Speed Up Your SQL Code

VFP's SQL commands can be blazingly fast, but you have to set things up right.

Tamar E. Granor, Ph.D.

In my last article, I talked about the two functions in VFP that allow you to measure the optimization of SQL commands. This month, I'll look at what you can do to improve performance once you know that a query is sub-optimal.

As I explained last time, VFP's Rushmore engine uses indexes to optimize queries and other commands. Not surprisingly then, ensuring you have appropriate indexes and then exploiting them in your queries is the path to great performance.

Having the right tags

When a command filters on a condition and there's an index for that condition, 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 expression must exactly match the expression in the command. For example, using the Northwind Customers table, the query in **Listing 1** is not fully optimized because the index based on the City field uses the key expression UPPER(City). The alternate version of the query in **Listing 2** is optimized.

Listing 1. When the expression you're matching in a WHERE clause isn't identical to the key expression for an index, that filter condition can't be optimized.

```
SELECT CustomerID, CompanyName ;
       FROM Customers ;
       WHERE City="London" ;
       INTO CURSOR csrLondoners
```

Listing 2. Use the exact index key to get Rushmore optimization.

```
SELECT CustomerID, CompanyName ;
       FROM Customers ;
       WHERE UPPER(City)="LONDON" ;
       INTO CURSOR csrLondoners
```

Because the Customer table is quite small, the actual performance difference between the two versions is also small, but for large tables, matching the index can have a significant impact on performance.

If an index is based on several fields, you need to use the exact index expression in the command. The Northwind database doesn't offer a good example of this, but consider a Person table including fields cFirst and cLast and an index tag on the full name using the expression UPPER(cLast + cFirst). In that case, the query in **Listing 3** is not optimized, but the query in **Listing 4** is. In my tests, the second query runs in about one-third the time of the first.

Listing 3. This query can't be fully optimized because the filter condition doesn't match the index.

```
SELECT cFirst - (" " + cLast) AS FullName ;
   FROM Person ;
   WHERE UPPER(cLast) = "N" ;
   INTO CURSOR csrNamesWithN
```

Listing 4. Even if you only want to match part of a combined index, use the entire index expression for optimization.

```
SELECT cFirst - (" " + cLast) AS FullName ;
   FROM Person ;
   WHERE UPPER(cLast + cFirst) = "N" ;
   INTO CURSOR csrNamesWithN
```

Rushmore cannot use any indexes that are filtered, that is, created with a FOR clause in the INDEX command. It also cannot use any indexes that include the NOT operator; instead, index on the expression without NOT and apply NOT in the filter condition.

Expressions involving BETWEEN() and INLIST() are optimized if the first parameter can be optimized; the same is true for the corresponding SQL operators. However, expressions involving ISBLANK() and EMPTY() cannot be optimized, regardless of the parameter.

As with VFP's Xbase commands, in a filter expression the optimizable expression (the one that exactly matches an index) must be on the left-hand side of the comparison. However, for join conditions in the FROM clause, VFP looks at both sides of the expression and chooses which index to use.

When the WHERE clause contains multiple conditions combined with AND and OR, each condition is considered separately. Those that can be optimized are; the remaining conditions are checked against the records that match the optimizable conditions. VFP 9 improved optimization in queries involving complex conditions using OR.

In **Listing 5**, only those records where the City field contains "London" are read into memory. Then, each is checked to see that the Country field contains "UK" and those from other countries are eliminated. The number of records to check this way is quite small and should have only a minimal impact on performance.

Listing 5. VFP optimizes what it can, and then checks the remaining conditions sequentially.

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

VFP's optimization trick

VFP tries to give you results as fast as possible. However, in one situation, that goal can be a problem. When a query uses a single table, has only fields from that table in the field list (and no expressions or literals), puts the results into a cursor, and can be fully optimized, VFP filters the source table rather than creating a new file on disk. In some cases, such as reporting, this isn't a problem. But if you want to use that cursor in a subsequent query or display it in a grid, this trick causes problems.

You can check whether VFP has taken this route by examining DBF() for the resulting cursor. The code in **Listing 6** demonstrates, using the queries from **Listing 1** and **Listing 2**. If you see a temporary file for both queries, SET DELETED OFF and try again. (See the next section, "Deletion and Optimization" for an explanation.)

Listing 6. Some very simple queries based on a single table filter the original table rather than creating a new disk presence.

```
OPEN DATABASE HOME(2) + "Northwind\Northwind"

SELECT CustomerID, CompanyName ;
        FROM Customers ;
        WHERE UPPER(City)="LONDON" ;
        INTO CURSOR csrLondoners
?'With fully optimized query, , + ;
 ,DBF(„Londoners") = ,, DBF("Londoners")

SELECT CustomerID, CompanyName ;
        FROM Customers ;
        WHERE City="London" ;
        INTO CURSOR csrLondoners
?'With unoptimized query, , + ;
 ,DBF(„Londoners") = ,, DBF("Londoners")
```

In FoxPro 2.x and early versions of VFP, the only ways to work around VFP's good intentions were to force the query to be not fully optimized or to add an expression to the field list. Fortunately, the VFP team realized that there are legitimate reasons for preventing this trick. In VFP 5, the NOFILTER keyword was added; use it after INTO CURSOR to tell VFP to create a new file, no matter what, as in **Listing 7**. In addition, the READWRITE keyword added in VFP 6 has the same effect; in order to ensure that the cursor can be modified, VFP has to create a new file.

Listing 7. Add the NOFILTER clause to force VFP to create a "real" cursor.

```
SELECT CustomerID, CompanyName ;
        FROM Customers ;
        WHERE UPPER(City)="LONDON" ;
        INTO CURSOR Londoners NOFILTER
```

Deletion and optimization

VFP uses a two-step deletion mechanism. When you delete a record, whether you use the Xbase DELETE command, the SQL DELETE command or the deletion column in a grid or Browse, the record is marked as deleted. It's not physically removed from the table until you pack the table using the PACK command.

Because of this structure, VFP includes the SET DELETED command that determines whether other commands see records marked for deletion. SET DELETED OFF to include deleted records in processing; SET DELETED ON to omit them.

Having SET DELETED ON is equivalent to filtering every command on the expression NOT DELETED(). For many years, therefore, the standard advice for those operating with DELETED ON was to create an index for each table based on the DELETED() function, so that the implied filter could be optimized.

However, it turns out that, while an index on DELETED() allows commands to be fully optimized, such commands can be slower than they would be without that index. (The article that explained this paradox is by Chris Probst and appeared in *FoxPro Advisor* in May, 1999; it was updated and reprinted in March 2005.)

The issue is that there are only two possible values for this index and, most often, the vast majority of records have the same value (False). When Rushmore attempts to optimize the implied expression NOT DELETED(), it must read the whole portion of the index that corresponds to False. Especially in a network situation, loading that portion of the index into memory can take much longer than simply checking each record that meets all the other criteria to see whether it's deleted. (Keep in mind that filters that can't be optimized are checked after all the optimizable filters are applied, so in many cases, the number of records to check "manually" for deletion is quite small.) As a result, for a large table and a small result set, you're usually better off without an index based on the DELETED() function; the meaning of "large" and "small" depends on the amount of memory available.

VFP 9 changes the rules a little. The new binary index type is designed to create extremely small indexes for expressions with only two possible values. For example, on a test table with 1,000,000 records, a regular index based on DELETED() resulted in an index file of 3,205,632 bytes. Using a binary index, instead, the file size was 135,168 bytes. Clearly, reading the binary index is less likely to slow a command down.

For small tables where an index on DELETED() was already useful in optimization, a binary index speeds things up even more. Binary indexes also raise the threshold between tables where an index helps and those where an index hurts.

Binary indexes are only for optimization and can't be used for searching

Tuning memory

The Visual FoxPro engine knows how to use memory extremely efficiently, caching data to avoid time-consuming disk access. However, there's one situation where this ability can result in slow code; that's the case where VFP thinks it has memory, but is actually using page files on disk.

When you start VFP, it figures out how much memory to use; normally it takes about half of the physical memory of the computer. However, with other applications (including Windows itself) running, there's a good chance that amount of physical memory isn't actually available, and that behind the scenes Windows is swapping out to disk. But VFP doesn't know it, so it makes decisions that assume all the memory it's using is physical memory.

Fortunately, you can control the amount of memory VFP thinks it has. The function SYS(3050) lets you set both the foreground and background memory available to VFP. The syntax is shown in Listing 8.

Listing 8. You can control how much memory VFP uses. Sometimes, giving it less memory than the default speeds up execution.

```
cMemoryInBytes = SYS(3050, nType [,
                     nMemoryInBytes ] )
```

Pass 1 for nType to set the amount of memory available when VFP is in control (in the foreground) and 2 to set the memory available to VFP when another application is in the foreground. VFP rounds the value you pass for nMemoryInBytes down to the nearest 256K and allocates that much memory for itself. The function returns that value as a character string. Note that the value you pass is in bytes, not KB or GB.

VFP expert Mac Rubel did extensive testing of the effects of SYS(3050) with VFP 6 and VFP 7 and discovered that most often, you want the foreground setting to be less than the default. You need to test in your production environment to find the right setting, but a good place to start is around one-third of physical memory.

Make code pages match

In VFP 9, if the current code page (as indicated by CPCURRENT()) is different from the code page of a table (indicated by CPDBF()), operations involving character expressions from that table cannot use existing indexes for optimization (though VFP may still build temporary indexes). While this may seem arbitrary, it's one of a number of changes in VFP 8 and VFP 9 designed to prevent inaccurate query results.

An index is sorted according to the rules for the table's code page. Even if VFP translated from the table's code page to the current code page, the sort order might be different. This means comparisons to data using the current code page may be incorrect. This is what VFP's new rules prevent.

Although code pages are primarily used to handle different character sets, be aware that a table originally created in a DOS environment may be marked with the DOS code page (437) rather than the Windows ANSI code page (1252). You can use the CPZero.PRG program that comes with VFP to change the code page of those tables.

Make collation sequences match

Like code pages, collation sequences aid in working with languages other than English. A collation sequence indicates the sorting order of the characters.

Every index is created with an associated collation sequence. For optimization, VFP can only use indexes created with the collation sequence in effect when the command executes. In other words, for VFP to take advantage of a tag, IDXCOLLATE() for that tag must match SET("COLLATE"). This is true not only in VFP 9, but in earlier versions as well.

The bottom line

Index tags are the key to good performance in VFP's SQL commands. If a query seems to be slow, test with SYS(3054) to see whether it's optimized. If it's not, tweak either the command or the set of tags you maintain until you get acceptable performance. Remember that tags do have consequences when inserting data into tables, so it's not usually a good idea to add a tag for every field of every table.

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 was a Microsoft Support Most Valuable Professional from the program's inception until 2011; she is also 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.