Can't this application go any faster?

Tamar E. Granor
Tomorrow's Solutions, LLC
Voice: 215-635-1958
Email: tamar@tomorrowssolutionsllc.com

What do you do when your customer says that your application is too slow? How can you figure out what's slowing things down? How can you make it faster?

Optimization of a VFP application is more than just applying Rushmore correctly, though that's an important step. In this session, we'll explore techniques for measuring performance of a VFP application and look at things you can do to speed it up.

Introduction

It came as an email with the words "speed up" in the title. A long-time client was finally ready to address a bottleneck in the application I'd written and that they supplied to their customers. I'd actually wanted to take a look at this piece for quite a while, but it had never come to the top of their priority list until that email.

The process in question involved reading an XML file into DBFs, and then building an object model from the DBF data. While it was tempting to consider simply eliminating the DBFs and going right from XML to objects, I knew that such a change would require touching too many parts of the application.

I was pretty sure I could speed up the first part, going from XML to DBF, without too much digging. The existing code used the XMLAdapter class; I'd read Doug Hennig's FoxRockX article (January, 2013) about using Rick Strahl's dotNetBridge instead. I wasn't sure what I could do with the second part, but figured there must be some room for improvement.

I also realized right away that I needed a way to measure the speed of each process, so I could see whether I was making any improvement.

In this session, we'll look at the available tools for measuring performance, and examine the areas where I tried for improvement, seeing which helped and which didn't. One of the real takeaways for me from this experience was that even established best practices need to be re-examined in the light of slow performance, so we'll also look at performance for some of my best practices.

Measuring performance

There are several different things you can measure in VFP that might constitute "performance." The most obvious is simply how long it takes to do something. But VFP also provides a tool for measuring query optimization, and a set of tools for measuring performance of entire routines and individual lines.

Collecting start and end times

Let's start with the simplest item, measuring how long it takes to do something. You can do this by simply checking the time when you start and the time when you finish. The SECONDS() function is the easiest way to do so. Grab the value just before starting the process and right after finishing, as in **Listing 1**.

Listing 1. The SECONDS() function is the easiest way to measure how long a section of code takes.

```
LOCAL nStart, nEnd

nStart = SECONDS()
* Do the process to be measured.
nEnd = SECONDS()
* Elapsed time is nEnd-nStart
```

However, that approach raises several issues. First, SECONDS() returns the seconds since midnight, so if the test can run across multiple days, you need to collect the start and end dates as well. The DATETIME() function makes it easy to do that and you can still do arithmetic with it. Using DATETIME(), the model changes to **Listing 2**.

Listing 2.Use DATETIME() to measure start and end times if there's any chance your test could run across multiple days.

```
LOCAL tStart, tEnd

tStart = DATETIME()
* Do the process to be measured.
tEnd = DATETIME()
* Elapsed time in seconds is tEnd-tStart
```

The second issue is that in VFP, even slow things can be very fast. That is, seconds may be not a small enough measuring unit. In fact, SECONDS() returns three decimal places (that is, milliseconds). That's one reason you may choose to stick with SECONDS() rather than using DATETIME(), which has only one-second resolution.

But there are still cases where even millisecond resolution is not sufficient. For testing different approaches to a problem to see which is best, the solution is to perform the action multiple times (1000 or 10,000 or 100,000) in a loop. The code in **Listing 3** puts this model to work and demonstrates that FOR loops are about an order of magnitude faster than equivalent DO WHILE loops; it's included in the materials for this session as DoWhileVsFor.PRG.

Listing 3. To compare speed for different approaches, perform each in a loop.

```
#DEFINE PASSES 10000

LOCAL nDOWStart, nDOWEnd, nFORStart, nFOREnd, nPass, nCounter

* First, DO WHILE

nDOWStart = SECONDS()
FOR nPass = 1 TO PASSES
   nCounter = 1
   DO WHILE nCounter <= 1000
      nCounter = m.nCounter + 1
   ENDDO
ENDFOR
nDOWEnd = SECONDS()


* Now, FOR
nFORStart = SECONDS()
FOR nPass = 1 TO PASSES
   FOR nCounter = 1 TO 1000
   ENDFOR
ENDFOR
nFOREnd = SECONDS()
```

```
DEBUG
DEBUGOUT "DO WHILE--", PASSES, " passes:", nDOWEnd-nDOWStart
DEBUGOUT "FOR--", PASSES, " passes:", nFOREnd-nFORStart


RETURN
```

Of course, this approach isn't helpful when testing inside an application, but you can argue that if the elapsed time for a block is less than a millisecond, that's not a place to be putting your optimization efforts.

Christof Wollenhaupt recommends turning the loop idea inside out, and running the test for a fixed period of time, measuring how many times you get through the loop in that period. In this version, the winner is the approach that gives the larger result, indicating that you were able to execute it more times in the specified period. Using that approach, the same test as before looks like **Listing 4**. This version is included in the materials for this session as DoWhileVsForFixedTime.PRG.

Listing 4. This version of a timing test measures how many times you can accomplish a given task in a specified period of time.

```
#DEFINE SECONDSTORUN 5


LOCAL nDOWStart, nDOWEnd, nFORStart, nFOREnd, nDOWPasses, nFORPasses

* First, DO WHILE

nDOWPasses = 0

nDOWStart = SECONDS()
nDOWEnd = m.nDOWStart + SECONDSTORUN

DO WHILE m.nDOWEnd > SECONDS()
   nCounter = 1
   DO WHILE nCounter <= 1000
      nCounter = m.nCounter + 1
   ENDDO
   nDOWPasses = m.nDOWPasses + 1
ENDDO


* Now, FOR
nFORPasses = 0

nFORStart = SECONDS()
nFOREnd = m.nFORStart + SECONDSTORUN

DO WHILE nForEnd > SECONDS()
   FOR nCounter = 1 TO 1000
   ENDFOR
   nFORPasses = m.nFORPasses + 1
ENDDO
```

```
DEBUG
DEBUGOUT "DO WHILE--", m.nDOWPasses, " passes in ", SECONDSTORUN, "seconds"
DEBUGOUT "FOR--", m.nFORPasses, " passes in ", SECONDSTORUN, "seconds"
RETURN
```

This approach helps to smooth out testing, so you're not as dependent on how long the process you're testing actually takes. As long as you make the fixed period long enough to encompass multiple passes, you can run the test on any machine and not have to adjust it. You also know about how long your total test will take. When you choose an arbitrary number of passes, you don't know whether you're going to be waiting one second, ten seconds, ten minutes, or ten hours. You have to make an initial guess at the number of passes to give you a good test without a long wait, and then fine-tune it.

The third issue is that testing in the Windows environment is inherently flawed. That is, between Windows itself and various services that are always running, any one test result might be inaccurate. The solution here has two parts. First, before testing, turn off anything you can that might interfere, such as an email client, on-demand virus scanning, and so forth. Second, perform more than one test for each case. That advice is also important because VFP caches data, so the first time you run a process that uses DBFs, it's likely to take longer than subsequent runs.

You also need to decide what you're going to do with the speed test results. When I do quick comparisons of techniques for speed, I typically just echo the results to the screen with ? or use DEBUGOUT to send them to the Debugger's Debug Output window. (Be careful not to open the Debugger in the middle of a test because all VFP code runs slower with the Debugger open.)

When testing inside an application, though, such ad hoc approaches may not be helpful. That's especially true if you need to be able to test a compiled application. In that case, saving timing results to a file makes sense. My client's application already included a way to log information to a text file, with each message time stamped, so I just used the existing mechanism to send appropriate messages (such as "Starting phase 1") to the log. **Listing 5** shows a simplified version of the logging class I use. Once the class is instantiated, you can call a method with just the message to be logged, as in **Listing 6**, where I've assumed there's an application object with a property to reference the logger. The result is a line like the one shown in **Listing 7**. The materials for this session include Logger.PRG, which includes the class definition and the code to use it.

Listing 5. A simple logging class makes it easy to send information about execution times to a file for analysis.

```
DEFINE CLASS cusLogger AS Custom

cLogFile = ''

PROCEDURE LogIt
LPARAMETERS cLogString, lStartNewLogFile

IF EMPTY(This.cLogFile)
```

```
   This.cLogFile = FORCEPATH("Tracking.Log", SYS(2023))
ENDIF

STRTOFILE(TTOC(DATETIME()) + ":" + m.cLogString + CHR(13) + CHR(10), ;
          This.cLogFile, not m.lStartNewLogFile)

RETURN

ENDDEFINE
```

Listing 6. A simple method call adds a message to the log.

```
goApp.oLogger.LogIt("Starting phase 1.")
```

Listing 7. The call in **Listing 6** produces this line in the log.

```
03/16/15 03:10:35 PM:Starting phase 1.
```

Of course, you can also save a log in a table. In the simplest case, the table might be simply a datetime field and a memo field.

Coverage Logging and the Coverage Profiler

VFP includes a pair of tools that collect and present data about what lines of code run and how long they take. The Coverage Logger records each line of code that executes, including how long it took. The Coverage Profiler processes a coverage log and presents the information visually. Like many of the tools that come with VFP, the Coverage Profiler has an add-in mechanism that lets you extend its behavior. In addition, not only is the log created by the Coverage Logger easy to parse and process with custom code, but you can substitute a different tool to process the log.

Creating Coverage logs

You can start the Coverage Logger from the Debugger or with code. Interactively, use Tools | Coverage Logging … from the Debugger menu or click the Toggle coverage logging button on the Debugger's toolbar. In either case, the Coverage dialog, shown in **Figure 1**, appears. Specify the name of a file to hold the log, and indicate whether to add to an existing log or start a new one. Once you click OK, each line of VFP code that executes is logged. To turn logging off interactively, use the toolbar button.

Figure 1. The Coverage dialog lets you turn Coverage Logging on.

You can also turn coverage logging on and off programmatically using the SET COVERAGE command. SET COVERAGE TO a file to turn coverage on; issue SET COVERAGE TO without a filename to turn it off. When turning coverage on, you can use the optional ADDITIVE keyword to add to an existing log. For example, the code in **Listing 8** turns on coverage logging, and sets it up to add to an existing log called Cover.Log.

Listing 8. You can start coverage logging programmatically with the SET COVERAGE command.

```
SET COVERAGE TO Cover.Log ADDITIVE
```

Using the Coverage Profiler

The Coverage Profiler takes a coverage log and displays it with the relevant code. The tool has two main modes: coverage mode and profile mode. Coverage mode lets you see which lines of code were executed and which were not, in other words, how much of the code was *covered* by the test.

For optimization, we're interested in profile mode, which shows each line of code, and indicates how many times it ran, how long it took the first time it was executed, and the average time for all executions of that line. **Figure 2** shows an example. The log was created by running the Solution Samples application that comes with VFP, and running several of the samples. (You'll find a similar coverage log in the session materials as SolCover.LOG.)

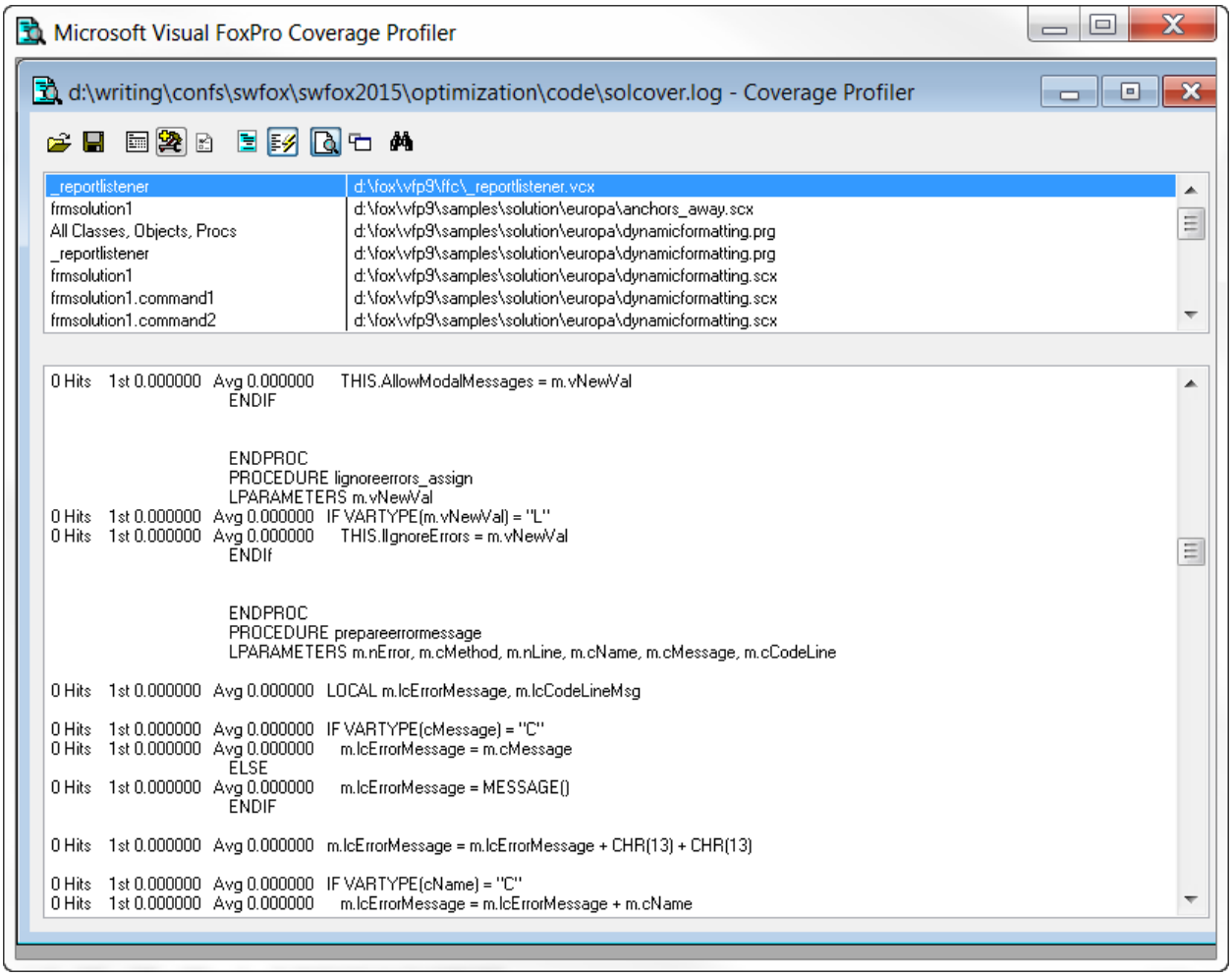

Figure 2. In Profile mode, the Coverage Profiler gives you information about the execution of each line.

The top frame shows the various sources of code in the log. Click on one and profile information for that block of code populates the bottom frame.

One thing that may not be obvious from the figure is that the times shown are rolled up. That is, if the line includes calls to other code, the time includes the time to execute that called code.

In my experience, while Profile Mode is somewhat helpful for eyeballing for slow code, it isn't really useful for serious optimization work. For that, you need either an add-in, to work directly with the coverage log, or to use an alternate profiling tool. All three approaches are discussed later in this section.

Profiling at runtime

In VFP 9, you can turn coverage on and off at runtime using the SET COVERAGE command. However, the coverage files produced at runtime can be hard to use with the Coverage Profiler because the paths shown for the files may not match their locations in your development environment. Rick Schummer created a tool to solve this problem; you give it

a coverage log and a little more information and it creates a copy of the coverage log with the paths fixed to match your environment.

With Rick's permission, the Coverage Log Path Fixer (originally distributed with the book "*What's New in Nine: Visual FoxPro's Latest Hits*") is included in the materials for this session. **Figure 3** shows the tool at work on the example log.

To use the tool, first point to the coverage log to be fixed. The tool automatically generates a name for the fixed version by appending "_fixed" to the file stem, but you can overwrite that, if you wish. Once you've pointed to the log to fix, the tool analyzes the log and produces a list of code folders it references. For each path shown in the first column of the bottom grid, you click the Select Directory button to point to the corresponding development environment folder. Once you've specified all the folders, click Fix Paths to generate the fixed log file.

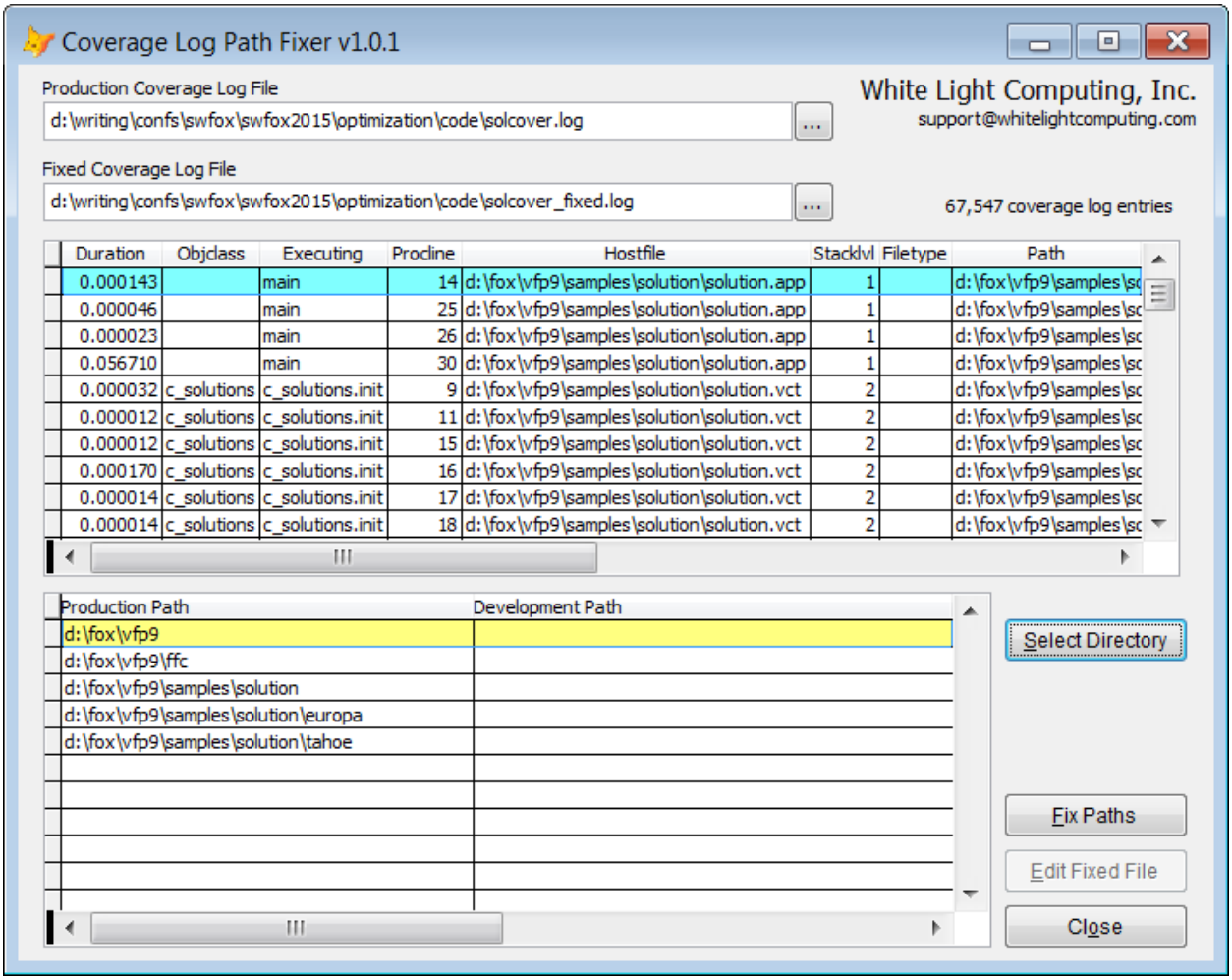

Figure 3. Rick Schummer's Coverage Log Path Fixer lets you fix the paths in a coverage log created in the runtime environment, so that you can use the Coverage Profiler on that file.

While I haven't tested this idea, I suspect this tool can also be used to adjust a coverage log created by a developer who has set up her development environment differently from

yours. (Of course, two people sharing code and using different directory structures will run into other problems.)

Coverage Profile add-ins

Like many of the tools that come with VFP, the Coverage Profiler has an open architecture. It includes a mechanism for hooking in additional tools, called "add-ins." (The VFP Help file contains some documentation for Coverage Profiler add-ins. This article by Lisa Slater Nicholls, who wrote the Coverage Profiler, provides a deeper look: http://spacefold.com/lisa/oldsite/LSN_CoverageExtend.ASPX.)

In this paper, we're not going to look at creating add-ins. Instead, we'll just look at some add-ins helpful for optimization. To run a Coverage Profiler add-in, you click the Add-Ins button on the Coverage Profiler toolbar; it's highlighted in **Figure 4**. The Add-ins dialog (see **Figure 5**) appears.

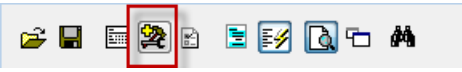

Figure 4. The Add-ins button lets you run Coverage Profiler add-ins.

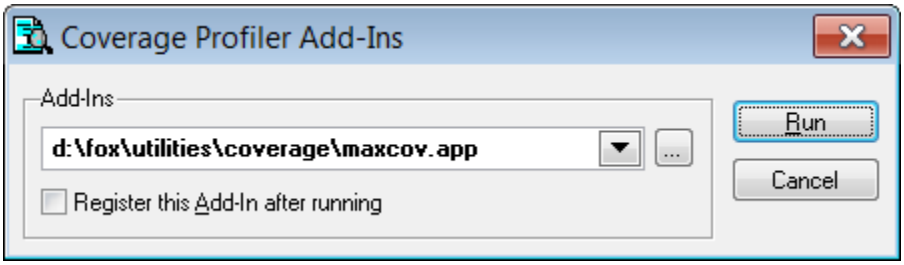

Figure 5. Use this dialog to specify an add-in to run and to register add-ins, so they're available in future without having to find them.

Markus Egger's MaxCov add-in offers several useful features; it's included in the materials for this session, with Markus's permission. It shows the total time spent on a line, which isn't available in the Coverage Profiler itself. It lets you choose a method to look at, or look at all methods; it also lets you choose to show only executed lines or only those lines that didn't execute. Most helpful for optimization is that it lets you highlight lines that are slow or that are used often. **Figure 6** shows MaxCov looking at the same log file as used for Figure 2.

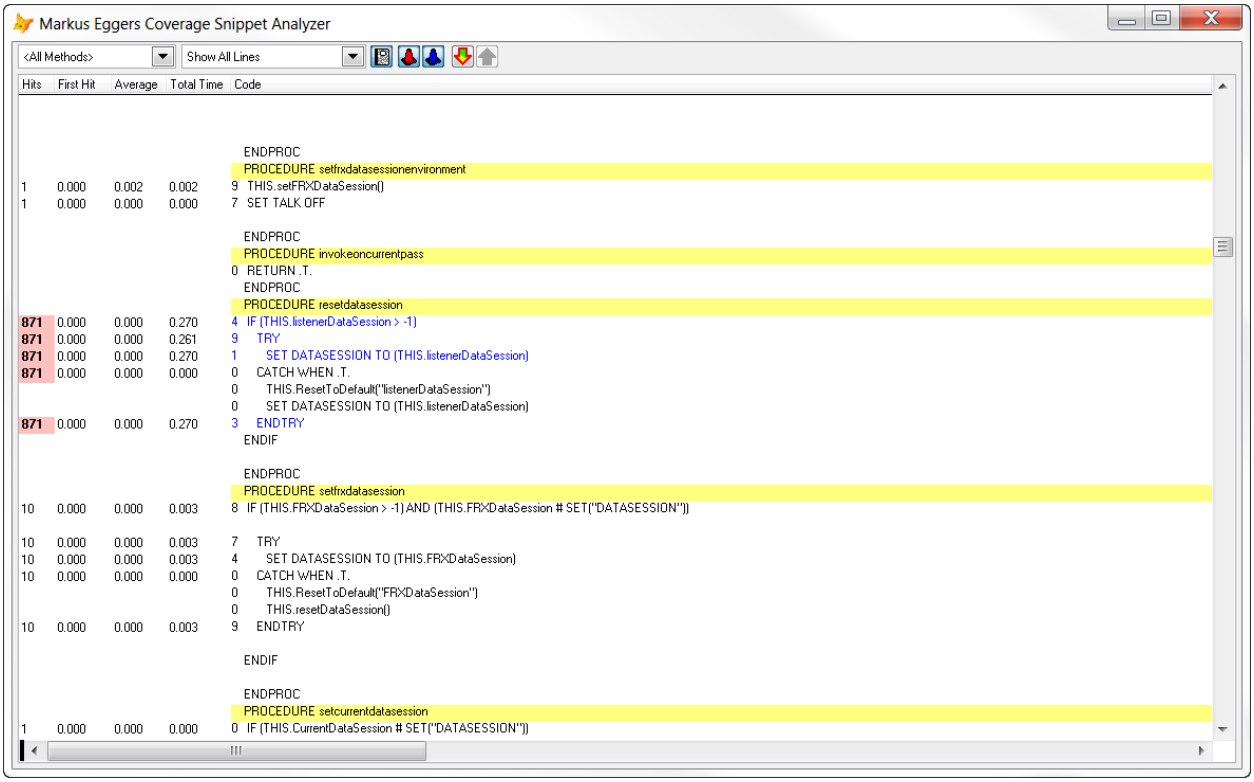

Figure 6. Markus Egger's Coverage Profiler add-in, MaxCov, makes it easier to find the slow parts in a coverage log.

The dropdowns in the toolbar let you choose a method and which lines to show. Method declaration lines (that is, PROCEDURE or FUNCTION lines) are highlighted in yellow to make it easy to see where one method stops and the next begins; the first button on the toolbar lets you toggle that feature.

The red down arrow button determines whether to highlight slow code; the blue down arrow determines whether to highlight both slow code and code that's executed many times. The last two buttons let you move quickly from one method to the next.

Although MaxCov offers more information and better ways to filter it than the Coverage Profiler itself, it needs some changes to be a tool I'd use often. Most important in my view is the ability to customize. I'd like to choose larger fonts, as well as be able to specify what I consider slow and how many hits make a line "frequently used."

The Solution Samples that come with VFP 9 include a Coverage Profiler add-in called "Coverage Profiler Performance Add-in." The file to point to with the Add-ins dialog is Samples\Solution\Coverage\covperfaddinfs.scx in your VFP installation.

Like MaxCov, this add-in shows the total time spent on a line. You can click any column header to sort by that column and click again to reverse the order. A toolbar button lets you restore the original order (though it's not clear what order that is). **Figure 7** shows this add-in as it opens with the same coverage log used for the other examples.

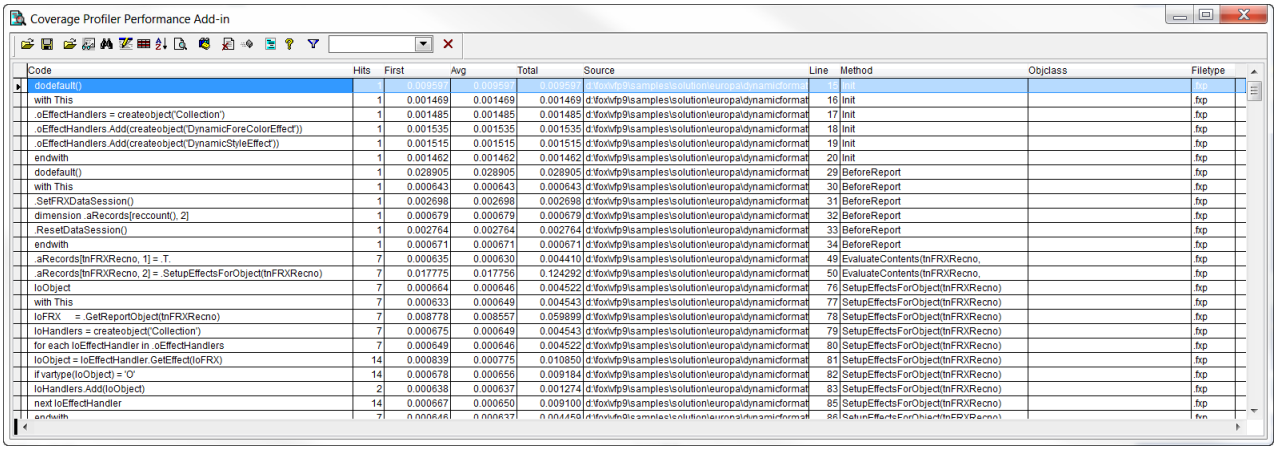

Figure 7. The Performance Add-in, found in the Solution Samples, lets you sort and filter the information from the log.

Beyond sorting the data, the Performance Add-in lets you search in any column, highlight slow lines, and filter data based on expressions you specify. An Options dialog lets you define the value that determines "slow," as well as which field (First, Avg, or Total) to base it on. Filters you specify are added to the dropdown in the toolbar (and remembered from one session to the next), so you can switch among them. **Figure 8** shows the add-in's form filtered so that only lines with a total time more than 0.1 seconds are shown.

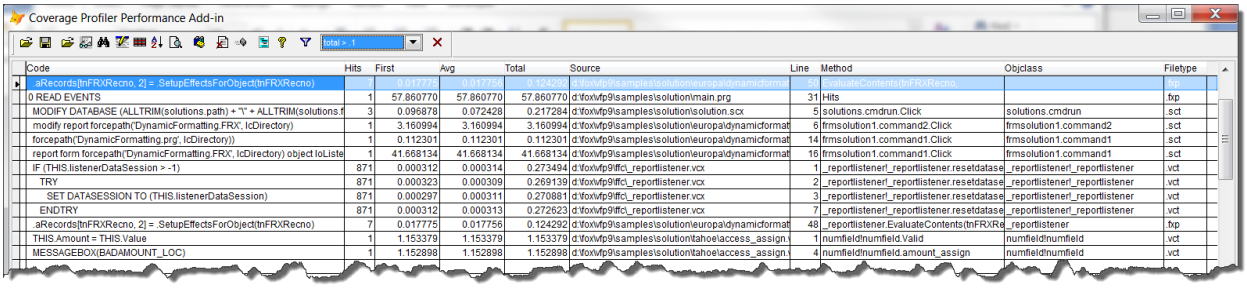

Figure 8. The Performance Add-in lets you filter coverage data based on expressions you specify.

This add-in also offers a couple of reports. The slow lines report shows just lines considered slow; your choices in the Options dialog determine which field it's based on and what percent of lines are included. **Figure 9** shows the report, using the default settings of the First field and 25%.

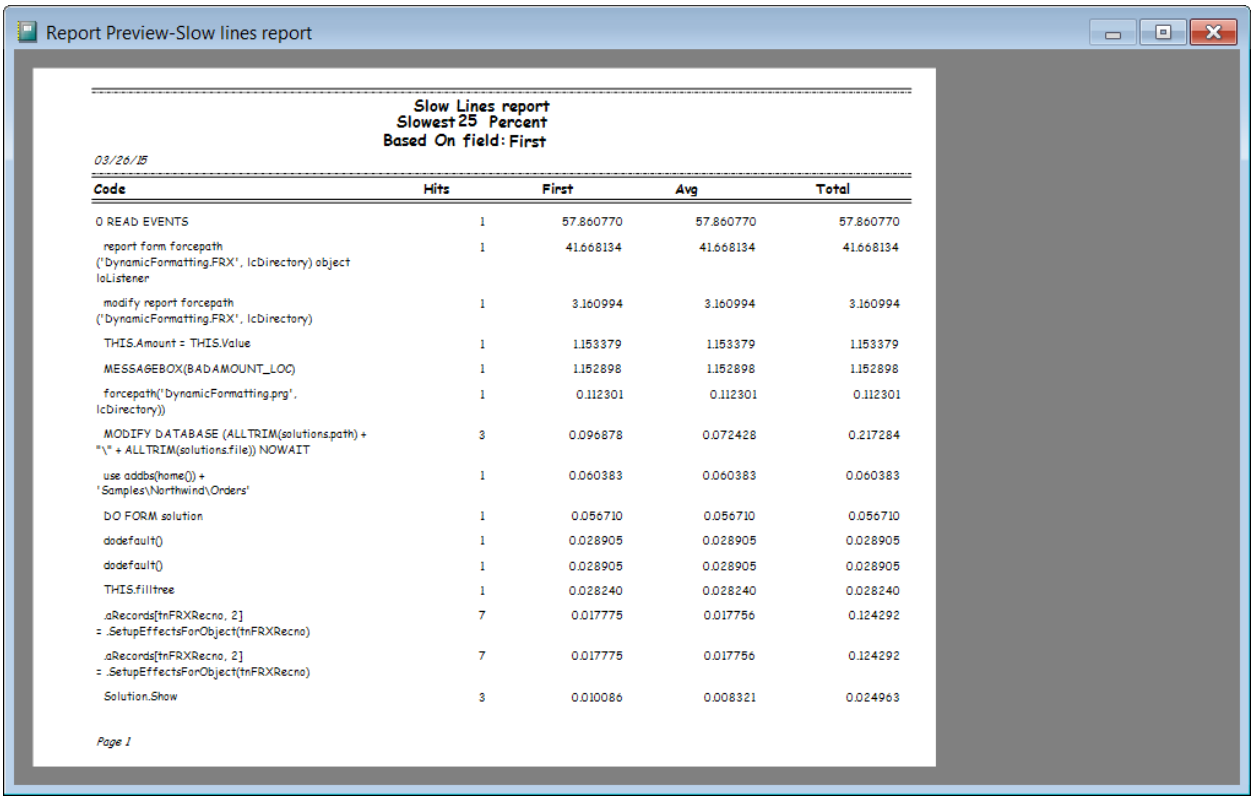

Figure 9. The Performance Add-in's Slow Lines report shows you the slowest lines in the log, using a percentage and field you specify.

Perhaps more useful is the Slow Methods report, which shows the slowest methods by total method time. The same percentage you specify in the Options dialog determines how many methods show. **Figure 10** shows the report with the percentage set to 40. Obviously, you want to choose this value based partly on the number of different methods in your log.

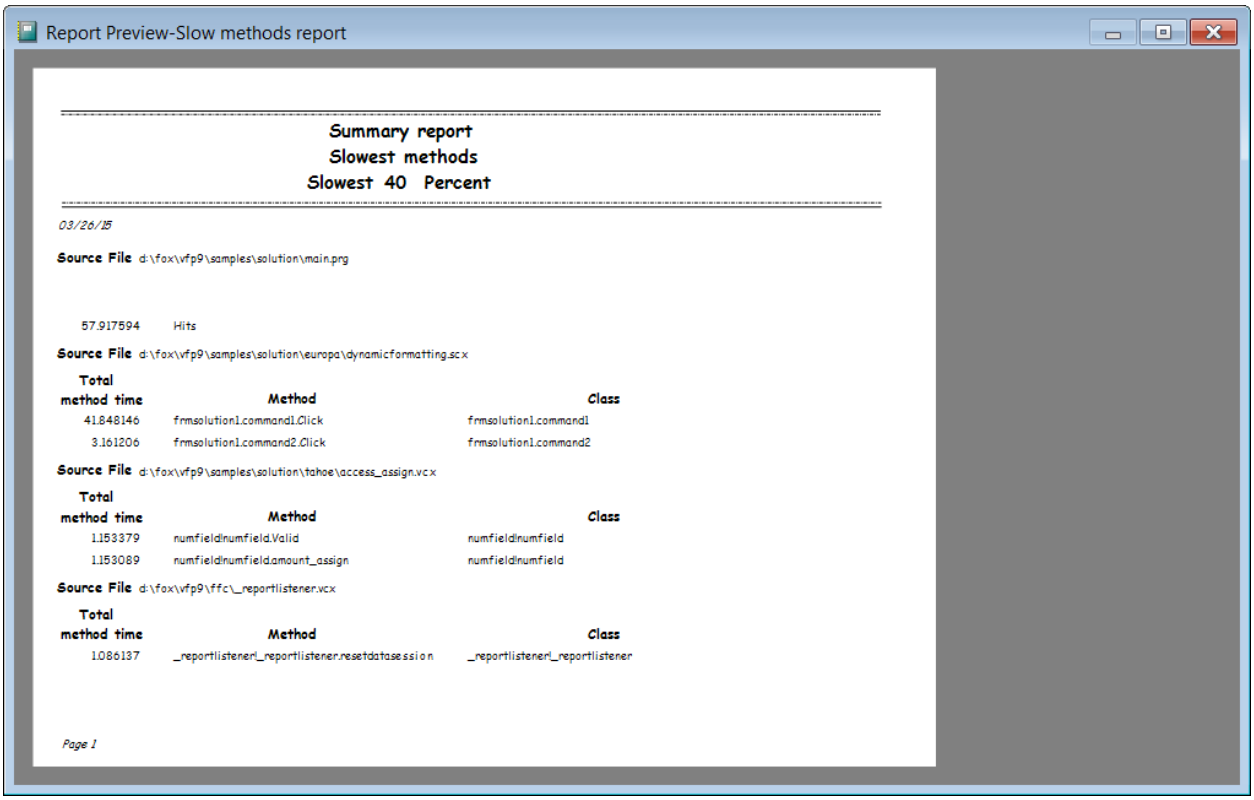

Figure 10. The Performance Add-in's Slow Methods report is based on total method time.

The source code for this add-in is in the folder, so you can change its behavior, if you wish.

I discovered this add-in while preparing this session, but wished I'd noticed it sooner, as it would have been quite useful when working with my client's application.

Exploring coverage logs

The logs created by the Coverage tool are simply comma-delimited files. That makes them easy to import into a table or cursor, after which you can do your own analysis of the data. **Listing 9** shows the first 26 lines from the example coverage log. **Table 1** lists the fields represented in the log.

Listing 9. A coverage log is a comma-delimited file.

```
0.000143,,main,14,d:\fox\vfp9\samples\solution\solution.app,1
0.000046,,main,25,d:\fox\vfp9\samples\solution\solution.app,1
0.000023,,main,26,d:\fox\vfp9\samples\solution\solution.app,1
0.056710,,main,30,d:\fox\vfp9\samples\solution\solution.app,1
0.000032,c_solutions,c_solutions.init,9,d:\fox\vfp9\samples\solution\solution.vct,2
0.000012,c_solutions,c_solutions.init,11,d:\fox\vfp9\samples\solution\solution.vct,2
0.000012,c_solutions,c_solutions.init,15,d:\fox\vfp9\samples\solution\solution.vct,2
0.000170,c_solutions,c_solutions.init,16,d:\fox\vfp9\samples\solution\solution.vct,2
0.000014,c_solutions,c_solutions.init,17,d:\fox\vfp9\samples\solution\solution.vct,2
0.000014,c_solutions,c_solutions.init,18,d:\fox\vfp9\samples\solution\solution.vct,2
0.000007,c_solutions,c_solutions.init,19,d:\fox\vfp9\samples\solution\solution.vct,2
0.000011,c_solutions,c_solutions.init,21,d:\fox\vfp9\samples\solution\solution.vct,2
```

```
0.000008,c_solutions,c_solutions.init,22,d:\fox\vfp9\samples\solution\solution.vct,2
0.000125,c_solutions,c_solutions.init,23,d:\fox\vfp9\samples\solution\solution.vct,2
0.000014,c_solutions,c_solutions.getdirectory,4,d:\fox\vfp9\samples\solution\solution
.vct,3
0.000010,c_solutions,c_solutions.getdirectory,5,d:\fox\vfp9\samples\solution\solution
.vct,3
0.000009,c_solutions,c_solutions.getdirectory,6,d:\fox\vfp9\samples\solution\solution
.vct,3
0.000008,c_solutions,c_solutions.getdirectory,7,d:\fox\vfp9\samples\solution\solution
.vct,3
0.000009,c_solutions,c_solutions.getdirectory,10,d:\fox\vfp9\samples\solution\solutio
n.vct,3
0.000006,c_solutions,c_solutions.getdirectory,13,d:\fox\vfp9\samples\solution\solutio
n.vct,3
0.000011,c_solutions,c_solutions.getdirectory,15,d:\fox\vfp9\samples\solution\solutio
n.vct,3
0.000009,c_solutions,c_solutions.init,25,d:\fox\vfp9\samples\solution\solution.vct,2
0.000007,c_solutions,c_solutions.init,26,d:\fox\vfp9\samples\solution\solution.vct,2
0.000007,c_solutions,c_solutions.init,27,d:\fox\vfp9\samples\solution\solution.vct,2
0.000109,c_solutions,c_solutions.init,31,d:\fox\vfp9\samples\solution\solution.vct,2
```

Table 1. Coverage logs contain these fields in this order. Note that the actual code is not included.

Field	Description
1	Execution time of the line, in seconds.
2	Class containing the executed line.
3	Object, method or procedure containing the executed line.
4	Line number within the method or procedure.
5	Fully-qualified name of the file containing the executed line.
6	Call stack level of the executed line.

To import the data, you simply need to create a table or cursor with the right fields, and use APPEND FROM, as in **Listing 10**. Once the data is in a cursor, you can use VFP's various data-handling tools on it. What I do most often is run queries to identify methods or individual lines of code that deserve my attention.

Listing 10. Importing coverage data to a cursor is easy.

```
CREATE CURSOR cov ;
   ( nTime N(12, 6), ;
     cClass c(30), ;
     cObj c(60), ;
     nLine i, ;
     cFile c(60), ;
     cStack i )

APPEND FROM GETFILE() TYPE DELIMITED
```

For example, the query in **Listing 11** finds the 100 lines that were executed most often. The query in **Listing 12** computes total time per method and sorts it into descending order; I use it to identify methods I should look at first when optimizing.

Listing 11. Run this query against the imported coverage log to find the lines executed most often.

```
SELECT COUNT(*) AS nCnt, cClass, cObj, nLine, cfile ;
   FROM cov ;
   GROUP BY cClass, cObj, nLine, cfile ;
   ORDER BY nCnt DESCENDING ;
   TOP 100 ;
   INTO CURSOR csrMostTimes
```

Listing 12. This query of the coverage data computes the total time spent in each method and sorts it with the longest first.

```
SELECT SUM(nTime) AS nTotalTime, cClass, cObj, cfile ;
   FROM cov ;
   GROUP BY cClass, cObj, cfile ;
   ORDER BY nTotalTime DESC ;
   INTO CURSOR csrMethodTime
```

ExploreLog.PRG, included in the materials for this session, includes the code to import the log, the two queries shown above, and a few other queries I've found useful.

An alternative profiler

One of the biggest benefits of the separation of creation of coverage logs from analysis of the logs is that you can actually substitute a totally different tool to process a coverage log. I'm aware of only one such tool. Created by Martina Jindrová, it's called CVP and it's available at http://gorila.netlab.cz/cvp.html.

To install, you download this tool and unzip it into a folder. Note that CVP comes with localization for Czech, English, French, Russian and Slovak. To specify one of those languages, double-click the appropriate .REG file. (I skipped this step and CVP defaulted to English for me. I don't know whether that's an overall default, or if it looked at the language for my Windows installation.)

To use CVP, just run CVP.EXE. When it opens, all you have is an empty window, but unlike most free utilities, this one comes with a Help file that's integrated into the tool. (Note that the actual Help content is partly in English and partly in what I think is Czech. In addition, in my environment, the Help topics didn't scroll properly, so I couldn't see anything below whatever fit on my screen.)

You can actually substitute this tool for the built-in coverage profiler by setting the system variable _Coverage to point to CVP.EXE. Then, when you choose Coverage Profiler from the menu, CVP runs. In my tests, however, when I did that, I couldn't switch back and forth between CVP and the VFP window.

To begin working with a coverage log, choose File | New from the menu and point to the log file. The log file is opened and analyzed and you're prompted to point to any projects it can't find (as in **Figure 11**, which shows the list of such projects for our example log); this allows you to deal with any path differences.

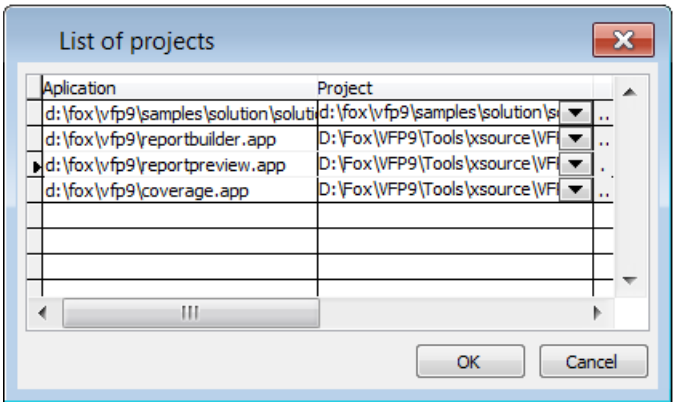

Figure 11. Use this dialog to deal with path differences between the installation that created the log and the development environment in which you're running CVP.

If there are still individual items for which source can't be found, the list of projects is followed by a list of modules that need to be located, as shown in **Figure 12**.

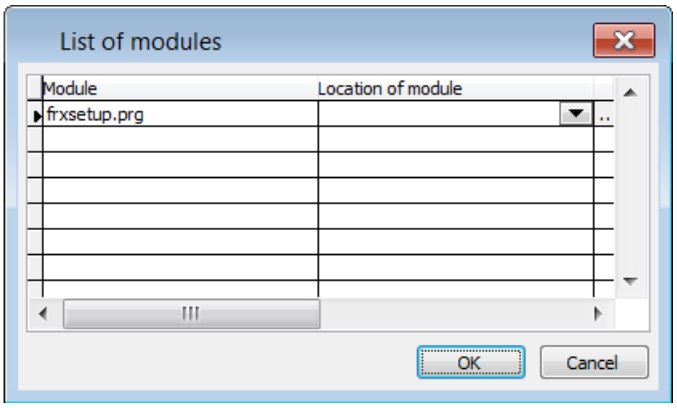

Figure 12. This dialog lets you locate individual items that CVP can't find.

Once you've given CVP as many hints to where things are located as you can, the main analysis window opens, as in **Figure 13**. (I found it makes sense to enlarge that window somewhat, as several columns of the main grid are hidden in this figure. Additional screenshots will show the enlarged window.) On the Analysis tab, indicate what code you're interested in, based on how long it took to run or how often it ran. You can specify thresholds for both individual lines and for entire methods and procedures. Once you've made your choices, click the analyze button. **Figure 14** shows the Analysis tab after specifying methods and procedures that took more than 1 second or executed more than 100 times.

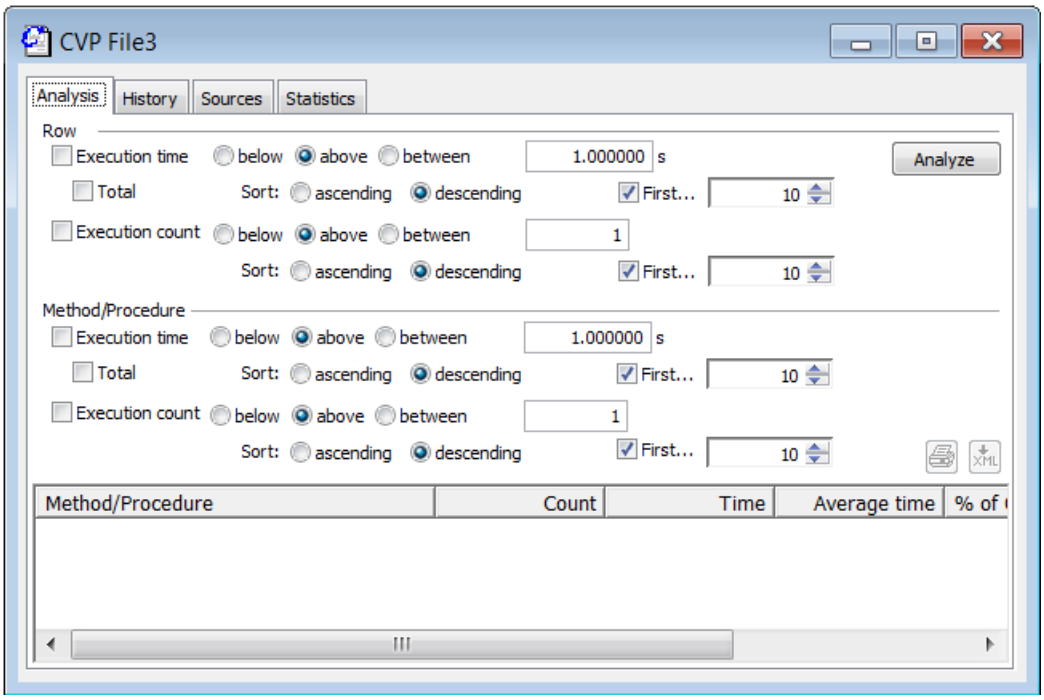

Figure 13. This is the main analysis window for CVP. You use it to indicate what code you're interested in.

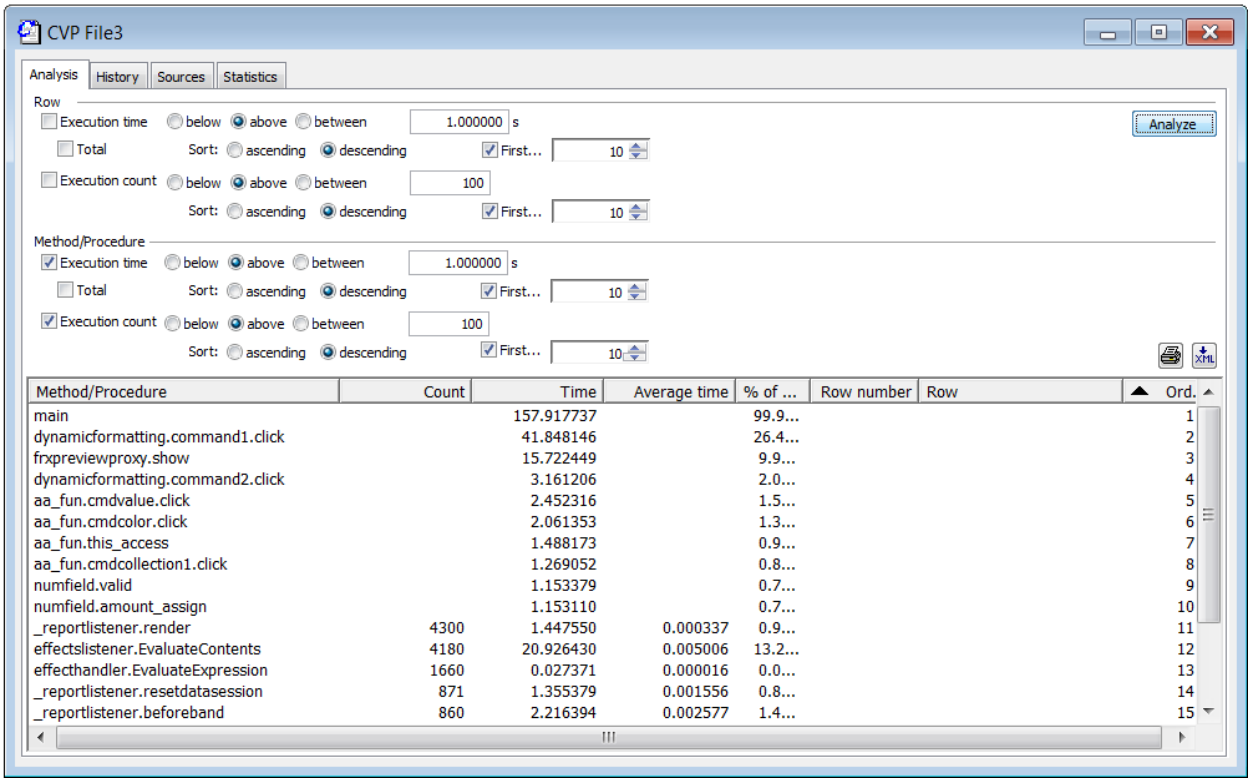

Figure 14. The Analysis tab of CVP lets you specify which rows and/or methods and procedures you want to look at.

The History tab (see **Figure 15**) shows the code from the Call Stack point of view. The left pane is a treeview of the call stack. Click on an item there to show the relevant code in the right pane. A yellow background means that the line calls another routine and you can double-click to switch to that routine. If there's more than one call to the routine, a dialog appears to let you indicate which call you're interested in.

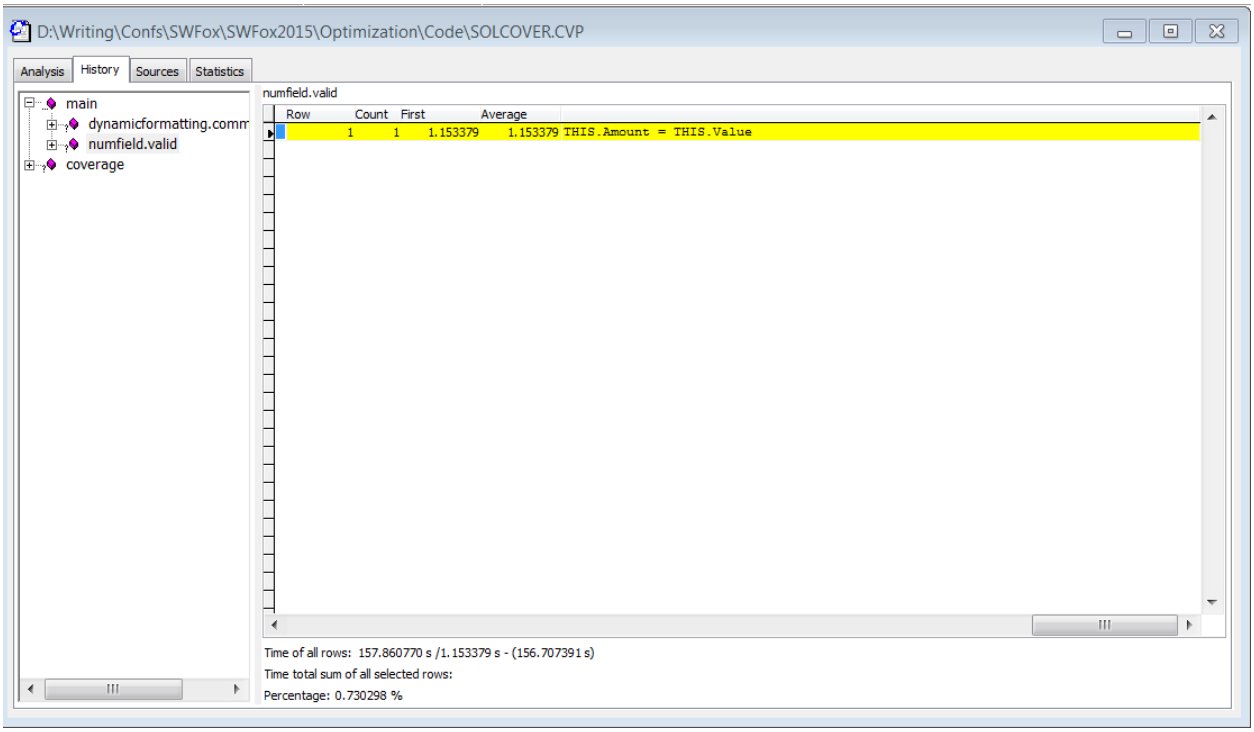

Figure 15. The History tab is organized based on the Call Stack. The right pane shows the actual data from the log file for the selected routine.

The Sources tab (**Figure 16**) is organized based on source code files. The left pane is a treeview of code files used in the log. The right pane shows the code for the selected item. The dropdown and spinner at the top of the right pane lets you indicate whether to show data for all calls to the routine, or for a specific call.

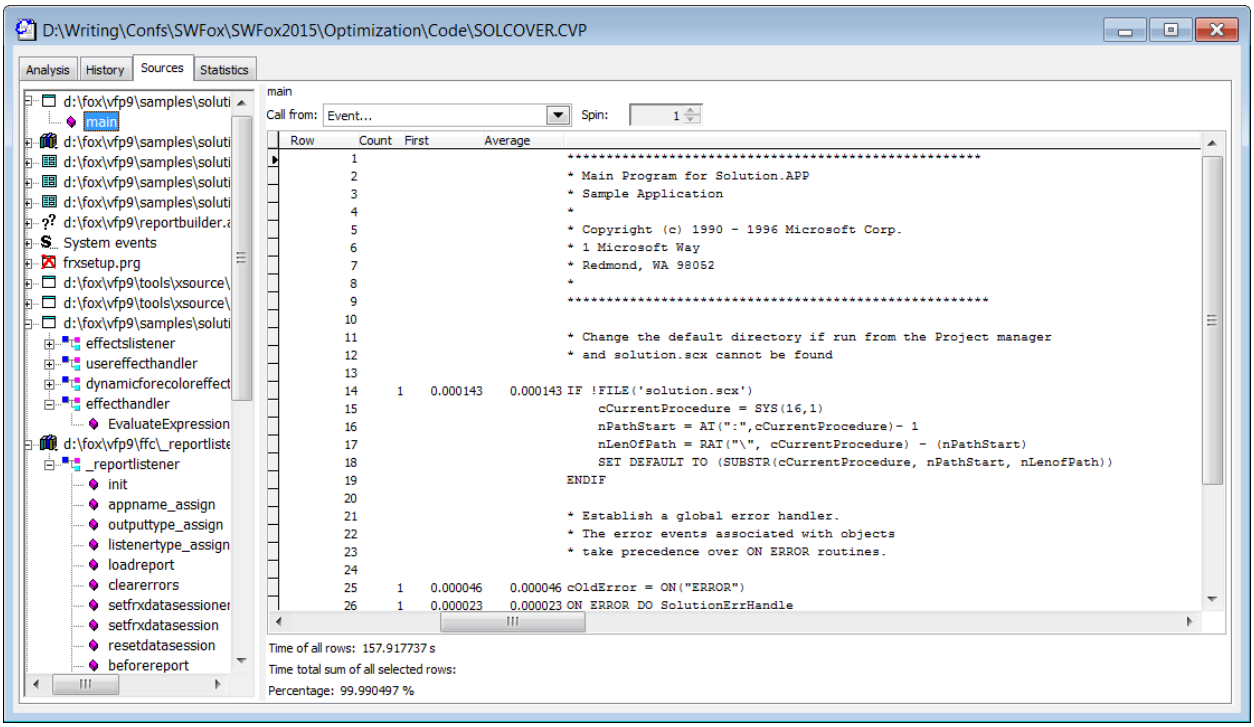

Figure 16. The Sources tab organizes the code based on its location.

The Statistics tab (**Figure 17**) lets you generate some overall statistics for your log. To do so, click the Read button. Among other things, it shows the line and the routine executed most often, and the line and the routine that took the most time; you can click the arrow next to any of those to jump right to the relevant code in the Sources window.

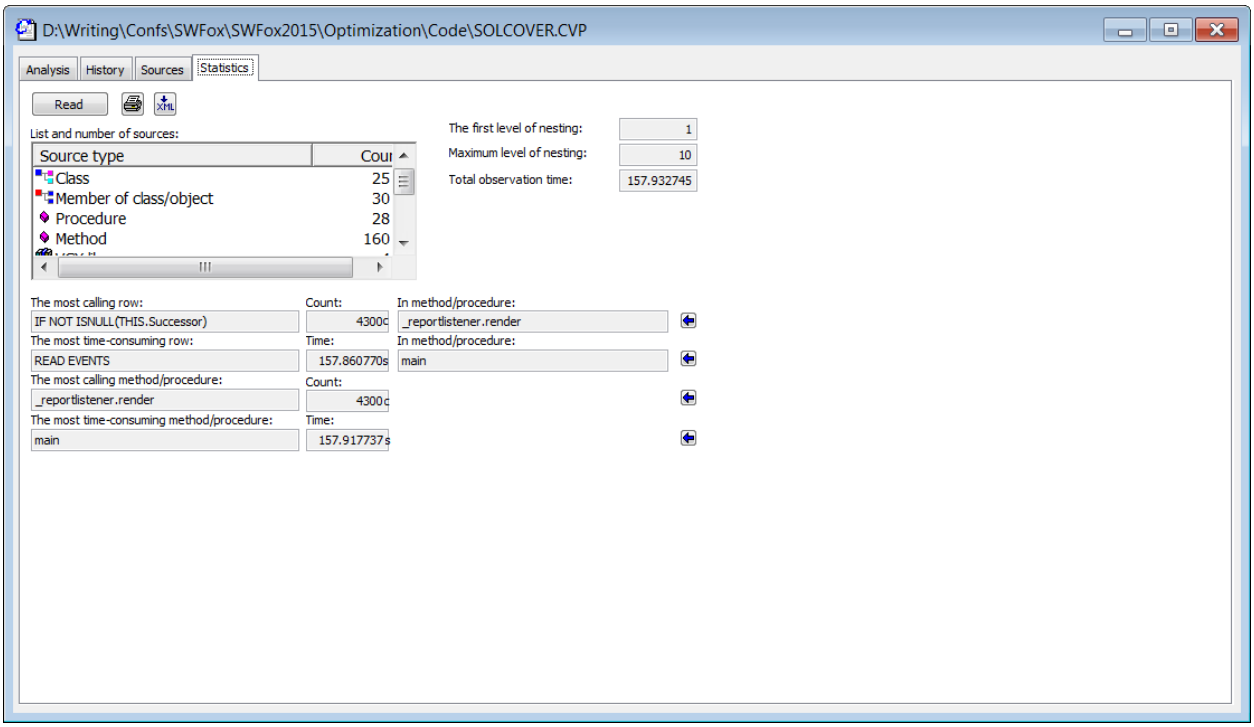

Figure 17. The Statistics page gives you basic information about your coverage log.

VFP's native Coverage Profile takes a long time to open a large log file. CVP allows you to store its initial conversion of the log to a table (though it's not the same table you'd get using APPEND FROM as described in "Exploring coverage logs," earlier in this document), so that when you want to work with the same file again, the initial processing can be skipped. The file has a CVP extension; you use File | Open to read it in when you re-open CVP.

I haven't worked much with CVP, but I certainly can see situations in which it would be useful. There are also changes I'd like to see, especially the ability to set font sizes, as many of them are small for my aging eyes. Despite that, I'm glad to have another tool in my optimization toolbox.

Checking query optimization

When you talk about optimizing Visual FoxPro code, one of the first things people tend to mention is Rushmore, the technology that makes table and cursor operations so fast. While there's no way to directly check the use of Rushmore on Xbase commands, VFP includes a pair of functions that let you see how SQL SELECT, UPDATE and DELETE commands are being optimized. SYS(3054) collects optimization information and SYS(3092) lets you send that information to a file.

SYS(3054), also known as *SQL ShowPlan*, does the heavy lifting here. Its syntax is shown in **Listing 13**. The key parameter here is nSetting; **Table 2** shows the accepted values. Omitting the nSetting parameter returns the current setting (as a string).

Listing 13. SYS(3054) returns information about how Rushmore is optimizing SQL commands.

```
cSetting = SYS(3054 [, nSetting [, cOutputVar ]])
```

Table 2. The value you specify for nSetting determines the output SYS(3054) produces.

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 original SQL 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 original SQL command in the output.

Basically, there are two pairs of choices: what to test and whether to include the original command. Other than for demos, I've never found a reason to look at optimization only for filters; I always use filters and joins. Similarly, I almost always include the SQL command in the output. So my most frequent choice for nSetting is 12.

The cOutputVar parameter is a little tricky (and, with the addition of SYS(3092) in VFP 9, almost obsolete). By default, SQL ShowPlan results are sent to the active window. When you pass the cOutputVar parameter, the results are stored only in the specified variable, and not sent to the active window.

The tricky part is that you pass the *name* of an existing variable into which the SQL ShowPlan output is stored, not the variable itself. **Listing 14** shows the wrong way and the right way to pass this parameter. (The reason it works this way is that VFP's built-in functions can't accept parameters by reference, only by value.)

Listing 14. You pass the name of the variable in which to store results to SYS(3054), not the variable itself.

```
LOCAL c3054Output

* This doesn't work.
SYS(3054, 12, c3054Output)

* This works
SYS(3054, 12, "c3054Output")
```

Listing 15 shows SYS(3054) at work; it's included in the session materials as Simple3054.prg. **Figure 18** shows the output that's stored in c3054Output (somewhat reformatted to fit the page). See the next section, "Understanding SQL ShowPlan results," to learn what the output tells you. It's worth noting that SYS(3054) also shows results from USEing a local view. However, in that case, it doesn't show the query itself, even if you pass an appropriate value for nSetting. (In fact, the query in **Listing 15** is a reformatted version of the query that defines the Northwind view product_sales_for_1997. An alternate version of the code that opens the view rather than just running the query is included in the session materials as View3054.prg.)

Listing 15. Wrap the query you want to test with calls to SYS(3054).

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

LOCAL c3054Output

SYS(3054, 12, "c3054Output")
SELECT Categories.categoryname, Products.productname,;
       SUM((Orderdetails.unitprice*Orderdetails.quantity* ;
           (1-Orderdetails.discount)/100)*100) AS productsales;
 FROM Categories ;
   JOIN Products ;
     ON  Categories.categoryid = Products.categoryid ;
   JOIN  Orders ;
     JOIN OrderDetails ;
       ON Orders.orderid = Orderdetails.orderid ;
     ON Products.productid = Orderdetails.productid;
 WHERE Orders.shippeddate BETWEEN {^1997/01/01} AND {^1997/12/31};
 GROUP BY Categories.categoryname, Products.productname ;
 INTO CURSOR csrSales1997

SYS(3054, 0)
```

```
SELECT Categories.categoryname, Products.productname,
SUM((Orderdetails.unitprice*Orderdetails.quantity*(1-Orderdetails.discount)/100)*100) AS productsales
FROM Categories JOIN Products ON  Categories.categoryid = Products.categoryid JOIN  Orders
JOIN OrderDetails ON Orders.orderid = Orderdetails.orderid ON Products.productid = Orderdetails.productid
WHERE Orders.shippeddate BETWEEN {^1997/01/01} AND {^1997/12/31}
GROUP BY Categories.categoryname, Products.productname INTO CURSOR csrSales1997
Rushmore optimization level for table categories: none
Rushmore optimization level for table products: none
Using index tag Shippeddat to rushmore optimize table orders
Rushmore optimization level for table orders: partial
Rushmore optimization level for table orderdetails: none
Joining table categories and table products using index tag Categoryid
Joining table orders and table orderdetails using index tag Orderid
Joining intermediate result and intermediate result using temp index
```

Figure 18. SYS(3054) output shows the query first (if you pass 2 or 12 for nSetting) and then the optimization results.

Beyond the unusual way of passing it, the cOutputVar parameter has a major limitation. It stores the SQL ShowPlan only for the most recent SQL command. That is, if you turn SQL ShowPlan on and then run code that includes multiple SQL commands, when you check the specified variable, it contains the output only for the last of those commands.

The SYS(3092) function was added in VFP 9 to provide a better solution. It lets you specify a file in which to store SQL ShowPlan results. The syntax for SYS(3092) is shown in **Listing 16**. The second parameter, cFileName, is the path and name of the file in which to store the results. Pass the empty string for cFileName to stop sending results to the specified file.

Listing 16. SYS(3092) lets you send SQL ShowPlan output to a file.

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

Listing 17 shows the same query as the previous example, but this time, the results are sent to a file called Optimization.txt. The file contents (identical to the previous output) are shown in **Listing 18**. Obviously, this approach is much more useful for testing optimization inside an application than sending the output to the active window or a variable. Be aware that, when using SYS(3092), you'll normally want to pass a variable to SYS(3054) anyway, as in the example; otherwise, the output is sent to both the file and the active window.

Listing 17. Put calls to SYS(3092) around the SYS(3054) calls to send SQL ShowPlan output to a file.

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

LOCAL c3054Output

SYS(3092, "Optimization.txt")
SYS(3054, 12, "c3054Output")
SELECT Categories.categoryname, Products.productname,;
       SUM((Orderdetails.unitprice*Orderdetails.quantity*
           (1- Orderdetails.discount)/100)*100) AS productsales;
 FROM Categories ;
   JOIN Products ;
     ON  Categories.categoryid = Products.categoryid ;
   JOIN  Orders ;
     JOIN OrderDetails ;
       ON Orders.orderid = Orderdetails.orderid ;
     ON Products.productid = Orderdetails.productid;
 WHERE Orders.shippeddate BETWEEN {^1997/01/01} AND {^1997/12/31};
 GROUP BY Categories.categoryname, Products.productname ;
 INTO CURSOR csrSales1997

SYS(3054, 0)
SYS(3092, "")
```

Listing 18. Wherever you send SQL ShowPlan output, you get the same results.

```
SELECT Categories.categoryname, Products.productname,
SUM((Orderdetails.unitprice*Orderdetails.quantity*(1-Orderdetails.discount)/100)*100)
AS productsales FROM Categories JOIN Products ON  Categories.categoryid =
Products.categoryid JOIN  Orders JOIN OrderDetails ON Orders.orderid =
Orderdetails.orderid ON Products.productid = Orderdetails.productid WHERE
Orders.shippeddate BETWEEN {^1997/01/01} AND {^1997/12/31} GROUP BY
Categories.categoryname, Products.productname INTO CURSOR csrSales1997
Rushmore optimization level for table categories: none
Rushmore optimization level for table products: none
Using index tag Shippeddat to rushmore optimize table orders
Rushmore optimization level for table orders: partial
Rushmore optimization level for table orderdetails: none
Joining table categories and table products using index tag Categoryid
Joining table orders and table orderdetails using index tag Orderid
Joining intermediate result and intermediate result using temp index
```

Understanding SQL ShowPlan results

The information provided by SYS(3054) tells you what order operations were performed in (the order shown in the result) as well as which index tags were used to optimize those

operations. As the parameters to the function indicate, you can divide the information into two groups, that related to filters and that related to joins.

Two kinds of lines in the output represent filters. Each line that begins "Rushmore optimization level" gives you an overall result for filtering of the specified table. The result can be "none," "partial" or "full." A result of "none" can indicate either that the table wasn't filtered in the query or that there were no appropriate tags to use in filtering that table. In **Listing 18**, the result is "none" for Categories, Products and OrderDetails because none of them is filtered (that is, no fields from those tables appear in the Where clause).

For each tag used to filter a table, the output includes a line that begins "Using index tag" and then indicates which tag and which table. In the example, there's only one such line, indicating that the Shippeddat tag of Orders was used for optimization. That's not surprising, since the query's WHERE clause filters on the ShippedDate field.

What may be surprising is that the summary line for the Orders tables says "partial" rather than "full." After all, the only filter for Orders is on ShippedDate and there's a tag for that. But what you can't see here is that I ran the query with SET DELETED ON; that adds an implied filter of NOT DELETED() for every table. Since there's no index tag on DELETED(), Orders is only partially optimized. If I SET DELETED OFF and run the example again, that section of the output changes to **Listing 19**. The question of whether to add a filter on DELETED() for each table is complex; it's discussed later in this paper in the section "The DELETED() dilemma and binary tags."

Listing 19. Issuing SET DELETED OFF removes an implied filter of NOT DELETED() for every table in a query, allowing the query to be fully optimized.

```
Using index tag Shippeddat to rushmore optimize table orders
Rushmore optimization level for table orders: full
```

Of course, you can have more than one optimizable filter for a table. With DELETED OFF, the query in **Listing 20** produces the SQL ShowPlan results in **Listing 21**.

Listing 20. This query has multiple filters on the Orders table.

```
SELECT CompanyName, Orders.OrderID, OrderDate, Quantity, Productname ;
   FROM Customers ;
     JOIN Orders ;
       ON Customers.CustomerID = Orders.CustomerID ;
     JOIN OrderDetails ;
       ON Orders.OrderID = OrderDetails.OrderID ;
     JOIN Products ;
       ON OrderDetails.ProductID = Products.ProductID ;
   WHERE OrderDate between DATE(1998, 4, 1) AND DATE(1998, 6, 30) ;
     AND ShippedDate between DATE(1998, 4, 1) AND DATE(1998, 6, 30) ;
     AND UPPER(Customers.City) IN ("LONDON", "MADRID", "PARIS") ;
   INTO CURSOR csrResult
```

Listing 21. When there are multiple filters on a given table, there may be more than one optimization of that table. Here, both filters on Orders can be optimized.

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

Not surprisingly, the remaining lines in SQL ShowPlan output, the ones beginning with "Joining" report the optimization of joins. The tag mentioned always belongs to the table listed second in that line so, for example, the line from **Listing 18** that reads:

```
Joining table orders and table orderdetails using index tag Orderid
```

indicates that the Orderid tag of OrderDetails was used to join that table with Orders. In general, if possible, Rushmore uses a tag from the larger table. (See the next major section of this paper, "The Rushmore engine," to understand why.)

The SQL ShowPlan output also shows you the order in which joins are performed. In the example from **Listing 18**, Categories and Products are joined. Then, Orders and OrderDetails are joined. Finally, the results of those two joins are joined. In this case, that's pretty much the order specified in the query. However, it's not at all unusual for Rushmore to choose to perform joins in a different order than specified. For example, the join section shown in **Listing 21** shows that Customers and Orders are joined first, which is the first join in the query as written. But then Products and OrderDetails are joined, and then the two intermediate results are joined. In each case, the initial joins involve joining a large table with a small one, so the index from the larger table is used.

Sometimes, the order changes, even without using two separate intermediate tables. That's the case for the query in **Listing 22**; the join portion of the SQL ShowPlan output is shown in **Listing 23**. Since Customers, Employees and Shippers are all quite small, it's hard to guess why Rushmore chose to join Employees to Orders first. However, it's not at all surprising that it chose to build a temporary index on that result when joining with the three-record Shippers table.

Listing 22. The order you specify for joins isn't always the order in which they're performed.

```
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
```

Listing 23. This SQL ShowPlan output for the query in Listing 22 shows that Rushmore chooses what it considers the best order to join tables, not necessarily the order you specify.

```
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
```

The Rushmore engine

One of the keys to optimizing VFP code is understanding Rushmore, the optimization approach used by the database engine. Rushmore applies to any Xbase command that has a FOR condition, and to the SQL SELECT, UPDATE and DELETE commands.

Rushmore works by using index tags to decide which records to include in a result. For each *optimizable condition* (defined in the next section "Having the right tags"), it creates a bitmap showing which records meet the condition, and then it combines all those bitmaps to determine which records actually have to be read. Any non-optimizable conditions are then checked against each remaining record.

Having the right tags

Because Rushmore uses index tags, the first key is having the right tags. For Rushmore to be applied, you need an index tag that exactly matches an expression. That expression can appear on either side of the comparison involved (though at some point in FoxPro history, the optimizable expression had to be on the left-hand side of the comparison). The key point is that the expression and the index key must be exactly the same. So, for example, if you have a tag on UPPER(Company), your comparison must use UPPER(Company), not Company for that tag to be used for optimization. For a tag on a more complex expression, like UPPER(cLast + cFirst), to be used, the comparison must include the whole expression.

Rushmore cannot use filtered indexes (that is, indexes created with the FOR clause) nor can it use tags that include the NOT operator.

The DELETED() dilemma and binary tags

It used to be an article of faith that, if your application ran with DELETED ON, every table should have a tag whose key was the DELETED() function. The practice first became common soon after Rushmore was added, when people noticed that doing so could speed up some very slow commands. However, later experiences demonstrated that in some situations, an index on DELETED() actually slowed things down. Here's why.

In general, the portion of an index file that has to be read to find out which records match a condition is much smaller than the actual data, so looking at tags reduces the amount of data that has to be pulled into memory.

But a tag on DELETED() has only two possible values, and in most situations, the values are not evenly distributed; there are far fewer deleted records. So the portion of the index file that represents the NOT DELETED() case can be quite large, and pulling it into memory can be slow.

Thus, for large tables in a networked situation, a tag on DELETED() is more likely to slow things down than speed them up. Exactly what "large" means in this context depends on a number of factors, including available memory, network speed, and so forth.

An additional factor was added in VFP 9 with the ability to create binary index tags. These tags can be used only for index expressions that have exactly two possible values; they're compressed considerably. In one test, I created a table with 1,000,000 records. Creating a regular index tag on DELETED() resulted in a file of 3,205,632 bytes. A binary index tag on DELETED() for that table was 135,168 bytes. Clearly, the use of a binary tag changes the cutoff between cases where indexing on DELETED() makes sense and those where it doesn't. (Be aware that binary tags can be used only for optimization, not for searching.)

Making it faster in the first place

In general, it's good advice not to worry too much about optimization until you get the code working and see which are the slow parts. However, there are some best practices that will make your code faster in the first place. So before we look at the process I used to speed up my client's application, let's examine a few of them.

Extra code in loops

One easy way to slow your application down is having code inside a loop that can be put outside the loop. For example, if you need to set a variable to be used inside the loop, but its value doesn't change in the loop, you should set it before the loop. Similarly, if you need to look something up and the look-up doesn't change on each pass through the loop, do it before the loop.

Listing 24 shows a program that demonstrates; it's included in the materials for this session as MinimizeLoops2.PRG. Here, the loop needs today's date for a calculation. The first version simply calls DATE() inside the loop. The second stores the value in a variable and uses that in the loop. In my tests looping through the Employee table to compute each age in each pass, I was able to get through more than 70% more passes by storing the date to a variable.

Listing 24. One way to speed things up is to make sure you're not doing anything repeatedly inside a loop that you could do outside the loop.

```
* Compare speed with call to DATE() inside loop
* and outside
```

```
#DEFINE SECONDSTORUN 5
#DEFINE LOOPPASSES 5000

LOCAL nInLoopStart, nInLoopEnd, nOutLoopStart, nOutLoopEnd
LOCAL nInPasses, nOutPasses
LOCAL nAge, dToday

OPEN DATABASE HOME(2) + "northwind\northwind"
USE Employees

nInLoopStart = SECONDS()
nInLoopEnd = m.nInLoopStart + SECONDSTORUN
nInPasses = 0

DO WHILE m.nInLoopEnd > SECONDS()
   * Now the loop that we're testing
   nInPasses = m.nInPasses + 1

   SCAN
      nAge = DATE() - Employees.Birthdate
   ENDSCAN

ENDDO

nOutLoopStart = SECONDS()
nOutLoopEnd = m.nOutLoopStart + SECONDSTORUN
nOutPasses = 0

DO WHILE m.nOutLoopEnd > SECONDS()
   * Now the loop that we're testing
   nOutPasses = m.nOutPasses + 1

   dToday = DATE()
   SCAN
      nAge = m.dToday - Employees.Birthdate
   ENDSCAN

ENDDO

DEBUG
DEBUGOUT "With DATE() inside loop, " + TRANSFORM(m.nInPasses) + " passes."
DEBUGOUT "With DATE() outside loop, " + TRANSFORM(m.nOutPasses) + " passes."
```

The materials for this session include another example (MinimizeLoops.PRG) where a calculation inside a loop needs the value of a field. While the speed improvement isn't as large as in the example above, it, too, benefits from grabbing the value once and using a variable instead.

Similarly, when loops are nested, be sure to put each line of code at the outermost level you can.

Use MDots

Perhaps the most controversial best practice in VFP coding is preceding any reference to a variable with "m." (known as "MDot"). It's necessary because when a variable and a field in the current table have the same name, VFP assumes you mean the field (except in situations where a field can't be used, such as the left-hand side of an assignment statement). However, many VFP developers use naming conventions that they believe eliminate this conflict. (As you can tell, I think that's a weak argument.)

Performance, however, is another reason to use MDots. When variables are preceded by MDot, VFP doesn't have to even check whether a table is open in the current work area and, if so, whether that table has a field of the same name. The more fields there are in that table, the more MDots help.

I tested by declaring and initializing two variables and then using them each in two lines of code (the variables represent length and width of a rectangle, and the code computes perimeter and area). In one case, the references to the variables are preceded by MDot; in the other, they're not. I put the calculations in a loop and tested to see how many times I could perform them in a fixed period of time. I ran the test first with no table open in the current work area, then with a table with five fields, and then repeatedly increasing by five fields up to 200. The test code, included in this session's materials as UseMDot.PRG, is shown in **Listing 25**.

Listing 25. Prefacing references to variables with MDot speeds them up. The more fields in the table open in the current work area, the more improvement you'll see.

```
* Compare speed with and without mdot

#DEFINE SECONDSTORUN 5

LOCAL nCase1Start, nCase1LoopEnd, nCase2LoopStart, nCase2LoopEnd
LOCAL nCase1Passes, nCase2Passes
LOCAL nLength, nWidth, nPerimeter, nArea

* Test multiple cases from no table open
* to table with many fields open.
* Store results in a cursor in a different workarea.

CREATE CURSOR csrMDotSpeeds (nFields N(3), nNoMDots I, nMDots I)
SELECT 0

LOCAL nFields, nField, cFieldList

* Initialize variables for calculations
nLength = 27.3
nWidth = 13.7

FOR nFields = 0 TO 200 STEP 5
   IF m.nFields <> 0
      cFieldList = ''
      FOR nField = 1 TO m.nFields
```

```
         cFieldList = m.cFieldList + "cField" + TRANSFORM(m.nField) + " C(5), "
      ENDFOR
      cFieldList = TRIM(m.cFieldList, ", ")

      CREATE CURSOR csrDummy (&cFieldList)
   ELSE
      SELECT 0
   ENDIF

   * Now do the test

   nCase1LoopStart = SECONDS()
   nCase1LoopEnd = m.nCase1LoopStart + SECONDSTORUN
   nCase1Passes = 0

   DO WHILE m.nCase1LoopEnd > SECONDS()
      nCase1Passes = m.nCase1Passes + 1

      nPerimeter = 2*nLength + 2*nWidth
      nArea = nLength * nWidth
   ENDDO

   nCase2LoopStart = SECONDS()
   nCase2LoopEnd = m.nCase2LoopStart + SECONDSTORUN
   nCase2Passes = 0

   DO WHILE m.nCase2LoopEnd > SECONDS()
      nCase2Passes = m.nCase2Passes + 1

      nPerimeter = 2*m.nLength + 2*m.nWidth
      nArea = m.nLength * m.nWidth

   ENDDO

   INSERT INTO csrMDotSpeeds VALUES (m.nFields, m.nCase1Passes, m.nCase2Passes)

   IF m.nFields <> 0
      USE IN csrDummy
   ENDIF
ENDFOR
```

In my tests, with only five fields in the table, the MDot case was able to complete 2.5% to 5% more repetitions. As the number of fields in the table increased, the difference between the two cases increased as well. With 80 fields, the MDot case completed 50% more repetitions. At the top end of the test, with somewhere around 165 or more fields, the MDot case ran more than twice as many repetitions in the same time.

It's worth noting that for this simple test, we're talking about millions of repetitions in five seconds, so the difference in any one case is very small. But in an application, there are likely thousands or tens of thousands of potentially ambiguous references to variables. (The fairly simple program in Listing 25 has 23 such references, though the actual code in the test includes only four such references.)

Use the right loop

The loop construct you use has performance implications. As noted in "Collecting start and end times," earlier in this paper, FOR is about an order of magnitude faster than DO WHILE. In addition, in most cases, SCAN is nearly twice as fast as DO WHILE for traversing a table. (The program DoWhileVsScanFixedTime.PRG in the session materials demonstrates.)

A more interesting case is the FOR EACH loop for looping through arrays or collections. Prior to VFP 8, there were only a few collections native to VFP, like the form's Controls collection and the grid's Columns collection. Most of the collections you needed to deal with, included some that appeared to be native (like the Projects and Files collections), were actually COM objects. As a result, FOR EACH was designed to work with COM objects. By default, the object it hands you each pass through the loop is a COM object.

In VFP 8, the Collection base class was added, giving us the ability to create our own native collections. Suddenly, having FOR EACH provide COM objects caused problems. Those objects didn't behave the way we expected. Not only that, but FOR EACH loops were slow.

So the FOXOBJECT keyword was added in VFP 9; when you add it to FOR EACH, the objects you're working with inside the loop are native VFP objects. Using FOXOBJECT, not only do the objects behave as expected, but FOR EACH without FOXOBJECT takes about 10 to 20 times as long as FOR EACH with FOXOBJECT. The bottom line is that when working with a native collection, you should always add FOXOBJECT to FOR EACH. UseFoxObject.PRG in the session materials demonstrates.

Taking out the slow parts

With a set of tools in place for measuring performance, and an understanding of VFP's Rushmore engine, I was ready to tackle my client's slow code. I found four data files to work with: one small, one medium, one large and one very large. My first step was adding code to log progress. Since the process of opening the data files has two phases (reading the XML into DBFs and converting the DBFs to objects), I added logging as shown in **Listing 26**. (The application's AddToLogFile method time-stamps every line it logs.)

Listing 26. In order to determine whether my changes were having any effect, I logged progress through file opening in an existing log file.

```
goApp.AddToLogFile("  Start phase 1")
* Code to read XML into DBFs
goApp.AddToLogFile("  End phase 1")

* Some error-handling code

goApp.AddToLogFile("  Start phase 2")
* Code to convert DBF data to objects
goApp.AddToLogFile("  End phase 2")
```

With logging in place, I used the application to open each of the four data files twice and computed the time for each phase in each case, using the timestamps from the log. I recorded the initial sets of timing in the relevant bug. (My client uses Bugzilla for bug

tracking; I tracked all progress in a single bug.) I created a simple ASCII table, as in **Listing 27**; as I made changes and retested, I was able to cut and paste this table into the bug and update the second and third columns.

Listing 27. Give yourself a way to see whether changes you make result in better performance. Here, a simple ASCII table records the results of timing tests.

```
+--------------------------------------+
| # of Nodes  | Phase 1    |  Phase 2   |
|             | (seconds)  |  (seconds) |
+--------------------------------------+
|    1        |    1       |     1      |
+--------------------------------------+
|    1        |    1       |     1      |
+--------------------------------------+
|    3        |    4       |     7      |
+--------------------------------------+
|    3        |    5       |     5      |
+--------------------------------------+
|    8        |   16       |    26      |
+--------------------------------------+
|    8        |   15       |    24      |
+--------------------------------------+
|   19        |   33       |    64      |
+--------------------------------------+
|   19        |   35       |    54      |
+--------------------------------------+
|   19        |   34       |    64      |
+--------------------------------------+
```

I also added a property, lOpenCoverageOn, to the application object and wrapped most of phase 2 of the process (for various reasons, a coverage log for phase 1 wasn't likely to be helpful) as in **Listing 28**.

Listing 28. This code lets me control whether to create a coverage log of the code I was trying to optimize.

```
IF goApp.lOpenCoverageOn
  SET COVERAGE TO FORCEPATH("OpenCoverage.log", SYS(2023))
ENDIF

* Call to main phase 2 code

IF goApp.lOpenCoverageOn
  SET COVERAGE TO
ENDIF
```

It's important to note that measuring speed for comparison and creating a coverage log are mutually exclusive activities. Creating a coverage log slows the application down, so runs with coverage logging on shouldn't be compared to runs without it.

Unneeded code

Among the first changes I made were ones that removed unnecessary code. While no one intentionally puts extra code in, over time, some code can become unnecessary. While a few such lines executing won't generally make much of a difference, executing unnecessary code lots of times can make a difference.

My client application tracks a group of settings and includes a timestamp for each to indicate when it was last changed. When I looked carefully, I found that I was setting the timestamp twice for each change: once explicitly in the same code that stored the new value and again in an assign method for the setting. I eliminated the update in the assign method and saw a very small improvement.

Questioning best practices

My philosophy of coding is to make each line of code as independent of its environment as possible. The client application I'm discussing in this paper strengthened my feelings about this practice because it uses several timers, so there's a chance of code running between any two consecutive lines. So, for example, if I have a SQL query, I can't assume that I can use _TALLY in the next line of code to find out how many rows were returned.

When I had to optimize the application, though, it turned out that some of what I considered best practices because they reduced dependency resulted in slower code.

Creating objects

One of the ways I like to make code independent is by using NewObject() rather than CreateObject() to instantiate objects. Since NewObject() expects the class library as a parameter, I don't have to worry about SET CLASSLIB (for classes contained in VCXs) or SET PROCEDURE (for classes contained in PRGs).

The code that converts from tables to objects instantiates hundreds or thousands of objects, depending on the data file size. In the application, I'd been seeing slower performance on the first network file opened after running the application, and some speed-up thereafter. Replacing NewObject() with CreateObject() eliminated that penalty.

To test without all the complications of the client application, I created a simple set of tables to represent students, departments and courses of a school. I generated data for 5,000 students; 2,000 classes in 20 departments; 1,000 instructors; and 50,000 many-to-many records linking students to courses. (My "framework" for generating sample data is described in detail in a paper on my website called "The Why and How of Test Data.") I then created a set of business objects with methods that create collections of the various entities. (This is purely for demonstration purposes. In most applications, it's unlikely you'd want to load all data into objects, rather than working with the tables themselves. My client application is unusual in that loading all the data into objects up front makes sense.)

When I changed the code to SET CLASSLIB and use CreateObject() instead of NewObject(), performance improved by about a third. For example, on one run, the NewObject() code

took 35 seconds to instantiate all 58,020 objects and the corresponding collections, while the CreateObject() version took 27 seconds. In a separate run that accessed the tables across a network connection, the NewObject() version took over 8 minutes, while the CreateObject() version took just over 5 minutes.

The two versions of the classes are included in the materials for this session as BizObjs.VCX and BizObjs2.VCX. The corresponding main programs that do the test are SchoolMain.PRG and SchoolMain2.PRG. The database and its tables are included in a folder called School.

Finding the right record

Another area where I prefer to write environment-independent code is when looking up one or a few records. In general, I use SQL SELECT rather than SEEK, so that the only thing I might need to restore is the work area. (In fact, I try to make my code independent of the work area as well; see "Writing work area-agnostic code," later in the document.)

My client application has a set of metadata tables and when converting from tables to objects, we need to look up information in the metadata for each setting (of which a large data file may have over 10,000). My code for that look-up was something like that shown in **Listing 29**, and was guaranteed to return a single record.

Listing 29. My original code used a query like this to extract meta-data.

```
SELECT Field1, Field2, Field3, Field4 ;
   FROM MyMetaTable ;
   WHERE MyMetaTable.iID = oSetting.iID ;
   INTO CURSOR csrSettingInfo
```

I modified the routine to use SEEK instead and then to directly address the fields of the metadata table. The SEEK looks like **Listing 30**.

Listing 30. This SEEK and direct reference to the fields is much faster than extracting the data of interest with SELECT.

```
SEEK oSetting.iID IN MyMetaTable ORDER iID
```

This was the single biggest improvement I was able to introduce during this process. I wasn't actually surprised that SEEK was faster, but I was surprised how much. Depending on the size of the data file, the time to complete the whole process using SEEK was reduced to anywhere from a quarter to a half of the time required with SELECT.

I created a routine to test the difference without all the overhead of my application. I also tested SEEK two different ways, first with the work area and order set before the test, and then using IN and ORDER as part of the SEEK command. The results were striking. Both versions using SEEK were about an order of magnitude faster than the single-result query. Not surprisingly, the version that assumed the right work area and index order was faster than the one that set it; in this case, it was about 30% faster. (See "Writing work area-agnostic code," later in this document for a broader look at the IN clause.) The test program is shown in **Listing 31** and included in this session's materials as SELECTvsSEEK.PRG.

Listing 31. When you're looking for a single record, SEEK is much faster than SELECT.

```
#DEFINE SECONDSTORUN 5
#DEFINE RECORDSTOFIND 500

LOCAL nCase1Start, nCase1LoopEnd, nCase2LoopStart, nCase2LoopEnd
LOCAL nCase1Passes, nCase2Passes
LOCAL nCase3LoopStart, nCase3LoopEnd, nCase3Passes

** Declare variables needed for test
LOCAL nRand, iRecNo, nClass, aInstructorIDs[RECORDSTOFIND]
LOCAL nClassCount, iInstructorID, cInstructor

** Do set-up work for the test
* Seed the random number generator
RAND(-1)

* Grab a bunch of instructor IDs from the Classes database
OPEN DATABASE school\school
USE Classes
nClassCount = RECCOUNT("Classes")

FOR nClass = 1 TO RECORDSTOFIND
   nRand = RAND()
   iRecNo = CEILING(m.nClassCount * RAND())

   GO m.iRecNo IN Classes
   aInstructorIDs[m.nClass] = Classes.InstructorID
ENDFOR

USE Instructors

** Now do the test

nCase1LoopStart = SECONDS()
nCase1LoopEnd = m.nCase1LoopStart + SECONDSTORUN
nCase1Passes = 0

DO WHILE m.nCase1LoopEnd > SECONDS()
   nCase1Passes = m.nCase1Passes + 1

   ** Do first test case
   FOR nClass = 1 TO RECORDSTOFIND
      iInstructorID = aInstructorIDs[m.nClass]

      SELECT Instructor ;
         FROM Instructors ;
         WHERE InstructorID = m.iInstructorID ;
         INTO CURSOR csrInstructor
   ENDFOR
ENDDO

nCase2LoopStart = SECONDS()
nCase2LoopEnd = m.nCase2LoopStart + SECONDSTORUN
nCase2Passes = 0
```

```
* Set order once
SELECT Instructors
SET ORDER TO PrimaryKey IN Instructors

DO WHILE m.nCase2LoopEnd > SECONDS()
   nCase2Passes = m.nCase2Passes + 1

   ** Do second test case
   FOR nClass = 1 TO RECORDSTOFIND
      iInstructorID = aInstructorIDs[m.nClass]
      SEEK m.iInstructorID
      IF FOUND()
         cInstructor = Instructors.Instructor
      ENDIF
   ENDFOR
ENDDO

nCase3LoopStart = SECONDS()
nCase3LoopEnd = m.nCase3LoopStart + SECONDSTORUN
nCase3Passes = 0

* Use IN and ORDER

DO WHILE m.nCase3LoopEnd > SECONDS()
   nCase3Passes = m.nCase3Passes + 1

   ** Do second test case
   FOR nClass = 1 TO RECORDSTOFIND
      iInstructorID = aInstructorIDs[m.nClass]
      SEEK m.iInstructorID IN Instructors ORDER PrimaryKey
      IF FOUND("Instructors")
         cInstructor = Instructors.Instructor
      ENDIF
   ENDFOR
ENDDO
DEBUG

** Change first expression to something meaningful
DEBUGOUT "SQL SELECT ", m.nCase1Passes, " times."
DEBUGOUT "SEEK with work area and order assumed ", m.nCase2Passes, " times."
DEBUGOUT "SEEK with IN and ORDER", m.nCase3Passes, " times."
```

The previous test addresses the case where you know there's no more than one matching record, that is, a simple look-up. But what about the case where you need to find all matches?

It turns out that, in this case, it matters whether Rushmore can optimize the search. To test, I compared a single SQL SELECT against LOCATE followed by a DO WHILE FOUND() loop. Using a field for which there was a tag, I tried both selecting a single field or selecting all fields in the query; the version using all fields is shown in **Listing 32** (SelectVsLocate.PRG in the materials for this session). In the LOCATE example, I copied the field or fields to local

variables. With local data, the LOCATE version was an order of magnitude faster. Accessing data across a network, the LOCATE version was only about 30% faster.

Listing 32. When looking for multiple records that match a value, LOCATE-CONTINUE is faster than SQL SELECT.

```
#DEFINE SECONDSTORUN 5
#DEFINE RECORDSTOFIND 200

LOCAL nCase1Start, nCase1LoopEnd, nCase2LoopStart, nCase2LoopEnd
LOCAL nCase1Passes, nCase2Passes


** Declare variables needed for test
LOCAL nRand, iRecNo, nInstructor, aInstructorIDs[RECORDSTOFIND]
LOCAL nInstructorCount, iInstructorID
LOCAL iClassID, cClassName, iDepartmentID, iSectionNumber
LOCAL cTerm, cUnits, nYear, cDaysAndTimes

** Do set-up work for the test
* Seed the random number generator
RAND(-1)

* Grab a bunch of instructor IDs from the Instructors database
OPEN DATABASE school\school
USE Instructors
nInstructorCount= RECCOUNT("Instructors")

FOR nInstructor = 1 TO RECORDSTOFIND
   nRand = RAND()
   iRecNo = CEILING(m.nInstructorCount * RAND())

   GO m.iRecNo IN Instructors
   aInstructorIDs[m.nInstructor] = Instructors.InstructorID
ENDFOR

USE Classes

** Now do the test

nCase1LoopStart = SECONDS()
nCase1LoopEnd = m.nCase1LoopStart + SECONDSTORUN
nCase1Passes = 0

DO WHILE m.nCase1LoopEnd > SECONDS()
   nCase1Passes = m.nCase1Passes + 1

   ** Do first test case
   FOR nInstructor = 1 TO RECORDSTOFIND
      iInstructorID = aInstructorIDs[m.nInstructor ]

      SELECT * ;
         FROM Classes ;
         WHERE InstructorID = m.iInstructorID ;
```

```
         INTO CURSOR csrClass
   ENDFOR
ENDDO

nCase2LoopStart = SECONDS()
nCase2LoopEnd = m.nCase2LoopStart + SECONDSTORUN
nCase2Passes = 0

* Set order once
SELECT Classes

DO WHILE m.nCase2LoopEnd > SECONDS()
   nCase2Passes = m.nCase2Passes + 1

   ** Do second test case
   FOR nInstructor = 1 TO RECORDSTOFIND
      iInstructorID = aInstructorIDs[m.nInstructor ]
      LOCATE FOR InstructorID = m.iInstructorID
      DO WHILE FOUND()
         iClassID = Classes.ClassID
         cClassName = Classes.ClassName
         iDepartmentID = Classes.DepartmentID
         iSectionNumber = Classes.SectionNumber
         cTerm = Classes.Term
         cUnits = Classes.Units
         nYear = Classes.Year
         cDaysAndTimes = Classes.DaysAndTimes

         CONTINUE
      ENDDO
   ENDFOR
ENDDO

DEBUG

** Change first expression to something meaningful
DEBUGOUT "SQL SELECT ", m.nCase1Passes, " times."
DEBUGOUT "LOCATE ", m.nCase2Passes, " times."
```

For my next test, I modified the search so that we were looking for records based on a pair of unindexed fields. In this case (included in the session materials as SelectVsLocateNoTag.PRG), with local data, the speed for the two cases was about the same; for network data, the SQL SELECT version was about an order of magnitude faster than the LOCATE version.

Interestingly, in my tests, the fully optimized query from **Listing 31** was slower than the unoptimized query from **Listing 32**. I suspect that says more about the size of the result sets from the queries than about how the queries are being executed.

The right tags, revisited

As I mentioned in "Having the right tags," earlier in this document, the key to VFP's Rushmore technology is having index tags that match the expressions you want to search

or filter on. In my client's application, there's both metadata and data in tables. As the content of the metadata tables expanded to cover new functionality, we failed to add corresponding tags. Although the largest metadata table contains only a few thousand records, I was able to make a significant performance improvement by adding a single tag to that table.

Since using the right tags is optimization 101, it's interesting that the complexity of the application kept me from seeing that I could profitably add tags here.

Writing work area-agnostic code

I consider it a best practice to write code that doesn't care what the current work area is as much as possible. So, I use the IN clause on any Xbase command that supports it and add the alias parameter to any function that accepts it. Doing so makes my code shorter and more robust. (For a full discussion of this topic, see http://www.tomorrowssolutionsllc.com/Articles/Working%20with%20Work%20Areas.pdf.)

After noting that some of my other best practices introduced performance penalties, I was concerned that these practices might also slow things down, so I tested.

My first test (shown in **Listing 33**, and included in the materials for this session as SpeedTestIn.PRG) indicates no significant difference regarding using IN as opposed to setting and restoring the work area. I tested three cases. In the first case, the correct work area was selected once before entering the test loop; the code makes the (risky) assumption that the work area remains unchanged throughout the test. The second case saves the current work area, and selects the right work area just before the relevant commands and finally, reselects the original work area. The final case looks like the code I generally write; no assumption is made about the work area, and each command includes the IN clause to ensure that it executes in the correct work area.

Listing 33. The way you ensure a command applies to the right work area doesn't matter from a performance perspective.

```
* Compare speed of explicit selection of workarea
* vs. using IN. Three cases:
* 1) select work area once and assume it's unchanged
* 2) select work area each time through the loop
* 3) use IN

#DEFINE SECONDSTORUN 5

LOCAL nCase1Start, nCase1LoopEnd
LOCAL nCase2LoopStart, nCase2LoopEnd
LOCAL nCase3LoopStart, nCase3LoopEnd
LOCAL nCase1Passes, nCase2Passes, nCase3Passes

** Declare variables needed for test
LOCAL nOldSelect
```

```
** Do set-up work for the test
OPEN DATABASE School\School
USE Classes
SELECT 0

** Now do the test

nCase1LoopStart = SECONDS()
nCase1LoopEnd = m.nCase1LoopStart + SECONDSTORUN
nCase1Passes = 0

SELECT Classes
DO WHILE m.nCase1LoopEnd > SECONDS()
   nCase1Passes = m.nCase1Passes + 1

   ** Do first test case
   DELETE FOR InstructorID = 25068
   DELETE FOR InstructorID = 25390
   RECALL ALL
ENDDO

nCase2LoopStart = SECONDS()
nCase2LoopEnd = m.nCase2LoopStart + SECONDSTORUN
nCase2Passes = 0

DO WHILE m.nCase2LoopEnd > SECONDS()
   nCase2Passes = m.nCase2Passes + 1

   ** Do second test case
   nOldSelect = SELECT()
   SELECT Classes
   DELETE FOR InstructorID = 25068
   DELETE FOR InstructorID = 25390
   RECALL ALL
   SELECT (m.nOldSelect)
ENDDO

nCase3LoopStart = SECONDS()
nCase3LoopEnd = m.nCase3LoopStart + SECONDSTORUN
nCase3Passes = 0

DO WHILE m.nCase3LoopEnd > SECONDS()
   nCase3Passes = m.nCase3Passes + 1

   ** Do third test case
   DELETE FOR InstructorID = 25068 IN Classes
   DELETE FOR InstructorID = 25390 IN Classes
   RECALL ALL IN Classes
   SELECT 0
ENDDO

DEBUG

** Change first expression to something meaningful
DEBUGOUT "SELECT once ", m.nCase1Passes, " times."
```

```
DEBUGOUT "SELECT each pass ", m.nCase2Passes, " times."
DEBUGOUT "Use IN ", m.nCase3Passes, " times."
```

Since the code using IN is shorter and safer, I'm relieved that there's no penalty for using it.

My next test focused on passing the right alias instead of setting the work area before calling a function. Since I felt SEEK offered the most robust test, the test also looks at explicitly setting the table order vs. passing the desired order. For completeness, the code also checks whether there's any difference between using the SEEK command and the SEEK() function. The test code is shown in **Listing 34**; it's included in the session materials as SpeedTestSeek.PRG.

Listing 34. There are a lot of different ways to do an indexed search. This code tests to see which is fastest.

```
* Compare speed of different approaches
* to SEEK. Six cases:
* 1) SEEK command with alias and order already set
* 2) SEEK command including alias, order already set
* 3) SEEK command with alias and order specified
* 4) SEEK() function with alias and order already set
* 5) SEEK() function including alias, order already set
* 6) SEEK() function with alias and order specified
* In SEEK command cases, FOUND() follows the command
* with respect to whether the alias is specified.

#DEFINE SECONDSTORUN 30
#DEFINE RECORDSTOFIND 100

LOCAL nCase1Start, nCase1LoopEnd, nCase2LoopStart, nCase2LoopEnd, ;
      nCase3LoopStart, nCase3LoopEnd
LOCAL nCase1Passes, nCase2Passes, nCase3Passes
LOCAL nCase4Start, nCase4LoopEnd, nCase5LoopStart, nCase5LoopEnd, ;
      nCase6LoopStart, nCase6LoopEnd
LOCAL nCase4Passes, nCase5Passes, nCase6Passes

** Declare variables needed for test
LOCAL nOldSelect, aStudentIDs[RECORDSTOFIND], nStudentCount, nStudent, iStudentID

SET TALK OFF

** Do set-up work for the test
OPEN DATABASE School\School
USE Students

* Build a list of student IDs to seek
nStudentCount = RECCOUNT("Students")
FOR nStudent = 1 TO RECORDSTOFIND
   nRand = RAND()
   iRecNo = CEILING(m.nStudentCount * RAND())

   GO m.iRecNo IN Students
   aStudentIDs[m.nStudent ] = Students.StudentID
ENDFOR
```

```
** Now do the test
* 1) SEEK command with alias and order already set

nCase1LoopStart = SECONDS()
nCase1LoopEnd = m.nCase1LoopStart + SECONDSTORUN
nCase1Passes = 0

SELECT Students
SET ORDER TO PRIMARYKEY   && STUDENTID

DO WHILE m.nCase1LoopEnd > SECONDS()
   nCase1Passes = m.nCase1Passes + 1

   ** Do first test case
   FOR nStudent = 1 TO RECORDSTOFIND
      iStudentID = aStudentIDs[m.nStudent]
      SEEK m.iStudentID
      IF FOUND()
         * Do something
      ENDIF
   ENDFOR

ENDDO

* 2) SEEK command including alias, order already set
nCase2LoopStart = SECONDS()
nCase2LoopEnd = m.nCase2LoopStart + SECONDSTORUN
nCase2Passes = 0

DO WHILE m.nCase2LoopEnd > SECONDS()
   nCase2Passes = m.nCase2Passes + 1

   ** Do second test case

   FOR nStudent = 1 TO RECORDSTOFIND
      iStudentID = aStudentIDs[m.nStudent]
      SEEK m.iStudentID IN Students
      IF FOUND("Students")
         * Do something
      ENDIF
   ENDFOR

ENDDO

* 3) SEEK command with alias and order specified

nCase3LoopStart = SECONDS()
nCase3LoopEnd = m.nCase3LoopStart + SECONDSTORUN
nCase3Passes = 0

DO WHILE m.nCase3LoopEnd > SECONDS()
   nCase3Passes = m.nCase3Passes + 1

   ** Do third test case
```

```
   FOR nStudent = 1 TO RECORDSTOFIND
      iStudentID = aStudentIDs[m.nStudent]
      SEEK m.iStudentID ORDER PrimaryKey IN Students
      IF FOUND("Students")
         * Do something
      ENDIF
   ENDFOR
ENDDO

* 4) SEEK() function with alias and order already set

nCase4LoopStart = SECONDS()
nCase4LoopEnd = m.nCase4LoopStart + SECONDSTORUN
nCase4Passes = 0

SELECT Students
SET ORDER TO PRIMARYKEY

DO WHILE m.nCase4LoopEnd > SECONDS()
   nCase4Passes = m.nCase4Passes + 1

   ** Do fourth test case
   FOR nStudent = 1 TO RECORDSTOFIND
      iStudentID = aStudentIDs[m.nStudent]
      IF SEEK(m.iStudentID)
      ENDIF
   ENDFOR
ENDDO

* 5) SEEK() function including alias, order already set

nCase5LoopStart = SECONDS()
nCase5LoopEnd = m.nCase5LoopStart + SECONDSTORUN
nCase5Passes = 0

DO WHILE m.nCase5LoopEnd > SECONDS()
   nCase5Passes = m.nCase5Passes + 1

   ** Do fifth test case
   FOR nStudent = 1 TO RECORDSTOFIND
      iStudentID = aStudentIDs[m.nStudent]
      IF SEEK(m.iStudentID, "Students")
      ENDIF
   ENDFOR
ENDDO

* 6) SEEK() function with alias and order specified

nCase6LoopStart = SECONDS()
nCase6LoopEnd = m.nCase6LoopStart + SECONDSTORUN
nCase6Passes = 0

DO WHILE m.nCase6LoopEnd > SECONDS()
   nCase6Passes = m.nCase6Passes + 1
```

```
   ** Do sixth test case
   FOR nStudent = 1 TO RECORDSTOFIND
      iStudentID = aStudentIDs[m.nStudent]
      IF SEEK(m.iStudentID, "Students", "PrimaryKey")
      ENDIF
   ENDFOR
ENDDO

DEBUG

** Change first expression to something meaningful
DEBUGOUT "SEEK Command, assuming alias and order ", m.nCase1Passes, " times."
DEBUGOUT "SEEK Command, assuming order", m.nCase2Passes, " times."
DEBUGOUT "SEEK Command, including alias and order", m.nCase3Passes, " times."
DEBUGOUT "SEEK Function, assuming alias and order ", m.nCase4Passes, " times."
DEBUGOUT "SEEK Function, assuming order", m.nCase5Passes, " times."
DEBUGOUT "SEEK Function, including alias and order", m.nCase6Passes, " times."
```

I set up a loop to run the tests repeatedly and store the results in a cursor. I felt this would help smooth out any issues caused by other things running at the same time.

I found that the SEEK() function is a little faster than the corresponding SEEK command. This was least true when alias and order were assumed (cases 1 and 4), with an average of less than a 1% difference. It was most true for the case where neither alias nor order was assumed (cases 3 and 6), where the function was about 10% faster than the command.

As for different ways of using the command and the function, not surprisingly, setting the work area and order once before the test loop (cases 1 and 4) was fastest, and explicitly specifying the alias and order (cases 3 and 6) was slowest. The differences were more pronounced for the command than for the function. **Table 3** shows the average differences, rounded to integers.

Table 3. The SEEK() function is faster than the SEEK command. Not surprisingly, assuming you already are in the right work area and have the right order set is the fastest case, though also the riskiest.

Faster Test Case	**Slower Test Case**	**Average difference**
Case 1: SEEK command, alias and order already specified	Case 2: SEEK command, alias included, order already specified	11%
Case 1: SEEK command, alias and order already specified	Case 3: SEEK command, alias and order included	27%
Case 2: SEEK command, alias included, order already specified	Case 3: SEEK command, alias and order included	15%
Case 4: SEEK function, alias and order already specified	Case 5: SEEK function, alias included, order already specified	9%
Case 4: SEEK function, alias and order already specified	Case 6: SEEK function, alias and order included	17%

Faster Test Case	Slower Test Case	Average difference
Case 5: SEEK function, alias included, order already specified	Case 6: SEEK function, alias and order included	7%

I already prefer the SEEK function and now that I know it's faster, I'll stick with it. In addition, I'll be cognizant of these results when optimizing, but also cautious about writing code that makes assumptions about work areas and index orders. After all, accurate code beats fast code every time.

Resources

Not surprisingly, lots of words have been written about code optimization. Here are a few good references.

Steve McConnell's classic "Code Complete" has a couple of chapters addressing the topic. While not all of his suggestions are relevant for VFP, there's plenty of meat there.

The VFP Help file includes a number of topics on optimization. Start with "Optimizing Applications."

I've written previously about Rushmore and performance of VFP's SQL commands. These two articles on my website go into more depth on SQL ShowPlan and optimizing queries: http://tinyurl.com/pjc9m7w and http://tinyurl.com/ohtbcyw.

"Hacker's Guide to Visual FoxPro," which I wrote with Ted Roche, Doug Hennig and Della Martin, has a chapter called "Faster Than a Speeding Bullet," that discusses various performance issues, some of which are covered in this paper and some of which are not.

The Bottom Line

Optimizing applications is an ongoing process. The first step is having some idea what's fast and what's slow in the language you're working in. Then, you can choose clearly faster alternatives when writing code in the first place.

On the other hand, for most applications, it's not a good idea to spend time squeezing in every millisecond of performance before the code actually works. In many cases, a slightly slower, but more readable or maintainable, option is a better choice.

Once an application works, if its performance is a problem, it's time to bring out the tools that let you figure out exactly what's slow and tune those portions.

