PINNACLE
FoxTalk
Solutions for Microsoft® FoxPro® and Visual FoxPro® Developers

A Case for Derivative Data

Peter de Valença

UNIX MAC DOS WIN 2.x 3.0 5.0 DOWNLOAD

Often, the data stored in an application isn't in a format that can be easily used by end-users in querying and reporting. In these cases, a second set of data in denormalized form is created. In this article, Peter describes a mechanism for determining when the original data and the derived data go out of sync.

WHEN a data model is complex, it's often difficult to provide access to that data to end-users in a form that they can easily understand. A solution to this problem is to derive a second set of data that's structured in a more comprehensible form. However, just as a man with one watch knows what time it is, but a man with two watches is never sure, keeping multiple data sets in sync poses certain problems.

Before I continue, I'll define a couple of terms. In searching for an expression that identifies data in a dataset that can be derived from other data in the same dataset at any time, I decided to introduce the expressions "derivative data" and "raw data." (For purposes of this article, a *dataset* is a set of databases that are logically all part of one datamodel. Most probably, but not necessarily, they're also physically gathered in the same spot.)

Raw data are the original data in a dataset. Derivative data (or "derived data") are in the same dataset and have been derived from the raw data. Moreover, ideally they shouldn't be out of sync with the raw data at any time.

Typically, the derivation is done with a procedure that takes as input a lot of data from several databases, performs one or more complex algorithms, and adds the resulting data to a database with a simpler physical structure. Reasons for doing so include: 1) performance, where the use of derivative data might speed things up, and 2) complexity, where some complex data transformations are needed at more spots in the application. Use of derivative data can simplify coding. A typical reason for *not* doing so is the frequent changing of original data. If the original data is changed too often, the performance could drop significantly, due to the need to refresh the derivative data.

The out-of-sync issue might pose practical problems. Immediate refresh (or re-derivation, or synchronization) of the derivative data whenever raw data changes is difficult to realize and will slow things down considerably. The alternative (and convenient) solution is to refresh the derivative data at crucial moments only.

From a database-management perspective, derivative data are redundant data. Codd's model of data normalization encourages us to banish redundant data. However, in a practical sense, they're not redundant, for they enable us to increase performance and simplify coding.

This article introduces a method for dealing with derivative data—this method will be referred to as the Derivative Data Method.

Requirements

Management of the derivative data in the dataset requires an administrative component and an operational component. The administrative component should know all the answers to all the questions that the operational component needs answered in order to perform actions.

The administration is best done by a dedicated module, which I'll name DERIVATIVE. In this article, it's referred to as the Derivative Module. Although it's tempting to also put the operational component in this module, it's better to have the elements of the operational component scattered over several other modules. (Read on for the argument.)

Let's presume that it's not the task of the Derivative Module to decide whether or not it's an appropriate and crucial moment to operate. Rather, the Derivative Module is invoked at appropriate places in other modules. If invoked, it's supposed to be a crucial moment to conditionally start an operation.

Items for operation are:

- Add certain derivative data if it's not yet present.
- Refresh certain derivative data if it's out of sync.

- Remove certain derivative data. (This might seem a weird operation, but it will prove to be a useful one.)

As a consequence of these demands, items for administration are:

- An ID for the derivative data.
- A Subset name to indicate a subset of that derivative data.
- Name and location of the database that contains the derivative data.
- The name and location of the "hookprogram" that will perform operations with regard to this derivative data.
- A date and time that mark when the derivative data was created or refreshed.
- A flag, date, and time that mark whether and since when the derivative data is out of sync.

So, the parameters of the Derivative Module should enable it to perform the following actions and tasks:

- Operation: Tell the module to derive certain data if it's not yet present.
- Operation: Tell the module to refresh certain derivative data if it's out of sync.
- Operation: Tell the module to remove certain or all derivative data if it's present.
- Administration: Set the out-of-sync flag, date, and time.

The hookprogram

The hookprogram is part of the operational component and specializes in certain derivative data. Typically, its functionality is described in the functional design of another module. This makes it more logical to put the hookprogram into that module, rather than in the Derivative Module.

The decision to permanently store the derived data—for improved performance or any other reason—makes it necessary to use the Derivative Module, but the code that creates the derivative data may remain in the "original" module. However, in order to be compatible with the Derivative Module, the hookprogram (which is in the original module) will have to conform to certain standards, as laid down by the Derivative Module.

Here are some requirements for the hookprogram:

- It can presume that no module other than the Derivative Module—not even a program in the same module—will invoke the hookprogram. (However, it should check for this anyway; see the next item.)
- The hookprogram should check for the correctness of the needed predeclared variables (which are declared in the Derivative Module and are discussed later).
- The flag checks aren't done by the hookprogram. Whenever the hookprogram is invoked, it should do its own job—the flags have been checked for in the Derivative Module.
- The hookprogram should return a result code of type Numerical, reflecting the state of the operation.

The hookmodule

I'll use the term "hookmodule" to identify the module that contains the hookprogram. Let's presume that each hookmodule is a separate app.

As there's no way of invoking a program in another app in a direct way, I need an alternative mechanism. (The intuitive "DO <hookprogram> IN <hookmodule> WITH <parameters>" won't do the job in the case of apps.) I need to call the app with a parameter that's understood by the app's main program. The parameter tells the main program that I want to contact a hookprogram that resides in that hookmodule. I could use the keyword "dvd" as a default, but let's not get too restrictive. There's plenty of room in the administrative department (a database, of course), so I'll also administer the keyword.

Predeclared variables

Other information is needed by the hookmodule's main program and the hookprogram. It might be tempting to pass that information with the help of additional parameters, and under normal circumstances this is good practice. However, it's likely that the hookmodule already has additional parameters with specific names and specific meanings and the alternative use of these parameters could be confusing. Therefore, it's better to let the Derivative Module put the information in variables with names that are understood by the hookmodule's main program and hookprogram.

- I need a variable that contains the name of the hookprogram; it's used by the hookmodule's main program.
- Another variable should contain a keyword that reflects the type of action. This variable is used by the hookprogram only.
- I can use an array for the additional information that the hookprogram needs to perform the action. The array's size and the datatypes of the elements can vary among hookprograms.

- I need a variable that contains a result code. It's initialized by the Derivative Module, changed to other codes by the hookmodule and hookprogram, and interpreted by the Derivative Module on return. Thus, the risk of failure is minimized.

Messages

While some idiosyncratic errors are best handled in the hookprogram, other errors can be dealt with by the Derivative Module, thus simplifying the coding of a hookprogram. That's why the result code that's returned to the Derivative Module should be numeric, rather than logical.

The result code

A public variable should be set to a number at various places, thus enabling the Derivative Module to interpret the status of its call:

- 0—It should be set to 0 in the hookprogram as soon as it's successfully done its job.
- 1—The Derivative Module should set it to 1 prior to invoking the hookmodule. A hookmodule that can't handle the keyword "dvd" also won't change the value, thus indicating that it was an inappropriate call.
- 2—It should be set to 2 in the hookmodule prior to invoking the hookprogram, thus indicating that the hookprogram doesn't exist.
- 3—It should be set to 3 at the top of the hookprogram, thus indicating a premature ending of the hookprogram due to an unexpected error.
- 4—It should be set to 4 in the hookprogram if the action keyword isn't yet supported by the hookprogram.
- 5—It should be set to 5 in the hookprogram if it thinks that the array is invalid in some way.

A negative number indicates failure. However, display of a message by the Derivative Module is inappropriate. Otherwise, a specific message will be displayed by the Derivative Module, unless it's 0.

The coding essentials

Here I declare four variables PUBLIC during initialization of the application:

```
* 1 - Name of hookprogram.
* 2 - Keyword for hookprogram.
* 3 - Additional info for hookprogram.
* 4 - Result code, or Status code.
PUBLIC QCdvdHook
PUBLIC QCdvdAction
PUBLIC ARRAY QAdvd[ 1 ]
PUBLIC QNdvdResult
*
QCdvdHook   = '?'
QCdvdAction = '?'
QNdvdResult = 0
```

The character "Q" is my indicator for public variables. Also, to further distinguish public variables from private variables, I write the first two characters in upper case. The character sequence "dvd" stands for "derivative data."

The administrative component of the Derivative Module will declare these variables prior to calling the module that's supposed to contain the hookprogram:

```
dimension QAdvd[ 2 ]
QCdvdHook   = 'dvd_insp'
QCdvdAction = 'refresh'
QAdvd[ 1 ]  = '1997'
QAdvd[ 2 ]  = '09'
QNdvdResult = 1
```

The next step—also done in the Derivative Module—is an activation of the hookprogram. Presume that the hookmodule's name is INSP_R.APP:

```
=insp_r( 'dvd' )        && This one ...
do insp_r with 'dvd'    && ... or this one.
```

Note that I'm not interested in the returned value. Its datatype is uncertain (for example, in the case of a misplaced call) and not needed, for the QNdvdResult variable is at our disposal for analysis.

The hookmodule's main program should have something like the following code:

```
parameter pAction
private cAction
*
cAction = lower( left( default( 'pAction', '' ), 4 ) )
*
do case
case cAction == 'dvd'
  *
  QNdvdResult = 2  && That is.. Hookprogram
                   &&   doesn't exist.
  =&QCdvdHook.()   && Invoke the hookprogram.
  RETURN           && Return whatever
                   &&   datatype you like.
endcase
```

The default() function is a UDF that will create a default if no parameter was passed. (See the sidebar "Set a Default.")

Note that the Derivative Module can display an appropriate message, based on the value of QNdvdResult, if the hookprogram couldn't be invoked.

The hookprogram also should check for the correctness of the needed predeclared variables:

```
private cAction, Lok
*
QNdvdResult = 3       && That is..
                      &&   Busy in hookprogram.
Lok         = .F.
cAction     = lower( left( QCdvdAction, 4 ) )
*
do case
case not inlist( cAction, "add", "refr" )
  *
  QNdvdResult = 4     && That is.. keyword
                      &&   not yet supported.
  RETURN
  *
case alen( QAdvd ) # 2          && Rudimentary check.
 *
  QNdvdResult = 5     && That is.. Invalid array.
  RETURN
  *
case not msg( 1860 )  && warning message?!
  QNdvdResult = -1
  RETURN
  *
case cAction = 'add'  && add
  ...
case cAction = 'refr' && refresh
  ...
endcase
*
if Lok
  QNdvdResult = 0     && That is.. Action
                      &&   successfully finished.
else
endif
*
RETURN
```

Most messages can be dealt with entirely by the Derivative Module, thus simplifying the coding of a hookprogram. A normal termination results in the return of the result code 0.

It must be simple to set the out-of-sync flag for certain derivative data. Suppose the ID is "insp" and you want to make all subsets out-of-sync (oos). You'd need only three parameters:

```
if <some condition>
  *
  * parms: action, id, subset
  *
  =derivate( "make_oos", "insp", "*" )
endif
```

The asterisk "*" serves as a wildcard character, thus indicating all subsets.

It must be almost as simple to conditionally refresh (or synchronize) the derivative data only at the time it's needed, rather than whenever it gets out of sync. So, the following code can be expected in another module, preceding the code that relies on the derivative data:

```
*   Check whether the derivate data is out of sync.
*   If it is, synchronize it for this period.
*
if derivate( 'is_oos?', 'insp', "199709" )
  *
  if not derivate( 'sync', 'insp', ;
                   "199709", "1997", "09" )
    *
    <do cleanup>
    RETURN  && .. for it is still out of sync.
  endif
endif
```

In this example, the ID's subset names reflect periods. Moreover, when told to synchronize, the hookprogram seems to need the year and month. They're passed as additional parameters and will be put into the QAdvd[] array by the Derivative Module prior to the call of the hookprogram.

So let's code . . .

And that's exactly what I've done. I've succeeded, but it was more difficult than I expected it to be. It took me several days to get things straight, and functionality changed several times. (I had to rewrite more than a few portions of this text to reflect those changes.) But now it's up and running, and the feeling is good.

You should try the accompanying code and example in this month's Subscriber Downloads at www.pinpub.com/foxtalk.

Conclusion

You might not be impressed by the idea at all at first glance, but think about it a little longer . . . Derivative data (that is, redundant data) is what Codd tried to put a spell on, and while most of us have saved "derivative data" for later use in harsh situations, very few of us have created an administrative system that tells us whether this data is out of sync and needs a refresh. I wonder what Codd would have to say about the Derivative Data Method. ▲

DOWNLOAD **04DEVAL.ZIP at www.pinpub.com/foxtalk**

Peter de Valença is a freelance software developer who works on projects for large companies and has specialized in FoxPro for the past five years. He has a Ph.D. in social psychology from the University of Amsterdam. Peter would like to solicit your feedback about this article. pvalenca@digiface.nl (preferred), pvalenca@compuserve.com.

Set a Default

The following code will conditionally set a default for a parameter:

```
FUNCTION default
  parameter pVar, pDefault
  if type( pVar ) == type( 'pDefault' )
    RETURN &pVar
  else
    RETURN pDefault
  endif
```

An example:

```
* x.prg
* sample PRG
parameters pWhat, pFnd, pFrom
private cWhat, nFnd, dFrom
cWhat = default( 'pWhat', 'disp' )
nFnd  = default( 'pFnd', 0 )
dFrom = default( 'pFrom', {1/1/80} )
? cWhat
? nFnd
? dFrom

do x with "XX", , {1/1/91}
 "XX"
0
{1/1/1991}

do x with , 34
 "disp"
34
{1/1/1980}
```