Here a subquery, there a subquery

VFP 9 allows you to use subqueries in lots of places

by Tamar E. Granor, Technical Editor

Visual FoxPro 9 continues the trend of expansion of SQL capabilities that began in VFP 8. (See my article in the August, 2003 issue for a look at the SQL changes in VFP 8.) VFP 9 removes many limits previous versions placed on SQL commands. For example, VFP 8 and earlier limit you to no more than 9 UNIONs in a single query; in VFP 9, UNIONs are unlimited.

One major area of enhancement is the use of subqueries. In VFP 8 and earlier, you're limited to one level for subqueries and they can appear only in the WHERE clause. VFP 9 allows subqueries to be nested, and to appear in the field list and FROM clause of a query and the SET clause of the SQL UPDATE command.

This article looks at the changes to subqueries in the previous paragraph; watch for future articles describing other changes to VFP's SQL sublanguage. The examples in this article use the TasTrade database that comes with VFP.

What is a subquery?

Some SQL commands need to use information that can't be pulled directly from the underlying tables. A subquery, which is a query within another SQL command, lets you collect that information on the fly.

A very common use for a subquery is to find all the records that are in one table, but not in another. For example, to get a list of TasTrade customers who didn't place an order in 1994, you could use this query:

```
SELECT Company ;
  FROM Customer ;
  WHERE Customer_ID NOT IN ;
    (SELECT Customer_ID FROM Orders ;
       WHERE YEAR(Order_Date) = 1994)
```

The subquery, enclosed in parentheses, creates a list of all customers who did place an order in 1994. The WHERE clause then extracts customers not in that list.

The subquery above can be executed on its own. In some cases, a subquery uses fields from the main query in its WHERE clause; such a subquery is said to be *correlated*.

For example, suppose you want to get information about each customer's most recent order. You want to know the Order_ID, the order date and the name of the shipper. The most common solution to this problem is to use two queries in sequence. The first query finds the most recent order for each customer, and then the second query extracts information about that order. However, a correlated subquery provides another way to solve this problem; Listing 1 demonstrates the technique.

Listing 1. Using a correlated subquery—The subquery here finds the date of the most recent order for each customer, allowing the main query to retrieve additional data for that order.

```
SELECT Orders.Order_ID, ;
       Customer.Company_Name as Cust_Name, ;
       Shippers.Company_Name AS Ship_Name, ;
       Orders.Order_Date ;
  FROM Orders ;
    JOIN Customer ;
      ON Orders.Customer_ID = Customer.Customer_ID ;
    JOIN Shippers ;
      ON Orders.Shipper_ID = shippers.Shipper_ID ;
  WHERE Orders.Order_Date = ;
    (SELECT MAX(Order_Date) ;
       FROM Orders Ord ;
       WHERE Orders.Customer_ID=Ord.Customer_ID );
  ORDER BY Cust_Name ;
  INTO CURSOR MostRecentOrders
```

Overall, subqueries make it possible to solve many problems with a single query rather than a series of queries. The enhancements in VFP 9 increase the number of problems that can be solved by a single query.

Nested Subqueries

In VFP 8 and earlier, subqueries cannot be nested, that is, a subquery cannot contain another subquery. VFP 9 removes this rule, allowing unlimited nesting of subqueries.

Because we've always had the limit, it may be hard initially to think of a situation in which you'd want to put a subquery in a subquery. In this example (FirstNotLast.PRG on this month's Professional Resource CD), the goal is to find products a company included in its first

TasTrade order, but not its most recent. Listing 2 shows a query that provides the desired results.

Listing 2. Nested subqueries—Nesting the subqueries here allows us to retrieve this information using a single query rather than a series of several queries.

```
SELECT Customer.Company_Name, Product_ID ;
  FROM Order_Line_Items ;
    JOIN Orders ;
      ON Order_Line_Items.Order_ID = Orders.Order_ID ;
    JOIN Customer ;
      ON Orders.Customer_ID = Customer.Customer_ID ;
    WHERE Orders.Order_Date = ;
      (SELECT MIN(Order_Date) ;
         FROM Orders Ord ;
          WHERE Orders.Customer_ID=Ord.Customer_ID );
      AND Product_ID NOT IN ;
        (SELECT Product_ID ;
           FROM Order_Line_Items OLILast;
             JOIN Orders OrdLast;
               ON OLILast.Order_ID = OrdLast.Order_ID ;
             JOIN Customer CustLast;
               ON OrdLast.Customer_ID = ;
                  CustLast.Customer_ID ;
           WHERE OrdLast.Order_Date = ;
             (SELECT MAX(Order_Date) ;
                FROM Orders Ord ;
                WHERE OrdLast.Customer_ID = ;
                      Ord.Customer_ID ) );
  INTO CURSOR FirstNotLast
```

This query uses a number of subqueries. The first subquery (SELECT MIN(Order_Date)…) finds the first order for a specified customer; it's correlated with the Orders table in the main query. After that, there's a nested subquery that finds the list of products in the customer's most recent order; it uses a nested subquery to identify the most recent order.

Nested subqueries are also useful when you're dealing with tables that contain a single hierarchy, such as a bill of materials or employment information. They let you climb several levels in a single query.

Computed fields

In VFP 8 and earlier, the only place you can put a subquery in a SELECT statement is in the WHERE clause. VFP 9 supports subqueries in two additional places, the field list and the list of tables.

A subquery in the field list of a query lets you compute a column of the result, much like including an expression in the field list. A subquery

used this way must return a single field (that is, have only one field in its field list) and no more than one record. If the subquery returns no records, the field is set to .null. in the result.

Using a subquery in the field list is especially useful when grouping is involved. Suppose you want to find the total amount of each customer's orders in a specified year. That's not difficult. The query in Listing 3 does the trick.

Listing 3. Computing total orders—This query computes the total orders in a specified year for each customer, but provides only the customer's ID.

```
SELECT Customer.Customer_ID, ;
       SUM(quantity*unit_price) AS TotalOrders ;
  FROM Customer ;
    JOIN Orders ;
      ON Customer.Customer_ID = Orders.Customer_ID ;
    JOIN Order_Line_Items;
     ON Orders.Order_ID = Order_Line_Items.Order_ID ;
  WHERE BETWEEN(Order_Date, ;
                DATE(m.nYear,1,1), ;
                DATE(m.nYear,12,31)) ;
  GROUP BY 1 ;
  INTO CURSOR CustomerTotal
```

The problem with this query is that the only information it provides about the customer is the Customer_ID field. It's likely that you also want to retrieve additional information about the customer, such as the company name, address, phone and fax.

You can get what you want without using a subquery, but it's clumsy. Your choices are to include all the additional fields from Customer in the GROUP BY clause or to wrap each of them in an aggregate function like MAX(). Because all the records in each group refer to the same customer, either approach produces correct results.

However, each field you add to the GROUP BY clause or wrap with MAX() slows the query down a little. VFP 9 offers a new solution to the problem: compute the total orders in the field list and omit the GROUP BY clause entirely. Listing 4 (CustomerTotal.PRG on the PRD) shows this approach.

Listing 4. Subquery in field list—Using a subquery in the field list to compute aggregate results works around the problem of retrieving additional data from a parent table.

```
SELECT Customer.Customer_ID, Customer.Company_Name, ;
       Customer.Address, Customer.City, ;
       Customer.Region, Customer.Postal_Code, ;
       Customer.Phone, Customer.Fax, ;
```

```
       (SELECT SUM(quantity*unit_price) ;
          FROM Orders ;
            JOIN Order_Line_Items;
              ON Orders.Order_ID = ;
                 Order_Line_Items.Order_ID ;
          WHERE BETWEEN(Order_Date, DATE(m.nYear,1,1), ;
                        DATE(m.nYear,12,31)) ;
            AND Customer.Customer_ID = ;
                Orders.Customer_ID ) AS yTotal ;
  FROM Customer ;
INTO CURSOR CustomerTotal
```

As with other calculated fields, use the AS keyword to specify a name for the field computed by the subquery. In the example, the new field is called yTotal.

In my tests, the query in Listing 4 was about 12% faster than a query using MAX() around each field from Customer. The query using MAX() was about 12% faster than listing all the Customer fields in the GROUP BY. As the number of fields from Customer increased, the advantage of using a subquery went up.

Derived tables

A subquery in the FROM clause is called a *derived table*; it allows you to create a cursor on the fly and join it with other tables. Like a subquery in the field list, a derived table is especially useful when you're working with aggregates and GROUP BY.

Consider the query in Listing 1 that retrieves information about the most recent order for each customer. A derived table offers an alternative approach that avoids a correlated subquery and can be faster in some situations. The query in Listing 5 (MostRecentOrderDetails.PRG on the PRD) retrieves the same data as the query in Listing 1.

Listing 5. Using a derived table—The ability to put a subquery in the FROM clause makes it easier to retrieve additional information when choosing records with MAX() or MIN().

```
SELECT Orders.Order_ID, ;
       Customer.Company_Name AS Cust_Name, ;
       Shippers.Company_Name AS Ship_Name, ;
       Orders.Order_Date ;
  FROM Orders ;
    JOIN ;
     (SELECT Customer_ID, MAX(Order_Date) AS Order_Date ;
        FROM Orders CheckOrderDate ;
        GROUP BY 1) RecentOrder ;
      ON Orders.Customer_ID = RecentOrder.Customer_ID ;
```

```
    AND Orders.Order_Date = RecentOrder.Order_Date ;
  JOIN Customer ;
    ON Orders.Customer_ID = Customer.Customer_ID ;
  JOIN Shippers ;
    ON Orders.Shipper_ID = shippers.Shipper_ID ;
ORDER BY Cust_Name ;
INTO CURSOR MostRecentOrders
```

Derived tables must be assigned a local alias; in the example, it's RecentOrders.

The derived table is joined to Orders based on the Customer_ID and the date of the most recent order.

VFP creates derived tables before executing the main query (which makes sense). As a result, these subqueries cannot be correlated, that is, they may not refer to fields of the tables in the main query.

Computing replacements with a subquery

Like SELECT, the SQL UPDATE command supports subqueries in the WHERE clause to let you determine which records are affected by the command. Beginning in VFP 9, UPDATE also allows you to use subqueries in the SET clause to compute the new values for a particular field. However, any UPDATE command can include only one subquery, so if you use a subquery in SET, you can't have one in WHERE, and only one field can be computed this way in a given UPDATE command.

Suppose you have a data warehouse table (SalesByProduct) that contains one record for each product. It's designed to show sales from a particular month at any time. That is, at the end of the month, you summarize that month's sales by product and store it in SalesByProduct. You could use a query to recreate the table each month, but the new ability to use a subquery in UPDATE offers another possibility, shown in Listing 6. (This code is in SubqueryInSet.PRG on this month's PRD. The PRD also contains CreateWarehouse.PRG, which creates the SalesByProduct table and initially populates it.)

Listing 6. Compute new values on the fly—The SQL UPDATE command now allows you to use a subquery in the SET clause to compute the new values.

```
UPDATE SalesByProduct ;
  SET TotalSales = (;
    SELECT NVL(SUM(quantity*unit_price),$0) ;
      FROM Order_Line_Items ;
        JOIN Orders ;
          ON Order_Line_Items.Order_ID = ;
```

```
            Orders.Order_ID ;
      WHERE MONTH(Order_Date) = nMonth ;
        AND YEAR(Order_Date) = nYear;
        AND Order_Line_Items.Product_ID = ;
            SalesByProduct.Product_ID)

UPDATE SalesByProduct ;
  SET UnitsSold = (;
    SELECT CAST(NVL(sum(quantity),0) as N(12)) ;
      FROM Order_Line_Items ;
        JOIN Orders ;
          ON Order_Line_Items.Order_ID = ;
             Orders.Order_ID ;
      WHERE MONTH(Order_Date) = nMonth ;
        AND YEAR(Order_Date) = nYear;
        AND Order_Line_Items.Product_ID = ;
            SalesByProduct.Product_ID)
```

The limit of one subquery per UPDATE command means that this approach requires two UPDATE commands, one to update the number of items sold and the other to update the total amount the items sold for.

Lots more changes

The ability to put subqueries in the field and FROM lists of SELECT and the SET clause of UPDATE makes it easier to write code for a variety of situations. But VFP 9 offers many more enhancements to the SQL portion of the language. In future articles, I'll look at some of the other changes.