Data Handling Issues, Part I

Doug Hennig

The ability to handle multiple sets of data is a frequent requirement in business applications. So is the management of primary key values for tables. In this first of a two-part article, Doug takes on these data handling issues and looks at the best way to deal them.

FoxPro is a database management system, so data handling is the thing it does best. However, many issues related to data handling can complicate even the simplest application. The next two columns will look at data handling issues that frequently arise. In this issue, we'll explore some strategies for handling multiple sets of data (such as test and production versions) in FoxPro 2.x and VFP, and look at several problems that come up when you use primary keys and how to solve them.

Handling Multiple Sets of Data

The idea of multiple sets of data comes up frequently in database applications. For example, accounting applications usually allow the user to manage more than one company by having each company's data stored in a separate set of tables. Even simpler applications benefit from having multiple data sets by providing test and production versions of the tables. This way, inexperienced users can learn how to use the system or new features can be tested against test data without worrying about ruining production data.

Multiple data sets are handled in one of two ways: the tables exist in a single directory, with different table names for different sets, or the tables in different sets have the same names but are located in different directories. An example of the former method is an accounting system that has table names such as GLACCT<nn> and ARCUST<nn>, where <nn> is a two-digit value representing a data set number. The disadvantage of this mechanism is that the directory will quickly accumulate a lot of files, which makes managing the files more complex and caused file searches (in DOS, anyway) to dramatically slow down. As a result, the latter mechanism is preferred.

Working with multiple data sets is relatively easy. Usually, you'll have a set of tables common to the application regardless of the data set (for example, security or application configuration tables) and a set of tables specific to each data set. The common tables might be in the same directory as the application or perhaps in a DATA or COMMON subdirectory, while the data set tables will exist in one or more separate directories. In a simple case, such as test and production data, the directory names might be hard-coded in the application, such as DATA and PRODUCTION. In more complex applications, such as multi-company data sets, you might have a DATASET table (in the common set) with at least two fields: the name of the company and the directory where the company's data set is located.

Here's how selecting a data set works in general:

- Close any currently open tables that aren't in the common set.
- Decide which directory the desired data set is stored in. For example, you might have a menu item that switches between test and production data. For a multi-

company application, the user might select a company to work with from a dialog, and your program looks up the company name in the DATASET table to find the directory that company's data set is stored in.

- Open the tables in that directory.

In FoxPro 2.x, the specifics of opening the tables can be handling in a variety of ways. You can SET DEFAULT TO or SET PATH TO the specified directory and open the tables, your table opening routine can accept a directory parameter and open the tables in that directory, or you can use a data-driven approach. Using SET DEFAULT TO and SET PATH TO probably aren't the best way to go unless your application is very simple, because it makes locating other pieces of the application more complex. You can easily create a table opening routine that handles both common and data set tables. The following routine does just that (this example is for illustration purposes only; a real routine would include parameter and error checking).

```
parameters tcDirectory
if tcDirectory = 'COMMON'
  use SECURITY in 0
  use USERS in 0
  use CONFIG in 0
  use DATASET order COMPANY in 0
else
  if used('CUSTOMER')
    use in CUSTOMER
  endif used('CUSTOMER')
  use (tcDirectory + 'CUSTOMER') in 0
  if used('ORDERS')
    use (tcDirectory + 'ORDERS') in 0
  endif used('ORDERS')
  if used('ACCOUNTS')
    use (tcDirectory + 'ACCOUNTS') in 0
  endif used('ACCOUNTS')
endif tcDirectory = 'COMMON'
```

A data-driven table opening routine is a better idea. You create a table (perhaps called TABLMAST) containing the name of each table in your application and a flag indicating whether the table is common or not. The data-driven routine opens each table defined in TABLMAST rather than hard-coding the names of the tables to open. This routine is called OPENDBFS.PRG on the source code disk.

```
parameters tcDirectory
private lcTable
select TABLMAST
scan
  lcTable = trim(TABLMAST.TABLE)
  do case
    case tcDirectory = 'COMMON' and TABLMAST.COMMON
      use (lcTable) in 0
    case tcDirectory <> 'COMMON' and ;
      not TABLMAST.COMMON
      if used(lcTable)
        use in (lcTable)
      endif used(lcTable)
      use (tcDirectory + trim(TABLMAST.TABLE)) in 0
  endcase
endscan
```

At application startup, your program would use the following code to open the common tables:

```
do OpenDBFS with 'COMMON'
```

If you allow the user to choose a company to work with, the following code would create an array of companies defined in DATASET, ask the user to choose one (using a routine called GETCOMP.PRG, which isn't shown here), and then open the tables for the selected company:

```
select COMPANY ;
  from DATASET ;
  into array laComoany ;
  order by 1
lcCompany = GetComp(@laCompany)
if not empty(lcCompany)
  select DATASET
  seek upper(lcCompany)
  do OpenDBFS with trim(DIRECTORY)
endif not empty(lcCompany)
```

As usual, things aren't quite as simple in Visual FoxPro (VFP). The problem is that the database container (DBC) contains a hard-coded directory reference to each table, and each table has the location of the DBC it belongs to hard-coded in its DBF header. This means you can't have a single DBC for the tables in multiple data sets. While you might be tempted to use free tables in this case, you lose the advantages a DBC provides, including field and table validation, referential integrity, and transaction processing. Although you can write a program that changes the directory to a table in the DBC, that won't work in a multi-user environment since you change the directory to a table for all users when you do that.

The only realistic solution is to put a copy of the DBC and its tables in each data set directory, and then open the appropriate DBC when you want to work with a data set. "What about common tables?", I'm sure you're thinking. You'll need to create at least two DBCs for the application: one for the common tables and one for the data set tables. The common DBC and its tables go in your common table location.

Here's a VFP version of OpenDBFS called OpenData. It uses a data-driven approach because we can get a list of the tables for each database using ADBOBJECTS(). This program is on the source code disk.

```
lparameters tcDirectory
local lcDirectory, ;
  laTables[1], ;
  lnTables, ;
  lnI, ;
  lcTable
if tcDirectory = 'COMMON'
  open database COMMON
else
  if dbused('DATA')
    set database to DATA
    close database
  endif dbused('DATA')
  lcDirectory = alltrim(tcDirectory)
  lcDirectory = lcDirectory + ;
    iif(right(lcDirectory, 1) = '\', '', '\')
```

```
  open database (lcDirectory + 'DATA')
endif tcDirectory = 'COMMON'
lnTables = adbobjects(laTables, 'Table')
for lnI = 1 to lnTables
  use (laTables[lnI]) in 0
next lnI
```

See the README.TXT file on the source code disk for examples of using OpenData with the two data sets on the disk (in the COMPANY1 and COMPANY2 subdirectories).

An additional complication with VFP is that the DataEnvironment of forms and reports has a hard-coded reference to the DBC that each table in the DataEnvironment belongs to. Fortunately, the Database property of each Cursor object in the DataEnvironment can be changed at run-time. The following code can be placed in the BeforeOpenTables method of the DataEnvironment. It assumes the name of the directory for the database is stored in a global variable called gcDirectory, and changes the location of the database for every table not in the COMMON database to that directory. Of course, to avoid using global variables, you'd probably store the location of the current database in a property of your application class instead.

```
if type('gcDirectory') = 'C'
  lnEnv = amembers(laEnv, This, 2)
  for lnI = 1 to lnEnv
    oObject = evaluate('This.' + laEnv[lnI])
    if upper(oObject.BaseClass) = 'CURSOR' and ;
      not 'COMMON' $ upper(oObject.Database)
      lcDBC = oObject.Database
      oObject.Database = fullpath(gcDirectory + ;
        substr(lcDBC, rat('\', lcDBC)), curdir())
    endif upper(oObject.BaseClass) = 'CURSOR' ...
  next lnI
endif type('gcDirectory') = 'C'
```

The Employee form on the source code disk shows an example of using this mechanism. To test this from the Command window, initialize gcDirectory to either COMPANY1 or COMPANY2 (the two subdirectories containing data sets), and DO FORM EMPLOYEE. You'll see a different set of records for each directory.

Several maintenance issues are more complicated when you provide multiple data sets to an application. For example, your "recreate indexes" routine needs to recreate indexes for all data sets. When you install a new version of an application that has database changes from the existing version, you need to update the DBC and table structures for each data set. These aren't difficult issues, just ones you need to be prepared for.

Primary Key Issues in VFP

VFP makes ensuring records have a unique key value automatic—you simply define one of the tags as a primary index, and VFP will automatically prevent two records from having the same key value. Let's look at some issues related to primary keys.

The primary key for a table can be of any data type, but Integer and Character are the most common choices. I prefer Integer primary keys because:

- they only take four bytes
- you can have up to 2 billion keys (or 4 billion if you use negative values as well)
- no data conversion is required when incrementing the next available value

- SQL SELECT statements that join two tables perform fastest using Integer keys

Character keys have two advantages over Integer:

- You cannot easily concatenate two Integer values when creating compound primary keys (a compound key is one that consists of more than one field). However, painful experience has led me to avoid using compound keys whenever possible, so this isn't usually a concern.
- Combo boxes used to select a value from a related table are more complicated to work with when the primary key for the related table isn't Character.

There are two kinds of primary keys for a table: user-defined and system-defined (also known as "surrogate"). I prefer surrogate keys that the user never sees because it avoids all kinds of complications like cascading primary key changes (since the primary key never changes, there is no need to cascade it), especially using the current Referential Integrity Builder, which doesn't properly handle compound primary keys.

A simple routine such as the following can be used to assign the next available value to the primary key for a table. This routine, called NEXT_ID.PRG on the source code disk, works with both Character and Integer (or other numeric) keys. It looks up the next available key value for the specified table in a table called NEXTID, which consists of two fields: TABLE C(128) and ID I(4).

```
lparameters tcTable
local lnCurrSelect, ;
  llUsed, ;
  lnCurrReprocess, ;
  luKey, ;
  lcKey, ;
  lcField

* Save the current work area, open the NEXTID table
* (if necessary), and find the desired table. If it
* doesn't exist, create a record for it.

lnCurrSelect = select()
llUsed       = used('NEXTID')
if llUsed
  select NEXTID
  set order to TABLE
else
  select 0
  use NEXTID order TABLE again shared
endif llUsed
seek upper(tcTable)
if not found()
  insert into NEXTID values (tcTable, 0)
endif not found()

* Increment the next available ID.

lnCurrReprocess = set('REPROCESS')
set reprocess to 10 seconds
if rlock()
  replace ID with ID + 1
  luKey = ID
  unlock
endif rlock()
```

```
* Set the data type of the return value to match
* the data type of the primary key for the table.

lcKey = dbgetprop(tcTable, 'Table', 'PrimaryKey')
if not empty(lcKey)
  lcField = key(tagno(lcKey, tcTable, tcTable), ;
    tcTable)
  if type(tcTable + '.' + lcField) = 'C'
    luKey = str(luKey, fsize(lcField, tcTable))
  endif type(tcTable + '.' + lcField) = 'C'
endif not empty(lcKey)

* Cleanup and return.

set reprocess to lnCurrReprocess
if not llUsed
  use
endif not llUsed
select (lnCurrSelect)
return luKey
```

Where should the code in NEXT_ID go? The place that immediately comes to mind is in the stored procedures for the database. The advantage of putting the code there is that the database becomes self-contained; you don't need to ship a separate PRG to someone in order to use your database. However, since NEXT_ID is likely to be used in every database you create, this raises a maintenance issue. What if you discover a bug in NEXT_ID or want to enhance its functionality? You'd have to change it in every database you ever created, including all of those at every client site. In addition, if you succumbed to the temptation to tweak the code slightly in different databases because you had different needs in each, you'll really have a tough time updating the code for each one. In my opinion, only database-specific code (such as validation and other business rules) belong in the stored procedures of a database. Generic routines such as NEXT_ID belong in stand-alone PRGs or in a library of routines (whether procedural or in a class library).

When should you call NEXT_ID to assign a surrogate key to a record? The obvious place is in the insert trigger for the table, but as you've probably discovered by now, VFP won't allow you to change the contents of any field in the current table in trigger code. You could call it in the Save code for your form, but that breaks encapsulation for the database—the key value won't be assigned properly when records are added any place other than the form. The correct place to assign the primary key is in the Default value for the primary key field by specifying NEXT_ID('<table>') as the expression, where <table> is the name of the table. Whenever a new record is added, VFP will call NEXT_ID to assign a default value to the primary key field.

What happens if the user adds a new record to the table, then cancels the addition? Using buffering and TABLEREVERT(), you can easily discard the added record, but the KEY field in NEXTID.DBF is still incremented. If you're using meaningless surrogate keys, this isn't a problem, but if the primary key is a check or invoice number, you might not want to "waste" a primary key value. To prevent this, you can have your forms work on a view of the table. Only when TABLEUPDATE() is used to write the new view record to the table is NEXT_ID called; if the user discards the new view record, the table isn't affected so the next available key value isn't incremented.

Many developers minimized the requirement to PACK a table in FoxPro 2.x applications by recycling deleted records. This simple yet powerful idea works as follows:

- When a record is deleted, blank the fields in the record (using BLANK in FoxPro 2.6 or by replacing each field with a blank value in prior versions). This causes the record to "float" to the top when a tag is active.
- When a new record is needed, first SET DELETED OFF (so FoxPro can "see" deleted records), then GO TOP (or LOCATE) to move to the first record in index order, and check to see if it's deleted. If so, RECALL the record. If not, there are no deleted records to recycle, so use INSERT INTO or APPEND BLANK to create a new record.

This mechanism doesn't work in VFP for one simple reason: you can't have two records with the same primary key value, even a blank value. Even deleted records are checked for primary key duplication, so as soon as you try to delete a record and blank its key value when another such record already exists, you'll get the dreaded "Uniqueness of index is violated" error.

There are two solutions to this problem. One, a technique pioneered by the late Tom Rettig, involves using a filter on the primary key of NOT DELETED(). With such a filter, VFP no longer checks deleted records for duplicate primary keys, so this mechanism works. Unfortunately, this filter can only be created visually using the Table Designer, not programmatically using INDEX ON or ALTER TABLE, so you cannot recreate this index at a client site if indexes need to be rebuilt. The other solution is to create a tag on DELETED() (which is a good idea anyway because it helps Rushmore do its job better) and use that tag to locate a deleted record to reuse. The logic behind this technique is now:

- When a record is deleted, don't blank the fields; just delete the record. This causes the record to "float" to the set of deleted records when the DELETED tag is active but keeps its primary key value so the record is unique.
- When a new record is needed, first SET DELETED OFF (so FoxPro can "see" deleted records), then set the order to the DELETED tag and go to the first record. If it's deleted, RECALL it and assign a new primary key (if desired). If not, there are no deleted records to recycle, so use INSERT INTO or APPEND BLANK to create a new record.

The following routine, called NEW_REC.PRG on the source code disk, can be used to recycle deleted records.

```
local lcCurrDelete, ;
  lcCurrOrder, ;
  llDone, ;
  lcBlank, ;
  laFields[1], ;
  lnFields, ;
  lnI, ;
  lcField, ;
  lcDefault

* Save the current DELETED setting and order and
* set them as we require.
```

```
lcCurrDelete = set('DELETED')
lcCurrOrder  = order()
set deleted off
set order to DELETED descending

* Go to the top of the file, which should be the
* first deleted record if there are any. Then
* process records until we get what we want.

locate
llDone = .F.
do while not llDone
  do case

* If this record is deleted and we can lock it, set
* each field to its default value, blank any fields
* we don't have a default for, and recall the
* record.

    case deleted() and rlock()
      lcBlank  = ''
      lnFields = afields(laFields)
      for lnI = 1 to lnFields
        lcField   = laFields(lnI, 1)
        lcDefault = laFields(lnI, 9)
        if empty(lcDefault)
          lcBlank = iif(empty(lcBlank), 'fields ', ;
            lcBlank + ',') + lcField
        else
          replace (lcField) with evaluate(lcDefault)
        endif empty(lcDefault)
      next lnI
      blank &lcBlank next 1
      recall
      llDone = .T.

* If this record is deleted but we can't locked it,
* try the next one.

    case deleted()
      skip

* If this record isn't deleted, there aren't any
* more, so just add a new one.

    otherwise
      append blank
      llDone = rlock()
  endcase
enddo while not llDone

* Clean up things we changed.

if lcCurrDelete = 'ON'
  set deleted on
endif lcCurrDelete = 'ON'
set order to (lcCurrOrder)
return
```

Conclusion

We looked at two aspects of data handling that frequently arise in applications: working with multiple data sets and primary keys issues. Next month, we'll look at more issues, including validation "gotchas", lookup tables, and how some new data dictionary features in VFP 5 will affect your applications.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Sask., Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Data Dictionary for FoxPro 2.x and Stonefield Database Toolkit for Visual FoxPro. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at user groups and regional conferences all over North America. CompuServe 75156,2326.