# Using SQLite to Speed Up Pollen Renders

THURSDAY, JUNE 21ST, 2018

I use a system written in Pollen and Racket[1,2] to generate this blog. It renders the HTML file for most individual pages in less than half a second per page —not blazing, certainly, but serviceable. However, when rendering pages that combine multiple posts, my code has been interminably slow—40 seconds to render the home page, for example, and 50 seconds to render the RSS feed.

By creating my own cache system using a SQLite database, I was able to drop those times to less than a second each.

My old approach, which used only Pollen's functions for retrieving docs and metas, had some pretty glaring inefficiencies[3]—so glaring, in fact, that I probably could have cut my render times for these "problem pages" down to 3–6 seconds just by addressing them. But I thought I could get even further by making my own cache system.

Pollen's cache system[4] is optimized for fetching individual docs and metas one by one, not for grabbing and dynamically sorting a whole bunch of them at once. Also, Pollen's cache doesn't store the rendered HTML of each page. There are good reasons for both of these things; Pollen is optimized for making books, not blogs, and the content of a book doesn't often change. But Pollen was also designed to be infinitely flexible, which allows us to extend it with whatever facilities we need.

Here is the gist of the new approach:

1. Create an[5] SQLite database
2. In the template code for individual posts (i.e. at the same time the post is being rendered to HTML) save the post's title, date and a copy of the rendered HTML in the database.

1. http://pollenpub.com
2. This post assumes some familiarity with Pollen and SQL.
3. For example, in the code for my RSS feed, I was fetching the docs and metas from every document in the entire pagetree (that is, every post I had ever written) into a list of `structs`, even though I would only use the five most recent posts in the feed itself.
4. http://docs.racket-lang.org/pollen/Cache.html
5. After hearing SQLite's author on this podcast (https://changelog.com/podcast/201) I fell into the habit of pronouncing it the way he does: S-Q-L-ite, as though it were a mineral. Hence, "an".

3. In the "combination pages" (such as `index.html` and `feed.xml`), query the SQLite database for the HTML and spit that out.

By doing this, I was able to make the renders 50–80 times faster. To illustrate, here are some typical render times using my old approach[6]:

```
joel@macbook$ raco pollen render feed.xml
rendering feed.xml
rendering: /feed.xml.pp as /feed.xml
cpu time: 51819 real time: 52189 gc time: 13419

joel@macbook$ raco pollen render index.html
rendering index.html
rendering: /index.html.pp as /index.html
cpu time: 39621 real time: 39695 gc time: 7960
```

And here are the render times for the same pages after adding the SQLite cache:

```
joel@macbook$ raco pollen render feed.xml
rendering feed.xml
rendering: /feed.xml.pp as /feed.xml
cpu time: 659 real time: 660 gc time: 132

joel@macbook$ raco pollen render index.html
rendering index.html
rendering: /index.html.pp as /index.html
cpu time: 824 real time: 825 gc time: 188
```

This still isn't nearly as fast as some static site generators[7]. But it's plenty good enough for my needs.

## Creating the cache

I made a new file, `util-db.rkt`[8] to hold all the database functionality and to provide functions that save and retrieve posts. My `pollen.rkt` re-provides all these functions, which ensures that they are available for use in my templates and `.pp` files.

The database itself is created and maintained by the `template.html.p` file used to render the HTML for indivdual posts. Here is the top portion of that file:

6. https://github.com/otherjoel/thenotepad/blob/2fd5c69b126f8695929a2c12b68b437afdc48416/index.html.pp
7. Hugo (https://gohugo.io) has been benchmarked (https://youtu.be/CdiDYZ51a2o) building a complete 5,000 post site in less than six seconds total.
8. You can browse the source code for this blog (https://github.com/otherjoel/thenotepad) on its public repository.

template.html.p

```
...◊
(init-db)◊
(define-values (doc-body comments) (split-body-comments doc))◊
(define doc-body-html (->html (cdr doc-body)))◊
(define doc-header (->html (post-header here metas)))◊
(save-post here metas doc-header doc-body-html)
<!DOCTYPE html>
<html lang="en"> ...
```

The expression `init-db` ensures the database file exists, and runs some `CREATE TABLE IF NOT EXISTS` queries to ensure the tables are set up correctly. The `save-post` expression saves the post's metas and rendered HTML into the database.

Some more notes on the code above: The `split-body-comments` and `post-header` functions come from another module I wrote, `util-template.rkt`. The first separates any comments (that is, any `◊comment` tags in the Pollen source) from the body of the post, which lets me save just the body HTML in the database. The second provides an X-expression for the post's header, which includes or omits various things depending on the post's metas.

## Database design

Internally, `util-db.rkt` has some very basic functions that generate and execute the SQL queries I need.

If you watch the console while rendering a single post, you'll see these queries being logged (indented for clarity):

```
CREATE TABLE IF NOT EXISTS `posts`
  (`pagenode`,
   `published`,
   `updated`,
   `title`,
   `header_html`,
   `html`,
   PRIMARY KEY (`pagenode`))

CREATE TABLE IF NOT EXISTS `posts-topics`
  (`pagenode`,
   `topic`,
   PRIMARY KEY (`pagenode`, `topic`))

INSERT OR REPLACE INTO `posts`
  (`rowid`, `pagenode`, `published`, `updated`, `title`, `header_html`, `html`)
  values ((SELECT `rowid` FROM `posts` WHERE `pagenode`= ?1), ?1, ?2, ?3, ?4, ?5, ?6)

DELETE FROM `posts-topics` WHERE `pagenode`=?1
```

```
INSERT INTO `posts-topics` (`pagenode`, `topic`)
  VALUES ("posts/pollen-and-sqlite.html", "SQLite"),
       ("posts/pollen-and-sqlite.html", "Pollen"),
       ("posts/pollen-and-sqlite.html", "Racket"),
       ("posts/pollen-and-sqlite.html", "programming")
```

The schema and queries are designed to be *idempotent*, meaning I can safely run them over and over again and end up with same set of records every time. So, no matter what state things are in, I don't have to worry about ending up with duplicate records or other out-of-whacknesses when I render the site.

The database is also designed to need as few queries as possible, both when saving and when fetching data: for example, using a single query to either create something if it doesn't exist, or to replace it if it does exist. Also, where applicable, creating multiple records in a single query (as in the last example above).

Finally, in the interests of keeping things "simple", I have tried to keep the database *disposable*: that is, no part of the site's content has its origin or permanent residence in the database itself. I can delete it at any point and quickly rebuild it. That way my use of it remains limited to my original plan for it: a speed enhancement, and nothing more.

## Building the site: order of operations

Since home page, RSS feed, and other aggregation pages are now getting all their content from the SQLite database, it's important that the database be up to date before those pages are rendered.

I already use a makefile[9] to make sure that when things change, only the parts that need updating are rebuilt, and only in a certain order. I won't go into detail about how the makefile works (a topic for another post, perhaps), but, in short, when I run `make all`, it does things in roughly the following order:

1. If any of the "core files" have changed, force a re-render of *all* individual posts.
2. If any individual posts have been changed or added, render just those to HTML. (This step is automatically skipped if step 1 was run, because that step will already have brought all the posts up to date.)
3. If any posts *or* core files have been changed or added, force a re-render of the "aggregation pages", such as the home page, the Topics page, and the RSS feed.

9. https://en.wikipedia.org/wiki/Makefile

This way, the pages that update the cache database are rendered before the pages that draw from the database, and everything is hunky-dory.

## The Topics system

This blog makes use of "topics", which are basically like tags. A post can have zero or more topics, and a topic can have or more posts.

This many-to-many relationship is another good fit for the database. You'll have noticed above that I store the topic/post relationships in a separate table. With each topic-post pair stored in its own row, it is fast and easy to fetch them all out of the database at once. The `util-db.rkt` module provides a function, `topic-list`, which does this. Here's what you see if you call it from DrRacket:

```
> (topic-list)
"SELECT `topic`, p.pagenode, p.title FROM `posts-topics` t INNER JOIN `posts` p ON t.pagenode = p.
    pagenode ORDER BY `topic` ASC"
'(("Active Directory" ("posts/laptop-user-not-authenticating-in-nt.html" "Laptop User Not
    Authenticating in an NT Domain After Changing Password"))
  ("Apple" ("posts/siri-slow-unreliable-and-maybe-not.html" "Siri: Slow, unreliable, and maybe not a
      priority at Apple"))
  ("audio"
   ("posts/how-to-convert-mp3-files-for-use-as-on-hold-music.html" "How to convert mp3 files for use
     as on-hold music")
   ("posts/how-to-record-with-a-yeti.html" "How to Record With a Yeti and Audacity (and eliminate
      background noise)"))
...
```

The SQL query uses `INNER JOIN` to link the posts in the `posts-topics` table with their titles in the `posts` table, resulting in a list of rows with three columns each: topic, pagenode (the relative path to the HTML file), and title. The `topic-list` function then groups these rows by the topic name and returns the resulting list.

The `topics.html.pp` file can make use of this nested list with some help from Pollen's new `for/splice` function[10] (Thanks Matthew!):

topics.html.pp

```
...
<table>◊
for/s[topic (topic-list)]{
  <tr>
    <td><a name◊="#(car topic)◊">(car topic)</a></td>
    <td><ul>◊
     for/s[post (cdr topic)]{
      <li><a href◊="/(list-ref post 0)◊">(list-ref post 1)</a></li>
```

10. /posts/a-quick-pollen-macro.html

```
    }</ul></td>
  </tr>
}
</table> ...
```

## Removing a post

One last consideration: *removing* a post is no longer as simple as deleting its Pollen source document. I have to remove it from the SQLite database as well, otherwise it will continue to appear on the home page, RSS feed, etc.

There are a few ways I could do this. The simplest would be to delete the database file and rebuild the site from scratch. Or I could open the database in a program like DB Browser for SQLite[11] and delete the rows manually. Or I could write a script to automate this. I don't often need to delete posts, so I'll probably put off writing any scripts for now.

11. http://sqlitebrowser.org