Session X

Tables vs. Views

Whil Hentzen

Hentzenwerke Corporation

735 North Water Street

Milwaukee WI 53202-4104

414.224.7654

whil@hentzenwerke.com

In pre-VFP days, developers divided into two camps. On the one side, there were those who advocated direct editing of tables. On the other side were those who argued for the use of an intermediary for editing - using memory variables for editing and then committing the changes to the table once finished.

Those days are pretty much gone, and good thing, too, because there's a new point of contention - whether or not to use VFP's views in lieu of table based data handling. This session will examine the mechanisms available for both and then discuss some pros and cons to each.

Note: This session addresses local views as an alternative to table-based applications.

Tables - the old ways

Let's revisit how data handling was performed in the pre-VFP days in order to give us some common ground and a bit of perspective.

Direct Editing

In the direct table editing approach, the table would be opened in the screen's setup code and then the user would move to a record via screen navigation controls. Upon commencement of editing, the record would be locked and all other users would be prohibited from accessing it in a non-read-only manner.

Changes would be saved when the user left a field or moved off the record completely.

Simple, but not friendly in terms of multi-user conflict resolution.

Memory Variables

Now let's examine the editing memory variables approach.

In a screen's setup code, the table would be opened so that it was available for use. Then the **scatter memo memvar** command was used to create a series of memory variables that contained the data of the current record. These memory variables generally mapped directly to screen objects with the same name; in some cases, there was a 'translation' program that converted a value in a table to a value that was displayed on the screen. Foreign keys used for lookups and abbreviations that mapped to long captions for radio buttons are two typical examples.

The user could make changes to the values on the screen for as long as they desired without having to worry about conflicts with other users. The user would commit their changes either explicitly, via a "Save" button of some sort, or implicitly, whereby the behind-the-scenes code would perform the same work automatically whenever the user left a field or moved off the record. In both cases, the **gather memo memvar** command would be used. The record would be locked only for the brief instant that the **gather** was being performed.

Conflict Resolution

There was no conflict resolution with direct editing. Only one user had a whack at the data at any one time. So let's move on. Since multiple users could perform **scatter** commands and each edit their own copy of the data, it was possible for User A to change the value of a field, only to have that change be overwritten by User B's change moments later.

When this situation was not acceptable (in many environments, this situation either arose infrequently enough so as not to be an issue, or wasn't a concern at all), a more complex mechanism could be set up. Instead of using a single set of memory variables, data from the table was moved into an array that stored the original value in the table as well as the value being edited by the user.

Upon a commit, the two columns in the array would be compared with the data currently in the table (so as to detect changes made in the interim), and the user was warned of the differences.

Buffering

Everything Changes

Now that you've remembered the issues with editing memvars, time to forget. VFP's buffering functionality means that we no longer edit memvars - we edit the tables directly. Each field on a form is mapped directly to a field in the database. Goodbye to scatter and gather! Or so it seems…

Nothing Changes

We needed to review the techniques used with scatter and gather because it's the "deja vu all over again." To be sure, we'll never type "scatter" or "gather" again, but that doesn't mean the intermediary concept we were using has been cast aside - the only difference is that VFP handles the intermediary for us!

The Essentials

The Database Container

Forms and Data Environments

Buffering Settings

Mapping Fields to Controls

Commits and Undos

Implicit: moving off the record

Explicit: tableupdate(), tablerevert()

Tableupdate() can fail

Creating and Installing Views

The View Designer

The View Designer operates just like the Query Designer, which is good since a View is pretty much just an updateable cursor. The primary difference is an additional tab that allows you to specify how the update will work.

Essentials

Unlike a table, a View must reside in a database. Thus, a database must be open to create a View.

A View can access non-VFP data - including DBF data from older applications.

A View can represent a non-normalized join of multiple tables.

A View can represent a pre-filtered set of data.

A View’s “filter” can be user-driven (not “user-defined”)

Update Criteria

The Table drop down defines which tables can accept changes.

Send SQL Updates indicates whether changes are actually sent to tables.

The Field name list box: shows which fields are key fields, and which are updateable

Note key issues with multi-table Views.

- In parent, the key is obvious (the PK.)
- In child, the key is the child PK because that's used to move changes to the table from the view
- Tthe FK in the child is simply used to link the tables in the view

Reset Key and Update All buttons are used to “automate” the Field name list box.

Trick: help is available via right context menu

Tables Or Views - $64 or so

Tables

“We know how to do this.”

Existing functionality - little to create.

Indexes, filtering, searching and other familiar mechanisms.

Views

Location of data - Non VFP, Remote, Offline.

Set-based, not entire table - affects functionality and perception.

Only one index.

Performance depends on amount of data in View.

Resolving conflicts

Primary keys