After a long wait, OUTER JOINs are finally supported in Visual FoxPro 5.0! This article discusses the syntax and techniques to employ this powerful new feature.

For years, one of my most frequent complaints about the SQL implementation in FoxPro has been its lack of support for OUTER JOINs. I have really come to appreciate Jerry Ela's article, "Stalking the Outer Join," in the September 1992 issue of FoxTalk for explaining how to simulate an OUTER JOIN in FoxPro 2.0. (Thanks, Jerry, wherever you are!) I'd grab this article off the shelf several times a year, having memorized its location. Eventually I memorized the technique (I'm a slow learner), and it was a red-letter day when I successfully created (without referring to Jerry's article!) a three-table OUTER JOIN in the Command window, and it ran correctly the first time! This technique involved performing multiple queries, or use of subqueries, and performing a UNION of the results (yuck!). It worked, but it wasn't pretty.

Now, with the release of Visual FoxPro 5.0, Microsoft has expanded its SQL implementation to include OUTER JOINs.

What is an Outer Join?
When performing a simple two-table query in versions of FoxPro prior to Visual FoxPro version 5.0, the type of join between the two tables is called a "natural" or "inner" join. If you apply the ideas of set theory to the operation, you're producing a result set that is an intersection of the two tables. This can be represented by a Venn diagram, as shown in Figure 1.

![Figure 1: Intersection of two tables represented by a Venn diagram.](image)

Thus, the result set for an INNER JOIN ignores any record from either table that doesn't "join," that is, that doesn't satisfy the join condition specified in the WHERE clause of the SELECT statement. In other words, parent records without children and child records without a corresponding parent are not included in an INNER JOIN. Prior to the release of Visual FoxPro 5.0, the syntax for the simplest example of a two-table "natural" or INNER JOIN was accomplished as shown in the following pseudo-code:

```sql
SELECT <field list> ;
FROM <table1>, <table2> ;
WHERE <table1.fieldName> = <table2.fieldName>
```

However, many times you want to produce a result set that not only includes the records from one (or both) of the tables that satisfy the join condition but also from records that do not satisfy the join condition. OUTER JOINs allow us to include records

```sql
SELECT <field list> ;
FROM <table1>, <table2> ;
WHERE <table1.fieldName> = <table2.fieldName>
```

...
that do not meet the join condition.

For example, if you look at the sample data that ships with Visual FoxPro (the TESTDATA database located in the \VFP\SAMPLES\DATA subdirectory), there is an EMPLOYEE table that can be joined to the ORDERS table (using the Emp_ID field), allowing a query to report order information related to the employee who wrote the order. If a new employee hadn't yet written an order, such a query wouldn't include that employee in the result set. It may be desirable, simply for the sake of good order, to report on all employees and their activities, without having to make inferences about the absence of an employee’s name from such a report. This requirement makes a natural or “inner” join inadequate to report this information because unless the employee has written at least one order, there is no link between the employee and the ORDERS table.

**LEFT, RIGHT -- Is that my RIGHT or your RIGHT?**

When constructing an OUTER JOIN on two tables, it's not sufficient to simply call it an "OUTER JOIN," as the non-matching information could come from one table, the other, or both. Hence, there are three types of OUTER JOINs: a LEFT OUTER JOIN, a RIGHT OUTER JOIN, and a FULL JOIN.

In Venn diagrams like the ones shown in Figure 2a, Figure 2b, and Figure 2c, it's easy to visualize the records involved in a RIGHT, LEFT or FULL join.

*Figure 2a: A LEFT OUTER JOIN*

*Figure 2b: A RIGHT OUTER JOIN*
A LEFT OUTER JOIN as illustrated in Figure 2a would include all records from the CUSTOMER table (on the left) and only the matching records from the ORDERS table (on the right). Figure 2b illustrates a RIGHT OUTER JOIN whose result set will include all of the records from the PRODUCTS table (on the right) and only the matching records from the ORDITEMS table on the left. Figure 2c illustrates a FULL OUTER JOIN, in which all records from both the REPS table and the CUSTOMER table are returned in the result set, without regard to whether they meet the join condition.

The two tables diagrammed in Figure 2c (REPS and CUSTOMER) are included in the accompanying Download file. Their contents and structures are shown in Table 1a and Table 1b.

Table 1a. CUSTOMER table.
The situation represented by these tables is one in which Larry, Moe, and Curly are in-house sales reps. They aren't assigned to specific accounts; rather, they are responsible for servicing the "house" accounts. "House" accounts don't have permanently assigned sales representatives. Bullwinkle, Rocky, and Droopy, on the other hand, are the only representatives who service certain large accounts.

The following query illustrates a FULL join:

```sql
SELECT customer.cCompany, reps.cRepname
FROM reps
FULL OUTER JOIN customer
ON reps.cRep_ID = customer.cRep_ID
```

Table 1b. REPS table.

<table>
<thead>
<tr>
<th>CCOMPANY</th>
<th>CREP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>MegaCorp</td>
<td>000004</td>
</tr>
<tr>
<td>Momenpop Int'l</td>
<td>000000</td>
</tr>
<tr>
<td>MicroCorp</td>
<td>000000</td>
</tr>
<tr>
<td>Big Spaceships, Inc.</td>
<td>000005</td>
</tr>
<tr>
<td>Little Scooters, Ltd.</td>
<td>000000</td>
</tr>
<tr>
<td>General Products</td>
<td>000006</td>
</tr>
</tbody>
</table>

Table 2. Result set.

<table>
<thead>
<tr>
<th>CCOMPANY</th>
<th>CREPNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NULL.</td>
<td>Larry</td>
</tr>
<tr>
<td>.NULL.</td>
<td>Moe</td>
</tr>
<tr>
<td>.NULL.</td>
<td>Curly</td>
</tr>
<tr>
<td>MegaCorp</td>
<td>Bullwinkle</td>
</tr>
<tr>
<td>Big Spaceships, Inc.</td>
<td>Rocky</td>
</tr>
<tr>
<td>General Products</td>
<td>Droopy</td>
</tr>
<tr>
<td>Momenpop Int'l</td>
<td>.NULL.</td>
</tr>
<tr>
<td>MicroCorp</td>
<td>.NULL.</td>
</tr>
<tr>
<td>Little Scooters</td>
<td></td>
</tr>
</tbody>
</table>
The SQL SELECT statement requests the contents of the cCompany field from the CUSTOMER table. When there is no record in CUSTOMER that matches a record in the REPS table, the query returns a value of .NULL. for the requested field. Likewise, the three house accounts have no full time sales rep assigned; hence, the appearance of .NULL. for the cRepName field for those customer records. .NULL. values can be seen as “placeholders” that allow the display of records from the table on the OUTER side of a join.

While .NULL. may mean a lot to you, it won’t mean much to your clients and users. You can clean this up a bit by modifying your query to use Visual FoxPro’s NVL() function:

```
SELECT NVL(customer.cCompany, "House Accounts") AS Company, 
      NVL(reps.cRepname, "House") AS Representative 
FROM reps 
FULL OUTER JOIN customer 
ON reps.cRep_ID = customer.cRep_ID
```

The result set is shown in Table 3.

**Table 3. Result set.**

<table>
<thead>
<tr>
<th>Company</th>
<th>Representative</th>
</tr>
</thead>
<tbody>
<tr>
<td>House Accounts</td>
<td>Larry</td>
</tr>
<tr>
<td>House Accounts</td>
<td>Moe</td>
</tr>
<tr>
<td>House Accounts</td>
<td>Curly</td>
</tr>
<tr>
<td>MegaCorp</td>
<td>Bullwinkle</td>
</tr>
<tr>
<td>Big Spaceships, Inc.</td>
<td>Rocky</td>
</tr>
<tr>
<td>General Products</td>
<td>Droopy</td>
</tr>
<tr>
<td>Momenpop Int'l</td>
<td>House</td>
</tr>
<tr>
<td>MicroCorp</td>
<td>House</td>
</tr>
<tr>
<td>Little Scooters Ltd</td>
<td>House</td>
</tr>
</tbody>
</table>

The ease with which you can now perform these kinds of queries has important implications for your database designs. In this example, it’s convenient to have a Rep ID for house accounts (“000000”) and still have Rep IDs for individual inside sales representatives! Without easy access to OUTER JOINs, you’d be likely to create a "House Account" customer record, have the same Rep ID for each inside sales representative, have two tables for sales reps (one for inside, one for outside sales), overload the sales rep table by including an inside/direct field, by having two Rep ID fields -- one for inside, one for direct sales representatives -- or some combination of several of these techniques.

The New FROM JOIN syntax in Visual FoxPro 5.0

In versions of FoxPro prior to version 5.0, the join conditions were specified in the WHERE clause. This caused confusion for some, particularly those who were first learning FoxPro’s SQL implementation, as this is also where (no pun intended) the filter conditions were placed.

Although VFP 5.0 still supports specifying join conditions in the WHERE clause (backward compatibility, remember?), it’s a good idea to ensure that all newly written queries make use of the new extensions to the FROM clause, reserving the WHERE clause to its proper role of filtering the results. To take advantage of OUTER JOINs, you need to adopt the new syntax.

The new syntax is seen in the FROM clause. Up to now, the FROM clause has simply included a comma-delimited list of tables, and aliases if needed, following the FROM keyword. All these tables are opened, and then both joined and filtered in the WHERE clause. In versions of FoxPro through 3.0b, the minimum executable two-table SQL statement was as follows:

```sql
SELECT <field list> FROM table1,table2 
WHERE table1.field = table2.Field
```

If you need a Cartesian product (where every record in table1 is matched with every record in Table 2) or you aren’t concerned about running out of disk space, you could even eliminate the WHERE clause.

Here’s the equivalent minimalist SQL statement using the new syntax:
To produce a Cartesian product you'd eliminate the ON clause:

```sql
SELECT <field list> FROM table1 JOIN table2
```

To make matters worse, the Query Designer makes use of a rather counter-intuitive structure that, according to Microsoft, complies both with the ANSI SQL '92 standard and produces queries in the form that ODBC expects. My advice in the past has been to rely heavily on the Query Designer when learning to write SQL statements, but after getting the hang of it, the Query Designer can be abandoned in favor of the greater flexibility of "hand coded" queries. Assuming that some current problems with the Query Designer are resolved in time for the product's release, there may be some advantage to hand coding some queries as you learn how the new syntax behaves.

Here is a simplified (if less rigorous) example of the structure that you find in a three-table query generated by the Query Designer:

```sql
SELECT table1.FieldName, ;
    table2.FieldName, ;
    table3.FieldName, ;
FROM table3 ;
JOIN table2 ;
    JOIN table1 ;
ON table1.FieldName = table2.FieldName ;
    ON table3.FieldName = table2.FieldName
```

I'll refer to this as the ANSI '92 or "nested" structure (note how the ON clauses and the JOIN clauses match up in a "nested" pattern).

At this point, another alternative structure that works consistently and accurately is a FROM clause structure that I call "sequential." I'd like to mention here that Chin Bae went down this road before me, and discovered that this structure works, and (as you can see) is much more intuitive. Chin suggested the following rewrite of the syntax, which illustrates this alternative structure:

```sql
FROM [Database!]Table [LocalAlias]
    INNER | LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER] JOIN
    INNER | LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER] JOIN
    ...
[ON [Database!]Table [LocalAlias] .Column_Name =
    [Database!]Table [LocalAlias] .Column_Name]
```

As of this writing ( mid-July 1996), the documentation for the FROM clause in a SQL statement reads as follows:

```
FROM [DatabaseName!]Table [Local_Alias]
    INNER | LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER] JOIN
        , [DatabaseName!]Table [Local_Alias]
        INNER | LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER] JOIN
        ...
    [ON [DatabaseName!]Table [Local_Alias] .Column_Name =
        [DatabaseName!]Table [Local_Alias] .Column_Name]
```

Some of the join-type specifiers are optional, as shown in Table 4.

### Table 4. Equivalent JOIN keywords.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Is Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOIN</td>
<td>INNER JOIN</td>
</tr>
<tr>
<td>RIGHT JOIN</td>
<td>RIGHT OUTER JOIN</td>
</tr>
<tr>
<td>LEFT JOIN</td>
<td>LEFT OUTER JOIN</td>
</tr>
<tr>
<td>FULL JOIN</td>
<td>FULL OUTER JOIN</td>
</tr>
</tbody>
</table>

Perfectly clear, right?
When first working with this new syntax, I think that this structure is much more intuitive than the "nested" structure. I encourage using this "sequential" structure at first, then adopting the apparently "standard" ANSI '92 structure once you're comfortable with its behaviors.

Keep the following points in mind when you construct your FROM clause and join conditions:

- The JOIN keyword is the reference point for RIGHT OUTER and LEFT OUTER JOINS. A RIGHT OUTER JOIN will return all records from the table to the RIGHT of the JOIN keyword, and a LEFT OUTER JOIN will return all records from the table to the LEFT of the JOIN keyword. This is more evident in the ANSI '92 standard structure, in which JOIN separates table names, without intervening ON clauses.
- The JOIN keyword has the effect of opening the table that immediately follows it, just as the FROM keyword opens the table (or tables) immediately following. You can't reference a table in the ON clause until it has been opened. Thus, the following rewrite of the previous query will trigger an error, "SQL: Column <fieldName> is not found":

```
SELECT table1.FieldName, table2.FieldName, table3.FieldName,
FROM table1
JOIN table2
ON table2.FieldName = table1.FieldName
JOIN table3
ON table3.FieldName = table2.FieldName
```

This error occurs because the SQL command is evaluated from left to right, and the first ON clause refers to table2, which isn't opened until the second JOIN is evaluated.
- Each JOINON clause creates an intermediate result, which is then used by any other JOINON clauses that the query may include.

How does the new syntax work?
The most important thing to remember in understanding how to apply the new syntax is the last point I made in the preceding section: each JOINON clause creates an intermediate result.

To understand this, consider the pseudocode example I used earlier. Here I've rewritten it, but only to change the indenting and the location of the line-continuation characters (;):

ANSI '92/Nested structure
```
SELECT table1.FieldName, table2.FieldName, table3.FieldName,
FROM ;
table3 JOIN ;
table1 JOIN table2 ;
ON table1.FieldName = table2.FieldName
ON table3.FieldName = table2.FieldName
```

"Sequential" structure
```
SELECT table1.FieldName, table2.FieldName, table3.FieldName,
FROM
table1 JOIN table2 ;
ON table2.FieldName = table1.FieldName ;
JOIN table3 ;
```

The way this translates into actual use, using this "pseudocode" example is much clearer:
At first glance, a subtle point made in these two examples might pass unnoticed. Because each JOINON clause will create an intermediate result, and the command is evaluated from left-to-right, the order in which the JOINON clauses appear is critical to getting expected (and correct) results! If you examine the previous "sequential" structured query, as the command is evaluated, you'll see that the first pair of tables linked by a JOINON clause is table1 and table2. This JOIN produces an intermediate result, which is then JOINed with table3 in the next JOINON clause, using the condition table3.FieldName = table2.FieldName.

Compare this to the ANSI '92/Nested structure. As that command is evaluated from left-to-right, the first JOIN <table> clause is followed by another JOIN, rather than an ON clause. Therefore the intermediate results of the second or innermost join must be evaluated first. The first join evaluated is as follows, just as it is in the sequential structure:

```
table1 JOIN table2 ON table2.fieldName = table1.fieldName
```

The effect is for Visual FoxPro to evaluate the FROM expression from the "inside out," producing an intermediate result from the innermost JOINON clause, then applying the next outermost JOIN to that intermediate result set. It's possible to get the same result set from either structure, but the order in which the tables are named with their associated JOIN clauses and awareness of the intermediate result sets is critical to getting the result sets you're looking for.

This is a surprise to those of us familiar with SQL syntax in versions of FoxPro through 3.0b, where the order of the tables named in the FROM clause and the order of join conditions in the WHERE clause had no effect on the result set.

Because this is such a new way to approach SQL commands, I'll include two examples for each query (where applicable) in the rest of this article, including the more intuitive "sequential" structure and the less intuitive, but more orthodox structure that adheres to the ANSI SQL '92 standard. I'm finding it to be both a challenge and a way to sharpen my skills in using the new SQL syntax to write all queries using both structures.

The formatting you will see me use, particularly for the ANSI '92 structure, is designed to help conceptualize exactly what is happening:

```
FROM ;
<table1> JOIN ;
<table2> JOIN <table3> ;
ON
```

What the foregoing is trying to show is that the innermost expression (which is evaluated first) JOINs Table 2 and Table 3, and then Table 1 is joined to this intermediate set.

One last, but very important point to make regarding the new FROMJOINON syntax is that the two structures I describe here are not mutually exclusive. It's not a matter of "either/or," as the two structures can be combined in the same query, as shown in this bit of pseudocode:

```
SELECT <stuff> ;
FROM <table1> JOIN <table2> ;
ON <condition>
JOIN <table3> ;
JOIN <table4> ;
ON <condition> ;
ON <condition3>
```

Here the join between Table 1 and Table 2 is made first, and the intermediate result is joined with the result of joining Table 3 and Table 4.

Use of parentheses (which are allowed in Visual FoxPro 5.0 queries, simply as a self-documenting notation) or a slightly different way of formatting the query can also help to understand how the query is executed:

```
SELECT <stuff>
FROM ;
(<table1> JOIN <table2> ON <condition>) ;
JOIN ;
(<table3> JOIN <table4> ON <condition>) ;
ON <condition>
```

My thanks to Matt Peirse on the Beta forum for getting me to see this!

**Application of OUTER JOINS**

Let's take a look at a fairly straightforward query that provides information without making use of OUTER JOINs. Drawing on the TESTDATA database that ships with Visual FoxPro, the following query generates a result set that shows each product in the PRODUCTS table and the total units sold for each product:

```
SELECT SUM(orditems.quantity) AS purchased, ;
products.product_id, ;
```
When you run this query, you'll see that there have been purchases for every product. As a result, you can modify the query as follows to do a RIGHT OUTER JOIN between the ORDITEMS table and the PRODUCTS table:

```
SELECT SUM(orditems.quantity) AS purchased, 
 products.product_id, 
 products.eng_name ;
FROM testdata!orditems ;
JOIN testdata!products ;
ON orditems.product_id = products.product_id ;
GROUP BY products.product_id ;
ORDER BY products.product_id ;
```

The results are identical! This is because in the join between the ORDITEMS table and PRODUCTS table, for every record in each table, there exists a matching record in the table to which it is joined. In set terminology, the intersection is equal to the union. Therefore, an OUTER JOIN and an INNER JOIN will both yield the same result set. (See Figure 3.)

**Figure 3: Intersection is equal to the union.**

![Intersection is equal to the union](image)

To properly see the effect of this OUTER JOIN, let's assume for the moment that one of the products is a new product offering and that your query will show how sales are running for the entire product line, including products that may show no sales to date. To simulate this situation, open the ORDITEMS table and issue the following commands in the Command window:

```
DELETE FOR Product_ID = "    60"
SET DELETED ON
```

Now when you run the query, the "Purchased" column will show .NULL for Pierrot Camembert (the English name for product_id 60).

To look at something a bit more complex, you can create a query that reports product sales by customer:

```
SELECT customer.company, 
 SUM(orditems.quantity) AS purchased ;
products.product_id, 
products.eng_name ;
FROM testdata!customer ;
JOIN testdata!orders ;
ON customer.cust_id = orders.cust_id ;
GROUP BY products.product_id ;
ORDER BY products.product_id ;
```

http://foxtalknewsletter.com/ME2/Audiences/Segments/Publications/Print.asp?Module=...  14.02.06
Here is the same query, but conforming to the ANSI '92 standard:

```
SELECT customer.company, ;
  SUM(orditems.quantity) AS purchased ;
products.product_id, ;
products.eng_name ;
FROM ;
testdata!products JOIN ;
testdata!orditems JOIN testdata!customer ;
  ON customer.cust_id = orders.cust_id ;
  ON orditems.order_id = orders.order_id ;
  ON orditems.product_id = products.product_id ;
GROUP BY customer.cust_id, products.product_id ;
ORDER BY customer.company, products.product_id
```

If you examine the product_id numbers for an individual customer, the list is discontinuous -- that is, some of the 77 products are missing. Obviously, this is because not all products are purchased by all customers. Rather than inferring which products are not being purchased by a given customer (after all, not all product lines have a conveniently continuous sequence of product IDs!), it would be helpful to explicitly list all products and show their sales as .NULL. (or 0) if no sales of that product have been made to a particular customer.

You could try to introduce the OUTER JOIN, requesting all records from the PRODUCTS table:

```
SELECT customer.company, ;
  SUM(orditems.quantity) AS purchased ;
products.product_id, ;
products.eng_name ;
FROM ;
testdata!customer ;
  INNER JOIN testdata!orders ;
  ON customer.cust_id = orders.cust_id ;
  INNER JOIN testdata!orditems ;
  ON orders.order_id = orditems.order_id ;
  RIGHT OUTER JOIN testdata!products ;
  ON orditems.product_id = products.product_id ;
GROUP BY customer.cust_id, products.product_id ;
ORDER BY customer.cust_id, products.product_id
```

Here's the equivalent query using the ANSI '92 compliant syntax. This rewrite requires that you use a LEFT OUTER JOIN because the PRODUCTS table is now to the left of the JOIN keyword:

```
SELECT customer.company, ;
  SUM(orditems.quantity) AS purchased ;
products.product_id, ;
products.eng_name ;
FROM ;
testdata!products LEFT OUTER JOIN ;
testdata!orditems JOIN ;
  testdata!orders JOIN testdata!customer ;
  ON customer.cust_id = orders.cust_id ;
  ON orditems.order_id = orders.order_id ;
  ON orditems.product_id = products.product_id ;
GROUP BY customer.cust_id, products.product_id ;
ORDER BY customer.cust_id, products.product_id
```

Running this query yields almost the same result set as the prior example, which did *not* make use of the OUTER JOIN! A clue to the reason that you're getting this result is that a record for Product_ID 60, Pierrot Camembert, now appears as the first record (see Table 5).

Your OUTER JOIN joins the ORDITEMS table and the PRODUCTS table. Because all PRODUCTS records have at least one match in the ORDITEMS table, again, the OUTER JOIN is equivalent to the INNER JOIN except for Product_ID 60. Thus, the only additional record that the OUTER JOIN placed in the result set is the only product that hasn't been purchased by any customer.

**Table 5.** An OUTER JOIN adds a record to the result set.

http://foxtalknewsletter.com/ME2/Audiences/Segments/Publications/Print.asp?Module=... 14.02.06
To get the results you're seeking -- that is, a result set that shows the entire product line along with a particular customer's total purchases of each product -- you can rewrite the query as follows:

```
SELECT customer.company, 
    SUM(orditems.quantity) AS purchased, 
    products.product_id, 
    products.eng_name 
FROM testdata!customer 
    INNER JOIN testdata!orders 
    ON customer.cust_id = orders.cust_id 
    INNER JOIN testdata!orditems 
    ON orders.order_id = orditems.order_id 
RIGHT OUTER JOIN testdata!products 
    ON orditems.product_id = products.product_id 
WHERE customer.cust_id = "BOTTM" 
GROUP BY products.product_id 
ORDER BY products.product_id
```

Or using the ANSI SQL '92 structure, rewrite it as follows:

```
SELECT customer.company, 
    SUM(orditems.quantity) AS purchased, 
    products.product_id, 
    products.eng_name 
FROM testdata!products LEFT OUTER JOIN 
    testdata!orditems JOIN 
    testdata!orders JOIN testdata!customer 
    ON customer.cust_id = orders.cust_id 
    ON orditems.order_id = orders.order_id 
    ON orditems.product_id = products.product_id 
WHERE customer.cust_id = "BOTTM" 
GROUP BY products.product_id 
ORDER BY products.product_id
```

This query makes two changes to the previous query. First, you filter the results to examine an individual customer (in the WHERE clause) and, second, you no longer group or order by the customer, since you're examining one customer at a time. Because you're filtering the result set to include only those ORDITEMS records that apply to a particular customer (Bottom-Dollar Markets), you get the results you're looking for, a portion of which is shown in Table 6.

Table 6. A result set using an OUTER JOIN and WHERE filter.

<table>
<thead>
<tr>
<th>Company</th>
<th>Product_id</th>
<th>Eng_name</th>
<th>Purchased</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom-Dollar Markets</td>
<td>1</td>
<td>Dharamsala Tea</td>
<td>60.000</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>2</td>
<td>Tibetan Barley Beer</td>
<td>30.000</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>3</td>
<td>Licorice Syrup</td>
<td>20.000</td>
</tr>
<tr>
<td>.NULL.</td>
<td>4</td>
<td>Chef Anton’s Cajun Seasoning</td>
<td>.NULL.</td>
</tr>
<tr>
<td>.NULL.</td>
<td>5</td>
<td>Chef Anton’s Gumbo Mix</td>
<td>.NULL.</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>6</td>
<td>Grandma’s Boysenberry Spread</td>
<td>12.000</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>7</td>
<td>Uncle Bob’s Organic Dried Pears</td>
<td>20.000</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>8</td>
<td>Northwoods Cranberry Sauce</td>
<td>16.000</td>
</tr>
<tr>
<td>.NULL.</td>
<td>9</td>
<td>Mishi Kobe Beef</td>
<td>.NULL.</td>
</tr>
</tbody>
</table>

To "clean up" the result set, you can again make use of the NVL() function to substitute meaningful values for the .NULL. values that the OUTER JOIN produces. A sample of the result set can be seen in Table 7.

Table 7. Result set with .NULL. values replaced.
<table>
<thead>
<tr>
<th>Company</th>
<th>Product_id</th>
<th>Eng_name</th>
<th>Purchased</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom-Dollar Markets</td>
<td>1</td>
<td>Dharamsala Tea</td>
<td>60</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>2</td>
<td>Tibetan Barley Beer</td>
<td>30</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>3</td>
<td>Licorice Syrup</td>
<td>20</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>4</td>
<td>Chef Anton's Cajun Seasoning</td>
<td>0</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>5</td>
<td>Chef Anton's Gumbo Mix</td>
<td>0</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>6</td>
<td>Grandma's Boysenberry Spread</td>
<td>12</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>7</td>
<td>Uncle Bob's Organic Dried Pears</td>
<td>20</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>8</td>
<td>Northwoods Cranberry Sauce</td>
<td>16</td>
</tr>
<tr>
<td>Bottom-Dollar Markets</td>
<td>9</td>
<td>Mishi Kobe Beef</td>
<td>0</td>
</tr>
</tbody>
</table>

SELECT NVL(customer.company,"Bottom-Dollar Markets") AS company, 
products.product_id, 
products.eng_name, 
NVL(SUM(orditems.quantity),0000) AS purchased;

My friend Anders Altberg demonstrated how to significantly improve the performance of even this simple query by removing the filter (WHERE) condition, including this condition as part of the JOIN condition on the CUSTOMER table instead:

SELECT customer.company, 
SUM(orditems.quantity) AS purchased, 
products.product_id, 
products.eng_name;
FROM 
testdata!products LEFT OUTER JOIN 
testdata!orditems JOIN 
testdata!orders JOIN testdata!customer;
ON customer.cust_id = orders.cust_id AND;
customer.cust_id = "BOTTM";
ON orditems.order_id = orders.order_id;
ON orditems.product_id = products.product_id;
group by products.product_id ;
order by products.product_id;

In my testing on a 486/66 with 16M, running Visual FoxPro 5.0 on Windows NT 3.51, the query ran in the neighborhood of 2.7 seconds when placing the filter condition in the WHERE clause, but just under a second when the filter condition was included as a JOIN condition instead! Anders explained that this is because the filter condition gets evaluated only for that intermediate set, not for all records in all intermediate sets.

Before moving on, I'll share with you the query sent to me by Matt Peirse, which successfully generates a result set that succeeds where one of my earlier queries failed. It shows all customers, their total purchases of each product, and .NULL.s for the products that a particular customer hasn't purchased:

```
SELECT company, 
       SUM(quantity) as purchased, 
       products.product_id, 
       products.eng_name 
FROM 
  customer JOIN products on .t. 
  LEFT JOIN orders JOIN orditems 
  ON orders.order_id = orditems.order_id) 
  ON customer.cust_id=orders.cust_id 
  AND products.product_id=orditems.product_id 
GROUP BY company,products.product_id ;

GROUP BY products.product_id 
ORDER BY products.product_id 

ORDER BY company,products.product_id ;

GROUP BY company,products.product_id

GROUP BY products.product_id ;

ORDER BY products.product_id 
```

This query combines (as I described earlier) the "sequential" and "nested" syntax, and makes an interesting use of a Cartesian product in the first JOIN between CUSTOMER and PRODUCTS. If you examine this query carefully, and can explain how it works, you'll be in good shape in applying the new syntax in your own work.

The last example using OUTER JOINs illustrates the use of local aliases with the new JOIN syntax. Local aliases are used whenever a single table must be opened more than once to allow JOINs to be established to use two different join conditions.

The data for this example comes from our company's line of business. Our product line includes almost 1,000 part numbers. It's often necessary to interchange one of our part numbers to a competitor's part number, or vice versa. To permit this, our system includes an interchange table with the structure shown in Table 8a. A sample of the INTERCHG.DBF data in the accompanying Download file appears in Table 8b.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CCOMP_ID</td>
<td>Character 6</td>
<td>The competitor's ID - Foreign Key</td>
</tr>
<tr>
<td>2</td>
<td>CPART</td>
<td>Character 6</td>
<td>Our part number - Foreign Key</td>
</tr>
<tr>
<td>3</td>
<td>CCOMPPART</td>
<td>Character 10</td>
<td>The competitor's part number - Foreign Key</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ccomp_id</th>
<th>Cpart</th>
<th>Ccomppart</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>6363</td>
<td>18-4611</td>
</tr>
<tr>
<td>000000</td>
<td>6365</td>
<td>18-4612</td>
</tr>
<tr>
<td>000000</td>
<td>6364</td>
<td>18-4613</td>
</tr>
<tr>
<td>00000A</td>
<td>6300</td>
<td>4243</td>
</tr>
<tr>
<td>00000A</td>
<td>6301</td>
<td>4244</td>
</tr>
<tr>
<td>00000A</td>
<td>6302</td>
<td>4291</td>
</tr>
</tbody>
</table>

Table 8a. Structure for table INTERCHG.DBF.

Table 8b. Sample of data in INTERCHG.DBF.
The cComp_ID field allows the table to be related into a file with competitor information. Table 9a shows the structure, and Table 9b the contents of this table.

**Table 9a. Structure for table COMPETIT.DBF.**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CCOMP_ID</td>
<td>Character</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>CCOMPET</td>
<td>Character</td>
<td>15</td>
</tr>
</tbody>
</table>

**Table 9b. Data contained in COMPETIT.DBF.**

<table>
<thead>
<tr>
<th>Ccomp_id</th>
<th>Ccompet</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>ABC Company</td>
</tr>
<tr>
<td>00000A</td>
<td>PQR Company</td>
</tr>
<tr>
<td>00000F</td>
<td>XYZ Company</td>
</tr>
</tbody>
</table>

The cPart field in the INTERCHG table allows the table to be joined into our product table, OURLINE, with the structure shown in Table 10a, and a sample of its contents is illustrated in Table 10b.

**Table 10a. Structure for table OURLINE.DBF.**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Width</th>
<th>Dec</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPART</td>
<td>Character</td>
<td>6</td>
<td>Our part number - Primary Key</td>
</tr>
<tr>
<td>2</td>
<td>CAPPLIC</td>
<td>Character</td>
<td>60</td>
<td>Vehicle application</td>
</tr>
<tr>
<td>3</td>
<td>ION_HAND</td>
<td>Integer</td>
<td>4</td>
<td>Qty. on hand</td>
</tr>
<tr>
<td>4</td>
<td>YJOBBER</td>
<td>Currency</td>
<td>8</td>
<td>4 Jobber price</td>
</tr>
<tr>
<td>5</td>
<td>YCORE</td>
<td>Currency</td>
<td>8</td>
<td>4 Core value</td>
</tr>
<tr>
<td>6</td>
<td>CPOPCODE</td>
<td>Character</td>
<td>3</td>
<td>Popularity code</td>
</tr>
</tbody>
</table>

**Table 10b. Sample of data in OURLINE.DBF.**

<table>
<thead>
<tr>
<th>Cpart</th>
<th>Capplic</th>
</tr>
</thead>
<tbody>
<tr>
<td>6300</td>
<td>Mustang, T-Bird Turbo Coupe 1987-89</td>
</tr>
<tr>
<td>6301</td>
<td>Mustang, T-Bird Turbo Coupe 1987-89</td>
</tr>
<tr>
<td>6302</td>
<td>Jeep Cherokee, Comanche, Wagoneer, Wrangler</td>
</tr>
<tr>
<td>6303</td>
<td>Jeep Cherokee, Comanche, Wagoneer, Wrangler</td>
</tr>
</tbody>
</table>
Printed reports of interchange information have, in the past, presented some difficulties. First, in order to provide a report that shows which of our part numbers have not yet been interchanged to the competitor's number, it was necessary to jump through the hoops I mentioned earlier to simulate an OUTER JOIN. This was complicated further by the frequent need to provide interchange information for more than one competitor in the report. Now, with FoxPro support of OUTER JOINs, this task can be accomplished with a single, simple query.

In the following SELECT statement, note that because we want to report interchange information for two competitors, the INTERCHG table is opened twice, once with an alias of "ABC" and again with an alias of "XYZ." The LEFT OUTER JOINs ensure that we retrieve all records from the OURLINE table, without regard to whether or not we've interchanged that part for a particular competitor, and the query is filtered on the competitor ID number for only the two competitors that are of immediate interest. The SELECT command also makes use of the NVL function to present a string of dashes instead of a .NULL. value:

```
SELECT ourline.cpart AS Our_No, 
    NVL(abc.ccomppart, "-----") AS ABC_No, 
    NVL(xyz.ccomppart, "-----") AS XYZ_No 
FROM ourline 
LEFT OUTER JOIN interchg ABC 
    ON ourline.cpart = ABC.cpart 
LEFT OUTER JOIN interchg XYZ 
    ON ourline.cpart = XYZ.cpart 
WHERE ABC.ccomp_id = "000000" 
    AND XYZ.ccomp_id = "00000F" 
ORDER BY ourline.cpart
```

And again, using the ANSI SQL '92 structure:

```
SELECT ourline.cpart AS Our_No, 
    NVL(abc.ccomppart, "-----") AS ABC_No, 
    NVL(xyz.ccomppart, "-----") AS XYZ_No 
FROM 
    interchg XYZ RIGHT OUTER JOIN 
    ourline LEFT OUTER JOIN interchg ABC 
    ON ourline.cpart = ABC.cpart
```
And finally, using the trick of including WHERE conditions as JOIN conditions instead:

```
SELECT ourline.cpart AS Our_No, 
    NVL(abc.ccomppart,"-----") AS ABC_No, 
    NVL(xyz.ccomppart,"-----") AS XYZ_No 
FROM 
    interchg XYZ RIGHT OUTER JOIN 
    ourline LEFT OUTER JOIN interchg ABC 
    ON ourline.cpart = ABC.cpart 
    AND ABC.ccomp_id = "000000" 
    ON ourline.cpart = XYZ.cpart 
    AND XYZ.ccomp_ID = "00000F" 
ORDER BY ourline.cpart
```

The result set is shown in Table 11.

**Table 11. Result set.**

<table>
<thead>
<tr>
<th>Our_no</th>
<th>Abc_no</th>
<th>Xyz_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>6300</td>
<td>18-4257</td>
<td>4079</td>
</tr>
<tr>
<td>6300</td>
<td>18-4257</td>
<td>4099</td>
</tr>
<tr>
<td>6301</td>
<td>18-4258</td>
<td>4078</td>
</tr>
<tr>
<td>6301</td>
<td>18-4258</td>
<td>4098</td>
</tr>
<tr>
<td>6302</td>
<td>18-4339</td>
<td>1013</td>
</tr>
<tr>
<td>6303</td>
<td>18-4340</td>
<td>1012</td>
</tr>
<tr>
<td>6304</td>
<td>-----</td>
<td>2143</td>
</tr>
<tr>
<td>6305</td>
<td>-----</td>
<td>2142</td>
</tr>
<tr>
<td>6306</td>
<td>18-4275</td>
<td>2135</td>
</tr>
<tr>
<td>6307</td>
<td>18-4276</td>
<td>2134</td>
</tr>
<tr>
<td>6308</td>
<td>18-4311</td>
<td>4095</td>
</tr>
<tr>
<td>6309</td>
<td>18-4312</td>
<td>4094</td>
</tr>
<tr>
<td>6310</td>
<td>18-4293</td>
<td>3055</td>
</tr>
<tr>
<td>6311</td>
<td>18-4294</td>
<td>3056</td>
</tr>
<tr>
<td>18-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As a developer who once found it useful to memorize the month, year, and page number of Jerry Ela's *FoxTalk* article on how to simulate OUTER JOINs, I can attest to their utility. Microsoft has placed the ability to perform this important function at our fingertips with the release of Visual FoxPro 5.0, and it's well worth your time to learn to apply it in your applications.