August, 2004

Do More with VFP's SQL Commands

In my last article, I looked at new and expanded uses of subqueries in VFP 9. This article considers additional changes related to VFP's SQL sublanguage. While some of these involve subqueries, they're more focused on other areas.

With VFP 9, there are no limits

As I mentioned in my previous article, VFP 9 removes limits in a number of areas. Table 1 shows the changes in limits related to SQL commands. With the exception of the nesting of subqueries (which was discussed in my previous article), the old limits may seem generous to you, but each can be a problem in some circumstances.

Table 1. No limits—VFP 8 and earlier versions limit queries in a number of ways. VFP 9 lifts many of those limits.

Description	**Limit in VFP 8 (and earlier)**	**Limit in VFP 9**
Total number of joins and subqueries	9	No limit
Number of UNIONs	9	No limit
Number of tables and aliases referenced	30	No limit
Number of items listed in IN clause	24	Based on SYS(3055) setting
Nesting level for subqueries	1	No limit

The total number of joins and subqueries tends to come up most with a well-normalized database when you need to denormalize it for some process. Consider the Northwind database that comes with VFP 8 and 9. If you want to unfold the order data to create a record for each order of each product, containing all of the available information for that item (customer, shipper, supplier, etc.), you end up with the query in Listing 1 (included on this month's Professional Resource CD

as ManyTables.PRG). This query joins 11 tables. It runs in VFP 9, but fails in VFP 8 with error 1805 (SQL: Too many subqueries).

Listing 1. Using many tables—Flattening a normalized database can involve many tables.

```
SELECT ProductName, CategoryName, OrderDate, ;
       Customers.CompanyName AS CustomerName, ;
       OrderDetails.Quantity, ;
       OrderDetails.UnitPrice, ;
       Suppliers.CompanyName AS SupplierName, ;
       Employees.LastName, Employees.FirstName, ;
       Territories.TerritoryDescription, ;
       Region.RegionDescription, ;
       Shippers.CompanyName AS ShipperName;
  FROM Orders ;
    LEFT JOIN Customers ;
      ON Orders.CustomerID = Customers.CustomerID ;
    LEFT JOIN OrderDetails ;
      ON Orders.OrderID = OrderDetails.OrderID ;
    LEFT JOIN Products ;
      ON OrderDetails.ProductID = Products.ProductID ;
    LEFT JOIN Categories ;
      ON Products.CategoryID = Categories.CategoryID ;
    LEFT JOIN Suppliers ;
      ON Products.SupplierID = Suppliers.SupplierID ;
    LEFT JOIN Shippers ;
      ON Orders.ShipVia = Shippers.ShipperID ;
    LEFT JOIN Employees ;
      ON Orders.EmployeeID = Employees.EmployeeID ;
    LEFT JOIN EmployeeTerritories ;
      ON Employees.EmployeeID = EmployeeTerritories.EmployeeID ;
    LEFT JOIN Territories ;
      ON EmployeeTerritories.TerritoryID = Territories.TerritoryID ;
    LEFT JOIN Region ;
      ON Territories.RegionID = Region.RegionID ;
  ORDER BY Products.ProductID, Orders.OrderDate ;
  INTO CURSOR Unfolded
```

While this example is a little contrived, it's not unusual to need to joins more than 9 tables when denormalizing. The same situation can call for use of more than 30 tables or aliases. The limit on UNIONs can be a problem when combining data from disparate sources.

The limit of items in the IN clause is rarely a problem when you write the code, but there are situations where another application (such as Crystal Reports) queries VFP data and can generate a long IN clause.

The IN clause lets you filter data based on membership in a list of items. For example, you might find those people whose last names are colors using a query like this:

```
SELECT cFirst, cLast ;
   FROM Person ;
   WHERE UPPER(cLast) IN ("BLACK", "BROWN", "GREEN", "SILVER", "WHITE")
```

If the list had too many items, it would be simple to put them in a cursor and use a join instead of listing them explicitly in the IN clause:

```
CREATE CURSOR Names (cName C(25))
INSERT INTO Names VALUES ("BLACK")
INSERT INTO Names VALUES ("BROWN")
INSERT INTO Names VALUES ("GREEN")
INSERT INTO Names VALUES ("SILVER")
INSERT INTO Names VALUES ("WHITE")

SELECT cFirst, cLast ;
   FROM FORCE Names ;
     JOIN Person ;
       ON UPPER(cLast) = RTRIM(cName)
```

However, some applications that use VFP data through the OLE DB provider generate queries using the IN clause. For those applications, the limit of 24 items is a major problem.

In VFP 9, the limit has been raised and is configurable. To increase the limit, raise the setting of SYS(3055). In my tests, it appears that using the default SYS(3055) setting of 320, the IN clause can contain 154 items. SYS(3055) is increased in increments of 8, and my tests indicate that each increase of 8 for the function allows you to add up to four more items to the IN clause.

The processing of the IN clause has also been changed in VFP 9. In previous versions, internally, IN was implemented using the INLIST() function. This is no longer true, as INLIST() is still limited to 24 values. IN now uses a strategy that does comparisons only until the first match is found; this means you can speed execution by putting the most likely values earlier in the list.

Correlated subqueries and grouping

In earlier versions of VFP, a correlated subquery cannot include a GROUP BY clause. This is rarely an issue because most often, the field you'd want to group by is the same one used to correlate with the main query. Thus, there's no reason to group the results.

For example, the query in Listing 2 retrieves information about the most recent order for each customer. The subquery finds the date of each customer's most recent order. Ordinarily, to do that, the subquery would be grouped on Customer_ID. However, since the

subquery is correlated on Customer_ID, each time it's executed, it looks only at orders for a single customer and no GROUP BY clause is needed.

Listing 2. No need to group in a correlated subquery—This query uses a subquery correlated on the customer ID, so there's no need to add a GROUP BY clause to the subquery to find each customer's most recent 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
```

However, there are some situations where you want to group on one field but correlate on another. This is most likely to happen when you want to check for the existence of some situation.

Consider the query in Listing 3 (SuspiciousOrders.PRG on this month's PRD); it collects information about "suspicious" orders. A suspicious order is any order totaling more than $4000 that's shipped to an address other than the customer's company address. The subquery finds orders that meet the criteria. It's correlated on the Customer_ID field, but groups on Order_ID to compute the total for an order.

Listing 3. Suspicious orders—This query uses a correlated subquery with a GROUP BY clause to find orders of more than $4000 being shipped to something other than the customer's address.

```
SELECT Company_Name, Ord.Order_ID, Ord.Order_Date ;
 FROM Customer ;
   JOIN Orders Ord;
     ON Customer.Customer_ID = Ord.Customer_ID ;
  WHERE Ord.Order_ID IN (;
    SELECT Orders.Order_ID;
    FROM Orders ;
      JOIN Order_Line_Items ;
        ON Orders.Order_ID=Order_Line_Items.Order_ID ;
        AND Orders.Customer_ID=Customer.Customer_ID ;
        AND Orders.Ship_to_Address <> Customer.Address ;
    GROUP BY Orders.Order_ID ;
    HAVING SUM(Quantity*Unit_Price)> 4000 ) ;
```

```
  INTO CURSOR Suspicions
```

Use TOP N in subqueries

The TOP N clause lets you return only partial results from a query. For example, in a query that computes the annual sales by salesman, you could limit the result to the top 10 sellers. The selection is made based on the order of the result set (determined by the ORDER BY clause) and can choose either a specific number of records or a percentage of the result set. For example, the query in Listing 4 finds the lowest 10% of products based on number sold. (You get the bottom 10% here because they're in ascending order.)

Listing 4. TOP N lets you limit results—This query computes the number of each product sold and returns only the bottom 10 percent.

```
SELECT TOP 10 PERCENT Product_ID, ;
       SUM(Quantity) AS nSales ;
   FROM Order_Line_Items ;
   GROUP BY Product_ID ;
   ORDER BY nSales ;
   INTO CURSOR LowSales
```

Suppose you're considering discontinuing those items. You want to know which of your customers have bought them so you can ensure you won't lose customers. To find that information, you need to match the results of this query with data in the Customer and Orders tables. In VFP 8 and earlier versions, you can't use the TOP N clause in a subquery, so you need two queries to find this information. You'd run the query in Listing 3 and then use another query to join that result with the Customer and Orders tables.

VFP 9 permits TOP N in subqueries, as long as the subqueries are not correlated. So, you can find the desired information with a single query. One solution, shown in Listing 5 and included on the PRD as Discontinue.PRG, uses a derived table.

Listing 5. Top Sellers—This query extracts the names of the customers who bought low-selling products by creating a derived table, using a TOP N subquery.

```
SELECT DISTINCT Company_Name, English_Name ;
   FROM Customer ;
     JOIN Orders ;
       ON Customer.Customer_ID = Orders.Customer_ID ;
     JOIN Order_Line_Items ;
       ON Orders.Order_ID = Order_Line_Items.Order_ID ;
     JOIN ( ;
      SELECT TOP 10 PERCENT Product_ID, ;
             SUM(Quantity) AS nSales ;
        FROM Order_Line_Items ;
```

```
      GROUP BY Product_ID ;
      ORDER BY nSales );
      AS LowSales ;
   ON Order_Line_Items.Product_ID = ;
      LowSales.Product_ID ;
  JOIN Products ;
    ON LowSales.Product_ID = Products.Product_ID ;
ORDER BY English_Name, Company_Name ;
INTO CURSOR BoughtLowSellers
```

Correlated updates

The SQL UPDATE command provides an easy way to change data in many records at once. However, in VFP 8 and earlier, when you want to indicate which records to update based on data in another table, you have to use a subquery.

VFP 9 offers an alternative way to specify the records to be updated. The FROM clause now accepts multiple tables with join conditions. Only those records in the target table (the one listed after the UPDATE keyword) that are specified by the join conditions (combined with the filter conditions in the WHERE clause) are updated.

Consider a data warehouse (SalesByProduct) designed to hold sales by product for a single month. (This is the same data warehouse I described in my last article.) The new FROM clause makes it possible to update this table with a single command, shown in Listing 6 (and included on this month's PRD as CorrelatedUpdate.PRG).

Listing 6. Correlated update—This command uses a derived table and the ability to join tables in the UPDATE command to put a new month's data into the data warehouse table.

```
UPDATE SalesByProduct ;
   SET SalesByProduct.TotalSales = ;
         NVL(MonthlySales.TotalSales, $0), ;
      SalesByProduct.UnitsSold = ;
         NVL(MonthlySales.UnitsSold, 0) ;
   FROM SalesByProduct ;
     LEFT JOIN (SELECT Order_Line_Items.Product_ID, ;
          SUM(Quantity*Order_Line_Items.Unit_Price) ;
            AS TotalSales, ;
          SUM(Quantity) AS UnitsSold ;
      FROM Order_Line_Items ;
        JOIN Orders ;
          ON Order_Line_Items.Order_ID = Orders.Order_ID ;
          AND (MONTH(Order_Date) = nMonth ;
          AND YEAR(Order_Date) = nYear) ;
      GROUP BY 1);
   AS MonthlySales ;
   ON SalesByProduct.Product_ID = MonthlySales.Product_ID
```

Correlated deletes

Like the UPDATE command, through VFP 8, the SQL DELETE command works with a single table and any connection between the records in that table and data in any other table has to be specified using a subquery. VFP 9 lets you specify multiple tables with join conditions in the DELETE command to determine which records to delete.

In VFP 8 and earlier, the syntax for DELETE is:

```
DELETE FROM Table WHERE Conditions
```

With only a single table, there's no need to specify which table is the target of the deletion. The ability to list multiple tables in VFP 9 means that we need to indicate from which table records are to be deleted. The rule is that whenever the FROM clause contains more than one table, the name of the target table must be listed between DELETE and FROM, like this:

```
DELETE Table ;
  FROM Table ;
    JOIN AnotherTable ;
      ON JoinConditions
```

Neither the TasTrade nor Northwind databases that come with VFP offer opportunities to demonstrate this functionality; they're designed with the assumption that records won't be deleted, just marked unused, and have referential integrity rules to make it very difficult to delete data.

However, assume you have Products and Supplier tables with a structure like the corresponding tables in TasTrade. Due to shipping problems, you've decided to eliminate all products coming from Australia. The command in Listing 7 will do the trick. It's included on the PRD as DeleteProducts.PRG; the code there also creates and populates cursors to demonstrate.

Listing 7. Deleting based on outside data—The new FROM clause of DELETE lets you decide which records to delete based on data from other tables.

```
DELETE Products ;
   FROM Products ;
     JOIN Supplier ;
       ON Products.Supplier_ID = Supplier.Supplier_ID ;
     WHERE UPPER(Supplier.Country) = "AUSTRALIA" ;
```

Use field names in ORDER BY

The UNION clause allows a single query to combine results from several SELECTs. When a query has a UNION clause, there's a single ORDER BY clause that applies to the overall result of the UNION. In VFP 8 and earlier, the ORDER BY clause had to refer to fields by their position in the result. VFP 9 allows you to use the field names instead. This makes the code easier to read and easier to maintain.

For example, consider the query in Listing 8 (included on the PRD as AllCcompanies.PRG) that creates a list of the companies TasTrade does business with. (This might be needed, for example, for a holiday mailing list or for legal reasons.) In earlier versions, the ORDER BY clause would read ORDER BY 6, 3 rather than ORDER BY Country, City. If the list of fields in the query changed, the ORDER BY clause might no longer be correct.

Listing 8. Ordering in a UNION—In VFP 9, the ORDER BY clause in a query using UNION can refer to fields by their name, rather than their position in the field list.

```
SELECT Company_Name, Address, City, Region, ;
       Postal_Code, Country ;
   FROM Customer ;
UNION ;
SELECT Company_Name, Address, City, Region, ;
       Postal_Code, Country ;
   FROM Supplier ;
ORDER BY Country, City ;
INTO CURSOR AllCompanies
```

In Listing 8, the fields from the original tables have the same names, so there's no question as to what names to specify. However, sometimes in a UNIONed query, the matching fields have different names. When that happens, VFP chooses the field names from the last query in the UNION to name fields in the result. The ORDER BY clause must use those names.

Rather than relying on VFP's behavior, a better practice is to use the AS keyword to rename the fields in the query so that there's no question as to the field names in the result.

Insert from UNIONed query

VFP 8 introduced the INSERT INTO SELECT syntax that allows you to add records to a table directly from a query. However, the query could not use UNION. VFP 9 lifts this restriction.

For example, suppose you have a data warehouse for TasTrade that contains annual sales information by employee and product. That is, this warehouse contains one record for each product-employee combination for each year, showing the units sold and total sales. There's also a record for each employee each year, showing the total number of items sold and total sales volume for that employee. (This is a different data warehouse than the one described earlier in this article.) At the end of each year, you want to update the warehouse with all data from the preceding year. The structure of the data warehouse is:

```
Product_ID C(6)
Employee_ID C(6)
nYear N(4)
nTotalSales Y
nUnitsSold N(8)
```

The command in Listing 9 adds all the necessary records. The first query in the UNION computes the annual total for each product-employee combination, while the second query computes the annual employee totals. This query, along with code to create the warehouse (as a cursor, though you'd want an actual table in production) in included on this month's PRD as WarehouseUnion.PRG.

Listing 9. Updating with a UNION—In VFP 9, the query used in an INSERT INTO command can include a UNION.

```
INSERT INTO Warehouse ;
SELECT CrossProd.Product_ID, ;
       CrossProd.Employee_ID, ;
       m.nYear as nYear, ;
       NVL(nUnitsSold, 0), NVL(nTotalSales, $0);
   FROM (SELECT Employee.Employee_ID, ;
                Products.Product_ID ;
      FROM Employee, Products) AS CrossProd ;
     LEFT JOIN ( ;
      SELECT Product_ID, Employee_ID, ;
             SUM(Quantity) AS nUnitsSold, ;
             SUM(Quantity * Unit_Price) AS nTotalSales ;
         FROM Orders ;
           JOIN Order_Line_Items ;
             ON Orders.Order_ID = ;
                Order_Line_Items.Order_ID ;
         WHERE YEAR(Order_Date) = m.nYear ;
         GROUP BY Product_ID, Employee_ID ) ;
         AS AnnualSales ;
      ON CrossProd.Employee_ID = ;
         AnnualSales.Employee_ID ;
      AND CrossProd.Product_ID = AnnualSales.Product_ID ;
UNION ;
SELECT "Total" AS Product_ID, Employee.Employee_ID, ;
```

```
      m.nYear AS nYear, ;
      CAST(NVL(SUM(Quantity),0) as N(12)) ;
        AS nUnitsSold, ;
      NVL(SUM(Quantity * Unit_Price), $0) ;
        AS nTotalSales ;
  FROM Orders ;
    JOIN Order_Line_Items ;
      ON Orders.Order_ID = Order_Line_Items.Order_ID ;
     AND YEAR(Order_Date) = m.nYear ;
    RIGHT JOIN Employee ;
      ON Orders.Employee_ID = Employee.Employee_ID ;
  GROUP BY Employee.Employee_ID ;
ORDER BY 2, 1
```

Note the use of derived tables in the first query in the union. Without that feature, this code would require multiple queries. The code also uses a feature I've rarely used in production, generating a cross-product (also known as a Cartesian join) between two tables. It's necessary in order to get a complete list of employees matched with products.

Get your results faster

It seems that every version of VFP makes something faster and VFP 9 is no exception. In addition to the items already mentioned that provide faster ways of doing a particular task, two query-related items have been speeded up.

The LIKE operator lets you compare strings (much like the = operator). It accepts two wildcard characters in the expression being compared. The "_" wildcard indicates a single character, while the "%" wildcard stands for 0 or more characters. In VFP 8 and earlier, LIKE with a "%" is only partially optimizable. In VFP 9, a query like this one can be fully optimized, if the appropriate indexes exist:

```
SELECT Customer_ID, Company_Name ;
  FROM Customer ;
  WHERE UPPER(Company_Name) LIKE "P%" ;
  INTO CURSOR PCompanies
```

This improvement is most significant in the case where VFP takes a shortcut in providing query results. When the FROM clause of a query contains a single table, and the field list uses only fields from that table (no expressions), and the WHERE clause is fully optimizable, VFP filters the original table rather than creating a new file. (This is a mixed blessing; it's blazingly fast, but the resulting cursor can't be used in some situations. You can turn this behavior off by adding the

NOFILTER clause to the query.) Prior to VFP 9, a query involving LIKE with "%" wasn't eligible for this shortcut. Now it is.

The second item that has been accelerated is the computation of TOP n results. When a query uses TOP n or TOP n PERCENT, the VFP engine has to compute the whole result set and then figure out which records are at the top. The algorithm used to do so has been improved considerably in VFP 9. However, you're likely to see a difference only when working with large tables.

I tested with two tables, one containing nearly 75,000 records and the other containing more than a million records. In each case, I selected the top 20 items. With the 75,000 record table, the timing was the same in VFP 9 and VFP 8. However, with the million record table, VFP 9 found the top 20 records in about one-third the time it took in VFP 8.

Querying buffered data

The last item solves a problem that's been around since VFP 3. When a table is buffered and you run a query against that table, VFP has always queried the data on disk, not the current buffer. Xbase code (like SEEK or CALCULATE) was needed to look up data in the buffer or compute totals for buffered data.

VFP 9 offers an alternative. The new WITH clause lets you indicate whether or not to look at buffered data. The query in Listing 10 uses the new syntax to ensure it sees any changes to Customer in the current buffer.

Listing 10. Looking at buffered data—The new WITH (Buffering = .T.) clause lets a query use data in buffers rather than data on disk.

```
SELECT Country, CNT(*) ;
   FROM Customer WITH (Buffering = .T.) ;
   GROUP BY Country ;
   INTO CURSOR BufferedCount
```

The PRD contains QueryWithBuffering.PRG, a program that demonstrates the effects of the WITH clause. The program is self-contained and cleans up after itself, leaving your TasTrade data unchanged.

The bottom line

The SQL portion of VFP continues to grow. The more I work with the new VFP 9 features, the more possibilities I see for using them.