Combining Query Results

SQL Server lets you do something other than UNION when combining the results of two similar queries.

Tamar E. Granor, Ph.D.

We continue our look at T-SQL elements that aren't available in VFP's SQL with a look at the EXCEPT and INTERSECT operators that let you combine query results in a way other than putting them all together. These operators give you more elegant ways to solve some common problems.

Given that SQL is a set-based language, it shouldn't be a surprise to see set-related terms like UNION and INTERSECT. T-SQL also supports the set theory notion of a complement through the EXCEPT operator. All three operators are used for combining the results of multiple queries into a single result.

Combining query results with UNION

UNION works pretty much the same way in VFP's SQL as in SQL Server. It lets you dump the results of two or more queries into a single data set.

For example, the query in Listing 1 shows the CustomerID for any customer who placed orders in either June, 1997 or March, 1998. (Perhaps there was a special promotion we want to check on.) Two separate queries, one to collect customers who ordered in June, 1997, another to collect those who ordered in March, 1998, are combined.

Listing 1. The UNION clause lets you combine the results of multiple queries.

```
SELECT CustomerID ;
   FROM Orders ;
   WHERE MONTH(OrderDate) = 6 ;
     AND YEAR(OrderDate) = 1997 ;
   UNION ;
SELECT CustomerID ;
   FROM Orders ;
   WHERE MONTH(OrderDate) = 3 ;
     AND YEAR(OrderDate) = 1998
```

Of course, we probably want more information about those customers than their IDs. The query in Listing 2 (CustomersOrderedEitherMonth.PRG in this month's downloads) uses the UNIONed query in a derived table and collects the name of each relevant company.

Listing 2. You can use a query with a UNION inside other queries.

```
SELECT CompanyName ;
   FROM Customers ;
   JOIN (;
    SELECT CustomerID ;
      FROM Orders ;
      WHERE MONTH(OrderDate) = 6 ;
        AND YEAR(OrderDate) = 1997 ;
    UNION ;
    SELECT CustomerID ;
      FROM Orders ;
      WHERE MONTH(OrderDate) = 3 ;
        AND YEAR(OrderDate) = 1998) Ordered ;
   ON Customers.CustomerID = ;
      Ordered.CustomerID ;
   ORDER BY CompanyName ;
   INTO CURSOR csrOrderedInEither
```

The key rule for using UNION is that the field lists of all the UNIONed queries have to have the same number of fields and that corresponding fields (that is, fields in the same position in the field lists of the various queries) have to be of compatible data types.

Before VFP 8, "compatible data types" was quite strict. Fields actually had to be of the same data type, and the first query in the UNION determined the size of each field. But in VFP 8 and 9, the SQL engine is smarter and can handle similar data types, making intelligent decisions about the type and size of the final result. For example, if one query contains a character field and the other contains a memo field in the same position, the final result uses a memo field.

SQL Server has the same rule not only for UNION, but for INTERSECT and EXCEPT, and handles the necessary type conversions behind the scenes.

Listing 3 (in this month's downloads as CustomersOrderedEitherMonth.SQL) shows a SQL Server query analogous to the VFP example. It collects the names of those customers who placed orders in either June, 2007 or March, 2008. Here, the UNION is in a CTE (common table expression), the results of which are used to collect the actual customer names. In AdventureWorks 2008, there are two types of customers: individuals and stores, so the Customer table contains foreign keys to both Person and Store. Some Customers are linked to both, leading to the left joins in the main query.

Listing 3. UNION in SQL Server is analogous to UNION in VFP.

```
WITH csrEitherMonth (PersonID, StoreID)
AS
(SELECT Customer.PersonID, Customer.StoreID
   FROM Sales.Customer
     JOIN Sales.SalesOrderHeader
       ON Sales.Customer.CustomerID =
          Sales.SalesOrderHeader.CustomerID
   WHERE MONTH(SalesOrderHeader.OrderDate) = 6
     AND YEAR(OrderDate) = 2007
UNION
SELECT Customer.PersonID, Customer.StoreID
   FROM Sales.Customer
     JOIN Sales.SalesOrderHeader
       ON Sales.Customer.CustomerID =
          Sales.SalesOrderHeader.CustomerID
   WHERE MONTH(SalesOrderHeader.OrderDate) = 3
     AND YEAR(OrderDate) = 2008)

SELECT LTRIM(LastName) + ', ' +
       LTRIM(FirstName) AS Person,
       Name AS Store
   FROM csrEitherMonth
     LEFT JOIN Person.Person
       ON csrEitherMonth.PersonID =
          Person.Person.BusinessEntityID
     LEFT JOIN Sales.Store
       ON csrEitherMonth.StoreID =
          Store.BusinessEntityID
   ORDER BY Person, Store
```

By default, records that are identical in different initial results of UNION are consolidated in the final result. (A side effect of this behavior is that the final result is sorted on the first field.) You can override that behavior by including the ALL keyword.

Intersections instead of unions

What if the question is which customers made purchases in both of the specified months rather than either of them? Just as in Math class, we want to switch from a union to an intersection.

In VFP, have to take a totally different approach, using a pair of subqueries in the WHERE clause, as in Listing 4 (included in this month's downloads as CustomersOrderedTwoMonths.PRG). Each subquery collects data for a single month and the main query keeps only those records in the results of both subqueries.

Listing 4. Getting the intersection of two results in VFP calls for a pair of subqueries.

```
SELECT CompanyName ;
   FROM Customers ;
   WHERE CustomerID IN ;
     (SELECT CustomerID ;
        FROM Orders ;
          WHERE MONTH(OrderDate) = 6 ;
            AND YEAR(OrderDate) = 1997) ;
      AND CustomerID IN ;
     (SELECT CustomerID ;
        FROM Orders ;
          WHERE MONTH(OrderDate) = 3 ;
            AND YEAR(OrderDate) = 1998) ;
   ORDER BY CompanyName ;
   INTO CURSOR csrOrderedInBoth
```

In SQL Server, though, we can get the desired result by simply changing UNION to INTERSECT, as in Listing 5 (included in this month's downloads as CustomersOrderedTwoMonths.PRG). As you'd expect, INTERSECT performs an intersection between the two query results, so the final results contains only records that appear in both. As before, the INTERSECTed query is used in a CTE, and the main query uses that list of PersonIDs and StoreIDs to retrieve the actual names.

Listing 5. The INTERSECT keyword finds the intersection of the two results, keeping only those records that appear in both.

```
WITH csrBothMonths (PersonID, StoreID)
AS
(SELECT Customer.PersonID, Customer.StoreID
   FROM Sales.Customer
     JOIN Sales.SalesOrderHeader
       ON Sales.Customer.CustomerID =
          Sales.SalesOrderHeader.CustomerID
   WHERE MONTH(SalesOrderHeader.OrderDate) = 6
    AND YEAR(OrderDate) = 2007
INTERSECT
SELECT Customer.PersonID , Customer.StoreID
   FROM Sales.Customer
     JOIN Sales.SalesOrderHeader
       ON Sales.Customer.CustomerID =
          Sales.SalesOrderHeader.CustomerID
   WHERE MONTH(SalesOrderHeader.OrderDate) = 3
     AND YEAR(OrderDate) = 2008)

SELECT LTRIM(LastName) + ', ' +
       LTRIM(FirstName) AS Person,
       Name AS Store
   FROM csrBothMonths
     LEFT JOIN Person.Person
       ON csrBothMonths.PersonID =
          Person.Person.BusinessEntityID
     LEFT JOIN Sales.Store
       ON csrBothMonths.StoreID =
          Store.BusinessEntityID
   ORDER BY Person, Store
```

Finding unmatched records

One of the questions I see most frequently online is how to find every record in one table for which there's no matching record in a second table. For example, find all customers who placed no orders in a given time period.

In set theory, this is called the "complement" of the two sets. Note that unlike unions and intersections, complements are not commutative; that is, it matters which comes first. If you have sets A and B, A complement B (written "A-B") gives you all the members of A that are not in B, but B complement A is all the members of B that are not in A.

Using SQL, matching records in two tables is easy; just join the tables on the shared field, but finding those with no matches in a little trickier. There are a couple of ways to do it in VFP, but SQL Server provides an elegant and direct way to write this query.

The VFP approach I prefer uses a subquery, which selects all the relevant records in the second table. Then a NOT IN condition eliminates those

records from the final result. For example, Listing 6 shows a query that finds all Northwind customers who placed no orders in 1998. (It's included in this month's downloads as CustomerNoOrders.PRG.)

Listing 6. One way to find all customers who haven't placed an order in a given period is using NOT IN with a subquery.

```
SELECT CompanyName ;
   FROM Customers ;
   WHERE CustomerID NOT IN (;
      SELECT CustomerID ;
         FROM Orders ;
         WHERE YEAR(OrderDate) = 1998) ;
   ORDER BY Companyname ;
   INTO CURSOR csrNoOrders
```

You can accomplish the same thing using an outer join by testing the results for null in one of the fields of the second table. Listing 7 (CustomerNoOrdersJoin.PRG in this month's downloads) shows this approach to the same problem. The outer join between Customers and Orders provides a list of all customers. Those who placed any orders in 1998 are matched with the details of those orders. The ISNULL(OrderID) filter eliminates those records and keeps only the customers who had no matches.

Listing 7. You can also use an outer join to find unmatched records.

```
SELECT CompanyName ;
   FROM Customers ;
     LEFT JOIN Orders ;
       ON Customers.CustomerID =
          Orders.CustomerID ;
       AND YEAR(OrderDate) = 1998 ;
     WHERE ISNULL(OrderID) ;
   ORDER BY Companyname ;
   INTO CURSOR csrNoOrders
```

VFP appears to handle the two queries the same way internally. At least, SYS(3054) shows identical optimization.

You can solve the problem using the same approaches in SQL Server (see CustomersNoOrdersSubquery.SQL and CustomersNoOrdersLeftJoin.SQL in this month's downloads), but there's a better, more readable approach.

The EXCEPT operator gives you the complement of two query results, that is, all the records in the first result that are not in the second result. Listing 8 shows a query that returns the CustomerID for every customer that placed no orders in 2008.

Listing 8. Use EXCEPT to find all the records in one table that aren't matched in another.

```
SELECT CustomerID
   FROM Sales.Customer
EXCEPT
SELECT Sales.Customer.CustomerID
   FROM Sales.SalesOrderHeader
     JOIN Sales.Customer
       ON Sales.SalesOrderHeader.CustomerID =
          Sales.Customer.CustomerID
   WHERE YEAR(OrderDate)= 2008
```

The first query here simply pulls every CustomerID from the Customer table. The second collects the CustomerID of those customers who places orders in 2008. The EXCEPT operator then removes the second group from the first, giving us a list of customers with no orders in 2008.

The query that provides the individual and store names (shown in Listing 9 and included in this month's downloads as CustomersNoOrders.SQL) uses the query from Listing 8 in a CTE and then does the necessary joins to add the names.

Listing 9. Making the query using EXCEPT into a CTE makes it easy to collect additional information about the resulting records.

```
WITH csrNoSales (PersonID, StoreID)
AS

(SELECT PersonID, StoreID
   FROM Sales.Customer
EXCEPT
SELECT PersonID, StoreID
   FROM Sales.SalesOrderHeader
     JOIN Sales.Customer
       ON Sales.SalesOrderHeader.CustomerID =
          Sales.Customer.CustomerID
   WHERE YEAR(OrderDate)= 2008)

SELECT LTRIM(LastName) + ', ' +
       LTRIM(FirstName) AS Person,
       Name AS Store
   FROM csrNoSales
     LEFT JOIN Person.Person
       ON csrNoSales.PersonID =
          Person.BusinessEntityID
     LEFT JOIN Sales.Store
       ON csrNoSales.StoreID =
          Store.BusinessEntityID
ORDER BY Person, Store
```

As with many of the other items covered in this series, understanding INTERSECT and EXCEPT is probably easier than remembering to use them when the situation arise.

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.*