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. Then 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 I only came across two different types. The user interface. They would add a user interface to SQLite, removing the need for using SQL and the command line directly. 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). It is 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 to 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 was very powerful and did exactly what it said on the tin. However, it did not allow an 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 98 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, and de-fragmentation.

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.

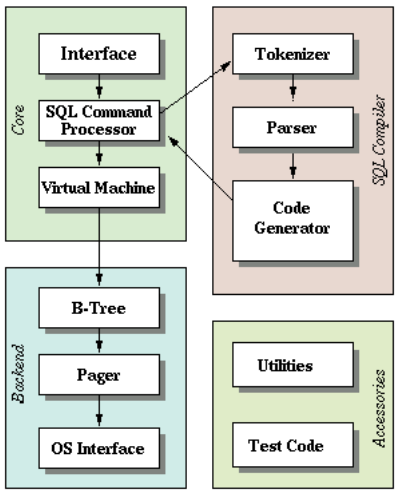

Figure 1.1: Sqlite architectural diagram (Hipp, 2000)

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. Appendix Table 8.1 shows the header layout. All multibyte fields are stored in a big-endian format.

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.

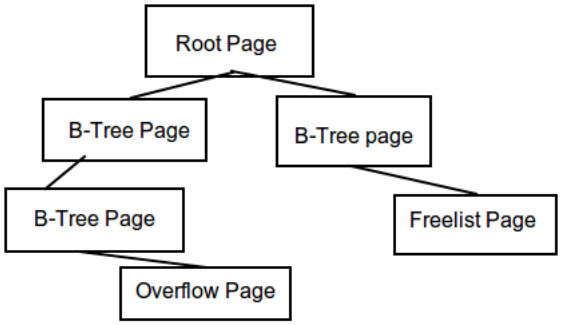

Figure 1.3: B-Tree page structure.

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 only the B-Tree pages.

At the start of each B-Tree page there is the B-Tree / page header. Following the header is a array of pointers to their cells. One thing to note is that the first page also has the database header. This mean you will have to skip it before reaching the page header. Figure 1.4 shows the full layout of the SQLite file.

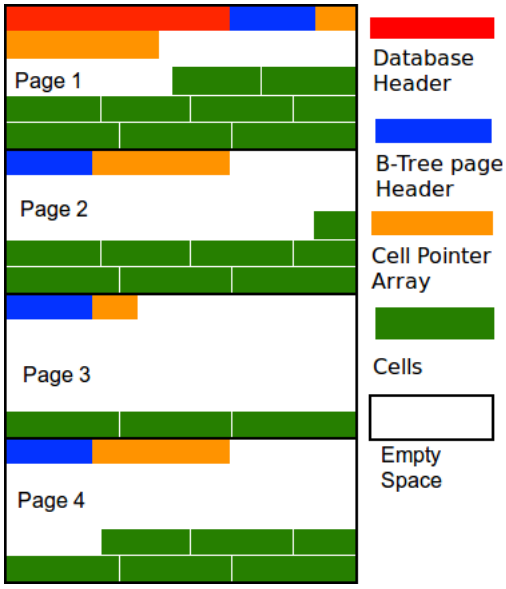

Figure 1.4: Sqlite file format, modified from Drinkwater (2011)

When the cells are added to the page, they start at the end of the page and work backwards towards the top. The main difference between each type of B-tree pages are found inside the cells as they carry the payload for the node.

As mentioned in section 1.5.1 there are four types of B-Trees. These can be split into two main types, and two sub types. The main types are 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, theses are often what the user has set inside the database, else SQLite will attach its own system. 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.

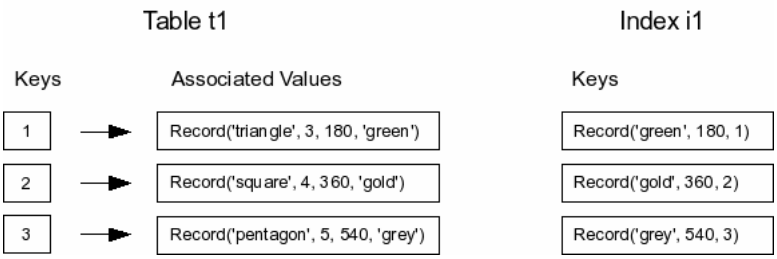

Figure 1.5: Example key pair database (Raymond, 2009)

Those sub 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.

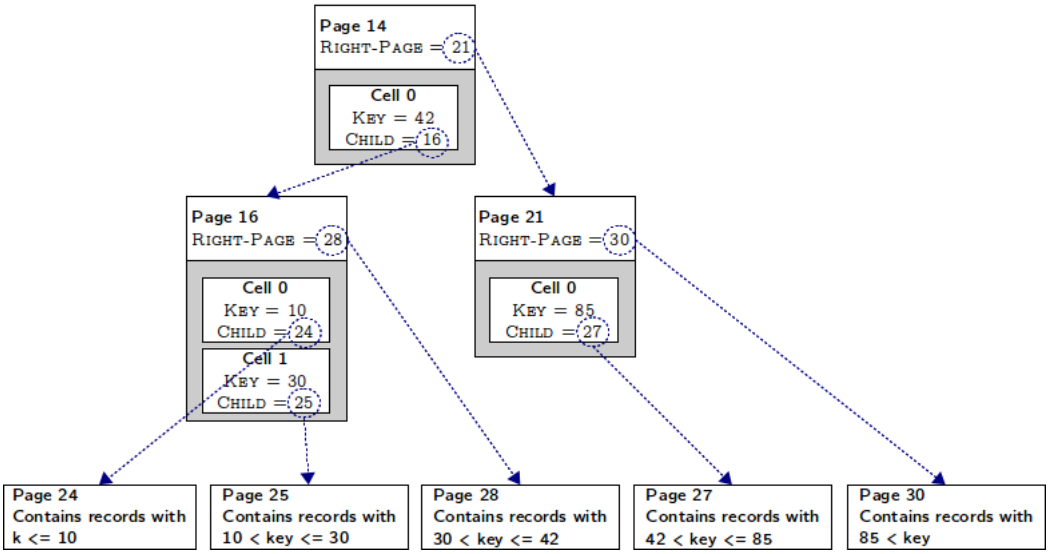

Figure 1.6: Physical layout of table B-Trees (Sotomayor, 2010)

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 1.1 show the all combinations of varints.

Bytes	**Value Range**	**Bit pattern**
1	7	0xxxxxxx
2	14	1xxxxxxx 0xxxxxxx
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 1.1: Varint combinations Raymond (2009)

1.5.5 B-Tree header

As mentioned in section 1.5.3 each page begins with a header. Table 1.2 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.

Byte Offset	Byte Size	Description
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 1.2: 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:

```
1    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:

```
1    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 1.3 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 1.3: 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 1.4:

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

Table 1.4: 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 1.5 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 1.5: 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. 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 1.6 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 1.6: 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.

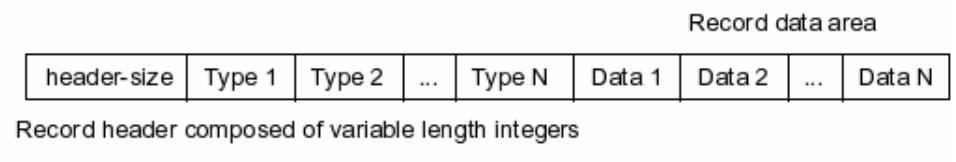

Figure 1.7: Database record format (Raymond, 2009)

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. And the following N the type of record.

As varints can be between 1 - 9 bytes it is important to keep track of the varints size. In order to count the number of values in the record. For example if we have N values, and the number of bytes as 8. This could mean there are 8 small values, or 1 large value, or anywhere in between. The different value types can be seem below in Table 1.7

Header Value	Byte Size	Description
0	0	Null
1	1	1 byte signed integer
2	2	2 byte signed integer

Header Value	Byte Size	Description
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

Table 1.7: 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 1.7. Where the size is 0 there is no varint to read from the data section and should be skipped.

1.5.10 Root 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 numbers 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.

In some of my test databases, the root page was a Interior Table B-Tree, so going by page numbers in order to find the schema table is a bad idea. The Table 1.8 below shows the payload / record 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

Field type	Field	Description
Integer	Root page	Root page number of this item
Text	SQL	Sql command used to create this object.

Table 1.8: 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 applications design. Starting with the high level, and going into more depth looking at each module.

2.1 Features

As mentioned in the last section, I wanted a cross platform tool, that would enable anyone to view the internal structure of a SQLite database. With consistency and reliability. The application comes with five main features, all building upon one feature.

The central feature is the visualiser. The visualiser allows you to see the broken down page structure and hierarchy. So you can see how you SQLite database is laved out. On top of this you can click a node in order to see more information about. Such as data, page number e.t.c

In addition to the visualiser, there is also a meta data tab, that will allow you to view, the header information in the database. Alongside other statistics that come from parsing the database. Such as number of tables, primary keys and so on.

The last main feature ties into the base feature, that allows real time updating of this data when any command from any system modifies the database in some way shape or form. The live updating allows you to step through the time line of updates that have occurs while the application is running. You can also pause it if you want to inspect a certain state.

Whenever a update occurs, all changes that happened are recorded inside a log, and a ”snapshot” of the database is taken. This snapshot is then presented to use, through the visualiser, metadata and log tabs. Creating the time line that you can then browse.

Apart from just showing you data, you can also execute SQL onto the database, through the SQL executor and view the schema and tables currently inside the database.

2.2 High level Overview

From the start I wanted to build a system, that was modular and self-contained. However, as the project went on a few adjust had to my original design. When

I ran into some implementation problems discussed in the next chapter. The figure 2.1 shows the original design.

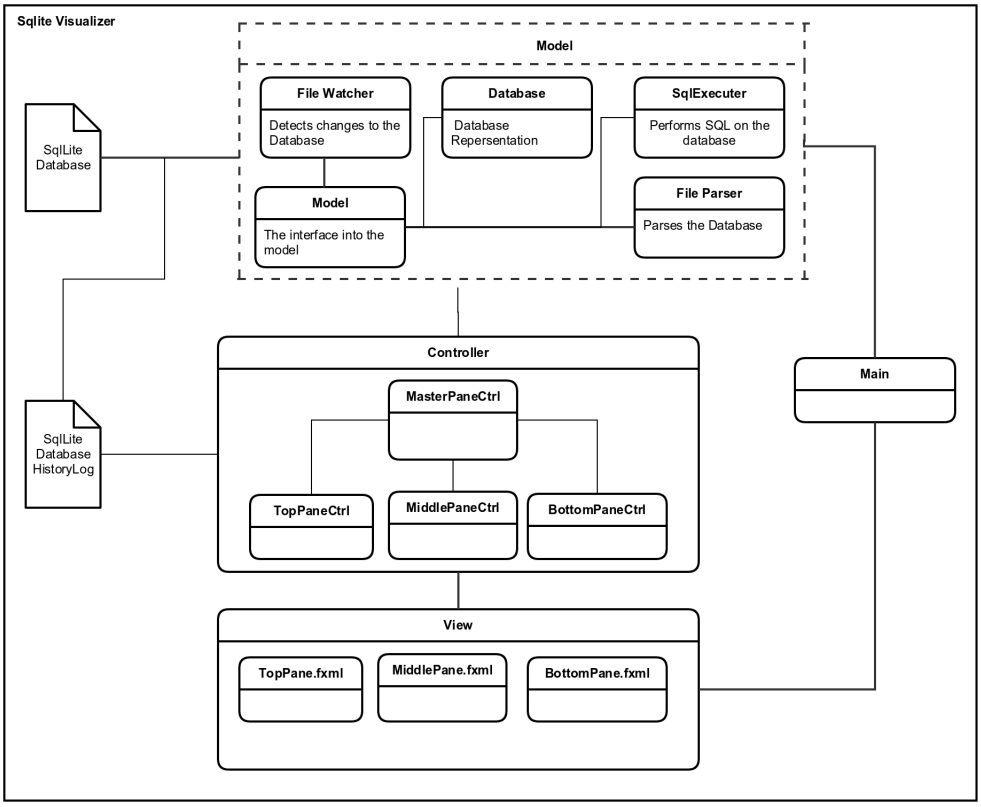

Figure 2.1: Original system diagram

As you can see, the design utilizes the MVC (Model-View-Controller) style architect in order to separate the interface from the data. This means the view will make requests to the controller, who will then in turn contact the model for information, then sending the information to be presented by the view. The idea being that the view could be switched or adjusted at anytime without breaking the application. an example of MVC can be seen below in figure 2.2.

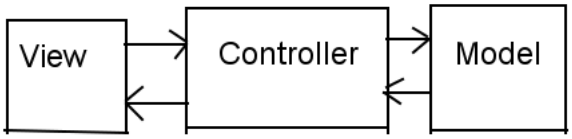

Figure 2.2: MVC architecture

The Model was going to run in its own thread so it could control, manage and prepare the data as it came in. This meant the view could request it when it wanted. One other thing to note, is the command logging was going to store them in to an external file, for the view. However, this design proved unusable and thus changed into the following seen in figure 2.3.

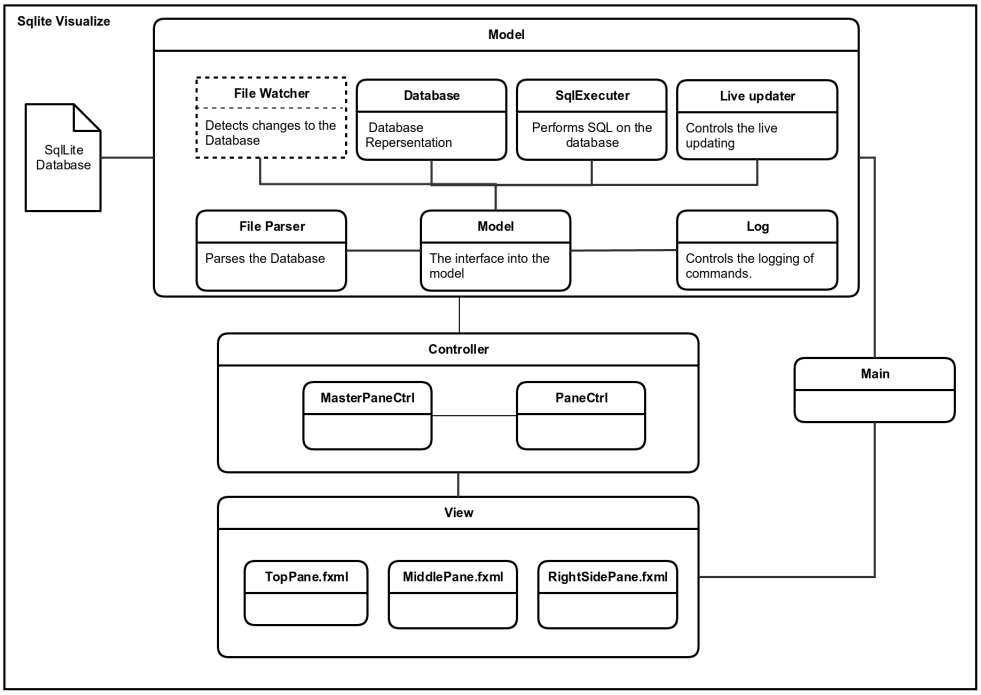

Figure 2.3: Final system diagram

Most of the changes are seen within the Model, with the addition of two new modules. And rather then running the whole thing inside a new thread only the file watcher is. On top of this the command logging is no longer written out to file. The final change is the reduction in the amount of controllers. I will go over each of the modules in the next part.

2.3 Module Overview

2.3.1 The Main

The main module represent the staring point of the application, it serves no other purpose other then to correctly initialise the view, model and controllers.

2.3.2 The View

The view, consists of three parts, the top, middle and right side. Since it follows the MVC style, these only contain the layout of each of the corresponding sections. The middle section however, changes depending on what is being viewed.

2.3.3 The Controller

The controller is made up of two parts, the master controller, and the pane controller. The pane controller is changed depending on what is being viewed. Similar to the views middle pane. The master controller coordinates what is being shown. Both of them communicate to the model to collect updates, and interact with the data.

2.3.4 The Model

The model is the most complicated section and is made up of seven modules. The model itself acts as a repository design with everything connecting and interacting though it.

All contact with the model will go though the model interface. This allows the controllers to communicate to the individual modules. In addition to this, it provides a small amount of functionality for setting up and closing. Such as opening the database. Since every module will require something from this action, the model interface will take care of setting up each module. And on exit making sure all threads have been stopped, and any other connections have been closed.

The Database, is the in program mapping of the SQLite database system. The database is made up of two parts, the data objects, and the interface into the data object. The data object are the mapping of the SQLite database. Containing the B-Tree system, and the data. The interface provides access to the history, allowing to program move along the database time line. And management of the data objects.

The file watcher, runs in its own thread, and utilises the observer pattern. It runs in a continuous loop waiting for a modification to happen on the database. Once a change is detect it sends a signal out over the observers, so the change is recorded. Although, if they did not tune in to the observer, the rest of the program would not receive database updates.

The File parser, parses any given valid database file. Converting it into the database object.

The log, takes any two database objects and records the changes between them.

The live updater, acts like a master controller for the modules, apart from the SQL executor, file watcher and model interface. It controls what happens when a change is detected, and as such is registered to the file watchers observer. When a change is detected, the first thing it does is contact the file parser for the updated database object, send it off to the log, to record changes, and then stores it in the database module. As it controls what happens and when it can also not update, allowing the parsing to pause.

The SQL executor, controls the SQL connections, and executes SQL commands onto the database.

2.4 The User interface

****
TODO: WHEN FINISHED!
****

3 Implementation

In this section I will go over the previously discussed modules, and how they are implemented. And why some thing turned out the way they did.

3.1 The tools

Before we look at the implementation, we must first pick our tools. For this particular application I when with Java. Using JavaFX for the user interface and controllers. This meant right from the get go I had cross platform support. A MVC style architecture and with JDBC open access to SQL database connections. The only major issue was speed, as SQLite is known to be fast, whether my application could keep up with the requests that where being performed. However, as SQLite only allows one writer at a time this so was never an issue. The other downside to using Java was not having direct access to the SQLite API through its own interface. But, after looking at the interface everything that i needed was supported through JDBC.

3.2 The Modules

3.2.1 The view and controller

To begin with we will look at the view, and controllers, as previously mentioned, the architecture is MVC. And we are using JavaFX. JavaFX comes with a whole host of tools for working with the view, and controllers.

Firstly, they have their own file type that is heavily based on HTML, with support for CSS styling. The file can include / imports Java classes into it, allowing for custom items. Each item can be given a unique id that allows it to be controlled with via Java.

On the controller side of things, once you are set as the controller for a particular fxml file. The annotation @FXML followed by the type, and unique id, allows Java to inject the item from the fxml file into the variable, giving you full control of it.

The view is made up of four sections, the menu bar, containing the file, edit and other drop downs, including the icons, and tabs. The other three section, represent the left, middle, and right sections of the central pane. This means that any one time i can display three different items. Each of the section have their own fxml

file, depending on the situation they may also share a controller. This can be seen below in figure 3.1.

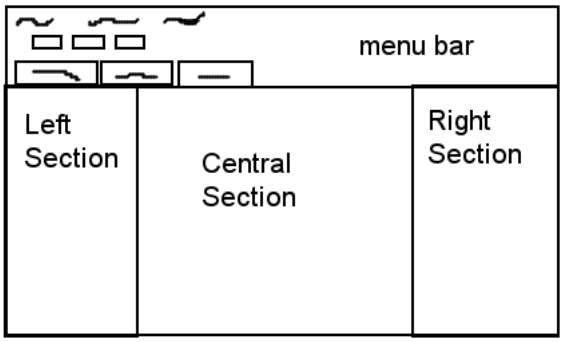

Figure 3.1: View breakdown

Looking at the breakdown, you will notice that the menu bar will never change. and should never change as its the navigation used to get around the program. Using this fact the menu bar controller also double up as a 'master' controller. By this I mean it controls what is currently seen in the other three sections. Loading and freeing up to load the necessary sections for that tab. As the central pane is simple split pane, allowing each sections size to be adjusted to fit the users needs. If a section is not needed its just a matter of hiding that panes split bar.

The controllers for each of the sections extend a abstract controller class. The controller class, enforces a model interface object into the constructor. And implements Observer. The model interface allows each controller to separately contact the model, as previously mentioned to collect the data for the view. And by implementing observer we can register our controllers for the signal when the database is updated. Meaning we can collect the updated information as soon as it's ready. Without having to wait, or having a manual refresh button.

3.2.2 Model interface

The model interface is how the controllers contact the other sub modules. Its a repository design, that helps keep everything de-coupled, with the exception of the Live updater. All modules including the model interface implement their representing interface. Allowing the implementation to change while keeping the same external view. This enables the design to be adapted to other systems other then SQLite. Below figure 3.2 show the layout of the model interface.

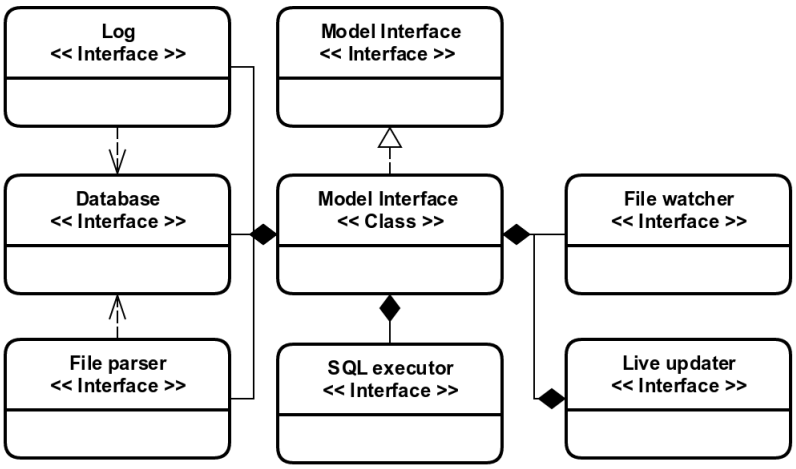

Figure 3.2: Model interface

As you can see everything is attached to the model interface, and only the live updater is the exception with a copy of the model. In addition to provide access to the other modules, it has a very small amount of implementation, that is only used when every module is affected. Such as the case of setting up, closing and opening a database, which are all calls to the corresponding method on the modules interfaces.

3.2.3 The database

The database module is made up of two parts, the interface and the storage. The interface is what everything else has access to allowing it to control the storage without interference. The storage is made up of database items, that contain a ”snapshot” of the current state of the database file.

The interface allows adding of new database items, retrieving of database items and stepping through the time line of database objects. And complete clearing of database items, used with setting up and opening a new database, in order to make sure that the items are not mixed up with the different files.

The database items in the storage, are a data structure, made of two parts the meta-data, and B-Tree. The meta-data contains all the information in the header, including a few others such as number of tables, file name, and more. The B-Tree is a custom implementation, holding the various B-Tree pages, as represented in the file. The database items are filled with information via the file parser. Out of all the classes, the database items, are the most used, being sent to the view for displaying, and modified by the various other modules during creation.

3.2.4 File watcher

In order to workout when the database was updated, to collect live data. I had two options, use SQLites API or watch the file. The API provided by SQLite is on a per connection base meaning that I would only revive signals when my own application sent of SQL commands, which are of no use to me, as I already know when the commands a sent out. This left me with the latter option, watching the file. Since SQLite is a single file, every time the file / database was updated so would the last modified time. In addition to this as SQLite only allows one writer at a time, I would revive a signal every update consistently.

The original implementation utilised Java's WatchService api. However, when I used it, I found it to be hit or miss whether it would register the change. And at one point failed to detect any changes. So i ended up rolling my own solution, which is a simple while true loop, recorded the last modified time, then when the time differs we send out a single to the rest of the application.

Due to the polling nature of this module, it runs inside its own thread, and communicates over an observer patten, that any another class can tune into, providing they implement the Observer interface. This meant that the thread could process the updated databases without stopping or slowing down the user interface and other interactions.

3.2.5 File parser

The file parser does exactly what it says on the tin, it takes a database file, a database object, and converts the file into the object. Parsing the database, started off with the, checking the magic number, then the header, before moving onto the pages. The magic number and header information where all about reading the first 100 bytes, correctly. For the pages I relied heavily upon recursion.

First I would parse the page header, then switch into the method, that dealt with that type of page, who would then call the original method, when it reached a page number. Each page was represented as a different node. As the recursive method returned a node, representing that node, and all of its children, left a elegant and effective design, unless we run out of stack with a huge database. Below is the psudocode of the algorithm:

```
1 public void parseBTree(stream, database) {
2     database.getBTree().setRoot(parsePage(stream, 1,
3                                 database.getPageSize()));
4 }
```

```

public Node parsePage(stream, pageNumber, pageSize) {
    Node node = new Node();
    PageHeader header = parseHeader(stream, pageNumber, pageSize);

    BTreeCell cell;
    switch(header.getType()) {
        case (TABLE_BTREE_LEAF_CELL) {
            cell = parseTableBtreeLeafCell(stream, pageNumber,
                                           pageSize);
        }
        ....
    }
    if (header.getType() == INTERIOR_CELL) {
        node.addChild(parsePage(in, pageHeader.getRightMostPointer(),
                      pageSize));
    }
    node.setData(cell);
    return node;
}

public cell parseTableBtreeLeafCell(InputStream, PageHeader, Node) {
    Cell cell = new Cell();

    int cellPointers[] = header.getCellPointers();
    foreach(cellpointer) {
        cell.data = readData();
        if (cell has pageNumber) {
            node.addChild(parsePage(in, pagenumber,
                      pageSize));
        }
    }

    return cell;
}
```

While that solves how to go about parsing the tree, one major problem was decoding the 'varints' mentioned in section 1.5.4 especially as we needed to count the number of bytes for the record headers. Below shows the psudocode algorithm that I made in order to decrypt them:

```
private long[] decodeVarint(stream) {
    long[] value = new long[];
    byte[] varint = new varint[9];
```

```
4
5       for (i = 0 to 9) {
6           varint[i] = stream.readByte();
7           if (first bit is not set) {
8               break;
9           }
10      }
11
12      if (i == 0) {
13          value[0] = 0;
14          value[1] = 1;
15      } else {
16          for (j == 0 to i) {
17              varint[j] = (varint[j] << 1);
18          }
19          value[0] = varint.toLong();
20          value[1] = i + 1;
21      }
22      return value;
23  }
```

The first value returned is the value of the varint, and the second its size. Now we have that all sorted decoding the record headers, and other section of the file is easy.

3.2.6 The log

The log, as briefly mentioned in the design section, had a complete change compared to the original plan. The original plan was to have the log run on its own without having to relay on other modules, and would retrieve the original SQL commands that where sent to it. While looking at SQLite there were three ways I could have done it in addition to my final implementation.

The first technique I looked at utilised SQLites triggers. Triggers execute SQL commands when, a Delete, insert or update is performed on a table, with a optional where clause. Using this Chirico (2004) used three separate triggers to long the time, changes before and after, and type of action performed on the table. The last part is one of the reason why I went for another technique. Firstly, I would need to have three triggers per table in the database, so N*3 triggers where N is the number of tables. Secondly, in order to accomplish this, I needed my own table that it changes where the changes are stored to, hence the log file in my original design, where i would attach to the database and write to it. lastly, the triggers

mean altering the database file, this is something I wanted to avoid as much as possible.

The second solution, was to try and hook into SQLite through its API more specifically the sqlite3_trace function. You pass it a callback function, that is called with the SQL commands, at various stages as it passes through the system. Unfortunately for me at the current time the JBDC for SQLite did not support the function that I needed. So I ended up writing a couple C functions that I could then call from Java in order to access the functions. It worked for the most part, apart from that method only calls the callback function for SQL sent from the current application. Which is useless to me as I wanted to see all the changes.

So with two ways down the third way was to write my own extension to SQLite, or download the source code, and modify to suit my needs it. The seemed to be way to far from the original path, and if I used a custom version it means that it would be limited to only my version of SQLite. And as mentioned previously, I wanted to not modify the data if possible, so writing an extension, that would have to be loaded into SQLite and attached to the database, possibly conflicting with any other extensions they might have left me with my final option.

The final option, while less sophisticated then the others, works well, although I do not get the original requests. I do end up recording the time, and all changes that happened per command. Since the database storage contains all of the previous versions like a snapshot of the database. when a update comes in I simply compare the new updated database to the last database that passed through the application.

In order to compare database though required looping through every data value in both trees and comparing them, not only the data vale, but also the added pages and removal of pages. This could not be detected through any of the other techniques. Clearly looping through every single item in a larger database would quickly become a bottleneck, and slow the application and parsing down. So in order to speed it up, I did two things, firstly hashed the data array, If the hashes matched then we do not have to loop through the data. the second this was to adjusted my B-trees into a modified version of the Merkle Tree patented by Merkle (1988). The basic idea behind the merkle tree is that each node in the tree has a hash of its childrens hash, all the way down to the leaf node, who hash is based on the contents. Below figure 3.3 shows a digram of the merkle tree.

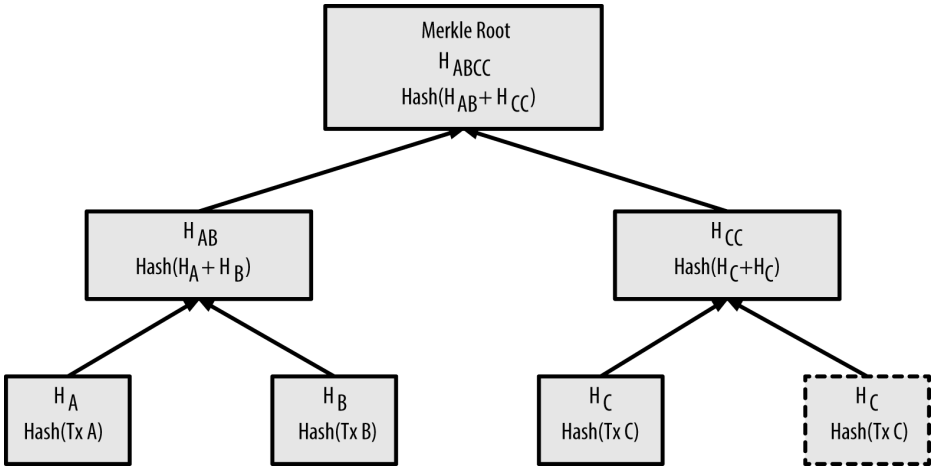

Figure 3.3: Node hashes in a merkle tree (Antonopoulos, 2014)

This means we can tell if there is any change in the current section tree just by comparing the nodes hashes without having to loop over them. meaning that we will only loop over the tree at all when a change is seen. otherwise we can skip it. Creating an unnoticeable change unless many nodes are changed at once. The hash for node is calculated off the hash of the data and number of children meaning we will also know if a page has been added or removed.

In addition to this, when it detects an update to the data, it will mark that page as modified, a simple boolean value. and record the store the string value, from the old page and the new page. If instead it was a removal or addition, of data, it will simply store the value, and leave the removed or non-existent value. Something similar happens with added and removed pages. With the addition of pages, the newer page will be marked as changed. Often when this happens a pointer to the new page will also be placed somewhere, so this is also recorded. When a page is removed, utilising the data from the old tree we can see exactly what was gone and record it, but there is nothing to marked as changed, apart from the pointers in other pages to that page.

3.2.7 Live Updater

The Live updater has undergone many changes from the start, although its position in the architecture has not changed, it gradually was morphed and shaped by the rest of the application. Originally it stared out as a relatively simple class that would control the parsing of the application. By contacting the database and telling it when to move along the time line, allowing the pausing of live updates. It would also receive the update signal from the file watcher, and parse

the file, moving it into the database storage. Basically an extra more controlled, and tailored interface into the database interface linking it to the file parser.

Then as time went on I wanted to have additional information into the metadata of the database, tat required running SQL onto the database, else i would have to loop through the entire tree again. So it needed access to the SQL execute, which we will cover in the next section. Then I ran into problems with the log feature, and it was all to easy to stick the log module inside this one. However, this class soon became bloated, as I added new features, since this was the only one that had access to all the needed resources.

Rather then stick with it I decided rather then fighting against my designs it would be better to dedicate this as a master module, that would orchestrate the process when an update signal is revived. This allowed me to move the bloat back into their correct modules. And now it does exactly what I said in the beginning, acts as a tailored interface into the database interface, contacts the file parser, SQL executor, and Log modules to control the parsing of the updated database.

3.2.8 SQL executor

The SQL executor manages the JDBC connections to the database, making sure that it connects, closes and commits any changes that are needed onto the database. It interface provides four methods to the other modules, connect, close, perform select, perform update and get database meta-data. Unlike th rest of the modules this was one of the more straight forward and simple to implement.

3.3 User Interface

I mentioned in the last section how the view is put together, but now we have looked at each of the modules is turn. I wanted to go through how each of the different tabs / features are put together. In order to better understand how each module is interacted with.

****
TODO WHEN THE UI IS FINISHED
****

4 System Operation

5 Testing

Testing is an important part of making all application, and as such will go over the ways that I went about testing my application.

5.1 Test data

In order to assure that my program ran correctly under a variety of circumstances, I used a variety of databases including different sizes, and headers.

5.2 Unit tests

Throughout the implementation stage I kept close to test driven development, and as such have written a lot of unit tests utilising the JGroups framework. In total I have around TODO:X unit tests. All passing. The unit tests are written to use mocks where dependences are needed allowing me to make sure its correctly testing.

In order to test the user interface I found a test framework that works alongside called testFx that is specifically designed to test JavaFX. You pass it the root node of your scene, and pass it commands, such as click, with either the id, or name of the item. It will then automatically control the mouse, interacting with the user interface.

5.3 Integration tests

In addition to unit testing, I performed integration tests that would test the interactions between the various modules in order to check that they are working correctly. Such as the live updater and it corresponding calls to the other modules. This also included testing that the user interface would correctly interact with the module and its various modules correctly.

6 Evaluation

6.1 System Performance

The system was... Speed

6.2 Design principles

I followed..

7 Conclusion

References

Antonopoulos A. (2014), Mastering Bitcoin: Unlocking Digital Cryptocurrencies, Book, and Online publication, O'Reilly Media, http://chimera.labs.oreilly.com/books/1234000001802/index.html. Last Accessed 29th January 2016.

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.

Chirico M. (2004) SQLite Tutorial, On line publication, http://souptonuts.sourceforge.net/readme_sqlite_tutorial.html. Last Accessed 29th January 2016.

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

Merkle R. (1988). "A Digital Signature Based on a Conventional Encryption Function". Advances in Cryptology CRYPTO '87. Lecture Notes in Computer Science 293. p. 369.

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.

8 Appendix

8.1 SQlite File format

8.1.1 The SQLite header layout

Table 8.1 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 8.1: Sqlite Header, modified from Hipp (2000)