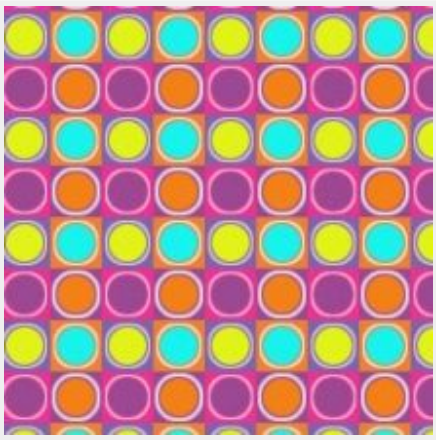
Non Recursive CTEs Explained and Why to Use Them



Introduction to Non Recursive CTEs

In this article we explore non recursive CTEs (Common Table Expressions). This is a broad class, and basically covers every form of CTEs except those that call themselves. This other class is called the recursive CTEs; they are covered in the next article.

If you're unfamiliar with CTEs 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 non recursive CTEs 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. You can get started using these free tools using my Guide *Getting Started Using SQL Server*.

WITH Clause

The WITH keyword is used when you start defining your common table expressions. It is used for both recursive CTEs as well as non recursive CTEs.

CTEs are table expressions, meaning they return a temporary result that can be used in the scope of an SELECT, INSERT, UPDATE, DELETE, or APPLY statement.

Here is generalized example of WITH:

```
WITH TableExpressionName (Column1, Column2, …, ColumnN)
AS
(Query Definition)
```

The CTE has two parts. The first part defines the name of the CTE and it the columns contained within it. This is the table expression's definition. The second part is the query definition. This is the SELECT statement used to populate the expression with rows.

You can define more than one CTE in a statement. You see an example of this further along in this article.

When writing the *query definition* keep in mind that the following cannot be used within it:

- ORDER BY, unless you also use as TOP clause
- INTO
- OPTION clause with query hints
- FOR BROWSE

For more information regarding the WITH keyword, refer to the MSDN documentation.

Reasons to use non recursive CTEs:

There are many reasons to use non recursive common table expressions. They include:

- **Readability** – Non recursive CTEs can make your query easier to read by organizing complex code, and eliminating the need to repeat complicated expressions.
- **Substitute for a View** – Views are great for encapsulating query logic and promoting reuse, but there are times when you you're either unable to create a view due to permissions, or the view would only be used in one query.
- **Limitations** – Overcome SELECT statement limitations, such as referencing itself (recursion), or performing GROUP BY using non-deterministic functions.

- **Ranking** – Whenever you want to use ranking function such as ROW_NUMBER(), RANK(), NTILE() etc.

Let's look at each of these reasons, in turn, using examples. By the time we're done, you'll see how to use one or more non recursive CTEs in a statement to work with joins. Also, you'll see how you can replace a correlated subquery with a two non recursive CTEs.

Readability

As queries get larger is can become really difficult to understand how they work. In my mind, readability doesn't mean the query has less lines. Instead, it means that it is understandable to you and others. I think CTEs help improve readability several ways.

They help separate out query logic. If you are joining two complex queries, you can use non recursive CTEs to separate out the complexity of the queries from the actual join. This not only helps in debugging, as you can independently run the query definition to test it, but you can more easily identify the parts used to do the join.

Also, CTEs improve readability by eliminating repeating expressions. There are situations where you want to display an expression and then sort by it. In SQL server, the expression has to be repeated in both the SELECT clause and ORDER BY; however, if you use a CTE, this isn't the case.

Readability such as Reference the resulting table multiple times in the same statement.

For the first couple of examples, we're going to assume our original query is the following:

```
SELECT   Substring(Person.LastName,1,1) + ' ' + Person.FirstName as
SortValue,
         Employee.NationalIDNumber,
         Person.FirstName,
         Person.LastName,
         Employee.JobTitle
FROM     HumanResources.Employee
         INNER JOIN
         Person.Person
         ON HumanResources.Employee.BusinessEntityID =
person.BusinessEntityID
WHERE    Person.LastName >= 'L'AND
         (Employee.JobTitle Like 'Marketing%' OR
```

```
                                    Employee.JobTitle Like 'Production%')
ORDER BY Substring(Person.LastName,1,1) + ' ' + Person.FirstName
```

We'll now take this statement show you how it is easier to read and maintain using CTEs. Of course our example isn't that complex, so it is still pretty readable, but I think as we work through the solution, it will help you see how CTEs can really help you in real world situations where your queries can be thirty or more lines long!

CTE joined to normal table

The first thing we can do is move the query used to retrieve person rows in to a CTE as so

```
WITH Employee_CTE (BusinessEntityID, NationalIDNumber, JobTitle)
AS       (SELECT BusinessEntityID,
                 NationalIDNumber,
                 JobTitle
          FROM   HumanResources.Employee
          WHERE  (Employee.JobTitle LIKE 'Marketing%'
                  OR Employee.JobTitle LIKE 'Production%'))
SELECT  Substring(Person.LastName, 1, 1) + ' ' + Person.FirstName as
SortValue,
         Employee_CTE.NationalIDNumber,
         Person.FirstName,
         Person.LastName,
         Employee_CTE.JobTitle
FROM     Employee_CTE
         INNER JOIN
         Person.Person
         ON Employee_CTE.BusinessEntityID = Person.BusinessEntityID
WHERE    Person.LastName >= 'L'
ORDER BY Substring(Person.LastName, 1, 1) + ' ' + Person.FirstName;
```

Notice that the CTE is defined to return three columns: BusinessEntityID, NationalID, and JobTitle.

Also, you see we've pulled in the filtering criterial into the CTEs query definition. The overall query is still a bit messy, but hopefully you're starting to see how CTEs can help to separate various query operations to make queries easier to read.

The next thing we could do to simplify the query is to create another CTE to handle Person table rows.

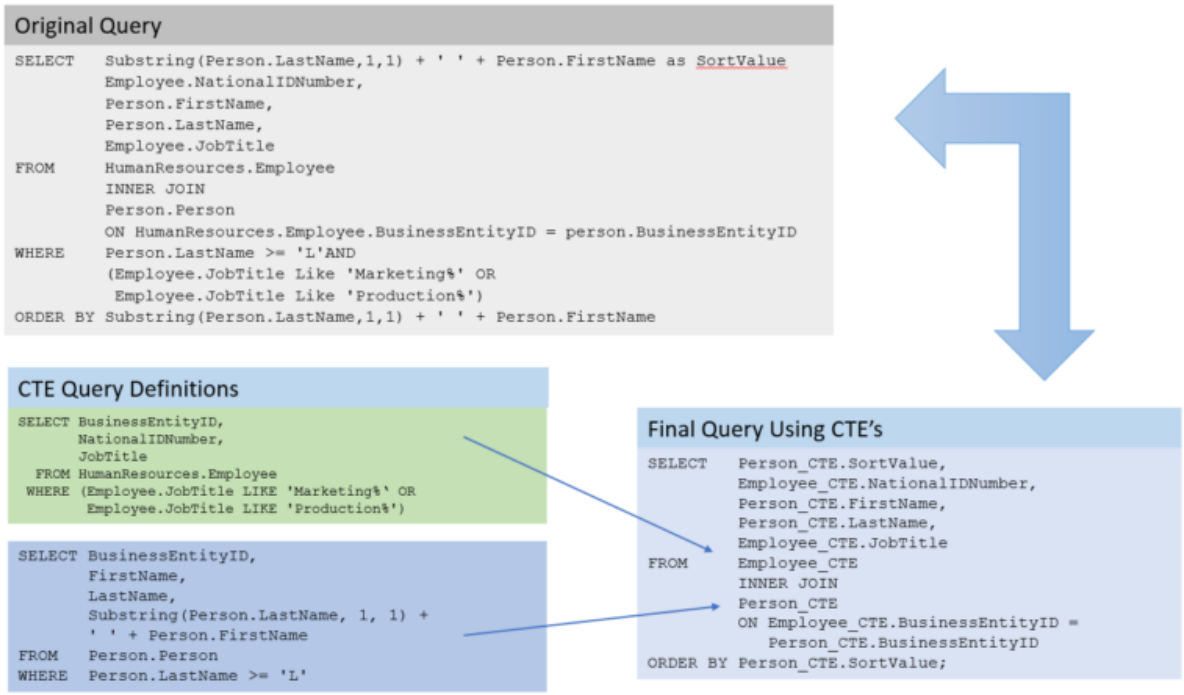
CTE joined to another CTE

In the example below are two CTEs. I've colored them blue and green respectively. The blue colored CTE is the same as the above example, the green one is newly added, and really helps to simplify the overall query.

```
WITH     Employee_CTE (BusinessEntityID, NationalIDNumber, JobTitle)
AS       (SELECT BusinessEntityID,
                 NationalIDNumber,
                 JobTitle
          FROM   HumanResources.Employee
          WHERE  (Employee.JobTitle LIKE 'Marketing%'
                  OR Employee.JobTitle LIKE 'Production%')),
         Person_CTE (BusinessEntityID, FirstName, LastName, SortValue)
AS       (SELECT BusinessEntityID,
                 FirstName,
                 LastName,
                 Substring(Person.LastName, 1, 1) + ' ' +
Person.FirstName
          FROM   Person.Person
          WHERE  Person.LastName >= 'L')
SELECT   Person_CTE.SortValue,
         Employee_CTE.NationalIDNumber,
         Person_CTE.FirstName,
         Person_CTE.LastName,
         Employee_CTE.JobTitle
FROM     Employee_CTE
         INNER JOIN
         Person_CTE
         ON Employee_CTE.BusinessEntityID = Person_CTE.BusinessEntityID
ORDER BY Person_CTE.SortValue;
```

In our original example the SortValue expression, Substring(Person.LastName, 1, 1) + ' ' + Person.FirstName, was repeated in both the SELECT clause and ORDER BY statement. Now, by placing the SortValue expression within the CTE, we only need to define it once!

Compare the original query to the final one:



Two CTE's joined to one another.

I think the original query is harder to read and maintain for these reasons:

1. The SortValue expression is listed twice.
2. The filtering of both tables are in the same expression.
3. Since queries for both the employee and person are in the same statement it is difficult to separate them for debugging purposes.

I like having the statement broke out into a CTE because:

1. If I need to verify what rows are being returned by a CTE, it's as simple as running the query definition in a separate query window.
2. Expression, such as that used for SortValue, aren't repeated.
3. The final Query Using CTEs is easier to read. You can focus on the join conditions, and not be distracted by filtering nor expressions.

CTE joined to Self

As you have seen you can join a CTE to another table, or CTE. But did you know you can join a CTE to itself?

Note: a table is joined to itself is a self-join.

Consider the following CTE:

```
WITH Department_CTE (Name, GroupName)
AS (SELECT Name,
           GroupName
      FROM HumanResources.Department)
SELECT D1.Name,
       D2.Name
FROM   Department_CTE AS D1
INNER JOIN
Department_CTE AS D2
ON d1.GroupName = d2.GroupName
```

Here you can see one CTE is defined as Department_CTE and that this is then used twice within the query.

The result of this query is to list combinations of department names within the same department group:

	Name	Name
1	Engineering	Engineering
2	Tool Design	Engineering
3	Research and Development	Engineering
4	Engineering	Tool Design
5	Tool Design	Tool Design
6	Research and Development	Tool Design
7	Sales	Sales
8	Marketing	Sales
9	Sales	Marketing
10	Marketing	Marketing
11	Purchasing	Purchasing
12	Shipping and Receiving	Purchasing
13	Engineering	Research and Developme
14	Tool Design	Research and Developme
15	Research and Development	Research and Developmen
16	Production	Production

CTE Self Join

Substitute for a View

As your SQL becomes more complex you'll find that using views are a great way to hide the inner workings of a query and allow you to just focus on the results. This is especially true when you're working with data involving multiple joins.

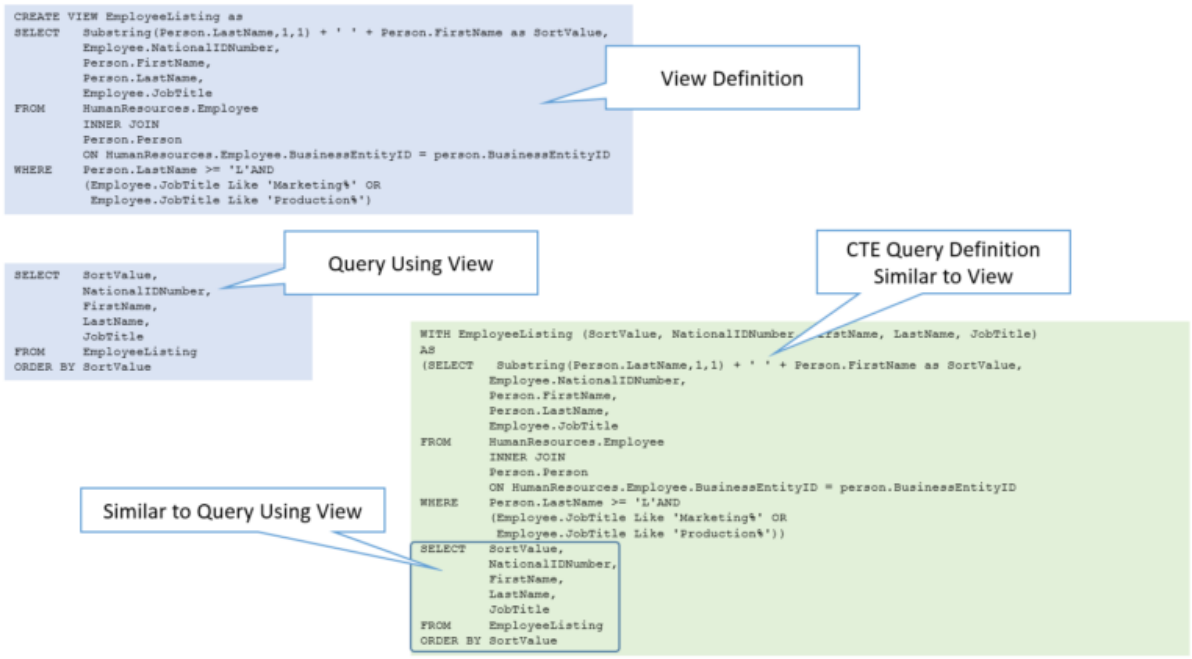
With a view you can encapsulate all that query and join logic into the view. Queries using the view are simpler and easier to read.

But there are times when it may not make sense to create a view. For instance, if you don't have permissions to create database objects, such as when using a third party database, or when the only time you'll need to us the view is just once.

In these cases, you can use a common table expression.

When creating a CTE, keep in mind the CTEs query definition is the same as the query used to create the view.

In the diagram below we show a query two ways. In blue you'll see defined as view, and then used in a query. It is then shown in green as a CTE.



Substituting a View with a CTE

We then take the same query used to define the view and use that for the CTE query definition. Finally, you can see that the CTEs final query is exactly like that used to reference the view.

CTE versus Derived Table

Derived tables are table results defined in the FROM clause. Given that derived tables return a table expression, it should be no surprise that you can use CTEs in their place.

Consider the following query. The derived tables are color coded in red and green.

```
SELECT Quota.TerritoryID,
       Quota.TerritoryQuota,
       Sales.TerritorySales,
       Sales.TerritorySales - Quota.TerritoryQuota
FROM   (SELECT   TerritoryID,
                 SUM(SalesQuota) AS TerritoryQuota
        FROM     Sales.SalesPerson
        GROUP BY TerritoryID) AS Quota
INNER JOIN
       (SELECT   SOH.TerritoryID,
                 SUM(SOH.TotalDue) AS TerritorySales
        FROM     Sales.SalesOrderHeader AS SOH
        GROUP BY SOH.TerritoryID) AS Sales
ON Quota.TerritoryID = Sales.TerritoryID
```

We can make it easier to read the statement by pulling out the table expression into a CTE. Here is the same query using CTEs instead of derived tables:

```
WITH Quota (territoryid, quota)
 AS  (SELECT   territoryid,
               Sum(salesquota) AS TerritoryQuota
        FROM   sales.salesperson
      GROUP BY territoryid),
     Sales (territoryid, sales)
AS  (SELECT   SOH.territoryid,
              Sum(SOH.totaldue) AS TerritorySales
       FROM   sales.salesorderheader AS SOH
     GROUP BY SOH.territoryid)
SELECT Quota.territoryid,
       Quota.quota,
       Sales.sales,
       Sales.sales - Quota.quota
FROM   Quota
       INNER JOIN
       Sales
       ON Quota.territoryid = Sales.territoryid;
```

By using CTEs we move the code used to define the query results for Quota and Sales away from the portion used to combine the table together.

I think this makes it much easier to maintain. For instance, if you need to make changes, it's easier to know where to make a change. Also, it makes it easier to be able to see what is actually being joined together. The queries for each derived table are cluttering that portion of the SELECT statement.

CTEs versus Subqueries

CTEs and subqueries are similar, but CTEs have capabilities not found with subqueries.

Subqueries and CTEs:

- Are table expressions created at run-time. They are temporary objects.
- Can be created and used within stored procedures, triggers, and views.
- Can be correlated or non-correlated. A CTE can reference a CTE previously defined in the same statement.
- Can be in in SELECT, FROM, WHERE, HAVING, IN, EXISTS clauses.

There are some differences between subqueries and CTEs, notably:

- A subquery is defined within an outer query. A CTE is defined before calling it from within the query.
- A CTE can reference itself, a subquery cannot.
- A CTE can reference other CTEs within the same WITH clause (Nest). A subquery cannot reference other subqueries.
- A CTE can be referenced multiple times from a calling query. A subquery cannot be referenced.

CTE versus Correlated Subquery

Correlated subqueries can also be replaced with non recursive CTEs. This shouldn't be a new concept, as we've seen in the past it is easy to convert a correlated sub query into a join. Given this, and knowing that joins are easily moved into non recursive CTEs, you can see the possibilities.

Let's that the following correlated subquery which displays the sales order information. The subquery is used to calculate the average line total for each sales order.

```
SELECT salesorderid,
       salesorderdetailid,
       linetotal,
       (SELECT Avg(linetotal)
        FROM   sales.salesorderdetail
        WHERE  salesorderid = SOD.salesorderid) AS AverageLineTotal
 FROM   sales.salesorderdetail SOD
```

To replicate this query using a CTE we first need to create query definition to calculate the average line total for each Sales Order.

This common table expression is then joined to the sales order table to obtain our final result.

```
WITH linetotal_cte (salesorderid, averagelinetotal)
AS   (SELECT    salesorderid,
                Avg(linetotal)
      FROM      sales.salesorderdetail
      GROUP  BY salesorderid)
 SELECT SOD.salesorderid,
        SOD.salesorderdetailid,
        SOD.linetotal,
        LT.averagelinetotal
 FROM   sales.salesorderdetail SOD
 INNER JOIN linetotal_cte LT
 ON LT.salesorderid = SOD.salesorderid
```

Limitations

Non Recursive CTEs can also be use to overcome limitations such as “enable grouping by a column that is derived from a scalar subselect, or a function that is either not deterministic or has external access.” (TechNet)

Suppose you wanted to know how many departments have the same number of employees. What query could you use?

Finding the number of employees by department name is pretty easy. We can use the following query to do so:

```
SELECT GroupName,
       Name,
       (SELECT Count(1)
        FROM   HumanResources.EmployeeDepartmentHistory AS H
        WHERE  D.DepartmentID = H.DepartmentID
               AND H.EndDate IS NULL) AS NumberEmployees
FROM    HumanResources.Department AS D;
```

The number of employees in each department is calculated using a scalar sub select. This is colored red in the above query for easy identification. Here is a sample result:

	GroupName	Name	NumberEmployees
1	Research and Development	Engineering	6
2	Research and Development	Tool Design	4
3	Sales and Marketing	Sales	18
4	Sales and Marketing	Marketing	9
5	Inventory Management	Purchasing	12
6	Research and Development	Research and Development	4
7	Manufacturing	Production	179
8	Manufacturing	Production Control	6
9	Executive General and Administration	Human Resources	6
10	Executive General and Administration	Finance	10
11	Executive General and Administration	Information Services	10
12	Quality Assurance	Document Control	5
13	Quality Assurance	Quality Assurance	6
14	Executive General and Administration	Facilities and Maintenance	7
15	Inventory Management	Shipping and Receiving	6

Scalar sub-select results.

Now let's take this once step further and count how many departments have the same number of employees. To do this we should group on NumberEmployees, but this statement is invalid:

```
SELECT (SELECT Count(1)
          FROM   HumanResources.EmployeeDepartmentHistory AS H
         WHERE  D.DepartmentID = H.DepartmentID
                AND H.EndDate IS NULL) AS NumberEmployees,
        Count(Name) SameCount
FROM   HumanResources.Department AS D
GROUP BY NumberEmployees
```

Due to the grouping restriction. The subquer is a scarlar subselect and we're trying to group by it. That is a SQL violation! As is

```
SELECT  (SELECT Count(1)
  FROM   HumanResources.EmployeeDepartmentHistory AS H
WHERE    D.DepartmentID = H.DepartmentID
        AND H.EndDate IS NULL) AS NumberEmployees,
        Count(Name) SameCount
FROM   HumanResources.Department AS D
GROUP BY  (SELECT Count(1)
           FROM   HumanResources.EmployeeDepartmentHistory AS H
           WHERE  D.DepartmentID = H.DepartmentID
                  AND H.EndDate IS NULL)
```

To solve this problem, we can define and use a CTE:

```
WITH     Department_CTE (GroupName, Name, NumberEmployees)
AS       (SELECT GroupName,
                 Name,
                (SELECT Count(1)
                 FROM   HumanResources.EmployeeDepartmentHistory AS H
                 WHERE  D.DepartmentID = H.DepartmentID
                        AND H.EndDate IS NULL) AS NumberEmployees
          FROM   HumanResources.Department AS D)
SELECT   NumberEmployees,
         Count(Name) SameCount
FROM     Department_CTE
GROUP BY NumberEmployees;
```

Now we query the CTE and group by the result of the scalar sub select (red text). The results returned are:

	NumberEmployees	SameCount
1	2	1
2	4	2
3	5	1
4	6	5
5	7	1
6	9	1
7	10	2
8	12	1
9	18	1

Final Results of Grouping

From this we can see there are 5 departments with 6 employees.

Use with Ranking functions

You can use ranking functions such as RANK() and NTILE() in conjunction with windowing functions to return the top items within a group.

Suppose we want to return the top sales within each territory. To do this we can RANK sales within each territory as 1,2,…, and so on, and then select those with a rank of 1.

Here is the query you can use to generate sales ranks within territories:

```
SELECT RANK() OVER(PARTITION BY S.TerritoryID ORDER BY SOH.TOTALDue desc)
      ,S.TerritoryID
      ,SOH.SalesOrderNumber
      ,SOH.TotalDue
FROM  Sales.SalesPerson S
INNER JOIN Sales.SalesOrderHeader SOH
ON S.BusinessEntityID = SOH.SalesPersonID
WHERE S.TerritoryID is not NULL
```

Given this, then our job is to pick only those rows whose RANK() is one. These are the top ranked sales within each territory. You may think you could repeat the ranking expression in the WHERE clause to filter, but this isn't possible. Windows functions, such as partition are not allowed in the WHERE clause and if you try to "wrap" it into a subquery you'll get an error since the subquery returns more than one row.

It seems we are stuck. But this is where non recursive CTEs come to our aid.

With a CTE we can define a query to return the sales data ranked by territory and then query that to only return those items ranked first in each territory:

```
WITH   SalesRankCTE (SalesPersonID, Name, TerritoryID, SalesRanking,
SalesOrderNumber, TotalDue)
AS     (SELECT SalesPersonID,
               P.FirstName + ' ' + P.LastName,
               S.TerritoryID,
               RANK() OVER (PARTITION BY S.TerritoryID ORDER BY
SOH.TOTALDue DESC),
               SOH.SalesOrderNumber,
```

```
                SOH.TotalDue
        FROM    Sales.SalesPerson AS S
        INNER JOIN
        Sales.SalesOrderHeader AS SOH
        ON S.BusinessEntityID = SOH.SalesPersonID
        INNER JOIN
        Person.Person AS P
        ON P.BusinessEntityID = S.BusinessEntityID
        WHERE  S.TerritoryID IS NOT NULL)
SELECT SalesPersonID,
       Name,
       TerritoryID,
       SalesRanking,
       SalesOrderNumber,
       TotalDue
FROM   SalesRankCTE
WHERE  SalesRanking = 1;
```

To help you see this, the original query used to generate rankings is highlighted in blue. The filter to only show the top ranked sales in each territory is in red.

Kris Wenzel | Categories ↓ | Tags ↓

Related Posts

Joins versus Subqueries SQL Puzzle

What is the difference between a subquery and inner join?

Learn how to Calculate the Median Value using PERCENTILE_DISC

Learn to use the Data Dictionary in SQL Server

←Previous post

Next post→