This paper was originally presented at the Southwest Fox conference
in Gilbert, Arizona in October, 2019. http://www.swfox.net

SQLite for the VFP Developer

Rick Borup
Information Technology Associates, LLC
701 Devonshire Dr., Suite 127
Champaign, IL 61820
www.ita-software.com
rborup@ita-software.com

SQLite is a lightweight, open-source, cross-platform, single-file SQL database engine that's perfect for many situations where you want to store and retrieve information quickly and reliably. SQLite databases are widely used by many software applications, often behind the scenes; in fact, it's likely you already have at least one SQLite database on your computer without even knowing it. As a Visual FoxPro developer, you can leverage the power and convenience of SQLite in your own applications, too. Come to this session to learn about SQLite and how to use it.

Introduction

SQLite is a free relational database management system (RDMS) packaged as a C-language library and distributed in a single file. It is fast, serverless, highly reliable, requires no configuration, has a small footprint (less than 1MB on disk), and is cross-platform.

The developers of SQLite placed the source code in the public domain, so no licensing is required and there are no restrictions on its use. SQLite implements most of the DDL, DML, and TCL language feature of the SQL92 standard, making it instantly familiar to anyone who has worked with SQL in other relational database management systems.

Reasons to use SQLite

Simplicity

SQLite requires no installation and no configuration. Simply copy the library to the desired location on a local drive and you're ready to go.

Cost

SQLite is free and requires no license.

Speed and reliability

SQLite's motto is "Small, fast, reliable. Choose any three."

Standards

SQLite conforms to the SQL92 standard, so it is familiar to anyone with experience in other SQL-based relational database management systems.

Reasons not to use SQLite

Security

SQLite does not have any mechanism to implement security via usernames or passwords. There is no login – access to a SQLite database is controlled only by file system permissions. Therefore, SQLite is not a good solution for storing sensitive data nor for any situation in which more robust methods and/or more granular levels of access control are required.

Concurrent access

SQLite has a locking mechanism that enables concurrent access to a database from multiple connections. This mechanism permits parallel read operations, but requires exclusive access to the database file for write operations. If a write operation returns an error due to the database file being locked, the application can wait—normally a very short time—until the lock is released and try again. However, SQLite does not come with any built-in support for client-server style access by multiple computers over a network. It is therefore not a good choice for applications where there is an elevated possibility of update conflicts due to high transaction rates and/or multiple concurrent updates. See https://www.sqlite.org/lockingv3.html for more information.

Copyright 2019, Rick Borup

Database size

Although SQLite databases can be multiple gigabytes in size and are not subject to the 2-gigabyte limit of Visual FoxPro, there is a practical limit to how big one can grow before performance degradation becomes an issue. Performance is affected by many factors in addition to the size of the database file, so there is no hard and fast rule here.

SQLite databases you may already have

Because SQLite is used "behind the scenes" by so many applications, it's very likely there are already some SQLite databases on your computer you don't even know about. One way to discover them is to search your local file system for *.sqlite or *.db, two of the most common file name extensions for SQLite database files. Here are some examples:

Microsoft® Visual Studio[1] stores some information in SQLite databases located in *%localappdata%\Microsoft\VisualStudio\vshub\Settings*.

The **Firefox web browser** makes extensive use of SQLite databases to store information about the user's profile. If Firefox is installed on your computer, look for SQLite databases in *%appdata%\Mozilla\Firefox\Profiles\<randomString>.default* and its subfolders. Some examples are the *cookies.sqlite*, *favicons.sqlite*, and *formhistory.sqlite* database files, which are interesting to explore.

Evernote, the popular note taking app, stores the local copy of your notes and other content in a SQLite database. Look for the file with your Windows username and a .exb file extension in the *%userprofile%\Evernote\Databases* folder.

Syncfusion Metro Studio, a free program to create custom icons from a set of base icons supplied with the product, uses SQLite internally to store the keyword references that facilitate searching its icons. If you have Metro Studio installed on your computer you can find the *keywords.sqlite* database in *%localappdata%\Syncfusion\Metro Studio*.

You can find a list of some well-known companies and popular apps that use SQLite at www.sqlite.org/famous.html and in the "Who uses SQLite" section on the home page of www.w3resource.com/sqlite/.

A word about pronunciation

As with anything SQL-related, there is always the question of whether it's pronounced "sequel" or "ess-que-el". The book *Using SQLite*, which was one of my primary references for this paper, refers to "an SQLite database", suggesting it's pronounced "ess-que-el lite". In the Microsoft world, however, SQL Server is generally pronounced "sequel server" and as a Visual FoxPro developer I favor the Microsoft pronunciation. I therefore tend to think of SQLite as "sequel lite" so I refer to "a SQLite database" in this paper. FWIW, Microsoft Word disagrees and flags this as a grammatical error – go figure.

[1] Microsoft Visual Studio 2017 Community Edition

Installing SQLite on Windows

Getting started is easy. Go to the download page at sqlite.org/download.htm and locate the section title Precompiled Binaries for Windows. The package you want for starters is the SQLite Tools bundle (see Figure 1), which contains *sqlite.exe* along with two other files. *Sqlite.exe* is the SQLite application, a command-line shell built on top of the SQLite core database engine. You can use the command-line shell to interact with the SQLite database engine from a command window.

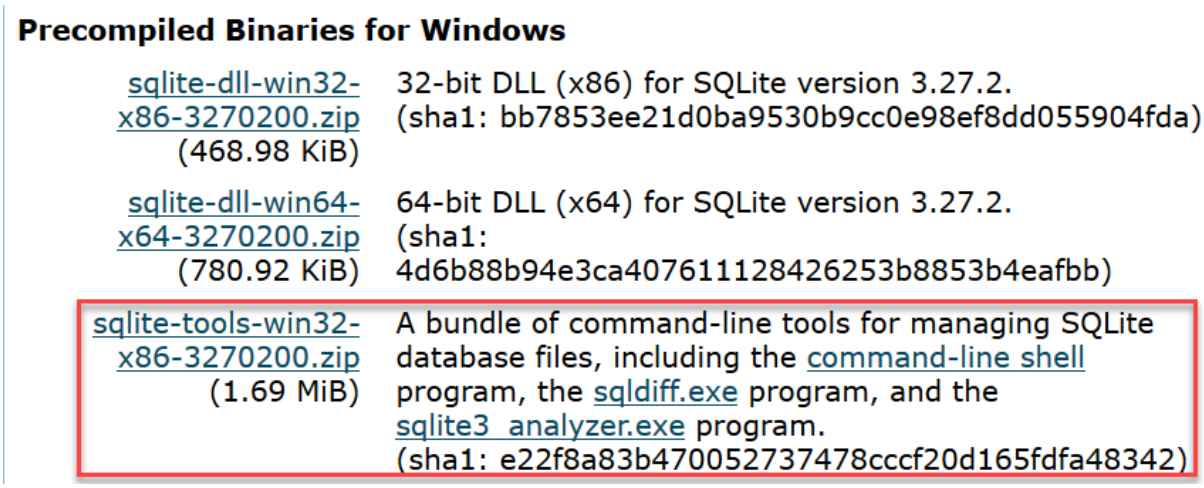


Figure 1: Download the zip bundle containing the command-line shell from the SQLite download page.

Create a new root folder on your hard drive and extract the contents of the zip file into it; the conventional location is C:\SQLite. That's it – there's no setup, no installer, nothing to register.

Open a command window, CD to the folder where you installed SQLite, and type *sqlite3* at the prompt. SQLite responds with its version number and other information.

Listing 1: Type "sqlite3" at the command prompt to display the SQLite version and other information.

```
C:\SQLite>sqlite3
SQLite version 3.27.2 2019-02-25 16:06:06
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

SQLite databases are stored in a single file. These are typically saved with a .db file name extension, but .sqlite work equally well as do others including no file extension at all. In any case, you do not need an existing SQLite database file to work with SQLite from a command window. When you run SQLite from the command prompt without specifying a database file name, as shown in Listing 1, it connects to an in-memory database you can use as a

sandbox for learning and exploration. If you decide you want to keep your work, you can use the *.save* dot-command[2] to write the in-memory database to a permanent file.

The SQLite command line interface

SQLite comes with a built-in command line interface, aka a shell. The shell understands two types of commands: ordinary SQL statements and SQLite dot-commands. Both types can be entered at the *sqlite>* prompt in the command window.

SQL statements such as CREATE TABLE, SELECT, UPDATE, DELETE, etc., can span more than one line and must be terminated with a semi-colon. Dot-commands are used to control SQLite configuration options; they begin with a period, must be contained on a single line, and do not require a semi-colon at the end. Appendix A is a listing of SQLite dot-commands.

Creating a new table

As a first exercise, let's create a new table. The table will be created in the transient (in-memory) database that's created when *sqlite3.exe* is run from a command window without specifying an existing database, as shown in Listing 1. We'll name it the *people* table and define three columns: an integer primary key, a last name, and a first name. The code looks like this:

Listing 2: SQL statements must end with a semi-colon. This statement is all one line in the command window.

```
sqlite> create table people ( pk integer, lastName varchar(20), firstName
varchar(20));
```

Unlike the SQLite dot-commands, SQL statements must end with a semi-colon. If you enter a SQL statement, omit the semi-colon, and press the Enter key, SQLite does not flag it as an error. Instead, it assumes you want to continue the statement on a new line and provides an ellipsis prompt for the continuation. This is handy for avoiding line overflow when entering longer commands, but it can trip you up at first if you're not expecting it. The code in Listing 2 could also be entered on multiple lines like this:

```
sqlite> create table people (
   ...> pk integer,
   ...> lastName varchar(20),
   ...> firstName varchar(20)
   ...> );
```

Now that we have a table, let's insert some data.

```
sqlite> insert into people values (1, 'Adams', 'Amy');
sqlite> insert into people values (2, 'Baker', 'Bob');
sqlite> insert into people values (3, 'Cooper', 'Carol');
```

[2] The SQLite shell understands a set of commands that begin with a period. These are called dot-commands. For example, the *.help* command lists all the dot-commands and the *.save <filename>* command saves the current database to a file.

We can now run a SQL SELECT statement to verify the results. By default, the output from the query is sent to stdout (standard output, i.e., the command window) and is delimited with pipe characters.

```
sqlite> select * from people;
1|Adams|Amy
2|Baker|Bob
3|Cooper|Carol
```

So far, all of this exists only in memory. If you want to save your work, run the ".save" command and supply the desired database file name. If you don't specify a path the file is created in the current folder.

```
sqlite> .save test.db
```

If you do want to specify a path, use the conventional folder hierarchy syntax with forward slashes. If the current folder is C:\SQLite and you want to save the file to a different folder, use forward slashes to specify the folder hierarchy, like this:

```
sqlite> .save ../path/to/folder/test.db
```

Alternatively, you could use the ".cd" dot-command to change the working directory to the folder of your choice before saving.

```
sqlite> .cd ../path/to/folder
sqlite> .save test.db
```

When you're finished with the current session, use the ".quit" command to return to the Windows command prompt.

```
sqlite> .quit
C:\SQLite>
```

Working with the results of a query

When you're programmatically fetching data from any SQL database, the results of the query are accessible to your program for further processing in whatever form the language you're using makes available. In Visual FoxPro, for example, we're accustomed to getting the results back in the form of a VFP cursor named *query,* unless something different was specified in the SELECT statement's INTO clause.

How does this work with SQLite? If you're using Python, you create a cursor object to hold the results. If you're using the Visual FoxPro QODBC driver you get a VFP cursor, as we'll see in later examples. But what happens when you run a query from the SQLite command-line interface? Unlike Visual FoxPro, the SQLite's SELECT statement does not have an INTO clause. So where do the results of a query end up and how do you work with them?

SQLite's default behavior is simply to display the query results in the command window – useful to look at but not very useful for anything else.

Nonetheless, there are ways you can exercise some control over the results of a query run from the command line. One way is to use the *.mode* dot-command to specify the format of the output in conjunction with the *.output* dot-command to determine where the output is written.

For example, say we want the result to be a text file in some suitable format such as tab-delimited or comma-separated values (csv). Listing 3 shows how to select data from the *people* table and send the output to a csv file.

Listing 3: The results of a SQL query can be formatted as CSV and sent to a file.

```
sqlite> .mode csv
sqlite> .output people.csv
sqlite> select * from people;
```

This creates a file named *people.csv* whose content looks like this:

```
1,Adams,Amy
2,Baker,Bob
3,Cooper,Carol
```

Listing 4 is the same as the above except the output is sent to a tab-delimited file named *people.txt*.

Listing 4: The results of a query can also be formatted as a tab-delimited file.

```
sqlite> .mode tabs
sqlite> .output people.txt
sqlite> select * from people;
```

The content of the resulting *people.txt* file is shown below, with the » character inserted here just to make the tabs visible:

```
1» Adams» Amy
2» Baker» Bob
3» Cooper»   Carol
```

The *.mode* dot-command can also be set to configure the output as an HTML table.

```
sqlite> .mode html
sqlite> .output people.html
sqlite> select * from people;
```

The result is an HTML table suitable for copying into a <table> element in an HTML document.

```
<TR><TD>1</TD>
<TD>Adams</TD>
<TD>Amy</TD>
</TR>
<TR><TD>2</TD>
```

```
<TD>Baker</TD>
<TD>Bob</TD>
</TR>
<TR><TD>3</TD>
<TD>Cooper</TD>
<TD>Carol</TD>
</TR>
```

Copying the contents of a table to another table

Here's another example to illustrate what can be accomplished from the command line interface. This one takes advantage of a couple of other SQLite dot-commands options to copy the contents of a table to another table within the same database.

Assuming the target table does not already exist, the first step is to create it.

```
sqlite> create table foo (
   ...> pk integer,
   ...> lastName varchar(20),
   ...> firstName varchar(20)
   ...> );
```

The *.mode* dot-command supports an *insert* mode. In this mode, the output of a query is not a set of rows but rather a set of INSERT statements. If a table name is specified in the *.mode* dot-command it is used as the target table name in the INSERT statements.

The second step in this example is to set the mode to *insert* with a table name of *foo*. The output is a set of SQL INSERT statements, so let's send the output to a file named *insert.sql*.

```
sqlite> .mode insert foo
sqlite> .output insert.sql
sqlite> select * from people;
```

The contents of *insert.sql* look like this:

```
INSERT INTO foo VALUES(1,'Adams','Amy');
INSERT INTO foo VALUES(2,'Baker','Bob');
INSERT INTO foo VALUES(3,'Cooper','Carol');
```

The example is trivial, but think about the power of this technique for a table with dozens of columns and hundreds or thousands of rows. It can potentially save a lot of typing.

The third step is to use the *.read* dot-command to execute the SQL statements in the *insert.sql* file.

```
sqlite> .read insert.sql
```

Before this command is run, table *foo* has no rows because it's a newly created table. Afterwards, the contents of table *foo* are identical to the contents of the *people* table. This can be verified by running a query after first restoring *mode* and *output* to their default values.

```
sqlite> .mode list
sqlite> .output stdout
sqlite> select * from foo;
1|Adams|Amy
2|Baker|Bob
3|Cooper|Carol
```

You have probably already figured out that the *.read* dot-command can read and execute *any* file containing valid SQL statements, which opens up a lot of possibilities.

The *.tables* dot-command lists the tables in the current database. As expected, there are now two tables in *test.db*.

```
sqlite> .tables
foo     people
```

Another way to accomplish the same thing is to use the *.dump* dot-command. This command, which optionally takes the name of a table as a parameter, generates SQL statements to both create and populate the table(s) in the database.

Listing 5 shows the command to generate the SQL statements to create and populate the *people* table, along with its output.

Listing 5: The *.dump* dot-command generates the SQL statements necessary to create and populate a table.

```
sqlite> .dump people
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE people (
pk integer primary key asc,
lastName varchar(20),
firstName varchar(20)
);
INSERT INTO people VALUES(1,'Adams','Amy');
INSERT INTO people VALUES(2,'Baker','Bob');
INSERT INTO people VALUES(3,'Cooper','Carol');
COMMIT;
```

To create table *foo* from this code, simply modify all references to *people* to refer to *foo* and then run the code, for example by saving the output to a .sql text file and using *.read* to execute it.

SQLite metadata

SQLite databases store metadata in internal structures known as catalogs. Catalog names begin with the prefix *sqlite_*. Every SQLite database has at least one catalog named *sqlite_master*, which is the master record of the database objects. Table 1 shows the structure of the *sqlite_master* catalog table.

Table 1: The structure of the *sqlite_master* catalog table.

Column Name	Type	Description
type	text	type of database object
name	text	object ID
tbl_name	text	name of associated table
rootpage	integer	internal use
sql	text	SQL used to define the object

SQLite catalogs can be queried like any other table. One way to explore the structure of a SQLite database is to run a *SELECT * FROM sqlite_master* query. Where the *.tables* dot-command lists the tables in the database, querying the *sqlite_master* table also reveals the structure of each table. Listing 6 shows the results of running this query on the *test.db* database created in the examples above. Note that the output is in the form of the SQL statements necessary to create each table.

Listing 6: The content of the *sqlite_master* table includes the code necessary to create the table..

```
sqlite> select * from sqlite_master;
table|people|people|3|CREATE TABLE people (
pk integer primary key asc,
lastName varchar(20),
firstName varchar(20)
)
table|foo|foo|4|CREATE TABLE foo (
pk integer,
lastName varchar(20),
firstName varchar(20)
)
```

Using the structure in Table 1 as a guide, you can interpret the results of the query and see that this database contains two tables (*people* and *foo*) with each table having the column names, sizes, and data types shown.

Another way to explore the structure of a SQLite database is to run the *.schema* dot-command, which shows the CREATE statements. This command takes an optional parameter to limit the output to a pattern like a single table name.

```
sqlite> .schema people
CREATE TABLE people (
pk integer primary key asc,
lastName varchar(20),
firstName varchar(20)
);
```

Copyright 2019, Rick Borup

Use the *--indent* option for a bit nicer-looking output.

```
sqlite> .schema --indent people
CREATE TABLE people (
  pk integer primary key asc,
  lastName varchar(20),
  firstName varchar(20)
);
```

Note that the database containing the table(s) whose schema you want to see must be open in order to run the *.schema* command.

Working with multiple SQLite databases

SQLite makes it possible to work with more than one database open at the same time. This is useful if you want to perform queries that reference data in two different databases.

As an example, let's fire up the SQLite command-line and open the *test.db* database.

```
C:\SQLite>sqlite3
sqlite> .open /path/to/test.db
```

A shortcut is to pass the name of the database as a parameter to the *sqlite3* command.

```
C:\SQLite>sqlite3 /path/to/test.db
```

The *.open* dot-command closes the current database and opens the one specified in the command.[3] The *.databases* dot-command lists the names and filenames of the attached databases. Running this command shows that only one database is open.

```
sqlite> .databases
main: test.db
```

The ATTACH DATABASE command attaches another database file to the current instance of SQLite3. Place the filename of the database in quotes and provide a schema name in the 'as' clause.

```
sqlite> attach database 'test2.db' as db2; -- the word 'DATABASE' is optional
```

The ".databases" dot-command now shows there are two attached databases.

```
sqlite> .databases
main: test.db
db2: test2.db
```

With the *test2.db* database now attached as *db2*, we can run queries like the following…

[3] If you specify a database file name that doesn't exist, SQLite creates an empty file with that name. I frequently forget to add the ".db" file name extension and write ".open test" thinking that, as in VFP, this will open the *test* database. In fact, what it does is create an empty database with file name *test* and no extension.

```
sqlite> select * from db2.people;
1|Adams|Amy|female
2|Baker|Bob|male
3|Cooper|Carol|female
4|Davis|David|male
```

… and we can run a query that references data in both databases, for example asking for all rows that have a matching last name in both tables.

```
sqlite> select * from people where lastName in ( select lastName from db2.people);
1|Adams|Amy
2|Baker|Bob
3|Cooper|Carol
```

The row for David Davis exists in the *people* table in *test2.db* but not in *test.db* so the result, as expected, is the set of three rows that appear in both.

The *.open* dot-command closes all currently attached databases and opens the one specified. If two or more databases are attached and you want to detach one of them individually, use the DETACH DATABASE command.

```
sqlite> detach database 'db2'; -- the word 'DATABASE' is optional here, too
```

Running Windows commands from within SQLite

The SQLite *.shell* dot-command provides access to the operating system shell from within SQLite. This is useful in several situations.

Earlier, when we saw that the *.cd* dot-command can be used to change the current working directory, you may have noticed that the SQLite command-line prompt did not show the path. In other words, regardless of what the current folder is, the SQLite prompt is always the same.

```
sqlite>
```

This is nice if you're working in a folder with a long path because you're not starting from the middle or the far-right edge of the line when you type a command. However, if you're moving around a lot between folders, it may be difficult to remember which one you're working in. You can use the *.shell* dot-command to find out, like this:

```
sqlite> .shell cd
C:\path\to\current\folder
```

Similarly, you can call the DIR command to find out what's in the current folder. The output is the same as running the DIR command from the Windows command prompt.

```
sqlite> .shell dir *.db
```

When you're using the SQLite command-line interface you'll quickly find yourself working from the bottom line in the command window. There may be times when you'd like to get

clear the screen at start again from the top. There is no "clear screen" command in SQLite, but you can use the *.shell* dot-command to run CLS.

```
sqlite> .shell cls
```

Summary

The SQLite command line is a good learning tool, although it's probably most frequently used to experiment with queries in the same way we as VFP developers might use the VFP command window. But as you can see, there are certainly some useful things you can do with it.

The SQLite3 Utility programs

In addition to the SQLite command-line interface, the SQLite3 tools bundle comes with two utility programs: *SQLite3 analyzer* and *SQLDiff*.

SQLite3 analyzer utility

The SQLite3 analyzer utility is primarily intended to generate information about space is used by the database, but it also offers another way to explore the basic structure of a SQLite database. If you installed the SQLite3 tools bundle as described earlier, you'll find the *sqlite3_analyzer.exe* in the same folder as *sqlite3.exe*.

Like the command shell, the SQLite3 analyzer utility is run from a command prompt. It generates a large amount of information that scrolls by quickly in the command window, so it's a good idea to capture the output to a file as shown below.

```
C:\SQLite>sqlite3_analyzer test.db >analysis.txt
```

Open *analysis.txt* in Notepad or any text editor and have a look at the information generated by the analyzer. Among other things, it shows the names of the tables in the database including the *sqlite_master* catalog table.

```
*** Page counts for all tables with their indices ****************************
FOO.......................................... 1           33.3%
PEOPLE....................................... 1           33.3%
SQLITE_MASTER................................ 1           33.3%
```

To give you an idea of the level of detail available, the full content of this *analysis.txt* file is included in Appendix B.

SQLDiff utility

The third file that comes with the SQLite3 tools bundle is *sqldiff.exe*, a command-line tool that displays the difference between two SQLite databases. To illustrate how this works, we'll start by making a copy of the *test.db* database and calling it *test2.db*. Making a complete copy of a SQLite database requires no special tools—simply use the file system to create a copy of the original database file with a different name.

Initially, *test2.db* is identical to *test.db*. Run the following commands to make some modifications to the structure and content of the *people* table in *test2.db*…

Listing 7: Add a column for gender and populate its values.

```
sqlite> .open test2.db
sqlite> alter table people add column gender char(10);
sqlite> update people set gender = 'male' where lastName = 'Baker';
sqlite> update people set gender = 'female' where is null gender;
sqlite> insert into people values (4, 'Davis', 'David', 'male');
```

… and do a SELECT to verify the results.

```
sqlite> select * from people;
1|Adams|Amy|female
2|Baker|Bob|male
3|Cooper|Carol|female
4|Davis|David|male
```

Now that the *people* table in the *test2.db* database has different structure and content than the *people* table in the *test.db* database, we can run *sqldiff.exe* to see how it works. Note that sqldiff.exe is a separate file so it's run from the Windows command prompt, not from the *sqlite>* prompt. The code in Listing X assumes the two database files are in some folder other than C:\SQLite.

```
sqlite> .quit
C:\SQLite>sqldiff ../path/to/test.db ../path/to/test2.db
```

The default output from the SQLDiff utility is a set of the SQL statements necessary to transform the source database into the destination database. This not only illustrates the differences but also provides a script you can run if you actually *do* want to transform the source into a duplicate of the destination. Listing 8 is the SQLDiff output from comparing *test.db* to *test2.db*.

Listing 8: The SQLDiff utility generates the code necessary to transform one database into another.

```
ALTER TABLE people ADD COLUMN gender;
UPDATE people SET gender='female' WHERE pk=1;
UPDATE people SET gender='male' WHERE pk=2;
UPDATE people SET gender='female' WHERE pk=3;
INSERT INTO people(pk,lastName,firstName,gender) VALUES(4,'Davis','David','male');
```

Note that while the SQL statements in Listing 8 accomplish the same result, they are not the same as the SQL statements we ran to modify the *people* table in *test2.db*. In Listing 7, when we modified the structure of the database manually, we chose to insert the gender for Bob Baker by selecting on the *lastName* field and then to insert the gender for the other two rows by selecting on gender not null. In contrast, the SQL statements generated by the SQLDiff utility insert the gender for all three rows by selecting on their primary key. Different SQL statements, same results.

The SQLDiff utility accepts options to alter its output. For example, the *--table* option limits the output to the differences between the specified table, as shown in Listing 9.

Listing 9: The *--table* option tells SQLDiff to look only at the specified table.

```
C:\SQLite> sqldiff --table people ../path/to/test.db ../path/to/test2.db
```

In this case, the only differences between these two databases are in the *people* table so the output is the same as without the *--table people* option.

The *--schema* option displays the differences in the schema but not the differences in content.

```
C:\SQLite> sqldiff --schema ../path/to/test.db ../path/to/test2.db
ALTER TABLE people ADD COLUMN gender;
```

See www.sqlite.org/sqldiff.html for a complete reference to the SQLDiff utility.

How SQLite implements SQL

Notable differences

As noted earlier, SQLite implements most of the SQL92 standard but there are some differences and omissions. Among these are:

- SQLite supports RENAME TABLE, ADD COLUMN, and RENAME COLUMN but does not support DROP COLUMN, ALTER COLUMN, ADD CONSTAINT, and other variations of ALTER TABLE.
- SQLite supports LEFT [OUTER], INNER, and CROSS joins but does not support RIGHT or FULL joins.
- SQLite support views but they are read-only.
- SQLite is "flexibly typed" (their term for it). For example, it does not throw an error if you try to store a string to a column defined as INTEGER, although the resulting value depends on the string.
- SQLite does not have a true Boolean datatype. Instead, it uses integer value 1 for true and 0 for false. As of version 3.23.0, SQLite recognizes the keywords TRUE for 1 and FALSE for 0. See the section on SQLite datatypes for more information.
- SQLite does not have a true date or datetime datatype. Instead, dates and times can be stored as a string in several formats including ISO-8610, with or without the time portion (e.g., '2019-04-27 12:47:00' or just '2019-04-27')[4], an integer as the number

[4] See https://en.wikipedia.org/wiki/ISO_8601 and https://xkcd.com/1179/

of seconds since midnight UTC on January 1, 1970 (e.g., 1556387220)[5], or as a real value that is a fractional Julian day number (e.g., 2458600.727083).[6]

However, SQLite does provide date and time functions such as date( ), time( ), and datetime( ).[7] The datetime() function returns a string formatted as YYYY-MM-DD HH:MM:SS, which can be stored in a text field. SQLite allows this field to be defined as type *datetime*, so the name and data format match up nicely with VFP's *datetime* data type. See Listing 10 for an example.

- SQLite accepts either single-quotes or double-quotes around string literals.
- SQLite implements an AUTOINCREMENT option for primary keys but it doesn't necessarily work as VFP developers might expect. See www.sqlite.org/autoinc.html for more information.

Listing 10: SQLite's datetime() function can be used to store dates in a format that matches VFP's datetime.

```
sqlite> create table foo ( invoicedate datetime);
sqlite> insert into foo values ( datetime());
sqlite> select * from foo
2019-10-02 21:32:46
```

SQLite datatypes

The values stored in a SQLite database are defined as belonging to one of five *storage classes*: NULL, INTEGER, REAL, TEXT, or BLOB. A storage class comprises one or more type names, or datatypes, that can be used when defining columns in a CREATE TABLE statement. For example, the CHARACTER, VARCHAR, and TEXT data type names fall into the TEXT class while the INTEGER, SMALLINT, and BIGINT data type names fall into the INTEGER class. The amount of storage allocated to column on disk depends on the datatype within its class.

SQLite comparison operators

SQLite implements the standard set of SQL comparison operators including "=", "==", "<", "<=", ">", ">=", "!=", "<>", "IN", "NOT IN", "BETWEEN", "IS", "IS NOT", and "LIKE".

SQLite Studio

As useful as the SQLite command line interface is, there are times—probably *most* of the time—when you'd rather use a graphical user interface. This section shows how to install and use the SQLite Studio, a free GUI for SQLite.

[5] See https://www.epochconverter.com/
[6] See https://aa.usno.navy.mil/jdconverter
[7] See https://www.sqlite.org/lang_datefunc.html

Installing SQLite Studio

Download a Windows distribution of SQLite Studio from sqlitestudio.pl. The most recent update as of this writing is v3.2.1 released on 2018-07-27. You can choose the conventional Windows installer or a stand-alone zip file for portable use. If you choose the zip file, note that its contents are organized under a root folder named SQLiteStudio so you can extract the zip file to the root of a drive letter, like C:\, and everything will be in C:\SQLiteStudio. The executable file for SQLite Studio is *SQLiteStudio.exe*.

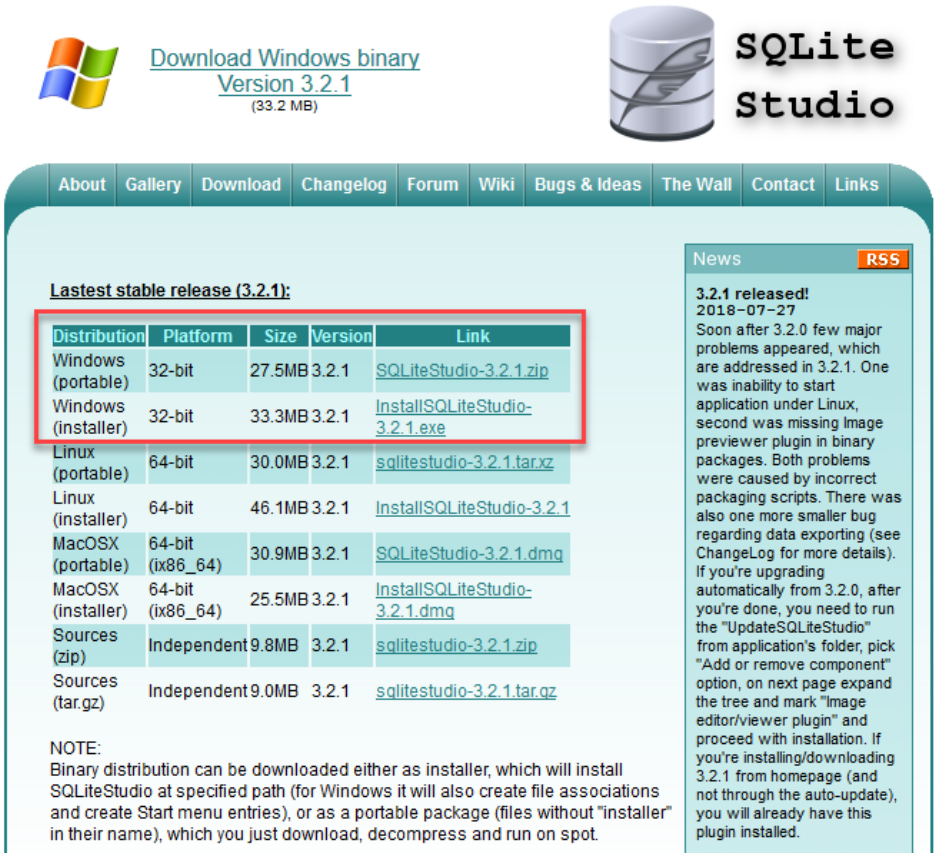


Figure 2: SQLite Studio for Windows can be downloaded either as a conventional installer or as a zip file.

Adding a database to SQLite Studio

The SQLite Studio window features a conventional menu and toolbar, with a list of databases along the left margin and a main display area to its right. On first run, no databases are open and none are listed. Before you can open (connect to) a database you need to add it to the list. Click Database on the main menu and choose “Add a database”. This opens the Database window from which you can select the desired SQLite database (see Figure 3).

The *Name* field is automatically populated with the name of the database file (without the file extension) but you can change this to something more meaningful if you want to. In this example the name was changed to *SWFox2019 Test Database*. Mark the *Permanent* check box if you want to keep this database in the list for future use. Click OK to finish.

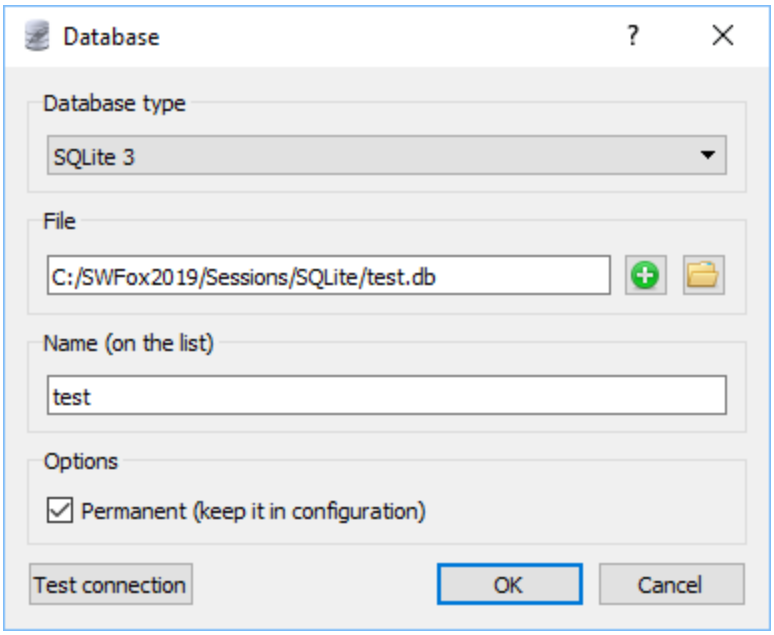


Figure 3: Add a database to SQLite Studio and mark the *Permanent* check box to keep in the list for future use.

Connecting to a database

To connect to a database in the list, either double-click on its name or select it and choose *Connect to the database* from the Database menu. SQLite Studio displays a hierarchical list of the tables and views in the database. The list can be expanded to show columns, indexes, and triggers for the selected table, as shown in Figure 4. Right-clicking an icon in the list opens a context-sensitive popup menu that provides access to several actions such as create, edit, and delete.

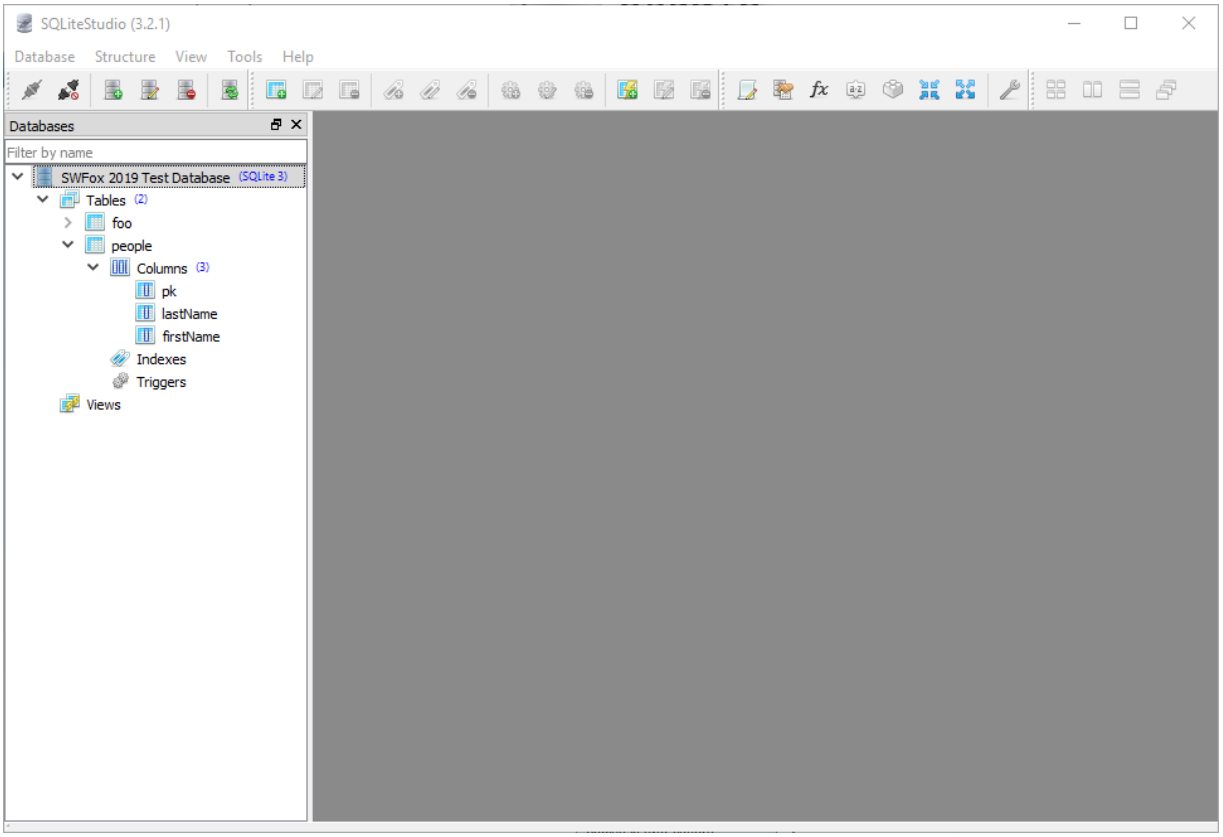


Figure 4: When connected to a database, the list on the left can be expanded to show its tables and views.

Using the SQL Editor to execute queries

The SQL Editor enables you to work write SQL statements and view results from within SQLite Studio, much in the same way SQL Server Management Studio does in Microsoft SQL Server. To begin, choose *Open SQL editor* from the SQLite Studio Tools menu.

Enter the desired SQL statements in the Query area of the SQL Editor. Click the blue arrowhead button on the SQL Editor toolbar or press F9[8] to execute the query. The results are displayed in the lower portion of the window, as shown in Figure 5.

[8] My first thought was, can I change this to F5? The answer is yes, but with a caveat. Whle SQLite Studio is highly configurable, F5 is the generic "refresh" action in several places so it's probably best to stick with F9 as the *execute SQL* hot key.

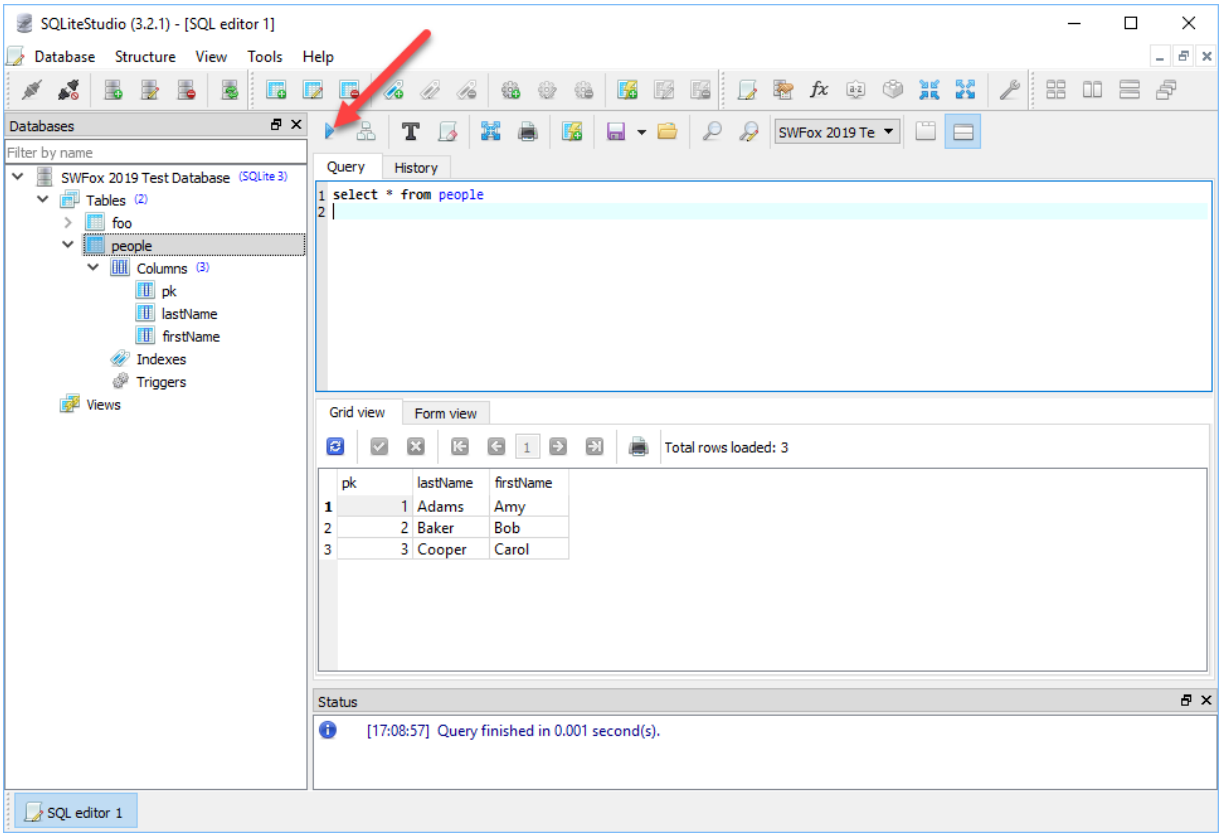


Figure 5: Click the blue arrowhead or press F9 to execute a query from the SQL Editor.

By default, the SQL Editor executes all the SQL statements in the Query window. As far as I can tell, there is no GO statement to serve as a batch separator like there is in Microsoft SQL Server Transact-SQL. However, there is an option to execute only the statement under the cursor.

The SQL statements in the Query window can be saved to a file by pressing Ctrl+S or clicking the *Save SQL to file* icon (the purple diskette) on the SQL Editor toolbar.

Exporting the results of a query

SQLite Studio provides an export feature to save the results of a query to a file in one of several formats. With the desired query in the Query window, click the *Export results* icon (the blue four-sided arrowheads in the shape of "X") to begin the export process.

The first step enables you to confirm the query to be executed or to edit it if needed. In the next step, select an export format, an output file name, and other options. Export formats are CSV, HTML, JSON, PDF, SQL, and XML.

In contrast to the HTML file generated from the command-line interface in the earlier example, the HTML file generated by the SQLite Studio export feature has several options and can be a rendered as a complete HTML document. Figure 6 shows an HTML file, including the formatting and the footnote, generated by SQLite Studio from the *SELECT * FROM people* query as displayed in a browser (image border added for display purposes).

#	pk integer	lastName varchar	firstName varchar
1	1	Adams	Amy
2	2	Baker	Bob
3	3	Cooper	Carol

Document generated by SQLiteStudio v3.2.1 on Fri Apr 19 17:47:07 2019

Figure 6: SQLite Studio can generate a fully formatted HTML document from a query.

Summary

This brief introduction to SQLite Studio is intended to give you a general idea of its usefulness as a GUI for SQLite. More information about how to configure and use SQLite Studio can be found in the *SQLite Studio User Manual* at github.com/pawelsalawa/sqlitestudio/wiki/User_Manual.

SQLite Reader browser add-on

Another tool, although more rudimentary, visual tool for working with SQLite databases is the free SQLite Reader add-on for Firefox, Chrome, and Opera web browsers.[9] Figure 7 shows SQLite Reader as rendered using the Firefox add-on. The large rectangular area at the top is actually a clickable button to open a SQLite database, or you can drag and drop a SQLite database onto that area from the File Explorer.

[9] I have used the SQLite Reader add-on in Firefox with no problems. Comments on this app's page on the Chrome Web Store page suggest some people have had issues with it in Chrome. I have not tried it in Opera.

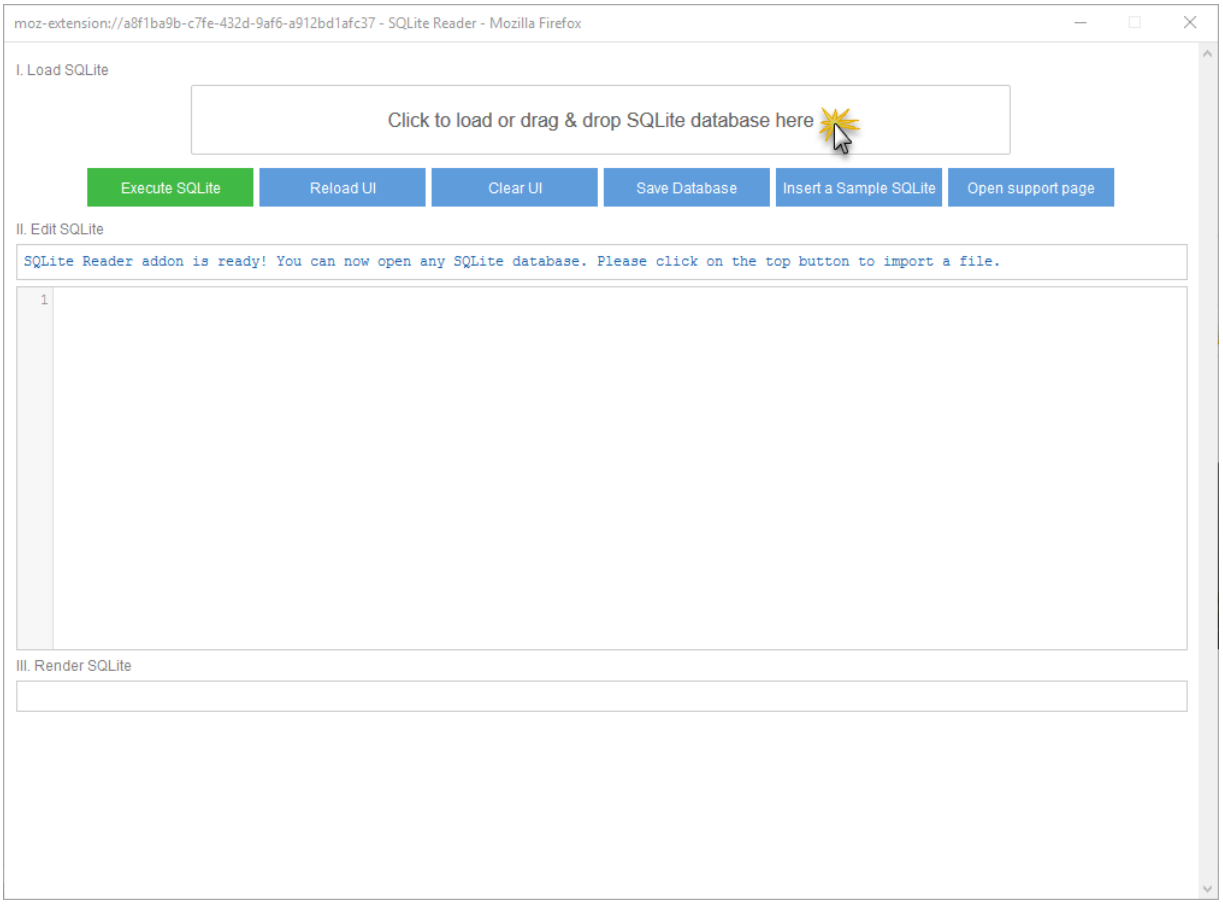


Figure 7: SQLite Reader is a free add-on for the Firefox web browser. Click the top button where indicated to open a SQLite database.

Dragging a SQLite database file onto the top area from File Explorer opens the database and runs a query on the *sqlite_master* table to display the structure of the database. Figure 8 shows the result after opening the *test.db* database.

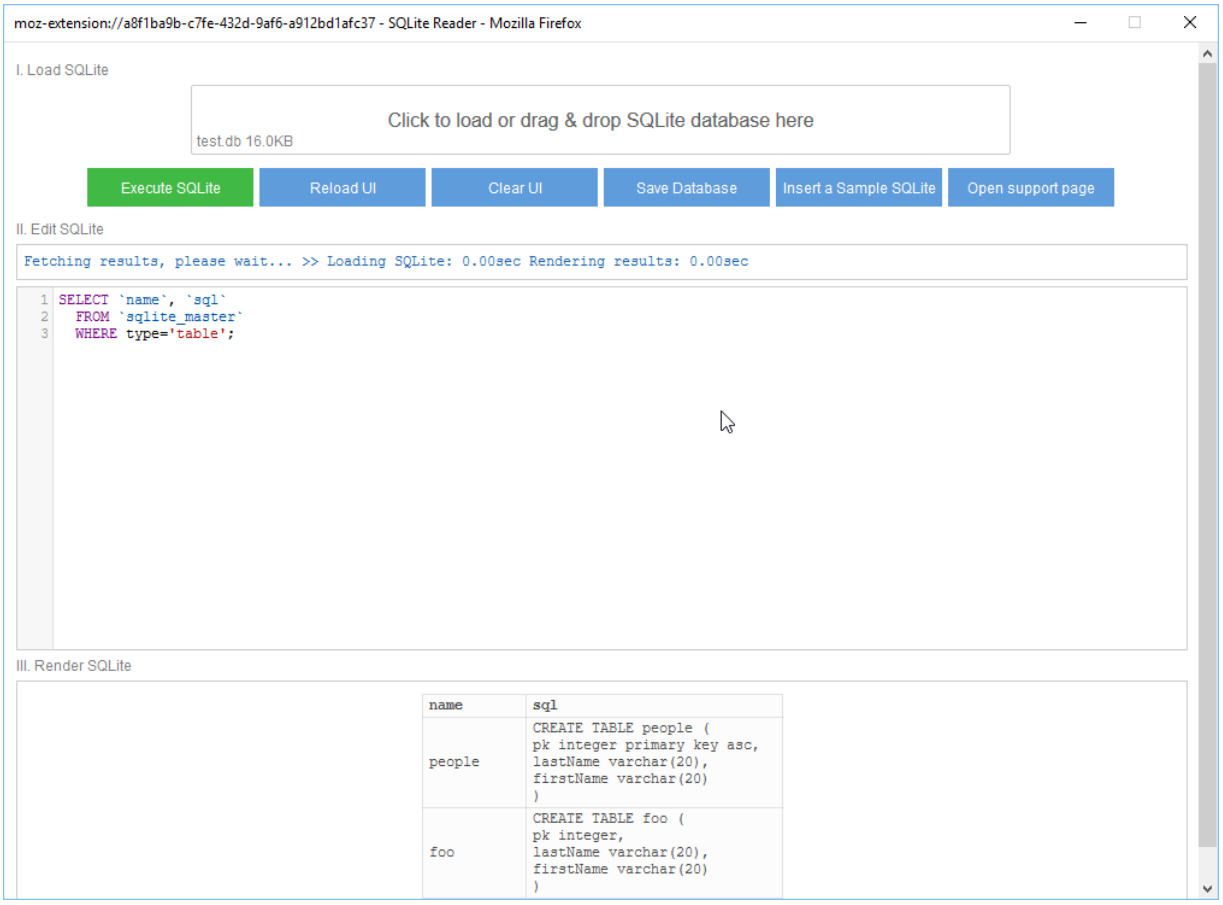


Figure 8: Opening a SQLite database in the SQLite Reader add-on for Firefox generates and runs a query on the *sqlite_master* table.

You can enter your own SQL commands in the Edit SQLite area and run them by clicking the *Execute SQLite* button, as shown in Figure 9.

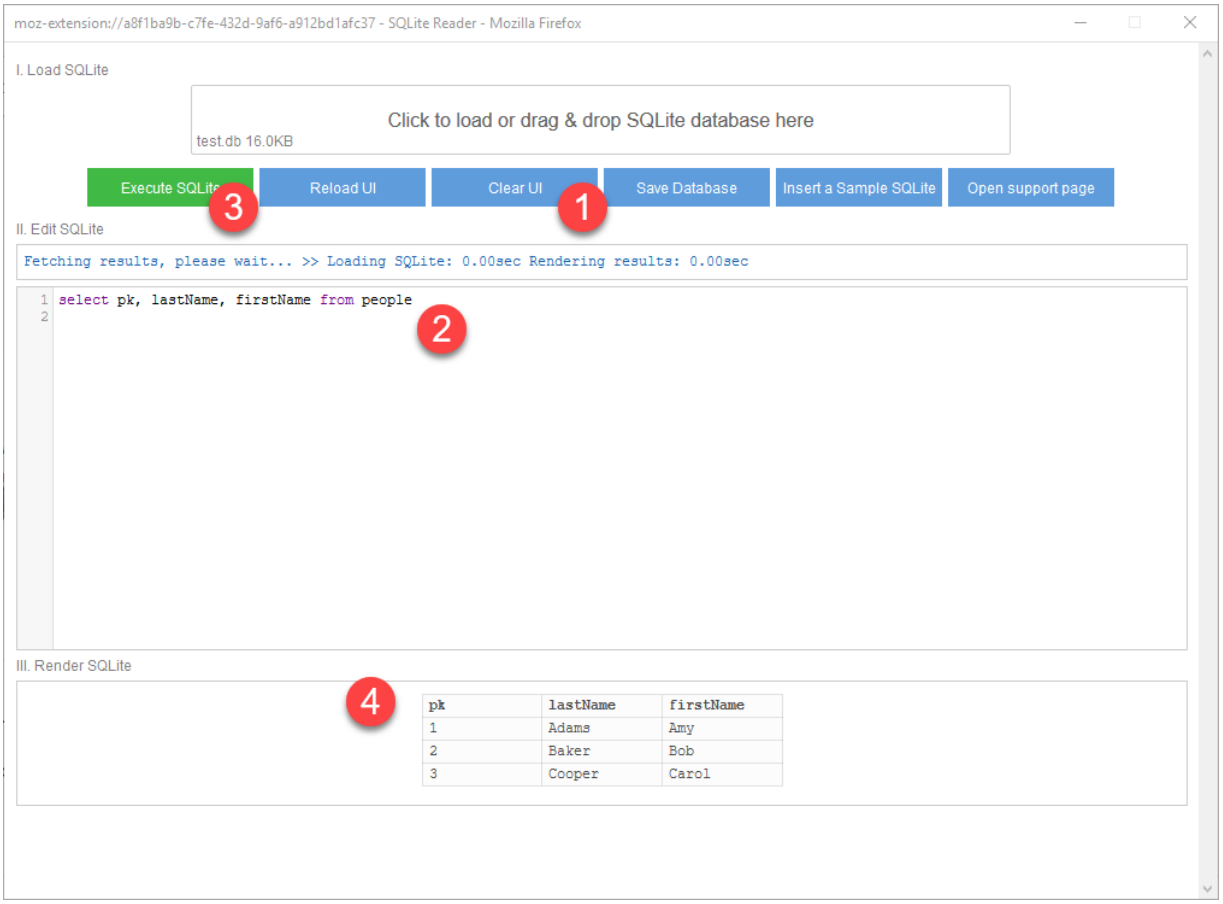


Figure 9: You can use SQLite Reader to enter and execute SQL statements.

Clicking the *Insert a Sample SQLite* button generates a SQL script in the Edit SQLite area to create and populate a table with sample data, as shown in Listing 11.

Listing 11, SQLite Reader can generate a table with sample data.

```
DROP TABLE IF EXISTS colleagues;
CREATE TABLE colleagues(id integer, name text, title text, manager integer, hired
                        date, salary integer, commission float, dept integer);
  INSERT INTO colleagues VALUES (1,'JOHNSON','ADMIN',6,'2011-12-17',18000,NULL,4);
  INSERT INTO colleagues VALUES (2,'HARDING','MANAGER',9,'2011-02-02',52000,300,3);
  INSERT INTO colleagues VALUES (3,'TAFT','SALES I',2,'2015-01-02',25000,500,3);
  INSERT INTO colleagues VALUES (4,'HOOVER','SALES II',2,'2011-04-02',27000,NULL,3);
  INSERT INTO colleagues VALUES (5,'LINCOLN','TECH',6,'2012-06-23',22500,1400,4);
  INSERT INTO colleagues VALUES (6,'GARFIELD','MANAGER',9,'2013-05-01',54000,NULL,4);
  INSERT INTO colleagues VALUES (7,'POLK','TECH',6,'2014-09-22',25000,NULL,4);
  INSERT INTO colleagues VALUES (8,'GRANT','ENGINEER',10,'2014-03-30',32000,NULL,2);
  INSERT INTO colleagues VALUES (9,'JACKSON','CEO',NULL,'2011-01-01',75000,NULL,4);
  INSERT INTO colleagues VALUES (10,'FILLMORE','MANAGER',9,'2012-08-
09',56000,NULL,2);
  INSERT INTO colleagues VALUES (11,'ADAMS','ENGINEER',10,'2015-03-15',34000,NULL,2);
  INSERT INTO colleagues VALUES (12,'WASHINGTON','ADMIN',6,'2011-04-
16',18000,NULL,4);
```

```
  INSERT INTO colleagues VALUES (13,'MONROE','ENGINEER',10,'2017-12-
03',30000,NULL,2);
  INSERT INTO colleagues VALUES (14,'ROOSEVELT','CPA',9,'2016-10-12',35000,NULL,1);

 SELECT name, hired FROM colleagues ORDER BY hired ASC;
 SELECT title, COUNT(*) AS count, (AVG(salary)) AS salary FROM colleagues
        GROUP BY title ORDER BY salary DESC;
```

If you want to run this script, I'd suggest doing it in a new instance of SQLite Reader so it uses an in-memory database rather than affecting whatever database you may already have opened. Use the *Save Database* button if you want to save your work; the file is saved as *sql.db* in the default download folder for your browser.

The *Open support page* button opens the webpage at https://mybrowseraddon.com/sql-reader.html with additional information about SQLite Reader.

DB Browser for SQLite

DB Browser for SQLite is a free, open-source tool that provides a visual interface to view, edit, and query SQLite databases. It is available for download from https://sqlitebrowser.org/ as a Windows installer or as a no-install zip file in both 32-bit and 64-bit formats. A portable app version is also available.

The default configuration of the *DB Browser for SQLite* interface after opening the *test.db* database is shown in Figure 10. Note that the *Database Structure* tab on the left side is selected.

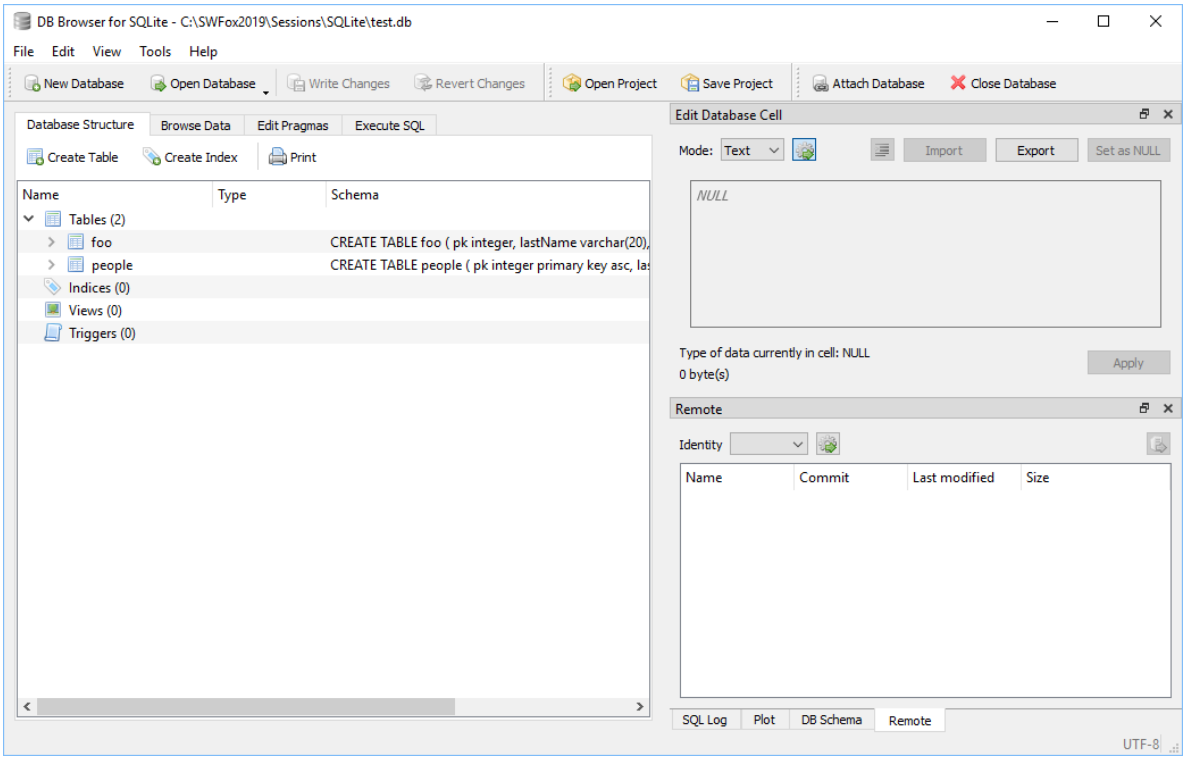


Figure 10: The default configuration of the DB Browse for SQLite interface after opening a database.

To view the contents of a table, select it in the *Database Structure* tree view and then select the *Browse Data* tab. This result is a SELECT * query, as shown in Figure 11.

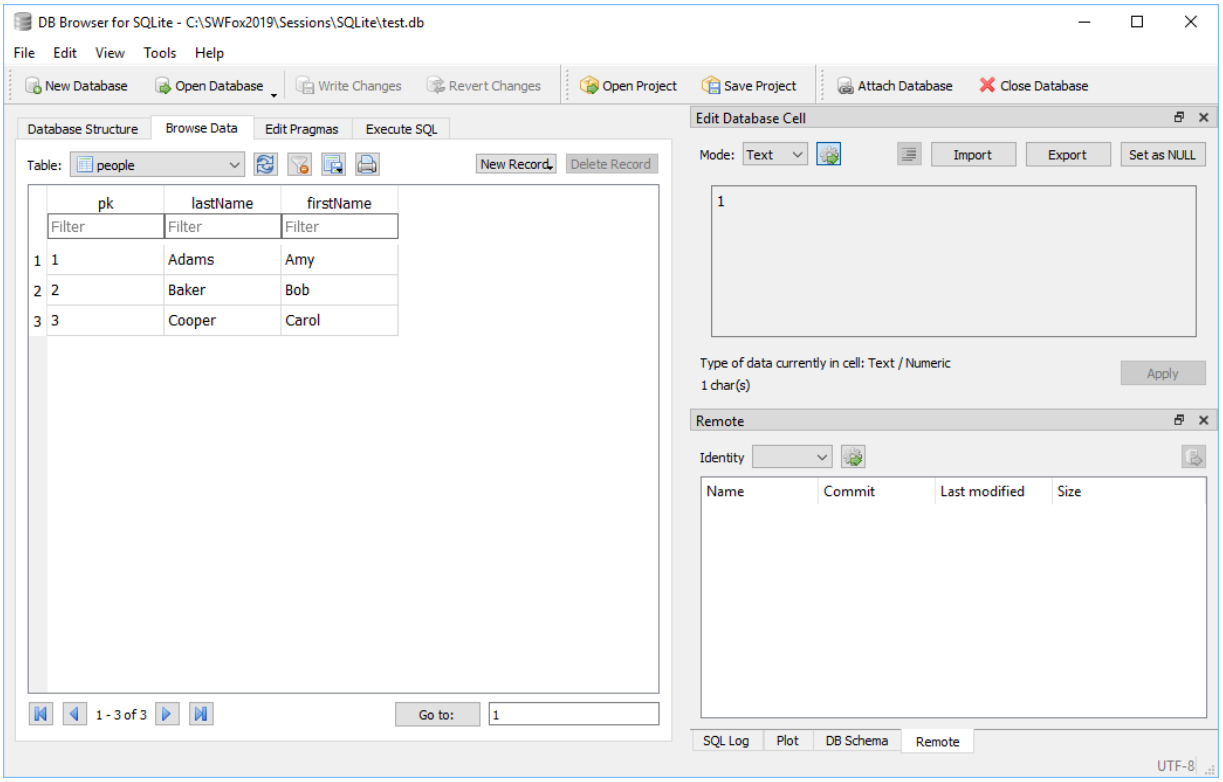


Figure 11: The Browse Data tab displays the contents of the selected table.

You can also write and execute your own SQL commands from the *Execute SQL* tab. Use the VCR-style arrow icons on this tab's toolbar to run your SQL command or commands. The F5 key also works here. Figure 12 shows how the result is displayed.

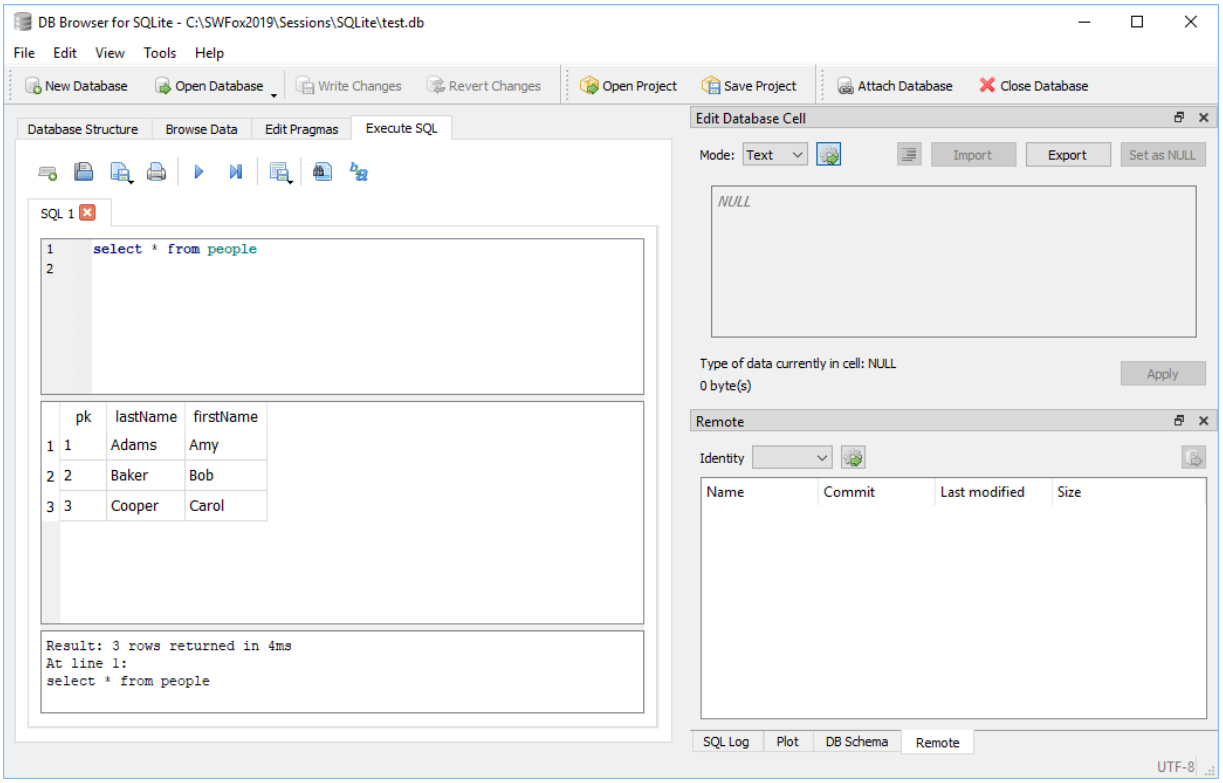


Figure 12: You can run your own SQL commands from the *Execute SQL* tab.

There is a lot more to this tool than I've explored here, including the import, export, and other features on the right side of the interface. More information is available on the Wiki, which you can get to by choosing Help | Wiki on the main menu or directly in your browser at https://github.com/sqlitebrowser/sqlitebrowser/wiki.

Sample SQLite database

Up to this point we've been working with *test.db*, a trivially small database created as a learning tool. SQLite is not limited to such small or single-table databases; it's capable of working with much more complex schemas involving multiple tables, views, indices, triggers, and even tables larger than the 2GB limit we're stuck with in Visual FoxPro.

One example of a more realistic database schema is the sample provided by the folks at the SQLite Tutorial website, a great resource with a wealth of examples and information about SQLite. You can download the sample database, named *chinook*, for your own use from www.sqlitetutorial.net/sqlite-sample-database/.

The *chinook* database has eleven tables. The database schema reproduced here in Figure 13 comes from the schema diagram provided by SQLite Tutorial.

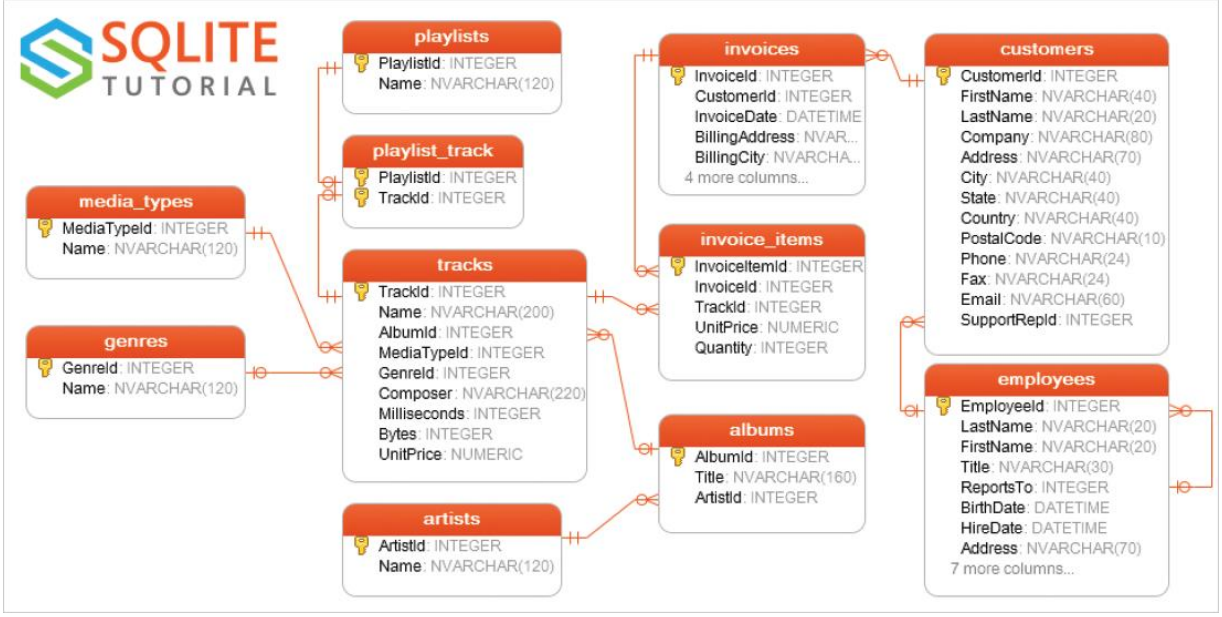


Figure 13: The chinook database from SQLite Tutorial has eleven tables.
Source: www.sqlitetutorial.net/wp-content/uploads/2018/03/sqlite-sample-database-diagram-color.pdf

Figure 14 shows the chinook database open in SQLite Studio, where you can see some of the tables have at least one index.

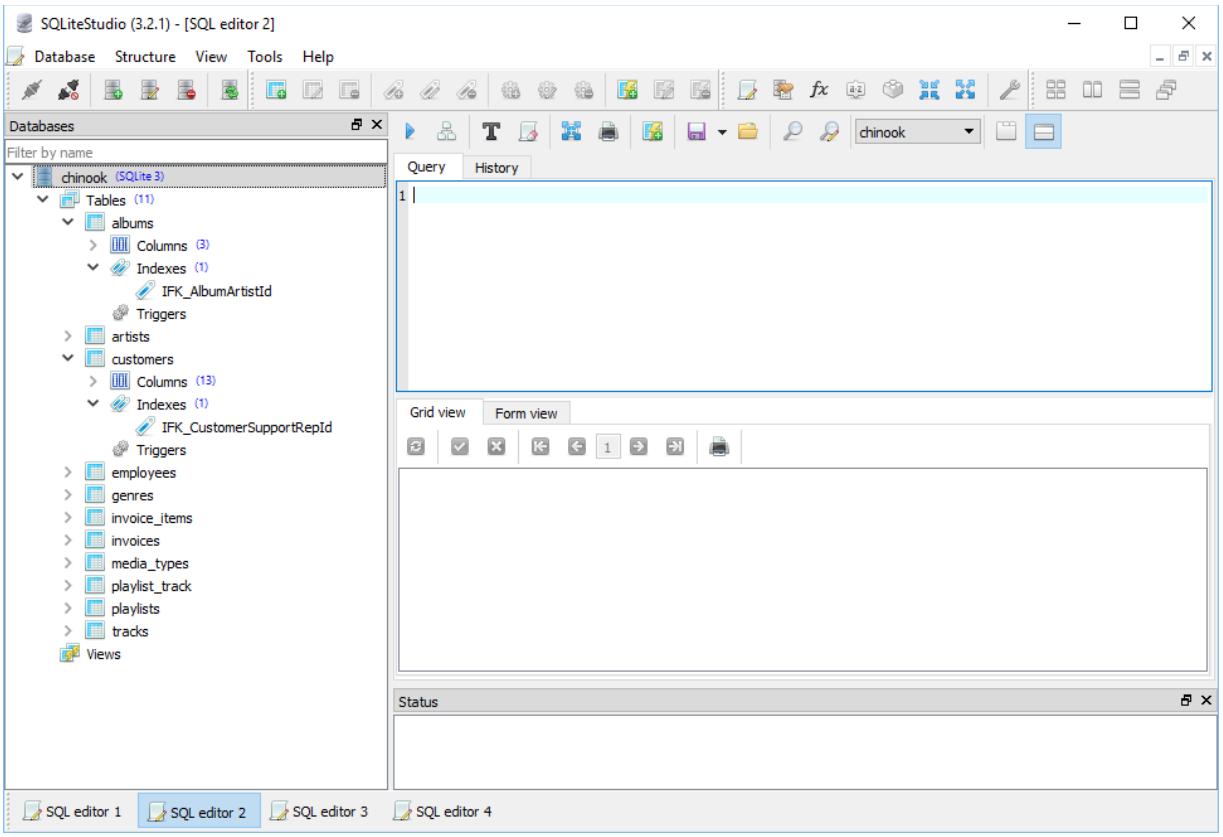


Figure 14: Opening the chinook database in SQLite Studio shows that some of the tables are indexed.

Using the schema information in Figure 13, you can construct queries to pull data from the tables in this database and explore the results. Figure 15 shows a query to pull an alphabetical listing of customers and their invoices, along with the first few rows of results.

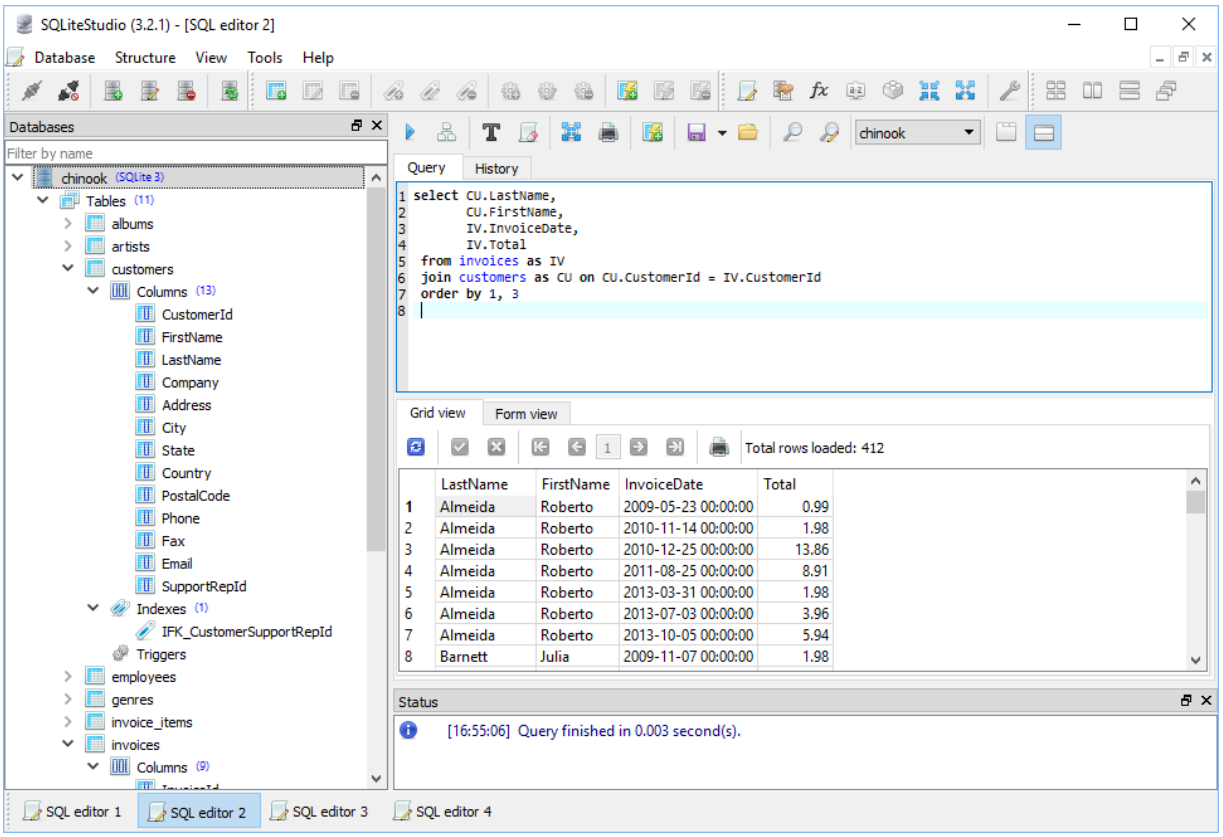


Figure 15: This query generates an alphabetical listing of customers and their invoices.

Note the syntax for the table aliases in lines 5 and 6 in Figure 15. Although included here for the sake of completeness, the AS keyword is optional like it is in VFP. However, unlike VFP, IN is not acceptable as an alias for the invoices table because it's a keyword.

The SQLite ODBC driver

To interact with SQLite databases from Visual FoxPro you need to install the SQLite ODBC driver, which is available from www.ch-werner.de/sqliteodbc/. There are 32-bit and 64-bit versions of the driver. As noted on that webpage, if you're using 32-bit software you should install the 32-bit driver even on a 64-bit machine. This applies to us as Visual FoxPro developers because VFP is 32-bit software.

Installing the SQLite ODBC driver

The installer for the 32-bit driver is the file named *sqliteodbc.exe*. The current version of the driver as of this writing (April 2019) is v0.9996. The webpage has links for downloading earlier versions, but you'll probably want to choose the current one unless you have special requirements. Download and run the desired installer.

The default location for the 32-bit SQLite ODBC driver is *C:\Program Files (x86)\SQLite ODBC Driver*. After installing the driver that folder contains a couple of dozen files. One of them is sqlite3.exe, the SQLite 3 database engine. This may or may not be the same version you installed above but it doesn't matter, other than for you to be aware of which version of the database engine you're using when you use the SQLite ODBC driver.

To check the version installed by the ODBC driver, open a command window, CD to the SQLite ODBC Driver folder, and run *sqlite3* from the prompt (see Listing 12; The version of the SQLite database engine installed by the SQLite ODBC driver may not be the same as the most recent version available from the SQLite website itself.). The version of the database engine installed by the v0.9996 ODBC driver installer is 3.22.0, which is about a year older than the most recent version 3.27.2 available from the sqlite.org website.

Listing 12; The version of the SQLite database engine installed by the SQLite ODBC driver may not be the same as the most recent version available from the SQLite website itself.

```
C:\SQLite>cd \program files (x86)\sqlite odbc driver
C:\Program Files (x86)\SQLite ODBC Driver>sqlite3
SQLite version 3.22.0 2018-01-22 18:45:57
Enter ".help" for usage hints.
sqlite>
```

To confirm installation of the SQLite ODBC driver was successful, open the ODBC Data Source desktop app on your computer and click on the System DSN tab. You should see the SQLite3 Datasource listed as shown in Figure 16.

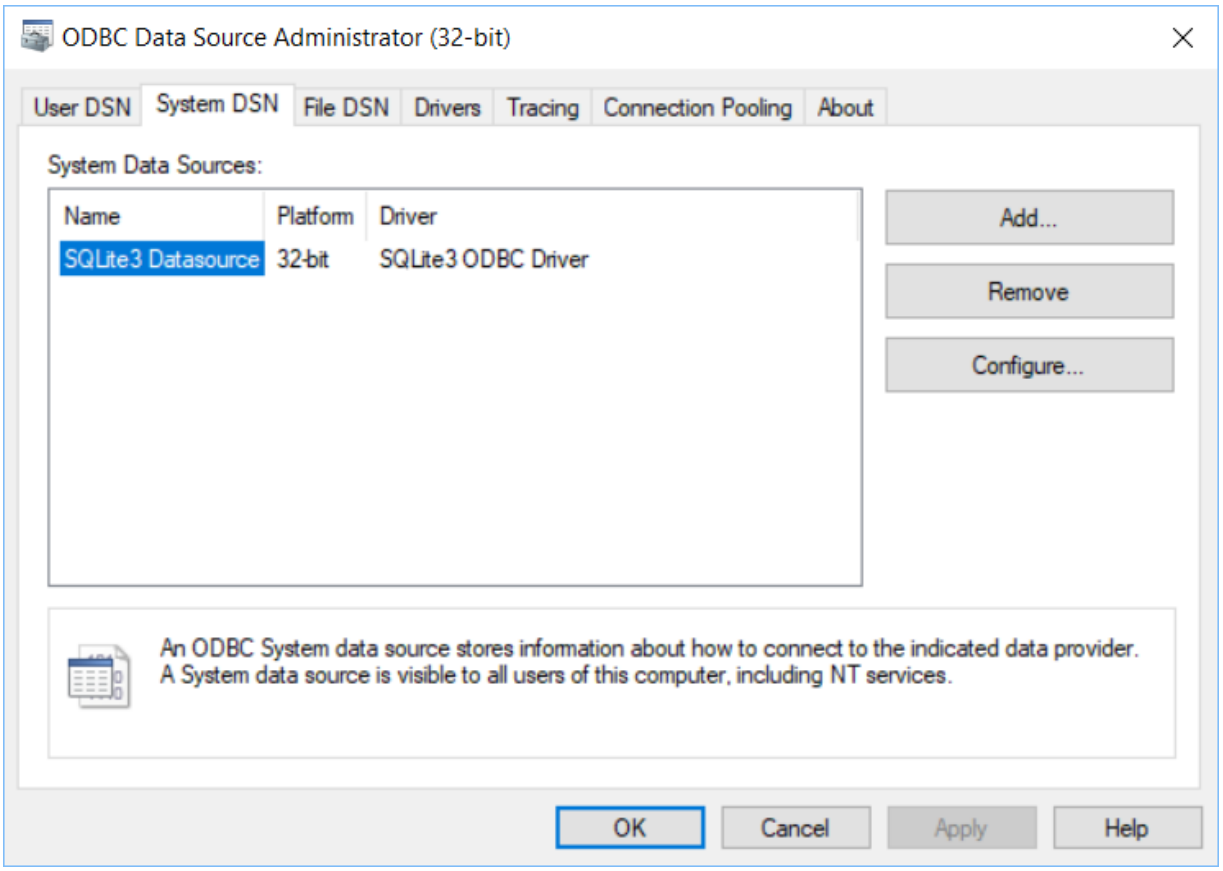


Figure 16: The SQLite ODBC Driver is listed on the System DSN tab of the 32-bit ODBC Data Source Administrator.

For our purposes we can leave all the configuration options at their default values but if you want to see what options are available, click the Configure button and have a look around as shown in Figure 17. Note that the field for the database name is empty: if you want to, you can tie this driver to a specific SQLite database but a more flexible approach is to specify the database name as an *ad hoc* value in the ODBC connection string.

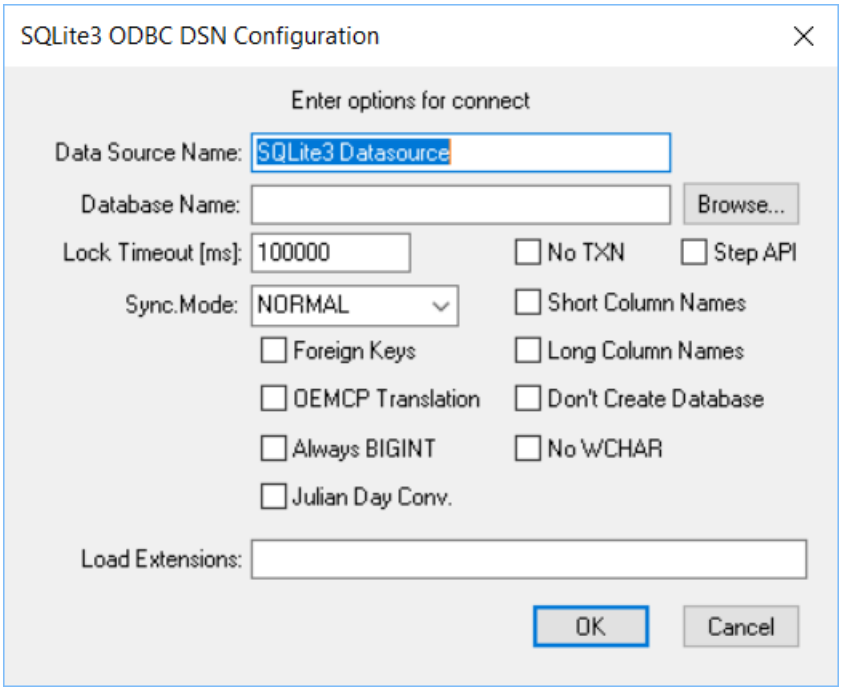


Figure 17: The SQLite3 ODBC DSN configuration options.

The SQLite ODBC driver implements most of the functionality you'll need to work with SQLite databases, but it's a good idea to read the documentation[10] for information and restrictions.

Using the SQLite ODBC driver with VFP

The easiest way to experiment with the SQLite ODBC driver from VFP is to use SQL Pass Through (SPT) with a DSN-less connection. Specifying the full path and file name of the desired SQLite database in the connection string enables you to switch from one database to another with a simple change.

Getting data from SQLite into VFP

Listing 13 shows the VFP code necessary to establish a DSN-less connection to the *test.db* database, construct a SQL statement to select the contents of the *people* table, execute the statement with SQLEXEC and place the results in a cursor named *csrPeople*, and then disconnect from the database.

Listing 13: The VFP code to connect to a SQLite database, select the contents of a table, and disconnect.

```
lcConnString = [DRIVER={SQLite3 ODBC Driver};DATABASE=C:\SQLite\test.db;]
lnConnHandle = SQLSTRINGCONNECT( lcConnString)
IF lnConnHandle = -1
   WAIT WINDOW "Failed to connect"
   RETURN
ELSE
```

[10] http://www.ch-werner.de/sqliteodbc/html/index.html

```
   WAIT WINDOW "Connected with connection handle " + TRANSFORM( lnConnHandle)
ENDIF
lcSQL = "Select * from people"
SQLEXEC( lnConnHandle, lcSQL, "csrPeople")
BROWSE TITLE "Data from SQlite test database" NOEDIT NOCAPTIONS
SQLDISCONNECT( lnConnHandle)
```

The Browse command returns the expected results, shown in Figure 18.

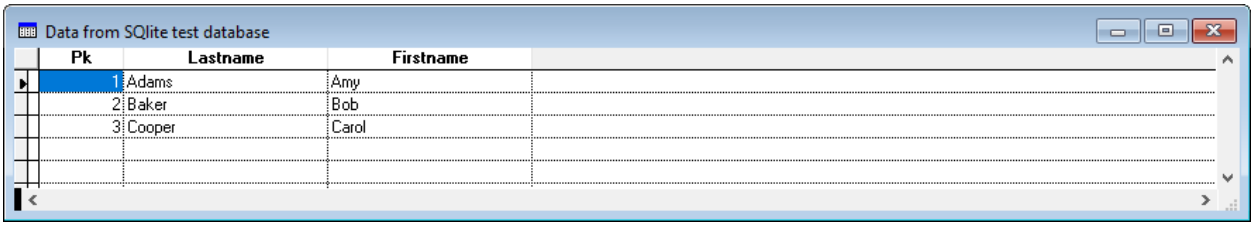


Figure 18: The cursor created by the SQLite ODBC driver shows the expected results from the *people* table.

Disconnecting from the database does not close the cursor, so we can still work with it.

Any time you're transposing data from one type of database to another you need to be cognizant of potential for data type conversions. Listing 14 is the output of the DISPLAY STRUCTURE command to explore the structure of the *csrPeople* cursor returned by the ODBC driver in response to this query.

Listing 14: The structure of the *csrPeople* cursor created by querying the SQLite database.

```
Structure for table:          C:\USERS\RICK\APPDATA\LOCAL\TEMP\0002WLXL01HG.TMP
Number of data records:       3
Date of last update:            /  /
Code Page:                    1252
  Field   Field Name    Type       Width  Dec  Index  Collate  Nulls  Next  Step
      1   PK            Integer        4                        Yes
      2   LASTNAME      Character     20                        Yes
      3   FIRSTNAME     Character     20                        Yes
** Total **                           46
```

In this example, the data types all come in pretty much as expected. The *PK* primary key field is an integer just as it is in the SQLite database. The *lastName* and *firstName* fields remained as strings but came in as C(20) in the VFP cursor even though they were defined as varchar(20) in SQLite.

Things get a little more interesting when we start working with date and logical value fields. Using the SQLite command-line interface, let's add a date column to the *people* table and then insert a new row with a value in that column.

```
sqlite> alter table people add column birthDate date;
sqlite> insert into people values ( 4, 'Davis', 'David', '1994-11-22');
```

Note that the date value is specified as a string in YYYY-MM-DD format.[11] After making these changes and re-running the query from VFP, we get the result shown in Figure 19.

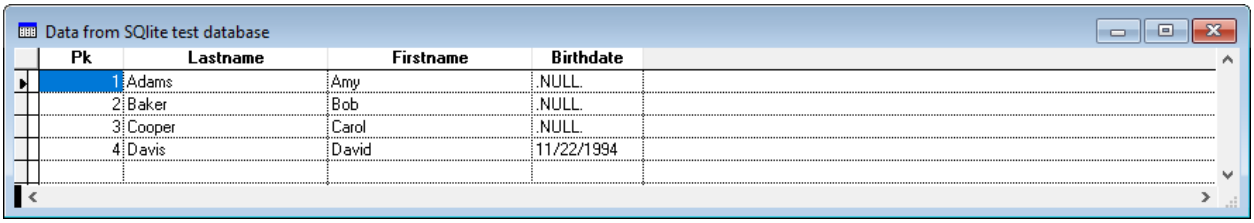


Figure 19: The SQLite database now has a date column, which comes in as a VFP date field in the cursor.

The good news is that the birth date column in the cursor created by the ODBC query is an actual VFP date field and contains the expected value in row 4 (see Listing 15). The bad news is that while that column contains the expected value in row 4, it contains NULLs instead of the empty date in rows 1 through 3. This is because there is no value in the birthdate column for rows 1 through 3 in the SQLite database.

Listing 15: The *birthdate* column in the SQLite database comes back as a VFP data field in the cursor.

```
Structure for table:          C:\USERS\RICK\APPDATA\LOCAL\TEMP\0002WLXL01LF.TMP
Number of data records:       4
Date of last update:            /  /
Code Page:                    1252
  Field   Field Name    Type       Width  Dec  Index  Collate  Nulls  Next  Step
      1   PK            Integer        4                        Yes
      2   LASTNAME      Character     20                        Yes
      3   FIRSTNAME     Character     20                        Yes
      4   BIRTHDATE     Date           8                        Yes
** Total **                          54
```

The cursor returned by the SQLite ODBC query is updateable, so if the NULLs are problematic for your situation you can easily replace them with the empty date in your VFP code.

```
REPLACE ALL birthDate WITH {} FOR ISNULL( birthDate)
```

Logical values present a similar challenge. Because there is no Boolean data type in SQLite, 'true' is represented by 1 and 'false' is represented by 0. It's customary to use the integer data type for true/false columns because it gives better performance than strings.

Let's add a true/false column to the people table so we can identify those who drink coffee and those who don't. The initial value for a new column is NULL, so the third line is just a quick way to set drinksCoffee false for all rows where drinksCoffee isn't marked true.

```
sqlite> alter table people add column drinksCoffee tinyint;
sqlite> update people set drinksCoffee = 1 where pk = 1 or pk = 2;
```

[11] If you specify the value for the date without the quote marks (i.e, as 1994-11-22), SQLite interprets it as a numeric expression, evaluates it, and inserts the result 1961 into the date field. Guess how I discovered that…

```
sqlite> update people set drinksCoffee = 0 where drinksCoffee is null;
```

Running the ODBC query again in VFP and browsing the cursor gives the results shown in Figure 20.

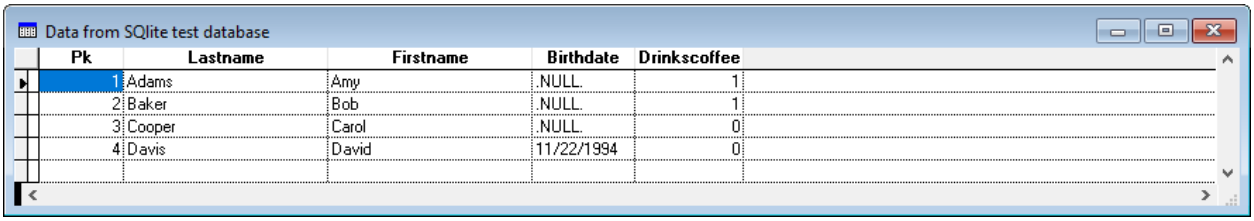


Figure 20: The results of the ODBC query after adding the true/false field in SQLite.

The structure of the cursor now shows that the *drinksCoffee* field is of course an integer.

Listing 16: The *tinyint* data type in SQLite converts to an integer data type in VFP.

```
Structure for table:          C:\USERS\RICK\APPDATA\LOCAL\TEMP\0002WLXL01U9.TMP
Number of data records:       4
Date of last update:            /  /
Code Page:                    1252
  Field   Field Name    Type       Width  Dec  Index  Collate  Nulls  Next  Step
      1   PK            Integer        4                        Yes
      2   LASTNAME      Character     20                        Yes
      3   FIRSTNAME     Character     20                        Yes
      4   BIRTHDATE     Date           8                        Yes
      5   DRINKSCOFFEE  Integer        4                        Yes
** Total **                          58
```

Because it's an integer value in the VFP cursor, we can't simply replace the value in the *drinksCoffee* column with logical true (.T.) or false (.F.). It seems to me there are two ways to deal with this in our VFP code.

The first is simply to leave it as is and to deal with integer values 0 and 1, either by comparing explicitly for 0 or 1 or by using something like `IIF(drinksCoffee = 1,.T.,.F.)` anywhere you want to use a logical expression.

The other way is to create a new cursor or table and add a column to store the coffee drinker information as a VFP logical value.

Listing 17: Add another column to be populated with a logical value equivalent to the value in the original *drinkscoffee* column.

```
CREATE CURSOR csrPeople2 ;
   ( pk I NULL, ;
     lastName c(20) NULL, ;
     firstName c(20) NULL, ;
     birthdate D NULL, ;
     drinksCoffee I NULL, ;
     lCoffeeDrinker L;
    )
```

```
APPEND FROM DBF( "csrPeople")
UPDATE csrPeople2 SET lCoffeeDrinker = .T. WHERE drinksCoffee = 1
UPDATE csrPeople2 SET lCoffeeDrinker = .F. WHERE drinksCoffee = 0
BROWSE NOEDIT NOCAPTIONS TITLE "People table with Boolean coffee drinkers"
```

This seems like the better approach because it avoids the need to use an inline workaround for the integer 0 and 1 values everywhere the column is referenced in your code.

As a final exercise, let's add a column to record the last time a row was updated. A datetime data type makes sense for this column.

```
sqlite> alter table people add column lastUpdated datetime;
```

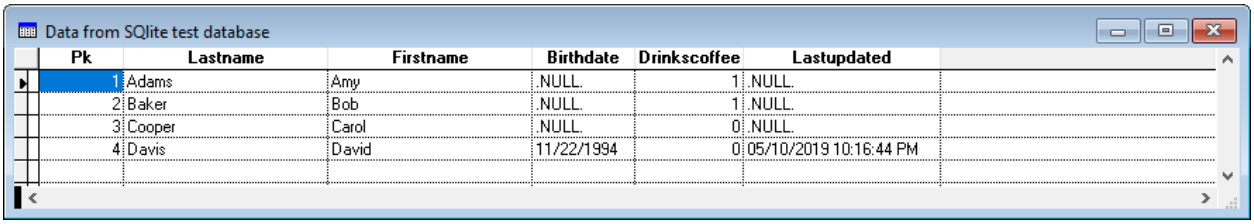
SQLite supports several date and time functions including *datetime( )*.[12] When no parameter is passed, *datetime()* returns the current date and time formatted as YYYY-MM-DD hh:mm:ss. We can use this value to populate the *lastUpdated* column in the database and view the results, as shown in Listing 18.

Listing 18: SQLite's *datetime()* functions returns a value equivalent to VFP's datetime data type.

```
sqlite> update people set lastUpdated = datetime() where pk = 4;
sqlite> select * from people where pk = 4;
4|Davis|David|1994-11-22|0|2019-05-10 22:16:44
```

The value in the datetime column is formatted as YYYY-MM-DD hh:mm:ss, which is equivalent to VFP's datetime format. One wrinkle is that the value returned by SQLite's *datetime( )* function is UTC time. I ran this code at 5:16:44 PM Central Daylight Time, which on that date was UTC minus 5 hours.

If we re-run the ODBC query from VFP, the resulting cursor now has a column for the *lastUpdated* value. Note that VFP has appended "PM" to the datetime value in the cursor.



Pk	Lastname	Firstname	Birthdate	Drinkscoffee	Lastupdated
1	Adams	Amy	.NULL.	1	.NULL.
2	Baker	Bob	.NULL.	1	.NULL.
3	Cooper	Carol	.NULL.	0	.NULL.
4	Davis	David	11/22/1994	0	05/10/2019 10:16:44 PM

Figure 21: The cursor now has a datetime column for the *lastUpdated* value.

Looking at the structure of the cursor, we can see that the *lastUpdated* column is a VFP *datetime* data type and that the value of the data in the SQLite database was correctly converted into the VFP format.

Listing 19: The SQLite datetime value was converted to a VFP datetime value.

```
Structure for table:          C:\USERS\RICK\APPDATA\LOCAL\TEMP\0002WLXL04M3.TMP
```

[12] See https://www.sqlite.org/lang_datefunc.html for complete information.

```
Number of data records:       4
Date of last update:            /  /
Code Page:                    1252
  Field   Field Name    Type      Width  Dec  Index  Collate  Nulls  Next  Step
      1   PK            Integer       4                        Yes
      2   LASTNAME      Character    20                        Yes
      3   FIRSTNAME     Character    20                        Yes
      4   BIRTHDATE     Date          8                        Yes
      5   DRINKSCOFFEE  Integer       4                        Yes
      6   LASTUPDATED   DateTime      8                        Yes
** Total **                          66
```

Inserting data from VFP into SQLite

As you would expect, inserting data from a VFP database into a SQLite database is simply a matter of creating a SQLite database with the desired structure and then writing VFP code to insert data using the SQLite ODBC driver.

Continuing with the example of the *people* table, let's assume we do not yet have a SQLite database. The first step is to use the SQLite command-line interface to create one. The *.open* dot-command creates a new file with the specified name, assuming a file with that name does not already exist in the current folder.

```
sqlite> .open fromVFP.db -- create a new SQLite datbase file named fromVFP.db
sqlite> create table people (
   ...> pk integer,
   ...> lastName varchar(20),
   ...> firstName varchar(20),
   ...> birthDate date,
   ...> drinksCoffee integer
   ...> );
```

This is a simple table structure and therefore not difficult to type in from the SQLite command line. For a table with many more columns, where the probability of typos when entering it manually is greater, a better approach might be to create a SQL script to create the *people* table and save it to a file that SQLite can read.

Listing 20: Save the code to create the *people* table to a file, for example *createPeople.sql*.

```
create table people (
   pk integer,
   lastName varchar(20),
   firstName varchar(20),
   birthDate date,
   drinksCoffee integer
)
```

The use the SQLite *.read* dot-command to execute the code in the SQL file. The *.schema* dot-command is there so we can confirm the structure of the table came out as intended.

Listing 21: Use the SQLite *.read* dot-command to execute the code in the createPeople.sql file.

```
sqlite> .open fromVFP.db -- create a new SQLite datbase file named fromVFP.db
sqlite> .read createPeople.sql
sqlite> .schema
CREATE TABLE people (
   pk integer,
   lastName varchar(20),
   firstName varchar(20),
   birthDate date,
   drinksCoffee integer,
    lastupdated datetime
);
```

In VFP, we can now write code to insert data into the *people* table in the new *fromVFP* SQLite database. Listing 22 illustrates how to do this using VFP's TEXTMERGE function to create the INSERT SQL statements along with the appropriate trimming and quoting of string values and the handling of NULLs. The VFP date field is formatted as *YYYY-MM-DD* and the datetime field is formatted as *YYYY-MM-DD hh:mm:ss*.

Listing 22: This code inserts data from a VFP cursor into a SQLite table.

```
lcConnString = [DRIVER={SQLite3 ODBC
Driver};DATABASE=C:\SWFox2019\Sessions\SQLite\fromVFP.db;]
lnConnHandle = SQLSTRINGCONNECT( lcConnString)
IF lnConnHandle = -1
   WAIT WINDOW "Failed to connect"
   RETURN
ELSE
   WAIT WINDOW "Connected with connection handle " + TRANSFORM( lnConnHandle)
ENDIF
SELECT csrPeople
SET TEXTMERGE ON
SCAN
   lcLastName = ['] + ALLTRIM( csrPeople.lastName) + [']
   lcFirstName = ['] + ALLTRIM( csrPeople.firstName) + [']
   lcBirthDate = ;
      IIF( ISNULL( csrPeople.birthDate), "NULL", ;
           ['] + ALLTRIM( STR( YEAR( csrPeople.birthDate))) + "-" + ;
           PADL( ALLTRIM( STR( MONTH( csrPeople.birthDate))), 2, "0") + "-" + ;
           PADL( ALLTRIM( STR( DAY( csrPeople.birthDate))), 2, "0") + ['])
   lcLastUpdated = ;
       IIF( ISNULL( csrPeople.lastUpdated), "NULL", ;
          ['] + ALLTRIM( STR( YEAR( csrPeople.lastUpdated))) + "-" + ;
          PADL( ALLTRIM( STR( MONTH( csrPeople.lastUpdated))), 2, "0") + "-" + ;
          PADL( ALLTRIM( STR( DAY( csrPeople.lastUpdated))), 2, "0")+ SPACE(1) + ;
          PADL( ALLTRIM( STR( HOUR( csrPeople.lastUpdated))), 2, "0") + ":" + ;
          PADL( ALLTRIM( STR( MINUTE( csrPeople.lastUpdated))), 2, "0") + ":" + ;
          PADL( ALLTRIM( STR( SEC( csrPeople.lastUpdated))), 2, "0") + ['])
   TEXT TO lcSQL NOSHOW PRETEXT 15
      INSERT INTO people VALUES (
         <<csrPeople.pk>>,
         <<lcLastName>>,
         <<lcFirstName>>,
```

```
        <<lcBirthdate>>,
        <<csrPeople.drinksCoffee>>,
        <<lcLastUpdated>>
        )
   ENDTEXT
   SQLEXEC( lnConnHandle, lcSQL)
ENDSCAN
SET TEXTMERGE OFF
SQLDISCONNECT( lnConnHandle)
```

Use cases for SQLite with VFP

Data that already exists in SQLite

There are many situations in which it can be useful to access SQLite databases from VFP. For example, there are publicly accessible SQLite databases you can find and download for your own use, assuming they're appropriately licensed. One such source is a collection of databases from Public Affairs Data Journalism at Stanford University, available at http://2016.padjo.org/tutorials/sqlite-data-starterpacks/.

One of the databases in this collection is a set of baby names and their popularity by state and gender for the year 2015. This database can be downloaded from http://2016.padjo.org/files/data/starterpack/ssa-babynames/babynames-gendered-2015.sqlite. Note that the publisher has chosen to use the .sqlite file extension for this database, which is okay because SQLite does not care about file name extensions.

SQLite Studio is a good way to inspect an unfamiliar SQLite database. Figure 22 shows the baby names database open in SQLite Studio, revealing that it has one table called *gendered_names*. Expanding the Columns and the Indexes nodes in the treeview displays the columns names and indexes.

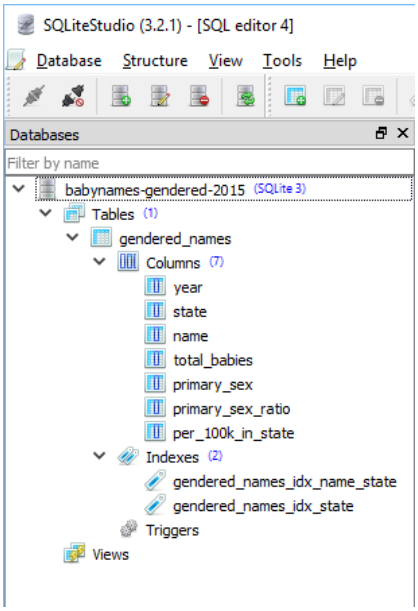


Figure 22: SQLite Studio is one way to explore the structure of an unfamiliar SQLite database.

As an exercise, you can run the following query to see how many babies in your state were given your name in 2015.

```
sqlite> select name, total_babies from gendered_names where name = '<your name>' and
state = '<your state>';
```

You can pull the entire list for your state, in order by name and popularity, as follows. It will most likely be a very long list, so you may want to send the output to a file as demonstrated in Listing 3 and Listing 4.

```
sqlite> select name, total_babies from gendered_names where state = '<your state>'
order by 2 desc, 1;
```

Knowing how to access data like this from VFP enables you to incorporate into a VFP app.

Tables larger the 2GB

As noted earlier, SQLite tables are not subject to the 2GB limit like VFP tables are. If you have a VFP app using VFP data with a table or tables approaching 2GB, or if you want to use VFP to work with a database that is already larger than 2GB, SQLite may be a viable solution. Whil Hentzen wrote a small book on this subject a few years ago in which he describes his experience with such a project.[13] His book includes a lot of good detail about accessing SQLite data from VFP, including a chapter on error handling.

As an alternative to SQL Server

Microsoft SQL Server is probably the most common alternative to native VFP databases for VFP apps. While SQL Server Express is a good choice for clients whose applications do not need the power of the Standard or Enterprise editions, it still requires installation on the client's computer or network file server along with some ongoing maintenance tasks. Because it requires no installation when distributed with a VFP app, SQLite can be a good alternative in situations where clients lack the desire and/or the technical resources to support even SQL Server Express.

As a common denominator for sharing databases

A lot of my work involves exchanging data on behalf of my clients with 3rd parties who use various types of databases. The lowest common denominator for exchanging data is usually a flat file format such as CSV, tab-delimited, pipe-delimited, XML, or a specific EDI format. Because it is easily consumed from virtually any source, SQLite can be a way to exchange actual relational data between disparate types of systems instead of falling back to simple flat files.

[13] *Using SQLite to Bypass the 2 GB .DBF Filesize Limit* by Whil Hentzen, edited by Ted Roche, Copyright 2012, 2013, 2015 Whil Hentzen, Hentzenwerke Publishing, Inc., ISBN 978-1-930919-76-1

Summary

SQLite is a small, free, cross-platform, SQL-compliant database engine widely used by many applications both large and small. SQLite databases are contained in a single file, making them ideal for applications where simplicity and portability is the goal. The SQLite ODBC driver enables Visual FoxPro developers to incorporate SQLite databases into their VFP applications using familiar SQL statements to insert, update, and delete data.

Resources

Online

SQLite Home Page
https://www.sqlite.org

SQLite Download Page
https://www.sqlite.org/download.html

How to Download & Install SQLite
http://www.sqlitetutorial.net/download-install-sqlite/

SQLite Release History
https://www.sqlite.org/changes.html

SQLite Studio
https://sqlitestudio.pl

SQLite Studio User Manual
https://github.com/pawelsalawa/sqlitestudio/wiki/User_Manual

SQLite ODBC driver downloads and information
http://www.ch-werner.de/sqliteodbc/

SQLite ODBC documentation
http://www.ch-werner.de/sqliteodbc/html/index.html

SQLite Tutorial
http://www.sqlitetutorial.net/

SQLite Tutorial
https://www.w3resource.com/sqlite/

Biography

Rick Borup is owner and president of Information Technology Associates, LLC, a professional software development, computer services, and information systems consulting firm he founded in 1993. Rick earned BS and MBA degrees from the University of Illinois and spent several years developing software applications for mainframe computers before turning to PC

Copyright 2019, Rick Borup

database development tools in the late 1980s. He began working with FoxPro in 1991, and has worked full time in FoxPro and Visual FoxPro since 1993. He is a co-author of the books Deploying Visual FoxPro Solutions and Visual FoxPro Best Practices for the Next Ten Years. He has published articles in FoxTalk, FoxPro Advisor, and FoxRockX and is a frequent speaker at Visual FoxPro conferences and user groups. Rick is a Microsoft Certified Solution Developer (MCSD) and a Microsoft Certified Professional (MCP) in Visual FoxPro.

Copyright © 2019 Rick Borup. Windows® is a registered trademark of Microsoft Corporation in the United States and other countries. All other trademarks are the property of their respective owners

Appendix A

This is the list of SQLite dot-commands generated by the SQLite ".help" dot-command. Listing X shows how to generate this list and direct the output to a file named help.txt. The last line restores output to standard output, i.e., the command window.

```
sqlite> .mode list
sqlite> .output help.txt
sqlite> .help
sqlite> .output stdout
```

Note: the output file is created in Unix/Mac format, meaning lines are terminated only with a line-feed character rather than the PC format of carriage-return plus line-feed.

```
.archive ...             Manage SQL archives
.auth ON|OFF             Show authorizer callbacks
.backup ?DB? FILE        Backup DB (default "main") to FILE
.bail on|off             Stop after hitting an error.  Default OFF
.binary on|off           Turn binary output on or off.  Default OFF
.cd DIRECTORY            Change the working directory to DIRECTORY
.changes on|off          Show number of rows changed by SQL
.check GLOB              Fail if output since .testcase does not match
.clone NEWDB             Clone data into NEWDB from the existing database
.databases               List names and files of attached databases
.dbconfig ?op? ?val?     List or change sqlite3_db_config() options
.dbinfo ?DB?             Show status information about the database
.dump ?TABLE? ...        Render all database content as SQL
.echo on|off             Turn command echo on or off
.eqp on|off|full|...     Enable or disable automatic EXPLAIN QUERY PLAN
.excel                   Display the output of next command in a spreadsheet
.exit ?CODE?             Exit this program with return-code CODE
.expert                  EXPERIMENTAL. Suggest indexes for specified queries
.fullschema ?--indent?   Show schema and the content of sqlite_stat tables
.headers on|off          Turn display of headers on or off
.help ?-all? ?PATTERN?   Show help text for PATTERN
.import FILE TABLE       Import data from FILE into TABLE
.imposter INDEX TABLE    Create imposter table TABLE on index INDEX
.indexes ?TABLE?         Show names of indexes
.limit ?LIMIT? ?VAL?     Display or change the value of an SQLITE_LIMIT
.lint OPTIONS            Report potential schema issues.
.load FILE ?ENTRY?       Load an extension library
.log FILE|off            Turn logging on or off.  FILE can be stderr/stdout
.mode MODE ?TABLE?       Set output mode
.nullvalue STRING        Use STRING in place of NULL values
.once (-e|-x|FILE)       Output for the next SQL command only to FILE
.open ?OPTIONS? ?FILE?   Close existing database and reopen FILE
.output ?FILE?           Send output to FILE or stdout if FILE is omitted
.print STRING...         Print literal STRING
.progress N              Invoke progress handler after every N opcodes
.prompt MAIN CONTINUE    Replace the standard prompts
.quit                    Exit this program
.read FILE               Read input from FILE
.restore ?DB? FILE       Restore content of DB (default "main") from FILE
.save FILE               Write in-memory database into FILE
```

```
.scanstats on|off        Turn sqlite3_stmt_scanstatus() metrics on or off
.schema ?PATTERN?        Show the CREATE statements matching PATTERN
.selftest ?OPTIONS?      Run tests defined in the SELFTEST table
.separator COL ?ROW?     Change the column and row separators
.sha3sum ...             Compute a SHA3 hash of database content
.shell CMD ARGS...       Run CMD ARGS... in a system shell
.show                    Show the current values for various settings
.stats ?on|off?          Show stats or turn stats on or off
.system CMD ARGS...      Run CMD ARGS... in a system shell
.tables ?TABLE?          List names of tables matching LIKE pattern TABLE
.testcase NAME           Begin redirecting output to 'testcase-out.txt'
.timeout MS              Try opening locked tables for MS milliseconds
.timer on|off            Turn SQL timer on or off
.trace ?OPTIONS?         Output each SQL statement as it is run
.vfsinfo ?AUX?           Information about the top-level VFS
.vfslist                 List all available VFSes
.vfsname ?AUX?           Print the name of the VFS stack
.width NUM1 NUM2 ...     Set column widths for "column" mode
```

Appendix B

This is the full output of the SQLite3 analyzer utility run on the *test.db* database.

```
/** Disk-Space Utilization Report For test.db

Page size in bytes.............................. 4096
Pages in the whole file (measured)............... 3
Pages in the whole file (calculated)............. 3
Pages that store data............................ 3          100.0%
Pages on the freelist (per header)............... 0            0.0%
Pages on the freelist (calculated)............... 0            0.0%
Pages of auto-vacuum overhead.................... 0            0.0%
Number of tables in the database................. 3
Number of indices................................ 0
Number of defined indices........................ 0
Number of implied indices........................ 0
Size of the file in bytes........................ 12288
Bytes of user payload stored..................... 82           0.67%

*** Page counts for all tables with their indices ****************************

FOO.............................................. 1            33.3%
PEOPLE........................................... 1            33.3%
SQLITE_MASTER.................................... 1            33.3%

*** Page counts for all tables and indices separately ************************

FOO.............................................. 1            33.3%
PEOPLE........................................... 1            33.3%
SQLITE_MASTER.................................... 1            33.3%

*** All tables ***************************************************************

Percentage of total database..................... 100.0%
Number of entries................................ 8
Bytes of storage consumed........................ 12288
Bytes of payload................................. 281          2.3%
Bytes of metadata................................ 156          1.3%
Average payload per entry........................ 35.13
Average unused bytes per entry................... 1481.38
Average metadata per entry....................... 19.50
Maximum payload per entry........................ 104
Entries that use overflow........................ 0            0.0%
Primary pages used............................... 3
Overflow pages used.............................. 0
Total pages used................................. 3
Unused bytes on primary pages.................... 11851        96.4%
Unused bytes on overflow pages................... 0
Unused bytes on all pages........................ 11851        96.4%

*** Table FOO ****************************************************************

Percentage of total database.....................  33.3%
Number of entries................................ 3
```

```
Bytes of storage consumed..... 4096
Bytes of payload..... 41           1.0%
Bytes of metadata..... 20          0.49%
B-tree depth..... 1
Average payload per entry..... 13.67
Average unused bytes per entry..... 1345.00
Average metadata per entry..... 6.67
Maximum payload per entry..... 16
Entries that use overflow..... 0           0.0%
Primary pages used..... 1
Overflow pages used..... 0
Total pages used..... 1
Unused bytes on primary pages..... 4035        98.5%
Unused bytes on overflow pages..... 0
Unused bytes on all pages..... 4035        98.5%

*** Table PEOPLE ************************************************************

Percentage of total database.....  33.3%
Number of entries..... 3
Bytes of storage consumed..... 4096
Bytes of payload..... 41           1.0%
Bytes of metadata..... 20          0.49%
B-tree depth..... 1
Average payload per entry..... 13.67
Average unused bytes per entry..... 1345.00
Average metadata per entry..... 6.67
Maximum payload per entry..... 16
Entries that use overflow..... 0           0.0%
Primary pages used..... 1
Overflow pages used..... 0
Total pages used..... 1
Unused bytes on primary pages..... 4035        98.5%
Unused bytes on overflow pages..... 0
Unused bytes on all pages..... 4035        98.5%

*** Table SQLITE_MASTER *****************************************************

Percentage of total database.....  33.3%
Number of entries..... 2
Bytes of storage consumed..... 4096
Bytes of payload..... 199          4.9%
Bytes of metadata..... 116          2.8%
B-tree depth..... 1
Average payload per entry..... 99.50
Average unused bytes per entry..... 1890.50
Average metadata per entry..... 58.00
Maximum payload per entry..... 104
Entries that use overflow..... 0           0.0%
Primary pages used..... 1
Overflow pages used..... 0
Total pages used..... 1
Unused bytes on primary pages..... 3781        92.3%
Unused bytes on overflow pages..... 0
Unused bytes on all pages..... 3781        92.3%
```

```
*** Definitions ************************************************************

Page size in bytes

    The number of bytes in a single page of the database file.
    Usually 1024.

Number of pages in the whole file

    The number of 4096-byte pages that go into forming the complete
    database

Pages that store data

    The number of pages that store data, either as primary B*Tree pages or
    as overflow pages.  The number at the right is the data pages divided by
    the total number of pages in the file.

Pages on the freelist

    The number of pages that are not currently in use but are reserved for
    future use.  The percentage at the right is the number of freelist pages
    divided by the total number of pages in the file.

Pages of auto-vacuum overhead

    The number of pages that store data used by the database to facilitate
    auto-vacuum. This is zero for databases that do not support auto-vacuum.

Number of tables in the database

    The number of tables in the database, including the SQLITE_MASTER table
    used to store schema information.

Number of indices

    The total number of indices in the database.

Number of defined indices

    The number of indices created using an explicit CREATE INDEX statement.

Number of implied indices

    The number of indices used to implement PRIMARY KEY or UNIQUE constraints
    on tables.

Size of the file in bytes

    The total amount of disk space used by the entire database files.

Bytes of user payload stored

    The total number of bytes of user payload stored in the database. The
```

```
    schema information in the SQLITE_MASTER table is not counted when
    computing this number.  The percentage at the right shows the payload
    divided by the total file size.

Percentage of total database

    The amount of the complete database file that is devoted to storing
    information described by this category.

Number of entries

    The total number of B-Tree key/value pairs stored under this category.

Bytes of storage consumed

    The total amount of disk space required to store all B-Tree entries
    under this category.  The is the total number of pages used times
    the pages size.

Bytes of payload

    The amount of payload stored under this category.  Payload is the data
    part of table entries and the key part of index entries.  The percentage
    at the right is the bytes of payload divided by the bytes of storage
    consumed.

Bytes of metadata

    The amount of formatting and structural information stored in the
    table or index.  Metadata includes the btree page header, the cell pointer
    array, the size field for each cell, the left child pointer or non-leaf
    cells, the overflow pointers for overflow cells, and the rowid value for
    rowid table cells.  In other words, metadata is everything that is neither
    unused space nor content.  The record header in the payload is counted as
    content, not metadata.

Average payload per entry

    The average amount of payload on each entry.  This is just the bytes of
    payload divided by the number of entries.

Average unused bytes per entry

    The average amount of free space remaining on all pages under this
    category on a per-entry basis.  This is the number of unused bytes on
    all pages divided by the number of entries.

Non-sequential pages

    The number of pages in the table or index that are out of sequence.
    Many filesystems are optimized for sequential file access so a small
    number of non-sequential pages might result in faster queries,
    especially for larger database files that do not fit in the disk cache.
    Note that after running VACUUM, the root page of each table or index is
    at the beginning of the database file and all other pages are in a
```

```
    separate part of the database file, resulting in a single non-
    sequential page.

Maximum payload per entry

    The largest payload size of any entry.

Entries that use overflow

    The number of entries that user one or more overflow pages.

Total pages used

    This is the number of pages used to hold all information in the current
    category.  This is the sum of index, primary, and overflow pages.

Index pages used

    This is the number of pages in a table B-tree that hold only key (rowid)
    information and no data.

Primary pages used

    This is the number of B-tree pages that hold both key and data.

Overflow pages used

    The total number of overflow pages used for this category.

Unused bytes on index pages

    The total number of bytes of unused space on all index pages.  The
    percentage at the right is the number of unused bytes divided by the
    total number of bytes on index pages.

Unused bytes on primary pages

    The total number of bytes of unused space on all primary pages.  The
    percentage at the right is the number of unused bytes divided by the
    total number of bytes on primary pages.

Unused bytes on overflow pages

    The total number of bytes of unused space on all overflow pages.  The
    percentage at the right is the number of unused bytes divided by the
    total number of bytes on overflow pages.

Unused bytes on all pages

    The total number of bytes of unused space on all primary and overflow
    pages.  The percentage at the right is the number of unused bytes
    divided by the total number of bytes.

*******************************************************************************
The entire text of this report can be sourced into any SQL database
```

```
engine for further analysis.  All of the text above is an SQL comment.
The data used to generate this report follows:
*/
BEGIN;
CREATE TABLE space_used(
   name clob,        -- Name of a table or index in the database file
   tblname clob,     -- Name of associated table
   is_index boolean, -- TRUE if it is an index, false for a table
   is_without_rowid boolean, -- TRUE if WITHOUT ROWID table
   nentry int,       -- Number of entries in the BTree
   leaf_entries int, -- Number of leaf entries
   depth int,        -- Depth of the b-tree
   payload int,      -- Total amount of data stored in this table or index
   ovfl_payload int, -- Total amount of data stored on overflow pages
   ovfl_cnt int,     -- Number of entries that use overflow
   mx_payload int,   -- Maximum payload size
   int_pages int,    -- Number of interior pages used
   leaf_pages int,   -- Number of leaf pages used
   ovfl_pages int,   -- Number of overflow pages used
   int_unused int,   -- Number of unused bytes on interior pages
   leaf_unused int,  -- Number of unused bytes on primary pages
   ovfl_unused int,  -- Number of unused bytes on overflow pages
   gap_cnt int,      -- Number of gaps in the page layout
   compressed_size int  -- Total bytes stored on disk
);
INSERT INTO space_used
VALUES('sqlite_master','sqlite_master',0,0,2,2,1,199,0,0,104,0,1,0,0,3781,0,0,4096);
INSERT INTO space_used
VALUES('people','people',0,0,3,3,1,41,0,0,16,0,1,0,0,4035,0,0,4096);
INSERT INTO space_used VALUES('foo','foo',0,0,3,3,1,41,0,0,16,0,1,0,0,4035,0,0,4096);
COMMIT;
```