Hosting SQLite databases on Github Pages

(or any static file hoster)

APR 17, 2021 • LAST UPDATE MAY 03, 2021

I was writing a tiny website to display statistics of how much sponsored content a Youtube creator has over time when I noticed that I often write a small tool as a website that queries some data from a database and then displays it in a graph, a table, or similar. But if you want to use a database, you either need to write a backend (which you then need to host and maintain forever) or download the whole dataset into the browser (which is not so great when the dataset is more than 10MB).

In the past when I've used a backend server for these small side projects at some point some external API goes down or a key expires or I forget about the backend and stop paying for whatever VPS it was on. Then when I revisit it years later, I'm annoyed that it's gone and curse myself for relying on an external service - or on myself caring over a longer period of time.

Hosting a static website is much easier than a "real" server - there's many free and reliable options (like GitHub, GitLab Pages, Netlify, etc), and it scales to basically infinity without any effort.

So I wrote a tool to be able to use a real SQL database in a statically hosted website!

Here's a demo using the World Development Indicators dataset - a dataset with 6 tables and over 8 million rows (670 MiByte total).

DEMO

INPUT SQL EDIT

```
select country_code, long_name from wdi_country limit 3;
```

RUN

As you can see, we can query the `wdi_country` table while fetching only 1kB of data!

This is a full SQLite query engine. As such, we can use e.g. the SQLite JSON functions:

DEMO

INPUT SQL EDIT

```
select json_extract(arr.value, '$.foo.bar') as bar
  from json_each('[{"foo": {"bar": 123}}, {"foo": {"bar": "baz"}}]') as
arr
```

RUN

We can also register JS functions so they can be called from within a query. Here's an example with a `getFlag` function that gets the flag emoji for a country:

JS DEMO

INPUT JAVASCRIPT EDIT

```
function getFlag(country_code) {
  // just some unicode magic
  return String.fromCodePoint(...Array.from(country_code||"")
    .map(c => 127397 + c.codePointAt()));
}

await db.create_function("get_flag", getFlag)
return await db.query(`
  select long_name, get_flag("2-alpha_code") as flag from wdi_country
    where region is not null and currency_unit = 'Euro';
`)
```

RUN

> Press the Run button to run the following demos. You can change the code in any way you like, though if you make a query too broad it *will* fetch large amounts of data ;)

Note that this website is 100% hosted on a static file hoster (GitHub Pages).

So how do you use a database on a static file hoster? Firstly, SQLite (written in C) is compiled to WebAssembly. SQLite can be compiled with emscripten without any modifications, and the sql.js library is a thin JS wrapper around the wasm code.

sql.js only allows you to create and read from databases that are fully in memory though - so I implemented a virtual file system that fetches chunks of the database with HTTP Range requests when SQLite tries to read from the filesystem: sql.js-httpvfs. From SQLite's perspective, it just looks like it's living on a normal computer with an empty filesystem except for a file called `/wdi.sqlite3` that it can read from. Of course it can't write to this file, but a read-only database is still very useful.

Since fetching data via HTTP has a pretty large overhead, we need to fetch data in chunks and find some balance between the number of requests and the used bandwidth. Thankfully, SQLite already organizes its database in "pages" with a user-defined page size (4 KiB by default). I've set the page size to 1 KiB for this database.

Here's an example of a simple index lookup query:

DEMO

DEMO

INPUT SQL EDIT

```
select indicator_code, long_definition from wdi_series where
indicator_name
    = 'Literacy rate, youth total (% of people ages 15-24)'
```

RUN

Run the above query and look at the page read log. SQLite does 7 page reads for that query.

- Three page reads are just some to get some schema information (these are already cached)
- Two page reads are the index lookup in the index `on wdi_series (indicator_name)`
- Two page reads are on the `wdi_series` table data (the first to find the row value by primary key, the second to get the text data from an overflow page)

Both the index as well as the table reads are B-Tree lookups.

A more complex query: What are the countries with the lowest youth literacy rate, based on the newest data from after 2010?

DEMO

INPUT SQL EDIT

```
with newest_datapoints as (
  select country_code, indicator_code, max(year) as year from wdi_data
  join wdi_series using (indicator_code)
  where
    indicator_name = 'Literacy rate, youth total (% of people ages 15-
24)'
    and year > 2010
  group by country_code
)
select c.short_name as country, printf('%.1f %%', value) as "Youth
Literacy Rate"
from wdi_data
  join wdi_country c using (country_code)
  join newest_datapoints using (indicator_code, country_code, year)
order by value asc limit 10
```

RUN

The above query should do 10-20 GET requests, fetching a total of 130 - 270KiB, depending on if you ran the above demos as well. Note that it only has to do 20 requests and not 270 (as would be expected when fetching 270 KiB with 1 KiB at a time). That's because I implemented a pre-fetching system that tries to detect access patterns through three separate virtual read heads and exponentially increases the request size for sequential reads. This means that index scans or table scans reading more than a few KiB of data will only cause a number of requests that is logarithmic in the total byte length of the scan. You can see the effect of this by looking at the "Access pattern" column in the page read log above.

All of this only works well when we have indices in the database that match the queries well. For example, the index used in the above query is a `INDEX ON wdi_data (indicator_code, country_code, year, value)`. If that index did not include the value column, the SQLite engine would have to do another random access (unpredictable) read and thus HTTP request to retrieve the actual value for every data point. If the index was ordered `country_code, indicator_code, ...`, then we would be able to quickly get all indicators for a single country, but not all country values of a single indicator.

We can also make use of the SQLite FTS module so we can do a full-text search on the more text-heavy information in the database - in this case there are over 1000 human development indicators in the database with longer descriptions.

DEMO

INPUT SQL EDIT

```
select * from indicator_search
where indicator_search match 'educatio* femal*'
order by rank limit 10
```

RUN

The total amount of data in the `indicator_search` FTS table is around 8 MByte. The above query should only fetch around 70 KiB. You can see how it is constructed here.

And finally, here’s a more complete demonstration of the usefulness of this system - here’s an interactive graph showing the development of a few countries over time, for any countries you want using any indicator from the dataset:

Countries:

United States × Germany × India × China × Korea ×

Indicator:

Individuals using the Internet (% of population)

▶ Extra information about this indicator

Note that many indicators are only available for some countries, for example the indicator *"Women who believe a husband is justified in beating his wife when she burns the food"* is based on surveys only conducted in lower-developed countries.

Bonus: DOM as a database

Since we’re already running a database in our browser, why not use our browser as a database using a virtual table called `dom`?

DEMO

INPUT SQL EDIT

```
select count(*) as number_of_demos from dom
  where selector match '.content div.sqlite-httpvfs-demo';
select count(*) as sqlite_mentions from dom
  where selector match '.content p' and textContent like '%SQLite%';
```

RUN

We can even insert elements directly into the DOM:

DEMO

INPUT SQL EDIT

```
insert into dom (parent, tagName, textContent)
    select 'ul#outtable1', 'li', short_name
    from wdi_country where currency_unit = 'Euro'
```

RUN

Output:

And update elements in the DOM:

DEMO

INPUT SQL EDIT

```
update dom set textContent =
  get_flag("2-alpha_code") || ' ' || textContent
from wdi_country
where selector match 'ul#outtable1 > li'
  and textContent = wdi_country.short_name
```

RUN

Of course, everything here is open source. The main implementation of the sqlite wrapper is in sql.js-httpvfs. The source code of this blog post is a pandoc markdown file, with the demos being a custom "fenced code block" React component.

View post source on GitHub