Yurii Rashkovskii for Omnigres
Posted on Feb 6

What happens if you put HTTP server inside Postgres?

#webdev #postgres #performance

Cover photo by Clark Tibbs on Unsplash

Benchmarks and performance claims are attention-grabbers, but that's not what drew me to work on Omnigres. When I first built a prototype of its HTTP server, I didn't foresee the desire to share the numbers. As we all know, getting benchmarks right is hard, and everybody's mileage may vary. But I'll show you some numbers here anyway. It'll be great to validate or invalidate my findings!

But first, what's Omnigres? The shortest definition I came up with is "Postgres as a Platform."

What do I mean by this? Well, this comes from the idea that much of your application and its infrastructure can live inside or next to Postgres. Your business logic, deployment orchestration, caching, job queues, API endpoints, form handlers, HTTP server, you name it. Instead of building a system composed of multiple services, you only need one server that takes care of it. And if it can scale, you've got something great!

I am not the only one intrigued by this idea (and many don't like it, either): check out this HN thread.

And it is not "one-size-fits-all." But it fits a good set of problems. Anyway, that's a subject for another conversation.

In the past weeks, I've taken the task of adding an embedded HTTP server to Omnigres. Since Omnigres is implemented in C , it was only natural for me to choose libh2o to implement the functionality of an HTTP server. It did help that H2O is known for its good performance characteristics.

The idea was relatively simple: initialize a few HTTP worker processes, each quickly handing off details of the incoming HTTP requests to a Postgres instance in the same process.

Sounds a bit weird, right? But bear with me, I will try to explain what's going on.

Below is a sample piece of code of how an HTTP request can be handled.

```
SELECT omni_httpd.http_response(headers => array[omni_httpd.http_header('content-type', 't
       FROM request
       INNER JOIN users ON string_to_array(request.path,'/', '') = array[NULL, 'users', us
UNION
SELECT omni_httpd.http_response(status => 404, body => json_build_object('method', request
       FROM request
ORDER BY priority DESC
```

The first query in the union joins the request path of `/users/:user` with the `users` table on `users.handle` column and serves an HTML response with that user's name.

The second query, being lower in priority, simply returns a 404 Not Found response with a JSON that contains some of the request details.

```
$ http POST localhost:9000/users/johndoe
HTTP/1.1 200 OK
Connection: keep-alive
Server: omni_httpd-0.1
content-type: text/html
transfer-encoding: chunked

Hello, <b>John</b>!

$ http POST localhost:9000/test\?q
HTTP/1.1 404 OK
Connection: keep-alive
Server: omni_httpd-0.1
content-type: text/json
transfer-encoding: chunked

{
    "method": "POST",
    "path": "/test",
```

```
    "query_string": "q"
}
```

Once all this worked, I decided to try benchmark it (mostly because I wanted to see how bad would it be):

```
$ wrk http://localhost:9000/users/johndoe -c 20 -t 10
Running 10s test @ http://localhost:9000/users/johndoe
  10 threads and 20 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   354.85us    1.40ms  47.24ms   97.05%
    Req/Sec    12.51k     2.37k   25.01k    73.84%
  1256199 requests in 10.10s, 179.70MB read
Requests/sec: 124366.73
Transfer/sec:     17.79MB
```

That's on MacBook Pro M1 Max. I've also run the same test on an older x86_64-based Linux machine and got about 70K/sec.

For reference, I decided to try comparing this with a similar (but quick-and-dirty-written) Node.js implementation that goes to the database for the same effect:

```
const http = require("http");

const { Pool } = require('pg')

const pool = new Pool({ /* connection details */, max: 10, idleTimeMillis: 1000 * 60});

const host = 'localhost';
const port = 8000;
const requestListener = function (req, res) {
    const split = req.url.split('/');
    if (split.length == 3 && split[1] == 'users') {
      pool.query({name: "q", text: "SELECT users.name FROM users WHERE users.handle = $1",
        then((qres) => {
          if (qres.rows.length == 1) {
            res.setHeader("Content-Type", "text/html");
            res.writeHead(200);
            res.end(`Hello, <b>${qres.rows[0].name}</b>!`);
          } else {
            res.setHeader("Content-Type", "application/json");
            res.writeHead(200);
            res.end(JSON.stringify({method: req.method, path: req.url}));
          }
        })
    } else {
```

```
      res.setHeader("Content-Type", "application/json");
      res.writeHead(200);
      res.end(JSON.stringify({method: req.method, path: req.url}));
    }
};
const server = http.createServer(requestListener);
server.listen(port, host, () => {
    console.log(`Server is running on http://${host}:${port}`);
});
```

And I re-ran the test:

```
$ wrk http://localhost:8000/users/johndoe -c 20 -t 10
Running 10s test @ http://localhost:8000/users/johndoe
  10 threads and 20 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.11ms    0.99ms  33.09ms   99.15%
    Req/Sec     1.89k   166.30     2.19k    92.18%
  190319 requests in 10.10s, 33.76MB read
Requests/sec:  18838.56
Transfer/sec:      3.34MB
```

From the surface, it looks like omni_httpd is faster for the time being! I am sure with time it'll get slower and I did not explore enough optimizations available for the Node.js experiment (please contribute to make this picture fairer!). But it shows what bundling can do to performance!

I want to emphasize that performance was not and is not a primary goal behind this work. What I think is more important there is changing the way we think about serving HTTP clients, doing data-aware routing, etc. There's still a lot of research to be done on this end.

I also want to mention that this is still early work. There's a lot to be done to make this server production-grade (starting from enabling HTTPS support, and improving the architecture to make HTTP2 multiplexing really work, but not ending there), as well as building out other extensions to make building applications really easy and convenient.

If you want to chat more, join us on Discord or Telegram!

Recently Heroku shut down free Heroku Dynos, free Heroku Postgres, and free Heroku Data for Redis on November 28th, 2022. So Meshv Patel put together some free alternatives in this classic DEV post.

Omnigres

More from Omnigres

Why not Rust for Omnigres?

#rust #postgres #c