Julia Evans

How does SQLite work? Part 1: pages!

• how-things-work • favorite •

This evening the fantastic Kamal and I sat down to learn a little more about databases than we did before.

I wanted to hack on SQLite, because I've used it before, it requires no configuration or separate server process, I'd been told that its source code is well-written and approachable, and all the data is stored in one file. Perfect!

To start out, I created a new database like this:

```
drop table if exists fun;
create table fun (
    id int PRIMARY KEY,
    word string
);
```

Just a primary key and a string! What could be simpler? I then wrote a little Python script to put the contents of `/usr/share/dict/words` in the database:

```
import sqlite3
c = sqlite3.connect("./fun.sqlite")
with open('/usr/share/dict/words') as f:
    for i, word in enumerate(f):
        word = word.strip()
        word = unicode(word, 'latin1')
        c.execute("INSERT INTO fun VALUES (?, ?);", (i, word))
c.commit()
c.close()
```

Great! Now we have a 4MB database called `fun.sqlite` for experimentation. My favorite first thing to do with binary files is to `cat` them. That worked pretty well, but Kamal pointed out that of course `hexdump` is a better way to look at binary files. The output of `hexdump -C fun.sqlite` looks something like this:

```
|.............{.n|
|.a.R.D.4.%......|
|................|
|...y.n._.N.>.,.$|
|................|
|..............F.|
|..EAcevedo.E...D|
|Accra's.D...CAcc|
|ra.C..#BAccentur|
|e's.B...AAccentu|
|re.A..!@Acapulco|
|'s.@...?Acapulco|
|.?...>Acadia's.>|
|...=Aradia.=...<|
|Ac's.<...;Ac.;..|
|%:Abyssinian's.:|
|..!9Abyssinian.9|
|..#8Abyssinia's.|
|8...7Abyssinia.7|
```

I've pasted the first few thousand lines of the hexdump in this gist, so you can look more closely. You'll see that the file is alternately split into words and gibberish – there will be a sequence of mostly words, and then unreadable nonsense.

Of course there's a rhyme to this reason! The wonderfully written File Format for SQLite Databases tells us that a SQLite database is split into **pages**, and that bytes 16 and 17 of our file are the **page size**.

My `fun.sqlite` starts like this:

```
00000000  53 51 4c 69 74 65 20 66  6f 72 6d 61 74 20 33 00  |SQLite format 3.|
00000010  04 00 01 01 00 40 20 20  00 00 00 27 00 00 0c be  |.....@  ...'....|
          ^^ ^^
        page size :)
```

so our page size is `0x0400` bytes, or 1024 bytes, or 1k. So this database is split into a bunch of 1k chunks called pages.

There's an index on the `id` column of our `fun` table, which lets us run queries like `select * from fun where id = 100` quickly. To be a bit more precise: to find row 100, we

don't need to read every page, we can just read a few pages. I've always understood indexes in a pretty vague way – I know that they're "some kind of tree", which lets you access data in O(log n), and in particular that databases use something called a **btree**. I still do not really know what a btree is. Let's see if we can do any better!

Here's where it starts to get really fun! I downloaded the sqlite source code, and Kamal and I figured out how to get it to compile. (using nix, which is a totally other story)

Then I put in a print statement so that it would tell me every time it accesses a page. There's about 140,000 lines of SQLite source code, which is a bit intimidating!

It's also incredibly well commented, though, and includes adorable notes like this:

```
/************** End of btree.c ************************************************/
/************** Begin file backup.c *******************************************/
/*
** 2009 January 28
**
** The author disclaims copyright to this source code.  In place of
** a legal notice, here is a blessing:
**
**    May you do good and not evil.
**    May you find forgiveness for yourself and forgive others.
**    May you share freely, never taking more than you give.
**
*************************************************************************
** This file contains the implementation of the sqlite3_backup_XXX()
** API functions and the related features.
```

My next goal was to get SQLite to tell me how it was traversing the pages. Some careful grepping of the 140,000 lines led us to this function `btreePageFromDbPage`. All page reads need to go through this function, so we can just add some logging to it :)

```
/*
** Convert a DbPage obtained from the pager into a MemPage used by
** the btree layer.
*/
static MemPage *btreePageFromDbPage(DbPage *pDbPage, Pgno pgno, BtShared *pBt){
  MemPage *pPage = (MemPage*)sqlite3PagerGetExtra(pDbPage);
  pPage->aData = sqlite3PagerGetData(pDbPage);
  pPage->pDbPage = pDbPage;
  pPage->pBt = pBt;
  pPage->pgno = pgno;
  printf("Read a btree page, page number %d\n", pgno); // added by me!
  pPage->hdrOffset = pPage->pgno==1 ? 100 : 0;
  return pPage;
}
```

Now it'll notify us every time it reads a page! NEAT! Let's experiment a little bit.

```
sqlite> select * from fun where id = 1;
Read a btree page, page number 1
Read a btree page, page number 5
Read a btree page, page number 828
Read a btree page, page number 10
Read a btree page, page number 2
Read a btree page, page number 76
Read a btree page, page number 6
1|A's

sqlite> select * from fun where id = 20;
Read a btree page, page number 1
Read a btree page, page number 5
Read a btree page, page number 828
Read a btree page, page number 10
Read a btree page, page number 2
Read a btree page, page number 76
Read a btree page, page number 6
20|Aaliyah
```

Those two rows (1 and 20) are in the same page, so it traverses the same path to get to both of them!

```
sqlite> select * from fun where id = 200;
Read a btree page, page number 1
Read a btree page, page number 5
Read a btree page, page number 828
Read a btree page, page number 11
Read a btree page, page number 2
Read a btree page, page number 76
Read a btree page, page number 2818
200|Aggie
```

Apparently `200` is pretty close in the tree, but it needs to go to page `2818` instead at the end. And `80000` is much further away:

```
sqlite> select * from fun where id = 80000;
Read a btree page, page number 1
Read a btree page, page number 5
Read a btree page, page number 1198
Read a btree page, page number 992
Read a btree page, page number 2
Read a btree page, page number 1813
Read a btree page, page number 449
80000|scarfs
```

If we go back and inspect the file, we can see that pages 1, 5, 1198, 992, 2, and 1813 are *interior nodes* – they have no data in them, just pointers to other pages. Pages 6, 2818, and 449 are *leaf nodes*, and they're where the data is.

I'm still not super clear on how exactly the interior pages are structured and how the pointers to their child pages work. It's time to sleep now, but perhaps that will happen another day!

Modifying open source programs to print out debug information to understand their internals better: SO FUN.

Strange Loop 2014 How does SQLite work? Part 2: btrees! (or: disk seeks are slow don't do them!)

Archives