Sqlite Visualiser

A look inside Sqlite.

By:
Paul Batty

Supervisor:
Andrew Scott

March 2016

The dissertation is submitted to
Lancaster University
As partial fulfilment of the requirements for the degree of
Integrated Masters of Science in Computer Science

Abstract

The Abstract.

1 Introduction

Sqlite unlike many other databases is a small, single file, self-contained database engine often used in embedded systems, storage or as an application file. Sqlite is used in many applications such as Firefox, Android and Windows 10. In addition to its wide adaptation Sqlite is server less, and has zero configuration putting it in a unique place among the other alternative systems. Despite the extensive research and testing performed on Sqlite none have attempted to visualise this data in real time.

This paper will help provide a way to see the Sqlite database in action, providing a useful tool for developers and researchers alike in understanding and debugging the internal structure of their own databases. In order to accomplish this paper will:

- Explore in depth the how the file format is put together (section 2). And how to traverse the file (section 2).
- Look at the design and development (section 3) including testing (section 6) of the tool. And how it takes this data and visualises it (section 4). Including the user experience (section 5).
- Evaluation of the tool (section 7) and where this research could be taken beyond this paper (section 8).

2 Background

2.1 The Problem

Throughout Sqlite's history many tests, papers, and tools have been developed in order to understand, and modify the future direction of Sqlite. However, when a user wants to understand at a deeper level how Sqlite is working or finding obscure bugs, they are stuck with manually trawling through a Hex editor. This paper aims to solve this by providing a visualisation of the internal structure, as well as a update log that is updated in real time, when the database is modified.

2.2 Sqlite

2.2.1 What is Sqlite

Sqlite is a single self-contained, serverless SQL database engine. Started on 29 May 2000 by D. Richard Hipp (Hipp, 2000) from gathered inspiration while working on software for guided missiles on a battleship where they needed a self-contained portable database. (Owens, 2006) He joined up with Joe Mistachkin followed by Dan Kennedy in 2002. Version 1.0 was released in August 2000, then in just over year on the 28 November 2001 2.0 which introduced, BTrees and many of the features seen in 3.0. Which came a lot later containing a full rewrite and improvement over 2.0, with the first public release on 18 June 2004. At the time of writing this paper we are currently sitting at version 3.10.4 (Hipp, 2000).

Sqlite is open source within the public domain making it accessible to everyone. The entire library size can be 350Kib, with some option features omitted it could be reduced to around 300Kib making it incredibly small compared to what it does. In addition to this the runtime usage is minimal with 4Kib stack space and 100Kib heap, allowing it to run on almost anything. Sqlite's main strength is that the entire database in encoded into a single portable file, that can be read, on any system whether 32 or 64 bit, big or small endian. It is often seen as a replacement for storage files rather then a database system (Hipp, 2000).

2.2.2 Where is Sqlite used

As Sqlite is a minimal portable database engine it has primarily two uses, The first as a relation database like any other, the second as a application file format. In the former case Sqlite would be set up the same way a tradition database system would be, with the primary purpose to hold the back end storage information. The latter case is more unique to Sqlite and is what sets it apart from the traditional database engines (Hipp, 2000). Because of this Sqlite is used everywhere, a few of the big names include, apple, android, adobe even a special version was produced specifically for windows 10. In fact Sqlite might be the single most deployed software currently (Hipp, 2000, 2015).

2.2.3 How is Sqlite works

Sqlite consist of three main parts, the backend, core and the SQL compiler. Figure 1 shows the architectural digram of Sqlite.

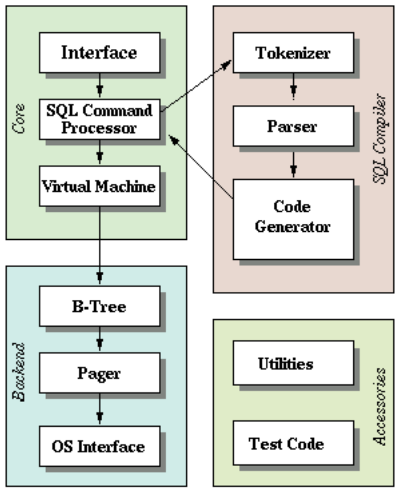

Figure 1: Sqlite architectural diagram (Hipp, 2000)

The SQL compiler is designed to take SQL strings and turn them into valid SQL commands, it is made up of three parts. Firstly the tokenizer takes a string containing SQL statements and turns it into tokens passing them one by one to the parser. The parser takes the tokens and assign meaning to them. Lastly the code generator takes the tokens from the parser and assembles complete SQL statements that are to be ran on the database file.

The Core itself is actually a virtual machine implementing a specifically designed computing engine to manipulate database files. The interface module defines the interface between the virtual machine and the SQL library including the external API. Knowing this we can see that the virtual machine takes the code output from the code generator to manipulate the backend / database.

The final main module is the backend which is the main focus of this paper, controlling the file format. The OS interface contains an abstraction layer to write the files to disk or memory. The Pager takes the B-Trees and is responsible for reading, writing and caching them. This involves locking, rollback and atomic commits of the database. Sqlite uses a B-Tree system to navigate and store the data on disk, this module contains the implementation of the B-tree and as such the defines the file format.

The last module, accessories is made up of two parts. Utility containing functions that are used all around Sqlite, such as memory allocation, string comparison, random number generator and symbol tables. The test section contains all the test scripts and only exist for testing purposes, of which contains over 811 times more code then the actual project.

2.3 The Sqlite file format

2.3.1 The page system

As mentioned is section 2.2.3 Sqlite works off of a B-tree structure to store and navigate the database. The B-Tree implementation is designed to support fast querying, the various B-Tree structures can be found in Comer (1979) paper. Sqlite also takes some improvements seen in Knuth (1973) book (Raymond, 2009).

The basic idea is that the file is made up of chunks or pages, each page is the same fixed size. The size of the pages are a power of two between 512 - 65536 bytes. Pages are numbered starting with 1 instead of 0. The maximum number of pages that Sqlite can hold is 2,147,483,646 with a page size of 512 bytes witch is around 140 terabytes. The minimum number of pages within a database is 1. There are five types of pages:

- Lock Byte Page
  The lock byte page appears between bytes, 1073741824 - 1073742335, if a database is smaller or equal to 1073742335 bytes it will not contain a lock byte page. It is used by the OS Interface mentioned in section 2.2.3.

- Freelist Page
  The freelist page is a unused page, often left behind when information is deleted from the database. The other type is a freelist trunk page containing page numbers of the other freelist pages.

- B-Tree Page
  The B-Tree page, contains one of the four types of B-Trees, more in section 2.3.3.

- Payload overflow page
  The payload overflow page is created to hold the remaining payload from a B-Tree cell when the payload is to large.

- Pointer map page
  Pointer map pages are inserted to make the vacuum modes faster. And are the reverse B-Tree going child to parent rather then parent to child. They exist in databases that have a non-zero value largest root B-Tree within the header. The first instance of these pages are at page 2.

2.3.2 The header

The first page before the root B-Tree contains a 100 byte header. This is what makes Sqlite have zero configuration as all the database settings are stored within this header, all multibyte fields are stored in a big-endian format. The header follows the following format:

Byte Offset	Byte Size	Description
0	16	A UTF-8 Header String followed by null terminator read as: "SQLite format 3" or in hex: "53 51 4c 69 74 65 20 66 6f 72 6d 61 74 20 33 00".
16	2	Page Size in bytes, power of two between 512 - 65536 bytes. if using version 3.7.0.1 and earlier between 512 - 32768, or 1 for 65536.
18	1	Write version, 1 for legacy; 2 for WAL.
19	1	Read version, 1 for legacy; 2 for WAL.
20	1	Bytes of unused space at the end of each page. This space is used by extensions, such as cryptographic to store a checksum, but normally 0.
21	1	Maximum embedded payload fraction, must be 64. Was going to be used to determine the maximum size of a B-Tree cell on a index B-Tree.
22	1	Minimum embedded payload fraction, must be 32. Was going to be used to determine the portion of a B-Tree cell that has one or more overflow pages on a index B-tree.
23	1	Leaf payload fraction, must be 32. Was going to be used to determine the portion of a B-Tree cell that has one or more overflow pages on a leaf or table B-Tree.
24	4	File change counter. It counts the number of times the database is unlocked after being modified. May not be incremented in WAL mode.

Byte Offset	Byte Size	Description
28	4	Size of the database in pages, Total number of pages.
32	4	Page number of first freelist page, or 0 if no freelist.
36	4	Number of freelist pages.
40	4	Schema Cookie. The schema version, each time the schema is modified this number is incremented.
44	4	Schema format number. either 1, 2, 3 or 4. 1. Format support back to version 3.0.0. 2. Varying number of columns within the same table. From Version 3.1.3. 3. Extra column can be non-NULL values. From Version 3.1.4. 4. Respects DESC keyword and boolean type. From Version 3.3.0.
48	4	Page cache size. suggestion only towards Sqlite's pager.
52	4	Page number of largest root B-Tree, when in vacuum mode else 0.
56	4	Text encoding. 1 for UTF-8. 2 for UTF-816le. 3 for UTF-816be.
60	4	User version. Set by and read by the user, not used by Sqlite.
64	4	Incremental-vacuum mode. Non 0 for true. 0 for false
68	4	Application ID. Used to associate the database with a application. 0 is Sqlite3 Database
72	20	Empty, Reserved for expansion.
92	4	Version-valid-for-number. Value of the change counter when the Sqlite version number was stored.
96	4	Version. Sqlite version.

Table 1: Sqlite Header, modified from Hipp (2000)

Immediately following the header is the root B-Tree which we will cover the the next section.

2.3.3 The Trees and Cells

The Trees and cells...

2.3.4 Encoding of the data

The Data is...

2.4 Similar Programs

2.4.1 Sqlite browser

One Similar program...

3 Design

3.1 System architecture

3.1.1 High level Overview

The Overall design...

3.1.2 Module Overview

The first module..

4 Implementation

4.1 The tools

I used..

4.2 The Modules

4.2.1 Database parser

The Database parser...

4.2.2 Log

The Log...

4.2.3 Live Updater

The Live updater...

5 System Operation

6 Testing

6.1 Code Tests

6.1.1 Unit tests

Unit testing...

6.1.2 Integration tests

Integration tests...

7 Evaluation

7.1 System Performance

The system was...

7.2 Design principles

I followed..

8 Conclusion

9 References

Comer D. (1979) Towards Computing Surveys. The Ubiquitous B-Tree, Computing Surveys, Vol 11, No. 2. Purdue University, West Lafayette, Indiana, June 1979, pages 121 - 137.

Knuth E. D. (1973) The Art Of Computer Programming, Volume 3: ”Sorting And Searching”, Addison-Wesley Publishing Company, Reading, Massachusetts. Pages 473 - 480.

Owens M. (2006). The Definitive Guide to SQLite, Berkeley, California, Apress.

Hipp R. (2000) Sqlite. On line publication, Wyrick Company, Inc, https://www.sqlite.org/. Last Accessed 17th January 2016.

Hipp R. (2015) SQLite: The Database at the Edge of the Network. On line Video, Skookum, https://www.youtube.com/watch?v=Jib2AmRb_rk. Last Accessed 17th January 2016.

Raymond, (2009) SQlite. On line publication, http://ray.bsdart.org/man/sqlite/. Last Accessed 17th January 2016.

10 Appendix

11 Code

11.1 More code

11.1.1 Even more code

This is some very important code. This is a very long sentance in order to see hoow latex copes with very very long lines of non stop text.

```
    // main
    public static void main(String args[]) {
        System.out.println("Hello␣World");
    }
```

And so on..