A quick primer on using the SQLite command line tool

Adapted from
http://www.squidoo.com/sqlitehammer
http://linuxgazette.net/109/chirico1.html

The SQLite Command Line tool is the best way to learn about SQLite and its SQL syntax. It is easy to download for Windows, Linux, or Macintosh, and is easy to use. It will expose you directly to the SQL commands common in other database environments.
Go to the downloads section of sqlite.org and download the command line program for Windows, Linux, or MacOS X.
The command line is a single file program. Just put the program somewhere on your system path, or copy it into the directory in which you are working.
The command line program can be run in interactive mode or batch mode. To start the program in interactive mode, do this

```
>sqlite3 myfirstdatabase.db
sqlite>.quit
```

to create an empty database called 'myfirstdatabase'.

Here are some important commands:

.quit - This command gets you out of the command shell. .exit also works for this.

.dump - This command dumps the entire contents of the database, including schemas (complete with tables, views, indices, triggers, and constraints), and data.

.schema - This command dumps the schema of all of the tables. If you have a table called "Names", and you run ".schema Names", it will just dump the schema for that table. This command is great if you don't remember which tables you have or the names of their fields.

.**help** - This command lists all of the SQLite command-line commands available.

create table table_name (column_name, type_name, ...); This is an important command that lets you create new tables, each with columns.

i**nsert into** table_name values (val1, val2...); - This command lets you insert rows into the table. The number and types of values must correspond to what was defined in the table schema.

select * from table_name; - This allows you to delete all rows from a table.

drop table table_name; - This allows you to get rid of a table.

Here is an example session, started from the command line prompt:

```
[you@localhost ]$ sqlite3 learn.db
SQLite version 3.5.6
Enter ".help" for instructions
sqlite> .schema
sqlite> create table learnTb (a integer, b string);
sqlite> .schema
CREATE TABLE learnTb (a integer, b string);
sqlite> insert into learnTb values (1,"Jay");
sqlite> insert into learnTb values (3,"Kay");
sqlite> insert into learnTb values (5,"May");
sqlite> insert into learnTb values (6,"Ray");
sqlite> .dump
BEGIN TRANSACTION;
CREATE TABLE learnTb (a integer, b string);
INSERT INTO "learnTb" VALUES(1,'Jay');
INSERT INTO "learnTb" VALUES(3,'Kay');
INSERT INTO "learnTb" VALUES(5,'May');
INSERT INTO "learnTb" VALUES(6,'Ray');
COMMIT;

sqlite> select * from learnTb;
1|Jay
3|Kay
5|May
6|Ray
sqlite> delete from learnTb;
sqlite> select * from learnTb;
sqlite> .schema
CREATE TABLE learnTb (a integer, b string);
sqlite> drop table learnTb;
sqlite> .quit
[jgodse@localhost learn]$
```

This session demonstrates the basics of using the SQLite command line program.
Remember that SQL commands end with an "**;**"

If a command runs on your command line, it will run in your program, so this tool is also a good debugging tool for applications that use SQLite.

SQLite is an ASCII database as opposed to the industrial grade MySQL which is a binary database. Database size is limited to 2Gig on SQLite.

Simple Logging With SQLite

SQLite is a great tool for building a logger because it gives you the ability to easily analyze logs with SQL queries, while providing simple file-based storage that you would get with using a file based logger. First create a logfile database with your command line tool.

```
>sqlite3 logs.db
```

Then go in and create a logs table.

```
sqlite3> CREATE TABLE logs (id integer primary key autoincrement, logtype text, logname text, logmessage text, logtime datetime);
```

This table has some interesting features:

1. The id is unique, and increments automatically. i.e. You don't have to keep track of it.
2. You can define custom log types for your application. For example, you can have logs such as 'critical', 'major', 'minor', 'trace'.Restricting yourself to these values allows you to filter based on criticality.
3. The logname allows you to name your logs...i.e. You can store multiple virtual log stream in one table. This allows different people or subsystems to own their own logs.
4. The logmessage lets you put in other useful information.
5. The logtime field allows you to take a time stamp.

After this, quit out of the logger;

```
sqlite3> .exit
```

How to log? The basic form of the SQL statement is as follows:

```
insert into logs (id,logtype,logname,logmessage,logtime) values (NULL, 'trace', 'UIEvent', 'mousepress on Form XYZ', datetime('now'));
```

Points to note:

1. Putting in NULL in the id field gets the database to autoincrement it.
2. datetime('now') is a built-in SQLite function that yields the current time stamp.

You probably want to have an API that takes in the logtype, logname, and logmessage as input parameters, and then calls the SQL API in the language of your choice. You'll also want to have function to initialize the log by opening a connection to logs.db, and holding it in a globally accessible handle or variable. I'll leave it as an exercise to wrap the insert statement in a log API call.

After your application uses your logs, you'll notice that logs.db has grown in size.

You can use the command line tool to analyze the logs in logs.db, or build another program to do the same.

Here are some of the queries you can perfrom on the log:

1. Show all logs in the last day (86400 seconds) => select * from logs where strftime('%s', 'now')-strftime('%s', logtime) < 86400;
2. Show all logs in the last hour (3600 seconds) => select * from logs where strftime('%s', 'now')-strftime("%s', logtime) < 3600;
3. Show all critical or major logs in the last 15 minutes (900 seconds) => select * from logs where strftime('%s', 'now') - strftime('%s', logtime) < 900 and logtype in ('critical','major');
4. Show all non-trace logs => select * from logs where logtype not in ('trace');
5. Show how many of each type of log in the last 6 hours => select logtype, count(logtype) from logs group by logtype;

NOTES:

Backup
Since an SQLite database is stored in one file with a name of your choice, just copy the file to your backup set. If you don't trust the binary file format (and a lot of folks don't trust it for any database), you could just dump the database to another file as a set of sql commands. For example, if your database is in myData.db, then you just do:

> sqlite3 myData.db ".dump" > backup.txt
"backup.txt" contains the SQL statements of the database.

Restore
Since an SQLite database is stored in one file with a name of your choice, just copy the file from your backup medium to the place where your database file is expected. Alternatively, if you stored it as SQL text, as above, and you want to restore the data from backup.txt above to a new file (newData.db) just run:

>sqlite3 newData.db ".read backup.txt"

Compression
If you want to compress the file, then use your favourite compression utility on the database file such as gzip or winzip. If your database file is getting large, and unwieldy, it could be because over time you have deleted stuff. SQLite leaves these records allocated. To clean up allocated but unused records, run the "vacuum" command through the database API, or run the following on the database file from the command line:

> sqlite3 myData.db "vacuum"
If you want a compressed read-only database, it'll cost you, but it isn't a lot of money if space is that critical. See http://www.hwaci.com/sw/sqlite/prosupport.html for details.

Encryption
There are free and paid versions of SQLite that encrypt the data files. If you have budget, get it from Hwaci software. I did it in a previous gig and it worked very well. You basically get one extra file with all of the encryption algorithms and APIs, and you just compile it and link it into your project. Your database connection APIs change because you need a password, but that's about it. Hwaci's version is very easy to use. It cuts performance by about half.

Authorization & Access Control

Since the only access to the database file is through the file system or programming API, you get the first level of access restriction through your file system priveleges for each user. (It naturally follows that there is no such notion as "GRANT" and "REVOKE"). Once you are "in" to the database, you have full access. Any limited access to specific tables and fields is through the application that embeds SQLite. This works because SQLite is a serverless database. i.e. Your application is the server, and that is where you do the whole ACL, roles & rights, authorization, authentication, et cetera.

Integrity Check

To run integrity check, run the "pragma integrity_check" command from the API or the command line. For example, do:

> sqlite3 test.db "pragma integrity_check"

You can access SQLite from within python. Here is a simple example that shows you how to build a database, a table, populate the table and get data from it.

EXAMPLE

```
#!/usr/bin/python
# python and sqlite

#import sql
import sqlite3

#create and connect to a database
con = sqlite3.connect('mydatabase.db')
cur = con.cursor()

#create a table
cur.execute('CREATE TABLE tableA (id INTEGER PRIMARY KEY, name VARCHAR(50))')

#insert data
cur.execute('INSERT INTO tableA (id, name) VALUES(NULL, “samuel”)')
con.commit()

#retrieve data
print cur.lastrowid
cur.execute('SELECT * FROM tableA')
        print cur.fetchall()
```