Extend your queries with APPLY

SQL Server offers a way to combine tables with table-valued functions, and to create the equivalent of correlated derived tables.

Tamar E. Granor, Ph.D.

As I've been digging into SQL syntax that's not supported in VFP, I've been able to understand what most of the language elements do, even when I've struggled to find meaningful examples. But getting my head around the APPLY operator took a look of effort, trials and research.

It was worth it, though, because CROSS APPLY and OUTER APPLY offer useful capabilities. I'm hopeful that I can explain and demonstrate in a way that makes your journey easier than mine was.

Start with CROSS JOIN

The best way to understand what CROSS APPLY does is to start with the idea of a *cross join*, also known as a *Cartesian join*. That's a join where each record in the first table is matched with each record in the second table. It's almost always something to avoid, but there are a few cases where it's useful.

Visual FoxPro doesn't have direct syntax for cross joins, but you can use them by specifying tables without a join condition. For example, suppose you want to get a list of all months for a period of years (that is, one record for each month for each specified year). If you already have a cursor containing years and another containing the months, you could use a query like that in Listing 1 to combine them.

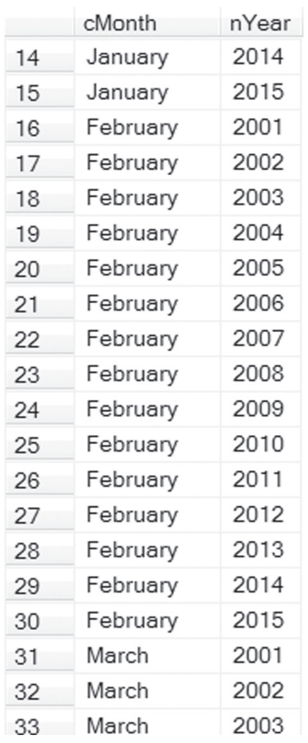
Listing 1. Cross joins are useful for building a data set from a set of "building blocks."

```
SELECT cMonth, nYear ;
  FROM csrMonths, ;
       csrYears ;
  INTO CURSOR csrEveryMonth
```

SQL Server supports cross joins directly with the CROSS JOIN syntax. The SQL Server query equivalent to Listing 1 is shown in Listing 2. Partial results are shown Figure 1; the #Years table in this case contains all the values from 2001 to 2015.

Listing 2. SQL Server includes CROSS JOINs directly.

```
SELECT cMonth, nYear
  FROM #Months
   CROSS JOIN #Years
```



	cMonth	nYear
14	January	2014
15	January	2015
16	February	2001
17	February	2002
18	February	2003
19	February	2004
20	February	2005
21	February	2006
22	February	2007
23	February	2008
24	February	2009
25	February	2010
26	February	2011
27	February	2012
28	February	2013
29	February	2014
30	February	2015
31	March	2001
32	March	2002
33	March	2003

Figure 1. CROSS JOIN matches every record in the first table with every record in the second table.

I chose this example intentionally because ensuring that data is shown for every month is one of the few cases where I've actually used a cross join. For something like creating a data warehouse, you might use a cross join to match every employee with every product, so that your data contains all possible combinations.

APPLY vs. JOIN

With the idea set that the CROSS keyword means match all items on both sides, we can move on to APPLY. What makes APPLY powerful is that it can be "correlated," that is, the item on the right can be defined using one or more fields from the item on the left. An "item" here can be a derived table or a table-valued function. Normally, the definition of a derived table can't refer to fields from another table in the main query.

One place this ability is useful is in providing an alternative approach to finding the top N items in each group. I showed one solution to that problem in the May, 2014 issue; that approach uses the OVER clause with the RANK function.

With CROSS APPLY, we can take a more direct route to the "top N per group" problem. Listing 3 shows how to find the 5 employees who most recently joined each department; it's included in this month's downloads as MostRecentEmpsByDept.SQL. The derived table finds the five most recent employees for a specific department; the WHERE condition in the subquery limits it to the department that's currently being considered in the outer query. CROSS APPLY then matches all five of those records to the current department record.

Listing 3. One use for CROSS APPLY is to find the TOP N records in each group.

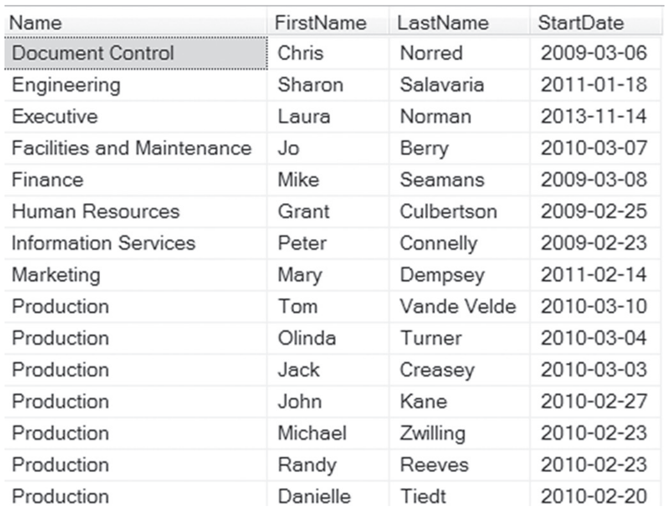
```
SELECT Name, FirstName, LastName, StartDate
  FROM HumanResources.Department Dept
    CROSS APPLY
    (SELECT TOP 5 FirstName, LastName,
                  StartDate
      FROM HumanResources.Employee Emp
       JOIN
  HumanResources.EmployeeDepartmentHistory EDH
         ON Emp.BusinessEntityID =
            EDH.BusinessEntityID
       JOIN Person.Person
         ON Emp.BusinessEntityID =
            Person.BusinessEntityID
       WHERE EDH.DepartmentID =
             Dept.DepartmentID
         AND EndDate IS NULL
       ORDER BY StartDate DESC) csrDeptTop
```

In my tests, this version and the version using OVER (included in this month's downloads as NewestEmployeesByDept.SQL) are equally speedy. Some of the resources I used to learn about CROSS APPLY indicate that it's usually faster than the OVER solution.

APPLY seems even more natural if we want to choose the top N percent for each group. Imagine that instead of finding the 5 employees who joined a department most recently, you want to find the most recent 5%. Just add the PERCENT keyword to the derived table in Listing 3 and you're set. Listing 4 (including in this month's downloads as MostRecentEmpsByDeptPercent.sql) shows the query. Figure 2 shows partial results.

Listing 4. APPLY works well for finding the top N percent for each group.

```
SELECT Name, FirstName, LastName, StartDate
   FROM HumanResources.Department
     CROSS APPLY
     (SELECT TOP 5 PERCENT
             FirstName, LastName, StartDate
       FROM HumanResources.Employee
        JOIN
   HumanResources.EmployeeDepartmentHistory EDH
          ON Employee.BusinessEntityID =
             EDH.BusinessEntityID
        JOIN Person.Person
          ON Employee.BusinessEntityID =
             Person.BusinessEntityID
        WHERE EDH.DepartmentID =
              Department.DepartmentID
          AND EndDate IS NULL
        ORDER BY StartDate DESC) csrDeptTop
```



Name	FirstName	LastName	StartDate
Document Control	Chris	Norred	2009-03-06
Engineering	Sharon	Salavaria	2011-01-18
Executive	Laura	Norman	2013-11-14
Facilities and Maintenance	Jo	Berry	2010-03-07
Finance	Mike	Seamans	2009-03-08
Human Resources	Grant	Culbertson	2009-02-25
Information Services	Peter	Connelly	2009-02-23
Marketing	Mary	Dempsey	2011-02-14
Production	Tom	Vande Velde	2010-03-10
Production	Olinda	Turner	2010-03-04
Production	Jack	Creasey	2010-03-03
Production	John	Kane	2010-02-27
Production	Michael	Zwilling	2010-02-23
Production	Randy	Reeves	2010-02-23
Production	Danielle	Tiedt	2010-02-20

Figure 2. CROSS APPLY lets you find the first N% for each group.

In SQL Server 2012 and later, you can accomplish almost the same thing with OVER and the PERCENT_RANK function (shown in Listing 5 and included in this month's downloads as NewestEmployeesByDeptPercent.SQL), but in earlier versions, you'd have to use RANK and do some arithmetic to find the top five percent.

Listing 5. SQL Server 2012 and later let you find top N percent for each group using OVER and PERCENT_RANK.

```
WITH EmpRanksByDepartment AS
(SELECT FirstName, LastName, StartDate,
        Department.Name AS Department,
        PERCENT_RANK() OVER
         (PARTITION BY Department.DepartmentID
          ORDER BY StartDate Desc)
         AS EmployeeRank
FROM HumanResources.Employee
 JOIN HumanResources.EmployeeDepartmentHistory
      EDH
   ON Employee.BusinessEntityID =
      EDH.BusinessEntityID
 JOIN HumanResources.Department
   ON EDH.DepartmentID =
      Department.DepartmentID
 JOIN Person.Person
   ON Employee.BusinessEntityID =
      Person.BusinessEntityID
 WHERE EndDate IS NULL)

 SELECT FirstName, LastName, StartDate,
        Department
  FROM EmpRanksByDepartment
    WHERE EmployeeRank <= 0.05
    ORDER BY Department, StartDate desc ;
```

The queries in Listing 4 and Listing 5 do not return identical data sets. Because of the way PERCENT_RANK handles ties, Listing 5 returns one more record. The speed of the two queries is identical in my tests.

APPLY with table-valued functions

APPLY is particularly useful when you have a table-valued function and want to join its results with a table. As its name implies, a table-valued function is a function stored in the database that returns not a single value, but a table. It may return one record with multiple fields or multiple records of one or more fields.

A table-valued function can appear on the right side of the APPLY operator; each record returned is joined to each record of the table on the left side of APPLY.

For example, suppose you want to find the top five products in sales for each month. As with the previous example, you can do this using OVER with RANK(); Listing 6 shows such a query; it's included in this month's downloads as Top5ProductsByMonth.SQL.

Listing 6. You can use OVER and RANK() to find the top-selling products for each month.

```
WITH MonthlyProductSales(iProductID, nMonth,
                         nYear, nSales)
AS
(SELECT SOD.ProductID, MONTH(OrderDate),
        YEAR(OrderDate), SUM(LineTotal)
  FROM [Sales].[SalesOrderDetail] SOD
    JOIN [Sales].[SalesOrderHeader] SOH
      ON SOD.SalesOrderID = SOH.SalesOrderID
  GROUP BY ProductID,
           MONTH(OrderDate), Year(OrderDate)),

RankedProductSales(iProductID, nMonth, nYear,
                   nSales, nRank)
AS
(SELECT MonthlyProductSales.*,
        RANK() OVER (
          PARTITION BY nMonth, nYear
          ORDER BY nSales DESC)
  FROM MonthlyProductSales)

SELECT nMonth, nYear,
       iProductID, Name, nSales
  FROM RankedProductSales
    JOIN Production.Product
      ON RankedProductSales.iProductID =
         Product.ProductID
  WHERE nRank <= 5
  ORDER BY nYear, nMonth, nRank ;
```

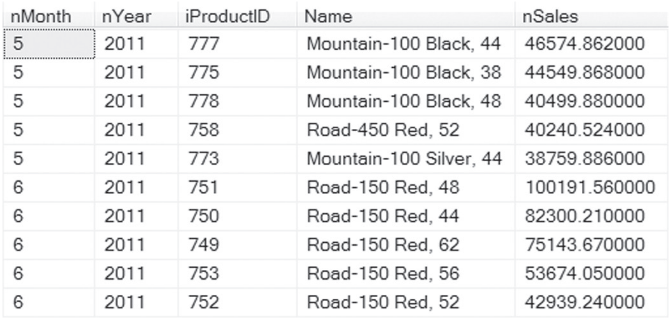
But APPLY gives us another approach, extracting the top five products one month at a time and joining them to the other information we want. Listing 7 shows a query analogous to the one in Listing 3, though more complex; it's included in this month's downloads as Top5ProductsByMonthCrossApply.SQL. The first CTE computes monthly sales for each product, while the second provides a list of all the month and year combinations included in the data. The innermost derived table (on the right-hand side of CROSS APPLY) finds the top five products for the month and year specified for the current record from csrProductSalesByMonth. The middle-level derived table performs the cross apply. Finally, the outer query adds the name of each product. Figure 3 shows partial results.

Listing 7. Once again, CROSS APPLY provides an alternate approach to finding the top N for each group.

```
WITH csrProductSalesByMonth
AS
(SELECT SOD.ProductID,
        MONTH(OrderDate) AS nMonth,
        YEAR(OrderDate) nYear,
        SUM(LineTotal) AS nSales
  FROM [Sales].[SalesOrderDetail] SOD
    JOIN [Sales].[SalesOrderHeader] SOH
    ON SOD.SalesOrderID = SOH.SalesOrderID
  GROUP BY ProductID,
           MONTH(OrderDate), YEAR(OrderDate)),

csrMonths
AS
(SELECT DISTINCT Month(OrderDate) AS nMonth,
        YEAR(OrderDate) as nYear
  from Sales.SalesOrderHeader)

SELECT nMonth, nYear, Product.ProductID,
       Name, nSales
  FROM Production.Product
    JOIN (
    SELECT csrTopSales.ProductID,
           csrMonths.nMonth,
           csrMonths.nYear,
           nSales
      FROM csrMonths
        CROSS APPLY
        (SELECT TOP 5 ProductID, nMonth,
                      nYear, nSales
          FROM csrProductSalesByMonth
          WHERE csrProductSalesByMonth.nMonth
                = csrMonths.nMonth
            AND csrProductSalesByMonth.nYear =
                csrMonths.nYear
          ORDER BY nSales desc) csrTopSales
      ) csrAllTopSales
    ON Product.ProductID =
       csrAllTopSales.ProductID
  ORDER BY nYear, nMonth, nSales DESC;
```



nMonth	nYear	iProductID	Name	nSales
5	2011	777	Mountain-100 Black, 44	46574.862000
5	2011	775	Mountain-100 Black, 38	44549.868000
5	2011	778	Mountain-100 Black, 48	40499.880000
5	2011	758	Road-450 Red, 52	40240.524000
5	2011	773	Mountain-100 Silver, 44	38759.886000
6	2011	751	Road-150 Red, 48	100191.560000
6	2011	750	Road-150 Red, 44	82300.210000
6	2011	749	Road-150 Red, 62	75143.670000
6	2011	753	Road-150 Red, 56	53674.050000
6	2011	752	Road-150 Red, 52	42939.240000

Figure 3. The query in Listing 7 uses CROSS APPLY to find the five top-selling products for each month.

While that query works, it's also pretty unwieldy. To make it easier to work with, we can replace the innermost derived table with a call to a table-valued function. If we need the top-selling products for a given month as part of multiple queries, then having such a function in the database allows us to avoid writing the same query repeatedly, as well.

First, we need to create the function. Listing 8 (CreateTopProductSalesFunctionInline.SQL in this month's downloads) shows the code to do so. It has three parameters: the month, the year, and the number of products to return. It returns a table with two columns: the product ID and the sales of that product for the specified month. Note that it actually handles both the computation of the total sales (done by the CTE in Listing 7) and choosing the top N.

Listing 8. We can store a function in the table to compute the top-selling products for a month. It returns a table.

```
CREATE FUNCTION
  dbo.TopProductSalesForMonthInline
  (@nMonth Int, @nYear Int,
   @nHowMany SmallInt)
RETURNS TABLE AS
RETURN
(SELECT TOP (@nHowMany) ProductID, nSales
  FROM (
    SELECT SOD.ProductID,
           MONTH(OrderDate) AS nMonth,
           YEAR(OrderDate) nYear,
           SUM(LineTotal) AS nSales
      FROM [Sales].[SalesOrderDetail] SOD
        JOIN [Sales].[SalesOrderHeader] SOH
          ON SOD.SalesOrderID =
             SOH.SalesOrderID
    WHERE MONTH(OrderDate) = @nMonth
      AND YEAR(OrderDate) = @nYear
    GROUP BY ProductID,
             MONTH(OrderDate),
             YEAR(OrderDate)
    ) MonthlyProductSales
  ORDER BY nSales DESC)
```

Once the function exists, we can use it in a query, as in Listing 9. There's still a CTE to get the list of months, but the main query is just a single CROSS APPLY—no derived tables at all. This query is included in this month's downloads as Top5ProductsByMonthCrossApplyFunctionInline.SQL

Listing 9. Using the table-valued function, the query to find the top 5 products for each month becomes much simpler.

```
WITH csrMonths (nMonth, nYear)
AS
(SELECT DISTINCT MONTH(OrderDate),
                 YEAR(OrderDate)
  FROM Sales.SalesOrderHeader)

SELECT nMonth, nYear,
       TPS.ProductID, Name, nSales
  FROM csrMonths
  CROSS APPLY
    dbo.TopProductSalesForMonthInline(
      csrMonths.nMonth, csrMonths.nYear,5) TPS
  JOIN Production.Product
    ON TPS.ProductID = Product.ProductID
  ORDER BY nYear, nMonth, nSales DESC
```

Note that the function created here is what's called an "inline function"; that's why its name includes "Inline." The performance of the query using the inline function is pretty much the same as the performance of the version using CROSS APPLY with a derived table. In my tests, both are a little faster than the OVER version.

However, my first attempt at using a table-valued function was much slower. That function (CreateTopProductSalesFunction.sql in this month's downloads contains code to create it) created what's called a multi-statement function. An inline function involves a single query wrapped in RETURN, while a multi-statement function has multiple commands wrapped in a BEGIN/END pair. In this test and others, it's clear that using an inline function with APPLY can boost performance, while a multi-statement function is likely to slow things town.

My reading also suggests that the relative speed of the three approaches depends a lot on what indexes are available; see the resources at the end of this article for some reading on that topic.

Is APPLY good for anything other than TOP N?

All the examples we've looked at so far provided alternate ways to find the top N records in each group. While that's the easiest example, not to mention the most commonly used in articles, APPLY does have other uses.

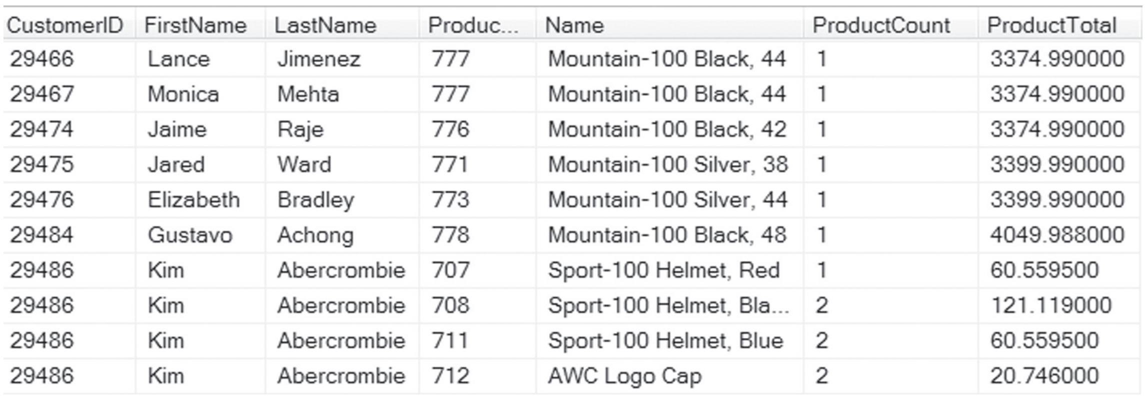
One is to calculate multiple values for each record to be matched. For example, suppose you want to know how many of each product a customer bought in a given period, and how much she spent total on each product. (I can imagine collecting this data to populate a data warehouse, for example.) You could do this with a CTE and some joins, as in Listing 10 (included in this month's downloads as CustomerSalesByProductJoin.SQL). Figure 4 shows partial results.

Listing 10. You can use a CTE to find out how many of each product each customer bought in a specified year.

```
WITH csrSalesByCustProd
  (CustomerID, ProductID,
   ProductCount, ProductTotal)
AS
(SELECT CustomerID, ProductID,
        COUNT(*), SUM(LineTotal)
  FROM Sales.SalesOrderHeader SOH
    JOIN Sales.SalesOrderDetail SOD
      ON SOH.SalesOrderID = SOD.SalesOrderID
  WHERE YEAR(OrderDate) = 2011
  GROUP BY CustomerID, ProductID)

SELECT Customer.CustomerID,
       FirstName, LastName,
       csrSalesByCustProd.ProductID, Name,
       ProductCount, ProductTotal
  FROM Sales.Customer
    JOIN csrSalesByCustProd
      ON Customer.CustomerID =
         csrSalesByCustProd.CustomerID
    LEFT JOIN Person.Person
      ON Customer.PersonID =
         Person.BusinessEntityID
    LEFT JOIN Production.Product
      ON csrSalesByCustProd.ProductID =
         Product.ProductID
  ORDER BY CustomerID, ProductID;
```

Alternatively, you can use CROSS APPLY with either a derived table or a table-valued function to do the same thing. Listing 11



CustomerID	FirstName	LastName	Produc...	Name	ProductCount	ProductTotal
29466	Lance	Jimenez	777	Mountain-100 Black, 44	1	3374.990000
29467	Monica	Mehta	777	Mountain-100 Black, 44	1	3374.990000
29474	Jaime	Raje	776	Mountain-100 Black, 42	1	3374.990000
29475	Jared	Ward	771	Mountain-100 Silver, 38	1	3399.990000
29476	Elizabeth	Bradley	773	Mountain-100 Silver, 44	1	3399.990000
29484	Gustavo	Achong	778	Mountain-100 Black, 48	1	4049.988000
29486	Kim	Abercrombie	707	Sport-100 Helmet, Red	1	60.559500
29486	Kim	Abercrombie	708	Sport-100 Helmet, Bla...	2	121.119000
29486	Kim	Abercrombie	711	Sport-100 Helmet, Blue	2	60.559500
29486	Kim	Abercrombie	712	AWC Logo Cap	2	20.746000

Figure 4. You might collect a list of products purchased by each customer in order to populate a data warehouse.

shows the code to create an inline table-valued function, included in this month's downloads as CreateCustomerProductSalesForYearFunctionInline.SQL. Listing 12 shows the query that uses the function; it's included in this month's downloads as CustomerSalesByProductFunction.SQL. In my tests, all three versions (CTE, CROSS APPLY with derived table, CROSS APPLY with inline table-valued function) give results within a few milliseconds of the same time. (The derived table version is included in this month's downloads as CustomerSalesByProduct.SQL, but is not shown here.)

Listing 11. The function created here returns a table containing the sales of each product for the specified customer in the specified year.

```
CREATE FUNCTION
  dbo.CustomerProductSalesForYear2
  (@CustomerID INT, @nYear INT)
RETURNS TABLE
  AS
RETURN(
SELECT ProductID,
       COUNT(*) AS ProductCount,
       SUM(LineTotal) AS ProductTotal
    FROM Sales.SalesOrderHeader SOH
      JOIN Sales.SalesOrderDetail SOD
      ON SOH.SalesOrderID = SOD.SalesOrderID
    WHERE SOH.CustomerID = @CustomerID
      AND YEAR(OrderDate) = @nYear
    GROUP BY ProductID)
```

Listing 12. CROSS APPLY matches each customer to the customer's function results, providing the same results as the query in Listing 10.

```
SELECT CustomerID, FirstName, LastName,
       CustProdSales.ProductID, Name,
       ProductCount, ProductTotal
   FROM Sales.Customer
     CROSS APPLY
       dbo.CustomerProductSalesForYear2(
         CustomerID, 2011) AS CustProdSales
     LEFT JOIN Person.Person
       ON Customer.PersonID =
          Person.BusinessEntityID
     LEFT JOIN Production.Product
       ON CustProdSales.ProductID =
          Product.ProductID
   ORDER BY CustomerID, ProductID;
```

OUTER APPLY, like OUTER JOIN

Until this point, we've looked only at the CROSS APPLY operator, which matches every record on the left with each record on the right. But like an inner join, if the right side produces no results for a given record on the left, that record doesn't appear in the result. With most of the examples we've considered so far, the most recent hires for each department and the top-selling products each month, that case is extremely unlikely to occur. Why would you have a department with no employees? In what month would no products have been sold? (I guess that could happen in a seasonal business that closes for some months of the year, but the queries we're using already omit those months.)

To demonstrate OUTER APPLY, instead consider the problem of showing all customers with their three largest (by amount) orders in a given period (say, a year). For each customer, we want one record for each of the top three. If the customer placed no orders in that period, include a single record for that customer.

We can do this with a CTE that uses OVER and RANK(), and then use an OUTER JOIN between the Customer table and the CTE. Listing 13 shows the code, included in this month's downloads as AllTopCustomerSalesJoin.SQL.

Listing 13. One way to find the top three sales for each customer in a specified year uses a CTE and OVER.

```
WITH csrTopThree
   (CustomerID, OrderDate, SubTotal)
AS
(SELECT CustomerID, OrderDate, SubTotal
  FROM (
    SELECT CustomerID, OrderDate, SubTotal,
           RANK() OVER (
             PARTITION BY CustomerID
             ORDER BY SubTotal DESC) AS nRank
      FROM Sales.SalesOrderHeader
      WHERE YEAR(OrderDate) = 2011
       ) csrCustomerOrders
  WHERE nRank <= 3)

SELECT Customer.CustomerID,
       FirstName, LastName,
       ISNULL(OrderDate, '') AS OrderDate,
       ISNULL(SubTotal, 0) AS OrderTotal
  FROM Sales.Customer
      LEFT JOIN csrTopThree
      ON Customer.CustomerID =
         csrTopThree.CustomerID
    JOIN Person.Person
      ON Customer.PersonID =
         Person.BusinessEntityID
  ORDER BY CustomerID, OrderTotal DESC;
```

As in the earlier examples, we can shorten the code and make it more readable by using APPLY. In this case, since we want all customers, we'll use OUTER APPLY, as in Listing 14; this query is included in this month's downloads as AllTopCustomerSales.SQL.

Listing 14. OUTER APPLY provides another way to list all customers with their largest orders in a specified year.

```
SELECT CustomerID, FirstName, LastName,
       ISNULL(OrderDate, '') AS OrderDate,
       ISNULL(SubTotal, 0) AS OrderTotal
  FROM Sales.Customer
    OUTER APPLY
     (SELECT TOP 3 OrderDate, SubTotal
        FROM Sales.SalesOrderHeader SOH
        WHERE SOH.CustomerID =
              Customer.CustomerID
         AND YEAR(OrderDate) = 2011
      ORDER BY SubTotal DESC
      ) CustTopSales
    JOIN Person.Person
      ON Customer.PersonID =
         Person.BusinessEntityID
  ORDER BY CustomerID, OrderTotal DESC;
```

On my machine, the OUTER JOIN version is faster, but both run in a few hundred milliseconds.

Also, as with the earlier example, you can create a table-valued function and then use it on the right side of the APPLY. Using an inline function, I see performance essentially identical to the version using OUTER APPLY with a derived table. This month's downloads include CreateTopCustomerSalesFunction.sql and AllTopCustomerSalesFunction.sql that, respectively, create the function and use it.

Measuring Performance

Over the years, I've written probably hundreds of timing tests in VFP, but I'd never done so in SQL Server until I was testing APPLY. It turns out to be quite simple to do a "quick and dirty" timing test.

Just as you would in VFP (and presumably, in most languages), grab the start time, run the process, grab the end time and subtract. The SQL Server GETDATE() function returns the current time as a datetime value. Listing 15 shows the skeleton for such a test.

Listing 15. SQL Server's GETDATE() function makes it easy to do "quick and dirty" timing tests.

```
declare @Start DateTime, @End DateTime;
select @Start = getdate();

-- Do the test

select @End = GETDATE();
SELECT @Start, @End,
       Datediff(ms, @Start, @End);
```

Why do I say this test is quick and dirty? For a number of reasons. First, the machine you're testing on is running a number of other things. At a minimum, you're running Windows, which has all kinds of things going in the background. Anything else you're running, like an email client, might steal some cycles, too.

Second, SQL Server undoubtedly caches some data. To avoid that effect, you'd at least have to close SSMS and restart it between tests; it's possible that you'd actually need to restart the machine to be sure to eliminate all caching effects.

Finally, the right way to actually test performance is to run lots of tests, not just one or two or five. The average of a series of tests (run under the conditions implied by the last two paragraphs) is a much better measure than any one test.

I should also point out that SSMS gives you the ability to see how a query is executed using its built-in Profiler. That's useful when you don't understand which part of a query is slowing it down.

Resources

I read a lot of articles and tried a lot of variations in my quest to understand APPLY. Here are a few of the more useful articles I found:

http://tinyurl.com/q5w2sp8
http://tinyurl.com/qced3ga
http://tinyurl.com/6qmjtss

If these don't help, there are quite a few other articles on the subject out there; you should have no trouble finding one that speaks to you.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. Tamar is author or co-author of a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with Visual FoxPro *and* Taming Visual FoxPro's SQL. *Her latest collaboration is* VFPX: Open Source Treasure for the VFP Developer, *available at www.foxrockx.com. Her other books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar was a Microsoft Support Most Valuable Professional from the program's inception in 1993 until 2011. She is one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.*