FoxRockX

Including missing data

Sometimes an outer join isn't sufficient to ensure that all the desired combinations are shown.

Tamar E. Granor, Ph.D.

November 2015
Number 47

In my recent series on the OVER clause, I failed to mention a trap in certain cases that can result in inaccurate data. This article explains the problem and demonstrates a solution that's applicable not just with OVER, but more broadly.

The OVER clause provides several ways of looking at data from multiple groups in a single result record. For example, you can use LAG and LEAD to include values from preceding and following records; my May, 2015 article shows how. Window frames, described in my March, 2015 article let you apply aggregate and analytical functions across a group of records.

Most of the examples in those articles use OVER in the context of time periods, such as days, months or years. For example, the query in Listing 1, drawn from the May article (and like all the examples in this article, using the AdventureWorks 2014 sample database), computes the quantity of each product sold in each year, and puts in a record with the previous year's sales and the following year's sales. Figure 1 shows partial results. The query is included in this month's downloads as SalesByYearWithPrevAndFoll.SQL.

Listing 1. LAG and LEAD let you pull data from preceding and following records into the current record.

```
WITH csrYearlySales
  (OrderYear, ProductID, NumSold)
AS
(SELECT YEAR(OrderDate) AS OrderYear,
        ProductID, SUM(OrderQty) AS NumSold
   FROM Sales.SalesOrderHeader SOH
     JOIN Sales.SalesOrderDetail SOD
       ON soh.SalesOrderID = sod.SalesOrderID
   GROUP BY YEAR(OrderDate), ProductID)

SELECT OrderYear, ProductID,
       LAG(NumSold) OVER
          (PARTITION BY ProductID
           ORDER BY OrderYear) AS PrevYear,
       NumSold AS CurrYear,
       LEAD(NumSold) OVER
           (PARTITION BY ProductID
            ORDER BY OrderYear) AS FollYear
   FROM csrYearlySales
   ORDER BY ProductID, OrderYear
```

OrderYear	ProductID	PrevYear	CurrYear	FollYear
2011	707	NULL	331	1278
2012	707	331	1278	2940
2013	707	1278	2940	1717
2014	707	2940	1717	NULL
2011	708	NULL	341	1387
2012	708	341	1387	3088
2013	708	1387	3088	1716
2014	708	3088	1716	NULL
2011	709	NULL	608	499
2012	709	608	499	NULL

Figure 1. Using LAG and LEAD, each row in this result shows data from three years.

However, there's an assumption built into the query—that each product was sold only in consecutive years. That is, LAG and LEAD look a specified number of *records* (by default, one) back and forward, based on the specified ordering; they don't consider the actual value of the ordering fields in those records to confirm that they're consecutive.

If you modify the query to show monthly sales with the previous and following month, as in Listing 2 (included in this month's downloads as SalesByMonthWithPrevAndFoll.SQL), the assumption and its flaws are more readily exposed. The first few lines of the results in Figure 2 make the problem obvious.

OrderMonth	OrderYear	ProductID	PrevMonth	CurrMonth	FollMonth
5	2011	707	NULL	24	58
7	2011	707	24	58	96
8	2011	707	58	96	141
10	2011	707	96	141	12
12	2011	707	141	12	61
1	2012	707	12	61	27
2	2012	707	61	27	93
3	2012	707	27	93	52
4	2012	707	93	52	162
5	2012	707	52	162	214
6	2012	707	162	214	197
7	2012	707	214	197	108
8	2012	707	197	108	147

Figure 2. Using LAG and LEAD with monthly data, you can see gaps that result in wrong data.

Listing 2. When you use LAG and LEAD with months (or days, rather than years), the underlying assumption is more apparent.

```
WITH csrMonthlySales
(OrderMonth, OrderYear, ProductID, NumSold)
AS
(SELECT MONTH(OrderDate) AS OrderMonth,
        YEAR(OrderDate) AS OrderYear,
        ProductID, SUM(OrderQty) AS NumSold
   FROM Sales.SalesOrderHeader SOH
     JOIN Sales.SalesOrderDetail SOD
       ON soh.SalesOrderID = sod.SalesOrderID
   GROUP BY YEAR(OrderDate),
            MONTH(OrderDate), ProductID)

SELECT OrderMonth, OrderYear, ProductID,
       LAG(NumSold) OVER
          (PARTITION BY ProductID
           ORDER BY OrderYear, OrderMonth)
          AS PrevMonth,
       NumSold AS CurrMonth,
       LEAD(NumSold) OVER
         (PARTITION BY ProductID
          ORDER BY OrderYear, OrderMonth)
         AS FollMonth
  FROM csrMonthlySales
  ORDER BY ProductID, OrderYear, OrderMonth
```

The first row represents May, 2011. It correctly shows null for the previous month, since this is the first month for that product, and 24 for May. However, it shows 58 for the following month, presumably June, 2011. But a look at the next row shows that, in fact, that 58 belongs to July, 2011; there were no sales of product 707 in June, 2011.

The same problem can occur with window frames using the ROWS keyword. Such a frame includes all records in the specified group without checking the data in the fields that specify the order. For example, the query in Listing 3 (included in this month's downloads as ProductSalesWithWeekly-WRONG.sql) is intended to compute units sold by day and by week for each product. It correctly computes daily sales, but because not every product is sold every day, the WeekSales field actually contains the number sold for the specified day and the three most recent days before it when the product was sold and the next three days on which the product was sold. You can see the problem in the partial results in Figure 3; for example, the first row shows a weekly total of 6, though only 3 units of product 707 were sold in the week between May 28 and June 3, 2011. The query is computing the total of the May 31, 2011 row and the next three rows.

OrderDate	ProductID	TodaysSales	WeekSales
2011-05-31 00:00:00.000	707	3	6
2011-06-05 00:00:00.000	707	1	7
2011-06-06 00:00:00.000	707	1	8
2011-06-08 00:00:00.000	707	1	9
2011-06-09 00:00:00.000	707	1	7
2011-06-13 00:00:00.000	707	1	7
2011-06-16 00:00:00.000	707	1	7
2011-06-20 00:00:00.000	707	1	8
2011-06-29 00:00:00.000	707	1	8
2011-06-30 00:00:00.000	707	1	8
2011-07-01 00:00:00.000	707	2	8
2011-07-06 00:00:00.000	707	1	8
2011-07-08 00:00:00.000	707	1	8

Figure 3. The results here show the flaw in computing weekly sales using ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING.

Listing 3. This query computes the daily sales for each product and is intended to include the weekly total (with the specified day in the middle) in each. It doesn't.

```
WITH csrSalesByProduct
   (ProductID, OrderDate, NumSold)
AS
(SELECT ProductID, OrderDate, SUM(OrderQty)
  FROM [Sales].[SalesOrderHeader] SOH
    JOIN [Sales].[SalesOrderDetail] SOD
      ON SOH.SalesOrderID =
         SOD.SalesOrderDetailID
  GROUP BY ProductID, OrderDate)

  SELECT OrderDate, ProductID,
         NumSold AS TodaysSales,
         SUM(NumSold) OVER (
```

```
            PARTITION BY productID
            ORDER BY ORDERDATE
            ROWS BETWEEN 3 PRECEDING AND
               3 FOLLOWING) AS WeekSales
   FROM csrSalesByProduct
   ORDER BY ProductID, OrderDate
```

Including missing data

In general, in SQL, the way we make sure to include rows that represent missing data is using outer joins. For example, the query in Listing 4 (included in this month's downloads as OneMonthSales.SQL) computes the number of each product sold in a specified month. Because of the LEFT JOIN in the main query, all products are included, even those not sold in the specified month.

Listing 4. The left outer join in this query ensures that every product is included in the results, not just those sold in October, 2013.

```
WITH csrProductSales (ProductID, NumSold)
AS
(SELECT ProductID, SUM(OrderQty)
   FROM [Sales].[SalesOrderDetail] SOD
     JOIN [Sales].[SalesOrderHeader] SOH
      ON SOD.SalesOrderDetailID =
         SOH.SalesOrderID
   WHERE MONTH(OrderDate) = 10
     AND YEAR(OrderDate) = 2013
   GROUP BY ProductID)

SELECT Product.ProductID, Name, NumSold
   FROM [Production].[Product]
     LEFT JOIN csrProductSales
       ON csrProductSales.ProductID =
          Product.ProductID
```

But the key to using an outer join this way is that we have a list of all products we can start with. In the earlier queries, we were missing months or days and the database doesn't contain a list of those. We could create a table to hold all dates of interest, but that would be a waste of space and would periodically require going in and adding more dates.

Fortunately, there are better ways to solve this problem. I'll show two solutions, both adapted from this discussion: http://stackoverflow.com/questions/11479918/include-missing-months-in-group-by-query.

To make it easier to follow, we'll start with a simpler problem, one that doesn't need OVER: counting how many orders were placed each day, essentially the same problem as in the StackOverflow post, but adapted to AdventureWorks.

The first approach, shown in Listing 5 uses a recursive CTE to build the list of dates. (I explained recursive CTEs in my March, 2014 article.) The first thing it does is extract the earliest and latest order dates to a pair of variables. (If you only want to report on particular dates, you can instead just set the variables appropriately.) The recursive CTE then uses the start date as the anchor and applies the DATEADD function to it to generate the remaining dates, stopping when it reaches the specified end date.

Once we have a complete list of dates, it's easy to do an outer join between that list and the sales orders to count how many there are each day. This query, shown in Listing 5, is included in this month's downloads as DailyOrderCountRecursive.SQL.

Listing 5. One way to make sure all dates are included uses a recursive CTE to generate the list.

```
DECLARE @StartDate DATETIME ;
DECLARE @EndDate DATETIME ;

SELECT @StartDate = MIN(Orderdate),
       @EndDate = MAX(OrderDate)
  FROM [Sales].[SalesOrderHeader];

-- recursive CTE
WITH AllDates (tDate)
AS (
    SELECT @StartDate
    UNION ALL
    SELECT DATEADD(DAY, 1, tDate)
    FROM AllDates
    WHERE tDate < @EndDate
    )

SELECT tDate,
       COUNT(SOH.SalesOrderID) AS OrderCount
FROM AllDates
  LEFT JOIN Sales.SalesOrderHeader AS SOH
    ON SOH.OrderDate >= tDate
    AND SOH.OrderDate < DATEADD(DAY, 1, tDate)
GROUP BY tDate
OPTION (MAXRECURSION 0);
```

The last line specifying MAXRECURSION 0 is needed because the default limit for recursion is 100. Setting that option to 0 means unlimited recursion.

The second approach to generating all the dates is a little harder to understand at first glance, but gives the same results. It also uses a CTE, but instead of using recursion, it generates a large set of numbers, keeps only the right quantity of numbers, and adds each to the start date using DATEADD. This version, included in this month's downloads as DailyOrderCountTopN.SQL, is shown in Listing 6.

Listing 6. This solution gets the list of dates by generating numbers to add to the start date.

```
DECLARE @StartDate DATETIME ;
DECLARE @EndDate DATETIME ;

SELECT @StartDate = MIN(Orderdate),
       @EndDate = MAX(OrderDate)
  FROM [Sales].[SalesOrderHeader];

WITH AllDates (tDate)
AS
(
  SELECT DATEADD(DAY, Num, @StartDate)
  FROM (
    SELECT TOP (DATEDIFF(DAY, @StartDate,
                         @EndDate) + 1)
      ROW_NUMBER() OVER
        (ORDER BY [object_id]) - 1 AS Num
      FROM sys.all_objects
      ORDER BY [object_id] ) AS Nums
)
```

```
SELECT AllDates.tDate,
       COUNT(SOH.SalesOrderID) AS OrderCount
FROM AllDates
  LEFT OUTER JOIN Sales.SalesOrderHeader SOH
    ON SOH.OrderDate >= AllDates.tDate
    AND SOH.OrderDate <
        DATEADD(DAY, 1, AllDates.tDate)
GROUP BY AllDates.tDate
ORDER BY AllDates.tDate;
```

As in the previous example, we start by figuring out the range of dates we want. The nested query in the CTE generates a list of numbers from 1 to the number of days we want; the way it does so is creative. It applies the ROW_NUMBER function to the system table called all_objects, and then uses TOP to limit the list to the desired number of days. The main query is then essentially the same as in the previous example, using an outer join to ensure we get results for every date.

A couple of caveats: first, this approach is limited by the size of the all_objects table. In my tests, I can't get more than 2708 distinct dates, even if I set the start and end dates to a larger range. Second, in my testing, for this query, the recursive solution is much faster than the Top N solution. (For other examples I tested, where the number of items to be generated is smaller, the Top N solution is still slower, but the difference is much less noticeable.)

Calling all months

The same two approaches work when it's all months you want to include, rather than all dates. The DateAdd() function makes it easy to switch from days to months, with one complication. We need to make sure that the start and end dates reflect the first of the month. We can do that with another application of DateAdd and DateDiff, as the code in Listing 7 demonstrates. This code is included in this month's downloads as MonthlyOrderCountRecursive.sql. Analogous code using the top N approach is included in the downloads as MonthlyOrderCountTopN.sql.

Listing 7. To show data for all months rather than all dates, change the first parameter to DateAdd and adjust the start and end dates.

```
DECLARE @StartDate DATETIME ;
DECLARE @EndDate DATETIME ;

SELECT @StartDate = MIN(Orderdate),
       @EndDate = MAX(OrderDate)
  FROM [Sales].[SalesOrderHeader];

-- correct to point to first of month
SET @StartDate = DATEADD(MONTH,
    DATEDIFF(MONTH, 0, @StartDate), 0);
SET @EndDate = DATEADD(MONTH,
    DATEDIFF(MONTH, 0, @EndDate), 0);

-- recursive CTE
WITH AllMonths (tDate)
AS
(SELECT @StartDate AS tDate
 UNION ALL
 SELECT DATEADD(m, 1, tDate)
   FROM AllMonths
   WHERE tDate < @EndDate)

SELECT DATENAME(MONTH, tDate) AS OrderMonth,
       YEAR(tDate) AS OrderYear,
       COUNT(SOH.SalesOrderID) AS OrderCount
FROM AllMonths
LEFT OUTER JOIN Sales.SalesOrderHeader as SOH
    ON SOH.OrderDate >= tDate
    AND SOH.OrderDate <
        DATEADD(MONTH, 1, tDate)
GROUP BY tDate
OPTION (MAXRECURSION 0);
```

The computation to adjust StartDate and EndDate to the first of the relevant months is worth a little discussion. First, DateDiff computes the number of months between SQL Server's "zero date" (January 1, 1900) and the specified date. It then adds that many months to the zero date to get the first day of the relevant month. While you could get to the first of the month by simply subtracting the right number of days from the date (using the DAY() function), the approach here has the advantage of setting the time to midnight as well, so it's the absolute start of the month, not just a random time on the right day.

Handling years is analogous in both cases. Change the first parameter to DateAdd() to YEAR, and modify the adjustment to StartDate and EndDate to go to the first of the year. This month's downloads include YearlyOrderCountRecursive.sql and YearlyOrderCountTopN.sql, showing the recursive and TOP N solutions, respectively.

Including missing records in two dimensions

The examples so far have required including missing records for a single table, but the queries in the first part of this article actually need to deal with two sets of missing values. Not only might there be some time periods (days, months, years, or whatever, depending on the particular query) for which there's no data at all, but there are also products which may not have been sold on a particular date or in a particular month or year.

For example, as indicated in Figure 2, though there are sales in June 2011, product 707 wasn't sold at all that month. So simply generating a list of all months doesn't provide a way to be sure we include all months for all products. To do that, we need to create a table that has that combination and outer join it to the actual data.

Consider the problem of showing the number of units ordered for each product on each day in a specified period. The table we need for the "all" side of an outer is one that has one record for each product for each day in the period. The query in Listing 8 (included in this month's downloads as DailyOrderCountByProductAllProductsRecursive.SQL) shows units sold for each product in 2013; it

uses a series of CTEs to create the necessary table, as well as to compute the daily units for each product in the specified period. The main query then joins the computed totals with the list of date/product combinations.

Listing 8. The CTEs here compute units sold by product by day, and create a list of all combinations of product and date. The two are joined to give a list of units sold per product each day in 2013.

```
DECLARE @StartDate DATETIME ;
DECLARE @EndDate DATETIME ;

SET @StartDate = '1/1/2013';
SET @EndDate = '1/1/2014';

-- recursive CTE
WITH AllDates (tDate)
AS
(SELECT @StartDate AS tDate
 UNION ALL
 SELECT DATEADD(DAY, 1, tDate)
   FROM AllDates
   WHERE tDate < @EndDate),

csrProductsByDays (tDate, ProductID)
AS
(SELECT tDate, ProductID
  FROM AllDates
    CROSS JOIN Production.Product),

csrProductOrders (OrderDate, ProductID,
                  NumOrdered)
AS
(SELECT OrderDate, ProductID, SUM(OrderQty)
  FROM Sales.SalesOrderHeader as SOH
    JOIN Sales.SalesOrderDetail SOD
      ON SOH.SalesOrderID = SOD.SalesOrderDe-
tailID
  WHERE OrderDate >= @StartDate
    AND OrderDate < @EndDate
  GROUP BY OrderDate, ProductID)

SELECT tDate, csrProductsByDays.ProductID,
       ISNULL(NumOrdered,0) AS NumOrdered
FROM csrProductsByDays
  LEFT OUTER JOIN csrProductOrders
      ON OrderDate >= tDate
      AND OrderDate < dateadd(DAY, 1, tDate)
    AND csrProductsByDays.ProductID = csrPro-
ductOrders.ProductID
ORDER BY ProductID, tDate
OPTION (MAXRECURSION 0);
```

The first CTE (AllDates) uses the recursive method to get a list of dates in the specified period.

The second CTE (csrProductsByDays) uses a cross-join to build a list of all date/product combinations. A cross-join is a join with no join condition; it matches every record in the first table with each record in the second table. Usually, cross-joins are something you try to avoid, but in cases like this, they're extremely useful.

The third CTE (csrProductOrders) computes the number of units sold for each product each day that it was sold.

Finally, the main query does an outer join between the results of the last two CTEs. Partial results are shown in Figure 4. Each record in the result that has 0 in the NumOrdered column was added by the outer join. If we ran the query that produces csrProductOrders on its own (with just the code to set up the @StartDate and @EndDate variables), those rows would be omitted.

tDate	ProductID	NumOrdered
2013-12-24 00:00:00.000	783	1
2013-12-25 00:00:00.000	783	6
2013-12-26 00:00:00.000	783	2
2013-12-27 00:00:00.000	783	0
2013-12-28 00:00:00.000	783	0
2013-12-29 00:00:00.000	783	0
2013-12-30 00:00:00.000	783	1
2013-12-31 00:00:00.000	783	13
2014-01-01 00:00:00.000	783	0
2013-01-01 00:00:00.000	784	0
2013-01-02 00:00:00.000	784	0
2013-01-03 00:00:00.000	784	0
2013-01-04 00:00:00.000	784	0
2013-01-05 00:00:00.000	784	0
2013-01-06 00:00:00.000	784	0

Figure 4. Each record with 0 in the NumOrdered column was added by the outer join.

This month's downloads include a version of this query using the TOP N approach as DailyOrderCountByProductAllProductsTopN.SQL.

Including missing records with OVER

Applying this technique with OVER isn't actually any different than using it in other queries. You use CTEs in the same way to ensure that you have all combinations to use in an outer join.

Listing 9 shows a query that correctly includes the sales of each product for each month, as well as sales of the product in the previous and following months in each record. The query is included in this month's downloads as SalesByMonthAllWithPrevAndFollRecursive.sql.

Listing 9. Using the same techniques as previous examples, you can get accurate results with LAG and LEAD, even if data is missing.

```
DECLARE @StartDate DATETIME ;
DECLARE @EndDate DATETIME ;

SELECT @StartDate = MIN(Orderdate),
       @EndDate = MAX(OrderDate)
  FROM [Sales].[SalesOrderHeader];

-- correct to point to first of month
SET @StartDate = DATEADD(MONTH,
    DATEDIFF(month, 0, @StartDate), 0);
SET @EndDate = DATEADD(MONTH,
    DATEDIFF(month, 0, @EndDate), 0);

WITH AllMonths (tDate)
AS
(SELECT @StartDate AS tDate
```

```
 UNION ALL
 SELECT DATEADD(m, 1, tDate)
   FROM AllMonths
   WHERE tDate < @EndDate),

csrProductsByMonths
  (OrderMonth, OrderYear, ProductID)
AS
(SELECT MONTH(tDate), YEAR(tDate), ProductID
  FROM AllMonths
    CROSS JOIN Production.Product),

csrMonthlySales
  (OrderMonth, OrderYear, ProductID, NumSold)
AS
(SELECT MONTH(OrderDate), YEAR(OrderDate),
        ProductID, SUM(OrderQty)
  FROM Sales.SalesOrderHeader SOH
    JOIN Sales.SalesOrderDetail SOD
      ON SOH.SalesOrderID = SOD.SalesOrderID
  GROUP BY YEAR(OrderDate), MONTH(OrderDate),
           ProductID)

SELECT PBM.OrderMonth, PBM.OrderYear,
       PBM.ProductID,
       ISNULL(LAG(NumSold) OVER
         (PARTITION BY PBM.ProductID
          ORDER BY PBM.OrderYear,
            PBM.OrderMonth), 0) AS PrevMonth,
       ISNULL(NumSold,0) AS CurrMonth,
       ISNULL(LEAD(NumSold) OVER
         (PARTITION BY PBM.ProductID
          ORDER BY PBM.OrderYear,
            PBM.OrderMonth), 0) AS FollMonth
  FROM csrMonthlySales MS
    RIGHT JOIN csrProductsByMonths PBM
      ON MS.OrderMonth = PBM.OrderMonth
    AND MS.OrderYear = PBM.OrderYear
    AND MS.ProductID = PBM.ProductID
  ORDER BY ProductID, OrderYear, OrderMonth
```

The CTEs here are quite similar to those in Listing 8, except that they're focused on months, not individual days. The main query uses a right outer join to ensure that every product/month combination appears in the results. The LAG and LEAD functions are wrapped in ISNULL, so that we get zeroes rather than nulls to represent months where a particular product wasn't sold. Figure 5 shows partial results.

OrderMonth	OrderYear	ProductID	PrevMonth	CurrMonth	FollMonth
12	2013	710	0	0	0
1	2014	710	0	0	0
2	2014	710	0	0	0
3	2014	710	0	0	0
4	2014	710	0	0	0
5	2014	710	0	0	0
6	2014	710	0	0	0
5	2011	711	0	33	0
6	2011	711	33	0	64
7	2011	711	0	64	75
8	2011	711	64	75	0
9	2011	711	75	0	181
10	2011	711	0	181	0
11	2011	711	181	0	7
12	2011	711	0	7	90

Figure 5. With records for each month/product combination, we can show each month with sales for the preceding and following months.

To add weekly sales of a product to the daily results (what the query in Listing 3 attempted), we need to create a table with each combination of product and date. Listing 10 demonstrates; here, I've also decided to include only products that have been sold at all, omitting those for which there are no recorded sales. As in the previous example, ISNULL converts nulls to zeroes. Partial results are shown in Figure 6; the query is included in this month's downloads as ProductSalesWithWeekly.sql.

Listing 10. To produce daily sales for product with a weekly total centered on that day requires four CTEs before the main query.

```
DECLARE @StartDate DateTime,
        @EndDate DateTime;

SELECT @StartDate = MIN(Orderdate),
       @EndDate = MAX(OrderDate)
  FROM [Sales].[SalesOrderHeader];

WITH csrSalesByProduct
  (ProductID, OrderDate, NumSold)
AS
(SELECT ProductID, OrderDate, SUM(OrderQty)
  FROM [Sales].[SalesOrderHeader] SOH
    JOIN [Sales].[SalesOrderDetail] SOD
      ON SOH.SalesOrderID =
         SOD.SalesOrderDetailID
  GROUP BY ProductID, OrderDate),

csrAllDays (tDate)
AS
(SELECT @StartDate AS tDate
 UNION ALL
 SELECT DATEADD(DAY, 1, tDate)
   FROM csrAllDays
   WHERE tDate < @EndDate),

csrAllProducts (ProductID)
AS
(SELECT DISTINCT ProductID
  FROM Sales.SalesOrderDetail),

csrAllProductsByDays (ProductID, tDate)
AS
(SELECT ProductID, tDate
   FROM csrAllProducts
     CROSS JOIN csrAllDays)

  SELECT tDate, APBD.ProductID,
         ISNULL(NumSold, 0) AS TodaysSales,
         ISNULL(SUM(NumSold) OVER (
         PARTITION BY APBD.ProductID
         ORDER BY tDate
         ROWS BETWEEN 3 PRECEDING
         AND 3 FOLLOWING),0) AS WeekSales
   FROM csrSalesByProduct SBP
     RIGHT JOIN csrAllProductsByDays APBD
     ON SBP.ProductID = APBD.ProductID
     AND SBP.OrderDate = APBD.tDate
   ORDER BY ProductID, tDate
   OPTION (MAXRECURSION 0);
```

This query uses four CTEs; they produce, respectively, the sales totals by product for each day, the list of all days in the specified period, the distinct products from the SalesOrderDetail table, and the cross-join of the dates with the products.

tDate	ProductID	TodaysSales	WeekSales
2011-05-31 00:00:00.000	707	3	3
2011-06-01 00:00:00.000	707	0	3
2011-06-02 00:00:00.000	707	0	4
2011-06-03 00:00:00.000	707	0	5
2011-06-04 00:00:00.000	707	0	2
2011-06-05 00:00:00.000	707	1	3
2011-06-06 00:00:00.000	707	1	4
2011-06-07 00:00:00.000	707	0	4
2011-06-08 00:00:00.000	707	1	4
2011-06-09 00:00:00.000	707	1	3
2011-06-10 00:00:00.000	707	0	3
2011-06-11 00:00:00.000	707	0	3
2011-06-12 00:00:00.000	707	0	2
2011-06-13 00:00:00.000	707	1	2
2011-06-14 00:00:00.000	707	0	2

Figure 6. By including all combinations of products and dates, we can compute running weekly sales for each product.

Putting it to work

In general, my strategy for writing complex queries is to break the problem into small parts and start with the parts for which the solution is clear to me. Both VFP and SQL Server Management Studio make working this way easy because you can run something and quickly see the results.

Figuring out how to include all dates (or months or years), along with all products or customers or whatever, definitely called for that strategy. Now that I've experimented with code to produce complete lists like this, I'm really happy to have added this tool to my toolbox.

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.*