Implementing Offline Views

By Doug Hennig

Introduction

The introduction of updateable views in Visual FoxPro 3 provided new capabilities for FoxPro applications. A new feature of VFP 5 extends the capabilities of updatable views even further: you can now take any view offline. An offline view is "disconnected" from the source of its data so data entry can continue while the tables the view is based on are unavailable. Sometime later, you can update the source tables by "reconnecting" the view to them. While you could do this type of thing in VFP 3 yourself, it required a lot of coordinating and some slick code. VFP 5 makes it very simple.

This document will look at the mechanics of offline views: how you take a view offline, considerations about working with offline views, updating the source tables, and taking a view back online. We'll also look at some uses of offline views: obvious ones like remote applications which allow querying and data entry on the road or branch offices, but also a less-obvious but equally interesting one, that of permitting data entry to carry on while table maintenance (reindexing, updating structures, packing, backing up, etc.) is being done.

The Basics of Offline Views

When you take a view offline, VFP creates a table containing all the records in the view. This table can be copied to a laptop or another system, and can be used for both querying or data entry. When you open the view, VFP actually opens this table rather than performing the normal SQL SELECT that occurs when you open an online view. Any changes made to the offline view (additions, edit, or deletions) are made in this table and are not written back to the source tables the view is based on.

In addition to the DBF file (and FPT file if the view has any Memo or General fields), VFP creates TBF and TDX files (and a TPT file if necessary for Memo or General fields) that it uses as a persistent table buffer for the DBF file. Although you might suspect these files are really table, index, and memo files with different extensions, you can't open the TBF table in VFP; trying to do so gives a "cannot open persistent table buffer directly" error. Doing some poking around, I found the TBF file contains copies of any records of the view that are modified, added, or deleted. However, the structure of this file is somewhat different that a normal DBF, so it really can't be used for anything directly.

There are two things you can do with the offline view when you're finished with it:

- Update the source tables the view is based on.
- Discard all the changes in the view, leaving the source tables unaltered.

Taking a View Offline

Taking a view offline is simple: ensure the database containing the view definition is the current database and use the new **createoffline()** function. It has the following syntax:

```
createoffline(<ViewName> [, <Path>])
```

<ViewName> is the name of view to take offline. Any view can be taken offline; you don't have to specially define a view as "offline-able". The optional <Path> is the directory and filename for the DBF that gets created. If you don't specify <Path>, VFP will create a table with the same name as the view (for example, if the view is called LV_CUSTOMER, the table will be named LV_CUSTOMER.DBF) in the current directory. Note the VFP documentation is incorrect regarding this parameter; it states you can specify just a directory for the table to go in, but in fact you need to specify the filename too (although an extension isn't required).

createoffline() returns .T. if the view is successfully taken offline. It'll return .F. if the view is already offline, if you don't specify a filename if you use the second parameter, if the view table or persistent table buffer files already exist (hopefully, this behavior will change in a future release; in my opinion, it should respect the setting of **set safety**), or if the source tables can't be opened.

Here's an example of this function:

```
llSuccess = createoffline('LV_CUSTOMER', 'OFFLINE\LV_CUSTOMER')
```

This example attempts to take the LV_CUSTOMER view offline and create a table called LV_CUSTOMER.DBF in the OFFLINE subdirectory of the current directory.

Here are some other notes about taking a view offline:

- **createoffline()** opens the tables the view is based on but not the view itself.
- **dbgetprop(<ViewName>, 'View', 'Offline')** returns .T. if a view is offline.
- VFP 5.0 (including 5.0a) has a bug: if **createoffline()** returns .F., indicating the view couldn't be taken offline for some reason, **dbgetprop(<ViewName>, 'View', 'Offline')** unexpectedly returns .T. Using **dropoffline()** (which we'll discuss later) doesn't work; it returns .F. indicating the view couldn't be taken back online, and **dbgetprop(<ViewName>, 'View', 'Offline')** continues to return .T. This really isn't an offline view, though; trying to open the view in online or admin mode (which we'll discuss in a moment) will give an "object is not an offline view" error. Fortunately, if you resolve the problem that caused **createoffline()** to fail, you can use it again to take the view offline.

Working with an Offline View

You open an offline view with the **use** command just as you do with an online view. However, as I noted earlier, VFP actually opens the table representing the offline view rather than using a SQL SELECT to create a cursor from the source tables.

Working with an offline view has several considerations:

- Changes made to the offline view (additions, edit, or deletions) are made in the view table and are not written back to the source tables the view is based on. This means someone opening a source table won't see the changes made in the offline view.

- The opposite is also true: you can't refresh the contents of an offline view from the source tables.

- The view table acts like an ordinary table as far as buffering goes. **`getfldstate()`** and **`getnextmodified()`** reflect the status of the view's buffer. **`tableupdate()`** writes changes in the view's buffer back to the view table and **`tablerevert()`** cancels changes. None of these functions has anything to do with the source tables.

- You can't modify the structure of the view table and you can't use the **`create trigger`**, **`insert`**, **`pack`**, or **`zap`** commands (although you can use **`delete all`**).

- Like an online view, an offline view automatically uses buffering, either row or table, so you'll need to have **`multilocks`** set on. You'll get an error message if you try to turn buffering off.

- Unlike an online view, opening an offline view doesn't automatically open the source tables the view is based on since the offline view is "unhooked" from these tables. This means you don't need to provide copies of the source tables (which might be very large or be non-VFP data sources) when you give someone the data for an offline view. It also means you can perform some types of maintenance on the source tables, such as recreating indexes, even while multiple users have the offline view open. We'll see examples of this later.

- Rules and triggers associated with the source tables don't fire when data entry is done in the offline view, only when you update the source tables (which we'll discuss in a moment). This means that unless you define rules for the offline view (using **`dbsetprop()`**) , it's possible bad data will get into the offline view and not be trapped until much later (when the update of the source tables is done).

- If you're using "surrogate" keys (sequential keys assigned by the system rather than the user), you'll need to assign key values as view records are added. The most common way of assigning surrogate keys is to use as the default value for the primary key field a call to a NEXTID program that grabs the next key value for this table from a table of key values. If you use this scheme, you'll need to use **`dbsetprop()`** to set the DefaultValue property of the field in the offline view so records added to the offline view get assigned key values.

- One complication with assigning surrogate keys: if offline views are used to permit data entry at remote locations (such as on a user's laptop), you'll need to assign a block of key values to each location so you don't have overlapping values. Alternatively, you could use a key value which is a combination of the user or location name and a sequential value, so it doesn't matter if two locations use the same sequential values.

- If an offline view is used for data entry at a remote location, you need to send the offline view table (including FPT), persistent table buffer files, and the database container (DBC, DCX, and DCT) to that location.

If you need to determine which records in a view were modified since the view was taken offline, use the **admin** clause of the **use** command. This requires exclusive access to the view table or you'll get an error message. Here's an example of this command:

```
use LV_CUSTOMER admin exclusive
```

A view opened in admin mode automatically uses table rather than row buffering. Trying to change the buffering mode for the table results in an error message. Once the view is open in admin mode, you can use **getnextmodified()** and **getfldstate()** to determine which records and which fields in those records were modified. Interestingly, opening an offline view in admin mode doesn't automatically open the source tables, so VFP must be using the TBF file discussed earlier to determine which records were changed. You can't update the source tables with changes made in the view when the view is open in admin mode; see the next section for information on updating source tables. However, you *can* use **tablerevert()** to back out any changes made in the offline view.

Updating Source Tables

When you're ready to update the source tables with changes made in the offline view, use the **online** clause of the **use** command to open the view in a "hooked" mode to the source table, then use **tableupdate().**

Opening a view in online mode requires exclusive access to the view table or you'll get an error message. This doesn't take the view online; it simply opens the view in preparation for updating the source tables. Here's an example of this command:

```
use LV_CUSTOMER online exclusive
```

As with admin mode, a view opened in online mode automatically uses table rather than row buffering. Opening an offline view in online mode doesn't automatically open the source tables until you issue a **tableupdate()** command.

After opening the view in online mode, use **tableupdate()** to update the source tables from changes in the offline view. This is the exact same process as updating from a table buffered table or online view: if conflicts are detected or rules (field or table validation, primary or candidate key, or triggers) are violated, **tableupdate()** will fail and you can use **getnextmodified()** and **getfldstate()** to determine which records and which fields in those records were modified and handle the conflicts or rule violations.

Updating the source tables from the offline view doesn't take the view back online, nor does it refresh the view with data from the source tables (for example, with changes made by other offline views or in the source tables directly). We'll see how to do these tasks in a moment.

Taking a View Back Online

The **dropoffline()** function takes a view back online. Here's the syntax:

```
dropoffline(<ViewName>)
```

It deletes the view table created when the view was taken offline and the persistent table buffering files (TBF, TDX, and TPT). Any changes made in the offline view that haven't been written back to the source tables are discarded. **dropoffline()** returns .T. if the view is successfully taken back online. It'll return .F. if the view is open at any workstation or if the view is already online.

Here's an example of this function:

```
llSuccess = dropoffline('LV_CUSTOMER')
```

Refreshing an Offline View

Although the help topic for **createoffline()** seems to indicate that you can refresh an offline view from the source tables if you open the view in online mode, this is in fact not true. You can't refresh an offline view from the source data at all; using the **requery()** or **refresh()** commands against an offline view, even if it's open in admin or online mode, give an "invalid operation for offline view" error. The only way to refresh the view is to take it back online (using **dropoffline()** after updating or discarding the changes) then take it offline again.

Uses for Offline Views

The most obvious use of offline views is to allow remote querying and data entry. However, another important reason for unhooking a view from the source tables is to permit maintenance tasks on the source tables, which normally requires exclusive access to the tables, while user querying and data entry continues. We'll look at a few scenarios where offline views make this much easier.

Mobile Querying and Data Entry

The typical situation for this scenario is a salesperson on road visiting customers. In the old days, if the salesperson wanted to record a new order for a customer, there were a couple of choices:

- They had to dial into the office system and run an order entry application at modem speeds.
- They had a data entry program on their laptop. This required a fairly sophisticated program at the office to consolidate information when the salesperson uploaded their orders.

With offline views, the second choice is much easier to implement. The salesperson can use the same order entry application used at their office (or perhaps a subset of it) but with offline copies of views for the customer, order, order detail, and product tables. When the salesperson returns to the office, the offline view files are copied from the laptop to the server and taken back online to update the source tables.

What if each salesperson needs just their own customers or products in their views; in other words, how do you deal with parameterized views? The problem is that when the view is taken offline, you have to specify what the parameters are, so every salesperson would get the same data. The solution is to take the view offline, specifying the parameters for a particular salesperson, and copy the view files to that salesperson's laptop. You would then take the view back online, then take it offline again for the next salesperson, specifying their parameters. The view must be left offline or take offline just before updating from each salesperson's data.

A minor complication is that if customer balances need to be updated, collisions need to be handled in a different way than just overwriting the data or rejecting the update. For

example, in the salesperson's view, the customer may have ordered $100.00 of products, but meanwhile, in the office's view, the customer paid a $50.00 invoice. The update routine needs to handle this type of update by examining the former and new values in the offline view and apply the difference to the value in the table. The product quantity on-hand needs to be updated in a similar manner since there could have been other orders for each product.

Multiple Locations

Several years ago, I wrote an application for a client that had twelve offices. Each office had their own copy of data tables, but the head office's files had to contain a complete set of data from each office as well as data maintained by the head office themselves. Once a month, each office sent their files to the head office and a complex consolidation program brought all the changes from all the offices together. This was not an easy program to write; it had to look for additions, deletions, and changes in each location's set of files and apply those changes to the head office files.

This situation would be *much* easier to deal with in VFP 5 using offline views. Here's the scheme I'd use if I was creating this application today:

- Each office would have a copy of the database and the offline view tables, and would do data entry into the offline views. The offline view tables would be in the same directory as the database to keep things simple.

- As often as necessary, each office sends their offline view files to head office. These files are kept in separate subdirectories, one for each office.

- One at a time, each office's offline view files are copied over the head office's view files. The view is then opened in online mode and **`tableupdate()`** is used to apply any changes made by that office to the source tables. The offline view files in that office's subdirectory are then deleted in case the process is run a second time.

- To refresh the branch office views, the offline view is dropped, then taken back offline, and the resulting offline view files shipped back to each office. Of course, this means that data entry at the offices cannot take place between the time that their files are shipped to head office and the refreshed files come back. However, since this process could be automated to run at night, this isn't much of a problem.

The MULTIPLE directory created when you unzip the source code files has some code that demonstrates this technique. If VFP isn't installed in a directory called VFP5 on the same drive as these files are installed, edit SETUP.PRG and change the path defined in lcDataDir as necessary. Run SETUP.PRG to create a copy of the VFP TESTDATA database in the MULTIPLE directory and copies of it in two subdirectories: OFFICE1 and OFFICE2. A new view, LV_CUSTOMER, is defined and taken offline.

After running SETUP, run NEWRECS.PRG. This will add new records in the offline views for the head office (CUST_ID 000), OFFICE1 (CUST_ID 111), and OFFICE2 (CUST_ID 222). After adding each record, a browse window is displayed so you can see the new records. After NEWRECS has finished, run CONSOLID.PRG. This program consolidates the changes made in each office into the head office CUSTOMER tables and displays a browse window of this table so you can see all changes.

Source Table Maintenance

If you have an application must be up all the time or if the tables are so large that the application can't be down for as long as it takes to recreate indexes, you have a problem: what happens if the server or a workstation crashes, corrupting the indexes and requiring them to be recreated?

Here's a good reason to have your forms and reports based on views rather than tables. You can take a view offline, which unhooks it from the source tables, and then recreate the indexes while users continue to query or perform data entry against the offline view. Once the reindexing has finished, you can update the source tables from the offline view. The only downtime are the brief periods required to take the view offline and later update the source tables and take the view back online. In fact, if you need to recreate indexes on a regular basis, you might consider using views that are *normally* offline, and only taken online long enough to refresh them periodically.

A minor complication, of course, is how do you close the view on each workstation long enough to take the view offline (and later back online) without calling up each user and telling them to get out of their data entry form for a few moments? The approach I came up with uses a "semaphore" scheme: each data entry form watches for a signal that maintenance is required. When this signal is received, the form is closed and then the application looks for a second signal that the view has changed state (either been taken offline or back online) and data entry can continue. Until the second signal is received, the user can't access the data entry form or any reports using the view.

The basis for watching for a signal is a timer object. There are actually three timers involved:

- A timer on the form used to do the reindexing waits until all users have closed the view. It then takes the view offline or online (depending on whether the reindexing is about to start or has been completed) and signals that to an application-level timer.

- A timer on data entry forms looks for a signal that the view must change state. When this signal is received, the form is closed and an application-level timer is started.

- The application-level timer, which is only enabled while a user is waiting for a view to change state, watches for a signal from the reindex form that the view's state has changed so data entry and querying can continue.

The files in the REINDEX directory created when you unzip the source code files demonstrates this scheme. Here are the various pieces:

- SYSINFO.DBF is a table that contains a logical CHNGSTATE (an abbreviation for ChangeState) field. When this field, which normally contains .F., is set to .T., it signals to all pieces that the state of view must be changed. This table is open at all times in the application.

- Any function in the application menu that needs access to the view has a Skip For SYSINFO.CHNGSTATE so the user cannot access that function while the state of the view is being changed.

- Any data entry form using the view has a timer object with code in its Timer() event to close the form and enable an application-level timer if SYSINFO.CHNGSTATE becomes .T. Here's the code from the timer object in the CUSTOMER form (in this code, oTimer is the application-level timer):

```
if SYSINFO.CHNGSTATE
  messagebox('Closing form to perform maintenance. ' + ;
    'The form will automatically reopen in just a ' + ;
    'moment.', 48)
  oTimer.Enabled = .T.
  Thisform.Release()
endif SYSINFO.CHNGSTATE
```

- At application startup, a timer object is instantiated but disabled. This timer, which is only enabled when a form is forced to close because a view is changing state, has code in its Timer() event that looks for SYSINFO.CHNGSTATE to become .F. so it can reopen any form that had to close while the view was changing state. Here's the code from the MyAppTimer class (in MYAPP.VCX) which is instantiated as oTimer:

```
if not SYSINFO.CHNGSTATE
  This.Enabled = .F.
  do form CUSTOMER
endif not SYSINFO.CHNGSTATE
```

- Of course, this code is hard-coded to open the CUSTOMER form because it's specific to this sample. In a real application, you'd likely have each form register itself with a manager object before closing down, and the manager object would be responsible for starting each form back up again.

- A form used to reindex tables sets SYSINFO.CHNGSTATE to .T. and starts a timer when the user indicates that reindexing should begin or has been completed. As we saw a moment ago, setting SYSINFO.CHNGSTATE informs any open form on any workstation that it needs to start the application-wide timer and then close itself. The reindex form timer checks periodically that all users have closed the view by trying to get exclusive access to it. Once this stage has been reached, the view is either taken offline or is taken back online (updating the source tables first) and SYSINFO.CHNGSTATE is set back to .F. (signalling to the application-wide timer that data entry can continue). Here's the code from the Timer() event of the timer object on the REINDEX form:

```
with Thisform

* Disable the timer (so it doesn't fire while we
* execute this code) and see if we can open the view
* exclusively.

  This.Enabled = .F.
  .CloseTables()
  select 0
  use LV_CUSTOMER exclusive
  do case

* We can open the view, so close it again, disable
* ourselves, and switch the state of the view.

    case used('LV_CUSTOMER')
      .CloseTables()
      .SwitchState()

* If we're still trying, see if we've exceeded the
* maximum amount of time to spend trying. If so, cancel
* the process.

    case seconds() - .nStartTime > .nMaxTime
```

```
      .cmdOffline.Click()

* Reenable the timer.

    otherwise
      This.Enabled = .T.
  endcase
endwith
```

If VFP isn't installed in a directory called VFP5 on the same drive as these files are installed, edit SETUP.PRG and change the path defined in lcDataDir as necessary. Run SETUP.PRG to create a copy of the VFP TESTDATA database in the DATA subdirectory. A new view, LV_CUSTOMER, is defined and SYSINFO.DBF is created.

After running SETUP, run MYAPP.PRG. Bring up an instance of the customer form (notice the form caption indicates the view is online) and then choose *Reindex Tables* from the menu. The Reindex Tables form shows a red light and indicates that reindexing cannot proceed because the view is online (the Reindex button is also disabled). Click the *Go Offline* button. Notice the light changes to yellow and indicates that we're waiting for the view to go offline. Also, the *Go Online* button's Caption changes to Cancel; this allows you to cancel the process while waiting. After five seconds (the Interval for the timer on the customer form), a dialog box will be displayed indicating that the customer form is being closed. After it's closed, the light on the Reindex Form turns green and indicates that reindexing can proceed. Also, the *Cancel* button's Caption changes to Go Online, which allows you to take the view back online, and the Reindex button becomes enabled. Immediately, the customer form opens automatically, but notice the caption now indicates the view is offline. Click the Reindex button to reindex the customer table. Edit a record in the customer form to show that data entry can continue. Now click on the *Go Online* button and notice the process repeat itself: the customer form closes, the Reindex Tables form indicates the view is back online, and the customer form reopens with the view online. Notice the changes you made to the offline view are now reflected in the online view.

You're probably wondering if updating the structure of the source tables is possible while the view is offline. The answer is: maybe. VFP gets very unhappy (and consequently, so do you <g>) if the structure of a table is changed and any views based on that table are affected. For example, if LV_CUSTOMER is a view on CUSTOMER defined using SELECT *, adding or removing a field from the table invalidates the view definition; you'll get the error "base table fields have been changed and no longer match view fields" and the view cannot be opened. (Interestingly, if the view is offline when this change is made, you can work with the view and even open it in online mode and update the source table. However, as soon as you take the view back online and try to open it, you'll get this error).

If the view was defined by explicitly listing each field (such as SELECT CUST_ID, COMPANY, CONTACT …), fields can be added to the table without affecting the view (although the view of course must be redefined if you want to be able to enter values into the new fields through the view). Removing a field from the table or renaming a field results in a "SQL column <fieldname> not found" error and the view can't be opened if it's an online view. If the view is offline, it can be opened but **`tableupdate()`** will fail because of the missing column.

The moral of this story is: depending on how your view is defined and what changes are being made to the table structure, you *might* be able to get away with altering a table

structure while data entry continues against an offline view of the table. You should redefine the view after altering the table to be safe.

Backing up is another maintenance task can that be done while the source tables are unhooked from the offline views.

Conclusion

Offline views extend the capabilities of updateable views, allowing you to add features to your applications that were difficult to implement in VFP 3. Try out the sample code accompanying this document, and start thinking of ways you can use offline views in your own applications.

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
Internet: dhennig@stonefield.com
World Wide Web: www.stonefield.com