# Using SQL-SELECT Effectively


## Tamar E. Granor, Ph.D.


## Overview

Visual FoxPro 5 added ANSI-compliant JOINs, including outer joins, to SQL SELECT. However, the syntax for using them isn't intuitive and it's possible to write a query that runs, but returns the wrong results. In addition, the Query Designer and View Designer are unable to generate certain kinds of queries correctly, so it's essential to be able to write them by hand. The structure of a query and the underlying data also affect performance.

This session explains the join syntax and shows how to write queries that return the right results. It also explores query performance and optimization techniques, including the ability to find out how VFP is applying Rushmore.

This session is aimed at users of VFP 5 and VFP 6, and covers native FoxPro commands, not client-server uses of SELECT.

## Join fundamentals

Whenever a query includes more than one table, it's necessary to indicate how records from the tables are to be matched up to produce the result set. This is called *joining* the tables and the specification is called a *join condition*.

Prior to VFP 5, join conditions had to be specified in the WHERE clause of the query along with filter conditions. VFP 5 added ANSI-compliant JOIN syntax to SELECT-SQL. JOIN is part of the FROM clause of SELECT, where the list of tables for the query is specified. The full syntax for specifying the FROM clause is quite complex, so we'll take it a little at a time.

To join two tables, you use syntax like:

```
<Table1> JOIN <Table2> ON <expression>
```

For each table, you can specify both the table name and a local alias (an alias to use for that table in the query). The expression indicates which field or fields of the tables should be matched. For example, we can join the Customer and Orders tables of the Tastrade sample data like this:

```
FROM Customer ;
  JOIN Orders ;
  ON Customer.Customer_Id = Orders.Customer_Id
```

The expression following the ON clause can be more complex than a simple comparison. For example, if the criteria for joining two tables involve multiple fields, you can use AND and OR to combine multiple comparisons.

## Multi-table joins

As soon as more than two tables are involved in a join, things get a lot more complex. When the tables are in a parent-child-grandchild relationship, though, it's still pretty simple. To add the order details to the join above, the FROM clause looks like:

```
FROM Customer ;
  JOIN Orders ;
    JOIN Order_Line_Items ;
    ON Orders.Order_Id = Order_Line_Items.Order_Id ;
  ON Customer.Customer_Id = Orders.Customer_Id
```

The second join is nested inside the first. Note that the first ON clause listed belongs to the last JOIN. From a logical perspective, VFP performs the innermost join first, then works its way outward. At runtime, the joins may be performed in that order or a different order, if the optimizer decides that another approach is faster.

Multi-table joins can use either the nested syntax above or a sequential syntax that completes the join of one pair of tables before adding a third. Using the sequential syntax, the join above can be written:

```
FROM Customer ;
  JOIN Orders ;
    ON Customer.Customer_Id = Orders.Customer_Id ;
  JOIN Order_Line_Items ;
    ON Orders.Order_Id = Order_Line_Items.Order_Id
```

For tables with a parent-child-grandchild relationship, the nested syntax makes sense, but some queries involve tables that don't have a parent-child-grandchild relationship. For example, sometimes you need to work with two children of the same parent. While the nested syntax can usually be used in such cases, getting it right is difficult and the resulting query can be hard to read. For example, using the Tastrade sample data, consider a query in which you want to find out the employee who took an order and the shipper used to send the order. You could use the nested syntax to write it, as follows:

```
SELECT shippers.company_name, ;
       orders.order_id, ;
       employee.last_name ;
   FROM shippers ;
     JOIN orders ;
       JOIN employee ;
       ON orders.employee_id = employee.employee_id ;
     ON shippers.shipper_id = orders.shipper_id
```

This query gives the right results, but implies a parent-child-grandchild relationship from shippers to orders to employees. Here's a sequential version of the query that's far more readable:

```
SELECT shippers.company_name, ;
       orders.order_id, ;
       employee.last_name ;
   FROM orders;
     JOIN shippers;
       ON shippers.shipper_id = orders.shipper_id;
     JOIN employee ;
       ON orders.employee_id = employee.employee_id
```

More importantly, note that attempting to write the nested version in a more intuitive manner produces a query that either fails to run or produces bad results. Here's a query that's only slightly different from the nested version above:

```
SELECT shippers.company_name, ;
       orders.order_id, ;
       employee.last_name ;
   FROM orders ;
     JOIN shippers ;
       JOIN employee ;
       ON orders.employee_id = employee.employee_id ;
     ON shippers.shipper_id = orders.shipper_id
```

When you run this query, one of two things happens. If the Orders tables is open, the query runs, but the results contain the same employee in each row rather than the one associated with the order. If the Orders table is closed, the error "SQL: Column 'EMPLOYEE_ID' is not found." appears. The problem is that VFP attempts to join Shippers and Employee using the condition orders.employee_id=employee.employee_id.

When a query involves more than three tables, it can be very difficult to use the nested format. Suppose we want to add the Customer table to the query above. We now have three tables in a parent-child relationship with Orders (Shippers, Employee, and Customer). How can you express this in a nested way?

The Query Designer is unable to produce a query that works, regardless of the order in which tables are added and even if you help it out along the way. You can write a nested query by hand, but it doesn't make a whole lot of sense to the reader:

```
SELECT Customer.company_name, ;
       Orders.order_id, ;
       Employee.last_name,;
       Shippers.company_name;
 FROM tastrade!customer ;
   JOIN tastrade!employee;
     JOIN tastrade!orders ;
       JOIN tastrade!shippers ;
       ON Shippers.shipper_id = Orders.shipper_id ;
     ON Employee.employee_id = Orders.employee_id ;
   ON Customer.customer_id = Orders.customer_id
```

Who'd think of listing the unrelated tables Customer and Employee first? The sequential version of the query is far more readable.

```
SELECT Customer.company_name, ;
       Orders.order_id, ;
       Employee.last_name,;
       Shippers.company_name;
 FROM tastrade!orders ;
   JOIN tastrade!customer ;
     ON Customer.customer_id = Orders.customer_id ;
   JOIN tastrade!employee  ;
     ON Employee.employee_id = Orders.employee_id ;
   JOIN tastrade!shippers ;
     ON Shippers.shipper_id = Orders.shipper_id
```

This version makes the relationships (or lack of them) among the tables clear.

You can mix nested and sequential joins in a single query. Here's one that lists the customer name, shipper, order date, and each product in an order:

```
SELECT customer.company_name, ;
       shippers.company_name, ;
       orders.order_date, ;
       products.english_name ;
  FROM customer ;
    JOIN orders ;
      JOIN order_line_items ;
        JOIN products ;
        ON order_line_items.product_id = ;
           products.product_id ;
      ON orders.order_id = order_line_items.order_id ;
    ON customer.customer_id = orders.customer_id ;
    JOIN shippers ;
      ON orders.shipper_id = shippers.shipper_id
```

Nested joins are used to move from Customer down to Products and then the sequential syntax indicates to join that result with Shippers.

## Adding outer joins to the mix

All the queries above use *inner joins*. With inner joins, only those records that match in the joined tables appear in the result. Any record in one table that has no counterpart in the table to which it's being joined is omitted. FoxPro has included inner joins since SQL SELECT was first added to the language and all of the queries above can be written with join conditions in the WHERE clause rather than using the new JOIN syntax.

However, many useful queries include the unmatched records in the result instead of omitting them. Such queries are called *outer joins*. There are three types of outer joins - they differ in which tables contribute unmatched records to the result. A *left outer join* includes unmatched records from the first (left) table listed in the join. Correspondingly, a *right outer join* includes unmatched records from the second (right) table listed. A *full outer join* includes unmatched records from both tables. Any fields in the result which come from unmatched records are filled with null values (.NULL.).

Each join in the FROM clause is considered separately for the purposes of distinguishing inner and outer joins and determining whether an outer join is left, right or full. So a single query may contain a left outer join, two inner joins and a full outer join, for instance. Also, note that the words "inner" and "outer" are usually omitted when talking about joins, so you'll hear people refer to joins (that is, inner joins), left joins, right joins and full joins. Similarly, the INNER and OUTER keywords of SELECT are optional.

To look at outer joins, we'll use a different set of sample data. This database is for a Couples club. The tables we're initially interested in are Couple, Member and Phones. There's a single record in Couple for each couple in the group, with a primary key CoupleId. Similarly, each individual member has a record in the Member table with primary key MemberId. The Couple table contains two fields that point to the Member table, HisId and HerId, to link to the man and woman who compose that Couple. The Phones table is linked to the Couple table by CoupleId; each record also indicates whether it's a home, work or fax number and, if appropriate, to which Member of the Couple it applies. Here's a simplified version of the three table structures:

**Couple.DBF**

```
Field Name        Type          Width       Dec
COUPLEID          Character         4
HISID             Character         4
HERID             Character         4
ADDRESS           Memo              4
CITY              Character        25
STATE             Character         2
ZIP               Character         9
ANNIVERSARY       Date              8
MEMBERSINCE       Date              8

Member.DBF

Field Name        Type          Width       Dec
MEMBERID          Character         4
FIRSTNAME         Character        15
LASTNAME          Character        25
BIRTHDATE         Date              8

Phones.DBF

Field Name        Type          Width       Dec
COUPLEID          Character         4
PHONETYPE         Character         1
WHOSEPHONE        Character         4
AREACODE          Character         3
PHONE             Character         7
```

What makes this database interesting is the double relationship between Couple and Member (called a *self-join).* Also important is that there may or may not be any phone records for a Couple. Also, in order to track some people on the club's mailing list as a courtesy (for example, staff members of the sponsoring organization), a Couple record is not required to have both HisId and HerId filled in; one of them must contain a valid id from Member, but the other may be empty.

As with inner joins, a two-table outer join is pretty simple to assemble. Here's a query that shows each couple's id and whatever phone numbers are on record. Every couple is included, even if no numbers are recorded.

```
SELECT Couple.CoupleId, PhoneType, Phone ;
  FROM Couple ;
    LEFT JOIN Phones ;
    ON Couple.CoupleId = Phones.CoupleId
```

This is a left join, so all Couple records are included, but some Phones records could be omitted. (Of course, in this case, if any Phones are omitted, it's a sign of trouble since every Phones record should be associated with a Couple record.) The PhoneType and Phone fields are drawn from Phones; they're filled with .NULL. for any couples with no phones listed.

Suppose we want to get a list of members of the club. If every couple were required to have both a husband and a wife, this would be done quite simply using either the nested or the sequential syntax. I prefer the sequential, like this:

```
SELECT Her.FirstName AS HerFirst, ;
       Her.LastName AS HerLast, ;
       His.FirstName AS HisFirst, ;
       His.LastName AS HisLast ;
  FROM Couple ;
```

```
    JOIN Member Her ;
      ON Couple.HerId = Her.MemberId ;
    JOIN Member His ;
      ON Couple.HisId = His.MemberId
```

The query uses *local aliases* Her and His to differentiate between the two uses of the Member table. The one joined to HerId is Her and the one joined to HisId is His.

But Couple may have either HerId or HisId empty. To include all couples, we need to use outer joins between Couple and Member. Both the nested and sequential syntax work:

```
SELECT Her.FirstName AS HerFirst, ;
       Her.LastName AS HerLast, ;
       His.FirstName AS HisFirst, ;
       His.LastName AS HisLast ;
  FROM Member Her;
    RIGHT JOIN Couple ;
      LEFT JOIN Member His ;
      ON Couple.HisId = His.MemberId ;
    ON Couple.HerId = Her.MemberId

SELECT Her.FirstName AS HerFirst, ;
       Her.LastName AS HerLast, ;
       His.FirstName AS HisFirst, ;
       His.LastName AS HisLast ;
  FROM Couple ;
    LEFT JOIN Member Her ;
      ON Couple.HerId = Her.MemberId ;
    LEFT JOIN Member His ;
      ON Couple.HisId = His.MemberId
```

Note that the direction of the join between Couple and Her is different in the nested query than in the sequential. That's because the order of the tables is different. LEFT and RIGHT are applied to the list of tables itself, not to the order in which tables are mentioned in the join expression following ON.

As queries involving outer joins become more complex, finding a structure that works can be very difficult, whether the nested or the sequential syntax is used. One problem is figuring out which kind of outer join to use.

To demonstrate the problem, let's add two more tables to our couples club database. The club has a board of directors with one couple serving in each position. We'll store the list of board positions in Board with a primary key PosnId and a description, Position. (The third field, DisplayOrder, is used for producing board lists, so that positions like President and Vice President sort to the top, while others can be listed alphabetically.) To link the positions to individuals, we use another table, OnBoard, which has a PosnId and a CoupleId (and a Year field to keep a historical record). Some board positions may not be filled. Here's the structure for these two tables:

```
Board.DBF

Field Name         Type          Width       Dec
POSNID             Character         2
POSITION           Character        15
DISPLAYORDER       Numeric           2

OnBoard.DBF
```

```
Field Name         Type           Width       Dec
POSNID             Character          2
COUPLEID           Character          4
YEAR               Numeric            4
```

Suppose we want to produce a list of the board of directors. Here's a first attempt at a query that assembles the board list:

```
SELECT Board.Position, ;
       Her.FirstName AS HerFirst, ;
       Her.LastName AS HerLast, ;
       His.FirstName AS HisFirst, ;
       His.LastName AS HisLast ;
  FROM Couple ;
    RIGHT JOIN OnBoard ;
      RIGHT JOIN Board ;
      ON OnBoard.PosnId = Board.PosnId ;
    ON Couple.CoupleId = OnBoard.CoupleId  ;
    JOIN Member Her ;
      ON Couple.HerId = Her.MemberId ;
    JOIN Member His ;
      ON Couple.HisId = His.MemberId
```

We use a right join between OnBoard and Board to be sure to pick up any unfilled positions and another right join between that result and Couple to carry the results along. But that's not good enough. When we join the His and Her versions of the Member table to get the actual names of the board members, we lose the unfilled positions. We have to continue to use outer joins each step of the way once we've used one. Here's a version that works:

```
SELECT Board.Position, ;
       Her.FirstName AS HerFirst, ;
       Her.LastName AS HerLast, ;
       His.FirstName AS HisFirst, ;
       His.LastName AS HisLast ;
  FROM Couple ;
    RIGHT JOIN OnBoard ;
      RIGHT JOIN Board ;
      ON OnBoard.PosnId = Board.PosnId ;
    ON Couple.CoupleId = OnBoard.CoupleId  ;
    LEFT JOIN Member Her ;
      ON Couple.HerId = Her.MemberId ;
    LEFT JOIN Member His ;
      ON Couple.HisId = His.MemberId
```

The problem with this structure is that, in some situations, it may include records that aren't wanted. In this query, every record in Couple that has a match in OnBoard is included in the two joins with Member. Suppose we were only interested in members of the board who are married couples and wanted to omit the Couple records with either HisId or HerId blank. (In this case, it's probably not an issue since only member couples should have matches in OnBoard. Staff should never show up there.) To do so, we'd have to restructure the query to perform the joins with Member first, then do the right joins with OnBoard and Board.

### Combining outer joins with filters

Filters in the WHERE clause make it even harder to get correct results with an outer join. Suppose we want a query that gives a list of every couple, including their home phone number, if it's listed. The first attempt is:

```
SELECT Her.FirstName AS HerFirst, ;
       Her.LastName AS HerLast, ;
       His.FirstName AS HisFirst, ;
       His.LastName AS HisLast, ;
       Phones.Phone ;
  FROM Couple ;
    LEFT JOIN Member Her ;
      ON Couple.HerId = Her.MemberId ;
    LEFT JOIN Member His ;
      ON Couple.HisId = His.MemberId ;
    LEFT JOIN Phones ;
      ON Couple.CoupleId = Phones.CoupleId ;
 WHERE Phones.PhoneType = "H"
```

This is the same query we've been using to match up couples with their phone numbers, except that we've added the condition that Phones.PhoneType must be "H". But this doesn't work because joins are performed before filters. The query does an outer join that includes at least one record for each couple, then filters out those records that don't have PhoneType="H". There are two problems with this, for our situation. First, some of the couples with no home phone may have other phone numbers listed, so no "null" record is created for them. Second, by explicitly accepting only records with PhoneType="H", we filter out any records that have .NULL. in PhoneType as a result of having been added by an outer join.

In a situation like this, where the filter condition we want to use actually changes the way we want the outer join to operate, we need to make the filter part of the join condition, like this:

```
SELECT Her.FirstName AS HerFirst, ;
       Her.LastName AS HerLast, ;
       His.FirstName AS HisFirst, ;
       His.LastName AS HisLast, ;
       Phones.Phone ;
  FROM Couple ;
    LEFT JOIN Member Her ;
      ON Couple.HerId = Her.MemberId ;
    LEFT JOIN Member His ;
      ON Couple.HisId = His.MemberId ;
    LEFT JOIN Phones ;
      ON Couple.CoupleId = Phones.CoupleId ;
         AND Phones.PhoneType = "H"
```

In this query, we remove non-home phone records as part of the join and then the outer join ensures that we have at least one record per couple. We'd use a similar structure to get a list of board members for a particular year.

Not every filter condition needs to be moved up this way. If the filter doesn't impact on the outer join, it can remain in the WHERE clause. For example, in our membership list query, any filter based on the Couple table can be put in the WHERE clause.

Filters that specifically should eliminate "outer joined" records can also go into the WHERE clause. For example, if we want to see only members born in a specific year, a condition like YEAR(His.Birthdate) = 1958 OR YEAR(Her.Birthdate) = 1958 could go into the WHERE clause:

```
SELECT Her.FirstName AS HerFirst, ;
       Her.LastName AS HerLast, ;
       His.FirstName AS HisFirst, ;
       His.LastName AS HisLast, ;
       Phones.Phone, ;
       Her.BirthDate AS HerBday, ;
       His.BirthDate AS HisBday ;
  FROM Couple ;
    LEFT JOIN Member Her ;
      ON Couple.HerId = Her.MemberId ;
    LEFT JOIN Member His ;
      ON Couple.HisId = His.MemberId ;
    LEFT JOIN Phones ;
      ON Couple.CoupleId = Phones.CoupleId ;
 WHERE YEAR(His.birthdate)=1958 ;
    OR YEAR(Her.birthdate)=1958
```

There's one additional caution about mixing filters with outer joins. Fields that are .NULL. because the record was added by an outer join test as less than actual values. This is probably most relevant when filtering on dates since a query like the preceding is unusual. Usually, we want all records before a specified date (older) or after the date (younger). But "outer joined" records turn up only in the older list, not in the younger:

```
SELECT Her.FirstName AS HerFirst, ;
       Her.LastName AS HerLast, ;
       His.FirstName AS HisFirst, ;
       His.LastName AS HisLast, ;
       Phones.Phone, ;
       Her.BirthDate AS HerBday, ;
       His.BirthDate AS HisBday ;
  FROM Couple ;
    LEFT JOIN Member Her ;
      ON Couple.HerId = Her.MemberId ;
    LEFT JOIN Member His ;
      ON Couple.HisId = His.MemberId ;
    LEFT JOIN Phones ;
      ON Couple.CoupleId = Phones.CoupleId ;
 WHERE YEAR(his.birthdate) < 1958 ;
    OR YEAR(Her.birthdate) < 1958
```

This query includes records with no birth dates as well as records for all the individual members (the "couples" with only one person). It's easy to see why the people with no recorded birth date show up. YEAR({ / / }) is 0, so a less than test picks it up. But why do the records with .NULL. for a birth date pass the test when they don't pass the reverse greater than test? I suspect it's because the testing is done at some intermediate processing stage when the outer join has been created, but the fields haven't yet been filled with null and so are, in fact, empty.

The most important thing to remember about using filters with outer joins is that the interactions are complex and you should test each query with a small, diverse data set before feeling confident that your queries work.

### Counting with Outer Joins

Going back to the Tastrade data, suppose we want to count the number of orders placed by each customer and include 0 for any customer who hasn't ordered. We might write this query:

```
SELECT Company_Name,COUNT(*) AS OrderCount;
  FROM Customer ;
    LEFT JOIN Orders ;
      ON Customer.Customer_Id = Orders.Customer_Id ;
  GROUP BY Customer.Customer_Id
```

A look at the results shows no records with OrderCount = 0. However, a look at the original data shows us that Uncle's Food Factory doesn't have any orders. Why does it contain 1 for OrderCount?

To see why, you have to understand the way the aggregate functions (COUNT(), SUM(), AVG(), MIN() and MAX()) and the GROUP BY clause work. First, the query performs joins and filtering and produces an intermediate result. The GROUP BY clause is applied to that intermediate result with the functions performed against those records. COUNT(*) indicates that we want the number of records in the group. Since every customer is included in the intermediate result, there's at least one record per customer, even though it may not be the result of a match with Orders.

The secret is to specify a field from Orders in the COUNT() function, for example, COUNT(Order_Id). The Order_Id field for the unmatched records contains .NULL., which is omitted by the aggregate functions. So, to count the number of orders per customer, we use:

```
SELECT Company_Name, COUNT(Order_Id) AS OrderCount;
  FROM Customer ;
    LEFT JOIN Orders ;
    ON Customer.Customer_Id = Orders.Customer_Id ;
  GROUP BY Customer.Customer_Id
```

The other aggregate functions don't run into this problem because they always require a field name.

# Improving Query Performance

There are a number of things you can do to optimize the performance of your queries (many of them will improve the speed of your Xbase code as well). In addition, VFP5 and later make it easier to see what part of a query needs improvement.

## Having the right tags

FoxPro's Rushmore optimization system is based on using available index tags to avoid reading actual data from the disk. When tags are available, FoxPro reads the (smaller) index files instead of the data and creates a list of records that meet the specified conditions. Then, only those records are retrieved.

In some cases, tags exist to optimize some conditions, but not others, so FoxPro handles the optimizable conditions first, then performs a sequential search through the records

that meet those conditions to test the others. (Actually, sometimes FoxPro creates its own temporary index tags for those conditions, but this is still slower than having the tag in the first place.)

Obviously, the more of your conditions FoxPro can optimize by using existing tags, the faster your queries will run.

What makes a condition optimizable? In filters, it's a tag whose key *exactly* matches the left-hand side of the condition. The word "exactly" is important here. If the left-hand side is sort of like the key, it's not good enough. For example, if you have an index on UPPER(LastName), a WHERE condition of LastName="Smith" can't be optimized. You need to use UPPER(LastName)="SMITH" instead.

Surprisingly, placing the optimizable expression on the left-hand side is also important. Given the same tag on UPPER(LastName), the WHERE condition "SMITH"=UPPER(LastName) is *not* optimized.

Joins also need an exact match to the tag – a tag on UPPER(LastName) won't optimize a join condition based on LastName. However, there may be tags for each side of the condition. The rule is that only one index tag is applied, but you can't predict which one it is. If only one table involved has an appropriate tag, that one is used, but if both tables have tags, VFP decides which one helps more. Sometimes, VFP rejects all available tags and creates its own.

**Deletion status counts**

FoxPro's two-stage deletion process (DELETE a record now and it's marked for removal at the next PACK) can have an effect on optimization. The SET DELETED command determines whether deleted records are included in query results or not. When DELETED is ON, records marked for deletion are filtered out.

The key word is "filtered." Removal of deleted records works just like a filter condition. With no help from you, each record's deleted status is checked sequentially. The process can be optimized however by providing an index tag with a key of DELETED().

Note that it doesn't matter whether any records are actually deleted or not. The issue is whether DELETED is ON or OFF. If it's ON, a tag on DELETED() helps to optimize your queries. If DELETED is OFF, the tag on DELETED() still helps if you include a WHERE condition of DELETED() or NOT DELETED() in your query.

**Memory matters – a lot**

In testing the sections above (both of which have been part of FoxPro folklore for a long time), I ran into a problem. I didn't see the results I expected. Generally, adding a tag on a field used in a join or filter sped things up, but not by orders of magnitude, as I'd expected. A tag on DELETED() sometimes seemed to slow things down very slightly, not speed them up. I asked someone else to test and he got similar results.

I knew I'd tested these rules in the past and seen tremendous differences. What was going on in this case? The other tester and I both had enormous quantities of memory and a large page file (in NT 4) which meant that large tables could be kept in memory and even huge ones could be moved in and out of memory without much difficulty. In this situation, using optimizable expressions didn't matter much and, in fact, as the number of tags grew so that the index file took up more room, things could slow down.

How can you deal with this if you don't know how capable users' machines will be? The slowdowns from additional tags, when they occurred, were tiny while the potential gains from adding tags are huge. So the lesson is clearly to add the tags. Users on loaded machines won't notice the difference, but users on low-end machines will appreciate all optimization.

The other lesson is that, as the Fox people have been telling us for a long time, memory is perhaps *the* most important factor in how fast FoxPro processes data.

However, FoxPro can have too much memory, too. When VFP starts, it grabs a chunk of memory to work with, usually about half of the available memory. (You can check with SYS(3050,1). Since other things are running at the same time (at least Windows, often much more), and those applications need memory as well, VFP can end up swapping data out to disk rather than using only actual, physical memory. VFP knows how to manage the memory it's been allocated extremely efficiently, but writing to disk is always slow.

Mac Rubel has done extensive testing in this area. His results are documented in a series of FoxPro Advisor articles. In general, it's wise to use SYS(3050) to *lower* VFP's memory allocation so that it only works with physical memory. On my 64-MB machine, I find that VFP's performance is maximized by setting the memory allocation around 24MB with a call like this SYS(3050, 1, 24000000).

## Testing Optimization

Until VFP5, the only way to figure out whether a query was fully optimized was to test it, over and over, until you found the fastest arrangement. Getting it right is difficult since test data sets are often smaller than the actual data to be used and many factors can impact the speed of an operation. (In addition, the memory issues above can make it very hard to tell which version will really be faster on a user's machine.)

In VFP5 and later, it's much easier to see whether a query is fully optimized and, if not, what is and what's not optimized. The SYS(3054) function controls a feature called SQL ShowPlan. There are three settings:

- SYS(3054,0) - turn off SQL ShowPlan
- SYS(3054,1) - turn on SQL ShowPlan for filters only
- SYS(3054,11) - turn on SQL ShowPlan for filters and joins

Note that the third setting is not documented in VFP5.

In VFP5, issuing SYS(3054) immediately produces a message in the active window, indicating the ShowPlan state. This message has been removed in VFP6. Instead, the function returns, as a character string, the setting you passed it.

The output from the function (described below) also appears in the active window. You can send all the messages elsewhere using SET ALTERNATE and prevent it from appearing by defining and activating a window off-screen.

One warning. Don't turn SYS(3054) on while you're running actual timing tests. The function slows things down and interferes with the results, so perform the two kinds of tests separately.

### Checking Filters for Optimization

For filters, SQL ShowPlan shows two kinds of information. First, it indicates which tags are being used to filter the table. Then, it provides an assessment of optimization for the table: none, partial or full.

Here's a simple query involving the Tastrade Customer table:

```
SELECT customer.company_name ;
  FROM customer ;
  WHERE company_name="H" ;
  INTO CURSOR test
```

ShowPlan provides the following information:

```
Rushmore optimization level for table customer: none
```

A look at the tags for Customer shows that, while there's a tag called Company_Na, the key for it is UPPER(Company_Name). A slight change to the query:

```
SELECT customer.company_name ;
  FROM customer ;
  WHERE UPPER(company_name)="H" ;
  INTO CURSOR test
```

gives us this ShowPlan output:

```
Using index tag Company_na  to rushmore optimize table customer
Rushmore optimization level for table customer: full
```

Matching the tag takes the query from totally unoptimized to fully optimized.

Be aware that ShowPlan indicates an optimization level of none for a table that's not filtered in the query. For example, for this query:

```
SELECT customer.company_name, orders.Order_date ;
  FROM customer ;
    JOIN orders ;
      ON customer.customer_id = orders.customer_id ;
  WHERE UPPER(company_name)="H" ;
  INTO CURSOR test
```

the filter-only version of ShowPlan gives this feedback:

```
Using index tag Company_na  to rushmore optimize table customer
Rushmore optimization level for table customer: full
Rushmore optimization level for table orders: none
```

The optimization level for Orders is none because there are no filters on Orders to be optimized.

ShowPlan lets us see the effect of a tag for DELETED() as well (even when our timing tests don't). SET DELETED ON and run the query above and the ShowPlan gives:

```
Using index tag Company_na  to rushmore optimize table customer
Rushmore optimization level for table customer: partial
Rushmore optimization level for table orders: none
```

The optimization level for Customer is now partial because of the implied filter created by SET DELETED. Add a tag based on DELETED() (call it IsDel) and the ShowPlan output becomes:

```
Using index tag Company_na  to rushmore optimize table customer
Using index tag Isdel to rushmore optimize table customer
Rushmore optimization level for table customer: full
Rushmore optimization level for table orders: none
```

The new IsDel tag helps to optimize the query.

### Checking Joins for optimization

When ShowPlan is enabled for joins as well as filters, the output also includes one line for each join, indicating how it was optimized if at all. The output even includes lines for any missing join conditions, indicating that a Cartesian join was used. (A Cartesian join is one in which every record of one table is matched with every record of another table.)

Here's the join portion of the ShowPlan output for the simple query above:

```
Joining table customer and table orders using index tag Customer_I
```

It indicates which tables were joined and which tag, if any, was used to join them. At most one tag is used even if both tables have appropriate tags. VFP decides which tag to use, if multiple tags are available.

If no tag is available, the ShowPlan output says "using temp index". Sometimes, ShowPlan says it's using a temporary index even when one table has a tag that applies. It appears that this happens when the tables are of very different sizes and only the smaller table has a tag. VFP decides that creating a temporary tag for the larger table is more efficient than using the existing tag of the smaller table.

For multi-table joins, the order in which join information appears indicates the order in which the joins are actually being performed. This order may be quite different from the logical order of the joins described above. For example, this query:

```
SELECT customer.company_name, ;
       orders.order_date, ;
       order_line_items.quantity, ;
       products.English_name ;
  FROM customer ;
    JOIN orders ;
      JOIN order_line_items ;
        JOIN products ;
        ON order_line_items.product_id = ;
           products.product_id ;
      ON orders.order_id = order_line_items.order_id ;
```

```
    ON customer.customer_id = orders.customer_id
```

produces the following ShowPlan output:

```
Rushmore optimization level for table customer: none
Rushmore optimization level for table orders: none
Rushmore optimization level for table order_line_items: none
Rushmore optimization level for table products: none
Joining table customer and table orders using index tag Customer_i
Joining intermediate result and table order_line_items using index tag Order_id
Joining table products and intermediate result using temp index
```

The tables here are joined in exactly the reverse order from what we'd expect based on the structure of the query. The sizes of the tables and the tags available lead the VFP engine to believe this is the optimal join order. (Optimization of filter conditions is none for all tables because the query contains no filter conditions.)

An earlier section of these notes discussed the two approaches for join conditions, nested and sequential. It appears that FoxPro uses the syntax to understand the desired result, but then joins the tables in the most efficient order. That is, from an optimization point of view, it doesn't matter whether you use nested syntax or sequential.

Adding an outer join changes the optimization result. If we make the join between Customer and Orders in the example above a left outer join so all customers appear in the result, ShowPlan produces this output:

```
Rushmore optimization level for table orders: none
Rushmore optimization level for table order_line_items: none
Rushmore optimization level for table products: none
Joining table orders and table order_line_items using index tag Order_id
Joining table products and intermediate result using temp index
Rushmore optimization level for table customer: none
Rushmore optimization level for intermediate result: none
Joining table customer and intermediate result using temp index
```

With outer joins involved, the nested vs. sequential issue is a little more significant since FoxPro can no longer rearrange the joins as it pleases. However, my tests show that, in most cases, the type of join syntax used makes little difference. When tables of vastly different sizes are involved in an outer join, the choice of syntax might make a difference since it may determine the order in which the tables are actually joined.

### Using ShowPlan information

The output from ShowPlan can help us to optimize queries. The simplest case, of course, is to add a tag or modify a condition so that it uses an existing tag, as in the UPPER(Company_Name) example above.

However, there are times when you know more about your data than VFP does. In such cases, you may want to insist that joins be performed in a certain order. The FORCE clause of SELECT (added in VFP 5) lets you specify that joins must be performed in the order listed rather than having the optimizer try to figure out the best choice. You can also use parentheses around join clauses to force some joins to be performed before others.

You can also use FORCE when you've determined the optimal order for joins. Arrange the query so that the joins are listed in that order and add the FORCE clause to prevent the VFP engine from trying to figure out which way to do things. This saves the time the optimizer would require to figure out which order to use.

### Optimization *can* be a problem

One of the ways VFP (and earlier versions of FoxPro) optimizes queries is by taking a shortcut. If a query is fully optimizable, involves a single table, has no calculated fields and no grouping, and puts its results in a cursor, VFP simply filters the source table. This saves the time needed to actually create the cursor. You can tell when VFP has done so by checking DBF() for the result cursor. For example, with DELETED OFF, the following query:

```
SELECT First_Name, Last_Name ;
  FROM Employee ;
    WHERE Group_Id = "     3" ;
  INTO CURSOR Group3
```

creates a cursor which is simply a filtered view of the Employee table. Checking DBF("Group3") produces:

```
H:\VFP5\SAMPLES\TASTRADE\DATA\EMPLOYEE.DBF
```

Cursors that are really filtered tables can't be used in certain operations and can give bad results in others. In previous versions of FoxPro and VFP, the only solution was to force the query to be less than fully optimizable. In VFP5 and later, the new NOFILTER clause forces VFP to create a "real" cursor. For example, after executing:

```
SELECT First_Name, Last_Name ;
  FROM Employee ;
    WHERE Group_Id = "     3" ;
  INTO CURSOR Group3 NOFILTER
```

a check of DBF("Group3") shows a temporary file such as:

```
C:\TEMP\08334986.TMP
```

In addition, VFP doesn't remember to apply a filter of DELETED()=.F. to the original table when a query is run with DELETED ON and no real cursor is created. If DELETED is subsequently turned OFF, the filtered cursor contains the deleted records of the original that otherwise meet its criteria. This can give wrong results.

Because of the bug involving deleted records, it's best to use NOFILTER any time VFP might fail to create a real cursor. Make exceptions only when the speed gain from filtering far outweighs the risk of mishandling deleted records.

## Summary

The new features introduced in VFP5, especially outer joins, make it possible to perform more and more of our data gathering using SELECT. However, the new syntax also makes it easier than ever to produce wrong or incomplete results. It's essential to test queries against real, complete data sets to ensure that they include exactly the right set of records.

The SYS(3054) function makes tuning queries simpler than in previous versions. Combined with the FORCE clause, you can squeeze every little bit out of performance from VFP.

## Acknowledgements

Andy Neil, Steve Sawyer, Anders Altberg and Chin Bae all helped me to assimilate and understand the new join syntax. Jim Slater tested my optimization results. Mark Wilden pointed out the problem with deleted records and filtered cursors. Andy Neil also reviewed an earlier version of these session notes and suggested several improvements. Thanks to all of them for making these notes better.

Copyright, 1998, Tamar E. Granor, Ph.D.