Inserting Large Amounts of Data into SQLite

Whil Hentzen

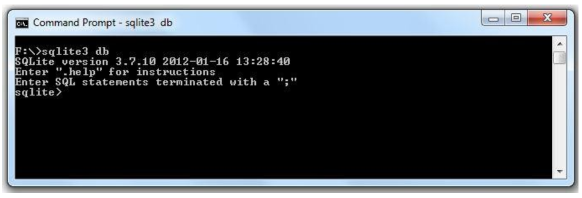
"We need to get away from DBFs" is a refrain I hear regularly from fellow developers. Be it due to perceived instability of the file format, the need for tables larger than 2 GB, or the result of political machinations, the result is the same – a desire to move to a SQL database back-end. SQLite can 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 this article, I'll show you how to work with SQLite directly in order to quickly insert large amounts of data into a SQLite database, and then how to export data from SQLite into other data file formats.

SQLite has native capabilities for importing and exporting data from its tables. These are handier and faster to use than writing code in VFP to import large quantities of pre-defined data. In order to do so, you need to use SQLite separately from Visual FoxPro, specifically, the SQLite Command Line Shell.

You know how one of the beauties of VFP is that you can create a table, add records, and query subsets of data simply by typing a few commands into the Command Window? SQLite is more like VFP than any other Client-Server tool in providing the same type of ease of use. The Command-Line Shell, available from the SQLite website, is the VFP Command Window equivalent.

Installing SQLite

VFP includes the Command Window as part of the vfp.exe executable. You can look at this in another way, that the executable for the VFP Command Window also includes the VFP data engine. Similarly, you can think of the SQLite Command-Line Shell as containing the SQLite engine, and you wouldn't be far wrong.
So simply by firing up the Shell, you've got the SQLite data engine running and available to you. To do so, put the sqlite3.exe file anywhere you want and execute it. For example, you can call it from a Windows Shell, as shown in **Figure 1**.

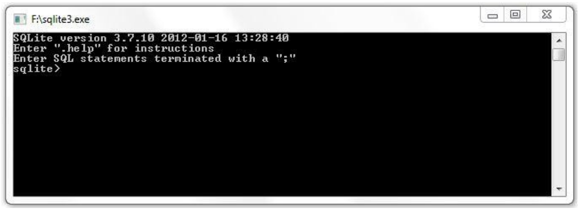


Command Prompt - sqlite3 db

```
F:\>sqlite3 db
SQLite version 3.7.10 2012-01-16 13:28:40
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Figure 1. Calling SQLite from a Windows Shell.

An advantage to calling SQLite from a Windows Shell is that you can include the name of the database you want to work with and have the database attached (SQLite's equivalent of VFP's 'open') automatically. Figure 1 shows the database 'db' being included in the call to the sqlite3 EXE. An alternative way of calling SQLite is by double-clicking on the file in a file manager. The result is shown in **Figure 2**.



F:\sqlite3.exe

```
SQLite version 3.7.10 2012-01-16 13:28:40
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Figure 2. The Command-Line Shell for SQLite.

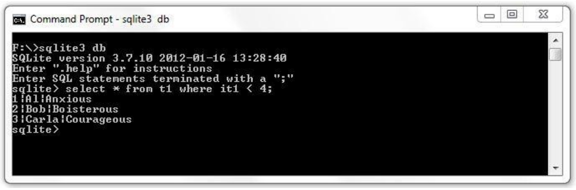
In this situation, you'll have to execute a command at the sqlite> prompt to attach a database – I'll cover this in a moment.

The SQLite Command Line Shell

The Command-Line Shell (referred to simply as the "Shell" from now on) works similar to the VFP command Window – you type commands and it echoes results. Two immediate differences; first, you need to terminate SQL commands with a semi-colon, like so:

```
sqlite> select * from table;
```

and second, results are returned inline with the command prompt as shown in **Figure 3** (like the DOS prompt and Windows Command Shell), as opposed to VFP that echoes results to the VFP desktop.



Command Prompt - sqlite3 db

```
F:\>sqlite3 db
SQLite version 3.7.10 2012-01-16 13:28:40
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from t1 where it1 < 4;
1¦Al¦Anxious
2¦Bob¦Boisterous
3¦Carla¦Courageous
sqlite>
```

Figure 3. Results are returned inline.

Types of Shell commands

Above, I mentioned that SQL commands need to be terminated with a semi-colon. Why didn't I simply say 'commands'? Because there are two types of commands you can execute in the Shell.

The first are standard SQL commands that are passed through to the SQLite Library for execution. The usual suspects include SELECT, CREATE INDEX, DROP TABLE, and so on. You can type a command all on one line, or you can press Enter whenever you like, and the Shell will display a second line that starts with a new prompt, like so:

```
sqlite> select it1, cnafirst, cnalast
sqlite> from t1
sqlite>where cnalast = 'Smith'
sqlite>
```

When you're done, type a semi-colon, press Enter, and the command will be executed.

The second type of commands are interpreted directly by the SQLite Shell program itself, and are identified by preceeding the command by a dot. Some of these commands control control how the Shell works, similar to VFP's SET commands. For example,

```
.header
```

controls whether headers are displayed in the output of a SELECT command. Others provide specialized data handling, or display information about the shell.

Importing text files

The .import command is the one we're interested in so that we can import text files.

.import dot command

The .import Command-Line Shell dot command imports a text file into a table. To use it, you'll need a source text file that is formatted to match the structure of the target table. The data in the text file has to follow these rules:

- For each desired row in the table, there must be a separate line in the text file.
- For each field in the table, there must be a data element (or a placeholder) in each line.
- Each data element must be separated from the next by the character designated as the separator. The default separator is the pipe (|), but can be changed by the .separator Command-Line dot command.

Thus, for the table created with this command:

```
sqlite> create table t1 (first_name,
last_name, room_no);
```

The text file would have to look like this:

Listing 1. t1_input.txt

```
al|weird|105
bob|barker|122
cher||999
donna|summer|
edgar|winter|400
madonna||999
```

Note that lines 3 and 6 do not have a value in the last name field, but there are two separators, creating a placeholder for the last name field. Line 4 does not have a value in the room number field, but there is a separator, again creating a placeholder for the room number field.

In the Shell, you'd then use the command:

```
sqlite> .import t1_import.txt t1
```

If you don't have the right number of separators in the text file, you'll get an error message like so:

```
sqlite> .import t1_import.txt t1
Error: t1.txt line 6: expected 3 columns of
data but found 2.
sqlite>
```

and no data will be imported. Blank rows at the end of the text file will similarly throw an error, as the blank row will be interpreted as not having enough columns.

Speed

The .import dot command is blindingly faster than using VFP and ODBC. I inserted 1.000,000 short records (three fields) using VFP. It took 16 and a half hours. (ODBC is slow.) Then I exported those million records into a flat file, then used the .import dot command to add those records to a SQLite table. The import command took six seconds.

Exporting data

You can get data out of a SQLite database and into a different file several ways without using VFP.

Redirect via .output

The first way is to simply redirect the output of SQL commands to a file, then execute the SQL command itself. This technique has the advantage of being able to pull just the selected data out of the database. For example,

```
sqlite> .output an_output_file.txt
sqlite> select * from t1;
sqlite> .output stdout
```

The first command defines the name of the text file that all output from the Shell will be directed to. The second is the command itself - the one that creates the data set. And the third redirects Shell output back to the screen. If you don't issue the third command (.output stdout), future commands will echo back to the an_output_file.txt file as well. For instance, try this:

```
sqlite> .output an_output_file.txt
sqlite> select * from t1;
sqlite> .output stdout
```

When you open up the "an_output_file.txt" file, you'll see the results of the SELECT command :

Listing 2. Results of .output being sent to a text file.

```
al|weird|105
bob|barker|122
cher||999
donna|summer|
edgar|winter|400
madonna||999
```

Create list of SQL commands

The .dump dot command will create SQL commands for every row in the specified table. For example, to "dump" table t1:

```
sqlite> .dump t1;
```

The results will look like this:

Listing 3. Results of the .dump dot command.

```
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE t1 (cnaf, cnal, room_no);
INSERT INTO "t1" VALUES('al','anxious',NULL);
<many rows>
INSERT INTO "t1" VALUES('madonna','','999');
COMMIT;
```

Having the results spit to the screen may not be that useful. Instead, how about sending them to a file:

```
sqlite> .output t1_output.txt
sqlite> .dump t1
sqlite> .output stdout
```

And the results of the .dump command will end up in the t1_output.txt file, a format much more usable.

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 none since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com