Recursive CTEs Explained

⊞ **essentialsql.com**/recursive-ctes-explained

July 9, 2016

Recursive CTE's

In this article we explore recursive CTE's (Common Table Expressions). Recursive CTEs are special in the sense they are allowed to reference themselves! Because of this special ability, you can use recursive CTEs to solve problems other queries cannot. Recursive CTEs are really good at working with hierarchical data such as org charts for bill of materials.

If you're unfamiliar with CTE's I would encourage you to read Introduction to Common Table Expressions. Once you're familiar, then come back to this article and we'll dig deeper into the reasons you would want to use recursive CTE's and provide you with some good examples along the way.

Note:

All the examples for this lesson are based on Microsoft SQL Server Management Studio and the AdventureWorks2012 database. Get started using these free tools with my Guide *Getting Started Using SQL Server*.

Introduction

A recursive CTE is a common table expression that references itself. Look up the definition for recursion on google and you'll see recursion defined as "the repeated application of a recursive procedure or definition." Its Latin root, recurrere, means to "run back."

As we'll see both the modern definition and its Latin root serve well to explain this concept.

What is recursion?

Recursion can be pretty abstract and difficult to understand. So before we go much farther learning about recursive CTEs, let's first look at an example to understand the general concept.

How Many People Are in front of me in a line?

No doubt you've been in a long line and wondered how many people were standing before you. Short of standing on your tippy toes and trying to count heads, is there another way to find the answer?
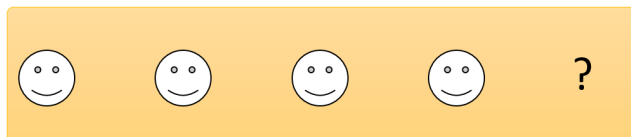
Sure! You could ask the person in front of you what place they were in line. If you know this you could just add one to it, to know your place!

Using recursion to get the answer we could use these instructions:

**Pass these directions to the person in front of you.**
- If there is no one in front of you, then tell the person behind you, you are 1.
- If you are told a number, then add 1 it, and pass the result to the person behind you.

At first the instructions are passed from person to person in line. Eventually we get to the front of the line, and that person has no one else to pass the instructions forward. So, instead, they tell the person behind them they are "1."

This in turn signals the person behind them, to add "1" to the number they were told and pass it back to the person behind them.

This repeats until the end of the line is reached. At this point, the person who was wondering how long the line is, just needs to add one to get the answer.

Let's see how this works visually. In the diagram below you can see where each person is forwarding the instructions to the next person in line.

Hey buddy, pass these instructions to the next guy.
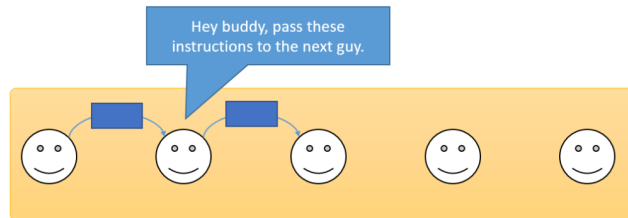
At some point we reach the first person in the line. They see there is no one in front of them and stop passing the notes. This is the terminating condition. As you can imagine, it is important to have a terminating condition. If one didn't exist, then we would try passing the note indefinitely. In computer terms, we would be in an infinite loop.

Hey, my number is "1"

1

Finally, the first person in the line gets the instructions, realizes there is not one else to pass them to. At this point he turns around to the person behind him and says he is "1."

I received a 1. Since 1 + 1 = 2, I'm passing back 2.

2
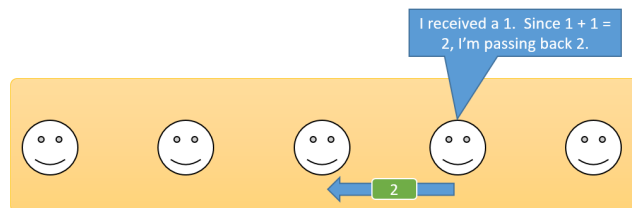
The second guy gets the number "1," adds "1" to it, and passes back "2."
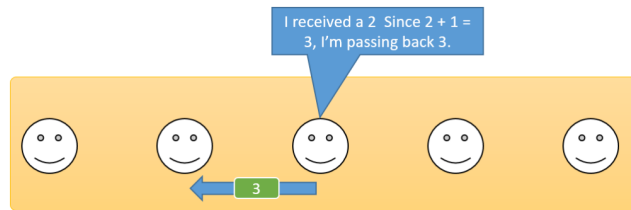
This continues on for each person until there is no one left. At this point the last person in line now knows their position.

Note: This example is inspired by Aaron Krolik's answers to a recursion question on Quora

What Makes Recursion?

A recursive routine has three main parts:

- Invocation – This is the part of your query or program that calls the recursive routine.
- Recursive Invocation of the Routine – This is the part of the routine that calls itself
- Termination Check – This is the logic that makes sure we eventually stop calling the routine, and start providing answers.

Let's take the example instructions from the above example :and identify the three parts:

Pass these directions to the person in front of you.
- If there is no one in front of you, then tell the person behind you, you are 1.
- If you are told a number, then add 1 it, and pass the result to the person behind you.

- Invocation – You hand the blue card to the last person in line.
- Recursive Invocation – Each person is told to pass the direction to the person in front.
- Termination Check – You check to see if there is no one in front of you.

Recursive CTEs

A recursive CTE is a CTE that references itself. In doing so, the initial CTE is repeatedly executed, returning subsets of data, until the complete result is returned.

Being able to reference itself is a unique feature and benefit. It allows recursive CTE's to solve queries problems that would otherwise require the use of temporary tables, cursors, and other means.

Recursive CTE's are well suited to querying hierarchical data, such as organization charts or production bill of materials, where each product's components are made up of subcomponents, and so on.

The general syntax for a recursive CTE is:

```
WITH cte_name (column1, column2, …)
AS
(
    cte_query_definition -- Anchor member
    UNION ALL
    cte_query_definition -- Recursive member; references cte_name.
)
-- Statement using the CTE
SELECT *
FROM   cte_name
```

The general layout is similar to a non-recursive CTE. It is defined using WITH, consists of a query definition, and precedes the statement using the CTE.

Yet, there are significant differences. A recursive CTE must contain a UNION ALL statement and, to be recursive, have a second query definition which references the CTE itself.
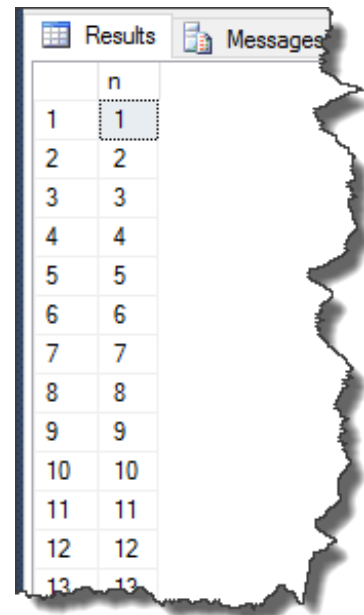
Let's look at a simple example.

Below is a recursive CTE that counts from 1 to 50.

```
WITH   cte
AS     (SELECT 1 AS n -- anchor member
         UNION ALL
         SELECT n + 1 -- recursive member
         FROM   cte
         WHERE  n < 50 -- terminator
         )
SELECT n
FROM   cte;
```

Here are the results:

There are many parts involved in this query, let's look at them:
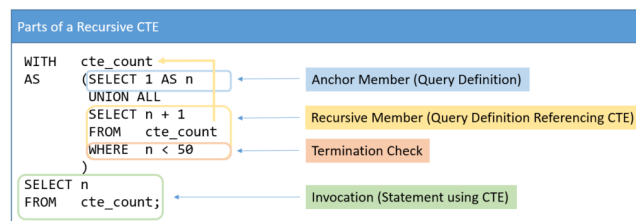
This recursive CTE consists of three main parts:

1. Invocation – This is the statement using the CTE.
2. Anchor Member – This portion executes first, and is only called once.
3. Recursive Member – The portion of the query is repeatedly executed until no rows are returned. The results of each execution are unioned with the prior results.
4. Termination Check – The termination check ensures the query stops.

When this query is run the Anchor Member is run once and its derived rows combined, via the UNION ALL, with the Recursive Member.

This query is repeatedly run until no rows are returned. This is why the termination check is so important.

If we weren't checking for N < 50, the query would always return a row, thus complete with an error. Here is an example of the error you can expect if the termination check is missing or faulty:

```
Msg 530, Level 16, State 1, Line 1
The statement terminated. The maximum recursion 100 has been exhausted before statement completion.
```

Of course there may be instance where the maximum recursion should be greater than 100. In this case you can adjust it using the MAXRECURSION query hint. Here we've increased the recursion to 200.

```
WITH   cte
AS     (SELECT 1 AS n -- anchor member
        UNION ALL
        SELECT n + 1 -- recursive member
        FROM   cte
        WHERE  n < 50 -- terminator
       )
SELECT n
FROM   cte
OPTION (MAXRECURSION 200);
```

When writing a Recursive CTE keep in mind the following guidelines:

- A recursive CTE must contain at least an anchor member and recursive member. Multiple anchor and recursive members can be defined, but all the anchor members should be before the first recursive member.
- The number of columns and their corresponding data types should be the same between the anchor and recursive members. This makes sense, as this is a condition for UNION ALL.
- The FROM clause of the recursive member can only refer to the CTE expression once.

In addition, the following items are not allowed in the recursive member query definition:

- GROUP BY
- HAVING
- LEFT, RIGHT, or OUTER JOIN
- Scalar aggregation
- SELECT DISTINCT
- Subqueries
- TOP

Note: For a more complete list of guidelines and restrictions, please see the MSDN articleWITH common_table_expression (Transact-SQL)

Let's move on to a more comprehensive example.

## Retrieve a Bill of Materials using a Recursive Common Table Expression

The Adventure Works company manufactures bicycles, and as you can imagine, they contain a lot of parts! To keep it all straight the production department has put together a bill of materials or BOM for short.

A BOM lists all the parts for each bike. Bikes are made up of many larger parts, called sub-assemblies, which in turn are made up of other sub-assemblies or components.

This creates a hierarchy of sorts with the product at the top. You can think of this as a parent-child relationship. The parent is the sub-assembly, and the parts making up the sub-assembly the children.

For instance, for a Bike, the wheel would be a sub assembly, which is compose of a rim and spokes.

Here are the table relationships for the AdventureWorks2012 BillOfMaterials:
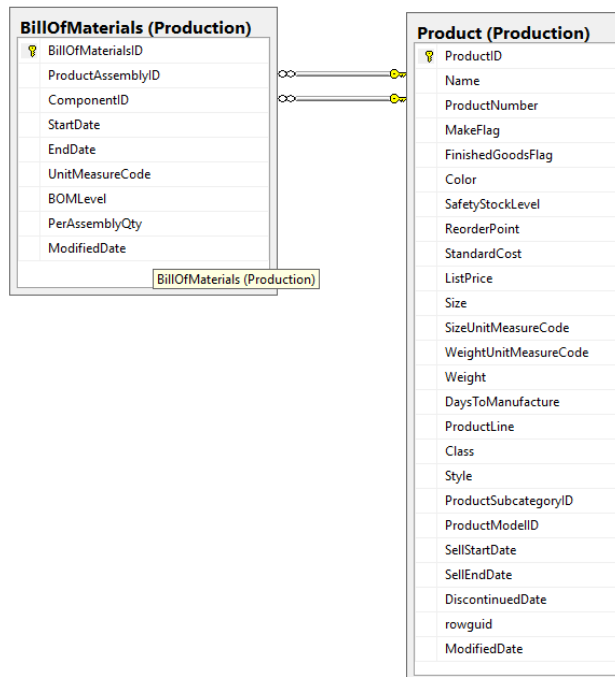
As you can see all products (parts) are listed in the Product table. The BillOfMaterials table contains pairs of ProductID numbers:

- ProductAssemblyID – The sub assembly containing the part (parent)
- ComponentID – Part in the sub assembly (child)

These are foreign keys referring to Product.ProductID values.

Given this suppose we want a list of a products sub-assemblies and its constituent parts? How can we do this?

Getting a list of the top level parts is pretty straight forward. They would be products corresponding to BillOfMaterials entries whose ProductAssemblyID is NULL. Here is a query to get those products:

```
SELECT P.ProductID,
       P.Name,
       P.Color
FROM   Production.Product AS P
       INNER JOIN
       Production.BillOfMaterials AS BOM
       ON BOM.ComponentID = P.ProductID
       AND BOM.ProductAssemblyID IS NULL
       AND (BOM.EndDate IS NULL
            OR BOM.EndDate > GETDATE());
```

And the resulting products:

So how do we get the products and their sub-assemblies?

One idea is to write another query that gets all products whose ProductAssemblyID is one of the top level products and use a UNION ALL to combine the results. We can easily do this with a subquery:

```
SELECT  P.ProductID,
        P.Name,
        P.Color
FROM    Production.Product AS P
        INNER JOIN Production.BillOfMaterials AS BOM
        ON BOM.ComponentID = P.ProductID
        AND BOM.ProductAssemblyID IS NULL
        AND (BOM.EndDate IS NULL
            OR BOM.EndDate > GETDATE())
UNION ALL
SELECT  P.ProductID,
        P.Name,
        P.Color
FROM    Production.Product AS P
        INNER JOIN Production.BillOfMaterials AS BOM
        ON BOM.ComponentID = P.ProductID
        AND (BOM.EndDate IS NULL
            OR BOM.EndDate > GETDATE())
        AND BOM.ProductAssemblyID IN
            (SELECT P.ProductID
             FROM   Production.Product AS P
                    INNER JOIN Production.BillOfMaterials AS BOM
                    ON BOM.ComponentID = P.ProductID
                    AND BOM.ProductAssemblyID IS NULL
                    AND (BOM.EndDate IS NULL
                        OR BOM.EndDate > GETDATE())
            );
```


The blue portion is the query returning the top level products. The green portion returns the sub-assemblies, and the portion in red, is the subquery used to return the set of ProductID's for the top level products used to match ProductAssemblyID's.

But there is an issue with query. It only returns Products or sub-assemblies used in the top level products. Sub-assemblies contained within these sub-assemblies aren't considered.

Of course we could continue to write queries to "dig" deeper into the hierarchy, but can you imagine? The queries would get really long, complicated, and always limited by the number of queries union together.

This is where recursive CTE's shine.

By using recursion, we're able to use the self-referring nature of the CTE to continue to dig into deeper levels of the BOM. Much like the counting example we showed previously, with the following recursive CTE, the common table expression is repeatedly called until the termination condition is met. In this query's case, that is once no further sub-assemblies or components are found.

Here is the recursive CTE you can try:

```sql
WITH cte_BOM (ProductID, Name, Color, Quantity, ProductLevel, ProductAssemblyID, Sort)
AS  (SELECT P.ProductID,
            CAST (P.Name AS VARCHAR (100)),
            P.Color,
            CAST (1 AS DECIMAL (8, 2)),
            1,
            NULL,
            CAST (P.Name AS VARCHAR (100))
     FROM   Production.Product AS P
            INNER JOIN
            Production.BillOfMaterials AS BOM
            ON BOM.ComponentID = P.ProductID
            AND BOM.ProductAssemblyID IS NULL
            AND (BOM.EndDate IS NULL
                OR BOM.EndDate > GETDATE())
     UNION ALL
     SELECT P.ProductID,
            CAST (REPLICATE('|---', cte_BOM.ProductLevel) + P.Name AS VARCHAR (100)),
            P.Color,
            BOM.PerAssemblyQty,
            cte_BOM.ProductLevel + 1,
            cte_BOM.ProductID,
            CAST (cte_BOM.Sort + '\' + p.Name AS VARCHAR (100))
     FROM   cte_BOM
            INNER JOIN Production.BillOfMaterials AS BOM
            ON BOM.ProductAssemblyID = cte_BOM.ProductID
            INNER JOIN Production.Product AS P
            ON BOM.ComponentID = P.ProductID
            AND (BOM.EndDate IS NULL
                OR BOM.EndDate > GETDATE())
    )
SELECT  ProductID,
        Name,
        Color,
        Quantity,
        ProductLevel,
        ProductAssemblyID,
        Sort
FROM    cte_BOM
ORDER BY Sort;
```

The anchor member, which is used to retrieve the top level products is in blue. The recursive member` is colored green. The recursive CTE is invoked by the portion colored black.

Here is the Bill of Materials results:

Some things that important to note with this query are:

1. You'll notice several columns are explicitly cast as VARCHAR(100). This to ensure the datatypes are the same for corresponding columns within the anchor and recursive query definition.
2. The level is incremented by one upon each recursive query definition call. The anchor query starts with 1, and incremented from there.
3. The level is used to help indent sub-assemblies within products. We use the REPLICATE string function as a helper.

4. A sort field is constructed from using product and sub-assembly names. If this wasn't done, then there wouldn't be a convenient way to order the results by sub-assembly.

The sort field is a key concept. When a recursive CTE runs remember it first gathers all the top level product IDs. Then it uses these to collect products used in those assemblies. This process repeats until no further product ID are found.

Because of this, rows are returned one level at a time. This is called a breadth-first search. For a BOM we typically want to drill into a sub-assembly and then is component parts or sub-assemblies. This is called a depth-first search.

In order to simulate this, we construct the sort field. The sort field is constructing the "path" of assemblies (think sub folders on your computer). When these paths are sorted, the items appear in depth-first-search order.

Conclusion

Common table expressions come in handy when you need to simplify a query. Though some would contend that using recursive CTEs doesn't lend to this goal, as they can be conceptually difficult to understand, they provide a means for an elegant solution.

There are many types of queries which are difficult to solve without recursive CTE's. Querying hierarchical data is one of these. Of course you can write looping structures to achieve the same goal, but if you don't have a means to write and execute a stored procedure, then recursive CTE's allow you to do so using a SELECT statement alone.