February, 2006

Advisor Answers

Importing Access Data

VFP 9/8/7

Q: How do you import tables from Access? Can you?

A: Last May, I showed how to create an Access database matching an existing VFP database. You want to do the reverse—create a VFP database to match an existing Access database. If all you want is the tables and the data, this is actually quite a lot simpler than going from VFP to Access. If you want to exercise more control over the process (such as omitting some tables or fields, or including existing indexes), you have to do a little more work.

Let's start with the simple approach. Using SQL Pass-through (SPT), VFP's technique for talking directly to a server, you can connect to the Access database, get a list of its tables, and then extract the data in them.

Step 1 is connecting to the database. Since I want to do the whole task in code, I'll use SQLStringConnect and build the connection string I need in the code. The connection string for SPT with an Access database looks like this (the line break here is just to fit on the page):

```
"Driver={Microsoft Access Driver (*.mdb)};Dbq=MyData.MDB;Uid=Admin;Pwd=;"
```

Substitute the name and path of your Access database for "MyData.MDB" and, if necessary, specify a different user id and password.

Once you're connected, VFP's SQLTables() function lets you retrieve a list of tables. Access databases actually can include not only tables, but views. In addition, every Access database includes a number of system tables. For our purposes, just the tables are needed; to limit SQLTables() to real tables, pass "Table" for the optional second parameter. The function puts the list into a cursor. If you don't specify otherwise, the cursor is called SQLResult. (In fact, that's the default name for any cursor created via SPT.) You can specify the cursor name by passing it as the third parameter.

Once you have the list, you just need to go through the cursor, read the data from each table and save it to a VFP table. There's one complication. Access allows tables with spaces in the name, but VFP can't directly address them. To work around this issue, though, you can surround the table name with square brackets.

Here's the complete program to create a copy of an Access database via SPT:

```
LOCAL cMDBFile, cDBName, cConnString, nHandle
LOCAL cTable, cQuery

cMDBFile = GETFILE("MDB","Access database", "Import")
IF EMPTY(cMDBFile)
   RETURN
ENDIF

* Create a VFP database
cDBName = JUSTSTEM(cMDBFile)
CREATE DATABASE (cDBName)

cConnString = ;
  "Driver={Microsoft Access Driver (*.mdb)}" + ;
  "Dbq=&cMDBFile;Uid=Admin;Pwd=;"
nHandle = SQLSTRINGCONNECT(cConnString)

SQLTABLES(nHandle, "Table", "TableList")

SCAN
   cTable = ALLTRIM(TableList.Table_Name)
   cQuery = "SELECT * FROM [" + cTable + "]"
   IF SQLEXEC(nHandle, cQuery, "CurTable") > -1
      * Create a real table
      COPY TO (cTable) DATABASE (cDBName)
   ELSE
      MESSAGEBOX("Problem with " + cTable)
   ENDIF
ENDSCAN

USE IN CurTable
USE IN TableList
SQLDISCONNECT(nHandle)

RETURN
```

For more control over the process or to recreate only the structure of the Access database, use ADOX (ADO Extensions). Start with the ADOX Catalog object, which provides a Tables collection. Each Table object has Columns and Indexes collections.

Once you create a table and its indexes, you can use ADO plus a CursorAdapter to copy the data from the Access table to the new table.

To open the Access database, use an ADO Connection object; once it's open, an ADOX Catalog object can access it.

```
#DEFINE JetProvider "Microsoft.Jet.OLEDB.4.0"

oConn = CREATEOBJECT("ADODB.Connection")

cConnString = "Provider="+JetProvider+ ;
   ";Jet OLEDB:Engine Type=5;Data Source=&cMDBFile"
oConn.ConnectionString = cConnString
oConn.Open()

oCatalog = CREATEOBJECT("ADOX.Catalog")
oCatalog.ActiveConnection = oConn
```

Like SQLTables(), the Catalog object shows you all types of tables in the database, so you need to limit the code to tables with the Type property set to "TABLE".

One of the easiest ways to create a table in VFP is to put the structure of the table into an array and use CREATE TABLE FROM ARRAY. The loop here populates the array by reading properties of the Column object. Column has explicit properties for the main characteristics of the field, such as Name, Type and DefinedSize. Additional characteristics are stored in a Properties collection of the Column object. This code extracts only the Nullable property from the collection, but additional information is available as well.

Access allows some characters in field names that VFP chokes on, so the code uses CHRTRAN() to get rid of those (including spaces). ADOX stores Type as a number. I wrote a function, ConvertToVFPType (not shown here), to convert from an ADO data type to the single character VFP expects.

```
* Build a list of columns
DIMENSION aFieldList[ oTable.Columns.Count, 5]
nColumn = 1
FOR EACH oColumn IN oTable.Columns
   aFieldList[nColumn, 1] = ;
      CHRTRAN(oColumn.Name,"?!@#$%^&* ","")
   aFieldList[nColumn, 2] = ;
      ConvertToVFPType(oColumn.Type)
   aFieldList[nColumn, 3] = ;
      MIN(oColumn.DefinedSize, 254)
   aFieldList[nColumn, 4] = oColumn.Precision
   IF oColumn.DefinedSize = 0 AND ;
      oColumn.Precision <> 0
      aFieldList[nColumn, 3] = oColumn.Precision+2
   ENDIF
   * Must deal with nulls
   aFieldList[nColumn, 5] = ;
      oColumn.Properties("Nullable").Value

   nColumn = nColumn + 1
ENDFOR

CREATE TABLE (cTable) FROM ARRAY aFieldList
```

To add indexes to the table, loop through the Indexes collection. One complication here is that Access allows indexes to be created based on multiple fields without actually providing the expression to combine them. (For example, in the Northwind database's Order Details table, there's an index based on OrderID and ProductID, both integer fields.) In converting such indexes, we need to make sure we don't try adding integers or dates. This code takes a brute force approach, using PADL(<field>, 20) for each field in the expression.

```
* Now get the indexes
FOR EACH oIndex IN oTable.Indexes
   cTag = LEFT(oIndex.Name, 10)
   cKey = ""
   lCombined = oIndex.Columns.Count > 1
   FOR EACH oColumn IN oIndex.Columns
      IF lCombined AND ;
         ConvertToVFPType(oTable.Columns( ;
         oColumn.Name).Type) <> "C"
         cKey = cKey + "+ PADL(" + ;
                oColumn.Name + ", 20)"
      ELSE
         cKey = cKey + "+" + oColumn.Name
      ENDIF
   ENDFOR
   cKey = SUBSTR(cKey, 2)
   lPrimary = oIndex.PrimaryKey
   lCandidate = oIndex.Unique

   DO CASE
   CASE lPrimary
      ALTER TABLE (cTable) ADD PRIMARY KEY &cKey ;
         TAG (cTag)
   CASE lCandidate
      ALTER TABLE (cTable) ADD UNIQUE &cKey ;
         TAG (cTag)
   OTHERWISE
      INDEX on &cKey TAG (cTag)
   ENDCASE
ENDFOR
```

The last step is copying the data. Using a CursorAdapter and an ADO RecordSet created and linked earlier in the program, a cursor is filled with the data, which is then copied to the new table.

```
* Now get the data
oCA.SelectCmd = "SELECT * FROM [" + cTable + "]"
oCA.CursorFill()

SELECT 0
USE (cTable) ALIAS __NewTable
APPEND FROM DBF("CurTable")
USE IN __NewTable
```

While this approach requires more code than SPT, it also gives you more opportunities to intervene and change the results. One downside of this version is that it creates each table with the fields in alphabetical order rather than in their original order. I haven't been able to find any properties in the ADOX model to work around this.

The ADOX version requires an OLE DB provider for the Jet engine, while the SPT version uses an ODBC driver. Both versions are included on this month's Professional Resource CD and subscriber downloads.

–Tamar