SQLITE INTERNALS

HOW THE WORLD'S MOST USED DATABASE WORKS

ABDUR- RAHMAAN JANHANGEER

SQLite Internals: How The World's Most Used Database Works

by Abdur-Rahmaan Janhangeer

build: 0.8.2

Chapter: Foreword on SQLite Internals

To all SQLite lovers. This book discusses SQLite internals in depth.

You can view [compileralchemy.com] or [contribute to the book] or [download the book] It is OpenSource! Feel free to contribute a section, propose rewrites, fix typos etc. If you have comments, mail them to `arj.python at gmail dot com`.

The book is in progress as of now!

Particular thanks to the LibSQL maintainers. Started this book as a series of presentations to DevFest and the LibSQL community. I wanted to contribute to LibSQL. A book is far better than presentation slides.

I also owe much to Dan Shearer from LumoSQL. For his time reviewing a pre-run of the presentation.

Also, i just could not find a sane free book on SQLite internals! Free books help keep human legacy around. Without books, you burn time, a lot of it.

If you want to discuss dbs all day long, i recommend joining the Multiprocess communnity #db (Phil Eaton).

Since i scrapped my notes together, this did not start as a proper book. PRs welcome.

Chapter: Introduction

SQLite is a file-based database which is extremely reliable and stable. It is the world's most used database. It's used on military devices, on planes (the A350 for instance) and in space. The codebase and mechanisms it uses is extremely complex. The seemingly simple nature of it and adoption makes a good case for deep diving into in a fascinating piece of software.

It also implemented many features years ahead of popular databases like partial indices.

It's pronounced S-Q-L-ite, like mineral. But whatever is easy to pronounce is fine [10].

Chapter: Contributors

```
Main content:
   Abdur-Rahmaan Janhangeer, https://github.com/Abdur-RahmaanJ
```

Thanks

```
Stephan Beal, https://github.com/sgbeal
   Reporting and correcting the contribution link
Jakub Martin, Author of OctoSQL, https://github.com/cube2222/octosql
   For popularising the book
```

General improvements

```
Jaime Terreu, https://github.com/Confidenceman02
Aryan Arora, https://github.com/aryanA101a
```

Chapter: The Story Behind

SQLite was written by Dwayne Richard Hipp. It is not uncommon to see it being abbreviated to D. Richard Hipp or DRH for short. The story of how the database came around is fascinating. It sheds light on the author's mindset and SQLite general coding culture.

DRH holds a computer science doctorate in computational linguistics without taking prior programming courses. He also has a masters in electrical engineering and went to work for Bell Labs! [9]

Since his early days he was very dedicated. He dropped out of academia as the race was full of candidates. He turned to consulting. During that time, he was signed a software contract with shipyard *Bath Iron Works*. His work involved finding the solution to pipe burst failure by controlling valves on a warship: the *DDG-79 Oscar Austin*.

Richard had a problem. The software often did not work as the database server was down all the time. The ship was using *Informix*. So, he thought of spinning his own database.

> one of the guys I was working with says, "Richard, why don't you just write one?" "Okay, I'll give it a try.
>
> …
>
> all government contracts got shut down, so I was out of work for a few months, and I thought, "Well, I'll just write that database engine now."

Contrary to many popular projects, Richard thought of a bytecode-driven engine since the begining. This shows his previous exposure to compiler crafstmanship.

> so I wrote a byte code engine that would actually run a query and then I wrote a compiler that would translate SQL into that byte code and voila, SQLite was born.

How SQLite picked up speed

SQLite was not an overnight success though people did realise it's potential since the early days. This is a list of some milestones which led to SQLite what it is today.

2000 - The Internet: Since the shipyard was adamant on *Informix*, SQLite was not used on the warship. Robert put the code out in the wild on the internet. One great moment was a personal initiative from a user running it on his *Palm Pilot*.

2001 - Motorola OS: Motorola was a phone manufacturing company. The operating system they were using had SQLite on it. They wanted some help. During the whole time, Richard has been working on the project as an OpenSource one. So, they proposed an $80k contract to Richard for support and enhancements. It was the first time that the author realized that OpenSource can bring in money. He rounded his OSS team and shipped the project. This would be the first in a series of long-lasting relationship with phone companies.

200x - America Online: The next serious company to reach out was America Online. They wanted the database on CDs they were mailing to customers. Richard enthusiastically accepted the offer and midway realized the solution he had in mind would not work. These types of challenges helped SQLite grow into a robust product. At one point they also requested to be able to handle binary data, the feature was incorporated in SQLite3.

2005 - Symbian OS: Symbian flew Richard to their office in London. Among many databases they evaluated, both OSS and closed-source, SQLite was chosen among 10 dbs [10]. Symbian was a great company but they had a problem. They wanted to ensure that the project lives on even if Richard is no longer around. They wanted to increase the bus factor by having a SQLite consortium.

200x - SQLite Consortium: Richard liked the idea of a consortium. He started devising a plan of his own. Luckily someone from the Mozilla foundation (Mitchell Baker) reached out to him. They did not like the way he was setting up the framework around the consortium by giving members voting rights. They proposed keeping the direction of the project in developers hand. The friend from Mozilla the current setup. It is the consortium which really helped SQLite keep going, stay current, relevent and vibrant.

200x - Google & Android: Google was a complete outsider to the phone game. Soon, they approached Richard for a daring project. Having a phone connected to the internet with a robust software lifecycle was something extraordinary. They wanted SQLite to behave perfectly on this innovation. Richard's experience with the phone industry knew that Android was going to be a huge hit.

> We were going around boasting to everybody naively that SQLite didn't have any bugs in it, or no serious bugs, but Android definitely proved us wrong. … It's amazing how many bugs will crop up when your software suddenly gets shipped on millions of devices.

200x - Rockwell Collins: Rockwell Collins was a multinational corporation providing avionics and information technology systems and services to government agencies and aircraft manufacturers. They wanted the *DO-178B* aviation quality standard for SQLite. It meant 100% MCDC test coverage. This helped shaped SQLite test-backed approach to development.

SQLite tests are better than even postgres which relies on peer reviews [3]. This allows the developers to experiment and change code fearlessly.

Chapter: Technical Context

SQLite is notorious for implenting a bunch of functionalities from scratch. It's a daring, amazing, bold and crazy spirit which requires confidence and professionalism. People also call it the *From First Principles* approach. With no internet at the tips of the fingers and no wikipedia to consult, the author deserves massive respect. His teachers must have been proud to have their student be the living embodiment of what computer science and software engineering should be about.

DRH does look for alternatives. He does try out libraries. But, at the end of the day he ends up coding from scratch.

Engine: First, he needed a database engine, he looked around, was not satisfied and went on to pull off his own implementation.

B-tree implementation: The same goes for the b-tree layer. Much like a hero from a movie, he pulled Donald Knuth's algorithm book from the shelf and coded the b-tree he needed. He also completed the book's exercise about deleting elements.

Parser: He doesn't understand the use of YACC, Bison and Lex when anybody can code their own parsers. He coded his own parser-generator called *Lemon*.

Version Control System: He was using Git, but some functionalities were scratching his itch to build his own Version Control System. So, as usual, he wrote *Fossil*. It's the VCS you would download and configure if you download the source as is from the website.

> … And it's GPL, and so SQLite Version 1 was GPL, it had to be because it was linking against the GPL library. But GDBM is only key-value, I can't do range queries with it. Then I said, "I'm gonna write my own B-tree layer

Disassembling and re-building is really in his DNA. He had failed episodes of course, but it demonstrates an incredible spirit.

> Printing was not an option. I looked at ways of making my own printer. … , there was not much electrical interface to it. So that didn't work out well. [10]

To drive the point home, i think we can leave it at this one.

> And the text editor that I used to write SQLite is one that I wrote myself. [10]

Why implement from scratch?

The from scratch spirit is much preferred as it enables the developers to have the freedom they want. They can choose what they want or how they implement things. Just wrapping over another library might be a problem waiting to happen.

We can expect the library to be fairly complex as there are several components present which require knowledge of their own.

At one point, DRH also notes that they were going to use the Berkeley DB at some point but decided against it due to vague documentation [10] and coded their own implementation. The were amused that sometimes after the licensing changed causing lots of people to forsake the DB.

> I never understood lex because it’s so easy to write a bunch of C codes faster then Lex [1]

General points before diving in

Competing with f-open: SQLite advertises itself as being in competition not with other databases but with saving custom data on file. If you want to save data to a file, just use and share SQLite databases.

Relationship with Postgres: SQLite tries hard to keep up to the SQL standard postgres adopts as the team considers the db as the best reference platform [11]. DRH was the keynote speaker at PGCon 2014 with a talk entitled “SQLite: Protégé of PostgreSQL”.

Relationship with TCL: Sometimes, SQLite talks are given at TCL conferences. This might be tripping from a conceptual and search point of view. SQLite started as a TCL extension.

The spirit of typing: SQLite preferred to be called flexibly typed rather than weakly typed. By design, the author aimed not to get in the way of the programmer by allowing data of a different type to be inserted in the db. It’s directly inspired by scripting languages.

The symbiotic relationship between SQLite And Fossil: SQLite’s code is managed by Fossil, it’s Version Control System. And, Fossil uses SQLite.

No license: Being in the public domain by waiving rights to the code is an incredible decision. Add to it no external dependencies it means that people using SQLite have the peace of mind that the SQLite authors are not going to sue them over some piece of code or worry about some 3rd party companies talking about stealing code.

One big source file: SQLite also provides a source file where all files are amalgamated into so that SQLite can be inserted easily into projects and compiled.

Stats: SQLite is about 160k lines of code as now, with some 230k if comments and blank lines are included.

Chapter: Overview

A rough overview of SQLite is as follows

```
--------------      ------------
| SQLite lib |  ⇐  | SQL code |
--------------      ------------
      ⇑  ⇓
---------------
| Binary file |
---------------
```

A brief overview of the compilation step is as follows. The compiler takes the SQL code and outputs bytecodes. The Virtual Machine (VM) takes the bytecode and executes it.

```
+----------+     +----------+     +----+
| Compiler | --> | bytecode | --> | VM |
+----------+     +----------+     +----+
```

The compilation and execution process

A better view of the process might be

```
  SQL
   |
   v
[ parser ]
   |
   v
[ code generator ]
   |
   v
[ VM ]
   |
   v
[ btree ]
   |
   v
[ pager ]
   |
   v
[ shim ]
   |
   v
[ OS Interface ]
```

The first part of the library is called the compiler. It is executed using the `sqlite3_prepare_v2()` function and outputs prepared statements aka bytecodes.

```
[ parser ]          \
   |                 \ compiler
   v                 /
[ code generator ]  /
   |
   v
[ VM ]
   |
   v
[ btree ]
   |
   v
[ pager ]
   |
   v
[ shim ]
   |
   v
[ OS Interface ]
```

The second part of the library runs the program. It is executed using the `sqlite3_step()` function.

```
[ parser ]
   |
   v
[ code generator ]
   |
   v
[ VM ]              \
   |                 \
   v                  \
[ btree ]              \
   |                    \ run the program
   v                    /
[ pager ]              /
   |                  /
   v                 /
[ shim ]            /
   |               /
   v              /
[ OS Interface ] /
```

The btree layer and onward is called the storage engine.

```
[ parser ]
   |
   v
[ code generator ]
   |
   v
[ VM ]
   |
   v
[ btree ]        \
   |              \
   v               \
[ pager ]           \
   |                 \ storage engine
   v                 /
[ shim ]            /
   |               /
   v              /
[ OS Interface ] /
```

Steps explanation

- 👉 **Tokeniser - Parser:** The parser is a push-down automaton parser. It is reentrant and thread-safe. It is generated by lemon. Relevent files include `parse.y`, `tool/lemon.c`. Outputs AST (`sqliteInt.h`).
- 👉 **Code generator:** It does semantic analysis. It does AST transformation using

query planning using `select.c`. It outputs bytecodes using `build.c`, `delete.c`, `expr.c`, `insert.c`, `update.c`. It is the section with the most lines of code.

- 👉 **Virtual Machine:** It is the section with the 2nd most number of lines of code. Relevant files includes `vdbe.c`, `vdbe.h`, `vdbeLnt.h`, `vdbe*.c`, `func.c`, `date.c`. It executes bytecode instructions from the previous step.

```
[ parser ]
    |
    v
[ code generator ]
    |
    v
[ VM ]
    | Interface defined by btree.h
    v
[ btree ]
    |
    v
[ pager ]
    |
    v
[ shim ]
    |
    v
[ OS Interface ]
```

- 👉 **B-tree:** SQLite uses both B+ and B- trees. B+ tree is used for storing tables and B- is used for indexes. There can be multiple btrees per database file. It is read using a cursor. Concurrent reads and writes on same table is done using different cursors.
- 👉 **Pager:** Also called page cache. Prevents from data corruption in case of power loss. It uses two mutually exclusive modes to achieve this. The Roll back mode or the Write Ahead Log (WAL) mode. It also enforces concurrency control. It is responsible for dealing with in-memory cache. Relevent files include `pager.c`, `pager.h`, `pcache1.c`, `pcache.c`, `pcache.h`, `wal.c`, `wal.h`.
- 👉 **Shim:** The Shim layer is responsible for compression, logging and encryption. It is used to emulate an OS layer. It is used for tests to simulate hardware failures. Relevant files include `test_multiplex.c`, `test_vfstrace.c`
- 👉 **OS Interface:** It is used for os-specific interfacing. It can be changed at runtime. It is responsible for I/O (`test_onefile.c`). Relevant files include `os.c`, `os_unix.c`, `os_win.c`, `os*.h`. The Virtual File System (VFS) is another name this layer.

Important concepts

Those are some concepts which occur frequently and it pays to know about them in advance.

Bytes

A byte consists of 8 bits.

B-tree

A B-tree is a data structure providing logarithmmic operation time. SQLite keeps the depth as low as possible. It plays on the breadth of the 2nd and 3rd layers. It provides storage in this usecase for key/data storage with unique and ordered keys.

Big and small endian

TODO

Var int

TODO

A variable-length integer or “varint” is a static Huffman encoding of 64-bit twos-complement integers that uses less space for small positive values. A varint is between 1 and 9 bytes in length. The varint consists of either zero or more bytes which have the high-order bit set followed by a single byte with the high-order bit clear, or nine bytes, whichever is shorter. The lower seven bits of each of the first eight bytes and all 8 bits of the ninth byte are used to reconstruct the 64-bit twos-complement integer. Varints are big-endian: bits taken from the earlier byte of the varint are more significant than bits taken from the later bytes.

Chapter: File & Record Format

A SQLite file is a series of bytes.

```
[b1 b2 b3 b4 b5 ...]
```

It is divided into equally-sized chunks called pages. There can be one or more pages.

```
----------------------------------------------
| page 1 | page 2 | page 3 | page 4 | page 5 |
----------------------------------------------
```

The first page is the most important. It declares vital information about the file. The first page looks like this. The first 16 bytes contains the string `SQLite format 3`. In hex it is like this `53 51 4c 69 74 65 20 66 6f 72 6d 61 74 20 33 00`, including the null terminator at the end `\000`.

The next two bytes states the file size. Before 3.7.0.1 it had to be a power of two between 512 and 32768. As from 3.7.1 it can be of size 65536. Since such a large number cannot fit in 2 bytes, the value is set to `0x00 0x01`. This represents big-endian 1 and is used to specify a size of 65536.

```
0                 16       18
------------------------------
| SQLite format 3 |  400   |
------------------------------
[                     page 1        ..
```

The first page

Here is a complete table about what the first page contains.

```
start byte - offset byte - description

00  16  The header string: "SQLite format 3\000"
16  02  The database page size in bytes. Must be a power of two
        between 512 and 32768 inclusive, or the value 1
        representing a page size of 65536.
18  01  File format write version. 1 for legacy; 2 for WAL.
19  01  File format read version. 1 for legacy; 2 for WAL.
20  01  Bytes of unused "reserved" space at the end of each page.
        Usually 0.
21  01  Maximum embedded payload fraction. Must be 64.
22  01  Minimum embedded payload fraction. Must be 32.
23  01  Leaf payload fraction. Must be 32.
24  04  File change counter.
28  04  Size of the database file in pages.
        The "in-header database size".
32  04  Page number of the first freelist trunk page.
36  04  Total number of freelist pages.
40  04  The schema cookie.
44  04  The schema format number.Supported schema formats are
        1, 2, 3, and 4.
48  04  Default page cache size.
52  04  The page number of the largest root b-tree page when
        in auto-vacuum or incremental-vacuum modes, or zero
        otherwise.
56  04  The database text encoding. A value of 1 means UTF-8.
        A value of 2 means UTF-16le. A value of 3 means UTF-16be.
60  04  The "user version" as read and set by the
        user_version pragma.
64  04  True (non-zero) for incremental-vacuum mode. False
        (zero) otherwise.
68  04  The "Application ID" set by PRAGMA application_id.
72  20  Reserved for expansion. Must be zero.
92  04  The version-valid-for number.
96  04  SQLITE_VERSION_NUMBER
```

The first page contains 100 bytes less storage space.

```
[ db header | free space ]
    |            |
   100 bytes     --- Can be any type of page
```

The free space can be of any type of page, but, it will contain less information than a typical page handles. This needs some adjustments in some cases in the way information is stored for that type of page.

Types of pages

In this section we pass over the different types of pages used by SQLite. Any page

- 👉 The lock-byte page
- 👉 A freelist page
- 👉 A b-tree page
- 👉 A payload overflow page
- 👉 A pointer map page

The Lock-Byte Page

This page is retained only to preserve backward compatibility. It was conceived for Microsoft 95. When it is present, it occurs at bytes offset 1073741824 and 1073742335. If the file doesn't have that many bytes, the page does not exist. If it does have the necessary bytes, there is only one such page. It's dealt with by the VFS layer rather than SQLite core.

Freelist pages

```
             has
             many
A freelist -------- freelist trunk page [ n1, n2, n3 ]
   page                                   |   |   |
                  A freelist leaf page ----   |   |
                                              |   ---- A freelist
                  A freelist leaf page --------        leaf page
```

Unused pages are stored on the freelist. It is a linked list of trunk pages with each page containing page numbers for zero or more freelist leaf pages, which contain nothing. These pages can be reused. When using the `VACCUM` command, the freelist is purged and a new database file is written. When auto-vaccum is enabled, freelist is not used a new compacted db is written on each commit.

B-Tree pages

```
               can be either
A b-tree page ---------------
                    \                       either
                     --- table b-tree page --------- leaf page
                     |                          \
                     |                           \__ interior page
                     |
                     |                       either
                     --- index b-tree page ---------- leaf page
                                                 \
                                                  \__ interior page
```

B-tree pages can be either a table page or an index page. A page is always either a leaf pae of an interior page.

A btree looks like this

```
                     root page
                     ---------------------------
                     | key | data | key | data |
                     ---------------------------
                              | data: interior page key
                ---------------
                |
interior page   |           interior page               interior page
--------------------------- --------------------------- ------------
| key | data | key | data | | key | data | key | data | | key |
--------------------------- --------------------------- --------- ...
                      |
                      -----
            leaf page     |         leaf page                 leaf page
----------------------  ----------------------   ----------------------
| key |    data      |  | key |    data      |   | key |    data      |
----------------------  ----------------------   ----------------------
|     key   |  data  |  |     key   |  data  |   | key |    data      |
----------------------  ----------------------   ----------------------
|   key   |   data   |  |   key   |   data   |   | key |    data      |
----------------------  ----------------------   ----------------------
```

Keys are integers. The data of a root page is the key of an interior page. The data of an interior page is the key of a leaf page. Database records are stored in the data section of a leaf page.

A key in a leaf table is a 64-bit signed int

An interior page contains k number of keys, at least 2, upto how many fits on page. This is unless page 1 is an interior b-tree page in which case it can handle one key only. It also contains k+1 number of pointers to child b-tree pages A pointer is a 32-bit unsigned integer page number of the child page.

Conceptually speaking, in an interior b-tree page, the pointers and keys logically alternate with a pointer on both ends, keys in ascending order from left to right.

```
[  pointer  -  key  - pointer - key - pointer - ... - pointer ]
    |           |                                        |
    --- cell ----                    stored separately ---
```

> A leaf page can be a root page. Root pages are identified by their root page number

There is one table b-tree in each db file for each rowid table.

A rowid table is a table which has a unique key to access data in the b-tree strorage engine.

About overflowing

If the data section in a leaf page becomes bigger than the space available in a page, it is linked to another page. If it's size exceeds the other page, it is added to yet other another page.

```
part of leaf page
----------------------------------------------
| key |          data          | page 23     |----
----------------------------------------------   |
                                                 |
overflow page                                    |
----------------------------------------------   |
|                  page 23                   |   |
|                                            |----
|                                            |
|                                            |----
----------------------------------------------   |
                                                 |
overflow page                                    |
----------------------------------------------   |
|                  page 27                   |   |
|                                            |----
|                                            |
|                                            |
----------------------------------------------
```

Large keys on index b-trees are split up into overflow pages so that no single key uses more than one fourth of the available storage space on the page and hence every internal page is able to store at least 4 keys

The integer keys of table b-trees are never large enough to require overflow, so key overflow only occurs on index b-trees.

Record format

The data part of a leaf page is stored in binary format and consists of 2 parts:

- 👉 The header
- 👉 The type part
- 👉 The data part

```
[ key ][ data ]
          |
          v
        [ header size | type1 | type2 | data1 | data2 ]
```

A row such as this

```
id     0
price  3
name   shoe
```

Would be encoded as

```
[ 04 | 01 | 01 | 21 ] [ 00 | 03 | shoe ]
```

Here’s the table SQLite consults for encoding and decoding

```
Serial Type, Content Size,   Value meaning
0            0           NULL
1            1           8-bit twos-complement integer.
2            2           big-endian 16-bit twos-complement integer.
3            3           big-endian 24-bit twos-complement integer.
4            4           big-endian 32-bit twos-complement integer.
5            6           big-endian 48-bit twos-complement integer.
6            8           big-endian 64-bit twos-complement integer.
7            8           big-endian IEEE 754-2008 64-bit floating , number.
8            0           integer 0. (Only available for schema format >= 4)
9            0           integer 1. (Only available for schema format >= 4)
10,11        variable    Reserved for internal use. These serial type codes
                         will never appear in a well-formed database file,
                         but they might be used in transient and temporary
                         database files that SQLite sometimes generates for
                         its own use. The meanings of these codes can shift
                         from one release of SQLite to the next.
N≥12, even   (N-12)/2    Value is a BLOB that is (N-12)/2 bytes in length.
N≥13, odd    (N-13)/2    Value is a string in the text encoding and
                         (N-13)/2 bytes in length. The nul terminator is not
                         stored.
```

```
[ 04 ] header size, including the size itself
[ 01 ] type 8-bit twos-complement integer.
[ 01 ] type 8-bit twos-complement integer.
[ 21 ] As 21 >= 13 and is odd,
       (N-13)/2 == length of string shoe in encoding defined in db
                   here we are assuming utf8
       (N-13)/2 == 4
       N == 4 * 2 + 13
       N == 21

[ 00 ] value of id field
[ 03 ] value of price field
[ shoe ] value of name field
```

Btree page format

This is what a b-tree page looks like.

```
----------------------
| 100 byte header    | (if page 1)
----------------------
| 8 or 12 byte       | b-tree page header
----------------------       08: leaf page
| cell pointer array |       12: interior page
----------------------
| free space         |
----------------------
| cell content area  |
----------------------
| reserved region    |
----------------------
```

The reserved region is found in all pages except the locking page. It can be used by extensions to write per-page information. It’s size is defined in the database header at an offset of bytes 20.

Here is the format of the b-tree page header.

```
Offset  Size    Description
0   1   The one-byte flag at offset 0 indicating
        the b-tree page type.
            02 (0x02): page is an interior index b-tree page.
            05 (0x05): page is an interior table b-tree page.
            10 (0x0a): page is a leaf index b-tree page.
            13 (0x0d): page is a leaf table b-tree page.
            Any other value for the b-tree page type is an error.
1   2   The two-byte integer at offset 1 gives the start of the
        first freeblock on the page, or is zero if there are no
        freeblocks.
3   2   The two-byte integer at offset 3 gives the number of cells
        on the page.
5   2   The two-byte integer at offset 5 designates the start of the
        cell content area. A zero value for this integer is interpreted
        as 65536.
7   1   The one-byte integer at offset 7 gives the number of fragmented
        free bytes within the cell content area.
8   4   The four-byte page number at offset 8 is the right-most pointer.
        This value appears in the header of interior b-tree pages only
        and is omitted from all other pages.
```

TOADD: Freeblock

Chapter: Rollback & WAL mode

In case of power cuts, SQLite ensures that data is not lost. The pager layer responsible for executing these two modes. The Write Ahead Log (WAL) mode is better than the Rollback for for two reasons:

- 👉 It is faster
- 👉 It allows reads and writes at the same time

The Rollback mode is the default primarily due to these reasons

- 👉 Some computers are still around which have weird memory mappings
- 👉 Several computers accessing the file might cause issues
- 👉 Backward compatibility is not guaranteed. Waiting until WAL is even more stable.
- 👉 Hash lookup for page in WAL mode is in shared memory

The Rollback mode

When reading occurs, the process acquires a shared lock.

```
|            |          |      |
| user space | os cache | disk |
|      a     |    a     |   x  |
|      a     |          |   x  |
|            |    a     |   x  |
|            |          |   x  |
                   🔒 shared*
```

A shared lock prevents processes from changing data.

When **writing**, a reserved lock is acquired. A journal is also created. Journals in the this mode have the `.database-journal` extension.

```
|            |          |      |
| user space | os cache | disk |
|      a     |    a     |   x  |
|      a     |          |   x  |
|            |    a     |   x  |
|            |          |   x  |
                   🔒 reserved*
|            |          |      | file.database-journal
|            |          |      |
|            |          |      |
|            |          |      |
|            |          |      |
|            |          |      |
| user space | os cache | disk |
|      a     |    a     |   x  |
|      a     |          |   x  |
|            |    a     |   x  |
|            |          |   x  |
                 🔒 reserved
|            |          |      | file.database-journal
|            |          |      |
|            |          |      |
|            |    a*    |      |
|            |    a*    |      |
```

Then the data is changed

```
|            |          |      |
| user space | os cache | disk |
|      b*    |    a     |   x  |
|      b*    |          |   x  |
|            |    a     |   x  |
|            |          |   x  |
                 🔒 reserved
|            |          |      | journal
|            |          |      |
|            |          |      |
|            |    a     |      |
|            |    a     |      |
```

Then the data in the journal cache is flushed to the journal on disk.

```
|            |          |      |
| user space | os cache | disk |
|      b     |    a     |   x  |
|      b     |          |   x  |
|            |    a     |   x  |
|            |          |   x  |
                 🔒 reserved
|            |          |      | journal
|            |          |      |
|            |          |      |
|            |    a     |   a* |
|            |    a     |   a* |
```

This takes some times and can be turned off but, won’t guarantee that data is safe during power failure.

The an exlclusive lock is acquired. This stops writing completely!

```
|            |          |      |
| user space | os cache | disk |
|      b     |    a     |   x  |
|      b     |          |   x  |
|            |    a     |   x  |
|            |          |   x  |
                  🔒 exclusive*
|            |          |      | journal
|            |          |      |
|            |          |      |
|            |    a     |   a  |
|            |    a     |   a  |
```

The new values are flushed to the os cache.

```
|            |          |      |
| user space | os cache | disk |
|      b     |    b*    |   x  |
|      b     |          |   x  |
|            |    b*    |   x  |
|            |          |   x  |
                  🔒 exclusive
|            |          |      | journal
|            |          |      |
|            |          |      |
|            |    a     |   a  |
|            |    a     |   a  |
```

Then it is flushed to disk

```
|            |          |      |
| user space | os cache | disk |
|      b     |    b     |   b* |
|      b     |          |   x  |
|            |    b     |   b* |
|            |          |   x  |
                  🔒 exclusive
|            |          |      | journal
|            |          |      |
|            |          |      |
|            |    a     |   a  |
|            |    a     |   a  |
```

When a commit occurs, it deletes the journal.

```
|            |          |      |
| user space | os cache | disk |
|      b     |    b     |   b  |
|      b     |          |   x  |
|            |    b     |   b  |
|            |          |   x  |
                 🔒 exclusive
|            |          |      | journal
|            |          |      |
|            |          |      |
|            |          |      |
|            |          |      |
```

If power loss before commit

Now, if there is a power loss before commit, the situation would be as follows.

```
|            |          |      |
| user space | os cache | disk |
|            |          |   b  |
|            |          |   x  |
|            |          |   x  |
|            |          |   x  |

|            |          |      | journal
|            |          |      |
|            |          |      |
|            |          |   a  |
|            |          |   a  |
```

When power is restored, a shared lock is acquired.

```
|            |          |      |
| user space | os cache | disk |
|            |          |   b  |
|            |          |   x  |
|            |          |   x  |
|            |          |   x  |
                 🔒 shared*
|            |          |      | journal
|            |          |      |
|            |          |      |
|            |          |   a  |
|            |          |   a  |
```

Then an exclusive lock is acquired.

```
|            |          |      |
| user space | os cache | disk |
|            |          |   b  |
|            |          |   x  |
|            |          |   x  |
|            |          |   x  |
                 🔒 exclusive*
|            |          |      | journal
|            |          |      |
|            |          |      |
|            |          |   a  |
|            |          |   a  |
```

Then data is copied from the journal disk to the journal cache.

```
|            |          |      |
| user space | os cache | disk |
|            |          |   b  |
|            |          |   x  |
|            |          |   x  |
|            |          |   x  |
                 🔒 exclusive
|            |          |      | journal
|            |          |      |
|            |          |      |
|            |    a*    |   a  |
|            |    a*    |   a  |
```

Then it is copied from the journal cache to the OS cache.

```
|            |          |      |
| user space | os cache | disk |
|            |    a*    |   b  |
|            |    a*    |   x  |
|            |          |   x  |
|            |          |   x  |
                 🔒 exclusive
|            |          |      | journal
|            |          |      |
|            |          |      |
|            |    a     |   a  |
|            |    a     |   a  |
```

Then it is flushed to disk

```
|            |          |      |
| user space | os cache | disk |
|            |    a     |   a* |
|            |    a     |   a* |
|            |          |   x  |
|            |          |   x  |
                 🔒 exclusive
|            |          |      | journal
|            |          |      |
|            |          |      |
|            |    a     |   a  |
|            |    a     |   a  |
```

The Write Ahead Log (WAL) mode

Just as in Rollback mode, first a shared lock is acquired. WAL journals have a `.database-wal` extension.

```
|            |          |      |
| user space | os cache | disk |
|      a     |    a     |   x  |
|      a     |          |   x  |
|            |    a     |   x  |
|            |          |   x  |
                 🔒 shared*
|            |          |      | file.database-wal
|            |          |      |
|            |          |      |
|            |          |      |
|            |          |      |
```

Then, the value is changed

```
|            |          |      |
| user space | os cache | disk |
|      b*    |    a     |   x  |
|      b*    |          |   x  |
|            |    a     |   x  |
|            |          |   x  |
                 🔒 shared
|            |          |      | file.database-wal
|            |          |      |
|            |          |      |
|            |          |      |
|            |          |      |
```

Now, if another process is accessing the db, the old value is flushed to the journal cache.

```
|            |            |          |      |
| user2      | user space | os cache | disk |
|     b*     |      b     |    a     <-  x  |
|     a*     |      b     |          |   x  |
|            |            |    a     <-  x  |
|            |            |          |   x  |
                 🔒 shared
                          |          |      |
                          |          |      | wal
                          |          |      |
                          |    b*    |      |
                          |    b*    |      |
```

The new process gets a snapshot of the data. For illustration purposes below, `b` from wal cache, `a` from os cache.

```
|            |            |          |      |
| user2      | user space | os cache | disk |
|     b      |      b     |    a     |   x  |
|     c*     |      b     |          |   x  |
|            |            |    a     |   x  |
|            |            |          |   x  |
                 🔒 shared
                          |          |      |
                          |          |      | wal
                          |          |      |
                          |    b     |      |
                          |    c*    |      |
```

Different reads and writes with snapshot isolation can occur.

```
|            |            |          |      |
| user2      | user space | os cache | disk |
|     b      |      b     |    a     |   x  |
|     c      |      b     |          |   x  |
|            |            |    a     |   x  |
|            |            |          |   x  |
                 🔒 shared
                          |          |      |
                          |          |      | wal
                          |          |      |
                          |    b     |   b* |
                          |    c     |   c* |
```

A checkpoint operation truncates the journal cache and disk content.

```
|            |         |      |
| user space | os cache | disk |
|            |    c     |   x  |
|            |    b     |   x  |
|            |    c     |   x  |
|            |    a     |   x  |
                 🔒 shared
             |         |      |
             |         |      | wal
             |         |      |
             |         |      |
             |         |      |
```

Chapter: Bytecodes

The entire Virtual Machine is contained in `vdbe.c`. On compiling the project, `vdbe.c` produces two files: `opcodes.h` which assigns a numerical value to opcodes and `opcodes.c` which designates a symbolic name for an opcode.

opcodes.h

```
/* Automatically generated.  Do not edit */
/* See the tool/mkopcodeh.tcl script for details */
#define OP_Savepoint       0
#define OP_AutoCommit      1
#define OP_Transaction     2
#define OP_Checkpoint      3
#define OP_JournalMode     4
#define OP_Vacuum          5
#define OP_VFilter         6 /* jump, synopsis: iplan=r[P3] zplan='P4'     */
...
```

opcodes.c

```
/* Automatically generated.  Do not edit */
/* See the tool/mkopcodec.tcl script for details. */
#if !defined(SQLITE_OMIT_EXPLAIN) \
 || defined(VDBE_PROFILE) \
 || defined(SQLITE_DEBUG)
#if defined(SQLITE_ENABLE_EXPLAIN_COMMENTS) || defined(SQLITE_DEBUG)
# define OpHelp(X) "\0" X
#else
# define OpHelp(X)
#endif
const char *sqlite3OpcodeName(int i){
 static const char *const azName[] = {
    /*   0 */ "Savepoint"        OpHelp(""),
    /*   1 */ "AutoCommit"       OpHelp(""),
    /*   2 */ "Transaction"      OpHelp(""),
    /*   3 */ "Checkpoint"       OpHelp(""),
    /*   4 */ "JournalMode"      OpHelp(""),
    /*   5 */ "Vacuum"           OpHelp(""),
    /*   6 */ "VFilter"          OpHelp("iplan=r[P3] zplan='P4'"),
    ...
 }
```

bytecodes are composed of two parts, the opname, short for operation name and the opargs, short for operation arguments.

```
opname oparg oparg oparg oparg oparg
```

Using the `EXPLAIN` keyword, we can view an output based on bytecodes.

```
sqlite3> EXPLAIN SELECT price FROM product WHERE price=100;
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Init           0     11    0                    0   Start at 11
1     OpenRead       0     30    0     3              0   root=30 iDb=0;
product
2     Rewind         0     10    0                    0
3       Column         0     2     1                    0   r[1]= cursor 0
column 2
4       RealAffinity   1     0     0                    0
5       Ne             2     9     1     BINARY-8       85  if r[1]!=r[2] goto
9
6       Column         0     2     3                    0   r[3]= cursor 0
column 2
7       RealAffinity   3     0     0                    0
8       ResultRow      3     1     0                    0   output=r[3]
9     Next           0     3     0                    1
10    Halt           0     0     0                    0
11    Transaction    0     0     25    0              1   usesStmtJournal=0
12    Integer        100   2     0                    0   r[2]=100
13    Goto           0     1     0                    0
```

Simplified opcode guide

A simplified guide designed for speedy referencing is available at the end of the book. It is meant to be within hands reach when dealing with bytecodes. If you want more info, read the source or refer to the opcode docs which contains nearly all that is in the source.

Chapter: Interesting Features

Virtual Tables

Common Table Expressions

Oracle needed recursive queries and they added common table expressions.

Save points

Partial Indices

Developed for Expensify.

Chapter: Knowing The Internals

WebSQL

- 👉 WebStorage on the web
- 👉 Not going to implement an engine from scratch: Use SQLite
- 👉 "User agents must implement the SQL dialect supported by Sqlite 3.6.19"
- 👉 Example exploit: Omer Gull - SELECT code execution FROM USING SQLite [4]
- 👉 Need upated version of SQLite -> conflict with requirement of 3.6.19
- 👉 Aug 2022 Chrome: Deprecating and Removing webSQL [5]
- 👉 Memory corruption available from JS
- 👉 Replaced by the beautiful IndexedDB written by a developer from the noble house of Oracle

Chapter: How SQLite Is modified

LibSQL

LibSQL is a great fork of SQLite with the aim of making SQLite Open Source. Currently, SQLite operates in a Source Open rather than OpenSource mode. It aims to state compatible with SQLite.

> With the advent of Wasm, SQL or NoSQL solutions can come to the web. One example is DuckDB-Wasm, another is absurd-sql. Based on these creations, we feel that the developer community can iterate on and create new storage solutions faster and better than browser vendors.

libSQL introduced native WASM support to SQLite

LumoSQL

LumoSQL is a clone that is 100% on time. It does not rely on merging the master. It has swappable db engine and btree. It has an edge on cryptography.

Martina Palmucci's Master Thesis [11]: Martina wrote a thesis entitled "Securing databases using Attribute Based Encryption and Shamir's Secret Sharing (SSS)" on the LumoSQL project. It has been merged. The thesis combines SSS and access based on user attributes like SELECT etc. It is abbreviated as ABE-SSS Attribute-based Encryption Shamir's Secret Sharing. There is an increased need to saveguard data privacy. File-based encryption means that the data is in the clear once the file is decrypted. Another layer of encryption at the data-level, particularly the field level protects against internal attacks.

SSS operates by having shares: secrets that, when combined together produce a key. Elliptic curves reveal interesting properties for cryptographic uses.

A standard protocol used is the Elliptic-curve Diffie–Hellman (ECDH). It allows two parties to create a shared secret across an unsecured channel. But, many protocols based on ECDH often require a prime-order group. Elliptic curve groups are often compound (group that is not made up of prime numbers). Using the Decaf technique, it is possible to obtain a prime-order group from an elliptic curve group. Applying the Decaf technique to Curve25519 yields Ristretto255.

Elliptic-curve Integrated Encryption Scheme (ECIES) is a hybrid encryption scheme.

cryptosystems inside it.

The projects implements access control using a policy tree made up of booleans. Attributes are encrypted using a private key. Resources have corresponding policy trees. Access to a resource is granted if the result of evaluating a user policy expression is true. For a resource tree there is a corresponding Shamir shares tree which is encrypted.

Distributed clones

TOADD

Bloomberg

Bloomberg uses the SQLite code generator and storage engine. The replaced the layers after by their own implementation of a scaled, massively concurrent, multi-data center storage engine.

Chapter: The Future

LibSQL and LumoSQL are great OpenSource projects.

Chapter: Ending Quotes

On not listening to institutionilized experts

> I had this crazy idea that I'm going to build a database engine that does not have a server, that talks directly to disk, and ignores the data types, and if you asked any of the experts of the day, they would say, "That's impossible. That will never work. That's a stupid idea." Fortunately, I didn't know any experts and so I did it anyway, so this sort of thing happens. I think, maybe, just don't listen to the experts too much and do what makes sense. Solve your problem.

On not pondering on what lies ahead too much

> If I'd known how hard it would be I probably never would've have written it [3]

On the opportunity to learn surrounding techs

> Apple I came out, and I was about to buy the Apple I and the Apple II came out. And I bought just the motherboard for an Apple II. Got it.
>
> Had to build my own keyboard, my own power supply, soldered it altogether. The first board I got didn't work. I called up Apple, they put me through the technical support and Steve Wozniak answers the phone. and said, "Oh, yeah. Send it back. We'll send you another board." They sent me another motherboard and that one worked. [10] (About his Apple II) With just 4k of RAM i could understand everything that was going on in that computer. I can understand everything the computer was doing there but now you know with the smallest computer having 4GB of RAM, there's no way someone coming into this now can understand everything that's going on in that computer. So, i started very simple. [9]

On how to learn the knowledge he posses

> I accumulated all this knowledge in the course of four decades, five decades almost. How do you learn that in 4 years of university? I don't know. … you have some things take that as an article of faith, yeah this works believe it. [9]

References

Simplified Opcode guide.

```
Goto ? P2 * * *
   An unconditional jump to address P2.
   P1 sometimes set to 1 for indentation purposes


Gosub P1 P2 * * *
   Write the current address onto register P1
   and then jump to address P2.


Return P1 ? P3 * *
   Jump to the address stored in register P1.  If P1 is a return address
   register, then this accomplishes a return from a subroutine.

   If P3:
      1: jump is only taken if register P1 holds an integer
         Used in combination with OP_BeginSubrtn
      0: then register P1 must hold an


EndCoroutine P1 * * * *
   The instruction at the address in register P1 is a Yield.
   Jump to the P2 parameter of that Yield.
   After the jump, register P1 becomes undefined.


Yield P1 P2 * * *
   Swap program counter with value in register P1. This
   has the effect of yielding to a coroutine.
   If the coroutine launched ends with Yield or Return:
      continue to the next instruction.
   If coroutine launched ends with EndCoroutine:
      jump to P2


HaltIfNull  P1 P2 P3 P4 P5, if r[P3]=null halt
   Check value in register P3.
   If NULL:
      Halt using P1, P2, and P4 as if this were a Halt instruction.
   else:
      routine is a no-op.
   P5 parameter should be 1.


Halt P1 P2 * P4 P5
   Exit immediately.  All open cursors, etc are closed
   automatically.
   P1: result code returned by sqlite3_exec(), sqlite3_reset(),
      or sqlite3_finalize().
      For a normal halt, this should be SQLITE_OK (0).

   If P1!=0:
      P2 will determine whether or not to rollback
      the current transaction.
      if P2==OE_Fail:
         Do not rollback
         Do the rollback
      if P2==OE_Abort:
         back out all changes that have occurred during this execution of the
         VDBE, but do not rollback the transaction.

   If P4 != null then it is an error message string.

   P5 is a value between 0 and 4, inclusive, that modifies the P4 string.

      0:  (no change)
      1:  NOT NULL contraint failed: P4
      2:  UNIQUE constraint failed: P4
      3:  CHECK constraint failed: P4
      4:  FOREIGN KEY constraint failed: P4

   If P5 != zero and P4 is NULL:
      everything after the ":" is omitted.

   There is an implied "Halt 0 0 0" instruction inserted at the very end of
   every program.  So a jump past the last instruction of the program
   is the same as executing Halt.

Integer P1 P2 * * *, r[P2]=P1
   32-bit integer value P1 is written into register P2.



TODO: complete
```

References

- [1] SQLite, A Database for the Edge of the Network, DRH, Databaseology Lectures, Carnegie Mellon (2015)
- [2] CORECURSIVE Podcast, Episode #066, The Untold Story of SQLite
- [3] Richard Hipp Speaks Out on
  SQLite, ACM SIGMOD interviews with DB people, Marianne Winslett and Vanessa Braganholo (2019), https://sigmodrecord.org/publications/sigmodRecord/1906/pdfs/06_Profiles_Hipp.pdf
- [4] DEF CON 27 - Omer Gull - SELECT code execution FROM USING SQLite, https://www.youtube.com/watch?v=JokZUjwGj4M
- [5] Deprecating and removing Web SQL, https://developer.chrome.com/blog/deprecating-web-sql/
- [6] Craft vulnerable db
  https://github.com/CheckPointSW/QueryOrientedProgramming/blob/master/qop.py
- [7] https://www.sqlite.org/fileformat.html#record_format
- [8] https://fly.io/blog/sqlite-internals-btree/
- [9] Richard Hipp, SQLite main author - Two Weeks of Database, https://www.youtube.com/watch?v=2eaQzahCeh4
- [10] Changelog podcast Episode 201, Why SQLite succeeded as a database
  https://changelog.com/podcast/201
- [11] https://www.pgcon.org/2014/schedule/attachments/319_PGCon2014OpeningKeynote.pdf
- [12] Securing databases using Attribute Based Encryption and Shamir's Secret Sharing (SSS), Martina Pulmucci,
  https://lumosql.org/src/lumosql/raw/c3f5ace49a2139e623be647a3a65753adfe4fddd46fb86f345e0bd75ff
  at=LumoSQL-Thesis-Martina-Palmucci-2022.pdf

Images

- DGG-79: https://news.usni.org/2019/06/04/uss-oscar-austin-fire-damage-repairs-will-stretch-into-2022