SQLite Visualiser: Building a tool to visualise the SQLite file format and log all changes.

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

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted version of this submitted work, I consent to this being stored electronically and copied for assessment purposes, including the Schools use of plagiarism detection systems in order to check the integrity of assessed work. I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Date:

Signed:

To get the code and other documents visit the website at:
http://www.lancaster.ac.uk/ug/battyp/

Abstract

This paper presents a tool designed to visualise the internal workings of a SQLite database file. Starting with the history of SQLite, its systems and file format. Finding the file is a series of fixed sized chunks / pages, and each page is a node in a much larger B-Tree structure. Secondly, constructing a model-view-controller style application that can then parse, and present this data in real time, while other systems access the file. Thirdly, how using TestFX and JUnit have helped build a robust application. Lastly, how the tool could be improved by polishing up the user interface, with customisation and other minor interactions. The overall system could be improved with increased performance and adding some much needed features. On top of this, a future project could look at the changes made to the database and turn them back into the original SQL queries.

Keywords: SQLite, databases, SQL, Java, JUnit, TestFx, JavaFX, B-Trees, Merkle trees, varints, Real time updates, User interface design

Introduction

SQLite is a small lightweight database engine that can perform many operations without the need to configure, manipulate, or go through a long winded install process. It is simple flexible, and widely distributed. 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 embedded OS's) and in the most unexpected places such as aircraft.

SQLite's systems and infrastructure, enable it to be flexible, fast and simple. The main focus of this paper however, was on the file format that it uses to store the entire database. How it was put together. How to traverse it. And why it is the way it is. In addition to this, the available tools for SQLite. This is covered in the first section of this paper.

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

While building the tool. It was important to see how it operated from 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. This is covered in sections four.

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

As this paper covers similar programs, there is a distinct shortage of tools that enable users to debug their database, while there is an abundance of user interfaces. Excluding the hex editor. Alongside this they often do not provide any inside into how SQLite is updating the database. Lastly, there is currently no way of logging commands that are executed onto the database, outside of your own connections.

To combat this 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.

1 Background

The following chapter will cover two completely different programs that operate on SQLite. Then look towards SQLite's beginnings, and where it is used. Starting back in spring of 2000. Finishing off with a look at the SQLite file format. Before moving onto the other sections.

1.1 Similar programs

While searching the web for tools only two different types appeared. The graphical interface. They would provide a graphical user interface into 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 completely outnumbered the technical tools, where only one could be found.

1.1.1 SQLite browser

The first tool, SQLite browser, made by Piacentini (2003) based off of the Arca database browser. Out of all the user interface tools out there 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. Below figure 1.1 shows a screen shot of the program.

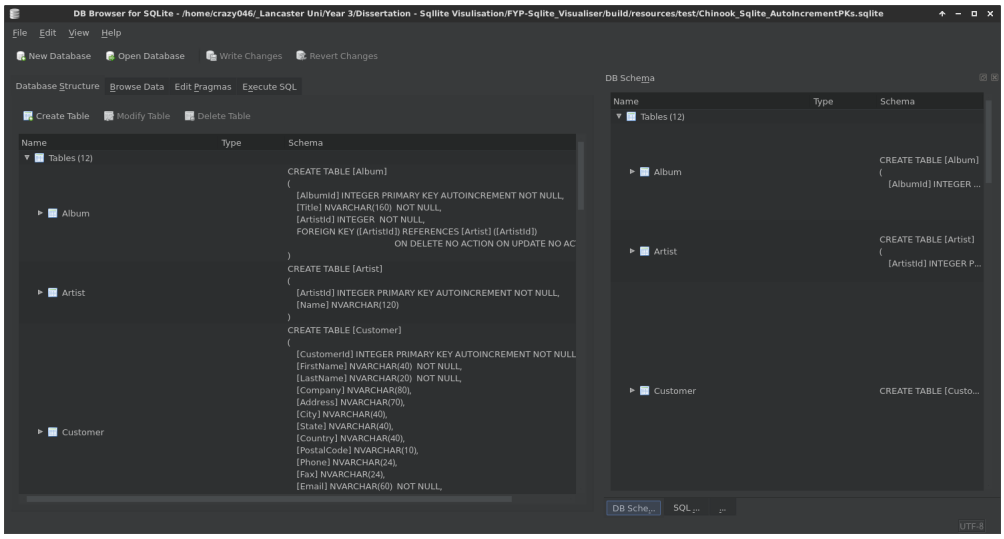

Figure 1.1: Screen shot of the SQLite database browser.

Apart from the usual features, such as viewing tables, schemas, and modifying them. The more unique features allowed exporting the tables to CSV (comma separated values), 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. The user interface and features built into the tool where simple and intuitive to navigate. It was very powerful and can be very usefully to anyone working with SQLite that does not want to use the command line. 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 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 mentioned later later on, the file format 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. See figure 1.2 for a screen shot.

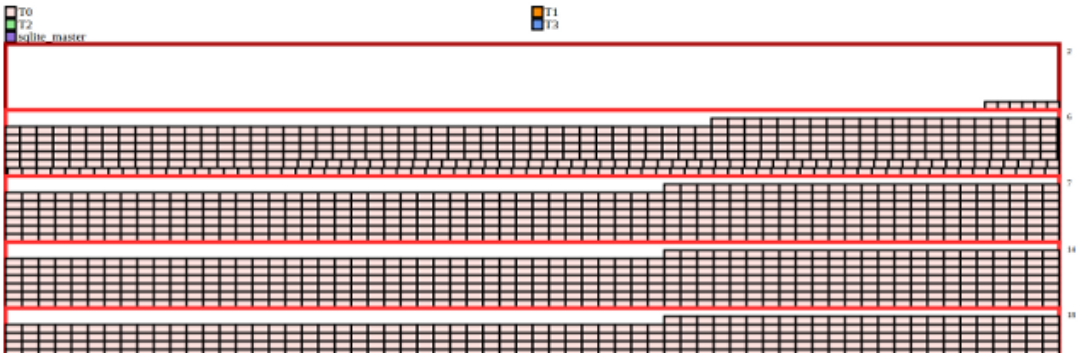

Figure 1.2: Screen shot of the SQLite fragmentation visualiser. (laysakura, 2012)

The tool runs 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 output to be added in. Some of the more advanced features allow 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. WAL mode, or write ahead logging, is an alternative to a rollback journal, where the changes are written to the log. This log is then inserted into the database file at each checkpoint during the transaction. The rollback journal takes the opposite approach and changes the database with the journal holding the backup. Overall, this tool provides a very useful insight into SQLite. On top of this, it clearly shows 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 he used the database system Informix. Which 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 than such a large, all powerful system. (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 improvements over 2.0, with the first public release on 18 June 2004. At the time of writing this paper the current version of SQLite is 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 than a database system. In fact Hipp (2000) 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 to be truly used for anything. Hipp has stated numerous times that SQLite might be the single most deployed software currently. Alongside zlib, libpng and libjpg. With the numbers in the billions and billions (Hipp, 2000).

This large distribution means 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. SQLite can be found 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 1.3 shows the architectural diagram of SQLite.

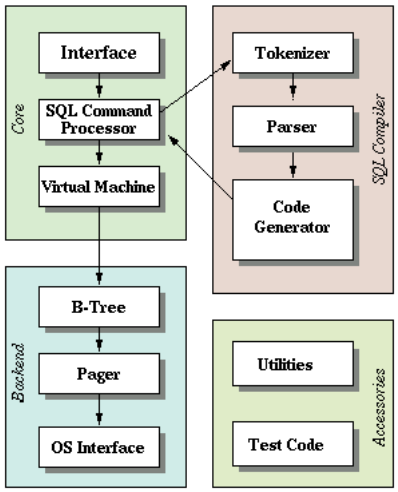

Figure 1.3: Sqlite architectural diagram (Hipp, 2000)

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 an assembly language program. The program is written in a assembly language that is specifically designed for SQLite. It is run by the virtual machine inside the core module. Once the assembly program is constructed it is sent 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 1.4 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 1.4: 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.

With this in mind the virtual machine takes the SQL input from the Interface, passes it onto the SQL Compiler. Then collecting the resulting program from the code generator, and executing this program to perform the original request that was sent in. Making it the heart or core of SQLite's operations.

1.4.3 The Backend

The final main module 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 what the page contains. 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 the needed information.

The OS Interface is a warehouse, providing a constant interface to 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 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 exists for testing purposes, of which contains over 811 times more code then the actual project and millions of test cases. As it covers every possible code path through SQLite. This is partly why SQLite is considered to be so reliable.

1.5 The SQLite file format

1.5.1 The page system

As mentioned in section 1.4.3 the B-Tree module looks after the pages including the organisation and relationships between them. Packing 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) The ubiquitous B-Tree paper. SQLite also takes some improvements seen in Knuth (1973) Sorting and searching 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 or 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 an 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 too large.
- Pointer map page
  Pointer map pages are inserted to make the vacuum modes faster. And is the reverse B-Tree going child to parent rather than 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 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 is covered in the next section. Appendix A shows the full header layout.

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 five types of pages. However, each page is linked together in a B-Tree format, where each page represents a node in the tree.

SQLite uses a both B+-Trees and B-Trees, throughout its systems. The B-Tree is a data structure built for systems that read and write large amounts of data blocks, such as file systems. In this case however, the pages representing are representing the data blocks.

B-Trees are made up of internal nodes, and leaves. Internal nodes make up the structure of the tree, connecting each part of the tree down to the leaves, like branches. The leaves are located at the end of the structure. The B-Tree stores data in both the internal nodes and the leaves, whereas the B+-Tree only stores data in the leaves. An example B-Tree can be seen below in figure 1.5.

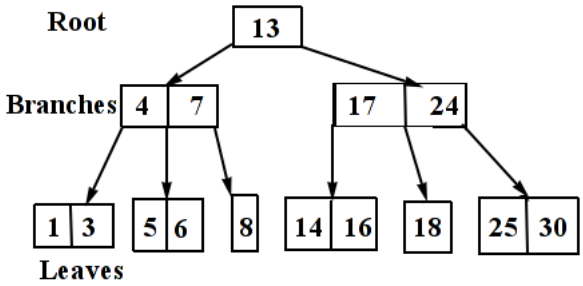

Figure 1.5: B-Tree structure.

If the above figure, is a B-Tree all of the items would be a valid value, and represent a key. However, in a B+-Tree, only the items in the leaves are valid values, and the rest act as keys, to guide a path to the correct value.

When searching the tree for a value of *N*, the value of the current node is taken. If the value *N* is smaller than the value of the current node, then the path to the left node is taken. If it is larger then the path to the right. Inside a B+-Tree, since values are in leafs only this is extended to be greater than or equal to.

Insertion and deletion follows the same pattern, upholding the ordering of the values. SQLite uses page type and relationships to determine to order of which the B-Trees nodes are connected, for example an overflow page can only be connected to a B-Tree cell, and the data for a specific table can be found attached to a common parent.

Below figure 1.6 shows an example B-Tree that could be found within SQLite.

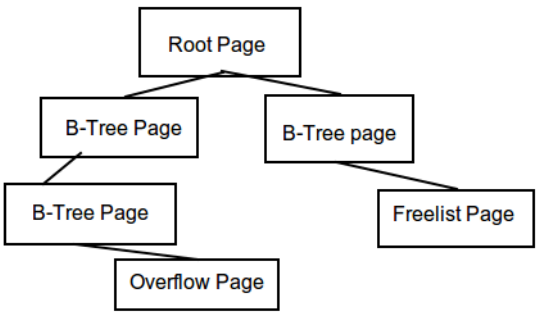

Figure 1.6: B-Tree page structure.

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

At the start of each B-Tree page there is the B-Tree / page header. Following the header is an array of pointers to their cells. One thing to note is that the first page also has the database header. That will have to be skipped before reaching the page header. Figure 1.7 shows the full layout of the SQLite file.

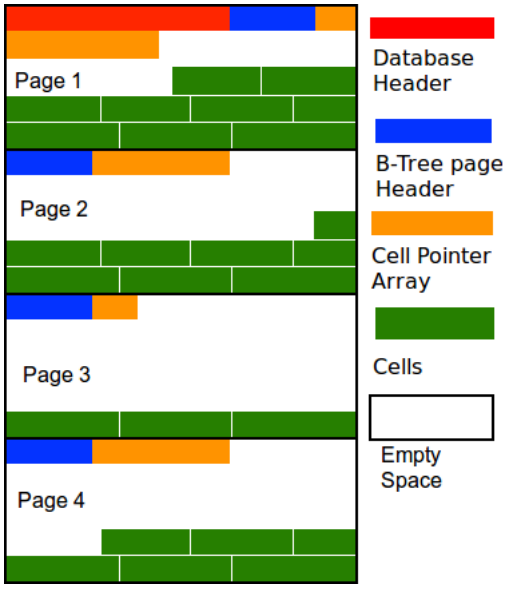

Figure 1.7: Sqlite file format, modified from Drinkwater (2011)

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 page can be seen 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. The Index B-Trees use database records as keys.

The sub types of B-Trees are broken down into Leaf and Interior. The leaves 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 types 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 contain *N* children and *N-1* 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 an Index B-Tree are of the same quantity as the associated database table, with the same fields and columns, between the tables and rows. This can be seen in figure 1.8. Index trees are used by SQLite to keep track the foreign keys and row relationships.

```
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.

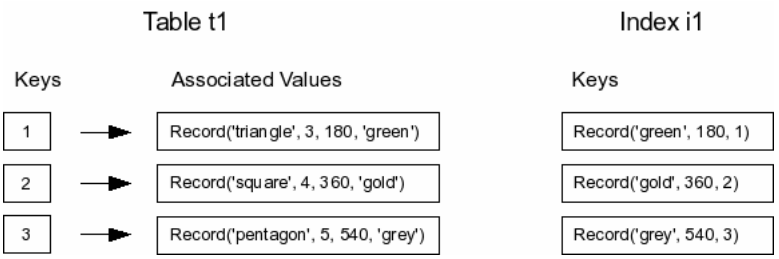

Figure 1.8: Example key pair database (Raymond, 2009)

The table B-Trees are completely different to the index trees. The interior type contains no data but only pointers to child B-Tree pages, as all the data is stored

on the leaf type. 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 1.9 shows the physical layout of the Table B-Tree.

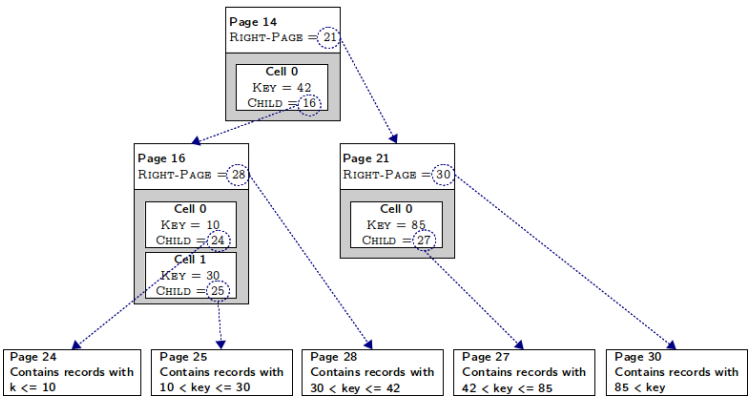

Figure 1.9: Physical layout of table B-Trees (Sotomayor, 2010)

1.5.4 Encoding of the data

SQLite uses a variable length integer or 'varint' in order to 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. Taking up 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 it is the ninth byte where all the bits are used. If the most significant bit is set then the next byte is needed. So if it is set in byte 1 then byte 2 is needed and so on until, either the bit is not set, or the 9th byte is reached.

For example, taking the following value in hex 5B and converting this to binary, creates the value 01011011, in this case the most significant bit is not set leaving the final value at 91. However, if the value in hex is 84 and converting this to binary creates the value 10000100, the most significant bit is set, therefore the next byte is needed. In this case the value of the next byte in hex is 60, converting this to binary creates 01100000 meaning that this varint is two byes long. In order to create the final value, the bytes need to be concatenated together leaving out the most significant bit. Creating the value 00001001100000 giving a final value of 608 in decimal (Drinkwater, 2011). Table 1.1 shows 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 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 1.2: 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 than 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 its 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-Trees use the database records as keys, their content also reflects this. Table 1.3 shows 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

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

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 too large.

Table 1.5: Page B-Tree leaf cell

1.5.8 Detecting an Overflow Page

In order to detect an overflow page, both types follow a very similar algorithm. The following algorithm is used in Index B-Tree cells for detecting an overflow page:

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

if (payload-size > max-local) {
    there is an overflow page.
}
```

Where bytes-of-unused-space is read in the Sqlite header at offset 20 and max-embedded-fraction at offset 12. The only difference in the Table B-Tree cell is the calculation of the max-local, using the following sum:

```
max-local := usable-size - 35;
```

If there is an overflow the following calculation can be used to work out the size of the record in this part of the Index B-Tree cell before jumping 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 - 12) * max-embedded-fraction / 255 - 23;

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

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

Similarity for the Table B-Tree the only difference is in the calculation of the max local where the following sum is used instead:

```
max-local := usable-size - 35
```

1.5.9 Overflow Page

Overflow pages as mentioned in section 1.5.1 are used to store the payload when it is too large to fit in a single cell. Overflow pages form a linked list with the first four bytes pointing to the next page number in the chain or zero if it is the last. Following the four bytes through to the last bytes is the payload content. Table 1.6 shows 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

1.5.10 Payload / Record format / Byte array

Throughout the previous sections the payload has been called 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 1.10.

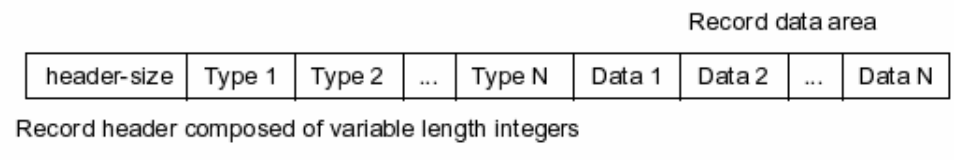

Figure 1.10: Database record format (Raymond, 2009)

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 record types.

As varints can be between 1 - 9 bytes it is important to keep track of the varints size. To count the number of values in the record. For example, if there are 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 seen 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
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 textit(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 1.10 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.

1.5.11 Root B-Tree and Schema Table

As mentioned in section 1.5.2 following the SQLite header is the root B-Tree. The root B-Tree is one of the Table B-Trees with a defined payload format (section 1.5.10). 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 only the ”sqlite_master” table will be accessed.

In some of the test databases, the root page was an 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
Integer	Root page	Root page number of this item
Text	SQL	Sql command used to create this object.

Table 1.8: Schema table layout

In order to tell if the current page is a table, the first column, should contain one of the four types, mentioned in the above table. Then read the page number in the fourth column to find the content for that table. Much like the right child pointer mentioned in section 1.5.5 this is the page number of the child not a pointer.

1.5.12 Parsing the file

In order to parse the database file. The first thing would be to read in the header (section 1.5.2) then read in the root page(s) (section 1.5.11), and follow the page numbers to all the other table root pages, then start parsing them until all paths have been followed. This format leans towards recursion rather than iteration although both are possible.

2 Design

Having looked at SQLite and the current tools. This chapter will cover the requirements set out for the tool, the features and a high level overview of the architectural design. Then going into more depth looking at each module that makes up the application. Finishing off looking at how the user interface could look.

2.1 Requirements

In the previous chapter this paper showed that there is a lack of tools available that allow an insight into SQLite and how it works. The exception to this was the SQLite fragmentation tool, that did show the file format though not the over arching structure. It also had some severe limitations as as to the types of databases that it would work with.

In addition to this SQLite currently provides no way to log changes performed to the database outside of the currently running program. This is implemented by SQLite itself, to get around this triggers and other such convoluted systems have to be used.

It also showed that the file system is put together, constructing a large B-Tree structure. As just mentioned there is no way to view this structure without using a hex editor, and manually following the links.

Lastly, most of the SQLite tools are font end user interfaces for SQLite databases. While this is useful for working with SQLite database they provide no way to debug or see what is going on.

In order to address the issues listed above, throughout the remaining sections this paper will design, implement and test a front end user interface tool, that can solve the current situation.

The tool itself must be reliable and support the majority of features in the current version of SQLite at the time of writing this paper. It must also not modify the database file in any way to preserve the database, and the data it contains, excluding parts of the application that are specifically designed to. Lastly, the tool must be cross platform and intractable through a user interface.

2.2 Features

As just mentioned, there are a lot of issues surrounding the current situation with SQLite tools. The tool this paper will design and implement contains five main features, on top of a single base feature.

The central feature is the visualiser. The visualiser allows the visualisation of the broken down page structure and hierarchy of the SQLite database file. Viewing the file broken into pages, and how they connect to each other in the B-Tree structure. On top of this clicking or otherwise interacting with a node to see more information about. Such as data, page number and pointers.

In addition to the visualiser, there is will be a metadata tab that will present the header information in the database. Alongside other statistics that come from parsing the database. Such as number of tables, primary keys and version numbers.

The base feature 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 will allow stepping through the time line of updates that have occurred while the application is running. Lastly, it can be paused to inspect a certain state / place in time.

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 the user, through the visualiser, metadata and log tabs. Creating the time line that can then be browsed.

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

Finally, the tool will come with a friendly user interface, through which the user can interact with each of the features. Each feature above aim to address the issues mentioned at the start of this chapter.

2.3 High level Overview

To begin with the main aim is to have a system that was modular, self-contained and adaptable. In order to accomplish this the application is built using a MVC (Model-View-Controller) architecture 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 backwards to be presented by the view. The idea being that the view could be switched or adjusted at any time without breaking the application. an example of MVC can be seen below in figure 2.1.

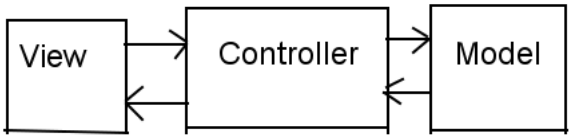

Figure 2.1: MVC architecture

With this in mind the bulk of the work is performed inside the model. However, like any project, along the way some problems occured while building the application, and so the original design had to be adjusted. Figure 2.2 shows the original design.

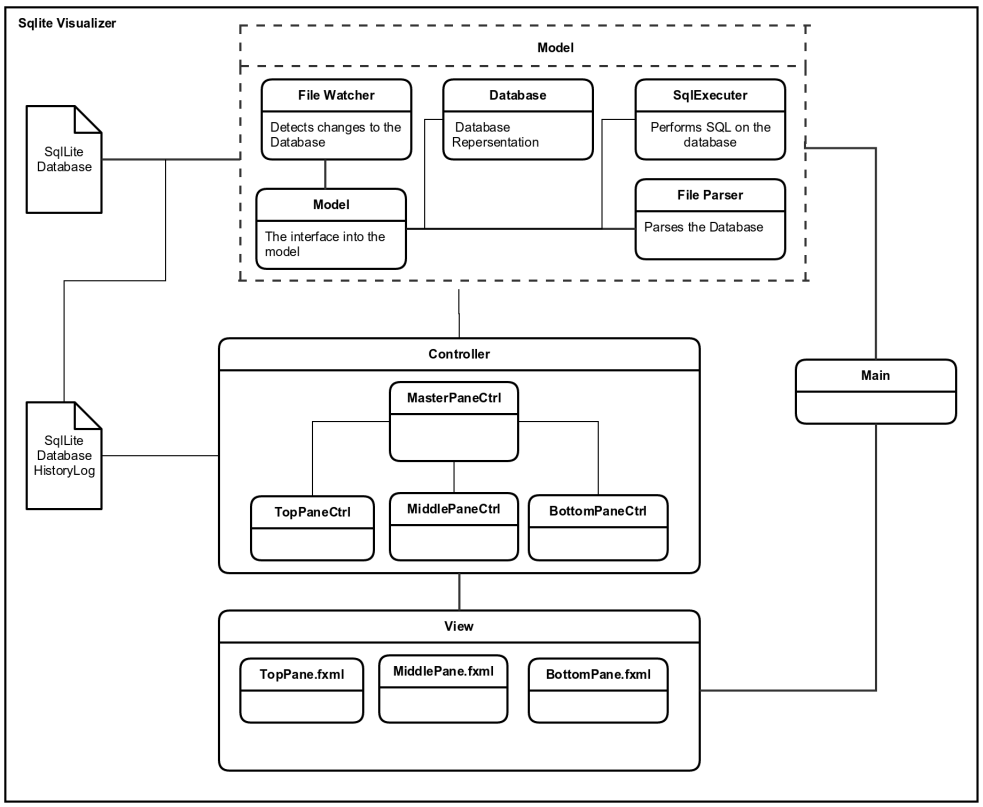

Figure 2.2: Original system diagram

The Model was going to run in its own thread so it could control, manage and prepare the data as it came in. This would allow the view could request it when it wanted. The model was also made up of five modules. With the logging stored into an external file, for the view to read when it wanted. In addition to this the controllers follow a hierarchical structure, with a master controller, controlling access to the model. However, this design proved unusable and thus changed into the following seen in figure 2.3.

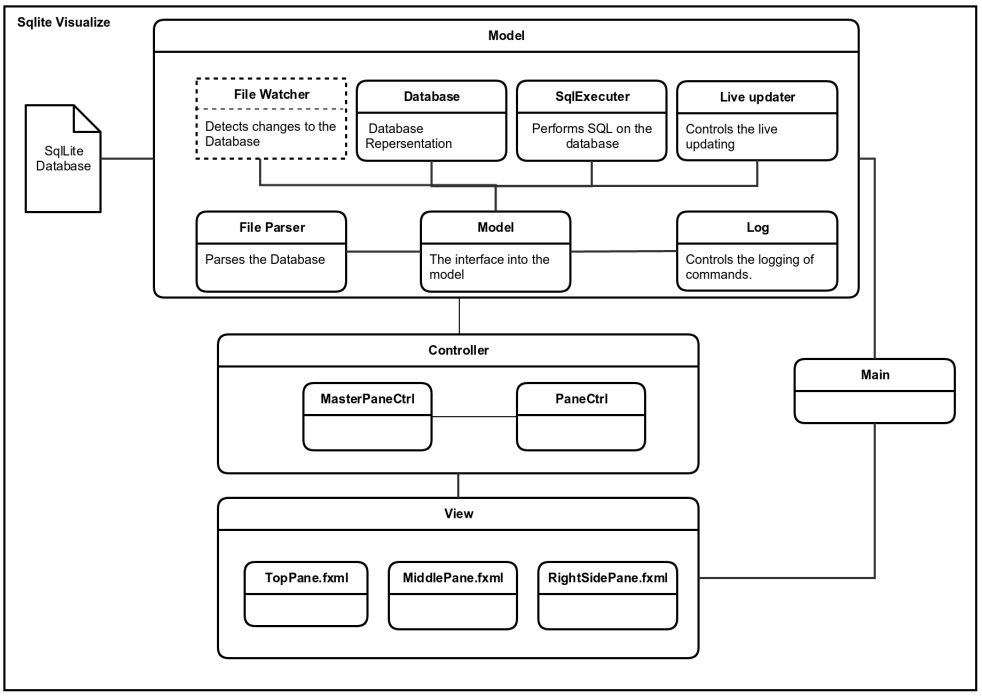

Figure 2.3: Final system diagram

Most of the changes are seen within the Model, with the addition of two new modules, and rather than 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. The next section will go over each of the modules in depth.

2.4 Module Overview

2.4.1 The Main

The main module represents the starting point of the application, it serves no other purpose other than to correctly initialise the view, model and controllers.

2.4.2 The View

The view consists of three parts, the top, middle and right side. This is the most flexible of the modules and can be adjusted depending on the user interface design. The view, is designed to hold the layout of the each section and provide interaction to the model, as mentioned previously. However, in this case, the view is made up of three parts, the top representing the top / menu bar section of the interface. The middle and right side sections to show the user what they are viewing.

2.4.3 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, respectively to the views middle and right pane. The master controller however stays through the running of the program and coordinates what is being shown. Both of them communicate to the model to collect updates, and interact with the data. The master controller allows the view to have some state and as such allows the different parts of the view to interact with each other.

2.4.4 The Model

The model is the most complicated section and is made up of seven modules. One for each of the main tasks to keep each one self contained. The model itself acts as a repository design with all the modules connecting and interacting though a shared object the model interface.

The first module, marked as ‘model’ represents the model interface. In which all communication with sub modules, from the controllers will go through. It is also the only class to have direct access to all sub modules, in order to keep each module as modular and independent as possible. In addition to this, it will provides a small amount of functionality such as setting up, closing and opening the database. Since every module will require something from this action.

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

The file watcher, runs inside its own thread, utilising the observer pattern. The observer patten allows any class to register to it, and will revive a signal when an event occurs. In this case it would be the updating of the database file. When a

change is detected, a signal will be sent out over the observers, alerting them to the change. Although, if they did not tune in to the observer, 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 the program flow when a change is detected, and as such is registered to the file watchers observer. When an update signal is sent, the first thing it does is contact the file parser for the updated database object, then sends it off to the log, to record changes, before storing it in the database module and incrementing the current position on the time line.

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

2.5 The User interface

The user interface is designed to be simplistic and easy to use and familiar to new users. The basic format is a menu bar at the top, with icons underneath. Just below them is a selection of tabs corresponding to the different views, that the application can take. Down the right hand side, however will be another static section representing the SQl executer. This can be seen below in figure 2.5.

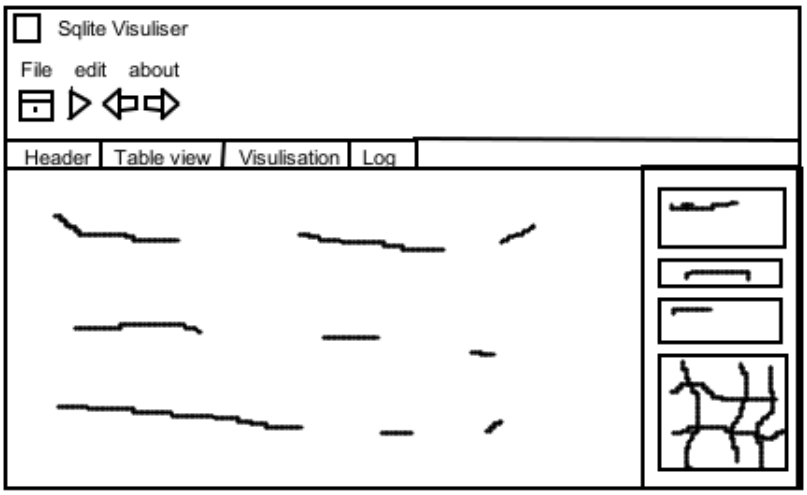

Figure 2.4: User interface design.

Originally the SQl executor was going to have its own tab. But it worked out better along side the content as you can then view what the commands are doing to the database, when they are ran. Or have the information as a reference when typing up commands, Below figure 1.1 shows the SQL executer User interface design.

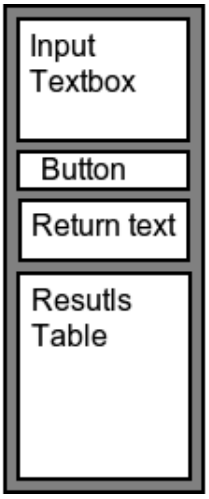

Figure 2.5: SQL executor design.

The user will type commands into the top text box, and then press the button to run the command. After the command is ran a message will be displayed in the return text box. And any returned data will be presented in the results table.

The base live updater is controlled via the icons and drop down menus with the

four other features having their own tab. The metadata tab, is designed to have different panes, split up in to data relevance, each showcasing the different values. For example one pane will show the size, another the version numbers and so on. This can be seen below in figure 2.6.

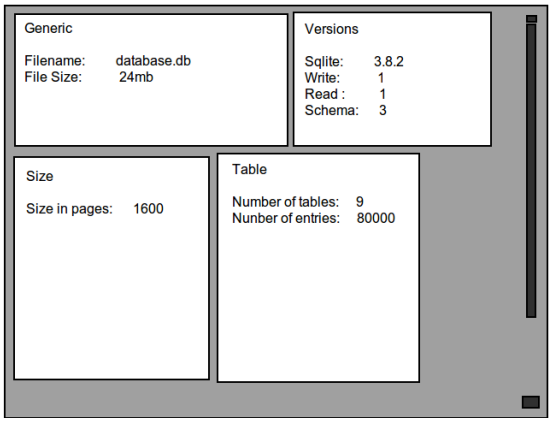

Figure 2.6: Metadata interface design.

The other feature that is closely tied to this information is the visualiser. The visualiser is made up of two parts, the left hand side where the data will be shown, and the central section showing the visualisation. The visualisation will display the file B-Tree as it is inside the file, with the pages represented as nodes. Then when a node is selected the data will be shown inside the data pane. This can be seen below in figure 2.7.

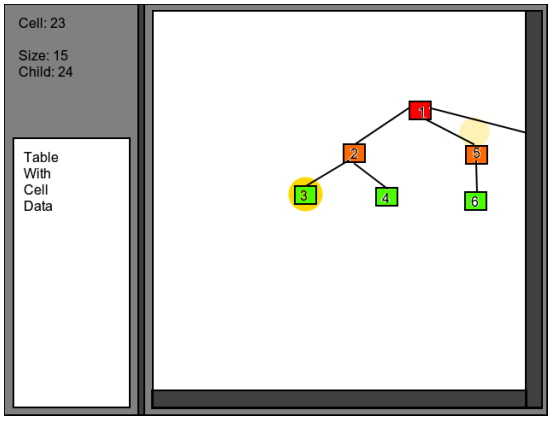

Figure 2.7: Visualiser interface design.

In the above figure, the glowing node tells the user that this node was updated in the last update. Along side this a log entry is created. The log tab much like

the metadata tab will be made up off small panes, titled with the date, and the collapsible content. The content will store what changed in that update this can be seen in figure 2.8

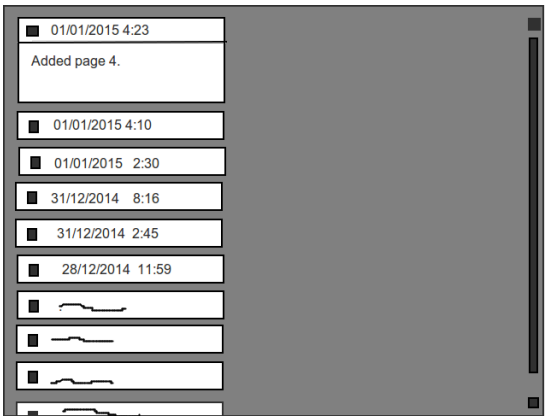

Figure 2.8: Log interface design.

The idea behind the folding panels, is to allow users to hide information that they do not need. As such if there is a large change it would not be filling the screen. The last feature and tab is the table and schema view. This, much like the visualiser uses two sections. The left, to display the list of tables and the schema. With the centre displaying the table data. This can be seen in figure 2.9.

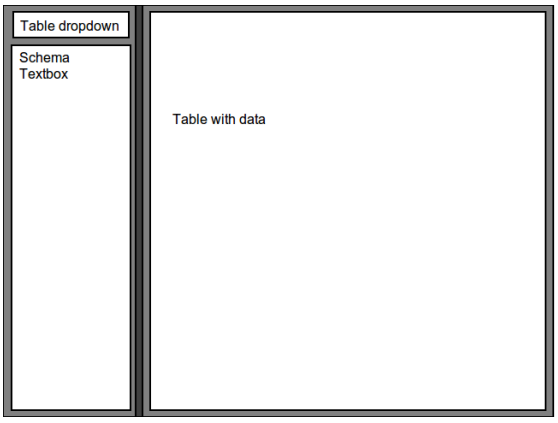

Figure 2.9: Table interface design.

3 Implementation

Having looked at the overall architecture of the application, and how it is build up. This section will go over the tools used during development, how the features were implemented and the problems encountered along the way.

3.1 The tools

For this particular application the programming language of choice was Java and JavaFX for the user interface and controllers. This meant right from the start the application 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 the application could keep up with the requests that where being performed. However, as SQLite only allows one writer at a time this 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 was needed is supported through JDBC.

3.2 The Modules

3.2.1 The view and controllers

As previously mentioned, the architecture is MVC with JavaFX and Java. JavaFX comes with a whole host of tools for working with the view, and controllers.

Firstly, each view or section can be represented using an FXML file. The FXML file is heavily based on HTML, including the support for CSS styling. Each file starts with a root node, normally one of the panes, such as border, anchor, and grid. For this application the majority of FXML files started with anchor panes, apart from the menu bar which used a border pane. Then following the root pane is the items to attached to it. Each item can be given a unique identifier that allows it to be controlled with via Java and CSS. In addition to this custom items can be included for use within JavaFX.

Secondly, the controller is a normal Java class set as the controller for a particular FXML file. This can be done in two ways. The first way is to inject the controller into the loading process. This allows the application to call other methods in side the controller, such as initialisation before the FXML file is loaded. It also gives more control over the controller, and where it is used. The second way is to specify the controller inside the FXML file, and Java will load the controller in when the file is loaded. But the access to the controller object is lost. Inside

the controller the annotation @FXML, allows Java to inject the item from the controller into the view, and vice versa. Giving full control over the FXML file and the object utilising the items unique identifier. Throughout this application the first method of loading the controllers were used, to allow controller sharing, and manual control over the controller.

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 three different views can be displayed. Each of the sections have their own FXML file, depending on the situation they may also share a controller. This can be seen below in figure 3.1.

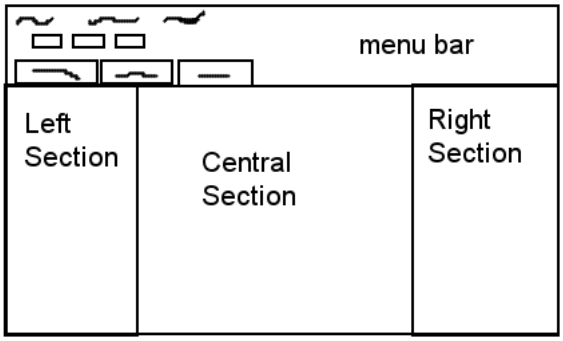

Figure 3.1: View breakdown

As mentioned earlier that the menu bar will never change, and should never change. Using this fact the menu bar controller also doubles up as a ‘master’ controller. It controls what is currently seen in the other three sections. Loading and freeing up the necessary sections for that tab.

The central pane is a split pane, with two split dividers allowing each sections size to be adjusted on the fly to fit the user’s needs. If a section is not needed it is just a matter of hiding that panes divider bar.

The controllers for each section 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. By implementing observer the controller can then be registered for the signal when the database is updated within the live updater (section 2.4.4). Meaning that the updated information can be collected as soon as it is ready, without having to wait, or

having a manual refresh button.

3.2.2 Model interface

The model interface is based on a repository design with all the sub modules attached to it. In order for the controller to communicate with the sub modules they must also first go through the model interface. This design helps keep everything separated and compact. All of the sub modules attached to the model interface implement their corresponding interface. Allowing the implementation to change while keeping the same external view. This enables the design to be adapted to other systems other than SQLite. Below figure 3.2 shows the layout of the model interface.

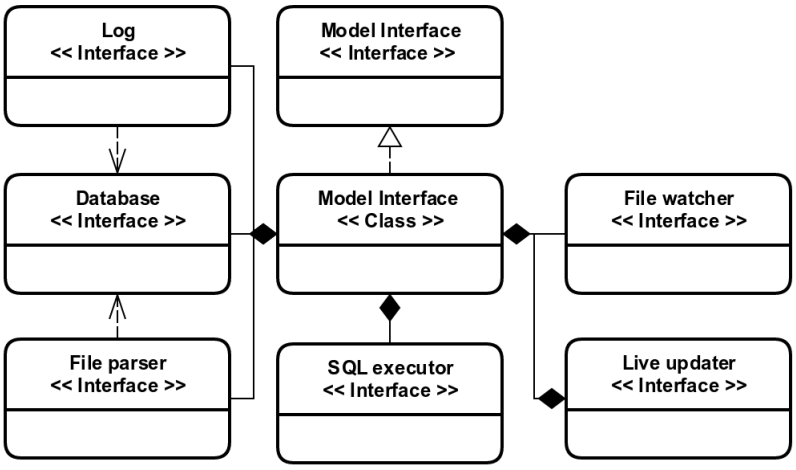

Figure 3.2: Model interface

Everything is attached to the model interface, apart from the live updater being the exception with a copy of the model. In addition to providing 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 controls access to the storage. Allowing the 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 different

files. The items are stored within an ArrayList, and an int counter is used to keep track of where in the array the application currently is.

The storage is made up of database items that contain a "snapshot" of the current state of the database file, the database item is 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 the number of tables, the file name, and page number. 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

There were two different options available in order to detect when the database was updated. Firstly, using SQLite's API provided by SQLite to detect the changes. However, the API works on a per connection basis, meaning that the signals would only be sent along the same connection that the request came from, which are of no use, since the application knows when a command is sent. Alternatively, the application could watch the file for modifications. Since SQLite is a single file, every time the file / database is updated the last modified time would also change. In addition to this as SQLite only allows one writer at a time, this would allow the application to revive a signal every update consistently.

The original implementation utilised Java's WatchService API. However, during use it tended to be hit or miss whether it would register the change. At one point it failed to detect any changes. In the end a custom solution had to be made, using a simple while true loop, recording the last modified time, then when the time differs it sends 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 pattern, that any another class can tune into, providing they implement the Observer interface. The thread could then process the updated databases without stopping or slowing down the user interface and other interactions.

The final implementation, effectively detected all changes performed to the database consistently. Since it was custom made it was also a lot lighter than the other alternatives. In addition to this the loop had no performance hit on the computation or the rest of the application.

3.2.5 File parser

The file parser takes a database file, a database object, and converts the database file into the database object. Parsing the database, starts with checking the magic number, then the header, before moving onto the pages. The magic number and header information is correctly reading the first 100 bytes of the file, See Appendix A for the header layout. To parse the pages the application heavily relied upon recursion.

First it 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 node, with contents of the node represented as a cell. The only main issue with this design is the size of the stack on a large 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  }
5
6  public Node parsePage(stream, pageNumber, pageSize) {
7      Node node = new Node();
8      PageHeader header = parseHeader(stream, pageNumber, pageSize);
9
10     BTreeCell cell;
11     switch(header.getType()) {
12         case (TABLE_BTREE_LEAF_CELL) {
13             cell = parseTableBtreeLeafCell(stream, pageNumber,
14                                            pageSize);
15         }
16         ....
17     }
18     if (header.getType() == INTERIOR_CELL) {
19         node.addChild(parsePage(in, pageHeader.getRightMostPointer(),
20                         pageSize));
21     }
22     node.setData(cell);
23     return node;
24 }
25
26 public cell parseTableBtreeLeafCell(InputStream, PageHeader, Node) {
27     Cell cell = new Cell();
28
```

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

    return cell;
}
```

During the process of parsing the tree the file parser will also need to decode the 'varints' mentioned in section 1.5.4 especially as they are needed to count the number of bytes for the record headers. Below shows the psudocode algorithm that decrypts them, to both retrieve the encrypted number and the number of bytes used:

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

    for (i = 0 to 9) {
        varint[i] = stream.readByte();
        if (first bit is not set) {
            break;
        }
    }

    if (i == 0) {
        value[0] = 0;
        value[1] = 1;
    } else {
        for (j == 0 to i) {
            varint[j] = (varint[j] << 1);
        }
        value[0] = varint.toLong();
        value[1] = i + 1;
    }
    return value;
}
```

The first value returned in the array is the value of the varint, and the second its size. The only issue with this algorithm is the use of the two for loops, increasing

the time complexity of the algorithm. However, during operation this number was always less then nine so was never a major problem.

3.2.6 The log

The log, throughout the project has under gone many design changes. The original plan was to have the log run on its own without having to rely on other modules, and would retrieve the original SQL commands that were sent to the database. However, as mentioned later this proved unattainable, leaving the only option to record the changes that occur when a command is sent. While working on the log there were three ways that this could have been implemented in addition to the final method.

The first technique utilised SQLites triggers. Triggers execute SQL commands when, a delete, insert or update is performed on a table, with an optional where clause. Using this Chirico (2004) used three separate triggers to log the time, changes before and after, and type of action performed on the table. The last part is one of the reasons why this could not be taken any further. Firstly, the application 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, an additional table would have to be created that the changes are stored to, hence the log file in the original design, where the application would attach to the database and write to it. Lastly, the triggers meant altering the database file, this is something needed to be avoided as much as possible, as to not impede on the running of the database.

The second solution, was to try and hook into SQLite through its API more specifically the sqlite3_trace function. It takes a callback function that is then called with the SQL commands, at various stages as it passes through the system. Unfortunately at the current time the JBDC for SQLite did not support the function that was needed. In order to get around this, a couple of functions had to be written in C that could then be called from Java in order to access the API. It worked for the most part, apart from that method only calls the callback function for SQL sent from the current connection. When the main aim of the application needed to collect all the changes, thus making this unusable.

The third way was to write a custom extension to SQLite, or download the source code and modify to suit the applications needs. The seemed to be way to far from the original path, and if the application used a custom version of SQLite it would only work on the custom version of SQLite. As mentioned previously, one of the goals is 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 is out of the question.

The final option, while less sophisticated than the others, it worked well enough, although it does not get the original requests. It does 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 it can simply compare the new updated database to the last database object that passed through the application.

To compare the database required looping through every data value in both trees and comparing them, not only the data values, but also the added and removed 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 two things had to be changed. Firstly, the data array had to be hashed, If the hashes match then there is no need to loop through the data. Secondly, is to adjust the cutom B-trees implementation into a modified version of the Merkle Tree 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's hash is based on the contents of the data. Below figure 3.3 shows a diagram of the merkle tree.

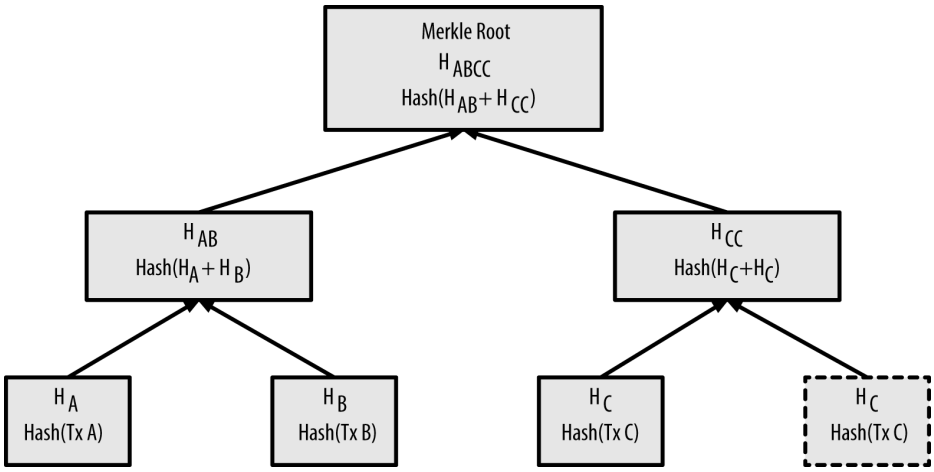

Figure 3.3: Node hashes in a merkle tree (Antonopoulos, 2014)

With this the application can tell if there is any change in the current section of the tree just by comparing the nodes hashes without having to loop over them, allowing the program to only loop over the nodes when a change is detected. Allowing it to skip the parts that have not been changed. Unless the entire database

is modified where the application will have to revert to looping over everything. The hash for each node is calculated off the hash of the data inside it, and its children's data. This will let the application detect if a page has been added, removed or modified.

When the application detects an update to the data, it will mark that page as modified, a simple boolean value. Recording the string value, from the old page and new page, into a log object. If instead it was a removal or addition, of data, it will store the new / removed value. Something similar happens with added and removed pages. With the addition of pages, the added 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 allows the application to see exactly what was page has been removed and can record it, but there is nothing to marked as changed, since it no longer exists, 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 the live updater started out as a module 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.

However, to collect metadata information about the database, the live updater had to run SQL onto the database, else it would have to loop through the entire tree again. So it needed access to the SQL executer, which is covered in the next section. Then the problems with the logging came up, and while working a way around these problem, it ended up inside of the live updater. As a result of this the module soon became bloated, as new features were added since this was the only module that had access to all the needed resources.

Rather than fighting against the application design, it would be better to dedicate this as a sort of master module, that would orchestrate the process when an update signal is revived. This allowed the extra tasks that where piled on to move on back into their correct modules, as a result of this it does exactly what was set out in the beginning. Acting as a tailored interface into the database interface, contacting the file parser, SQL executor, and Log modules to control the parsing of the updated

database.

When an update signal is received, the live updater requests a database object from the file parser, and adds the extra metadata from the SQL command. It then requests the previous database object from the database interface, and sends them to the log. Before storing the new database object inside the database interface. If the application is not paused it will then increment along the time line.

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. Its interface provides five methods to the other modules, connect, close, perform select, perform update and get database metadata. Unlike the rest of the modules this was one of the more straight forward and simple to implement, calling the corresponding functions on the JDBC API.

3.3 User Interface

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

The entire user interface is made up of eight FXML files and one CSS file, for styling. As mentioned previously, the FXML files contain the layout for each of the sections. Therefore each of the sections seen by the user represents a single FXML file. The application takes on a dark colour scheme used throughout, due to personal preference, however this could easily be swapped out for a lighter color scheme.

The menu bar is made up of one FXML file. In addition to the visual elements using the JavaFX key code combination class it provides keyboard short-cuts, such as ‘control o’ to open a database, and ‘control-q’ to quit. The controller for this FXML file contacts the model interface for the opening and closing the database, and the live updater for controlling the timeline. This can be seen below in figure 3.4.

Figure 3.4: Menu bar.

The right section is made up of a single FXML file, containing the SQL executer. As previously mentioned the SQL executor module enables arbitrary SQL commands to be ran on the database. As such this section contacts the SQL executor in order connect, run commands, and close the connection. The final interface can be seen below in figure 3.5.

Figure 3.5: SQL executor UI.

The metadata tab is made up of single FXML file. The outer layer is a scroll pane, with a flow pane content. The flow pane provides a dynamic layout to adjust to different resolutions alongside the scroll pane. The panels, inside of the flow pane are panes with grid pane content. The grid pane has two columns. The left or first column, representing the description or name and the right or second column the value. In order to collect the metadata, the controller needs to contact the database interface, to collect the current database object. The user interface can be seen below in figure 3.6.

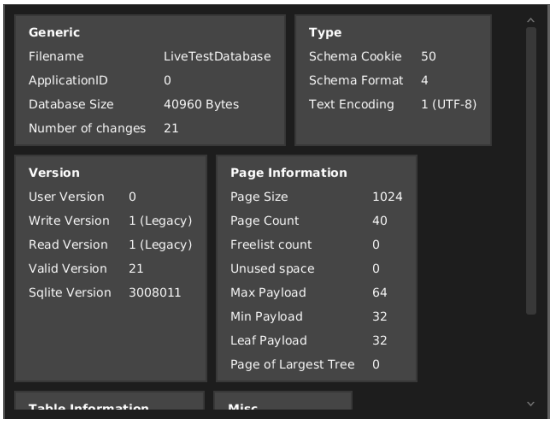

Figure 3.6: Metadata UI.

The table view is made up of two FXML files that share a controller. The table view allows users to select a table and view the schema and all the data within it. In order to accomplish this it contacts the SQL executor, and runs a select all query to collect the data, and a select from 'sqlite_master' to retrieve all tables and schemas. This interface can be seen below in figure 3.7.

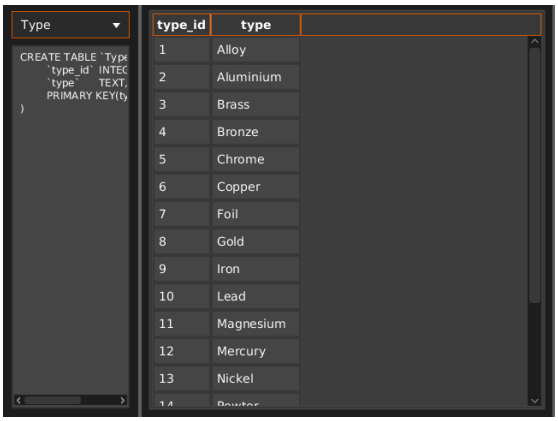

Figure 3.7: Table view UI.

The visualiser is made up of two FXML files, sharing a single controller. The centre section containing the visualisation of the database, uses a custom scroll pane node to enable, zooming in addition to scrolling. Each node is represented as a pane, with a CSS class for the colouring. The node, contains the corresponding B-Tree node, from the database object. This is then used to display the data within the left side pane. In order to draw the structure the first pass sorts out the horizontal position going top down. This is also used to load the nodes, via recursion. After all the nodes are loaded a second pass is used to then calculate

the horizontal positions using a bottom up approach. This interface can be seen below in figure 3.8.

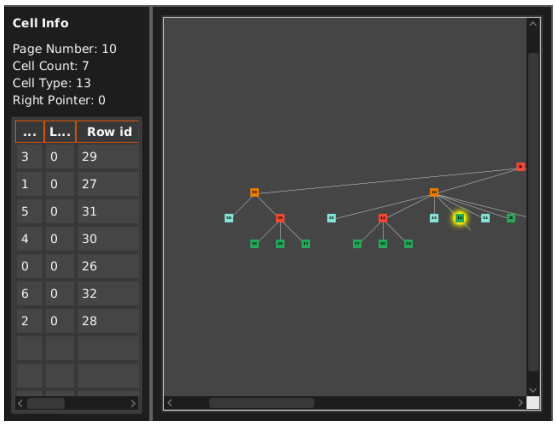

Figure 3.8: Visualiser interface design.

The log is made up of one FXML file. Similar to the metadata tab, it contains a single scroll pane, with a VBox inside, allowing it to contain infinite items. Each item is made up of a Titled pane. Where the title is the time and data of the update, and the content, the changes that where performed in the update. To collect the data, it contacts the Log module, and receives a list of log items. This can be seen below in figure 3.9.

Figure 3.9: Log UI.

4 Testing

As mentioned at the start of this paper, and during the design section. One of the aims is to make sure that the tool could be relied on. In order to accomplish this, a variety of testing methods and tools were used, this section will go over theses.

4.1 Test data

In order to assure that the program ran correctly under a variety of circumstances, a variety of different sized databases were used to test. The smallest made up of one table and page, and the largest twelve tables and 1066 pages. The smaller and medium sized databases were custom made for this project, while the larger one was taken from Rocha (2014) using real word data.

As stated at the start of this paper SQLite is intended to be used as a file format rather than a complete database system. Therefore testing the application on a multi-gigabyte file would be unnecessary. This can be assumed as any databases of that size would be better off in a different system (Hipp, 2000). Although where this is out of the developers control or there are no alternatives to their situation would be classed as an edge case.

4.2 Unit tests

Throughout the implementation stage the process kept close to test driven development, and as such a lot of unit tests have been written utilising the JUnit framework by Gamma E. and Beck K. (1997) . In total there are around 139 unit tests. All passing. The unit tests are written to use mocks where dependences are needed allowing the tests to make sure the application is working as intended.

The unit tests themselves, attempt to test all the possible action that could be performed to each exposed method in an attempt to make sure that each part of the program is operating as expected. Apart from the very few edge cases. However, some parts of the application are better tested then others due to the complex nature of some modules.

In order to test the user interface a test framework that works alongside JUnit called TestFX (Nilsson and Strath, 2012) was used, it is specifically designed to test JavaFX. It takes the root node of the scene to test. Then in the tests command methods are used such as click, move to and hover with either the identifier, or name of the item. It will then automatically control the mouse, interacting with the user interface.

TestFx works very well for testing the navigation methods around the user interface and making sure that the user can get from one part of the application to another. With that said for more complex and fine grained interaction such as the scroll pane within the visualiser, and the opening of titled panes in the log, were hard to test effectively, in such cases manual testing had to be used.

4.3 Integration tests

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

Integration tests where combined with the unit tests, For example when the TestFX tests clicks on the various buttons in the menu bar the signals are still sent to the model. This was used to effectively to make sure that the various elements with the program where working as expected.

4.4 Manual testing

Where such tests needed a human eye, such as the design and drawing of items, and other small interactions that could not be automated. Manually overseeing theses items and how they interacted had to be used. In addition to the testing of visual elements, other manual tests include how easy the interaction with the application, for example opening a file. This was used as in general the automated tests do not take into account how the application is displayed and interactive by an actual user.

5 Evaluation

Now that the paper is coming to and end, this section will reflect on the final results from this project, and whether we have met our overall aims. Including the design principles used through the undertaking of the project.

5.1 Design principles

While to project was going the overall design and structure keep as close to good OOP paradigms and designs as possible. As stated in the introduction the main aim was to build tool that anyone could use to debug their own databases, with the five main features visualisation, command logging, live updating, executing SQL and browsing data. Wrapped up within a user interface.

With support for other database system when needed, just by extending the interfaces and providing an implementation. In addition to sticking to good OPP design, TDD (Test driven development) was used in order to make sure that the final application, would work under a variety of circumstances. In the end theses aims where achieved. With the exception, of the lock byte and pointer map pages, as stated towards the start of the paper.

5.2 The System

The system architecture, works efficiently and follows a modular pattern making it adaptable to other systems, should the need rise. The modularisation, however, has caused some issues during development, as mentioned during the implementation section with the live updater. Having to move around, and adapt the modules to become disconnected from one and another. Doing so made it harder to implement some of the features. But, in the long run will end up creating a better and more robust application.

Although the application is built in Java it does not suffer any performance consequences on smaller databases as a result of this. Passing the larger database, using Java's nanosecond timer, takes on average 154.1 milliseconds to parse. While this may not seen like a long time if we scale it up to a few megabytes the parsing could potentially be a issue.

In addition to the performance scaling, there were a few feature that did not get added to the final application that would have enhanced it. Including the table exporter, to CSV, XML and JSon, a log exporter, and the ability to modify the database without having to use SQL, just by editing the values shown in the

interface. In addition to this I would have also liked to enhance the SQL editor, by providing auto complete and syntax highlighting. Lastly, I would have liked to enabled the table view, to collect the data from the snapshot rather then the database file. This would have enabled users to browser the data at a specific time slot easier, rather then having to inspect the nodes within the visualiser. Most of theses features were left out due to time constraints.

5.3 User interface

The overall user interface, looks clean and follows the standard layout seen in many other application. With the menu bar located at the top and the content underneath. This makes it intuitive and easy to pick up. With that said, support for a more customisable experience would enhance the experience, such as themes, text size and font type.

In addition to allowing the user to customise the interface, I would have also liked to improve the visualisation, to allow collapsible nodes, and have the nodes better represent what it contains, either as a minimised preview, or icons, rather then using colour coding.

Along side the visualiser, other aspects of the user interface could be polished up to make the overall experience enjoyable. Such as disabling the time line control buttons when they cannot move along the time line in that direction. Zooming to the mouse pointer position on the visualiser rather then always to the top left. Lastly, allowing the section to be popped out of the main window frame, allowing them to be dragged around separately. Although theses do not affect the application drastically, they are complementary to the entire experience.

6 Conclusion

At the start of this paper, I defined some very clear goals that I hoped to achieve by the end of it. Firstly, I wanted to extend my knowledge and understand why SQLite is so good, what makes it so prevalent and how it works. This included the file format and its systems. This was achieved throughout the first chapter of this paper.

The second aim was to take this knowledge and build a tool that could record all operations performed onto the database. And provide the same insight I have gathered throughout this project without having to look through a Hex editor. It should also be easy to use and well tested. Making it reliable and efficient. This was successfully achieved through the second to forth sections.

In conclusion this project has been an overall success. The main aims that I set out have all been met. However, the performance could still be improved. The user interface stills need some polishing in order to make it more user friendly. Providing a better visualisation of the database. Though the last stretch is always the longest and I could spend a very long time polishing the interface, and fixing all the edge cases that have not yet made themselves apparent. In the future, I would like to prove support for the other system such as extensions, lock byte and pointer map pages, and any other changes made to SQLite. Another project that would also be useful stemming off of this one is to try and recreate the original SQL query sent to the database based on the changes made, since it can only currently only list the changes.

But, with that said this has been an enjoyable project, and by then end of it all I have learnt a great deal about SQLite, Java and JavaFX, some more of the unique data structures such as the Merkle trees. And I have got a nice tool that can be used in the future, whenever I am working on a SQLite database to discover any problems.

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.

Gamma E. and Beck K. (1997), On line publication, Junit, http://junit.org/, Last Accessed 18th February 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.

Nilsson D. and Strath H. (2012), TestFX, On line publication, https://github.com/TestFX/TestFX. Last Accessed 17th February 2016

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

Rocha L. (2014), Chinook Database, On Line Publication, https://chinookdatabase.codeplex.com/, Last Accessed 10th March 2016

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.

7 Appendix

7.1 Appendix A

Table 7.1 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.
40	4	Schema Cookie. The schema version, each time the schema is modified this number is incremented.

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

7.2 Appendix B

Time in milliseconds to parse a database.

Time in milliseconds
210
147
147

Time in milliseconds
163
145
142
141
139
138
169

Table 7.2: Time in miliseconds to parse