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.

Introduction

Having studied Databases in my previous year, including SQLite. I remember being amazed that it could do so much without the need to configure, manipulate, or go through a long winded install process. It was so simple and flexible, anyone could use it. In fact SQLite takes pride that it is probably one of the most widely deployed database engines. And one of the top five most deployed software modules. Alongside zlib, libpng and libjpg. It finds itself inside all of the top browsers (Firefox, Google chrome and possibly Edge), Operating systems (Windows 10, IOS and embed OS's) and in the most unexpected places such as aircraft.

After doing some reading and looking at the SQLite claim to fame. I began to take a closer look at its systems. What makes it so flexible, fast and simple, to use. My main focus however was on the file format that it uses to store the entire database. This included many long nights looking at how it was put together. How to traverse it. And why it is the way it is. I also looked at the available tools that are available for SQLite. This is covered in the first section of this paper.

Understanding the file format was just the first stepping stone, as I then undertook a journey to build a tool that could traverse and read the file. While recording every operation that was and ever will be performed to it. This is covered in the second and third chapters.

While building by tool. I kept two things in mind. How it operated form a users perspective and the best ways to break it. This was to ensure that the tool was open to everyone, and would not fall down and crumble, at the first chance it got. This is covered in sections four and five.

Once the tool became well developed. I started looking towards the future of the tool. What could be added to make it ever more useful for developers, researchers and anyone else that is using SQLite systems. This is covered in the final sections, six and seven.

The main aim of this paper is to help you understand the SQLite file format and systems. While providing a useful tool that can help debug, manipulate and record your own SQLite databases, without the need for a hex editor.

1 Background

To begin with we will go over some programs that provide support for SQlite. The turn our attention to SQLites beginnings and where it is used. Starting back in spring of 2000. Then look at the SQLite file format in great depth, to understand it inside out, and how to parse it. Before moving onto the other sections.

1.1 Similar programs

While searching the vast web for other tools, that could help me. I only came across two different types. The user interface, that removed having the need for SQL and the command line. Or very technical tools, and no middle ground. This was particularly interesting as the number of user interface tools where everywhere, while I could only find one technical tool.

1.1.1 Sqlite browser

The first tool I came across was the SQlite browser made by Piacentini (2003) based off of the Arca database browser. Out of all the user interface tools that I came across this was the most polished. It allowed users to open, view and manipulate the database. Without having to learn SQL, or the command line. With the main aim to be as simple as possible.

Apart from the usual features, such as viewing tables, schemas, and modifying them. The more unique features allowed exporting the tables to CSV, producing SQL dumps, and acting as a sandbox. The sandbox allowed users to execute commands, see the changes but, nothing was actually performed until the user committed the changes onto the database. In addition this it provided a fully fledged SQL editor with auto-complete, syntax highlighting and loading and saving of commands in external files.

The tool itself was made in C/C++ and QT with support for SQlite databases up to version 3.8.2. It is available for all major platforms. While I was using this tool I found it simple and intuitive to navigate. It also was very powerful and did exactly what it said on the tin. However, it did not allow a insight into SQLite nor the logging of commands, produced by external programs.

1.1.2 Sqlite fragmentation

The second tool, is very unique. It showed a fragmented view of the database. Made by laysakura (2012). Written in python and published to Github. It only did

one thing but did it well. As we will find out later on. The file is made up of pages. This tool would scan the file, and produce a visualisation of the fragmentation status of each page. Much like that old Windows XP de-fragmentation tool.

The tool is ran though the command line, and produces a Json file with the analysis, before sending it to the visualiser that produces a SVG image output. This allows any type of out to be added in. Some of the more advanced features, is filtering certain pages out or in.

Although it is very powerful, it does not support WAL mode, slow on larger databases and can only cope with UTF-8 text support. But it provides a very useful insight into SQLite. On top of this, it clearly lays out the page format of the file. Which is very similar to where my tool is going.

1.2 What is Sqlite

D. Richard Hipp, the author of SQLite, Originally got the idea while working on a battleship. He was tasked with developing a program for the on board guided missiles. While working on the software used the database system Informix. That took hours to set up and get anywhere near useful. For the application that he was building, all he needed was a small self-contained, portable and easy to use database. Rather then the bloated mess that was Informix (Owens, 2006).

Speaking to a colleague in January of 2000, Hipp, disused his idea for a self contained embedded database. When some free time opened up on the 29th of May 2000, SQLite was born. It was not until August 2000 that version 1.0 was released. Then in just over a year on the 28 November 2001 2.0 which introduced, B-Trees and many of the features seen in 3.0 today. During the next year he joined up with Joe Mistachkin followed by Dan Kennedy in 2002. To help work for the 3.0 release. 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. After changes to the naming scheme, version 3 is currently supported to 2050. (Hipp, 2000).

Today, SQLite is open source within the public domain making it accessible to everyone. The entire library size is around 350Kib, with some optional 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. In fact Hipp (2000) has stated, ”Think of SQLite not as a replacement for Oracle but as a replacement for fopen()”.

1.3 Where is Sqlite used

SQLite being a relational database engine. As well as a replacement for fopen() (Hipp, 2000). Allows it be truly used for anything. Because of this SQLite might be the single most deployed software currently. Alongside zlib, libpng and libjpg. With the number in the billions and billions (Hipp, 2000).

Because of this SQLite can be found anywhere. Microsoft even approached Hipp, and asked for a special version to be made for use in Windows 10 (Hipp, 2015). In addition to Microsoft. Apple, Google and Facebook all use SQLite, somewhere within their systems. On top of all the big names. You can find it, within any another consumer device, such as Phones, Cameras and Televisions. This wide usage was picked up by Google and Hipp was awarded Best Integrator at OReillys 2005 Open Source Convention (Owens, 2006).

1.4 How Sqlite works

SQLite has a simple and very modular design. Consisting of eleven modules, and four subsystems. The backend, The core, The SQL compiler, and Accessories. Figure 1.1 shows the architectural digram of Sqlite.

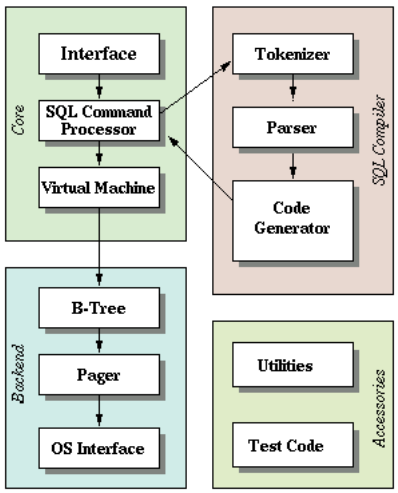


Figure 1.1: Sqlite architectural diagram (Hipp, 2000)

1.4.1 The SQL Compiler

The SQL compiler takes SQL strings and converts them into the core's virtual machine assembly language. The process starts with the tokenizer and parser. They both work closely together. Taking the SQL string and validating the syntax. Before converting it into a hierarchical structure. For use by the code generator. Both systems are custom made for SQLite. With the parser going under the name of Lemon (Owens, 2006). Lemon is designed to optimise performance and guard against memory leaks. Once they have successfully converted the SQL string into a Parse Tree, the parser passes it onto the code generator.

The code generator takes the parse tree from the parser, and translates it into a assembly language program. The program is written in a assembly language that is specifically designed for SQLite. It is ran by the virtual machine inside the core module. Once the SQL program is made it sends it off to the virtual machine for execution.

1.4.2 The Core

The Core itself is actually one single virtual machine implementing a specifically designed computing engine to manipulate database files. The language contains 128 instructions, all designed to manipulate and interact with the database, or prepare the machine for such operations. Figure 1.2 shown an example program.

```
1   SQL = SELECT * FROM examp;
2   addr  opcode       p1    p2    p3
3   ----  ------------ ----- ----- ----------------------
4   0     ColumnName   0     0     one
5   1     ColumnName   1     0     two
6   2     Integer      0     0
7   3     OpenRead     0     3     examp
8   4     VerifyCookie 0     81
9   5     Rewind       0     10
10  6     Column       0     0
11  7     Column       0     1
12  8     Callback     2     0
13  9     Next         0     6
14  10    Close        0     0
15  11    Halt         0     0
```

Figure 1.2: Select operation program from Hipp (2000)

The interface module defines the interface between the virtual machine and the SQL library. All libraries and external application use this to communicate with SQLite.

Knowing this we can see that the virtual machine takes the SQL input from the Interface, passes it onto the SQL Compiler. Then Collecting the outputted program from the code generator. And executing this program to perform the original request that was sent in. Making the the heart or core of SQLites operations.

1.4.3 The Backend

The final main module we will look at is the backend. It deals with the file interactions, such as writing, reading and ordering of the file. The B-Tree and pager work closely together to organise the pages, both of which do not care for the content. The B-Tree module is like a factory that maintains and sorts the relationships between each of the different pages within the file. Forming them into a tree structure that makes it easy to find what you are after.

The OS Interface is a warehouse, providing a constant interface to access the disk. It handles the locking, reading and writing of files across all types of operating systems. So the pager does not have to worry about how it is implemented, it can just tell it what it wants.

Lastly, the Pager is the transport truck, going between between the B-Tree (factory) and the OS interface (storage) to deliver pages at the B-tree requests. It also keeps the most commonly used pages in its cache, so it does not have to keep going through the OS interface in order to collect the pages, since it already has them.

1.4.4 The Accessories

The last module, accessories is made up of two parts. Utility and tests. The utility module contains functions that are used all across Sqlite, such as memory allocation, string comparison, random number generator and symbol tables. This basically acts as a shared system for all parts of SQlite. 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, and million of test cases. As it covers every possible code path through SQLite. This is partly why it is considered to be so reliable.

1.5 The Sqlite file format

1.5.1 The page system

As I mentioned in section 1.4.3 the B-Tree module looks after the pages including the organisation and relationships between them. And then packs them into a tree structure. This is the same structure that gets written to disk. 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 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 operating system not SQLite.
- 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 1.5.3.
- 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.

This paper will not cover the lock byte and pointer map pages.

1.5.2 The Header

The first step in parsing the SQLite file before we tackle the different pages is to read in the SQLite header. This is the first 100 bytes located in page one. The header stores all the necessary information to read the rest of the file. so reading it correctly is crucial. Immediately following the header is the root B-Tree which we will cover the the next section. Table 9.1 show the header layout. All multibyte fields are stored in a big-endian 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.
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.

Byte Offset	Byte Size	Description
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 1.1: Sqlite Header, modified from Hipp (2000)

1.5.3 The Trees and Cells

As mentioned in section 1.5.1 the file is split into pages and each page contains one of four types of pages. However each page is linked together in a B-Tree format, where each page represents a node in the tree. This is what the B-Tree module takes care of as mentioned in section 1.4.3. Below figure 1.3 shows an example file B-Tree.

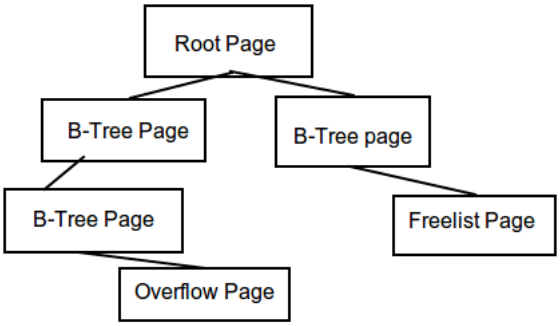


Figure 1.3: B-Tree page structure.

One thing to note is how the file is made up of mainly B-Trees. This is as briefly mentioned in the last section, pointer maps, lock byte and overflow pages only appear when the requirements are met. And Freelist pages when enough data has been deleted. This leaves the only B-Tree pages.

Each B-Tree page is slightly different but follow the same pattern. At the start of each page there is the B-Tree / page header. Following the header is a array of pointers to their cells. The cell layouts are where the main difference in B-Tree types become more apparent, as they store the payload. Figure 1.4 shows the full layout of the SQlite file.

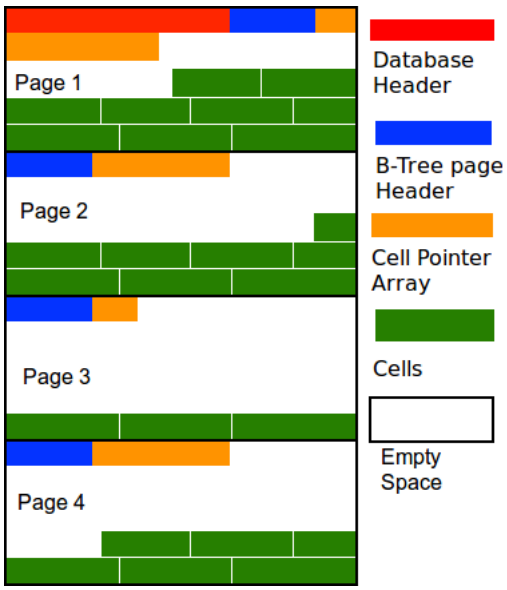
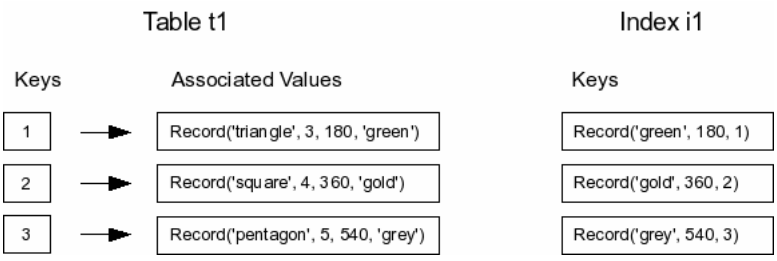


Figure 1.4: Sqlite file format, modified from Drinkwater (2011)

The cells start at the end of the page and work backwards towards the top.

There are two main types of B-Trees. Table and Index, both of which uses a key-value system in order to organise them. The Table B-Trees use 64 bit integers also known as row-id or primary key. The Index B-Trees uses database records as keys. This can be seen in the following example taken from Raymond (2009):

```
1 CREATE TABLE t1(a INTEGER PRIMARY KEY, b, c, d);
2 CREATE INDEX i1 ON t1(d, c);
3
4 INSERT INTO t1 VALUES(1, 'triangle', 3, 180, 'green');
5 INSERT INTO t1 VALUES(2, 'square', 4, 360, 'gold');
6 INSERT INTO t1 VALUES(3, 'pentagon', 5, 540, 'grey');
```



the format.

Figure 1.5: Example key pair database (Raymond, 2009)

Those two types of B-Trees are broken down into Leaf and Interior. The leafs are located at the end of the tree and contain no children. Whereas the interior will always have at least one single child. In addition to this all database records / values within the B-Trees are sorted using the following rules written by Raymond (2009):

1. If both values are NULL, then they are considered equal.
2. If one value is a NULL and the other is not, it is considered the lesser of the two.
3. If both values are either real or integer values, then the comparison is done numerically.
4. If one value is a real or integer value, and the other is a text or blob value, then the numeric value is considered lesser
5. If both values are text, then the collation function is used to compare them. The collation function is a property of the index column in which the values are found
6. If one value is text and the other a blob, the text value is considered lesser.

7. If both values are blobs, memcmp() is used to determine the results of the comparison function. If one blob is a prefix of the other, the shorter blob is considered lesser.

Overall the four type of B-Trees found inside Sqlite are:

- Index B-Tree Interior
- Index B-Tree leaf
- Table B-Tree Interior
- Table B-Tree leaf

In the case of index B-Trees, the interior trees contains N number of children and N-1 number of database records where N is two or greater. Whereas a leaf will always contain M database records where M is a one or greater. The database records stored inside a Index B-Tree are of the same quantity as the associated database table, with the same fields and columns. between the tables and rows. As can be seen above in figure 1.5. Index trees are used by SQlite to keep track the the foreign keys and row relationships.

The Table B-Trees are a little different as they store most of the data. Unlike index B-Trees the interior trees contain no data but only pointers to children B-Trees, as all the data is stored on the leaf nodes. The interior trees contain at least one pointer, and the leaf node contains at least one record. For each table that exists in the database, one corresponding Table B-Tree also exists, and that B-Tree contains one entry per row, appearing in the same order as the logical database. Figure 1.6 show the physical layout of the Table B-Tree.

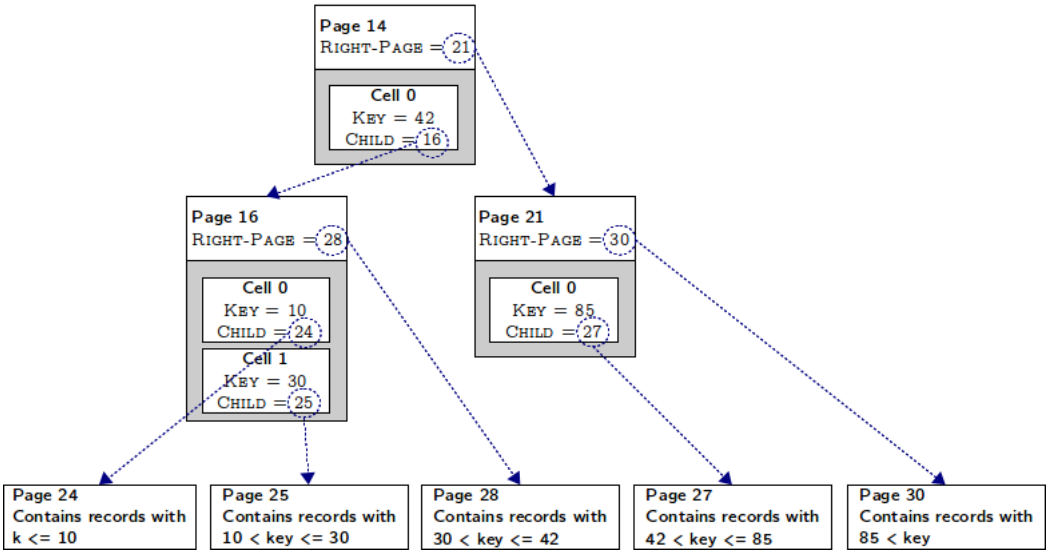


Figure 1.6: Physical layout of table B-Trees (Sotomayor, 2010)

1.5.4 Encoding of the data

Sqlite uses a variable length integer or 'varint' in order encode some of the values inside the database, since they use up less space for small positive values. A varint is a static Huffman encoding of a 64-bit two complements integer. A varint is between and 1 - 9 bytes in size. The maximum number a byte can hold is 127 as the most significant bit is needed as a flag unless we are in the ninth byte where all the bits are used. If the most significant bit is set then we need the next byte. So if it is set in byte 1 then we need byte 2 and so on.

If we have the following value in hex 5B and convert this to binary we have 01011011 as we can see the the most significance bit is not set leaving us with the value 91. However, if we have the value in hex 84 and convert this to binary we have the value 10000100, the most significant bit is set this means we need the next byte which has a value in hex of 60 converting this to binary leaves us with 01100000 meaning that this varint is two byes long. In order to create the final value we need to concatenate them together leaving out the most significant bit. creating the value 00001001100000 giving us a total value of 608 in decimal (Drinkwater, 2011). Table 1.2 show the all combinations of varints.

Bytes	Value Range	Bit pattern
1	7	0xxxxxxx
2	14	1xxxxxxx 0xxxxxxx

Bytes	Value Range	Bit pattern
3	21	1xxxxxxx 1xxxxxxx 0xxxxxxx
4	28	1xxxxxxx 1xxxxxxx 1xxxxxxx 0xxxxxxx
5	35	1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 0xxxxxxx
6	42	1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 0xxxxxxx
7	49	1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 0xxxxxxx
8	56	1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 0xxxxxxx
9	64	1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx xxxxxxxx

Table 1.2: Varint combinations Raymond (2009)

1.5.5 B-Tree header

As mentioned in section 1.5.3 the header for each type of B-Tree does not vary that much. Table 1.3 shows the header for the B-Trees.

Byte Offset	Byte Size	Description
0	1	The type of of B-Tree. Value of 2 is a interior index B-tree. Value of 5 is a interior table B-tree. Value of 10 is a Leaf index B-tree. Value of 13 is a Leaf table B-tree.
1	2	Offset of first freeblock on the page.
3	2	Number of cells on page.
5	2	Start of content area. A zero is seen as 65536.
7	1	Number of fragmented free bytes within the cells.
8	4	Interior B-Tree Pages only. The right most pointer.

Table 1.3: Sqlite B-Tree Header, modified from Hipp (2000)

Immediately following the header is the array of cell pointers, the number of cells is read at offset 3. Each cell pointer is 2 bytes in size. It is worth noting at this

point that the pointer and offsets start at the page offset rather then the start of the file, keeping each page self contained. Therefore in order to follow the cell pointers or the other offsets the following sum is needed to calculate its position in the file:

```
    cell = ((pageNumer - 1) * pageSize) + offset;
```

The right most pointer within interior B-Tree pages is the childs page number not offset therefore to calculate the page offset the following sum is used:

```
    pageOffset = ((pageNumer - 1) * pageSize);
```

1.5.6 Index B-Tree cell

As mentioned in section 1.5.3 Index B-Tree use the database records as keys. their content also reflects this. Table 1.4 show the layout of the cell:

Data type	Description
4 byte integer	Page number of child. Not on leaf cells.
Varint	Payload size.
byte array	Payload
4 byte integer	Page number of overflow Only if payload is to large.

Table 1.4: Index B-Tree cell

Much like the right child pointer mentioned in section 1.5.5 this is the page number of the child not a pointer. In order to determine if there is a overflow page the following calculation is used:

```
usable-size = page-size - bytes-of-unused-space;
max-local = (usable-size - 12) * max-embedded-fraction / 255 - 23;

if (payload-size > max-local) {
    we have a overflow page.
}
```

Where bytes-of-unused-space is read in the Sqlite header at offset 20 and max-embedded-fraction at offset 12. Once we know there is an overflow we can use the following calculation to work out the size of the record in this part of the cell before jumping over to the overflow page:

```
usable-size = page-size - bytes-of-unused-space;

min-local = (usable-size - 12) * min-embedded-fraction / 255 - 23;
record-size = min-local + (record-size - min-local) %
             (usable-size - 4);

if(record-size > max-local) {
    record-size = min-local;
}
```

1.5.7 Table B-Tree cell

The Table B-Trees as mentioned in section 1.5.3 hold most of the data, they also use row id's or primary keys as the keys to the records. Firstly the interior type only has two fields and no need for overflow. They follow the following format in Table 1.5:

Data type	Description
4 byte integer	Page number of child
Varint	Row id.

Table 1.5: Page B-Tree interior cell

Much like the right child pointer mentioned in section 1.5.5 this is the page number of the child not a pointer.

The Leaf type is a little more complex, this can be seen the Table 1.6 below:

Data type	Description
Varint	Size of payload.
Varint	Row id.
byte array	Payload.
4 byte integer	Page number of overflow Only if payload is to large.

Table 1.6: Page B-Tree leaf cell

In order to determine if there is a overflow page the following calculation is used:

```
usable-size = page-size - bytes-of-unused-space;
max-local := usable-size - 35;

if (payload-size > max-local) {
    we have a overflow page.
}
```

Where bytes-of-unused-space is read in the Sqlite header at offset 20 and max-embedded-fraction at offset 12 (see section 1.5.2 for more info). Once we know there is an offset we can use the following calculation to work out the size of the record in this part of the cell before jumping over to the overflow page:

```
usable-size = page-size - bytes-of-unused-space;

min-local = (usable-size - 12) * min-embedded-fraction / 255 - 23;
max-local = usable-size - 35;

local-size = min-local + (record-size - min-local) %
             (usable-size - 4);

if( record-size > max-local );
    record-size = min-local
```

1.5.8 Overflow Page

Overflow pages as mentioned in section 1.5.1 are used to store the payload when it is to large to fit in a single cell. overflow pages form a link list with the first four bytes point to the next page number in the chain or zero if it's the last. Following the four bytes through to the last bytes is the payload content. Table 1.7 show the layout of an overflow page.

Data type	Description
4 byte integer	Page number of next page in chain, or zero if last.
byte array	Payload.

Table 1.7: Overflow page

1.5.9 Payload / Record format / Byte array

Throughout the previous sections the payload has been called the data type byte array or database record. The payload follows a very specific pattern, and is used to store the schema as well as rows or records. It is split up in to two parts the cell header and cell content. The full record format can be seen in figure 1.7.

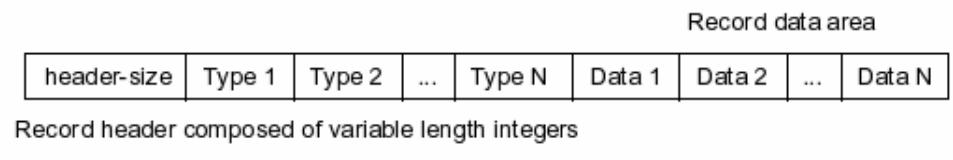


Figure 1.7: Database record format (Raymond, 2009)

The cell header is made up of N + 1 varints where N is the number of values in the record. The first varint is the number of bytes in the header. As varints can be between 1 - 9 bytes it is important to keep track of each of fhe following N varints size to count the number of values, as briefly mentioned in section 1.5.4. They contain the values types. The different value types can be seem below in Table 1.8

Header Value	Byte Size	Description
0	0	Null
1	1	1 byte signed integer
2	2	2 byte signed integer
3	3	3 byte signed integer
4	4	4 byte signed integer
5	6	6 byte signed integer
6	8	8 byte signed integer
7	8	8 byte IEEE floating point
8	0	Value 0, Schema 4 or greater only
9	0	Value 1, Schema 4 or greater only
10,11	0	Reserved for expansion
$\geq$ 12 and even	(N-12)/2	BLOB of size (N-12/2) long
$\geq$ 13 and odd	(N-13)/2	String of size (N-13/2) long

Header Value	Byte Size	Description

Table 1.8: Database record cell types

The cell content as shown in figure 1.7 follows the same format layout in the header, with the content size and type specified in table 1.8. Where the size is 0 there is no varint to read from the data section and should be skipped.

1.5.10 B-Tree and Schema Table

As mentioned in section 1.5.2 following the Sqlite header is root B-Tree. The root B-Tree is one of the Table B-Trees with a defined payload format (section 1.5.9). This B-Tree payload contains pointer / page number to all the other pages in the file. It is referred to as the ”sqlite_master” table. Without parsing it properly you will only be able to access the ”sqlite_master” table. An example of why can be seen in figure 1.6. The Table 1.9 below shows the layout of the ”sqlite_master” table.

Field type	Field	Description
Text	Type	The type of link: ’table’, ’index’, ’view’, or ’trigger’
Text	Name	Name of the object / table, including constraints
Text	Table Name	Table Name
Integer	Root page	Root page number of this item
Text	SQL	Sql command used to create this object.

Table 1.9: Schema table layout

Much like the right child pointer mentioned in section 1.5.5 this is the page number of the child not a pointer. Following it will take you to it’s root page.

1.5.11 Parsing the file

In order to parse the file, First thing would be to read in the header (section 1.5.2) then read in the root page(s) (section 1.5.10), and follow the page numbers to all the other table root pages, then start parsing them until all paths have been followed. This format learns towards recursion rather than iteration although both are possible.

2 Design

In this section I go over the high level overview of my tools design. Starting with the high level, and going into more depth looking at each module.

2.1 High level Overview

The five main..

2.2 Module Overview

The first module..

3 Implementation

3.1 The tools

I used..

3.2 The Modules

3.2.1 Database parser

The Database parser...

3.2.2 Log

The Log...

3.2.3 Live Updater

The Live updater...

4 System Operation

5 Testing

5.1 Code Tests

5.1.1 Unit tests

Unit testing...

5.1.2 Integration tests

Integration tests...

6 Evaluation

6.1 System Performance

The system was...

6.2 Design principles

I followed..

7 Conclusion

8 References

Sotomayor B. (2010), The xdb File Format. On line publication, University of Chicago, http://people.cs.uchicago.edu/~borja/chidb/chidb_fileformat.pdf. Last Accessed 18th January 2016.

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.

Laysakura (2012), Visualize SQLite database fragmentation, On Line Publication, https://github.com/laysakura/SQLiteDbVisualizer. Last Accessed 24th January 2016.

Piacentini M. (2003) DB Browser for SQLite, On Line Publication, http://sqlitebrowser.org/. Last Accessed 24th January 2016.

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.

Drinkwater R. (2011) An analysis of the record structure within SQLite databases, Forensics from the sausage factory, On line publication, http://forensicsfromthesausagefactory.blogspot.com/2011/05/analysis-of-record-structure-within.html, Last Accessed 17th January 2016.

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

9 Appendix

9.1 SQlite File format

9.1.1 The SQLite header layout

Table 9.1 show the header layout. All multibyte fields are stored in a big-endian 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.
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.

Byte Offset	Byte Size	Description
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 9.1: Sqlite Header, modified from Hipp (2000)