Julia Evans

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

• how-things-work •

Welcome back to fun with databases! In Part 1 of this series, we learned that:

- SQLite databases are organized into fixed-size **pages**. I made an example database which had 1k pages.
- The pages are all part of a kind of tree called a **btree**.
- There are two kinds of pages: **interior pages** and **leaf pages**. Data is only stored in leaf pages.

I mentioned last time that I put in some print statements to tell me every time I read a page, like this:

```
sqlite> select * from fun where id = 10;
Read a btree page, page number 4122
Read a btree page, page number 900
Read a btree page, page number 5
```

Let's understand a little bit more about how these btrees work! First, some theory.

Normally when I think about tracking tree accesses, I think about it in terms of "how many times you need to jump to a new node". So in a binary tree with depth 10, you might need to jump up to 10 times.

One of the most important things in database optimization is disk I/O. Reading more data than you absolutely need to read is really expensive, because seeking to a new location in a file takes a long time. It takes **way less** CPU time to search through your data than it does to read the data into memory (see: Computers are fast!, Latency Numbers Every Programmer Should Know).

So for a simple database scan, reading more data than you need is a huge problem

So far we know that:

1. Our database is organized into 1k pages

2. We want to read as little of the database as possible to write a query, and
3. My filesystem has a "block size" of 4k, which means that it's impossible to read less than 4k at a time.

We can conclude from this that we want to read **as few pages as possible**. btrees are organized so that each node has lots of children, to keep the depth small, and so that we won't have to read too many pages to find a row.

My 100,000 row SQLite database has a btree with depth 3, so to fetch a node I only need to read 3 pages. If I'd used a binary tree I would have needed to do log(100000) / log(2) = 16 seeks! That's more than five times as many. So these btrees seem like a pretty good idea.

The index and table btrees

So far we've been talking like there's only one btree. This isn't actually true at all! My database has one table, and two btrees.

Each table has a btree, made up of interior and leaf nodes. Leaf nodes contain all the data, and interior nodes point to other child nodes.

Every index for that table also has its own btree, where you can look up which row id a column value corresponds to. This is why maintaining lots of indexes is slow – every time you insert a row you need to update as many trees as you have indexes.

Let's dive into a query a bit more deeply. It turns out SQLite does a binary search inside every page of every btree to find out what node to go to next. I've printed out the indexes it tries while doing the binary search.

```
sqlite> select * from fun where id = 1000;

Read a btree page, page number 1

Searching for 1000...

Read a btree page, page number 4122
Number of cells in page: 62
idx: 30
idx: 14
idx: 6
idx: 2
idx: 0

Read a btree page, page number 900
Number of cells in page: 67
idx: 33
idx: 16
idx: 7
idx: 3
idx: 1
idx: 2

Read a btree page, page number 7
Number of cells in page: 69
```

```
idx: 34
idx: 16
idx: 7
idx: 3
idx: 1
idx: 2

-----------------------------------
Looking for key: 99001

Read a btree page, page number 2
Number of cells in page: 19
Current key: index   9, value 51627
Current key: index  14, value 75920
Current key: index  16, value 86203
Current key: index  17, value 91286
Current key: index  18, value 95577

Read a btree page, page number 4091
Number of cells in page: 67
Current key: index  33, value 97372
Current key: index  50, value 98277
Current key: index  58, value 98698
Current key: index  62, value 98927
Current key: index  64, value 99044
Current key: index  63, value 98985

Read a btree page, page number 4129
Number of cells in page: 59
Current key: index  29, value 99015
Current key: index  14, value 99000
Current key: index  21, value 99007
Current key: index  17, value 99003
Current key: index  15, value 99001
1000|yummier
```

Wow, that was a lot of work. There were actually two separate searches here, in two different btrees.

1. Look up the **rowid** associated with the number `1000` in the **index btree** – `99001`. (some rowid docs).
2. Look up `99001` in the **table btree**.

It's really neat to see it doing the binary search for `99001` inside each page! I couldn't quite figure out how to get it to print out the comparisons it was doing when looking up 1000 in the index btree, because it does some weird function pointer magic to do comparisons.

Kamal is parsing SQLite databases with the Python construct module next to me right now and it is AMAZING. There may be future installments. Who knows!

How does SQLite work? Part 1: pages! How to set up a blog in 5 minutes

Archives

 If you like this, you may like Ulia Ea or, more seriously, this list of blogs I love.
You might also like the Recurse Center, my very favorite programming community (my posts about it)