

# CTEs Getting to the Bottom of the Hierarchy

## *New T-SQL for You to Learn and Use*

Wayne Snyder, Mariner ([www.mariner-usa.com](http://www.mariner-usa.com))

Last time we looked at Common Table Expressions(CTEs), we covered nonrecursive uses. This time we will see how you can recurse through a hierarchy using recursive CTEs. If you want to learn more about SQL 2005, take a look at the **SQL Server 2005 training from Learnkey** ([www.Learnkey.com](http://www.Learnkey.com)). We go through the entire SQL engine to give you the training you need to be successful using SQL 2005.

If you haven't used a CTE, you should read the first article in this series first.

You can think of a recursive CTE (R-CTE) as a recursive query. Without CTEs, going through a hierarchy can be painful, often requiring temporary tables, and knowledge of the maximum depth of the hierarchy. R-CTEs can traverse a hierarchy very easily. We are going to use an R-CTE to traverse a hierarchy.

The basic syntax for a recursive CTE follows:

```
With CTE_Name (Column_Name_1, Column_Name_2 ... Column_Name_n )
As
(
    cte_Anchor_Member_Query_Definition
    UNION ALL or UNION EXCEPT OR INTERSECT
    Cte_Recursion_Query_Definition (Must refer to CTE_Name)
)

--TSQL which uses the CTE
```

Listing 1. Basic Syntax for recursive CTE

In the syntax above there are 2 queries combined with a union all. The first query is not recursive. It simply defines what is called the anchor members. The first query is executed once. If it returns no rows, the CTE exits. If the first query returns some rows, let's call them the "*current working set*". (This is my label, not Microsoft's.) The current working set must be referred to in the second query using the CTE\_Name. The second query is now executed. The result set from the second query becomes the "*current working set*". The second query is then executed again, repetitively, each time with the new working set from the prior execution, until no more rows are returned. Then all of the rows from all of the executions are returned from the CTE.

We wish to walk through the HumanResources.Employee table in AdventureWorks and return each EmployeeID, and his ManagerID. We need to either start at the top or the bottom of the hierarchy. It would be difficult to start at the bottom, because there is no

easy way to identify the rows at the bottom without traversing the entire hierarchy. It is easy however to start at the top. The top of the hierarchy is the row(s) for employees which have a ManagerID of Null.

```
USE AdventureWorks;
GO
SELECT EmployeeID, ManagerID, 0
FROM HumanResources.Employee e
WHERE ManagerID is NULL
GO
```

The query returns the result set below. These are the anchor members. I will explain the third column later.

	EmployeeID	ManagerID	(No column name)
1	109	NULL	0

Listing 1 - Anchor Members

Now let's put this into a CTE as below.

```
WITH EmpsCTE (empid, mgrid, level)
AS
(
SELECT EmployeeID, ManagerID, 0
FROM HumanResources.Employee e
WHERE ManagerID is NULL
)
SELECT * FROM EmpsCTE
GO
```

This CTE query returns the same result set as the original query. The only difference is that the last column is named level.

To make this a recursive CTE we need to add the recursive query. The recursive query must refer to the CTE\_name, in our case EmpsCTE. For the first recursion, we wish to find all of the employees whose manager is in the set of anchor members ("*current working set*"). The query below will do what we need.

```
SELECT EmployeeID, ManagerID, m.level + 1
FROM HumanResources.Employee e
INNER JOIN EmpsCTE AS m
ON e.ManagerID = m.empid
```

We simply join the original table back to the current working set of the CTE, looking for the employees whose ManagerID is equal to the EmpsCTE.empid. When we put this into the CTE, the first recursion will give us the employees who directly report to employee 109. This defines a new working set (the 2nd working set - WS2). The recursive query is then executed again, this time the EmpsCTE reference will refer to the WS2 working set. This recursion finds the employees at the next level down. This recursion continues until the recursive query returns no more rows.

Each recursion, then defines the level of the employee, which is the distance from the top employee. This also gives us the depth within the hierarchy for each employee. We do this by defining the level for the anchor member as 0. In the recursion, the level is defined as the previous level + 1 ( m.level + 1). This will allow you to find all employees who are 2 levels below a given employee, for instance.

Now let's put the whole thing inside the CTE.

```
WITH EmpsCTE (empid, mgrid, level)
AS
(
SELECT EmployeeID, ManagerID, 0
  FROM HumanResources.Employee e
 WHERE ManagerID is NULL
UNION ALL
SELECT EmployeeID, ManagerID, m.level + 1
  FROM HumanResources.Employee e
 INNER JOIN EmpsCTE AS m
   ON e.ManagerID = m.empid
)
SELECT * FROM EmpsCTE
GO
```

The results are listed below ( Listing 2.)

	empid	mgrid	level
1	109	NULL	0
2	6	109	1
3	12	109	1
4	42	109	1
5	140	109	1
6	148	109	1
7	273	109	1
8	268	273	2
9	284	273	2
10	288	273	2
11	290	288	3
12	285	284	3

Listing 2 - Recursive CTE Results  
(The list is truncated for brevity.)

If your use of this is well-defined and fixed, you might add the additional columns to the CTE that you might need - name for instance. If there are many columns, and many different uses, you might wish to leave the CTE with *only* the keys, to keep the width very narrow. You can then join the CTE results back to the original table to get whatever extra information you might need. In the example below, we get the Employee Name and Manager Name by joining back to Person.Contact

```
WITH EmpsCTE (empid, mgrid, level)
AS
(
```

```

SELECT EmployeeID, ManagerID, 0
  FROM HumanResources.Employee e
 WHERE ManagerID is NULL
UNION ALL
SELECT EmployeeID, ManagerID, m.level + 1
  FROM HumanResources.Employee e
     INNER JOIN EmpsCTE AS m
       ON e.ManagerID = m.empid
)
SELECT
  EmpsCTE.*,
  e.FirstName as EmpFirstName, e.LastName as EmpLastName ,
  m.FirstName as MgrFirstName, m.LastName as MgrLastName
  FROM EmpsCTE
 LEFT OUTER JOIN Person.Contact e
  ON EmpsCTE.empid = e.ContactID
 LEFT OUTER JOIN Person.Contact m
  ON EmpsCTE.empid = m.ContactID
GO

```

	empid	mgrid	level	EmpFirstName	EmpLastName	MgrFirstName	MgrLastName
1	109	NULL	0	Stephanie	Bourne	Stephanie	Bourne
2	6	109	1	Frances	Adams	Frances	Adams
3	12	109	1	James	Aguilar	James	Aguilar
4	42	109	1	Chris	Ashton	Chris	Ashton
5	140	109	1	Megan	Burke	Megan	Burke
6	148	109	1	Jared	Bustamante	Jared	Bustamante
7	273	109	1	Adrian	Dumitrascu	Adrian	Dumitrascu
8	268	273	2	Gary	Drury	Gary	Drury
9	284	273	2	John	Emory	John	Emory
10	288	273	2	Julie	Estes	Julie	Estes

Listing 3 - Joining the CTE to Other Tables.

We really don't want to copy the CTE all over the place, however. It would be better to define it once and reuse it. We could create a view, or a user-defined table-valued function. Since we are going to parameterize this, we will use a table-valued function.

Since our hierarchy could be quite large, and we know that many users will only want parts of the hierarchy, we will allow them to provide a ManagerID as a starting point, and retrieve the descendants for this node in the tree. This corresponds to people who report to this manager.

```

CREATE FUNCTION Descendants(@mgrID int)
  RETURNS TABLE AS
  RETURN (WITH EmpsCTE (empid, mgrid, level)
  AS
  (
  SELECT EmployeeID, ManagerID, 0
    FROM HumanResources.Employee e
   WHERE ManagerID = @mgrID
  UNION ALL
  SELECT EmployeeID, ManagerID, m.level + 1
    FROM HumanResources.Employee e

```

```

        INNER JOIN EmpsCTE AS m
            ON e.ManagerID = m.empid
    )
    SELECT * FROM EmpsCTE)
GO

```

You can then call this function, as in the following:

```

SELECT * FROM dbo.Descendants(109)
SELECT * FROM dbo.Descendants(284)
SELECT * FROM dbo.Descendants(273)
GO

```

You must provide a starting point ManagerID. However you could change the WHERE clause to allow null as the parameter which would include the entire hierarchy. (I will leave that as an exercise for you, but it shouldn't be difficult.)

Now you can easily access this CTE from the UDF to join:

```

SELECT
    Descendants.*,
    e.FirstName as EmpFirstName, e.LastName as EmpLastName ,
    m.FirstName as MgrFirstName, m.LastName as MgrLastName
FROM Descendants(109)
LEFT OUTER JOIN Person.Contact e
    ON Descendants.empid = e.ContactID
LEFT OUTER JOIN Person.Contact m
    ON Descendants.empid = m.ContactID

```

If you want only direct reports you may add "WHERE Descendants.Level = 1".

This opens the door for you to create a whole system of hierarchy searching functions. You might create a CTE which finds the ancestors of a particular employee, one which returns only the leaves of the hierarchy, or those employees which are n levels away from a particular manager.

Now it's up to you!

See you next time!

Wayne Snyder is a long time Learnkey author ([www.learnkey.com](http://www.learnkey.com)). Learnkey's SQL 2000 Administration Series won the SQL Server Magazine's Best Web Based Training in 2004. He is an author for The SQL Server Standard and SQL Server Magazine, an MCT, SQL Server MVP, and Managing Consultant for Mariner([www.mariner-usa.com](http://www.mariner-usa.com)). Wayne has worked with SQL since its first release, and currently focuses on SQL Business Intelligence.