Case Study: Using SQLite to break the 2GB Barrier

Whil Hentzen

"We need to get away from DBFs" is a refrain I hear regularly from fellow developers. Be it due to perceived instability of the file format, the need for tables larger than 2 GB, or the result of political machinations, the result is the same – a desire to move to a SQL database back-end. SQLite can be an excellent intermediate step – and possibly the final word - in the process of restructuring your application to talk to a SQL back-end.

In previous articles, I've helped you dip your toe into the SQLite pool, in preparation for using it as an application backend. In this article, I'm going to take a different direction and show you how I used SQLite to help a customer deal with an external data source that, in its latest release, had become too big to import into a VFP table.

My customer, we'll call him 'Al', has been downloading a data set full of demographic data from a government website and importing it into VFP for the better part of 15 years. As with many things data-related, this data set has been growing over the years, but just marginally.

Problems, Schmroblems

The latest release, however, was chock full of changes.

First, the previous releases consisted of hundreds of text files, each for a specific geographical region. Each file was between a half megabyte and two megabytes in size. The latest release combined all of those hundreds of files into a single file.

Second, the tables used to contain lookup codes used to identify a variety of descriptors, like so:

GeoID	Region	H1	H2	…
310M100US28180	71	213	160	

Code	Description
71	Southeastern Arizona, excl. Carlsbad

The latest release used the full text descriptors instead of the foreign key to a lookup table that contained the full description:

GeoID	Region	H1	…
310M100US28180	Southeastern Arizona, excl. Carlsbad	213	

Besides the expected issues with denormalized data, using full descriptors instead of foreign keys when you're dealing with millions of rows produces a lot of wasted space. A LOT.

Finally, in previous releases, just raw values were included. In the latest release, calculate values were included – both grand totals as well as percentages. For example, the raw data would include values for age groups:

< 15	16-19	20-24	…	75
109	621	539		57

The latest release also included totals for ranges:

Child < 20	Adult 20-59	Senior 60+	Total
730	12558	808	14096

It also included percentages, like so:

< 15	16-19	20-24	…	75
0.01	0.04	0.04		0

As a result, there was a significant superfluity of columns.

Additionally, the totals didn't add up (that's the government for you), but that simply meant that we wanted to discard those columns more than ever.

As a result of these changes, he had one file to work with, and it was suddenly much larger than

the combined text files in the previous release. On the order of three times as large – from a collection of files that totaled just over a gigabyte to a single file now well over three gigabytes. And we all know that VFP doesn't play well with tables of that size.

The Solution

Of course, a three gigabyte table is child's play for SQL databases like Microsoft SQL Server, MySQL or PostgreSQL. But Al was only interested in a subset of the data set, and so. This was a one-time use, just to parse the data he wanted out of the original file. Thus, it didn't make sense to deal with the infrastructure and learning curve of one of those applications. He simply wanted to grab a subset of the data, normalize it, and stuff it into his own application.

Enter SQLite.

Our plan was to (1) create a database and a single table in SQLite according to the data definition that was included with the data set itself, (2) import the text file, and then (3) extract the subset of the data that we were interested in – just the raw columns, no totals – and normalize the lookups again.

The first and third steps would be done with VFP. Connecting to SQLite and manipulating data has been discussed in Part One of this series, Getting Started. The second step would use the SQLite .import command as has been discussed in Part Three of this series, Inserting Large Amounts of Data.

Step 0. Preliminaries

The following were needed to perform this process.

1. Standard Visual FoxPro installation. See my article, Setting up VFP 9, in the May/June 2013 issue of FoxRockX.

2. SQLite ODBC driver. See my article, Getting Started with Client-Server with SQLite in the March/April, 2012 issue of FoxRockX. The latest version as of this writing is 0.993, released on May 23, 2013.

3. The SQLite engine, SQLITE3.EXE, available at sqlite.org. For the purposes of this article, this file will be located in the root of drive F. You can just copy this file to this location after downloading; no installation routine or configuration process is needed. The latest version as of this writing is 3.7.17, released on May 20, 2013.

4. The three gigabyte census text file, also located in the root of drive F. The text file's fields are delimited with pipes, like so:

```
AAA|BBB|123|1000|24.7|LASTFIELD
```

Step 1. Create a SQLite database and table

The data definition described a file layout that consisted of four groups of columns. The first group contained the descriptors for the row, maybe a half-dozen columns. The next three groups were identical, a series of counts for various sectors for males, for females, and then the combined totals of males and females.

There were 194 columns in the table. No one wants to type a CREATE TABLE with 194 column definitions. Not even I.

So instead of typing a long statement in the SQLite interface, I used VFP to create the CREATE TABLE string, and then executed that command via SQL Passthrough to create the table in the SQLite database.

In the examples that follow, I'm demonstrating the code but using an abbreviated set of columns. For example, for the first group, I am only showing two columns, not all six.

First, I created the leading fields:

```
local lcStr
lcStr = 'create table ALLDATA ';
+ ' (geoid c(14), occ c(5)'
```

Next, I created a series of fields of the form 'b01e' and 'b01m', 32 of each, where 'b' stands for 'both genders', the 'e' columns contains the estimated value and the 'm' column contains a percentage that is the margin of error for the estimate.

```
for li = 1 to 32
  lcStr = lcStr ;
  + ', ' ;
  + 'b' + padl(allt(str(li)),2,'0') + 'e ' ;
  + 'n(8)' ;
  + ', ' ;
  + 'b' + padl(allt(str(li)),2,'0') + 'm ' ;
  + 'n(8,2)'
next
```

Next, I repeated that code for both the male and female counts:

```
for li = 1 to 32
  lcStr = lcStr ;
  + ', ' ;
  + 'm' + padl(allt(str(li)),2,'0') + 'e ' ;
  + 'n(8)' ;
  + ', ' ;
  + 'm' + padl(allt(str(li)),2,'0') + 'm ' ;
  + 'n(8,2)'
next

for li = 1 to 32
  lcStr = lcStr ;
  + ', ' ;
  + 'f' + padl(allt(str(li)),2,'0') + 'e ' ;
  + 'n(8)' ;
  + ', ' ;
  + 'f' + padl(allt(str(li)),2,'0') + 'm ' ;
  + 'n(8,2)'
next
```

Finally, I closed the list of fields:

```
lcStr = lcStr + ')'
```

Now that the string has been assembled, connect to SQLite:

```
liH = sqlstringconnect( ;
  "DRIVER={SQLite3 ODBC Driver};" ;
  + "Database=f:\eedb;")
```

And execute the command, displaying success or failure in a messagebox (SQLEXEC returns a positive number for success and -1 for failure.):

```
liResult = sqlexec(liH, lcStr)
messagebox("Create Table result:" ;
  + str(liResult))
```

At this point, we've got an empty table called ALLDATA in the EEDB database located in the root of drive F.

The next step is to import the text file into the table.

Step 2. Import the data

After closing VFP (so that SQLite had exclusive use of the table), I opened up SQLite by double-clicking on the SQLITE3.EXE item in my file manager, producing a DOS window with a prompt:

```
sqlite>
```

Open the database:

```
sqlite> attach database EEDB as EEDB;
```

and import the three gigabyte text file, EEALL.DAT, into the ALL DATA table:

```
sqlite>.import eeall.dat ALLDATA
```

At this point, the table has all the rows. Os so you would think.

More Schmroblems

As luck would have it, the data definition file that came along with the data set was, um, shall we say, a little bit behind the times. Unfortunately, the table had 195 columns. More bluntly, it wasn't right. To save you the trouble of having to skip to the end of our story and find out that the butler didn't do it, the data definition file described a layout with one fewer columns than the actual data file contained.

Thus, I had created a 194 column table according to the data definition. When I tried to import the 195 column text file into the table, SQLite responded with the unexpected but perfectly reasonable

```
Error: eeall.dat line 1: expected 194 columns
of data but found 195
```

message. This is saying that the SQLite table was defined with 194 columns, and thus was expecting an input table of 194 columns, but the incoming table had 195. Really? It's like serving a Pinot Grigio with a side of raw cow.

So I was left with a three gig file whose format was unknown. Purists would argue that this is a moot point. One of the advantages of SQLite is that the data type included in field definitions is simply a suggestion - I could have just added a 195th dummy field to the table and the import would have imported the file, regardless of whether the data types in the file matched the data types in the definition, just fine.

But we wanted to know what the data looked like – we were eventually going to have to work with it in such a fashion that the data types *would* be important. The easiest way would be to open this text file in a text editor. Sure, easy.... except, it was a three gigabyte text file. Not so easy. Most text editors will choke on a file of that size, and one or two will even bring down Windows itself.

We're gonna need a bigger boat.

Another tool for our toolkit

Spelunking around the Web produced not only dozens of pictures of adorable kittens, but also a number of promising but ultimately unsatisfactory results, until I came across the magic "UltraEdit".

Packed with more features than Microsoft Word, the feature of UltraEdit I was most interested in is its capability to handle very, very (very, very, very) large files. It natively handles files over four gigabytes via disk-based handling on both 32- and 64-bit Windows.

So I cracked open a copy of UltraEdit and opened this three gigabyte file so I could examine the data itself. Turns out that the one-off error was in the first few columns. The data definition contained this:

```
geoid c(14), occ c(5)
```

while the data itself contained:

```
geoid c(14), geoidsp c(14), occ c(5)
```

Mystery solved. I changed the command to create the leading fields like so:

```
local lcStr
lcStr = 'create table ALLDATA ';
```

```
+ ' (geoid c(14), geoidsp c(14), occ c(5)'
```

And a few seconds later, had a 195 column table named ALLDATA in the EEDB SQLite database.

Running the .import command a second worked perfectly, and 5 minutes later I had a SQLite table with 2,570,751 rows in it:

```
sqlite> select count(*) from ALLDATA;
2570751
sqlite>
```

Yes, SQLite imported a 195 column, two and a half million row text file in under five minutes.

(Side note: while the .import command didn't need a semi-colon to execute it, the SQL command SELECT did.)

Step 3. Preparing a test table for slicing and dicing the data

You would think that the next step was to create a series of SELECTS in VFP that would extract the data that Al and I wanted. (Remember Al? This project is for Al.)

However, given that this is a 2.5 million row table, it behooves us to think about what we want to accomplish. Yes, VFP is fast, but not THAT fast that an errant operation on the table could bring our machine to its knees for a while.

A better solution is to create a small subset of the table, say, a dozen or maybe a hundred rows, and test our slicing and dicing on that table first.

The first step is to create a second table, say, SOMEDATA, in EEDB, using the same code described in Step 1, except changing the first line of the create table statement from

```
create table ALLDATA
```

to

```
create table SOMEDATA
```

After executing the SQLEXEC on this changed statement, the SQLite database EEDB contains two tables, ALLDATA and SOMEDATA. Next, how to get a few rows into SOMEDATA?

One way is to create a second text file that only contains the rows of interest, say, EESOME.DAT, and import it into SQLite with the .import command, just like we imported EEALL.DAT. To do so, open the file in our new best friend, UltraEdit, highlight the rows of interest, copy them, open a new editing session, paste, and, finally, save as EESOME.DAT. Be sure to include the carriage return at the end of the last row to be included in the new file.

With our new miniature text file, execute the following commands in SQLite:

```
sqlite>
sqlite> attach database EEDB as EEDB;
sqlite>.import eesome.dat SOMEDATA
sqlite> select count(*) from SOMEDATA;
101
sqlite>
```

While certainly expedient, doing the cut and paste routine probably offends a little bit of each of us, and doesn't teach us much either. Now that we've got our empty SOMEDATA table in EEDB, let's write a bit of code that copies some rows from ALLDATA to SOMEDATA.

The approach goes like this: Select a few records from ALLDATA into a temporary cursor, then go through that cursor record by record and do a SQL-INSERT to stuff them into the SOMEDATA table.

Here's the p-code. First, select a few records from ALLDATA:

```
select * from ALLDATA where (condition)
```

where the “condition” retrieves a limited number of rows. In this case, a single value in the geoid field matches about 400 rows:

```
lcStr = "select * from alldata " ;
  "where geoid = '310M100US10500'"
```

Next, stuff those records into a temporary cursor, in this case named 'csrTmp':

```
sqlexec(liH, lcStr,' csrTmp')
select csrTmp
```

Third, construct a string that contains the field names. (We can't just use the field string created for the CREATE TABLE command because we don't want the field type and length clauses.) We only have to create this string once, because it won't change record by record, while the values will.

```
lcStrFields = 'geoid, geoidsp, occ'
for li = 1 to 32
lcStrFields = lcStrFields ;
  + ', ' ;
  + 'b' + padl(alltrim(str(li)),2,'0') + 'e' ;
  + ', ' ;
  + 'b' + padl(alltrim(str(li)),2,'0') + 'moe'
next
```

The for/next segment is repeated, replacing the 'b' with 'm' and 'f'. Next, spin through the cursor

```
select csrTmp
scan
lcStrValues ;
  = "'" + alltrim(geoid) + "'," ;
  + "'" + alltrim(geoidsp) + "','" ;
```

```
  + alltrim(occ) + "'"
for li = 1 to 32
  lcStrValues = lcStrValues ;
  + "," ;
  + "'" ;
  + allt(eval('b' + padl(li,2,'0') + 'e')) ;
  + "'," ;
  + "'" ;
  + allt(eval('b' + padl(li,2,'0') + 'moe')) ;
  + "'"
next
```

and again, repeat the for/next segment, replacing the 'b' with 'm' and 'f'.

Finally, build the INSERT command and execute it:

```
  lcStrIns ;
    = "insert into SOMEDATA " ;
    + "(" + lcStrFields + ") " ;
    + " values " ;
    + "(" + lcStrValues + ")"
  liResult = sqlexec(liH, lcStrIns)
endscan
```

We now have a subset of ALLDATA in SOMEDATA. We're ready to experiment.

Naming conventions

So Al and I have both a small data set that we can experiment with and the complete data set, both in the EEDB database. What next?

As you well know, any time you experiment with data extraction and manipulation, you end up with dozens of programs named MAIN, MAIN1, MAINOLD, TEST, DELETEME, and so on. Of course, they're all perfectly commented, so it's easy to keep track of which program does what, right?

You also end up with a folder full of data files. Some are the results of preliminary tests, others are intermediate results, generated during a multi-step process, and others are the output desired. Sure, it's possible to create a folder structure where different categories of files are stored in their own folders, but for a quick and dirty project like this, that's likely overkill.

Pretty soon you lose track of which files are good and which are garbage. So let's create a naming scheme up front.

One way to go about this is to define the file categories, and name each file according to which they belong to. We'll have four categories of files:

- Input (masters)
- Processing (temporary)
- Intermediate results
- Output files

So a logical naming scheme could include some type of semaphore to indicate which category a file belongs to.

We'll also have several types of files – the original text files (.DAT extension), the SQLite database file (no extension), and, of course, a variety of VFP tables (.DBF extension.) Programs will have .PRG extensions.

Another possible consideration when naming is using a scheme so they're in alphabetical order when you view them in your file manager. Some of us are picky like that.

Here is the scheme we came up with.

Since some of these files will likely end up in another folder, say, as part of the application they're ultimately to be used with, it makes sense to start the filenames similarly. In this case, since these files are employee demographics, I start each file with "EE". They'll be nicely grouped when they end up in another folder.

Second, the first files we worked with, the SQLite database and the original text files, have simple names:

```
EEDB
EE100.DAT
EEBIG.DAT
```

For the files that we know we'll want to keep (the end results), starting the filenames with "EE" and ending with strings later in the alphabet provides a visual clue that these files are in order of use and production. Before we actually do the work, we think we'll end up with two files, the master data table and a child lookup table:

```
EEDB
EE100.DAT
EEBIG.DAT
EELOOKUP.DBF
EEMASTER.DBF
```

Now, what to do with the temporary and intermediate files?

I've always named temporary files, ones that are used for an instant and can then be instantly discarded, with the string "DELME" (for DELeteME) as the beginning of the name. That way, whenever I come across a folder with a DELME file in it, I know I can get rid of it without even looking at it.

Intermediate files, I'm going to argue, are similar in nature if they're constructed correctly. A processing program should always be re-runnable, never in danger of stomping over valuable files. As such, naming them along the same lines as "DELME" means that you know they can be deleted when you no longer need them. So, for instance names like:

```
EEDELME_JUSTTOTALS.DBF
EEDELME_GENDERGROUPS.DBF
```

and so on provide visual clues that these files are intermediate but can be discarded when desired. Most file managers allow you to search for files that contain a string in the middle of the filename, making their identification for later elimination trivial.

Step 4. Slicing and Dicing

Slicing and dicing a table involves selecting just certain columns and certain rows out of a table. Selecting a subset of columns is simple; just include the columns of interest in your SELECT statement. Selecting rows, however, can be considerably more complicated.

When you began using Fox (or dBASE, if your roots started there), you likely went through a learning phase with respect to syntax. For instance, you learned that

```
browse for lastname = 'Carl'
```

would pull out only last names that began with that exact string. Last names that started with “Car” or “CArl” or “carl” weren't included in the result set. You learned to do things like

```
browse for upper(lastname) = 'CARL'
```

for instance. When matching on a variable, you learned to trim the value of the variable, so that you didn't accidentally do something like this:

```
lcValue = myform.somefield
browse for upper(lastname) = upper(lcValue)
```

where somefield was 25 characters wide, and since this meant you were searching for

```
'CARL                '
```

you didn't find any get any hits for 'Carlson' or 'Carlsbad'.

You'll go through the same learning curve when matching with SQLite matching. The short explanation is that the same syntax you're used to using with Fox doesn't always translate to SQLite. Let's take a look at how to select character strings, numeric values, dates, and logical values.

Matching character strings

Matching an exact string works fine:

```
select * from TABLE where gender = “MALE”
```

does just what you would expect it to. However, partial matching of strings doesn't work.

```
select * from TABLE where gender = “M”
```

returns zero records, even when there are records that contain “MALE” in the gender field.

Instead, you need to use the SQLite “LIKE” operator along with the “%” wildcard, like so:

```
select * from TABLE where gender like 'M%'
```

You can also precede the character string with a wildcard in order to search for a string in the middle of the field, like so:

```
select * from TABLE where name like '%mith%'
```

This would find Pmith (the 'P' is silent), Smith, Smithson, and mithfortune.

Sometimes, though, just using a wildcard isn't what you want. SQLite parses the upper() and lower() functions properly, like so:

```
select * from TABLE ;
  where upper(name) like 'SMITH%'
```

In order to pull a certain number of characters out of a string, use substr(), just like you would in Fox:

```
select * from SOMEDATA ;
  where substr(geoid,10,4) = '1000'
```

SQLite doesn't have left() and right() functions; use substr() and just the search with the first character, like so:

```
select * from SOMEDATA ;
  where substr(geoid,1,5) = '12345'
```

Now let's look at other data types.

Matching numeric values

Numbers, on the other hand, are easy. Easy-peasy, in fact, because SQLite allows for all sorts of room for error.

There's a column in the table named B01E that contains numeric values. The following statement will pull out all rows that contain the value 15 in that column:

```
select * from SOMEDATA where b01e = 15
```

But it gets better. Suppose you thought that the column contained character values? The following statement will also work:

```
select * from SOMEDATA where b01e = '15'
```

In fact, it simply doesn't matter if you search for a numeric or character string, or if the column is defined as numeric or character – all combinations work.

In order to get a range of values, concatenate expressions with AND:

```
select * from SOMEDATA where b01e >= 15 ;
  and b01e <= 25
```

or OR:

```
select * from SOMEDATA where b01e <= 10 ;
  and b01e >= 1000
```

Matching dates

Date values are stored as strings in SQLite, so you don't have to mess around with any translations from one data type to another.

```
select * from TESTDATA ;
  where dob = '2013-07-01'
```

will retrieve all rows where Date of Birth is the first of July of this year. Since the date strings are stored in CCYYMMDD format, you can select ranges, like so:

```
select * from TESTDATA ;
  where dob < '2012-01-01'
```

to find all records with a Date of Birth in 2011 or earlier. Nor do you have to use date functions like month() or year(). Rather, substr() is your friend. To get all records for the month of June, regardless of year:

```
select * from TESTDATA ;
  where substr(dob,6,2) = '06'
```

Matching logical values

Like numerics and dates, logical values are stored as strings, either T/F or 1/0. Thus, selecting records is as simple as

```
select * from TABLE where alive = "T"
```

or

```
select * from TABLE where alive = 1
```

Now that we have the ability to select data of any field type, let's put it all together.

Step 5. Automating access via a SQLEXEC() shell

As you find yourself issuing the same series of commands over and over, you'll wish there was a better way. Most people begin by creating a series of programs, one to create the table, another to "drop" it after it gets munged and needs to be cleaned up, a third to issue a SELECT of one form or another. Then more programs to run additional SELECTS.

Eventually one ends up with that aforementioned collection of MAIN.PRG, MAIN1.PRG, SELECTDATA.PRG, SELDATA2.PRG, and so on – just as bad as the collection of result set files described earlier.

Since all of these programs have a lot of statements in common, I've tended to using a shell that contains common elements, and calls out to custom elements as needed. For example, let's look at the three functions mentioned above, creating a table, dropping it, and selecting data from it.

All three start out the same way – creating a connection:

```
liH = sqlstringconnect( ;
  "DRIVER={SQLite3 ODBC Driver};" ;
  + "Database=f:\eedb;")
```

After assembling a custom command, each program executes the command, using the handle created:

```
liResult = sqlexec(liH, lcStr)
messagebox("Create Table result:" ;
  + str(liResult))
```

Some optionally pass a cursor name:

```
lcStrCursor = 'csrEErows'
liResult = sqlexec(liH, lcStr, lcStrCursor)
```

And each finishes up by closing the connection:

```
? sqldisconnect(liH)
```

As a result, you can build a single program that uses each of these statements just once. Then, pass a parameter to the program that indicates which custom routine is to be executed.

```
* sqlshell.prg

lpara lcWhatToDo
* lcWhatToDo = 'C' && create
* lcWhatToDo = 'I' && insert
* lcWhatToDo = 'SC'  && select count
* lcWhatToDo = 'SR'  && select rows
* lcWhatToDo = 'D'  && drop table

if pcount() < 1
  messagebox("Must pass a parm.")
  return
endif

liH = sqlstringconnect( ;
  "DRIVER={SQLite3 ODBC Driver};" ;
  + "Database=f:\eedb;")

if liH < 1
  messagebox("Connection failed.")
  return
endif

lcStr = z_createStr(lcWhatToDo)

do case
case lcWhatToDo = 'C'
  lcDesc = 'Create Table'
  lcNaCursor = ''
case lcWhatToDo = 'SC'
```

```
  lcDesc = 'Select Count'
  lcNaCursor = 'csrEEcount'
case lcWhatToDo = 'SR'
  lcDesc = 'Select Rows'
  lcNaCursor = 'csrEErows'
case lcWhatToDo = 'D'
  lcDesc = 'Drop Table'
  lcNaCursor = ''
endcase

lnSec1 = seconds()
liResult = sqlexec(liH, lcStr, lcNaCursor)
lnSec2 = seconds()
lcElapsed = str(lnSec2-lnSec1,10,3)

messagebox(lcDesc +  ' result:' ;
  + str(liResult) + ' in ' ;
  + lcElapsed + ' sec.')

? sqldisconnect(liH)

return
```

The z_createStr() function called before the DO CASE code segment encapsulates the detailed work of building the SQL command for each specific case.

```
function z_createStr(lcWhatToDo)

local lcStr, li, lcVal

do case
case lcWhatToDo = 'C'
  (code for the 'create table' string)


case lcWhatToDo = 'I'
  (code for the 'insert' string)

case lcWhatToDo = 'SC'
  (code for the 'select count' string)

case lcWhatToDo = 'SR'
  (code for the 'select rows' string)

case lcWhatToDo = 'D'
  (code for the 'drop table' string)

endcase

return lcStr
```

Then, to run this program, simply execute

```
do sqlshell with 'C'
```

or whatever parameter you wish. The shell takes care of all of the setup and shutdown every time, and doesn't leave a lot of garbage around after you're done processing.

A hidden advantage to SQLSHELL

The SQLSHELL introduced in the last section is good for more than minimizing file cruft. It can also be used as the starting point to creating a front end for making access to SQLite data more VFP-like - that is, typing commands into a Command Window and getting results back.

Consider passing not a flag that identifies which type of hard-coded statement should be executed, but passing the entire SQL statement.

```
* sqlshell.prg

lparameters lcStr
if pcount() < 1
  messagebox("Need to pass a parm.")
  return
endif

local liH, lcStr, lcDesc, lcNaCursor, ;
  liResult, lnSec1, lnSec2, lcElapsed

liH = sqlstringconnect( ;
  "DRIVER={SQLite3 ODBC Driver};" ;
  + " Database=f:\eedb;")

if liH < 1
  messagebox("Connection failed.")
  return
endif

lcDesc = substr(lcStr,1,at(' ', lcStr, 2))
lcNaCursor = 'csrEEtest'

lnSec1 = seconds()
liResult = sqlexec(liH, lcStr, lcNaCursor)
lnSec2 = seconds()
lcElapsed = str(lnSec2-lnSec1,10,3)

messagebox(lcDesc +  ' result:' ;
  + str(liResult) + ' in ' ;
  + lcElapsed + ' sec.')

return
```

You could throw a browse command, like so, at the end, if you liked:

```
select (lcNaCursor)
browse last
```

At any rate, the idea is that now you can open VFP, type

```
do sqlshell with ;
  'select geoid,m01e,f01e,b01e from alldata'
```

and a few seconds later, the result set will be created and stuffed into the cursor defined in lcNaCursor, ready for you to examine and work with as desired.

Enhancements to the basic shell

Additional mods are possible, of course. For instance, you might want to create customized cursor names instead of a hard-coded value, so that as you execute subsequent commands, previous result sets don't get overwritten.

Another mod would be to more rigorous about the determining whether the command actually succeeded. As the saying goes, "It's all fun and games until somebody loses a half million rows."

The SQLEXEC() command returns a numeric result. If the command failed, the result will be -1.

If the command is continuing to process, the result will be 0. And once the command is finished, the result value will be the number of result sets produced. In our examples, we're only expecting a single result set, but you can configure SQLEXEC() to execute asynchronously, and thus keep returning result sets.

Since SQLEXEC()'s behavior of simply returning a -1 is the equivalent of a silent failure, it's best to explicitly check the result. In fact, it's imperative.

The call to messagebox() does provide a basic alert that Something Bad Happened, but that's all. Depending on your needs, you might want more information than just a "-1" displayed.

For instance, perhaps you want the offending command displayed for an eyeball inspection:

```
if liResult = -1
  messagebox(Command failed::' + chr(13) ;
    + lcStr + chr(13) ')
else
  messagebox(lcDesc +  ' result:' ;
    + str(liResult) + ' in ' ;
    + lcElapsed + ' sec.')
endif
```

It can be pain to examine a long command to determine just went wrong. Another enhancement would be to parse the command and offer to retry with a simpler version, say, no WHERE clause, or an * instead of a limited field list. Sure, these might impact performance on the full data set, but if you're running tests on the smaller SOMEDATA table, this is likely a negligible concern.

Another scenario you might run into is where the command runs fine on the test data set but fails on the full data set. Consider doing a SELECT DISTINCT on the field/fields in the WHERE clause, in order to determine if there are problems with some of the values in the domain. Perhaps a few of the rows in the full data set contain NULLs but the test data set doesn't?

Conclusion

SQLite's small footprint, flexibility with data types, and blinding speed make it an excellent mechanism for parsing large data sets. If you don't already have it in your tool chest, you should!

Author Profile

Whil Hentzen is a independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but frighteningly few since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com