The Fox Hunt

By Whil Hentzen

One of my favorite limericks is the one about the pelican - “A marvelous bird is the pelican; his beak can hold more than his bellly can; he can store in his beak; food enough for a week; but I’ll be damned if I know how the hell he can.” Another marvelous bird is the overloaded lookup table - and that’s the topic for this and the next couple months. “Overloading” refers to the practice of using an item for more than one use - in this specific instance, using the same table structure to store more than one type of logical entity. An example of an overloaded table is one that stores both customers and suppliers, using a code in a field to distinguish between the two, instead of keeping customers in one table and suppliers in a second.

A lot of lookup tables have the same types of requirements - typically these include a key value, description, and a user-entered code. Some lookup tables only require a description. The user selects a value via a dropdown listbox or perhaps a picklist inside a Browse. When the user selects the desired value, the associated key value - not the description itself - is placed in the table. For example, in a user maintenance screen, you might want to select the permission level for the user from a dropdown listbox. These permission levels include “Developer”, “Supervisor”, “Manager”, “Super-User”, “User” and “Read-Only Access.” These descriptions would be placed in a table along with unique ID values, and when you selected a permission level for the user displayed on the screen, the ID value would be placed in the USER table, not the description itself.

There are other types of lookup tables, however, that typically will contain a user code in addition to the description. This user code is the value that the user types into a GET field instead of selecting an item from a dropdown or a picklist. Examples would be the entry of a state’s two character abbreviation - “NY” instead of “New York” - or the entry of a four digit diagnosis code - “9023” instead of “Introductory Physical Examination.” A user-entered code is employed for one of two reasons - either the number of potential items to be selected is too large for a dropdown, or the user can enter a short code via the keyboard faster than using a dropdown or picklist mechanism. When the user enters the code, the program searches the lookup table for the code and only allows the entry if the value is found in the table.

Robust data entry screens include two additions features with user-entered codes. First, the description associated with a valid user-entered code is automatically displayed next to the code on the screen. When the user enteres “WI” into a state user-entered code field, the full name, “Wisconsin”, is displayed next to the GET field. This is particularly handy when entering numeric codes so that the user can verify they didn’t juxtapose numbers or miss a character. The other nice feature is displaying a list of valid codes if the user enters a code that isn’t found in the lookup table.

To illustrate the concept, we’re going to use a “User Status” field in the above USER table as a user-entered lookup. The status values can include “A” (active), “I” (inactive), “F” (terminated), “T” (temporary), and “S” (suspended).

In a system of any appreciable size, it is obvious that you could have quite a number of lookup tables floating around. Eventually, you're going to run into problems with file handles - trying to open a couple dozen lookup tables - or with performance - having to open and close tables as needed. And both problems are unnecessary, since the volume of data in these tables is usually quite small.

Why not combine all of these small lookup tables into one big table, and use another field to identify which lookup "table" it belongs to? An overloaded lookup table would look like this:

The first colum identifies which "table" the lookup record belongs to. The second field, cCd, is the unique ID for that specific lookup value. This will be the foreign key in the other table. Note that these IDs are unique throughout the lookup table - you won't find the same ID in PERMLEV and STATUS. The cDe field is the Description that populates a dropdown or picklist, or is shown as a result of the entry of a user entered code. And the final field, cScreenVal, is the user-entered code or value that the user enters in a GET field. I call this field "Screen Val" so as not to confuse it with a "code" like a key value. Note that a lookup table doesn't have to have a value in the cScreenVal field.

Using a single DBF to hold all of our lookup tables has a number of advantages. First, since it's only one table, we can keep it open all of the time, thus enhancing performance. Second, once we've opened it once, we can open it additional times under other aliases. These additional instances do not take up file handles and, with 255 work areas, we've certainly got plenty of room for most systems.

Second, since all of the lookup tables inside this one DBF have the same field names, it is much easier to create generic code to handle lookup values. And, finally, once we've built a "lookup maintenance" screen, we don't have to build additional maintenance screens for each of these little lookup tables, nor do we have to spend a lot of time creating a populating multiple tables.

In our next column, we're going to discuss a generic lookup maintenance screen. As we build it, we're going to find that we'll want some additional features that aren't covered by the simple, four column table structure outlined above. For example, we're going to want to be able to restrict access to some lookup tables according to the permission level of the current user. Additionally, we'll need a user-friendly name for each lookup table, since the user may not quite relate to names like "PERMLEV", "ST", "DIAGCD", and "ASMTST." We'll give them the ability to choose from lookup table names like "Permission Level", "State", Diagnosis Code" and "Assessment Status."

Thus, the structure of our lookup table will look like this:

In future columns, we’ll cover the integration of lookups in our data entry screens using just a couple of data-driven UDFs. Stay tuned!