Small. Fast. Reliable.
Choose any three.

Command Line Shell For SQLite

The SQLite library includes a simple command-line utility named **sqlite3** (or **sqlite3.exe** on windows) that allows the user to manually enter and execute SQL commands 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" followed by the name the file that holds the SQLite database. If the file does not exist, a new one is created automatically. The **sqlite3** program will 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 3.6.11
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
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>
```

Aside: Querying the SQLITE_MASTER table

The database schema in an SQLite database is stored in a special table named "sqlite_master". You can execute "SELECT" statements against the special sqlite_master table just like any other table in an SQLite database. For example:

```
$ sqlite3 ex1
SQLite version 3.6.11
Enter ".help" for instructions
sqlite> select * from sqlite_master;
    type = table
    name = tbl1
tbl_name = tbl1
rootpage = 3
     sql = create table tbl1(one varchar(10), two smallint)
sqlite>
```

But you cannot execute DROP TABLE, UPDATE, INSERT or DELETE against the sqlite_master table. The sqlite_master table is updated automatically as you create or drop tables and indices from the database. You can not make manual changes to the sqlite_master table.

The schema for TEMPORARY tables is not stored in the "sqlite_master" table since TEMPORARY tables are not visible to applications other than the application that created the table. The schema for TEMPORARY tables is stored in another special table named "sqlite_temp_master". The "sqlite_temp_master" table is temporary itself.

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
.databases             List names and files of attached databases
.dump ?TABLE? ...      Dump the database in an SQL text format
.echo ON|OFF           Turn command echo on or off
.exit                  Exit this program
.explain ON|OFF        Turn output mode suitable for EXPLAIN on or off.
.genfkey ?OPTIONS?     Options are:
                         --no-drop: Do not drop old fkey triggers.
                         --ignore-errors: Ignore tables with fkey errors
                         --exec: Execute generated SQL immediately
                       See file tool/genfkey.README in the source
                       distribution for further information.
.header(s) 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 on TABLE
.iotrace FILE          Enable I/O diagnostic logging to FILE
.load FILE ?ENTRY?     Load an extension library
.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      Print STRING in place of NULL values
.output FILENAME       Send output to FILENAME
.output stdout         Send output to the screen
.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
.schema ?TABLE?        Show the CREATE statements
.separator STRING      Change separator used by output mode and .import
.show                  Show the current values for various settings
.tables ?PATTERN?      List names of tables matching a LIKE pattern
.timeout MS            Try opening locked tables for MS milliseconds
.timer ON|OFF          Turn the CPU timer measurement on or off
.width NUM NUM ...     Set column widths for "column" mode
sqlite>
```

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 at least 10 characters wide. 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.

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.

Writing results to a file

By default, sqlite3 sends query results to standard output. You can change this using the ".output" command. Just put the name of an output file as an argument to the .output command and all subsequent query results will be written to that file. Use ".output stdout" 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
$
```

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 exactly the above query.

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
```

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
```

Other Dot Commands

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
----  ------------  -----  -----  ----------------------------------
0     ListOpen      0      0
1     Open          0      1      tbl1
2     Next          0      9
3     Field         0      1
4     Integer       20     0
5     Ge            0      2
6     Key           0      0
7     ListWrite     0      0
8     Goto          0      2
9     Noop          0      0
10    ListRewind    0      0
11    ListRead      0      14
12    Delete        0      0
13    Goto          0      11
14    ListClose     0      0
```

The ".timeout" command sets the amount of time that the **sqlite3** program will wait for locks to clear on files it is trying to access before returning an error. The default value of the timeout is zero so that an error is returned immediately if any needed database table or index is locked.

And finally, we mention the ".exit" command which causes the sqlite3 program to exit.

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
```