Data Strategies in VFP: Introduction

By Doug Hennig

Overview

There are a lot of ways you can access non-VFP data (such as SQL Server) in VFP applications: remote views, SQL passthrough, ADO, XML ... This document will examine the pros and cons of different mechanisms and discuss when it's appropriate to use a particular strategy. We'll also look at an exciting new technology in VFP called CursorAdapter that will make accessing remote data much easier than it is in earlier versions.

Introduction

More and more VFP developers are storing their data in something other than VFP tables, such as SQL Server or Oracle. There are a lot of reasons for this, including fragility of VFP tables (both perceived and actual), security, database size, and corporate standards. Microsoft has made access to non-VFP data easier with every release, and has even been encouraging it with the inclusion of MSDE (Microsoft Data Engine, a free, stripped-down version of SQL Server) on the VFP 7 CD.

However, accessing a backend database has never been quite as easy as using VFP tables. In addition, there are a variety of mechanisms you can use to do this:

- Remote views, which are based on ODBC connections.
- SQL passthrough (SPT) functions, such as SQLCONNECT(), SQLEXEC(), and SQLDISCONNECT(), which are also based on ODBC connections.
- ActiveX Data Objects, or ADO, which provide an object-oriented front end to OLE DB providers for database engines.
- XML, which is a lightweight, platform-independent, data transport mechanism.

Which mechanism should you choose? The answer (as it is for the majority of VFP questions <g>) is "it depends". It depends on a number of factors, including your development team's experience and expertise, your infrastructure, the needs of the application, anticipated future requirements, and so on.

Let's take a look at each of these mechanisms, including their advantages and disadvantages, some techniques to be aware of, and my opinion of where each fits in the overall scheme of things. We'll also take a look at what I think is one of the biggest new features in VFP 8, the CursorAdapter class, and how it makes using remote data access via ODBC, ADO, or XML much easier and consistent.

Remote Views

Like a local view, a remote view is simply a pre-defined SQL SELECT statement that's defined in a database container. The difference is that a remote view accesses data via ODBC (even if the data it's accessing is VFP) rather than natively.

You can create a remote view either programmatically using the CREATE SQL VIEW command or visually using the View Designer. In both cases, you need to specify an ODBC connection to use. The connection can either be an ODBC data source (DSN) set up on your system or a Connection object that's already been defined in the same database. Here's an example that creates a remote view to the Customers table of the sample Northwind database that comes with SQL Server (we'll use this database extensively in our examples). This was excerpted from code generated by GENDBC; there's actually a lot more code that sets the various properties of the connection, the view, and the fields.

```
CREATE CONNECTION NORTHWINDCONNECTION ;
   CONNSTRING "DSN=Northwind SQL; UID=sa; PWD=testdb; " + ;
   "DATABASE=Northwind; TRUSTED_CONNECTION=No"
CREATE SQL VIEW "CUSTOMERSVIEW" ;
   REMOTE CONNECT "NorthwindConnection" ;
   AS SELECT * FROM dbo.Customers Customers
DBSetProp('CUSTOMERSVIEW', 'View', 'UpdateType', 1)
DBSetProp('CUSTOMERSVIEW', 'View', 'WhereType', 3)
DBSetProp('CUSTOMERSVIEW', 'View', 'FetchMemo', .T.)
DBSetProp('CUSTOMERSVIEW', 'View', 'SendUpdates', .T.)
DBSetProp('CUSTOMERSVIEW', 'View', 'Tables', 'dbo.Customers')
DBSetProp('CUSTOMERSVIEW.customerid', 'Field', 'KeyField', .T.)
DBSetProp('CUSTOMERSVIEW.customerid', 'Field', 'Updatable', .F.)
DBSetProp('CUSTOMERSVIEW.customerid', 'Field', 'UpdateName', ;
   'dbo.Customers.CustomerID')
DBSetProp('CUSTOMERSVIEW.companyname', 'Field', 'Updatable', .T.)
DBSetProp('CUSTOMERSVIEW.companyname', 'Field', 'UpdateName', ;
   'dbo.Customers.CompanyName')
```

One of the easiest ways you can upsize an existing application is using the Upsizing Wizard to create SQL Server versions of your VFP tables, then create a new database (for example, REMOTE.DBC) and create remote views in that database that have the same names as the tables they're based on. That way, the code to open a remote view will be exactly the same as that to open a local table except you'll open a different database first. For example:

```
if oApp.lUseLocalData
  open database Local
else
  open database Remote
endif
use CUSTOMERS
```

If you're using cursor objects in the DataEnvironment of forms and reports, you have a little bit of extra work to do because those objects have a reference to the specific database you had selected when you dropped the views into the DataEnvironment. To handle this, put code similar to the following into the BeforeOpenTables method of the DataEnvironment:

```
local loObject
for each loObject in This.Objects
  if upper(loObject.BaseClass) = 'CURSOR' and not empty(loObject.Database)
    loObject.Database = iif(oApp.lUseLocalData, 'local.dbc', 'remote.dbc')
  endif upper(loObject.BaseClass) = 'CURSOR' ..
next loObject
```

One thing to be aware of: When you open a view, VFP attempts to lock the view's records in the DBC, even if it's only briefly. This can cause contention in busy applications where several users might try to open a form at the same time. Although there are workarounds (copying the DBC to the local workstation and using that one or, in VFP 7 and later, using SET REPROCESS SYSTEM to increase the timeout for lock contention), it's something you'll need to plan for.

There's quite a controversy over whether remote views are a good thing or not. If you're interested in reading about the various sides of the arguments, check out these links:

http://fox.wikis.com/wc.dll?Wiki~RemoteViews~VFP

http://fox.wikis.com/wc.dll?Wiki~MoreOnRemoteViews~VFP

http://fox.wikis.com/wc.dll?Wiki~Client/ServerDataAccessTechniques~VFP

http://fox.wikis.com/wc.dll?Wiki~Client/ServerTechniquesPerformance~VFP

Advantages

The advantages of remote views are:

- You can use the View Designer to create a remote view visually. Okay, until VFP 8, which fixes many of the known problems with the View Designer (especially with complex views involving more than two tables), this wasn't necessarily an advantage <g>, but even in earlier versions, simple views can be created very quickly and easily. It's great to visually see all the fields in the underlying tables, easily set up the various parts of the SQL SELECT statement using a friendly interface, and quickly set properties of the view using checkboxes or other UI elements.

- From a language point-of-view, remote views act just like tables. As a result, they can be used anywhere: you can USE them, add them to the DataEnvironment of a form or report, bind them to a grid, process them in a SCAN loop, and so forth. With some of the other technologies, specifically ADO and XML, you have to convert the result set to a VFP cursor before you can use it in many places in VFP.

- It's easier to convert an existing application to use remote views, especially if it already uses local views, than using any of the other techniques.

- Because you can add a remote view to the DataEnvironment of a form or report, you can take advantage of the visual support the DE provides: dragging and dropping fields or the entire cursor to automatically create controls, easily binding a control to a field by selecting it from a combobox in the Properties Window, and so on. Also, depending on the settings of the

AutoOpenTables and OpenViews properties, VFP will automatically open the remote views for you.

- It's easy to update the backend with changes: assuming the properties of the view have been set up properly, you simply call TABLEUPDATE(). Transaction processing and update conflict detection are built-in.

- Remote views are easy to use in a development environment: just USE and then BROWSE.

Disadvantages

The disadvantages of remote views are:

- Remote views live in a DBC, so that one more set of files you have to maintain and install on the client's system.

- Since a remote view's SQL SELECT statement is pre-defined, you can't change it on the fly. Although this is fine for a typical data entry form, it can be an issue for queries and reports. You may have to create several views from the same set of data, each varying in the fields selected, structure of the WHERE clause, and so forth.

- You can't call a stored procedure from a remote view, so a remote view needs direct access to the underlying tables. This may be an issue with your application's database administrators; some DBAs believe that data access should only be via stored procedures for security and other reasons. Also, because they are precompiled on the backend, stored procedures often perform significantly faster than SQL SELECT statements.

- When you use TABLEUPDATE() to write changes in the view to the backend database, you have little ability (other than by setting a few properties) to control how VFP does the update.

- As is the case with local views, if you use a SELECT * view to retrieve all the fields from a specific table and the structure of that table on the backend changes, the view is invalid and must be recreated.

- They work with ODBC only, so they're stuck in the "client-server" model of direct data connections. They can't take advantage of the benefits of ADO or XML.

- As I mentioned earlier, DBC lock contention is an issue you need to handle.

- Like other types of VFP cursors, you can't pass the result set outside the current data session, let alone to another application or tier. This limitation alone pretty much makes them useless in an n-tier application.

- Until VFP 8, which allows you to specify the connection handle to use when you open a remote view with the USE statement, you had little ability to manage the connections used by your application.

- The connection information used for a remote view is hard-coded in plain text in the DBC (notice the user name and password in the code shown earlier). That means that a hacker can easily discover the keys to your backend kingdom (such as the user name and password) using nothing more complicated than Notepad to open the DBC. This isn't much of an issue starting in VFP 7 because it allows you to specify a connection string when you open a remote view with the USE command, meaning that you can dynamically assemble the server, user name, and password, likely from encrypted information, just before opening the view.

Basically, it comes down to a control issue: remote views make it easy to work with backend data, but at the expense of limiting the control you have over them.

When to Use

Remote views are really only suited to client-server, 2-tier applications where you have a direct connection to the data. I believe that in the long run, most applications will be more flexible, robust, and have a longer shelf life if they're developed using an n-tier design, so remote views are best suited to those cases where you're upsizing an existing application and don't want to redesign/redevelop it or when your development team doesn't have much experience with other technologies.

SQL Passthrough

VFP provides a number of functions, sometimes referred to as SQL passthrough (or SPT) functions, which allow you to access a backend database. SQLCONNECT() and SQLSTRINGCONNECT() make a connection to the backend database engine; the difference between these two functions is that SQLCONNECT() requires an existing ODBC Data Source (DSN) that's been defined on the user's system, while SQLSTRINGCONNECT() simply passes the necessary information directly to ODBC, known as a DSNless connection. As you may guess, SQLDISCONNECT() disconnects from the backend. SQLEXEC() sends a command, such as a SQL SELECT statement, to the database engine, typically (but not necessarily, depending on the command) putting the returned results into a VFP cursor.

Here's an example that opens a connection to the Northwind database (this assumes there's a DSN called "Northwind" that defines how to connect to this database), retrieves a single customer record and displays it in a BROWSE window, and then closes the connection.

```
lnHandle = sqlconnect('Northwind')
if lnHandle > 0
  sqlexec(lnHandle, "select * from customers where customerid = 'ALFKI'")
  browse
  sqldisconnect(lnHandle)
else
  aerror(laErrors)
  messagebox('Could not connect: ' + laErrors[2])
endif lnHandle > 0
```

To use a DSNless connection instead, replace the SQLCONNECT() statement with the following (replace the name of the server, user ID, and password with the appropriate values):

```
lnHandle = sqlstringconnect('Driver=SQL Server;Server=(local);' + ;
  Database=Northwind;uid=sa;pwd=whatever')
```

The DispLogin SQL setting controls whether or not ODBC displays a login dialog. Although you might think it's handy to let ODBC worry about user names and passwords, you don't have any control over the appearance of the dialog or what happens if the user enters improper values. Instead, you're better off asking the user for the appropriate information in your own VFP dialog and then passing the information to ODBC. As described in the VFP help for the SQLCONNECT() function, you should use SQLSETPROP() to set DispLogin to 3.

Rather than having each component that requires access to remote data manage their own ODBC connections, use an object whose sole responsibility is managing the connection and accessing the data. SFConnectionMgr, which is based on Custom, is an example. It has several custom properties, some of which you must set before you can use it to connect to a remote data source.

Property	**Description**
cDatabase	The database to connect to; can leave blank if cDSN filled in.
cDriver	The ODBC driver or OLE DB provider to use; can leave blank if cDSN filled in.
cDSN	The ODBC DSN to connect to; leave blank to use a DSNless connection.
cErrorMessage	The message of any error that occurs.
cPassword	The password for the data source; can leave blank if a trusted connection is used.
cServer	The server the database is on; can leave blank if cDSN filled in.
cUserName	The user name for the data source; can leave blank if a trusted connection is used.
lConnected	.T. if we are connected to the data source.

It has several custom methods:

Method	**Description**
Connect	Connects to the data source.

Disconnect	Disconnects from the data source (called from Destroy but can also call manually).
Execute	Executes a statement against the data source.
GetConnection	Returns the connection handle or ADO Connection object.
GetConnectionString	Returns the connection string from the various connection properties (protected).
HandleError	Sets the cErrorMessage property when an error occurs (protected).

SFConnectionMgr isn't intended to be used directly, but is subclassed into SFConnectionMgrODBC and SFConnectionMgrADO, which are specific for ODBC and ADO, respectively. Those subclasses override most of the methods to provide the specific behavior needed. Let's look at SFConnectionMgrODBC.

The Init method saves the current DispLogin setting to a custom nDispLogin property and sets it to 3 so the login dialog is never displayed.

```
This.nDispLogin = sqlgetprop(0, 'DispLogin')
sqlsetprop(0, 'DispLogin', 3)
dodefault()
```

The Connect method checks to see if we're already connected, then either uses SQLCONNECT() if a DSN was specified in the cDSN property or builds a connection string and uses the SQLSTRINGCONNECT() function. Either way, if the connection succeeded, the nHandle property contains the connection handle and lConnected is .T. If it failed, nHandle is 0, lConnected is .F, and cErrorMessage contains information about what went wrong (set in the HandleError method, which we won't look at).

```
* If we're not already connected, connect to either the specified DSN or to
* the data source using a DSNless connection.

local lcConnString
with This
  if not .lConnected
    if empty(.cDSN)
      lcConnString = .GetConnectionString()
      .nHandle = sqlstringconnect(lcConnString)
    else
      .nHandle = sqlconnect(.cDSN, .cUserName, .cPassword)
    endif empty(.cDSN)

* Set the lConnected flag if we succeeded in connecting, and if not, get the
* error information.

    .lConnected = .nHandle > 0
```

```
    if not .lConnected
      .nHandle = 0
      .HandleError()
    endif not .lConnected
  endif not .lConnected
endwith
return This.lConnected
```

GetConnectionString returns a connection string from the values of the connection properties.

```
local lcSpecifier, ;
  lcConnString
with This
  lcSpecifier  = iif(upper(.cDriver) = 'SQL SERVER', 'database=', ;
    'dbq=')
  lcConnString = 'driver=' + .cDriver + ';' + ;
    iif(empty(.cServer),   '', 'server='   + .cServer   + ';') + ;
    iif(empty(.cUserName), '', 'uid='      + .cUserName + ';') + ;
    iif(empty(.cPassword), '', 'pwd='      + .cPassword + ';') + ;
    iif(empty(.cDatabase), '', lcSpecifier + .cDatabase + ';') + ;
    'trusted_connection=' + iif(empty(.cUserName), 'yes', 'no')
endwith
return lcConnString
```

The Execute method executes a statement (such as a SQL SELECT command) against the data source. It first calls Connect to ensure we have a connection, then uses the SQLEXEC() function to pass the statement to the data source.

```
lparameters tcStatement, ;
  tcCursor
local lcCursor, ;
  llReturn
with This
  .Connect()
  if .lConnected
    lcCursor = iif(vartype(tcCursor) = 'C' and not empty(tcCursor), ;
      tcCursor, sys(2015))
    llReturn = sqlexec(.nHandle, tcStatement, lcCursor) >= 0
    if not llReturn
      .HandleError()
    endif not llReturn
  endif .lConnected
endwith
return llReturn
```

The Disconnect method disconnects from the data source and sets nHandle to 0 and lConnected to .F.

```
with This
  if .lConnected
    sqldisconnect(.nHandle)
    .nHandle    = 0
    .lConnected = .F.
  endif .lConnected
endwith
```

Destroy simply disconnects and restores the saved DispLogin setting.

```
This.Disconnect()
sqlsetprop(0, 'DispLogin', This.nDispLogin)
dodefault()
```

Here's an example (taken from TestConnMgr.prg) that shows how SFConnectionMgrODBC can be used. It first connects to the SQL Server Northwind database and grabs all the customer records, and then it connects to the Access Northwind database and again retrieves all the customer records. Of course, this example uses hard-coded connection information; a real application would likely store this information in a local table, an INI file, or the Windows Registry to make it more flexible.

```
loConnMgr = newobject('SFConnectionMgrODBC', 'SFRemote')
with loConnMgr

* Connect to the SQL Server Northwind database and get customer records.

  .cDriver   = 'SQL Server'
  .cServer   = '(local)'
  .cDatabase = 'Northwind'
  .cUserName = 'sa'
  .cPassword = ''
  do case
    case not .Connect()
      messagebox(.cErrorMessage)
    case .Execute('select * from customers')
      browse
      use
    otherwise
      messagebox(.cErrorMessage)
  endcase

* Now connect to the Access Northwind database and get customer records.

  .Disconnect()
  .cDriver   = 'Microsoft Access Driver (*.mdb)'
  .cServer   = ''
  .cDatabase = 'd:\Program Files\Microsoft Visual Studio\VB98\Nwind.mdb'
  .cUserName = ''
  .cPassword = ''
  do case
    case not .Connect()
      messagebox(.cErrorMessage)
    case .Execute('select * from customers')
      browse
      use
    otherwise
      messagebox(.cErrorMessage)
  endcase
endwith
```

Advantages

The advantages of using SPT are:

- You have a lot more flexibility in data access than with remote views, such as calling stored procedures using the SQLEXEC() function.

- You can change the connection information on the fly as needed. For example, you can store the user name and password as encrypted values and only decrypt them just before using them in the SQLCONNECT() or SQLSTRINGCONNECT() functions. You can also change servers or even backend database engines (for example, you could switch between SQL Server and Access database by simply changing the ODBC driver specified in the SQLSTRINGCONNECT() call). As I mentioned earlier, this isn't nearly the advantage over remote views that it used to be, now that VFP 7 allows you to specify the connection string on the USE command.

- You can change the SQL SELECT statement as needed. For example, you can easily vary the list of the fields, the WHERE clause (such as changing which fields are involved or eliminating it altogether), the tables, and so on.

- You don't need a DBC to use SPT, so there's nothing to maintain or install, lock contention isn't an issue, and you don't have to worry about a SELECT * statement being made invalid when the structure of the backend tables change.

- As with remote views, the result set of a SPT call is a VFP cursor, which can be used anywhere in VFP without the conversion issues that ADO and XML have.

- Although you have to code for it yourself (this is discussed in more detail under Disadvantages), you have greater control over how updates are done. For example, you might use a SQL SELECT statement to create the cursor but call a stored procedure to update the backend tables.

- You can manage your own connections. For example, you might want to use a connection manager similar to the one discussed earlier to manage all the connections used by your application in one place.

Disadvantages

The disadvantages of using SPT are:

- It's more work, since you have to code everything: creating and closing the connection, the SQL SELECT statements to execute, and so on. You don't have a nice visual tool like the View Designer to show you which fields exist in which tables on the backend.

- You can't visually add a cursor created by SPT to the DataEnvironment of a form or report. Instead, you have to code the opening of the cursors (for example, in the BeforeOpenTables method), you have to manually create the controls, and you have to fill in the binding

properties (such as ControlSource) by typing them yourself (don't make a typo when you enter the alias and field names or the form won't work).

- They're harder to use than remote views in a development environment: instead of just issuing a USE command, you have to create a connection, then use a SQLEXEC() call to get the data you want to look at. You can make things easier on yourself if you create a set of PRGs to do the work for you (such as UseCustomers.prg, which opens and displays the contents of the Customers table). You can also use the SQL Server Enterprise Manager (or similar tool) to examine the structures and contents of the tables. You could even create a DBC and set of remote views used only in the development environment as a quick way to look at the data.

- As with remote views and other types of VFP cursors, you can't pass the result set outside the current data session. Instead, you may have to pass the information used to create the cursor (the SQL SELECT statement or stored procedure call, and even possibly the connection information) and let the other object do the work itself.

- Cursors created with SPT can be updatable, but you have to make them so yourself using a series of CURSORSETPROP() calls to the SendUpdates, Tables, KeyFieldList, UpdatableFieldList, and UpdateNameList properties. Also, you have to manage transaction processing and update conflict detection yourself.

- Since SPT cursors aren't defined like remote views, you can't easily switch between local and remote data using SPT as you can with remote views (by simply changing which view you open in a form or report).

- Like remote views, SPT is based on ODBC only, so you can't take advantage of the benefits of ADO or XML.

When to Use

As with remote views, SPT is best suited to client-server, 2-tier applications where you have a direct connection to the data. Since it's more work to convert an existing application to SPT than remote view or CursorAdapters (which we'll see later), SPT is best suited to utilities, simple applications, or narrow-focus cases.

ADO

OLE DB and ADO are part of Microsoft's Universal Data Access strategy, in which data of any type stored anywhere in any format, not just in relational databases on a local server, can be made available to any application requiring it. OLE DB providers are similar to ODBC drivers: they provide a standard, consistent way to access data sources. A variety of OLE DB providers are available for specific DBMS (SQL Server, Oracle, Access/Jet, etc.), and Microsoft provides an OLE DB provider for ODBC data sources.

Because OLE DB is a set of low-level COM interfaces, it's not easy to work with in languages like VFP. To overcome this, Microsoft created ActiveX Data Objects, or ADO, a set of COM objects that provide an object-oriented front-end to OLE DB.

ADO consists of several objects, including:

- Connection: This is the object responsible for communicating with the data source.
- RecordSet: This is the equivalent of a VFP cursor: it has a defined structure, contains the data in the data set, and provides properties and methods to add, remove, or update records, move from one to another, filter or sort the data, and update the data source.
- Command: This object provides the means of doing more advanced queries than a simple SELECT statement, such as parameterized queries and calling stored procedures.

Here's an example (ADOExample.prg) that gets all Brazilian customers from the Northwind database and displays the customer ID and company name. Notice that the Connection object handles the connection (the RecordSet's ActiveConnection property is set to the Connection object), while the RecordSet handles the data.

```
local loConn as ADODB.Connection, ;
  loRS as ADODB.Recordset
loConn = createobject('ADODB.Connection')
loConn.ConnectionString = 'provider=SQLOLEDB.1;data source=(local);' + ;
  'initial catalog=Northwind;uid=sa;pwd='
loConn.Open()
loRS = createobject('ADODB.Recordset')
loRS.ActiveConnection = loConn
loRS.LockType         = 3  && adLockOptimistic
loRS.CursorLocation   = 3  && adUseClient
loRS.CursorType       = 3  && adOpenStatic
loRS.Open("select * from customers where country='Brazil'")
lcCustomers = ''
do while not loRS.EOF
  lcCustomers = lcCustomers + loRS.Fields('customerid').Value + chr(9) + ;
    loRS.Fields('companyname').Value + chr(13)
  loRS.MoveNext()
enddo while not loRS.EOF
messagebox(lcCustomers)
loRS.Close()
loConn.Close()
```

Because of its object-oriented nature and its capabilities, ADO has been the data access mechanism of choice for n-tier development (although this is quickly changing as XML becomes more popular). Unlike ODBC, you don't have to have a direct connection to the data source.

For details on ADO and using it in VFP, see John Petersen's "ADO Jumpstart for Visual FoxPro Developers" white paper, available from the VFP home page (http://msdn.microsoft.com/vfoxpro; follow the "Technical Resources", "Technical Articles", and "COM and ActiveX Development" links to get to the document).

Advantages

The advantages of using ADO are:

- As with SPT, you have more flexibility in data access than with remote views, such as calling stored procedures.
- You can change the connection information on the fly as needed, such as encrypting the user name and password, changing servers, or even backend database engines.
- You can change the SQL SELECT statement as needed.
- There's no DBC involved.
- Although performance differences aren't significant in simple scenarios (in fact, in my testing, ODBC is faster than ADO), ADO is more scalable in heavily-used applications such as Web servers.
- Unlike VFP cursors, ADO objects can be passed outside the current data session, whether to another component in the application, to another application or COM object (such as Excel or a middle-tier component), or to another computer altogether. You can even send them over HTTP using Remote Data Services (RDS; see John Petersen's white paper for more information), although this is a problem when firewalls are involved.
- ADO is object-oriented, so you can deal with the data like objects.
- Depending on how it's set up, ADO RecordSets are automatically updateable without any additional work (other than calling the Update or UpdateBatch methods). Transaction processing and update conflict detection are built-in.
- You can manage your own connections.
- You can easily persist a RecordSet to a local file, then later reload it and carry on working, and finally update the backend data source. This makes it a much better choice for "road warrior" applications than remote views or SPT.

Disadvantages

The disadvantages of ADO are:

- It's more work, since you have to code everything: creating and closing the connection, the SQL SELECT statements to execute, and so on. You don't have a nice visual tool like the View Designer to show you which fields exist in which tables on the backend.
- An ADO RecordSet is not a VFP cursor, so you can't use it in places that require a cursor, such as grids and reports. There are functions in the VFPCOM utility (available for download from the VFP home page, http://msdn.microsoft.com/vfoxpro) that can convert a RecordSet

to a cursor and vice versa, but using them can impact performance, especially with large data sets, and they have known issues with certain data types.

- There's no visual support for ADO RecordSets, so have to code their creation and opening, you have to manually create the controls, and you have to fill in the binding properties (such as ControlSource) by typing them yourself. This is even more work than for SPT, because the syntax isn't just CURSOR.FIELD—it's RecordSet.Fields('FieldName').Value.

- They're the hardest of the technologies to use in a development environment, since you have to code everything: making a connection, retrieving the data, moving back and forth between records. You can't even BROWSE to see visually what the result set looks like (unless you use VFPCOM or another means to convert the RecordSet to a cursor).

- There's a bigger learning curve involved with ADO than using the cursors created by ODBC.

- You'll likely have to ensure the client has the latest version of Microsoft Data Access Components (MDAC) installed to ensure their version of OLEDB and ADO match what your application requires.

- ADO is a Windows-only technology.

When to Use

ADO is easily used when passing data back and forth between other components. For example, a method of a VFP COM object can easily return an ADO RecordSet to Excel VBA code, which can then process and display the results.

If you're designing an application using an n-tier architecture, ADO may be a good choice if you're already familiar with it or already have infrastructure in place for it. However, XML is fast becoming the mechanism of choice for n-tier applications, so I expect ADO to be used less and less in this area.

XML

XML (Extendible Markup Language) isn't really a data access mechanism; it's really a transport technology. The data is packaged up as text with a structured format and then shipped off somewhere. However, because it's simply text, XML has a lot of advantages over other technologies (as we'll discuss later).

A few years ago, Microsoft "discovered" XML, and has been implementing it pretty much everywhere since then. The data access technology built into the .NET framework, ADO.NET, has XML as its basis (in fact, one simplistic view is that ADO.NET is really just a set of wrapper classes that expose XML data via an OOP interface). Web Services, based on SOAP (Simple Object Access Protocol), use XML as the basis for communications and data transfer. XML is even quickly becoming the data transport mechanism of choice for n-tier applications, which for a long time favored ADO in that role.

VFP 7 added several functions that work with XML: XMLTOCURSOR(), which converts XML to a cursor, CURSORTOXML(), which does the opposite, and XMLUPDATEGRAM(), which generates an updategram (XML in a specific format for indicating changes to data) from changes to a cursor. Here's some VFP code (taken from XMLExample1.prg) that shows how VFP can deal with XML:

```
* Get the information for the ALFKI customer and display the raw XML.

close databases all
lcXML = GetCustomerByID('ALFKI')
strtofile(lcXML, 'ALFKI.XML')
modify file ALFKI.XML
erase ALFKI.XML

* Put it into a cursor, browse and make changes, then view the updategram.

xmltocursor(lcXML, 'CUSTOMERS')
set multilocks on
cursorsetprop('Buffering', 5)
browse
lcUpdate = xmlupdategram()
strtofile(lcUpdate, 'UPDATE.XML')
modify file UPDATE.XML

* By setting the KeyFieldList property, the updategram will only contain key
* and changed fields.

cursorsetprop('KeyFieldList', 'cust_id')
lcUpdate = xmlupdategram()
strtofile(lcUpdate, 'UPDATE.XML')
modify file UPDATE.XML
erase UPDATE.XML
close databases all

* This function returns the specified customer record as XML.

function GetCustomerByID(tcCustomerID)
local lcXML
open database (_samples + 'data\testdata')
select * from customer where cust_id = tcCustomerID into cursor Temp
cursortoxml('Temp', 'lcXML', 1, 8, 0, '1')
use in Temp
use in Customer
return lcXML
```

Note that until version 8, while VFP could create an XML updategram, it didn't have a simple way to consume one (that is, to update VFP tables). Visual FoxPro MVP Alex Feldstein wrote a routine to do this (http://fox.wikis.com/wc.dll?Wiki~XMLUpdateGramParse), but in VFP 8, you can use an instance of the new XMLAdapter class to do this (we'll take a look at that class in the "Data Strategies in VFP: Advanced" document.

Here's an example (XMLExample2.prg) that uses SQLXML to get a customer's record from SQL Server via a Web Server (we'll discuss SQLXML in more detail in the Advanced document), then send changes back.

```
* Get the information for the ALFKI customer and display the raw XML.

close databases all
lcXML = GetNWCustomerByID('ALFKI')
strtofile(lcXML, 'ALFKI.XML')
modify file ALFKI.XML
erase ALFKI.XML

* Put it into a cursor, browse, and make changes.

xmltocursor(lcXML, 'CUSTOMERS')
cursorsetprop('KeyFieldList', 'customerid')
set multilocks on
cursorsetprop('Buffering', 5)
browse

* Get the updategram and save the changes to SQL Server.

lcUpdate = xmlupdategram()
SaveNWCustomers(lcUpdate)
use

* This function uses SQLXML to get the specified customer record as XML.

function GetNWCustomerByID(tcCustomerID)
local loXML as MSXML2.XMLHTTP
loXML = createobject('MSXML2.XMLHTTP')
loXML.open('POST', 'http://localhost/northwind/template/ ' + ;
  'customersbyid.xml?customerid=' + tcCustomerID, .F.)
loXML.setRequestHeader('Content-type', 'text/xml')
loXML.send()
return loXML.responseText

* This function uses SQLXML to get the specified customer record as XML.

function SaveNWCustomers(tcDiffGram)
local loDOM as MSXML2.DOMDocument, ;
  loXML as MSXML2.XMLHTTP
loDOM = createobject('MSXML2.DOMDocument')
loDOM.async = .F.
loDOM.loadXML(tcDiffGram)
loXML = createobject('MSXML2.XMLHTTP')
loXML.open('POST', 'http://localhost/northwind/', .F.)
loXML.setRequestHeader('Content-type', 'text/xml')
loXML.send(loDOM)
```

Advantages

There are a lot of benefits to using XML:

- You have the same benefits over remote views, such as no DBC and an easily-changed SQL SELECT statement, as SPT and ADO have.

- Since XML isn't really a data access mechanism, you have the most flexibility in data access with XML. For example, you can use the VFP CURSORTOXML() function, Web Services,

middle tier components, ADO.NET DataSets, XMLHTTP, and many other mechanisms to get data from one place to another.

- Since it's just text, XML can be passed anywhere, even through firewalls (which is a problem for ADO).
- The XML DOM object provides an object-oriented interface to the XML data.
- XML is easily persisted anywhere: to a file, a memo field in a table, etc.
- XML is completely platform/operating system-independent.
- XML is human-readable, unlike the binary formats of other mechanisms.

Disadvantages

There are also some downsides to using XML:

- Belying its simple format, there's a bigger learning curve with XML than there is with the other mechanisms. The XML DOM object has its own object model, there's all kinds of new technology (and acronyms!) to learn like schemas, XSLT, XPath, XDR, and XQueries.
- The same set of data can be quite a bit larger in XML than in the other mechanisms because every element has beginning and ending tags. For example, the VFP CUSTOMER sample table goes from 26,257 bytes as a DBF file to 40,586 as XML.
- While CURSORTOXML() is fast, XMLTOCURSOR(), which uses the XML DOM object to do the work, can be quite slow, especially with large amounts of data.
- XML standards are still evolving.

When to Use

XML is great for lots of things, including storing configuration settings, passing small amounts of data between application components, storing structured data in memo fields, etc. However, XML is ideally suited to n-tier applications because it's easily transported (either between components or across a wire) and converted to and from data sets such as VFP cursors. Using XML updategrams (and the newer diffgrams), you can limit the amount of data being transported. If you're starting new n-tier projects, this is clearly the data access mechanism to use.

CursorAdapter

One of the things you've likely noted is that each of the mechanisms we've looked at is totally different from the others. That means you have a new learning curve with each one, and converting an existing application from one mechanism to another is a non-trivial task.

In my opinion, CursorAdapter is one of the biggest new features in VFP 8. The reasons I think they're so cool are:

- They make it easy to use ODBC, ADO, or XML, even if you're not very familiar with these technologies.
- They provide a consistent interface to remote data regardless of the mechanism you choose.
- They make it easy to switch from one mechanism to another.

Here's an example of the last point. Suppose you have an application that uses ODBC with CursorAdapters to access SQL Server data, and for some reason you want to change to use ADO instead. All you need to do is change the DataSourceType of the CursorAdapters and change the connection to the backend database, and you're done. The rest of the components in the application neither know nor care about this; they still see the same cursor regardless of the mechanism used to access the data.

We'll take a close look at CursorAdapter in the Advanced document. However, in the meantime, here's an example (CursorAdapterExample.prg) that gets certain fields for Brazilian customers from the Customers table in the Northwind database. The cursor is updateable, so if you make changes in the cursor, close it, then run the program again, you'll see that your changes were saved to the backend.

```
local loCursor as CursorAdapter, ;
  laErrors[1]
loCursor = createobject('CursorAdapter')
with loCursor
  .Alias              = 'Customers'
  .DataSourceType     = 'ODBC'
  .DataSource         = sqlstringconnect('driver=SQL Server;' + ;
    'server=(local);database=Northwind;uid=sa;pwd=;trusted_connection=no')
  .SelectCmd          = "select CUSTOMERID, COMPANYNAME, CONTACTNAME " + ;
    "from CUSTOMERS where COUNTRY = 'Brazil'"
  .KeyFieldList       = 'CUSTOMERID'
  .Tables             = 'CUSTOMERS'
  .UpdatableFieldList = 'CUSTOMERID, COMPANYNAME, CONTACTNAME'
  .UpdateNameList     = 'CUSTOMERID CUSTOMERS.CUSTOMERID, ' + ;
    'COMPANYNAME CUSTOMERS.COMPANYNAME, CONTACTNAME CUSTOMERS.CONTACTNAME'
  if .CursorFill()
    browse
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif .CursorFill()
endwith
```

Advantages

The advantages of CursorAdapters are essentially the combination of those of all of the other technologies.

- Depending on how it's set up (if it's completely self-contained, for example), opening a cursor from a CursorAdapter subclass can almost be as easy as opening a remote view: you simply instantiate the subclass and call the CursorFill method (you could even call that from Init to make it a single-step operation).

- It's easier to convert an existing application to use CursorAdapters than to use cursors created with SPT.

- Like remote views, you can add a CursorAdapter to the DataEnvironment of a form or report and take advantage of the visual support the DE provides: dragging and dropping fields to automatically create controls, easily binding a control to a field by selecting it from a combobox in the Properties Window, and so on.

- It's easy to update the backend with changes: assuming the properties of the view have been set up properly, you simply call TABLEUPDATE().

- Because the result set created by a CursorAdapter is a VFP cursor, they can be used anywhere in VFP: in a grid, a report, processed in a SCAN loop, and so forth. This is true even if the data source comes from ADO and XML, because the CursorAdapter automatically takes care of conversion to and from a cursor for you.

- You have a lot of flexibility in data access, such as calling stored procedures or middle-tier objects.

- You can change the connection information on the fly as needed.

- You can change the SQL SELECT statement as needed.

- You don't need a DBC.

- They can work with ODBC, ADO, XML, or native tables, allowing you to take advantage of the benefits of any of these technologies or even switch technologies as required.

- Although you have to code for it yourself (this is discussed in more detail under Disadvantages), you have greater control over how updates are done. For example, you might use a SQL SELECT statement to create the cursor but call a stored procedure to update the backend tables.

- You can manage your own connections.

Disadvantages

There aren't a lot of disadvantages for CursorAdapters:

- You can't use a nice visual tool like the View Designer to create CursorAdapters, although the CursorAdapter Builder is a close second.

- Like other types of VFP cursors, you can't pass the result set outside the current data session. However, since CursorAdapters are really for the UI layer, that isn't much of an issue.

- They're awkward to use in reports; we'll discuss this in more detail in the Advanced document.

- Like all new technologies, there's a learning curve that must be mastered.

When to Use

Because CursorAdapters create VFP cursors, you won't likely do much with them in the middle tier of an n-tier application. However, in the UI layer, I see CursorAdapters replacing other techniques for all remote data access and even replacing Cursors in new applications that might one day be upsized.

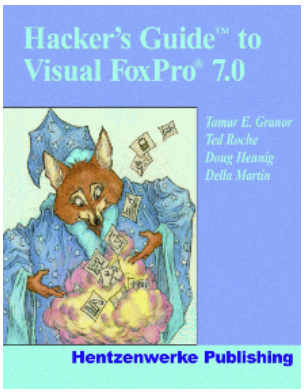
Summary

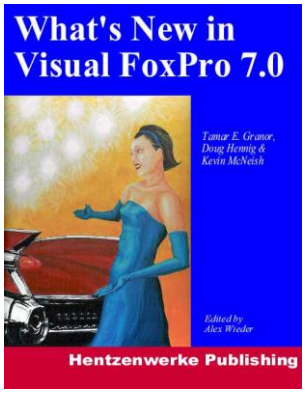
This document discussed the advantages and disadvantages of ODBC (whether remote views or SQL passthrough), ADO, and XML as means of accessing non-VFP data such as SQL Server or Oracle. As usual, which mechanism you should choose for a particular application depends on many factors. However, using the new CursorAdapter technology in VFP 8 makes the transition to remote data access easier, and makes it easier to switch between mechanisms should that need arise.

In the "Data Strategies in VFP: Advanced" document, we'll discuss the CursorAdapter class in detail, looking at specifics of accessing native data or non-VFP data using ODBC, ADO, and XML. We'll also look at creating reusable data classes, and discuss how to use CursorAdapters in reports.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and co-author of the award-winning Stonefield Query. He is co-author (along with Tamar Granor, Ted Roche, and Della Martin) of "The Hacker's Guide to Visual FoxPro 7.0" and co-author (along with Tamar Granor and Kevin McNeish) of "What's New in Visual FoxPro 7.0", both from Hentzenwerke Publishing, and author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's Pros Talk Visual FoxPro series. He writes the monthly "Reusable Tools" column in FoxTalk. He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals", both from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP).





Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341 Fax: (306) 586-5080
Email: dhennig@stonefield.com
Web: www.stonefield.com