August, 2003

Using SQL the VFP 8 way

Changes to the SQL engine in VFP 8 may break some existing code and offer new possibilities

by Technical Editor Tamar E. Granor, Ph.D.

The addition of several SQL commands in FoxPro 2.0 was a revolutionary change. It introduced a whole new way of looking at data. VFP 3 added a few more SQL commands to the language. Since then, despite a few minor changes, FoxPro's SQL sub-language has been pretty static.

That is, until VFP 8. VFP introduces several changes ranging from significant to minor, and adds one major new piece of functionality.

Grouping, aggregation and field lists

Since it was introduced, VFP's SQL SELECT command has allowed you to violate a rule that's enforced in many other implementations (including SQL Server). When a query includes a GROUP BY clause, most languages require that every item in the field list is either listed in the GROUP BY clause or uses one of the aggregate functions (CNT(), SUM(), AVG(), MIN(), MAX()). Until VFP 8, FoxPro has never enforced this rule, though you break it at your own risk. In VFP 8, the rule applies, by default.

Here's an example, using the TasTrade database that comes with VFP. In VFP 7 and earlier, the following query works to get a list of companies and their most recent order:

```
SELECT Customer.Customer_ID, Company_Name, ;
       MAX(Orders.Order_Date) ;
  FROM customer ;
    JOIN orders ;
      ON customer.customer_id = orders.customer_id ;
  GROUP BY 1 ;
  INTO CURSOR RecentOrder
```

The query works and gives accurate results, so why is it a problem? The answer lies in what GROUP BY actually does. When a query includes a GROUP BY clause, first all the records that fit the join and filter (WHERE) conditions are assembled into an intermediate result. Then, records that have the same value in the fields specified in the GROUP BY clause are consolidated into a single record, applying the

specified aggregate functions as appropriate. In the example, the first few records of the intermediate result look like this:

```
ALFKI  Alfreds Futterkiste                 08/21/92
ALFKI  Alfreds Futterkiste                 07/19/94
ALFKI  Alfreds Futterkiste                 08/27/94
ALFKI  Alfreds Futterkiste                 09/06/94
ALFKI  Alfreds Futterkiste                 12/09/94
ALFKI  Alfreds Futterkiste                 02/07/95
ANATR  Ana Trujillo Emparedados y helados  08/12/93
ANATR  Ana Trujillo Emparedados y helados  07/02/94
ANATR  Ana Trujillo Emparedados y helados  10/22/94
```

The GROUP BY clause takes the five records with Customer_ID ALFKI and creates a single record. For the order date column, the MAX() function is applied and that field is given the value 12/09/94. But what happens for the Company_Name field? In this case, it doesn't matter because Company_Name is the same for each of the five records. The answer, though, is that VFP arbitrarily takes the value from the last record in the group.

Here's an example where it matters. This time, we're looking for the first order from each customer and we want to know the order date, the order number, and the shipping charge.

```
SELECT Customer.Customer_ID, Orders.Order_Number, ;
       MIN(Orders.Order_Date), Orders.Freight ;
  FROM customer ;
    JOIN orders ;
      ON Customer.Customer_ID = Orders.Customer_ID ;
  GROUP BY 1 ;
  INTO CURSOR FirstOrder
```

In this case, the value in the Order_Number and Freight columns are totally misleading (and, in fact, always wrong). The data shown comes from the last record for that customer, no matter what its order date is.

The examples leave several questions. What are the new rules? How do we get the results we need? What about code that already works?

Let's start with the rules. Very simply, VFP 8 enforces the ANSI standard rule that every item in the field list must either be included in the GROUP BY clause or include an aggregate function. When you violate the rule, you get ERROR 1807, "SQL: GROUP BY clause is missing or invalid." Both of the examples above give error 1807 in VFP 8.

To get the results you need, there are two options, which depend on the situation. In some cases, you can add the extra field to the GROUP BY clause without changing the results. This is a possibility for the first example:

```
SELECT Customer.Customer_ID, Company_Name, ;
       MAX(Orders.Order_Date) ;
  FROM customer ;
    JOIN orders ;
      ON customer.customer_id = orders.customer_id ;
  GROUP BY 1, 2 ;
  INTO CURSOR RecentOrder
```

Since the Company_Name is unique to the Customer_ID, grouping on both fields gives you the same result as grouping on just the Customer_ID.

For this example, there's another solution, as well. You can apply the MIN() or MAX() function to the extra field, like this:

```
SELECT Customer.Customer_ID, MIN(Company_Name), ;
       MAX(Orders.Order_Date) ;
  FROM customer ;
    JOIN orders ;
      ON customer.customer_id = orders.customer_id ;
  GROUP BY 1 ;
  INTO CURSOR RecentOrder
```

Again, since the value is the same for every record, finding the minimum or maximum value has no effect.

There may be optimization consequences, so it's worth testing both solutions with your data to see which gives better performance.

For the second example, you could use MIN() for both the order number and order date fields since order numbers do, in fact, go up as time passes. But that doesn't solve the more general problem of how to find the other data associated with the first order, such as the correct freight charge. The solution, in general, is to use two queries in sequence, like this:

```
SELECT Customer.Customer_ID, ;
       MIN(Orders.Order_Date) AS Oldest ;
  FROM Customer ;
    JOIN Orders ;
      ON Customer.Customer_ID = Orders.Customer_ID ;
  GROUP BY 1 ;
  INTO CURSOR FirstOrder
SELECT Customer.Customer_ID, Oldest, ;
       Orders.Order_Number, Orders.Freight ;
```

```
  FROM Customer ;
    JOIN Orders ;
      ON Customer.Customer_ID = Orders.Customer_ID ;
    JOIN FirstOrder ;
      ON Orders.Customer_ID = FirstOrder.Customer_ID ;
      AND Orders.Order_Date = FirstOrder.Oldest ;
  INTO CURSOR OldestOrders
```

Be aware that you should be using this approach not only in VFP 8, but in earlier versions as well. Queries that work in VFP 7, but fail in VFP 8, may actually be giving you bad results in VFP 8.

What if you've checked your results for an existing application and the queries work as you expect, but you want to upgrade to VFP 8? Do you have to apply one of these solutions to each query. No, the Fox team gave us a way to make existing code work. The new SET ENGINEBEHAVIOR command and the synonymous function SYS(3099) let you determine whether the SQL engine uses the new rules or sticks with the old ones.

The syntax for SET ENGINEBEHAVIOR is:

```
SET ENGINEBEHAVIOR 70 | 80
```

That, is you set it to either the 7.0 or the 8.0 behavior. For SYS(3099), the syntax is:

```
nOldValue = SYS(3099, 70 | 80)
```

You pass the setting you want and the function returns the old setting.

You'd be wise to change the engine behavior only when have a body of existing code that needs to run in VFP 8. (Be aware that this setting affects some other items, which are described below.) For new code, you're strongly encouraged to conform to the new rules.

Working with Memo and General fields

Memo and General fields have always needed special treatment in VFP. VFP 8 highlights one area where they've been getting special treatment in earlier versions, though it wasn't acknowledged.

You can no longer include Memo and General fields in queries that use the DISTINCT clause. That includes both SELECT DISTINCT and UNION DISTINCT. For example, the following query is valid in VFP 7 and earlier versions, but not in VFP 8:

```
SELECT DISTINCT Customer_ID, Order_Date, Notes ;
```

```
  FROM Orders
```

There's a good reason for prohibiting such a query. Like a GROUP BY clause, SELECT DISTINCT takes records that match and consolidates them into a single record. Unlike GROUP BY, however, all fields are considered. However, it turns out that in VFP 7 and earlier, Memo and General fields were never compared in this situation. So, in the example above, if you have two orders from the same customer on the same date, but with different notes, you'd get one record in the result, and it would include the notes from one of the orders. In VFP 8, the query generates error 34, "Operation is invalid for a Memo, General, or Picture field."

When you combine two queries with the UNION clause, by default, UNION DISTINCT is used. So this new rule may actually affect queries that don't explicitly include the UNION keyword. Here's an example (albeit a poor one since you can get the same results by using an OR in the WHERE clause).

```
SELECT First_Name, Last_Name, Notes ;
  FROM Employee ;
  WHERE Country="UK" ;
UNION ;
SELECT First_Name, Last_Name, Notes ;
  FROM Employee ;
  WHERE Country="USA"
```

This change is also controlled by the SET ENGINEBEHAVIOR and SYS(3099) functions, so you can turn it off to keep existing code working. (Do make sure that your code doesn't assume that memo fields are being checked for uniqueness.) In many cases, you don't really care whether you check UNIONed queries for identical records, so you can also solve the problem by specifying UNION ALL.

Type coercion in UNIONs

Writing queries involving the UNION clause has always been tricky because the field lists have to be "union-compatible." In VFP 8, the definition of "union-compatible" has been loosened, making UNION more flexible and useful.

Naturally, when you combine two or more queries using UNION, the field lists of the queries have to contain the same number of items. In VFP 7 and earlier, corresponding fields must be the same type and size, and the field list of the result is determined by the first query in the union. (If the field list of a query after the first contains an expression, it can be shorter than the corresponding field in the first

query, but not longer.) For example, suppose you have two mailing lists that need to be combined, defined like this:

```
CREATE TABLE List1 (cFirstName C(10), cLastName C(15))
CREATE TABLE List2 (cFirstName C(12), cLastName C(12))
```

This attempt at a UNION of these two cursors generates error 1851 ("SELECTs are not UNION compatible"):

```
SELECT cFirstName, cLastName ;
  FROM List1 ;
UNION ;
SELECT cFirstName, cLastName ;
  FROM List2
```

In VFP 8, the rules have changed in two ways. First, rather than setting field sizes based on the first query in the UNION, each query is examined and fields are set to the longest size found. So, the example above works in VFP 8 and results in a cursor where cFirstName is 12 characters and cLastName is 15 characters.

The second change is even better. Corresponding fields no longer have to be exactly the same type—compatible types are converted behind the scenes. For example, if a particular field in one query is date and the same field in another query in the UNION is datetime, the resulting field is datetime. Similarly, if corresponding fields are character and memo, the result is a memo field. The Help for SELECT-SQL contains a complete list of which types can be coerced.

Adding selected records

The INSERT INTO command makes it easy to add records to a table, but until VFP 8, to add multiple records with INSERT INTO, you had to put the data into an array and use the FROM ARRAY clause. VFP 8 introduces a variation that allows INSERT INTO to get the records to add from a query. Since the query can involve multiple tables, this provides a far more capable facility than APPEND FROM.

Suppose you're creating a data warehouse. At the end of each month, you want to add summary information for that month's orders to an existing table. In VFP 7 and earlier, you'd need several commands. One way to do it is like this (though this approach works only if you're sure the array won't total more than 65,000 elements):

```
SELECT Customer.Company_Name, Orders.Order_Number, ;
       SUM(Order_Line_Items.Quantity * ;
           Order_Line_Items.Unit_Price);
  FROM Customer ;
```

```
    JOIN Orders ;
    JOIN Order_Line_Items ;
      ON Orders.Order_ID = Order_Line_Items.Order_ID ;
      ON Customer.Customer_ID = Orders.Customer_ID ;
  WHERE BETWEEN(Order_Date, dMonthStart, dMonthEnd) ;
  GROUP BY 1,2 ;
  INTO ARRAY aSummary
INSERT INTO WareHouse FROM ARRAY aSummary
```

With the new version of INSERT INTO, you can reduce this to one command and remove the concern about array size:

```
INSERT INTO WareHouse ;
  SELECT Customer.Company_Name, Orders.Order_Number, ;
         SUM(Order_Line_Items.Quantity * ;
             Order_Line_Items.Unit_Price);
  FROM Customer ;
    JOIN Orders ;
    JOIN Order_Line_Items ;
      ON Orders.Order_ID = Order_Line_Items.Order_ID ;
      ON Customer.Customer_ID = Orders.Customer_ID;
  WHERE BETWEEN(Order_Date, dMonthStart, dMonthEnd) ;
  GROUP BY 1,2
```

Of course, the WareHouse table needs to exist before either approach. If you want to test the code above, use this line to create an appropriate cursor:

```
CREATE CURSOR WareHouse ;
  (Company_Name C(40), Order_Number C(6), Order_Total Y)
```

What else is new?

There's one more small change in SELECT. When you use the LIKE operator for comparisons, you can specify "_" as a wildcard to match a single character. In VFP 7 and earlier, "_" doesn't match trailing spaces. In VFP 8, a trailing blank matches the "_" wildcard. For example, in VFP 7, this query returns only customers in the US, while in VFP 8, it includes UK customers as well:

```
SELECT Company_Name, Country ;
  FROM Customer ;
  WHERE Country LIKE "U__"
```

This change is affected by the engine behavior setting, so if you need the old behavior, you can get it.

In addition to all these changes in the language, the Query/View Designer underwent a major overhaul in VFP 8. While the details of the changes are beyond the scope of this article, highlights include the

ability to edit code in the SQL window and have your changes reflected in the designer itself, and major changes to the Joins page. If you've been avoiding this tool because it couldn't handle the queries you need to write, it's time to take another look.