Make Your Queries Fly

VFP provides the tools to let you figure out why SQL commands are slow. Learn how to check query optimization with a pair of built-in functions.

Tamar E. Granor, Ph.D.

In the January, 2010 issue, I wrote about changes in VFP's SQL commands in VFP 8 and 9. But one of the key elements in using SQL commands is that they can be very fast. When they're not, you need to figure out why. Fortunately, VFP includes a couple of functions that help you do so.

A key benefit of writing SQL commands rather than traditional Xbase commands is that you tell the computer what you want, but you don't have to figure out how to get it. That allows VFP's SQL engine to figure out the best way to get the results you've requested. The engine does its best to give you what you want as fast as it can, but sometimes that's still not good enough.

When a SQL command is too slow, you need to see what the engine is doing, so you can change something to make it faster. VFP provides two functions that let you understand what the engine doing, SYS(3054) and SYS(3092).

How VFP optimizes

Before looking at the functions, you need a basic understanding of how VFP's optimizer, known as Rushmore, works. Rushmore is based on indexes. When a command filters on an expression and there's an index for that expression, Rushmore uses the index to find the matching records rather than searching sequentially through the table. In almost every case, reading the index is faster than reading the actual records. For Rushmore to use an index, the index key must exactly match the expression in the command.

When a command involves multiple filter conditions, Rushmore looks for an index for each condition separately. It then takes whichever indexes it finds and looks to see which records they have in common.

Of course, this is an oversimplification. Conditions combined with AND are handled differently than conditions combined with OR. But the key point is that, for each optimizable condition, Rushmore reads into memory only the portion of the index that identifies records matching that condition, and then creates a bitmap indicating which records in the table fit. The bitmaps for the various conditions are combined appropriately.

Once all optimizable conditions have been combined, those records they identify are pulled into memory, and any remaining conditions are checked sequentially against those records.

Thus, the key element in optimizing SQL commands is ensuring that appropriate index tags exist. Not surprisingly then, the functions that let you see how your queries are being optimized present their results in terms of what tags the engine uses.

Seeing Rushmore's plan

The SYS(3054) function, added in VFP 5, is known as "SQL ShowPlan" because it displays the plan for executing a SQL command. It has several modes of operation, shown in Table 1.

When turning SQL ShowPlan on, you have two things to decide; there are two choices for each of them. The first decision is whether to show the plan for filters only or for both filters and joins; I rarely choose filters only because when I'm testing optimization, I want to know everything about how Rushmore is working. The second decision is whether to include the query itself in the output. For Command Window testing, that can be overkill, but when testing in an application (especially in conjunction with SYS(3092), it's very handy.

Table 1. SYS(3054) lets you turn SQL ShowPlan on and off. You also determine whether the command being tested is included in the output.

Value	Meaning
0	Turn off SQL ShowPlan
1	Turn on SQL ShowPlan for filters only
2	Turn on SQL ShowPlan for filters only and include the command in the output
11	Turn on SQL ShowPlan for filters and joins
12	Turn on SQL ShowPlan for filters and joins and include the command in the output

Let's look at some examples. I'll use data from the Northwind database that comes with VFP.

Examining filter optimization

We'll start by looking at optimization of filters only. Pass 1 or 2 to SYS(3054) to see the optimization plan for filters, but not for joins.

Listing 1 shows code to turn on SQL ShowPlan, then run a query, then turn SQL ShowPlan off. Figure 1 shows the output, which contains information about every tag used for optimization, plus the "optimization level" for each table in the query.

Listing 1. When you pass 1 or 2 to SYS(3054), it shows optimization only for filters. This code demonstrates.

```
SYS(3054,1)
SELECT OrderID, OrderDate, ;
       Customers.CompanyName AS Customer, ;
       Employees.LastName, ;
       Employees.FirstName, ;
       Shippers.CompanyName AS Shipper ;
  FROM Orders ;
   JOIN Customers ;
     ON Orders.CustomerID = ;
        Customers.CustomerID ;
   JOIN Employees ;
     ON Orders.EmployeeID = ;
        Employees.EmployeeID ;
   JOIN Shippers;
     ON Orders.ShipVia = Shippers.ShipperID ;
  WHERE BETWEEN(OrderDate, {^ 1997-2-1}, ;
                {^ 1997-2-28}) ;
  ORDER BY OrderDate DESC, LastName ;
  INTO CURSOR csrOrderInfo
SYS(3054, 0)
```

```
Using index tag Orderdate to rushmore optimize table orders
Rushmore optimization level for table orders: full
Rushmore optimization level for table customers: none
Rushmore optimization level for table employees: none
Rushmore optimization level for table shippers: none
```

Figure 1. The output from SYS(3054) shows the tags used to optimize each table and summarizes the optimization result for each.

At first glance, the information in Figure 1 might seem alarming, as it shows no optimization for three of the four tables in the query. However, a look at the query shows that only the Orders table is filtered, so there's no need to optimize the others.

"Full" and "none" aren't the only possible results. The optimization level for a table can also be "partial"; this occurs when there's more than one filter for a table, at least one of the filters can be optimized and at least one cannot. Listing 2 shows a query with two filters on the Northwind Customers table. That table has an index tag on UPPER(City), but has no tag on Country. Figure 2 shows the output.

Listing 2. When there's more than one filter for a table, as in this query, Rushmore optimizes what it can.

```
SYS(3054,1)

SELECT CompanyName ;
   FROM Customers ;
   WHERE Country = "UK" ;
     AND UPPER(City) = "LONDON" ;
   INTO CURSOR csrLondonEngland

SYS(3054,0)
```

```
Using index tag City to rushmore optimize table customers
Rushmore optimization level for table customers: partial
```

Figure 2. "Partial" optimization for a table indicates that there's more than one filter, and at least one cannot be optimized.

One case where you'll often see "partial" is with SET DELETED ON. Unless you have an index on DELETED(), every table with an optimizable filter condition will show "partial."

Even so, having a tag on DELETED() for each table isn't always the best choice. This is one of the cases where reading the relevant portion of the index may take longer than individually checking records not otherwise filtered. More on this subject in my next article.

Examining join optimization

Optimization of joins is a little more interesting. First, it tells you the order in which the joins were performed, which can be quite different from the order in which they appear in the query. In addition, often there are two index tags that can be used to optimize a join, one for each table. So Rushmore has to figure out which tag to use for a given join and what order of joins offers the best performance.

Pass 11 to SYS(3054) to see filter and join optimization. Pass 12 for the same information, plus the query itself. Figure 3 shows the output for

```
Using index tag Orderdate to rushmore optimize table orders
Rushmore optimization level for table orders: full
Rushmore optimization level for table customers: none
Rushmore optimization level for table employees: none
Rushmore optimization level for table shippers: none
Joining table employees and table orders using index tag Employeeid
Joining table shippers and intermediate result using temp index
Joining intermediate result and table customers using index tag Customerid
```

Figure 3. When you pass 11 or 12 to SYS(3054), you see optmization information for both filters and joins, including the order in which the joins are performed.

the query in Listing 1, but using SYS(3054, 11).

In this example, where all joins are inner joins, all the filtering is done first and then tables are joined. The logical order of joins, specified in the query, is Orders to Customers, then that result to Employees, and finally that result to Shippers. But the SYS(3054) output tells us that VFP first joined Employees and Orders, then joined that result to Shippers and finally joined that result with Customers.

When showing optimization of a join, SYS(3054) lists the table whose index was used for optimization second. So, in the example, the join between Employees and Orders was optimized using the EmployeeID index of Orders (which makes sense, as it's a much bigger table than Employees). For the join between that initial result and Shippers, the VFP engine decided that no existing index would be useful and created an index on the fly (listed as "temp index"). Again, that makes sense, because Shippers is a tiny table (with only three records),

so none of its indexes would help speed things up. Nonetheless, for the final join, an existing index on the Customers table was seen as offering more help than creating an index on the intermediate result.

When a query involves outer joins, optimzation results for filters and joins may be intermingled. That's because outer joins limit the order in which joins can be performed and may require some filters to be executed later than they would be with inner joins. Listing 3 shows a query involving an outer join; it totals the number of items and the total price for all seafood (category=8) items for all customers in France. The RIGHT JOIN of Customers ensures that every French customer is included in the output. Figure 4 shows the optimization plan.

Listing 3. Outer joins change the optimization picture, since they force some of the joins to happen after other joins.

```
SYS(3054, 11)

SELECT CompanyName, SUM(Quantity), ;
       SUM(Quantity * OrderDetails.UnitPrice);
  FROM Products ;
    JOIN OrderDetails ;
      ON OrderDetails.ProductID = ;
         Products.ProductID ;
        AND CategoryID = 8 ;
    JOIN Orders ;
      ON Orders.OrderID = ;
         OrderDetails.OrderID ;
    RIGHT JOIN Customers ;
      ON Customers.CustomerID = ;
         Orders.CustomerID ;
  WHERE Customers.Country = 'France' ;
  GROUP BY 1 ;
  INTO CURSOR csrSeafoodOrdersFrenchCustomers

SYS(3054, 0)

Using index tag Categoryid to rushmore optimize table products
Rushmore optimization level for table products: full
Rushmore optimization level for table orderdetails: none
Rushmore optimization level for table orders: none
Joining table products and table orderdetails using index tag Productid
Joining intermediate result and table orders using index tag Orderid
Rushmore optimization level for intermediate result: none
Rushmore optimization level for table customers: none
Joining table customers and intermediate result using temp index
```

Figure 4. When a query contains an outer join, the filters and joins can be mixed.

The results tell us that Products was filtered first (and that OrderDetails and Orders would have been filtered at the same time, if there were any filters on those tables), then joined with OrderDetails using the OrderDetails.ProductID tag. That intermediate result was joined with Orders using the OrderID tag from Orders. Then, the filter on the Country field of Customers was applied, but not optimized (because there's no tag on Country). Finally, the Customers table was joined with the intermediate result, and a temporary index was created, which makes sense because by this point, there would be quite a few records in the intermediate table.

In addition to using an existing index or creating an index on the fly, the output can indicate that no optimization was possible because a Cartesian join occurred. A Cartesian join is also known as a cross join or Cartesian product; it occurs when every record from one table is matched to every record from another table. Normally that's something you want to avoid, although there are a few cases where a Cartesian join is helpful in getting desired results.

Listing 4 shows a complex SQL INSERT that adds records to a data warehouse. The goal is to add one record to the warehouse for each combination of employee and product, showing how much of the product the employee sold in the specified year (indicated by nYear). Some employees may not have sold any of some products in the specified year, so we use a Cartesian join between Employees and Products to get every combination into the result. Figure 5 shows SYS(3054, 11) output for this command.

Listing 4. While Cartesian joins are normally to be avoided, in some situations, they solve a problem. The Cartesian join in this command ensures that we insert a record for every combination of employee and product.

```
INSERT INTO Warehouse ;
SELECT CrossProd.ProductID, ;
       CrossProd.EmployeeID, ;
       m.nYear as Year, ;
       NVL(UnitsSold, 0), NVL(TotalSales, $0);
  FROM ( ;
      SELECT Employees.EmployeeID, ;
             Products.ProductID ;
        FROM Employees, Products) ;
      AS CrossProd ;
    LEFT JOIN ( ;
      SELECT ProductID, EmployeeID, ;
             SUM(Quantity) AS UnitsSold, ;
             SUM(Quantity * UnitPrice) ;
               AS TotalSales ;
        FROM Orders ;
          JOIN OrderDetails ;
            ON Orders.OrderID = ;
               OrderDetails.OrderID ;
        WHERE YEAR(OrderDate) = m.nYear ;
        GROUP BY ProductID, EmployeeID ) ;
      AS AnnualSales ;
      ON CrossProd.EmployeeID = ;
         AnnualSales.EmployeeID  ;
      AND CrossProd.ProductID = ;
          AnnualSales.ProductID ;
  ORDER BY 2, 1


Rushmore optimization level for table employees: none
Rushmore optimization level for table products: none
Joining table employees and table products (Cartesian product)
Rushmore optimization level for table orders: none
Rushmore optimization level for table orderdetails: none
Joining table orders and table orderdetails using index tag Orderid
Rushmore optimization level for intermediate result: none
Rushmore optimization level for intermediate result: none
Joining intermediate result and intermediate result using temp index
Rushmore optimization level for table orders: none
Rushmore optimization level for table orderdetails: none
Joining table orders and table orderdetails using index tag Orderid
Rushmore optimization level for intermediate result: none
Rushmore optimization level for table employees: none
Joining table employees and intermediate result using temp index
```

Figure 5. Normally, seeing "(Cartesian product)" in SYS(3054) output is a red flag. Here, it's not a problem because the Cartesian join was intentional.

Managing ShowPlan output

By default, SYS(3054) sends its output to the active window. In VFP 7 and later, you can capture the output to a variable instead. Pass the name of the variable as the third parameter. (Note that you must pass the name, not the variable itself; that's because there's no way to pass parameters by reference to VFP's built-in functions.) Listing 5 demonstrates; after running a query, the variable cOptInfo contains the optimization information.

Listing 5. When you pass the name of a variable as the third parameter to SYS(3054), the optimization information is stored in that variable.

```
SYS(3054, 12, "cOptInfo")
```

The variable can hold the results for only a single SQL command. That is, you pass a variable name to SYS(3054) and run a SQL command. If you then run another SQL command, the variable is cleared and only the results from the second command are saved. This behavior makes it very difficult to take advantage of SYS(3054) in an application setting.

In VFP 9, the Fox team introduced a better approach that allows you to track optimization throughout an application. SYS(3092) lets you send optimization information to a file; call it before you set SYS(3054). The syntax for SYS(3092) is shown in Listing 6.

Listing 6. SYS(3092) lets you indicate where to store the optimization information produced by SYS(3054).

```
cLogFile = SYS(3092 [, cFileName
                [, lAdditive ] ] )
```

Call the function with no additional parameters (just SYS(3092)) to find the name of the currently active log file. When you pass a file name as the second parameter, that file becomes the active log file and the function returns that value. The lAdditive parameter determines whether new data is added to an existing file or the file is emptied first. Regardless, once you turn logging on, all SQL ShowPlan results are stored to the specified file.

To turn off the log so you can see its contents, call SYS(3092) again, passing the empty string for the file name.

Listing 7 shows a complete example. SYS(3092) sets up a log file, and then SYS(3054) is called to turn on SQL ShowPlan. A query is executed, and then SQL ShowPlan and the log are turned off.

Listing 7. Combine SYS(3054) with SYS(3092) to let you store optimization results in a file.

```
SYS(3092, "Optim.Log")
SYS(3054, 12, "cGrabOutput")
SELECT CustomerID, ;
       COUNT(DISTINCT OrderDate) ;
         AS DatesOrdered, ;
       COUNT(OrderDate) AS TotalOrders ;
   FROM Orders ;
   GROUP BY 1 ;
   INTO CURSOR csrHowManyOrders
SYS(3054, 0)
SYS(3092, "")
```

Turning on logging doesn't keep SYS(3054) from displaying its results in the active window. If you want to keep the output from showing in the active window, pass a variable name to SYS(3054), as in the example.

Logging optimization to a file isn't useful only for tracking multiple SQL commands. It's also handy for figuring out what's going on at a client site. You might set up a hidden mechanism in your application to turn logging of SQL ShowPlan on and off. When a client reports a slowdown, have the client turn logging on, run the troublesome process, turn logging off, and send you the log.

Using ShowPlan results

Once you see how VFP is optimizing your queries, you're halfway to speeding them up. The next step is to examine the results and make changes, to your data, to your code or to both to allow VFP to be smarter. In my next article, I'll look at some common issues in query optimization.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. She currently focuses on working with other developers through consulting and subcontracting. Tamar is author or co-author of nearly a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with VisualFoxPro and Taming Visual FoxPro's SQL . Her latest collaboration is Making Sense of Sedna and SP2. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Support Most Valuable Professional and one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.