Small. Fast. Reliable.
Choose any three.

Command Line Shell For SQLite

The SQLite project provides a simple command-line utility named **sqlite3** (or **sqlite3.exe** on windows) that allows the user to manually enter and execute SQL statements against an SQLite database. This document provides a brief introduction on how to use the **sqlite3** program.

Getting Started

To start the **sqlite3** program, just type "sqlite3" optionally followed by the name the file that holds the SQLite database. If the file does not exist, a new database file with the given name will be created automatically. If no database file is specified, a temporary database is created, then deleted when the "sqlite3" program exits.

When started, the **sqlite3** program will show a brief banner message then prompt you to enter SQL. Type in SQL statements (terminated by a semicolon), press "Enter" and the SQL will be executed.

For example, to create a new SQLite database named "ex1" with a single table named "tbl1", you might do this:

```
$ sqlite3 ex1
SQLite version 3.8.5 2014-05-29 12:36:14
Enter ".help" for usage hints.
sqlite> create table tbl1(one varchar(10), two smallint);
sqlite> insert into tbl1 values('hello!',10);
sqlite> insert into tbl1 values('goodbye', 20);
sqlite> select * from tbl1;
hello!|10
goodbye|20
sqlite>
```

You can terminate the sqlite3 program by typing your systems End-Of-File character (usually a Control-D). Use the interrupt character (usually a Control-C) to stop a long-running SQL statement.

Make sure you type a semicolon at the end of each SQL command! The sqlite3 program looks for a semicolon to know when your SQL command is complete. If you omit the semicolon, sqlite3 will give you a continuation prompt and wait for you to

enter more text to be added to the current SQL command. This feature allows you to enter SQL commands that span multiple lines. For example:

```
sqlite> CREATE TABLE tbl2 (
   ...>   f1 varchar(30) primary key,
   ...>   f2 text,
   ...>   f3 real
   ...> );
sqlite>
```

Double-click Startup On Windows

Windows users can double-click on the **sqlite3.exe** icon to cause the command-line shell to pop-up a terminal window running SQLite. Note, however, that by default this SQLite session is using an in-memory database, not a file on disk, and so all changes will be lost when the session exits. To use a persistent disk file as the database, enter the ".open" command immediately after the terminal window starts up:

```
SQLite version 3.8.5 2014-05-29 12:36:14
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open ex1.db
sqlite>
```

The example above causes the database file named "ex1.db" to be opened and used, and created if it does not previously exist. You might want to use a full pathname to ensure that the file is in the directory that you think it is in. Use forward-slashes as the directory separator character. In other words use "c:/work/ex1.db", not "c:\work\ex1.db".

Alternatively, you can create a new database using the default in-memory storage, then save that database into a disk file using the ".save" command:

```
SQLite version 3.8.5 2014-05-29 12:36:14
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> ... many SQL commands omitted ...
sqlite> .save ex1.db
sqlite>
```

Be careful when using the ".save" command as it will overwrite any preexisting database files having the same name without prompting for confirmation. As with the ".open" command, you might want to use a full pathname with forward-slash directory separators to avoid ambiguity.

Special commands to sqlite3

Most of the time, sqlite3 just reads lines of input and passes them on to the SQLite library for execution. But if an input line begins with a dot ("."), then that line is intercepted and interpreted by the sqlite3 program itself. These "dot commands" are

typically used to change the output format of queries, or to execute certain prepackaged query statements.

For a listing of the available dot commands, you can enter ".help" at any time. For example:

```
sqlite> .help
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail on|off           Stop after hitting an error.  Default OFF
.clone NEWDB           Clone data into NEWDB from the existing database
.databases             List names and files of attached databases
.dump ?TABLE? ...      Dump the database in an SQL text format
                         If TABLE specified, only dump tables matching
                         LIKE pattern TABLE.
.echo on|off           Turn command echo on or off
.eqp on|off            Enable or disable automatic EXPLAIN QUERY PLAN
.exit                  Exit this program
.explain ?on|off?      Turn output mode suitable for EXPLAIN on or off.
                         With no args, it turns EXPLAIN on.
.fullschema            Show schema and the content of sqlite_stat tables
.headers on|off        Turn display of headers on or off
.help                  Show this message
.import FILE TABLE     Import data from FILE into TABLE
.indices ?TABLE?       Show names of all indices
                         If TABLE specified, only show indices for tables
                         matching LIKE pattern TABLE.
.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 where MODE is one of:
                         csv      Comma-separated values
                         column   Left-aligned columns.  (See .width)
                         html     HTML <table> code
                         insert   SQL insert statements for TABLE
                         line     One value per line
                         list     Values delimited by .separator string
                         tabs     Tab-separated values
                         tcl      TCL list elements
.nullvalue STRING      Use STRING in place of NULL values
.once FILENAME         Output for the next SQL command only to FILENAME
.open ?FILENAME?       Close existing database and reopen FILENAME
.output ?FILENAME?     Send output to FILENAME or stdout
.print STRING...       Print literal STRING
.prompt MAIN CONTINUE  Replace the standard prompts
.quit                  Exit this program
.read FILENAME         Execute SQL in FILENAME
.restore ?DB? FILE     Restore content of DB (default "main") from FILE
.save FILE             Write in-memory database into FILE
.schema ?TABLE?        Show the CREATE statements
                         If TABLE specified, only show tables matching
                         LIKE pattern TABLE.
.separator STRING ?NL? Change separator used by output mode and .import
                         NL is the end-of-line mark for CSV
.shell CMD ARGS...     Run CMD ARGS... in a system shell
.show                  Show the current values for various settings
.stats on|off          Turn stats on or off
.system CMD ARGS...    Run CMD ARGS... in a system shell
.tables ?TABLE?        List names of tables
                         If TABLE specified, only list tables matching
                         LIKE pattern TABLE.
.timeout MS            Try opening locked tables for MS milliseconds
.timer on|off          Turn SQL timer on or off
```

```
.trace FILE|off        Output each SQL statement as it is run
.vfsname ?AUX?         Print the name of the VFS stack
.width NUM1 NUM2 ...   Set column widths for "column" mode
                         Negative values right-justify
sqlite>
```

Rules for "dot-commands"

Ordinary SQL statements are free-form, and can be spread across multiple lines, and can have whitespace and comments anywhere. But dot-commands are more restrictive:

- A dot-command must begin with the "." at the left margin with no preceding whitespace.
- The dot-command must be entirely contained on a single input line.
- A dot-command cannot occur in the middle of an ordinary SQL statement. In other words, a dot-command cannot occur at a continuation prompt.
- Dot-commands do not recognize comments.

And, of course, it is important to remember that the dot-commands are interpreted by the sqlite3.exe command-line program, not by SQLite itself. So none of the dot-commands will work as an argument to SQLite interfaces like sqlite3_prepare() or sqlite3_exec().

Changing Output Formats

The sqlite3 program is able to show the results of a query in eight different formats: "csv", "column", "html", "insert", "line", "list", "tabs", and "tcl". You can use the ".mode" dot command to switch between these output formats.

The default output mode is "list". In list mode, each record of a query result is written on one line of output and each column within that record is separated by a specific separator string. The default separator is a pipe symbol ("|"). List mode is especially useful when you are going to send the output of a query to another program (such as AWK) for additional processing.

```
sqlite> .mode list
sqlite> select * from tbl1;
hello|10
goodbye|20
sqlite>
```

You can use the ".separator" dot command to change the separator for list mode. For example, to change the separator to a comma and a space, you could do this:

```
sqlite> .separator ", "
sqlite> select * from tbl1;
hello, 10
goodbye, 20
sqlite>
```

In "line" mode, each column in a row of the database is shown on a line by itself. Each

line consists of the column name, an equal sign and the column data. Successive records are separated by a blank line. Here is an example of line mode output:

```
sqlite> .mode line
sqlite> select * from tbl1;
one = hello
two = 10

one = goodbye
two = 20
sqlite>
```

In column mode, each record is shown on a separate line with the data aligned in columns. For example:

```
sqlite> .mode column
sqlite> select * from tbl1;
one         two
----------  ----------
hello       10
goodbye     20
sqlite>
```

By default, each column is between 1 and 10 characters wide, depending on the column header name and the width of the first column of data. Data that is too wide to fit in a column is truncated. You can adjust the column widths using the ".width" command. Like this:

```
sqlite> .width 12 6
sqlite> select * from tbl1;
one           two
------------  ------
hello         10
goodbye       20
sqlite>
```

The ".width" command in the example above sets the width of the first column to 12 and the width of the second column to 6. All other column widths were unaltered. You can gives as many arguments to ".width" as necessary to specify the widths of as many columns as are in your query results.

If you specify a column a width of 0, then the column width is automatically adjusted to be the maximum of three numbers: 10, the width of the header, and the width of the first row of data. This makes the column width self-adjusting. The default width setting for every column is this auto-adjusting 0 value.

You can specify a negative column width to get right-justified columns.

The column labels that appear on the first two lines of output can be turned on and off using the ".header" dot command. In the examples above, the column labels are on. To turn them off you could do this:

```
sqlite> .header off
sqlite> select * from tbl1;
hello         10
```

```
goodbye       20
sqlite>
```

Another useful output mode is "insert". In insert mode, the output is formatted to look like SQL INSERT statements. You can use insert mode to generate text that can later be used to input data into a different database.

When specifying insert mode, you have to give an extra argument which is the name of the table to be inserted into. For example:

```
sqlite> .mode insert new_table
sqlite> select * from tbl1;
INSERT INTO "new_table" VALUES('hello',10);
INSERT INTO "new_table" VALUES('goodbye',20);
sqlite>
```

The last output mode is "html". In this mode, sqlite3 writes the results of the query as an XHTML table. The beginning <TABLE> and the ending </TABLE> are not written, but all of the intervening <TR>s, <TH>s, and <TD>s are. The html output mode is envisioned as being useful for CGI.

The ".explain" dot command can be used to set the output mode to "column" and to set the column widths to values that are reasonable for looking at the output of an EXPLAIN command. The EXPLAIN command is an SQLite-specific SQL extension that is useful for debugging. If any regular SQL is prefaced by EXPLAIN, then the SQL command is parsed and analyzed but is not executed. Instead, the sequence of virtual machine instructions that would have been used to execute the SQL command are returned like a query result. For example:

```
sqlite> .explain
sqlite> explain delete from tbl1 where two<20;
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Trace          0     0     0                    00
1     Goto           0     18    0                    00
2     Null           0     1     0                    00  r[1]=NULL
3     OpenRead       0     2     0     2              00  root=2 iDb=0; tbl1
4     Rewind         0     10    0                    00
5       Column         0     1     2                    00  r[2]=tbl1.two
6       Ge             3     9     2     (BINARY)       6a  if r[3]>=r[2] goto 10
7       Rowid          0     4     0                    00  r[4]=rowid
8       RowSetAdd      1     4     0                    00  rowset(1)=r[4]
9     Next           0     7     0                    01
10    Close          0     0     0                    00
11    OpenWrite      0     2     0     2              00  root=2 iDb=0; tbl1
12      RowSetRead     1     16    4                    00  r[4]=rowset(1)
13      NotExists      0     12    4     1              00  intkey=r[4]
14      Delete         0     1     0     tbl1           00
15    Goto           0     12    0                    00
16    Close          0     0     0                    00
17    Halt           0     0     0                    00
18    Transaction    0     1     0                    00
19    VerifyCookie   0     1     0                    00
20    TableLock      0     2     1     tbl1           00  iDb=0 root=2 write=1
21    Integer        20    3     0                    00  r[3]=20
22    Goto           0     2     0                    00
```

Notice how the shell changes the indentation of some opcodes to help show the loop structure of the VDBE program.

Writing results to a file

By default, sqlite3 sends query results to standard output. You can change this using the ".output" and ".once" commands. Just put the name of an output file as an argument to .output and all subsequent query results will be written to that file. Or use the .once command instead of .output and output will only be redirected for the single next command before returning the console. Use .output with no arguments to begin writing to standard output again. For example:

```
sqlite> .mode list
sqlite> .separator |
sqlite> .output test_file_1.txt
sqlite> select * from tbl1;
sqlite> .exit
$ cat test_file_1.txt
hello|10
goodbye|20
$
```

If the first character of the ".output" or ".once" filename is a pipe symbol ("|") then the remaining characters are treated as a command and the output is sent to that command. This makes it easy to pipe the results of a query into some other process. For example, the "open -f" command on a Mac opens a text editor to display the content that it reads from standard input. So to see the results of a query in a text editor, one could type:

```
sqlite3> .once '|open -f'
sqlite3> SELECT * FROM bigTable;
```

File I/O Functions

The command-line shell adds two application-defined SQL functions that facilitate read content from a file into an table column, and writing the content of a column into a file, respectively.

The readfile(X) SQL function reads the entire content of the file named X and returns that content as a BLOB. This can be used to load content into a table. For example:

```
sqlite> CREATE TABLE images(name TEXT, type TEXT, img BLOB);
sqlite> INSERT INTO images(name,type,img)
   ...>   VALUES('icon','jpeg',readfile('icon.jpg'));
```

The writefile(X,Y) SQL function write the blob Y into the file named X and returns the number of bytes written. Use this function to extract the content of a single table column into a file. For example:

```
sqlite> SELECT writefile('icon.jpg',img) FROM images WHERE name='icon';
```

Note that the readfile(X) and writefile(X,Y) functions are extension functions and are

not built into the core SQLite library. These routines are available as a loadable extension in the ext/misc/fileio.c source file in the SQLite source code repositories.

Querying the database schema

The sqlite3 program provides several convenience commands that are useful for looking at the schema of the database. There is nothing that these commands do that cannot be done by some other means. These commands are provided purely as a shortcut.

For example, to see a list of the tables in the database, you can enter ".tables".

```
sqlite> .tables
tbl1
tbl2
sqlite>
```

The ".tables" command is similar to setting list mode then executing the following query:

```
SELECT name FROM sqlite_master
WHERE type IN ('table','view') AND name NOT LIKE 'sqlite_%'
UNION ALL
SELECT name FROM sqlite_temp_master
WHERE type IN ('table','view')
ORDER BY 1
```

In fact, if you look at the source code to the sqlite3 program (found in the source tree in the file src/shell.c) you'll find a query very much like the above.

The ".indices" command works in a similar way to list all of the indices for a particular table. The ".indices" command takes a single argument which is the name of the table for which the indices are desired. Last, but not least, is the ".schema" command. With no arguments, the ".schema" command shows the original CREATE TABLE and CREATE INDEX statements that were used to build the current database. If you give the name of a table to ".schema", it shows the original CREATE statement used to make that table and all if its indices. We have:

```
sqlite> .schema
create table tbl1(one varchar(10), two smallint)
CREATE TABLE tbl2 (
  f1 varchar(30) primary key,
  f2 text,
  f3 real
)
sqlite> .schema tbl2
CREATE TABLE tbl2 (
  f1 varchar(30) primary key,
  f2 text,
  f3 real
)
sqlite>
```

The ".schema" command accomplishes the same thing as setting list mode, then

entering the following query:

```
SELECT sql FROM
   (SELECT * FROM sqlite_master UNION ALL
    SELECT * FROM sqlite_temp_master)
WHERE type!='meta'
ORDER BY tbl_name, type DESC, name
```

Or, if you give an argument to ".schema" because you only want the schema for a single table, the query looks like this:

```
SELECT sql FROM
   (SELECT * FROM sqlite_master UNION ALL
    SELECT * FROM sqlite_temp_master)
WHERE type!='meta' AND sql NOT NULL AND name NOT LIKE 'sqlite_%'
ORDER BY substr(type,2,1), name
```

You can supply an argument to the .schema command. If you do, the query looks like this:

```
SELECT sql FROM
   (SELECT * FROM sqlite_master UNION ALL
    SELECT * FROM sqlite_temp_master)
WHERE tbl_name LIKE '%s'
  AND type!='meta' AND sql NOT NULL AND name NOT LIKE 'sqlite_%'
ORDER BY substr(type,2,1), name
```

The "%s" in the query is replace by your argument. This allows you to view the schema for some subset of the database.

```
sqlite> .schema %abc%
```

Along these same lines, the ".table" command also accepts a pattern as its first argument. If you give an argument to the .table command, a "%" is both appended and prepended and a LIKE clause is added to the query. This allows you to list only those tables that match a particular pattern.

The ".databases" command shows a list of all databases open in the current connection. There will always be at least 2. The first one is "main", the original database opened. The second is "temp", the database used for temporary tables. There may be additional databases listed for databases attached using the ATTACH statement. The first output column is the name the database is attached with, and the second column is the filename of the external file.

```
sqlite> .databases
```

The ".fullschema" dot-command works like the ".schema" command in that it displays the entire database schema. But ".fullschema" also includes dumps of the statistics tables "sqlite_stat1", "sqlite_stat3", and "sqlite_stat4", if they exist. The ".fullschema" command normally provides all of the information needed to exactly recreate a query plan for a specific query. When reporting suspected problems with the SQLite query planner to the SQLite development team, developers are requested to provide the complete ".fullschema" output as part of the trouble report. Note that the sqlite_stat3

and sqlite_stat4 tables contain samples of index entries and so might contain sensitive data, so do not send the ".fullschema" output of a proprietary database over a public channel.

CSV Import

Use the ".import" command to import CSV (comma separated value) data into an SQLite table. The ".import" command takes two arguments which are the name of the disk file from which CSV data is to be read and the name of the SQLite table into which the CSV data is to be inserted.

Note that it is important to set the "mode" to "csv" before running the ".import" command. This is necessary to prevent the command-line shell from trying to interpret the input file text as some other format.

```
sqlite> .mode csv
sqlite> .import C:/work/somedata.csv tab1
```

There are two cases to consider: (1) Table "tab1" does not previously exist and (2) table "tab1" does already exist.

In the first case, when the table does not previously exist, the table is automatically created and the content of the first row of the input CSV file is used to determine the name of all the columns in the table. In other words, if the table does not previously exist, the first row of the CSV file is interpreted to be column names and the actual data starts on the second row of the CSV file.

For the second case, when the table already exists, every row of the CSV file, including the first row, is assumed to be actual content. If the CSV file contains an initial row of column labels, that row will be read as data and inserted into the table. To avoid this, make sure that table does not previously exist.

CSV Export

To export an SQLite table (or part of a table) as CSV, simply set the "mode" to "csv" and then run a query to extract the desired rows of the table.

```
sqlite> .header on
sqlite> .mode csv
sqlite> .once c:/work/dataout.csv
sqlite> SELECT * FROM tab1;
sqlite> .system c:/work/dataout.csv
```

In the example above, the ".header on" line causes column labels to be printed as the first row of output. This means that the first row of the resulting CSV file will contain column labels. If column labels are not desired, set ".header off" instead. (The ".header off" setting is the default and can be omitted if the headers have not been previously turned on.)

The line ".once *FILENAME*" causes all query output to go into the named file instead of

being printed on the console. In the example above, that line causes the CSV content to be written into a file named "C:/work/dataout.csv".

The final line of the example (the ".system c:/work/dataout.csv") has the same effect as double-clicking on the c:/work/dataout.csv file in windows. This will typically bring up a spreadsheet program to display the CSV file. That command only works as shown on Windows. The equivalent line on a Mac would be ".system open /work/dataout.csv". On Linux and other unix systems you will need to enter something like ".system libreoffice /work/dataout.csv", substituting your preferred CSV viewing program for "libreoffice".

Converting An Entire Database To An ASCII Text File

Use the ".dump" command to convert the entire contents of a database into a single ASCII text file. This file can be converted back into a database by piping it back into **sqlite3**.

A good way to make an archival copy of a database is this:

```
$ echo '.dump' | sqlite3 ex1 | gzip -c >ex1.dump.gz
```

This generates a file named **ex1.dump.gz** that contains everything you need to reconstruct the database at a later time, or on another machine. To reconstruct the database, just type:

```
$ zcat ex1.dump.gz | sqlite3 ex2
```

The text format is pure SQL so you can also use the .dump command to export an SQLite database into other popular SQL database engines. Like this:

```
$ createdb ex2
$ sqlite3 ex1 .dump | psql ex2
```

Loading Extensions

You can add new custom application-defined SQL functions, collating sequences, virtual tables, and VFSes to the command-line shell at run-time using the ".load" command. First, convert the extension in to a DLL or shared library (as described in the Run-Time Loadable Extensions document) then type:

```
sqlite> .load /path/to/my_extension
```

Note that SQLite automatically adds the appropriate extension suffix (".dll" on windows, ".dylib" on Mac, ".so" on most other unixes) to the extension filename. It is generally a good idea to specify the full pathname of the extension.

SQLite computes the entry point for the extension based on the extension filename. To override this choice, simply add the name of the extension as a second argument to the ".load" command.

Source code for several useful extensions can be found in the ext/misc subdirectory of the SQLite source tree. You can use these extensions as-is, or as a basis for creating your own custom extensions to address your own particular needs.

Other Dot Commands

There are many other dot-commands available in the command-line shell. See the ".help" command for a complete list for any particular version and build of SQLite.

Using sqlite3 in a shell script

One way to use sqlite3 in a shell script is to use "echo" or "cat" to generate a sequence of commands in a file, then invoke sqlite3 while redirecting input from the generated command file. This works fine and is appropriate in many circumstances. But as an added convenience, sqlite3 allows a single SQL command to be entered on the command line as a second argument after the database name. When the sqlite3 program is launched with two arguments, the second argument is passed to the SQLite library for processing, the query results are printed on standard output in list mode, and the program exits. This mechanism is designed to make sqlite3 easy to use in conjunction with programs like "awk". For example:

```
$ sqlite3 ex1 'select * from tbl1' |
>  awk '{printf "<tr><td>%s<td>%s\n",$1,$2 }'
<tr><td>hello<td>10
<tr><td>goodbye<td>20
$
```

Ending shell commands

SQLite commands are normally terminated by a semicolon. In the shell you can also use the word "GO" (case-insensitive) or a slash character "/" on a line by itself to end a command. These are used by SQL Server and Oracle, respectively. These won't work in **sqlite3_exec()**, because the shell translates these into a semicolon before passing them to that function.

Compiling the sqlite3 program from sources

The source code to the sqlite3 command line interface is in a single file named "shell.c" which you can download from the SQLite website. Compile this file (together with the sqlite3 library source code) to generate the executable. For example:

```
gcc -o sqlite3 shell.c sqlite3.c -ldl -lpthread
```