DBCX 2: The Quickening

Doug Hennig

DBCX is a public domain data dictionary extension utility that allows you to define extended properties for database objects. However, it suffers from a number of shortcomings, the biggest one being performance. DBCX 2 is a complete rewrite of DBCX that adds speed and flexibility to this powerful tool.

Early in the Visual FoxPro 3 beta, several prominent FoxPro developers realized the VFP database container didn't provide all the information typically found in a data dictionary (for example, while there's a Caption property for fields, that property doesn't exist for tables, views, or indexes), so a means of extending the DBC was required. They reasoned that it was better to create a public domain data dictionary extension utility that all developers could use rather than having many incompatible extensions, a problem that plagued FoxPro 2.6 add-ons. Thus, DBCX was born, the result of a collaboration between Flash Creative Management, MicroMega Systems, Neon Software (now F1 Technologies), and Stonefield Systems Group.

DBCX has been used directly in several third-party tools for VFP, including Stonefield Database Toolkit (SDT), Visual FoxExpress (VFE), FoxFire, and Visual Extend. It's also used indirectly by frameworks such as Visual MaxFrame and Codebook for Mere Mortals that have hooks to SDT or FoxFire features if they're available.

In the July 1997 issue of FoxTalk ("Using DBCX to Data-Drive Your Applications"), FoxTalk editor Whil Hentzen described the basics of DBCX and how to use it to create data-driven applications. I won't repeat the points covered in his article; I'll assume you're either somewhat familiar with DBCX because you have an add-on that uses it or you'll refer to Whil's article for background information (it's included in the ZIP file for this article available on the Subscriber Download Web site).

Shortcomings

While DBCX provided a common set of data dictionary extensions for VFP developers, there are several things about it that have always bothered those of us who use it extensively, primary me and Toni and Mike Feltman of F1 Technologies. In our defense, DBCX was designed and implemented when we were very new to VFP. Practical experience in the years since then has taught us all better ways to do things (who among us doesn't look back at their first application written in a new language and cringe at the way they did things back then?). Here are some of the more serious shortcomings in DBCX.

Synchronization

Extended properties for database objects are stored in meta data tables rather than in the DBC itself. The link between an object in the database and its record in a meta data table is provided with a DBCX ID. A unique ID is stored in the USER memo field of the DBC record, and the matching record in the meta data table has this same ID as its primary key. A DBCX method called DBGetDBCKey is used to find the object in the DBC and extract the ID from the USER memo. This ID is then passed to other methods (such as DBCXGetProp, which gets an extended property for an object) to find the matching meta data record. However, in the real world, things can happen to a database that cause the ID in the USER memo to get lost or point to the wrong record in the meta data tables. This means a significant amount of maintenance must be done to ensure these IDs are present and match the correct meta data record.

Performance

To get an extended property for an object, DBCX has to do the following: open the DBC as a table, find the record for the specified object, parse the USER memo to extract the ID, convert the ID (which is stored as a string) into a numeric, close the DBC table, and finally seek the ID in the meta data table. If this sounds like a lot of work, it is, and work corresponds to time in software.

An even worse performance drain is "validating" the meta data (ensuring that every database object has an ID and a matching record in the meta data tables, and that the extended properties for the meta data records contain the proper values). The DBCX Validate method processes one object in the database at a time. For example, to process a field object, Validate opens the corresponding table or view, gets

information about the field, and then closes the table or view. This means the table or view is opened and closed once for every field and index. Since opening and closing tables is one of the slowest things VFP can do, you can see that this can be a very time-consuming process. Put the database and tables on a LAN and it gets even worse. I know of several people whose databases are large enough that validation can take 45 minutes or more. Since validation is the cure to the synchronization problem mentioned earlier, this slow process is used a lot. To make matter worse, Validate processes an entire database at a time. Even if you just add a single field to a single table, Validate still has to process every object in the entire database.

Database-Only Orientation

Since each record in the meta data tables has a ID that references an object in a database, what about non-database objects such as free tables and calculated fields? Unfortunately, there's no way to support these in DBCX, meaning if you want meta data for these types of objects, you have to store it in a separate set of meta data tables and provide your own mechanism and toolset for access.

Knowledge of Details

The main DBCX class, MetaMgr, doesn't actually store any meta data itself. Instead, it's really just a manager of data dictionary extension managers. For example, the common meta data for a database is managed by a class called CdbkMgr (short for "Codebook manager") and the meta data SDT needs to do its tasks is managed by SDTMgr. Extension managers are registered with MetaMgr through a registry table called DBCXReg. The more managers registered, the more extended properties available for each database object.

One of the design goals for DBCX was the ability to work with extended properties without knowing which extension manager is responsible for it. Unfortunately, this goal was not achieved. To specify a property, you precede the name of the field in the meta data table with a prefix for the manager than maintains that meta data table. For example, to retrieve the extended InputMask property for a field, you specify CBmInFormat as the property. Thus, it's essential to have a list of the field names and managers handy when working with DBCX.

Missing Core Properties

CdbkMgr and its meta data table CdbkMeta were supposed to contain the common properties for all database objects, primarily structural information. However, some key properties were missed, such as the blocksize and code page for a table and the collate sequence for an index. Other common properties, such as caption and comments, were also omitted. As a result, different third-party tools that needed these properties contained their own versions of them. Thus, both SDT and VFE have a caption property for indexes and tables. If you use both tools, you have the possibility of two different captions for the same object.

Meta Data Tables Must Exist

MetaMgr's Init method needs to find DBCXReg, the DBCX registry table, to know which managers to instantiate. If this table can't be found, MetaMgr fails to instantiate. Each manager needs to find its own meta data table; if these tables can't be found, these managers will bomb. The problem is that MetaMgr doesn't create DBCXReg and none of the managers create their own meta data tables; they all expect that something else created them. Third-party tools such as SDT and VFE create these tables for you, but if you want to use DBCX as a public domain tool without these third-party add-ons, you have to write your own table-creation utility.

Method Clumsiness

The DBCX methods DBCXGetProp and DBCXSetProp work like their native VFP counterparts DBGETPROP() and DBSETPROP(): they get and set the values of extended properties for database objects. However, getting and setting properties is a two step process: you first have to get the DBCX ID for the object using DBGetDBCKey, then you use DBCXGetProp or DBCXSetProp. Also, the parameter order for DBGetDBCKey is different than it is for DBGETPROP and DBSETPROP; while the VFP functions expect the parameters <object name>, <object type>, <property>, DBGetDBCKey expects <database>, <object type>, <object name>. I wasn't the only one to find this method fail only to realize I'd specified the parameters in the wrong order.

Default Datasession

After you instantiate MetaMgr, you'll find several tables or cursors open: DBCXReg, DBCXProps (a cursor of all the properties managed by all the managers), and at least one table for each manager. Typically, MetaMgr is instantiated at application startup, so these tables are open in the default datasession. Although no other objects or code should manipulate these tables directly, there's nothing preventing that from happening since they aren't open in a datasession private to DBCX.

DBCX 2

Last summer and fall, Toni, Mike, and I discussed the need to improve on these shortcomings. I was working on the design for DBCX 2 when an idea hit me as I was mowing my lawn one afternoon (have you ever noticed the best ideas come when you're doing something mundane like showering or mowing the lawn so your brain can free-cycle?): why do we need an ID linking a database object with a meta data record? I and many others have written data dictionaries for FoxPro 2.6 and we didn't need an ID value; we just looked up an object in a table by its name. Why not do the same thing with DBCX?

This simple revelation lead immediately to the biggest improvements in DBCX 2: solving the synchronization, performance, method clumsiness, and database-only orientation problems. Since there's no ID value to link a database object with a meta data record, there's no way the two can get out of sync because we only care about the object name (the only thing to watch out for is when a database object is renamed). Since we can now do a SEEK for an object name in a meta data table, there's no longer a need to open the DBC as a table, find the object, parse the USER memo, and convert the character ID into a numeric one just so we can find the record in the meta data table by ID; the new mechanism is not only faster but eliminates the need for a two-step process to get or set an extended property value. Since an object is looked up by name rather than an ID found in a database, non-database objects such a free tables and calculated fields can now have their own meta data records.

Other improvements quickly followed:

- The Validate method was completely rewritten. It now processes objects in logical groups (for example, processing an entire table at once rather than one field or index at a time) and it's now granular (it can process a single table or even a single field rather than the entire database at a time). The result is a dramatic increase in speed; databases that took 45 minutes to validate now take 3 minutes, and the validation necessary to add a single table to the meta data now takes just seconds rather than forcing a revalidation of the entire database.
- CdbkMgr and CdbkMeta were replaced with a true "core properties" manager and meta data table (CoreMgr and CoreMeta) which manage a complete set of common extended properties, including all necessary structural information and other common properties such as caption and comment. CoreMgr automatically converts an older CdbkMeta table into the new CoreMeta table if necessary. CoreMgr is included in DBCXMGR.VCX rather than a separate VCX as CdbkMgr was.
- Long property names are now supported, meaning you don't have to specify the manager prefix and field name to get a property (although for backward compatibility you still can); you can now specify Expression as a property rather than CBmExpr. This is achieved with a cursor (DBCXProps) that provides a mapping between a property name and the names of the manager and meta data field that stores that property. This has a side benefit that a single property can be referenced with several names by simply having more than one record in this cursor. SDT 5.1, which is based on DBCX 2, uses this mechanism to automatically convert requests for properties that have moved from the SDT meta data table to the CoreMeta table. For example, captions for tables and indexes were stored in the SDTMeta Caption field in DBCX 1 but are stored in the CoreMeta cCaption field in DBCX 2. SDTMgr adds a second record for this property to DBCXProps that specifies SDTCaption as the long name. Thus, older code that requested the SDTCaption property in DBCX 1 will be returned the value stored in CoreMeta cCaption under DBCX 2. This and another new feature discussed in a moment eliminate the need to rewrite older code calling DBCX methods.
- The DBCX manager (DBCXMgr, renamed from MetaMgr for reasons I'll explain in a moment) and BaseMgr (the parent class for all extension managers) classes now have CreateDBCXMeta methods that will automatically create their required tables if they can't be found.

- Since we now have a one-step process for getting and setting extended property values, DBCXGetProp and DBCXSetProp now accept the same parameters in the same order as the VFP native DBGETPROP() and DBSETPROP() functions. Also, all methods accepting object names now accept the database name as well, using the same syntax as other VFP functions (<database>!<object name>), making it easier to work with multiple databases at a time.
- Because code calling DBCX 1 methods did things the "old" way (getting an DBCX ID for an object using DBGetDBCKey and then passing the ID to DBCXGetProp or DBCXSetProp), this code would normally have to be rewritten to do things the new way with DBCX 2. To eliminate this need, the DBCX manager was renamed from MetaMgr to DBCXMgr, and DBCXMgr was subclassed into a MetaMgr class. The MetaMgr class has a DBGetDBCKey method (which DBCXMgr does not) and its DBCXGetProp, DBCXSetProp, and other methods accept parameters in the DBCX 1 format. Thus, older code that instantiated MetaMgr can continue to use DBCX 1 method calls without being rewritten but can still take advantage of new DBCX 2 features, while new code can instantiate DBCXMgr and use the simpler DBCX 2 method calls.
- DBCXMgr is now a subclass of Form (with the Visible property set to .F.) rather than Custom. This has several advantages. DBCXMgr has a private datasession so the only access to meta data records is through DBCX methods rather than direct manipulation. DBCXMgr has a Show method that calls the ShowUI method of each manager and then makes the form visible. This provides managers with the ability to add objects to the DBCXMgr form; this might, for example, be used for visual rather than programmatic display and changing of extended properties. Neither BaseMgr nor CoreMgr have any code in ShowUI, but other managers may decide to implement this feature.
- Error handling has been improved (there was none before, so anything is an improvement <g>). BaseMgr.Error calls This.Parent.Error, which passes the error call to DBCXMgr. An individual manager could override this behavior by handling its own known errors and using DODEFAULT to pass the remainder to DBCXMgr; this is the same scheme described in my error handling articles in the December 1997 and January 1998 issues of FoxTalk. DBCXMgr.Error logs the error to the aErrorInfo array property and either passes the error on to an ON ERROR handler if one is in place or uses MESSAGEBOX to display an error message. There's also a Warning method that displays warning messages to the user when certain problems are detected if DBCXMgr is in "debug" mode. You'll normally use debug mode in a development or test environment but not in a production environment because, like ASSERT statements, it's meant to trap programmer errors, such as bad method calls, rather than environmental or unforeseen errors.
- BaseMgr has a cObjectTypesHandled property that contains the first letter of object types a manager will maintain properties for; the default value for this property is "DTVCRFIPU", meaning the manager will validate databases, tables, views, connections, relations, fields, indexes, view parameters, and user-defined objects. However, if a particular manager doesn't store any useful information for a certain object type, why have a record in the meta data table for each object of that type? That manager could remove the appropriate letter from this list and its Validate method would ignore objects of that type. For example, SDTMgr doesn't manage properties for connections, relations, or view parameters, so "C", "R", and "P" were removed from this property's value.

Using DBCX 2

Now that we've gone over the improvements in DBCX 2, let's look at some practical examples of using it. The ZIP file on Subscriber Downloads Web page contains several sample PRGs that demonstrate DBCX features.

To instantiate DBCXMgr, use CREATEOBJECT. DBCXMgr.Init accepts three parameters (the last of which is new in DBCX 2). The first is whether "debug" mode should be used or not, the second is the directory where the meta data tables can be found, and the third is whether the meta data tables should be created if they're not found. Here's an example:

```
set classlib to DBCXMgr additive
oMeta = createobject('DBCXMgr', .T., '', .T.)
```

In this case, DBCXMgr will be in debug mode, the meta data tables will be located in the current directory, and DBCXMgr will create them if they're not found. If DBCXMgr has to create the meta data

tables, it creates DBCXReg, automatically registers CoreMgr by adding a record for it to DBCXReg, then instantiates CoreMgr and calls its CreateDBCXMeta method to create CoreMeta. Note that creating the meta data tables doesn't populate them with meta data about database objects; we'll get to that in a minute.

Since DBCXMgr is in a private datasession, it doesn't "see" the databases used by your application, so you must do one of two things: specify the database name every time you specify an object name to a DBCX method (which would be a pain) or use the SetDatabase method to tell DBCX the default database to use. Here's an example that opens the TESTDATA database that comes with VFP and tells DBCXMgr we're using it.

```
open database (home() + 'SAMPLES\DATA\TESTDATA')
oMeta.SetDatabase(dbc())
```

To create meta data for the objects in the database, use the Validate method. If you want to see the progress of Validate, set the lShowStatus property to .T. Validate can accept object name and type parameters to just validate a single object; passing no parameters tells it to validate the entire default database.

```
oMeta.lShowStatus = .T.
oMeta.Validate()
```

Since free tables don't belong to a database, they have to be added to the meta data separately. Here's an example that creates a free table and meta data for it. Notice the leading "!" in the call to Validate so we indicate this table isn't part of the default database; use this syntax for all method calls for free tables.

```
create table TEST free (FIELD1 C(10), FIELD2 C(10))
index on FIELD1 tag FIELD1
index on FIELD2 tag FIELD2
use
oMeta.Validate('!test', 'Table')
```

Let's see how to set and get extended properties. Although a default caption is assigned to all non-field objects (field captions are stored in the DBC, so there's no need to store them in an extended property), a more appropriate caption may be desired. Also, since a free table doesn't have database properties, we'll want to assign field captions and comments as extended properties. If you use several calls in a row for the same object, you don't have to specify the object name or type each time; DBCX will use the same values as the previous call.

```
oMeta.DBCXSetProp('customer.company', 'Index', ;
  'Caption', 'Company Name')
? oMeta.DBCXGetProp('Caption')
oMeta.DBCXSetProp('!test.field1', 'Field', ;
  'Caption', 'First Field')
oMeta.DBCXSetProp('Comment', 'First Field Comment')
oMeta.DBCXSetProp('!test.field2', 'Field', ;
  'Caption', 'Second Field')
oMeta.DBCXSetProp('Comment', 'Second Field Comment')
```

Calculated fields can be defined by putting the calculation expression into the mExpr field in CoreMeta (the long property name is Expression). Here's an example that defines a Total Price field for the ORDITEMS table with the calculation ORDITEM.UNIT_PRICE * ORDITEMS.QUANTITY (in other words, the total price for a line item in an order), then uses the expression in a browse window. Note the use of ISNULL on the return value of DBCXGetProp to see if an object exists in the meta data, and AddRow to create a new meta data record.

```
if isnull(oMeta.DBCXGetProp('orditems.total_price', ;
  'User', 'Expression'))
  oMeta.AddRow('orditems.total_price', 'User')
  oMeta.DBCXSetProp('Expression', ;
    'orditems.unit_price * orditems.quantity')
  oMeta.DBCXSetProp('Caption', 'Total Price')
  oMeta.DBCXSetProp('Comment', ;
```

```
    'A calculated field of the total price')
endif isnull(oMeta.DBCXGetProp(...
lcExpr = oMeta.DBCXGetProp('Expression')
use ORDITEMS
browse fields ORDER_ID, LINE_NO, UNIT_PRICE, ;
  QUANTITY, TOTAL_PRICE = &lcExpr
```

New properties can be created using DBCXCreateProp. You specify the name of the property (manager prefix and field name), the name of the manager maintaining this property, the long name of the property, and its data type and size (optional for those data types that have a fixed size such as Date, Logical, and Currency). The following code creates a Select (can the index be selected by the user) extended property for indexes and set it to .T. for some but not all indexes.

```
oMeta.DBCXCreateProp('CBlSelect', 'oCoreMgr', ;
  'Select', 'L')
oMeta.DBCXSetProp('customer.company', 'Index', ;
  'Select', .T.)
oMeta.DBCXSetProp('customer.contact', 'Index', ;
  'Select', .T.)
oMeta.DBCXSetProp('customer.postalcode', 'Index', ;
  'Select', .T.)
oMeta.DBCXSetProp('customer.country', 'Index', ;
  'Select', .T.)
```

A new method in DBCX 2 called DBCXGetAllObjects creates an array of all objects with a certain property set to a specified value. By default, this method just puts the name of matching objects into a one-dimensional array, but you can specify another property (most commonly Caption) to put into the first column of a two-dimensional array (the second column is the object name). This method is ideal for creating an array used as the RowSource for a combobox or listbox in which the user can select an object from the list of matching object. The following code, for example, gets the name and caption of all indexes from the CUSTOMER table that have the Select property set to .T.; this could then be displayed to the user in a combobox so they can choose the sort order for a report or data entry form.

```
dimension laIndexes[1]
lnObjects = oMeta.DBCXGetAllObjects('index customer', ;
  @laIndexes, 'Caption', 'Select', .T.)
? ltrim(str(lnObjects)) + ;
  ' indexes selectable for CUSTOMER'
display memory like laIndexes
```

What if you'd rather maintain your own extended properties in a separate meta data table rather than in CoreMeta? Create your own extensions manager. Start by subclassing BaseMgr to create your own manager (put it into a different VCX so installing a new version of DBCXMGR.VCX doesn't wipe out your manager). Decide what properties you'll maintain and pick field names for each one. In addition to these, you'll need four other fields: one for the name of the database an object belongs to, one for the record type (such as "D" for database), one for the object name, and one to indicate when an object has been processed by Validate (this should be a DateTime field). You'll need a tag on UPPER(<database name field> + <record type field> + <object name field>) for searching for an object. Set the cDBCXAlias property to the name of your meta data table, cDBCNameField to the name of the field for the database (the default is DBCName), cRecTypeField to the name of the field for the object type (the default is RecType), cObjectNameField to the name of the field for the object name (the default is ObjectName), cProcessField to the name of the DateTime process field (the default is LastUpdate), cDBCXTag to the name of the tag for searching for an object (the default is ObjectName), and cPrefix to the prefix for your manager. Put code into the CreateDBCXMeta method to create your table and AddRow to add a new record for an object to your table. This may sound like a lot of work, but if you look at the code in CoreMgr's methods for ideas, you can probably create a new manager in just a few minutes. To register your manager in DBCXReg, either manually add a new record for it into this table or use the following DBCX method call:

```
oMeta.RegisterManager(<manager description>, <VCX directory>, <VCX name>, <class name>)
```

Conclusion

DBCX 2 is a great improvement over its predecessor: it's way faster, more flexible, and doesn't run into some of the problems DBCX 1 did. Did I also mention that it's a lot faster? Complete documentation for DBCX 2, including details on the differences between it and DBCX 1, can be found in DBCXREF.DOC included in the ZIP file on the Subscriber Downloads Web page.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit and Stonefield Query. 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 the 1997 and 1998 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP). CompuServe 75156,2326 or dhennig@stonefield.com.