error, it is a good practice to start a CTE with a semi-colon as shown in the following code. This helps when someone who doesn't know about CTEs is working on a query at a later time and inserts another query in the script before the CTE without a semi-colon.

```
;WITH departmentsCTE AS
```

By simply adding the semi-colon before the WITH keyword to terminate any previous statements, we get two result sets in the Results window as was intended by the original query.

```
;WITH departmentsCTE AS
(
  SELECT id, department, parent
    FROM Departments
)


SELECT *
```

```
FROM departmentsCTE;
```

Editor | Results | Messages

	id	department	parent
1	1	Camping	NULL
2	2	Cycle	NULL
3	3	Snowsports	NULL
4	4	Fitness	NULL
5	5	Tents	1
6	6	Backpacks	1

	id	department	parent
1	1	Camping	NULL
2	2	Cycle	NULL
3	3	Snowspo...	NULL
4	4	Fitness	NULL
5	5	Tents	1
6	6	Backpacks	1

Figure 2.8 Successful output from both queries when separated with a semi-colon.

Scope

When you come to a set of railroad tracks and there are no trains coming, it is easy to cross the tracks and continue down the road. In the event of a train being present, the crossing arms come down and block your access the road. When you are sitting at the railroad crossing you can see the other side by looking through the empty cars, but it is unreachable. Scope is a programming term very similar to this where some things are within reach, like the other side of the tracks when there is not a train, and some things are out of reach, like when there is a train crossing.

The question of scope comes up when discussing CTEs. Once a CTE is defined it can be referenced a number of times in the same T-SQL statement (for instance in a self-join where a query calls the CTE twice). The scope of the CTE is confined to a single T-SQL statement. Think of the scope of the CTE as being constrained to a single query.

In the next example, the last query in the code block, WHERE id = 2, will not run since the CTE scope ended with the first query. Similar to the railroad crossing with the crossing arms down, the last SELECT statement can't access the CTE as the earlier SELECT statement did.

```
WITH departmentsCTE(id, department, parent) AS
(
  SELECT id, department, parent
    FROM Departments)
```

```
SELECT * FROM departmentsCTE WHERE id = 1;
SELECT * FROM departmentsCTE WHERE id = 2;
```

One CTE can call another CTE, but even in this case the scope of the CTE is just a single statement.

If the results of a CTE need to be kept around beyond the scope of that single call, we can insert the results into a table, temp table or table variable to be used once the CTE query terminates.

When the previous query is run the Results window will show the results of the first SELECT statement, but the Messages window will show the error message thrown by the second SELECT statement attempting to access a CTE outside of its scope.

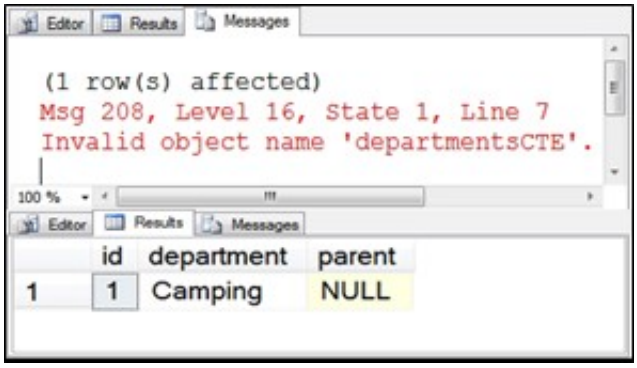


Figure 2.9 Results shown from first query and the error from the second query.

Why use a CTE?

When I was shopping for a car a salesperson once told me “if you want this car you need to buy it before we close today”. I pondered that question for a while and eventually asked “why?” I thought to myself, “is there someone else writing up an offer for this car?” After dodging the question for a bit, I finally was able to get the salesperson to admit that the reason I needed to buy the car today was that it was the last day of his pay cycle. If I purchased it today he would get commission for the sale on his next paycheck. It had nothing to do with the price changing, the availability, or with someone else ready to scoop up that particular car. It was all about his next paycheck. Once I understood his motivation I realized that it might be better to shop around a bit more.

CTEs may feel a bit strange, and may make you ask, “Why would I do that?” Similar to the car dealer, understanding why may affect your actions. The two biggest reasons to use CTEs are to simplify complex queries and to be able to write recursive queries. Recursion with a CTE is something that can easily be mastered even though others may consider it difficult.

Simplifying Queries with a CTE

Have you ever looked at a confusing query that, after you analyzed it, was in fact doing something really simple? It could be repeating code or using an old lengthy syntax. For readability, having a whole lot of repeating code in a huge query is not as clean as it could be. One way to get around this is to use a view in place of a derived table reference in big queries. This is creative but not optimal since you may need additional permissions on the database to create a view. Additionally, the view ends up hiding the query that you are using and is often times overlooked when debugging a problem. Views were a creative solution to the problem prior to the introduction of CTEs.

Most people consider a two-table join to be a very simple query. If one of those tables is a derived table it looks far more complicated than if we use a Common Table Expression. The following code shows the join with a CTE is much easier to read:

```
--Query1 Join with Derived table
SELECT * FROM (SELECT *
  FROM Employee
 WHERE HireDate > '1/1/2000') AS emp
 INNER JOIN Location AS lo
    ON emp.LocationID = lo.LocationID
```

```
--Query2 Join with CTE
;WITH emp AS
(
  SELECT *
    FROM Employee
   WHERE HireDate > '1/1/2000'
)
```

```
  SELECT *
    FROM emp
   INNER JOIN Location AS lo
      ON emp.LocationID = lo.LocationID
```

Two-table joins are not too difficult with a derived table, but sooner or later we may end up with one that is very confusing due to many levels of nested queries. We can use a CTE as a less confusing alternative.

If your experience is like mine, you will end up writing the query knowing that there has to be a better way of doing it other than to just keep writing the huge query, with lots of copying and pasting.

Once we master CTEs this old derived table way of sharing code will be a thing of the past. We won't be looking to views or table valued functions to do what we can do in a CTE. You will learn how to define different queries known as the Common Table Expression (CTE) that can be reused again and again in the same query without the sloppy overhead of the copy and paste of the derived table code.

When we simplify our code by using a CTE rather than a derived table, the queries will typically perform the same. There is no noticeable performance increase in using a CTE compared to a derived table, but it does make your code easier to read and easier to maintain.

Chapter 3 will go into detail on how to replace derived tables with a CTE.

Recursive Queries

A tent is a type of camping equipment and a backpacking tent is a type of tent. From a hierarchical standpoint you can show parents and children in the form of relations and levels in an outline like this:

- Camping - Level 1
    - Tent - Level 2

- Backpacking Tent - Level 3

Without CTEs, one of the common ways that a hierarchy like this was built was to write a derived table query for each level of the hierarchy. Once written, we then use UNION ALL to merge all of the queries into one result set. The derived table with the UNION ALL works as a great solution with two or three levels in the hierarchy but when we need to go to an unknown depth, that solution falls apart.

For programmers (and computer science students) recursion is one of those tools that (once it is understood) we end up wishing we had in T-SQL queries. Prior to CTEs this was extremely difficult to do, and couldn't happen in a single T-SQL statement.

The CTE will allow us to write recursive queries that permits some things that otherwise would be very challenging to do in a single query

Summary

Common Table Expressions are easy to get started with on all versions of SQL Server 2005 and newer. CTEs will likely feel very strange to begin with, but after implementing a few of them, they will become more comfortable.

A CTE is made up of the WITH keyword followed by the name of the CTE with optional columns. After that, the AS keyword is used to start the CTE query definition. Once the query definition is completed and surrounded with parentheses the CTE can be run using a SELECT statement.

```
WITH departmentsCTE(id, department, parent) AS
(
  SELECT id, department, parent
```

```
    FROM Departments
)
SELECT * FROM departmentsCTE;
```

Editor | Results | Messages

	id	department	parent
1	1	Camping	NULL
2	2	Cycle	NULL
3	3	Snowsports	NULL
4	4	Fitness	NULL
5	5	Tents	1
6	6	Backpacks	1
7	7	Sleeping Bags	1
8	8	Cooking	1
9	9	Bikes	2
10	10	Helmets	2

Figure 2.10 Output from the departmentsCTE query.

The scope of a CTE is confined to a single T-SQL statement, but the results can be saved and reused by placing them in a table, temp table, or table variable.

Some of the main reasons to use CTEs are as an alternative to those complex queries that use derived tables, or to create recursive queries.

Points to Ponder - What is a Common Table Expression?

1. Common Table Expressions require a name or an alias which often times includes the abbreviated term “CTE” to avoid confusion with other tables or view names.
2. A CTE acts like a temporary view or run time view which provides a tabular result set that can be queried like a regular table.

3. CTEs were first introduced in SQL Server 2005.
4. All editions of SQL Server support CTEs. Standard, Enterprise, Developer, Express.
5. A CTE can help simplify a complex query.
6. A CTE starts with the WITH keyword rather than SELECT.
7. The scope of a CTE is confined to a single query.
8. If there are multiple queries in a script, the previous query must be terminated with a semi-colon or the CTE will throw an error.
9. A simple CTE doesn't add much value over a similar derived table, however it is the foundation for some things that would be very hard to do without the simple CTE.
10. The specified columns in the CTE are optional but are beneficial in validating that the results will be the data type being asked for.
11. The CTE can be a great replacement for table variables or temp tables in some cases. There are cases where the temp table or table variable is a better solution.
12. Unlike the old temp table construct, CTEs are not written to disk and do not need to be de-allocated once executed.
13. CTEs allow users who can't create temp tables to perform more complex queries (e.g., users with read-only permission to the database). Prior to SQL 2005, it was common for users to be given a higher level of permissions (e.g., Admin rights) in order to be able to create temp tables for more complex queries. Therefore, CTEs help secure databases by allowing users with read access to create complex queries.

1. CTEs are available in SQL Azure also. All versions of SQL Server since SQL 2005 and all variations of SQL Server support CTEs.

Review Quiz - Chapter Two

1. What version of SQL Server do you need to be able to use CTEs?
2. Any version of SQL Server supports CTEs.
3. SQL Azure.
4. Any version of SQL Server 2005 or newer including SQL Azure.
5. Enterprise Edition of SQL Server 2005 or newer.
    1. When creating a CTE, the definition of the columns in the CTE declaration is_____?
6. A waste of my time.
7. Required.
8. Optional but slows down the overall query.

9. Optional but a good practice to help clarify what the CTE is doing.
    1. Choose two reasons to use CTEs.
10. Create recursive queries.
11. Create and store a result set that can be reused in multiple queries.
12. Simplify complex queries.
13. Speed up complex queries.

Answer Key

1. CTEs are available on any version of SQL Server 2005 or newer including SQL Azure, and they are available on all editions from the Express edition to the Enterprise edition including the Standard edition and the Developer edition. This makes (c) the correct answer.
2. When creating a CTE, the definition of columns in the CTE declaration is optional and a good practice to help clarify what the CTE is doing. This does not do anything to change the performance, and it is not required. Answer (d) is correct.
3. Creating recursive queries and simplifying complex queries are two great reasons to use CTEs. CTEs do not store the results to be used in multiple queries, and they generally don’t speed up the query compared to a derived table. The correct answers are (a) and (c).