

Recursive CTEs

The following is chapter 4 from my Common Table Expressions book in its entirety.

Recursive CTEs

Have you ever solved a maze puzzle on paper, or had the opportunity to visit a walk-through maze? You may have worked your way through by randomly turning left and right eventually finding the end. Perhaps tackling the maze more systematically might have been your strategy, adopting a decision-making process to determine your route. That decision process could be to take every left turn until you hit a dead end, then back up and take the next option to the right.

You can think of that as a simple set of rules, once you have entered the maze:

1. Proceed until an intersection or a dead end.
2. If it is a dead end, abort this path and return to the previous intersection.
3. At an intersection, follow the left-most turn and start this process again for that path.
4. After getting to the left-most turn, take the next turn to the right and repeat the process for that path, following the same process until you run out of turns.
5. If you find the end of the maze, stop.

This concept of working your way through the maze is a sample of recursion where at each turn in an intersection you are starting over with the exact same set of steps that you used to get there from the previous intersection.

In the previous example, step 3, which includes “start the process again for that path” is where we consider the set of rules that are calling you to start over at the new level. What if we were to write a query that used the following logic, similar to the maze path rules, in that at some point in the

process it is repeating based on the previous steps:

1. Select some starting set of data from table A.
2. Join that starting set of data to table A.
3. For the results from step 2, join that to Table A.
4. Repeat until there are no more items in the join.

Of course for this to work we would need the right set of relational data. For instance, the data that we used in the samples in the previous chapters for departments only uses two levels. Now that we are talking about recursion we will want to consider deeper levels of department categories.

Consider the following partial department outline:

- Camping
 - Backpacks
 - Cooking
 - Tents
 - 1 Person
 - 2 Person
 - Backpacking
 - Mountaineering
 - Family Camping
 - 3 Person
 - 4 Person
- Cycle
- Fitness

The first level has three departments; Camping, Cycle and Fitness. The second level, under Camping, has three sub departments; Backpacks, Cooking and Tents. The third level has four sub-departments; 1, 2, 3 and 4 person tents, and the fifth level has three sub-departments of Backpacking, Mountaineering, and Family Camping.

The query logic for this tree is as follows:

1. Select the top level departments into a result set.
2. Join the existing result set back to the original table to find the next departments level.
3. Repeat step 2 until there are no new results added.

From a process flow perspective (with no programming) we could probably walk through that logic description and draw out the same department tree as shown previously. But how is this done in T-SQL?

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>.

What is Recursion?

Recursion is a programming concept where a divide and conquer approach is taken. In divide and conquer the division is generally not equal. For instance, being asked to add up all the numbers from 1 to 10 could be done like this:

$10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1$

In an iterative approach, add 10 to 9, getting 19, then add 8 to that for 27, then add 7 for 34, then 6 for 40, then 5... until finishing with 55 as the total. The recursive approach would be to take 10 and add it to the sum of the numbers 1 to 9.

The sum of the numbers from 1 to 10 is 55:

$55 = 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1$
 $55 = 10 + (\text{sum of numbers 1 to 9})$
 $55 = 10 + (9 + (\text{sum of numbers 1 to 8}))$
 $55 = 10 + (9 + (8 + (\text{sum of numbers 1 to 7})))$

Eventually we get to:

$55 = 10 + (9 + (8 + (7 + (6 + (5 + (4 + (3 + (2 + 1))))))))$

In each step of the process we are dividing the problem, or breaking off a small chunk, and then passing the rest of the work on to the next step. That is how recursion works.

In the first example, if asked to sum the numbers from 1 to 10, and we already knew the answer of summing the numbers from 1 to 9, then the answer would be very simple to do. Just add 10 to 45 and get the result of 55, where 45 is the sum of the numbers from 1 to 9.

To add the numbers recursively with a programming language like C or C# the code may look something like this:

```
function sumParts(int n)
{
    int returnValue1;
    if n > 1
        returnValue = n+ sumParts(n - 1);
    else
        returnValue = n;
```

```
    return returnValue;  
}
```

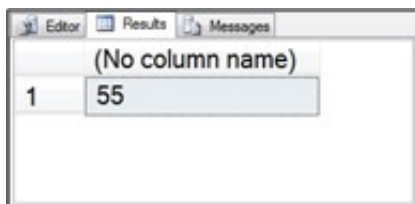
Part-way through the function, it ends up calling itself. This act of a function calling itself is what recursion is in a traditional function based implementation.

With SQL Server recursion it could be done in a similar way using a scalar valued function that calls upon itself. That would look something like the following:

```
CREATE FUNCTION dbo.SumParts (@Num integer)  
RETURNS integer  
AS  
BEGIN  
    DECLARE @ResultVar as integer = 0;  
    IF (@Num > 0)  
        SET @ResultVar = @Num + dbo.SumParts(@Num - 1);  
    RETURN @ResultVar;  
END  
GO
```

Calling on this function and passing in a value of 10 will get back 55. This would produce the output seen in Figure 4.1.

```
select dbo.SumParts(10);
```



	(No column name)
1	55

Figure 4.1 Running the SumParts(10) function shows a value of 55.

An alternative to that would be to implement the same thing using a recursive CTE.

Recursive CTE Syntax

The only variance from the standard CTE syntax to make a CTE recursive is it needs to be able to use the UNION ALL set operation to join a starting query (anchor query) with a second query (recursive query) that calls the CTE itself.

The following example will build a CTE query that recursively calculates results similar to the SumParts function, but without using any functions. In this example there will be no tables involved in the query. Instead, the CTE will be building on the initial query and doing mathematical calculations. After that we will walk through a recursive CTE query processing rows from a table.

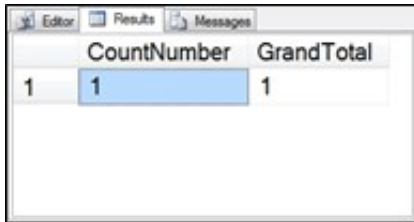
Anchor Query

The anchor query is the part of the CTE that starts off the recursive process. This is the set of data that will be passed into the recursive section. The anchor query can be written independent of the CTE, and when completed, added into the CTE.

The recursive part builds on the anchor so be sure to get the anchor query functioning correctly before moving on to the recursive query. The following query is the anchor or the starting point for the recursive CTE. In this example the anchor simply sets up a query that has two columns of output. The first column called CountNumber is used just to keep track of the number of levels through the recursion, and the GrandTotal column will be used to build up the total of all the

numbers as the query is called recursively.

```
SELECT 1 as CountNumber, 1 as GrandTotal;
```

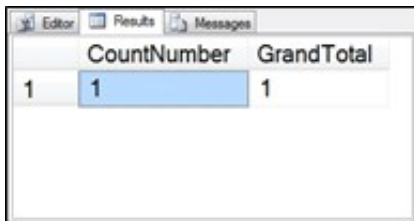


CountNumber	GrandTotal
1	1

Figure 4.2 Output from the preliminary anchor query.

Next, the anchor query is placed into a CTE, which when run produces the same output as the anchor query did alone. This does however need to be in a CTE in order to get to the recursive step.

```
;WITH SumPartsCTE AS  
(  
    SELECT 1 as CountNumber, 1 as GrandTotal  
)  
SELECT *  
    FROM SumPartsCTE;
```



CountNumber	GrandTotal
1	1

Figure 4.3 Output from the preliminary anchor query.

Recursive Query

Now that the anchor query is complete, we can build the recursive query by adding to the anchor query. The recursive query allows CTEs to do something that other queries can't do, reference itself. The recursive query will be called for each result that the CTE generates.

READER NOTE: *The next step will not run by itself and is there to show what this query looks like when it is partially done.*

The anchor query is setting count number to 1 and grand total to one for the first record as the common table expression. To calculate the sum of parts for 2 we need to add 1 to the count number, then add that value to the grand total, which should produce 3 as the sum of the parts for 2.

To write that in T-SQL the query might look something like this:

```
SELECT CountNumber + 1,  
       GrandTotal + CountNumber + 1
```

Where do the existing CountNumber and existing GrandTotal records come from? They can come from the existing CTE, if the CTE is referenced recursively in the FROM clause.

```
SELECT CountNumber + 1,  
       GrandTotal + CountNumber + 1
```


FROM SumPartsCTE

Next, add a UNION ALL statement into the existing CTE and add the recursive part of the query, which should run fine, but does it?

```
;WITH SumPartsCTE AS  
(  
    SELECT 1 as CountNumber, 1 as GrandTotal  
    UNION ALL  
    SELECT CountNumber + 1,  
           GrandTotal + CountNumber + 1  
    FROM SumPartsCTE  
)  
SELECT *  
FROM SumPartsCTE;
```

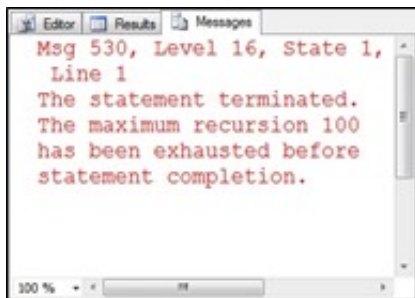
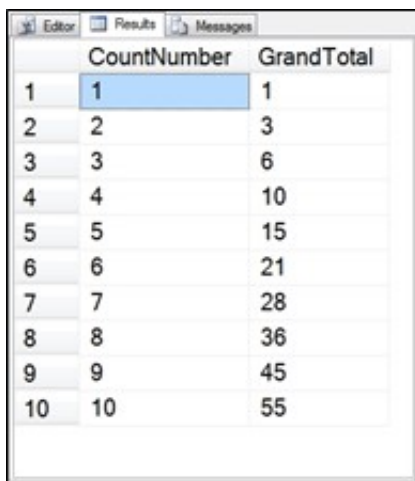


Figure 4.4 Maximum recursion exhausted error.

When the previous query is run it should show the maximum recursion exhausted error message. This means that the recursive query was run, and continued for 100 levels deep. The way to fix this is to specify a termination point in the WHERE clause so the recursion does not run into the previous error. In this example, the limit of 10 is specified in order to calculate the sum of parts for 10.

```
;WITH SumPartsCTE AS  
(  
    SELECT 1 as CountNumber, 1 as GrandTotal  
    UNION ALL  
    SELECT CountNumber + 1,  
           GrandTotal + CountNumber + 1  
    FROM SumPartsCTE  
    WHERE CountNumber < 10  
)  
SELECT *  
FROM SumPartsCTE;
```



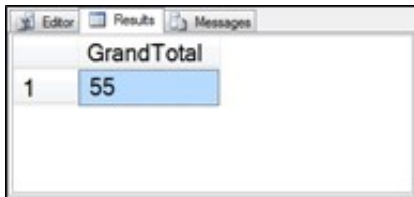
	CountNumber	GrandTotal
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15
6	6	21
7	7	28
8	8	36
9	9	45
10	10	55

Figure 4.5 Results from the recursive CTE.

In the result set, the sum of parts is calculated in the GrandTotal column for all numbers up to and including 10, but what we were looking for was the sum of parts for 10. We can grab just the final result by adding a WHERE clause to the query outside of the CTE. If the WHERE CountNumber = 10 clause was added inside the CTE query in place of the WHERE CountNumber

```
;WITH SumPartsCTE AS  
(  
    SELECT 1 as CountNumber, 1 as GrandTotal  
    UNION ALL  
    SELECT CountNumber + 1,  
           GrandTotal + CountNumber + 1  
    FROM SumPartsCTE  
    WHERE CountNumber < 10  
)  
SELECT *  
FROM SumPartsCTE  
WHERE CountNumber = 10;
```

```
UNION ALL
SELECT CountNumber + 1,
       GrandTotal + CountNumber + 1
FROM SumPartsCTE
WHERE CountNumber < 10
SELECT GrandTotal
FROM SumPartsCTE
WHERE CountNumber = 10;
```



	GrandTotal
1	55

Figure 4.6 Results after changing the WHERE clause.

Now the recursive CTE produces the same results as the original function did.

Recursive Rows

This is where CTEs get really interesting. The advantage that a CTE has over the scalar-valued function is that a CTE function can pass entire rows to the recursive part, rather than just a fixed number of input parameters like a scalar-valued function.

Let's start by breaking out each of the logical steps in order to walk through the hierarchy of departments from the Departments table.

The logic that will be used in this example will be:

1. Select the top level departments.
2. For departments at the current level, select the sub-departments of that department.
3. Repeat step 2 until there are no departments left.

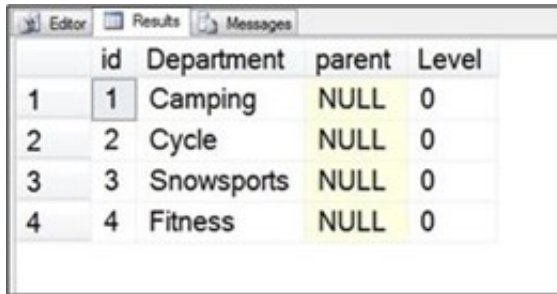
These steps could be implemented without recursion if we knew the number of levels. We could write one query to return each level, and combine all the results into one list with the UNION ALL set operator. What recursion allows us to do is to implement a generic query that will return the entire hierarchy independent of the number of levels. This type of CTE query is made up of an anchor query that starts the process for the first level and a recursive query which repeats the process for each level.

Anchor Query

For this example, the anchor query will have a column included called Level. The level starts at 0 and gets added to for each level through the hierarchy. To start with, the anchor query is only asking for the top level departments, the level will start at 0. This example shows the anchor query:

```
SELECT id, Department, parent, 0 AS Level
FROM Departments
WHERE parent is NULL;
```

When run, the anchor query produces the following results, showing the four top level departments. There are more departments to this store than shown in the outline at the beginning of the chapter.



The screenshot shows a SQL Server interface with three tabs: Editor, Results, and Messages. The Results tab is active, displaying a table with four columns: id, Department, parent, and Level. The table contains four rows of data, all with a parent value of NULL and a Level of 0.

	id	Department	parent	Level
1	1	Camping	NULL	0
2	2	Cycle	NULL	0
3	3	Snowsports	NULL	0
4	4	Fitness	NULL	0

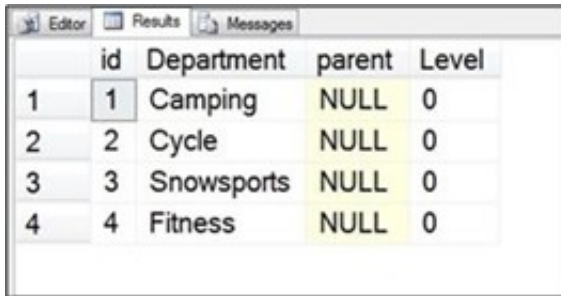
Figure 4.7 Results of the anchor query prior to adding it to the CTE.

Now that we have our anchor query, let's take it and drop it into a CTE. Start the CTE like any other CTE, using the WITH keyword and a name for the CTE, include the anchor query, and then SELECT from the CTE.

```
;WITH departmentsCTE AS  
(  
    SELECT id, Department, parent, 0 as Level  
    FROM Departments  
    WHERE parent is NULL  
)  
SELECT *  
FROM departmentsCTE
```

In this example the anchor query is selecting the id, Department, Parent and Level fields. The Level field is used later to determine how deep we are in the hierarchy.

When the CTE query is run the output will be just the same as the output from the stand-alone anchor query.



	id	Department	parent	Level
1	1	Camping	NULL	0
2	2	Cycle	NULL	0
3	3	Snowsports	NULL	0
4	4	Fitness	NULL	0

Figure 4.8 CTE query output is the same as that of the stand-alone anchor query.

With the anchor query in place we can see that the four top level department categories are shown. Think of this as the starting point, next the query needs to grab the child level of each department.

Recursive Query

Using the same query that we started with in the anchor query section, we now add a UNION ALL to the recursive part of the query. The recursive part of the query takes the CTE output and joins it back to the Departments table to get the next level of the department hierarchy. Think of the recursive query as simply taking the results of the anchor and joining them to the next level. The recursive part could look like the following:

```
SELECT d.Id, d.Department, d.Parent,  
      departmentsCTE.Level + 1 as Level  
FROM Departments d  
INNER JOIN departmentsCTE  
      ON DepartmentCTE.id = d.Parent
```

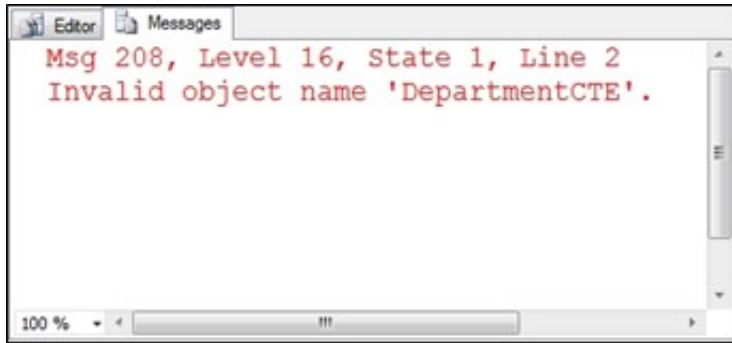


Figure 4.9 Error message from the recursive part outside of a CTE.

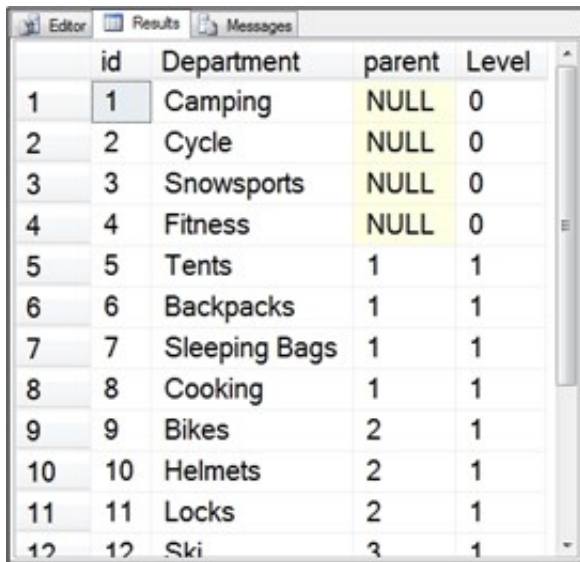
If we attempt to run that query outside of the CTE it isn't going to work because it references the departmentsCTE which would be out of scope. We will add the recursive query into the CTE using the UNION ALL.

```
;WITH departmentsCTE AS
(
    SELECT id, Department, parent, 0 as Level
    FROM Departments
    WHERE parent is NULL
    UNION ALL -- and now for the recursive part
    SELECT d.id, d.Department, d.Parent,
           DepartmentsCTE.Level + 1 as Level
    FROM Departments d
    INNER JOIN departmentsCTE
    ON DepartmentCTE.id = d.Parent
)
SELECT *
FROM departmentsCTE
ORDER BY parent;
```

The UNION ALL set operator is used to combine the result of the recursive query with the original results. Notice that in the INNER JOIN line we are joining to departmentsCTE, which is the name of the CTE that we are currently writing. Here we also include an ORDER BY clause so that the sub-

departments end up grouped together.

When this query runs it produces the following output:



	id	Department	parent	Level
1	1	Camping	NULL	0
2	2	Cycle	NULL	0
3	3	Snowsports	NULL	0
4	4	Fitness	NULL	0
5	5	Tents	1	1
6	6	Backpacks	1	1
7	7	Sleeping Bags	1	1
8	8	Cooking	1	1
9	9	Bikes	2	1
10	10	Helmets	2	1
11	11	Locks	2	1
12	12	Ski	3	1

Figure 4.10 Output from the recursive query.

The query shows just two levels, which, at this point could have been accomplished with a non-CTE query like this:

```
SELECT d2.id, d2.Department, d2.Parent,  
       case when d2.Parent is NULL  
       then 0 else 1 end as Level  
FROM Departments d1  
RIGHT JOIN Departments d2 on d1.id = d2.parent;
```

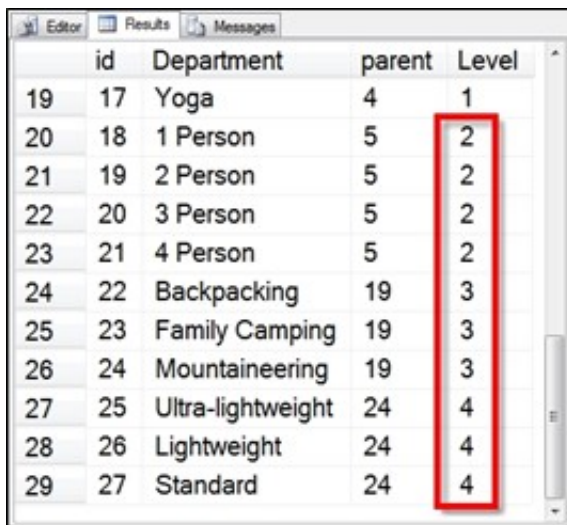
Where it gets interesting is if our department hierarchy has more than two levels. The non-CTE

version breaks down if there are more than a couple of levels to the hierarchy.

READER NOTE: Please run the *FillMoreDepartments* stored procedure in the *JProCo* database as shown here. This will add additional department hierarchy levels into the *Departments* table.

USE [JProCo];
execute FillMoreDepartments;

If there are more than two levels in the hierarchy, the non-CTE version of the query breaks down, but the recursive CTE query shows all the levels are with the correct level number, as shown here:



	id	Department	parent	Level
19	17	Yoga	4	1
20	18	1 Person	5	2
21	19	2 Person	5	2
22	20	3 Person	5	2
23	21	4 Person	5	2
24	22	Backpacking	19	3
25	23	Family Camping	19	3
26	24	Mountaineering	19	3
27	25	Ultra-lightweight	24	4
28	26	Lightweight	24	4
29	27	Standard	24	4

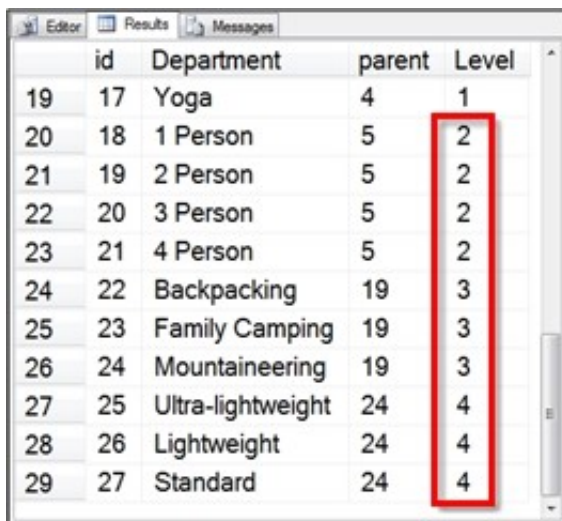
Figure 4.11 Recursive query showing more than the original 2 levels.

The non-CTE query could be expanded to include a second level, third level, a fourth level, and beyond, but each level would require more work and more duplicate code.

MAXRECURSION

An additional part of the recursive query is specifying the maximum depth the recursion will go before the query is aborted. The maximum level of recursion in SQL (if not specified) defaults to 100 levels. SQL Server generates an error if the query goes beyond that maximum level of recursion. We encountered this error in Figure 4.4. The maximum recursion can be overridden if a recursion deeper than 100 levels is needed.

The level is measured as the number of levels through the recursive path. Depending on how the query is written it may go deep or it may go wide. Take a look back to the output from the recursive query:



	id	Department	parent	Level
19	17	Yoga	4	1
20	18	1 Person	5	2
21	19	2 Person	5	2
22	20	3 Person	5	2
23	21	4 Person	5	2
24	22	Backpacking	19	3
25	23	Family Camping	19	3
26	24	Mountaineering	19	3
27	25	Ultra-lightweight	24	4
28	26	Lightweight	24	4
29	27	Standard	24	4

Figure 4.12 Output from the recursive departments query.

Here we see that it is showing rows 19 to 29 and that 29 is the total number of rows, but the level as it is shown in the query is the recursion level which is only 4 levels. In this example, level 0 is supplied by the anchor query, and the 4 recursive levels are level 1, 2, 3 and 4.

The syntax to set the maximum recursion is to use the **OPTION** keyword with **MAXRECURSION** at the end of the query.

OPTION (MAXRECURSION 4);

For instance, if we add that to our recursive query it would look like this:

```
;WITH departmentsCTE AS
(
    SELECT id, Department, parent, 0 as Level
      FROM Departments
     WHERE parent is NULL
    UNION ALL -- and now for the recursive part
    SELECT d.id, d.Department, d.Parent,
           DepartmentsCTE.Level + 1 as Level
      FROM Departments d
     INNER JOIN departmentsCTE
       ON DepartmentCTE.id = d.Parent
)
SELECT *
  FROM departmentsCTE
 ORDER BY parent
OPTION (MAXRECURSION 4);
```

Setting this query to have a **MAXRECURSION** of 4 is safe and it runs just fine, but if we adjust the **MAXRECURSION** to be 2, which is less than the number of levels returned, then we will see an error.

OPTION (MAXRECURSION 2);

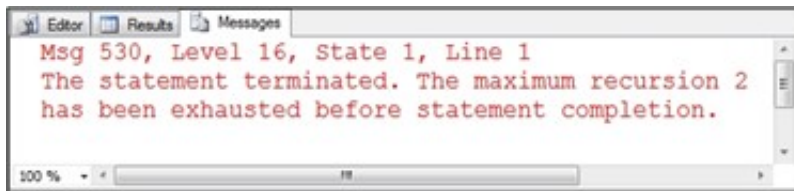


Figure 4.13 Result of recursion level being set to less than the number of levels returned.

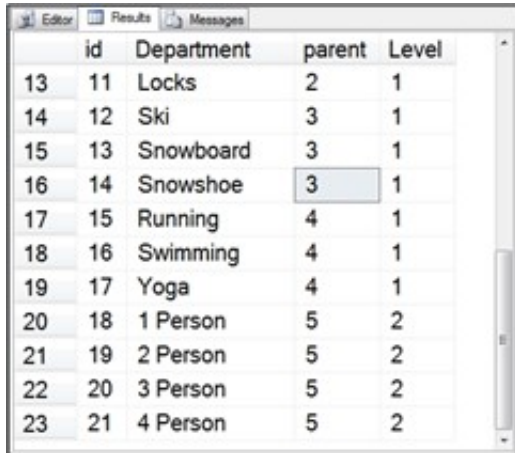
Using the default MAXRECURSION option is a good safety measure for most queries. Recursive queries that need to go beyond 100 levels deep will need to override the default. Think of the max recursion error message as an indicator that the query in the CTE hasn't properly terminated the recursion.

In the departmentsCTE example there were only 4 levels to the hierarchy after the anchor, but some queries could go much deeper. Take the example of a social networking site, where someone may have several friends, and they all have several friends, and on and on. From there, it is possible, that if a query was written to follow those relationships, the recursion could go millions of levels deep. But if we were trying to find close relationships we could terminate the recursion without an error by using a WHERE clause on the recursive query.

If we only wanted the recursion to go 2 (anchor + 2) levels deep, and we wanted to avoid the maximum recursion error message then we could terminate it like this:

```
;WITH departmentsCTE AS
(
    SELECT id, Department, parent, 0 as Level
    FROM Departments
    WHERE parent is NULL
    UNION ALL -- and now for the recursive part
    SELECT d.id, d.Department, d.Parent,
           DepartmentsCTE.Level + 1 as Level
    FROM Departments d
```

```
INNER JOIN departmentsCTE
  ON DepartmentCTE.id = d.Parent
WHERE Level )
SELECT *
  FROM departmentsCTE
 ORDER BY parent;
```



	id	Department	parent	Level
13	11	Locks	2	1
14	12	Ski	3	1
15	13	Snowboard	3	1
16	14	Snowshoe	3	1
17	15	Running	4	1
18	16	Swimming	4	1
19	17	Yoga	4	1
20	18	1 Person	5	2
21	19	2 Person	5	2
22	20	3 Person	5	2
23	21	4 Person	5	2

Figure 4.14 Query results with Level set to be less than 2.

In the results form Figure 4.14 it is only 2 levels deep and does not cause any max recursion errors. To specify no maximum on a recursive CTE, use the MAXRECURSION 0 option. This can be dangerous if a query gets out of control. A whole lot of memory and processor time could get taken up by a run-away recursive query.

```
OPTION (MAXRECURSION 0);
```

The MAXRECURSION option has a maximum input value of 32,767, but that doesn't mean that recursion is limited to 32,767 levels. Recursion can be deeper than that, just specify 0 for MAXRECURSION.

Bring it All Together

At this point we have seen a recursive CTE, but how does it work? Let's take a look at how SQL Server executes a CTE.

- Run the anchor query, save the output as R1.
- Pass R1 to the recursive query, save output into R2.
- Pass R2 to the recursive query, save output as R3.
- Pass R3 to the recursive query, save output as R4.
- ...
- Pass Rn to the recursive query, save output as Rn+1.
- Continue until Rn returns no output.

Even though the recursive part of the query is only listed once in the CTE, it behaves as though it was duplicated as many times as the depth of recursion is in the query.

Advanced Recursion

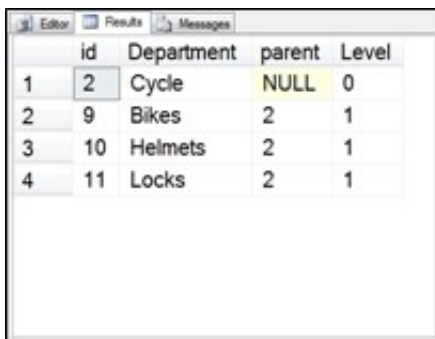
Multiple anchor queries and multiple recursive queries can be accomplished with recursive CTEs.

Using Multiple Anchor Queries

One example of multiple anchor queries would be if we wanted to look up sub-departments for two or more top level departments, but not for all of them. For instance if we wanted to only display the departments and sub-departments for Fitness and Cycling it could be done with multiple anchor queries.

To start with, we could use the departmentsCTE query from earlier in the chapter, but filter on just the Cycle department which has an id of 2. This displays the Cycle department and all its sub-departments.

```
;WITH departmentsCTE AS
(
    SELECT id, Department, parent, 0 as Level
      FROM Departments
     WHERE id = 2
    UNION ALL
    SELECT d.id, d.Department, d.Parent,
           DepartmentsCTE.Level + 1 as Level
      FROM Departments d
     INNER JOIN departmentsCTE
        ON DepartmentCTE.id = d.Parent
)
SELECT *
  FROM departmentsCTE
 ORDER BY parent;
```



	id	Department	parent	Level
1	2	Cycle	NULL	0
2	9	Bikes	2	1
3	10	Helmets	2	1
4	11	Locks	2	1

Figure 4.15 Cycle department with Cycle sub-departments.

Next, copy the anchor query and change the WHERE clause to filter on Department 4, which is Fitness.

```
SELECT id, Department, parent, 0 as Level
FROM Departments
WHERE id = 4
```

Then, drop it into the CTE query with another UNION ALL. Notice that there are now two UNION ALL statements, one between the two anchors and another between the anchors and the recursive query.

```
;WITH departmentsCTE AS
(
    SELECT id, Department, parent, 0 as Level
    FROM Departments
    WHERE id = 2
    UNION ALL
    SELECT id, Department, parent, 0 as Level
    FROM Departments
    WHERE id = 4
    UNION ALL
    SELECT d.id, d.Department, d.Parent,
        DepartmentsCTE.Level + 1 as Level
    FROM Departments d
    INNER JOIN departmentsCTE
        ON DepartmentCTE.id = d.Parent
)
SELECT *
FROM departmentsCTE
ORDER BY parent;
```




The screenshot shows a SQL query results window with tabs for 'Editor', 'Results', and 'Messages'. The 'Results' tab is active, displaying a table with four columns: 'id', 'Department', 'parent', and 'Level'. The table contains eight rows of data representing a hierarchy of departments. The first two rows, 'Cycle' (id 2) and 'Fitness' (id 4), are highlighted in yellow and have a 'parent' value of NULL and 'Level' of 0. The subsequent rows are sub-departments: 'Bikes' (id 9, parent 2, Level 1), 'Helmets' (id 10, parent 2, Level 1), 'Locks' (id 11, parent 2, Level 1), 'Running' (id 15, parent 4, Level 1), 'Swimming' (id 16, parent 4, Level 1), and 'Yoga' (id 17, parent 4, Level 1).

	id	Department	parent	Level
1	2	Cycle	NULL	0
2	4	Fitness	NULL	0
3	9	Bikes	2	1
4	10	Helmets	2	1
5	11	Locks	2	1
6	15	Running	4	1
7	16	Swimming	4	1
8	17	Yoga	4	1

Figure 4.16 Cycle and Fitness departments including sub-departments.

This multiple anchor CTE could have been created with a single anchor using the IN keyword in the WHERE clause like this:

```
WHERE id in (2, 4)
```

This simpler query would produce the exact same results as the multiple anchors example. With recursive CTEs it is common to find multiple ways to achieve the same results.

Using Multiple Recursive Queries

Multiple recursive queries work similar to multiple anchor queries, just add additional queries into the CTE separated by a UNION ALL. In the multiple recursive query example there are multiple queries that access the CTE itself.

An example of this would be to create a query to follow someone's family tree. There is a table in the example database called Royalty which contains the parent child relationships about many of the recent British royal family.



	id	name	mother	father
1	1	George V, King of England	NULL	NULL
2	2	Mary, Princess of Teck	NULL	NULL
3	3	George VI Windsor, King of England	2	1
4	4	Claude George Bowes-Lyon	NULL	NULL
5	5	Nina Cecilia Cavendish-Bentinck	NULL	NULL
6	6	Elizabeth Angela Marguerite Bowes-Lyon	5	4
7	7	William George I of the Hellenes	NULL	NULL
8	8	Olga Konstantinovna Romanova	NULL	NULL
9	9	Louis Alexander von Battenburg	NULL	NULL
10	10	Victoria von Hessen und bei Rhein	NULL	NULL
11	11	Andreas, Prince of Greece	8	7
12	12	Alice, Princess of Battenbugr	10	9
13	13	Phillip Mountbatten, Duke of Edinburgh	12	11
14	14	Elizabeth II Windsor, Queene of England	6	3
15	15	Charles Philip Arthur Windsor	14	13
16	16	Diana Frances (Lady) Spencer	18	19
17	17	William Arthur Phillip Windsor	16	15
18	18	Frances Ruth Burke Roche	NULL	NULL
19	19	Edward John Spencer	NULL	NULL

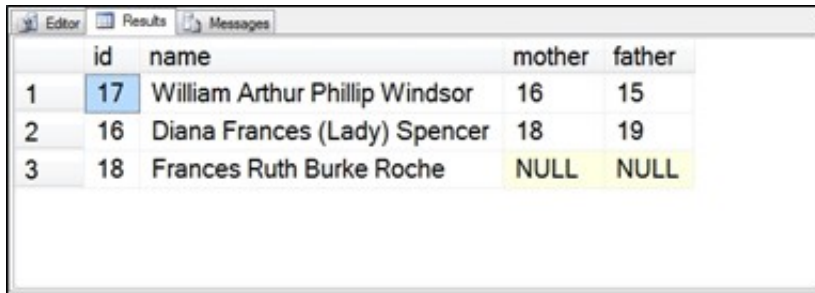
Figure 4.17 Contents of the Royalty table.

READER NOTE: The example we are going to use is not from a table in JProCo. Just follow along with the upcoming examples to understand the concepts.

If we wanted to create a query to trace Prince William’s maternal ancestors it would be straightforward with a CTE. We can see that Prince William has an id of 17 and a full name of “William Arthur Phillip Windsor”.

```
WITH RoyalTreeCTE AS
(
    SELECT * FROM Royalty WHERE id = 17
```

```
UNION ALL
SELECT r.*
  FROM Royalty AS r
  INNER JOIN RoyalTreeCTE AS rt ON rt.mother = r.id
)
SELECT *
FROM RoyalTreeCTE;
```



	id	name	mother	father
1	17	William Arthur Phillip Windsor	16	15
2	16	Diana Frances (Lady) Spencer	18	19
3	18	Frances Ruth Burke Roche	NULL	NULL

Figure 4.18 Prince William and his maternal ancestors from the CTE query.

Finding the paternal ancestors would be very similar. Instead of using the mother column, use the father column.

```
WITH RoyalTreeCTE AS
(
  SELECT * FROM Royalty WHERE id = 17
  UNION ALL
  SELECT r.*
    FROM Royalty AS r
    INNER JOIN RoyalTreeCTE AS rt ON rt.father = r.id
)
SELECT * FROM RoyalTreeCTE;
```



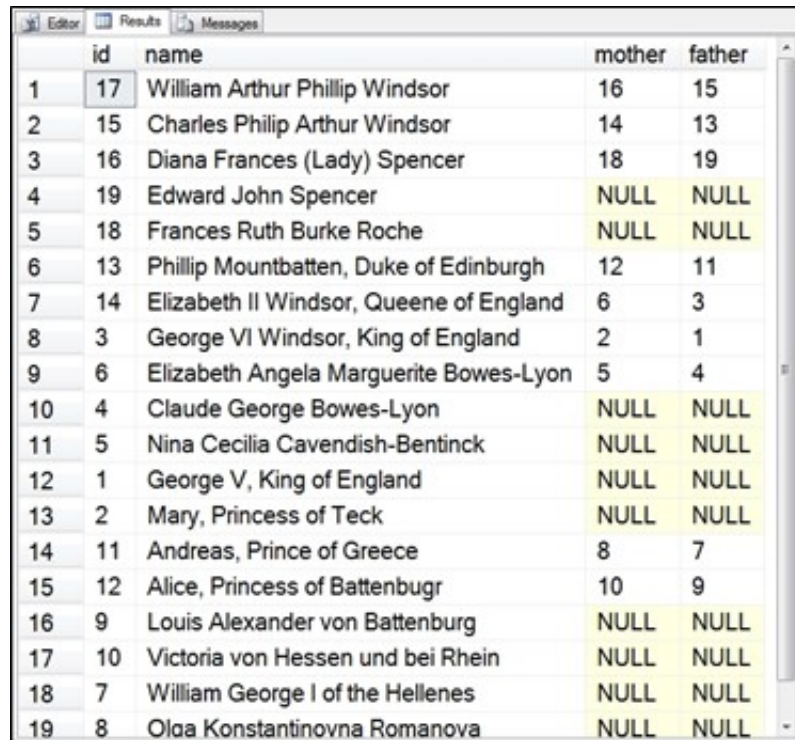
	id	name	mother	father
1	17	William Arthur Phillip Windsor	16	15
2	15	Charles Philip Arthur Windsor	14	13
3	13	Phillip Mountbatten, Duke of Edinburgh	12	11
4	11	Andreas, Prince of Greece	8	7
5	7	William George I of the Hellenes	NULL	NULL

Figure 4.19 Prince William and his paternal ancestors from the CTE query.

How can we write the query to find both the maternal and paternal sides of a family tree? Keep in mind that doing this will produce both the mother and father of each person in the tree.

WITH RoyalTreeCTE AS

```
(  
    SELECT *  
      FROM Royalty WHERE id = 17  
    UNION ALL  
    SELECT r.*  
      FROM Royalty r  
     INNER JOIN RoyalTreeCTE rt ON rt.father = r.id  
    UNION ALL  
    SELECT r.*  
      FROM Royalty r  
     INNER JOIN RoyalTreeCTE rt ON rt.mother = r.id  
)  
SELECT *  
  FROM RoyalTreeCTE;
```



	id	name	mother	father
1	17	William Arthur Phillip Windsor	16	15
2	15	Charles Philip Arthur Windsor	14	13
3	16	Diana Frances (Lady) Spencer	18	19
4	19	Edward John Spencer	NULL	NULL
5	18	Frances Ruth Burke Roche	NULL	NULL
6	13	Phillip Mountbatten, Duke of Edinburgh	12	11
7	14	Elizabeth II Windsor, Queene of England	6	3
8	3	George VI Windsor, King of England	2	1
9	6	Elizabeth Angela Marguerite Bowes-Lyon	5	4
10	4	Claude George Bowes-Lyon	NULL	NULL
11	5	Nina Cecilia Cavendish-Bentinck	NULL	NULL
12	1	George V, King of England	NULL	NULL
13	2	Mary, Princess of Teck	NULL	NULL
14	11	Andreas, Prince of Greece	8	7
15	12	Alice, Princess of Battenbugr	10	9
16	9	Louis Alexander von Battenburg	NULL	NULL
17	10	Victoria von Hessen und bei Rhein	NULL	NULL
18	7	William George I of the Hellenes	NULL	NULL
19	8	Olaa Konstantinovna Romanova	NULL	NULL

Figure 4.20 Prince William and his ancestors from the CTE query.

Summary

Recursion is one of the more useful reasons to use CTEs. The recursion is accomplished by specifying an anchor query to start the recursive process, and a recursive query that builds on the data from the anchor. What makes a CTE recursive is when the query inside the CTE references the CTE itself.

The maximum recursion of a query can control the depth that SQL Server will let a recursive query go. This is accomplished with the MAXRECURSION option at the end of the query.

CTEs can have multiple anchor queries and also have multiple recursive queries. The multiple

anchors or multiple recursive queries are separated with the UNION ALL keyword.

Points to Ponder - Recursive CTEs

1. A CTE is considered recursive when it references itself.
2. The MAXRECURSION option can be used to override the default maximum recursion level of 100. Setting MAXRECURSION to 0 indicates no maximum.
3. The part of the recursive CTE that starts the recursion is the anchor query.
4. There can be multiple anchor queries in a CTE using a UNION ALL.
5. The column data types of the recursive query in the CTE must match the data types of the columns in the anchor query.
6. The recursive part of the query can only refer to the CTE once. A self-join is not allowed in the recursive query.
7. Recursion stops when the recursive query returns no results.
8. There can be multiple recursive queries in a CTE using the UNION ALL operation to join their results.

Review Quiz - Chapter Four

1. For a CTE to be considered recursive it must:
2. Use the UNION ALL keywords.
3. Access more than one table.
4. Reference itself with a join.
5. Do both (a) and (c).
 1. How many recursive members can exist in a single recursive CTE? Choose all that are correct.
6. None
7. One
8. 5
9. Many
 1. Recursive CTEs have a limit to their levels of recursion. What is that limit?

- 10. 10
- 11. 100
- 12. 100 unless MAXRECURSION is set to something else.
- 13. It depends on the SQL Server configuration.

Answer Key

- 1. For a CTE to be considered recursive it must contain one recursive query. The recursive query is connected to the anchor query using the UNION ALL keywords (a). The recursive CTE must reference the CTE itself in a join (c). Answer (a) and answer (c) are both correct so the correct answer is (d) which is "Both (a) and (c)".
- 2. There can be many recursive members in a CTE, but for it to be considered recursive there must be at least one. The incorrect answer is (a), the correct answers are (b), (c) and (d).

There is no global configuration setting in SQL Server for the recursion level, so (d) is incorrect. (a) and (c) are not correct. The maximum number of recursion levels is set with the MAXRECURSION option, and it defaults to 100, and can be set to a specific value. Setting MAXRECURSION to 0 specifies no maximum. The correct answer is (c).