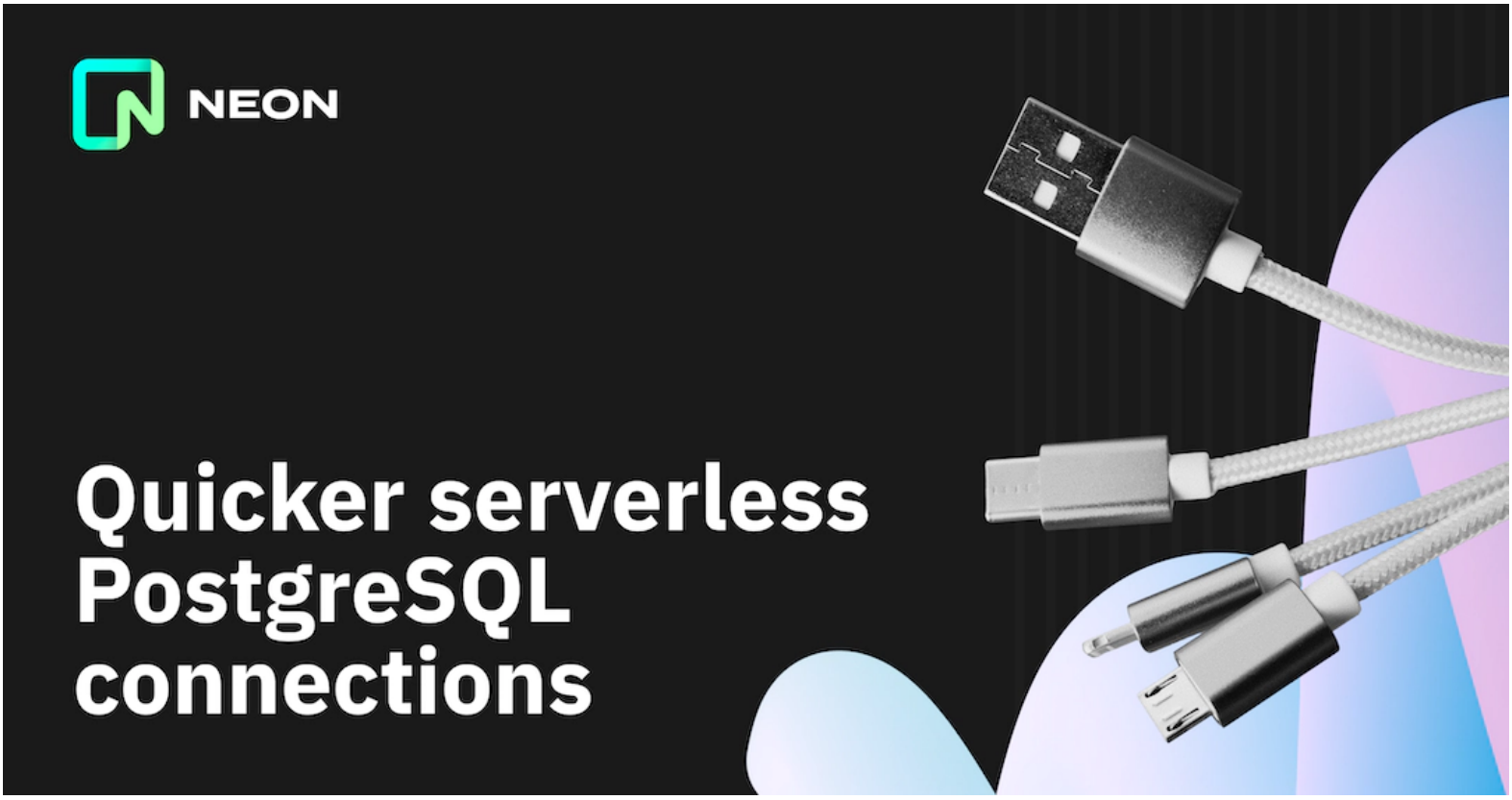
NEON

March 28, 2023 10 min read

Quicker serverless Postgres connections

Latency lowered by cutting network round-trips in half



Neon's serverless driver redirects the PostgreSQL TCP wire protocol over WebSockets. This makes ordinary, fully-functional PostgreSQL connections accessible from new environments — including serverless platforms like Cloudflare Workers and Vercel Edge Functions.

A key feature of these environments is that state is not generally persisted from one request to the next. That means we can't use standard client-side database connection pooling. Instead, we set up a new connection every time. And that makes the time taken to establish a new database connection particularly important.

Unless your database client is right next to your database, the great majority of the time it takes to connect will be spent waiting for data to travel back and forward across the network. Minimizing both the number and the length of these network round-trips is therefore critical to getting low latencies.

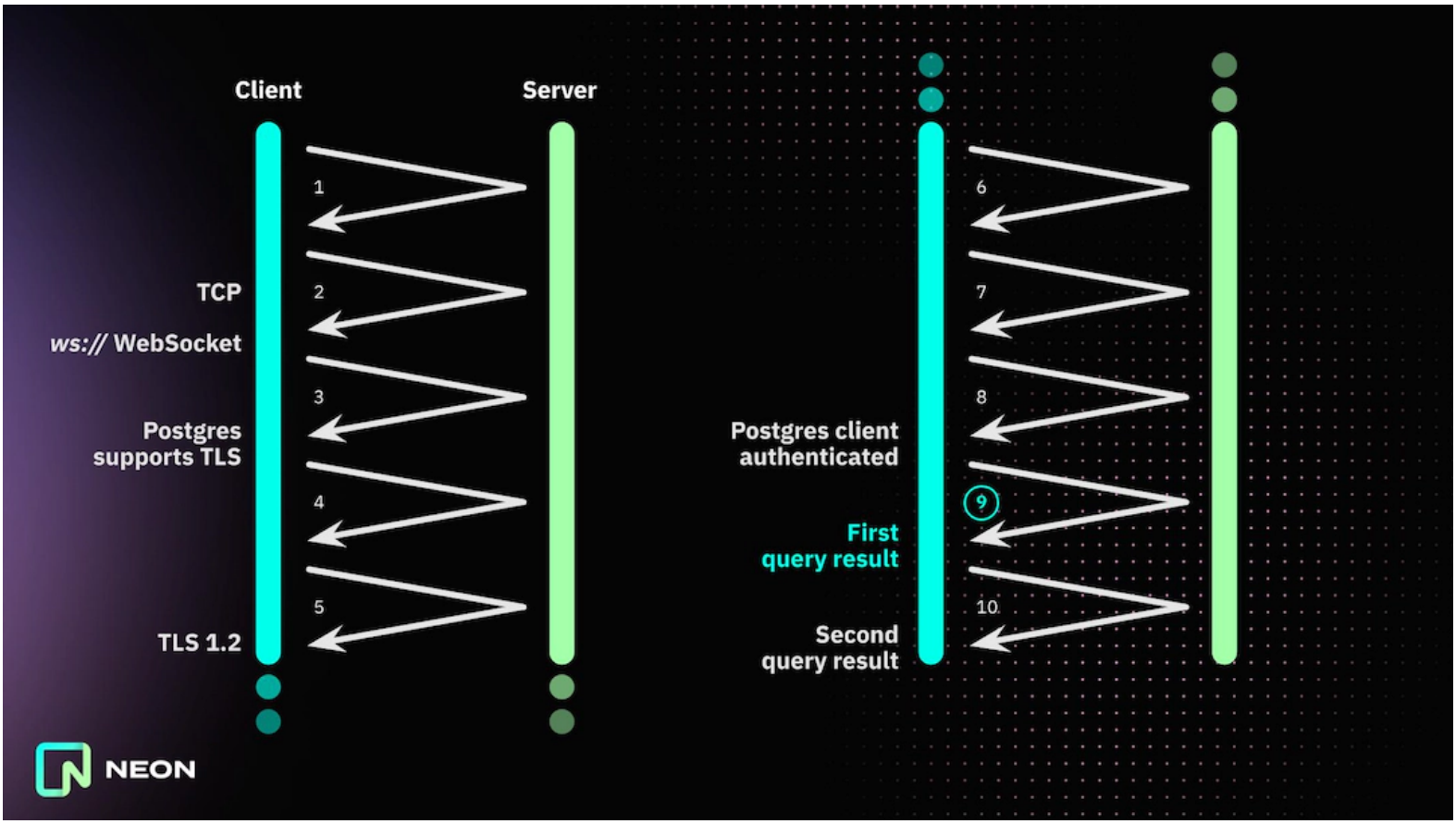
Read-replicas in multiple regions will help us minimize round-trip length. They're an important item on Neon's roadmap, but they're not ready yet. In the meantime, we're focused on bringing down the round-trip count.

Baseline: nine round-trips to first query result

The network round-trips underlying our first attempts to connect to Postgres over a WebSocket are shown below.

In summary, it took us nine round-trips to get back our first query result. That's one round-trip to establish a TCP connection; one to set up the WebSocket; one to check if Postgres supports TLS; two for TLS itself; three for Postgres connection and authentication; and one for the query and its result.

This is just one more round-trip than we'd expect using ordinary Postgres over TCP: that's the one devoted to establishing a WebSocket on top of the TCP connection.



Nine round-trips feels like a lot. How bad it turns out to be depends mainly on how far our packets have to travel. A round-trip between nearby US states might take 10ms, say, giving a total network latency of around 90ms. A round-trip from Europe to the US west coast and back, on the other hand, could take upwards of 100ms. In that case, we could be waiting *a whole second* or more for the first query result.

We knew we needed to do better than this.

Low-hanging fruit: an upgrade to TLS 1.3

The earliest versions of our serverless driver handled TLS via a C library compiled to WebAssembly. The first library we got working this way was **BearSSL**. BearSSL was a pleasure to work with, but it only supports up to TLS 1.2.

TLS 1.2 requires two round-trips to establish a new connection, where TLS 1.3 usually requires only one. That's because a TLS 1.3 client **assumes that the server will support one of its proposed ciphers**, and the server usually does. Switching out BearSSL in favor of **WolfSSL**, which supports TLS 1.3, thus saved us our first round-trip.

In the current version of the driver, we saved some serious weight — and also some compatibility headaches around loading WebAssembly — by switching out WolfSSL too. Instead, by default, we've moved encryption one level down the stack, so that we now run an unencrypted Postgres session over a secure `wss:` WebSocket. As far as we're aware, the platforms we run on all support TLS 1.3 for outgoing HTTPS and secure WebSocket connections.

Alternatively, our driver also offers an experimental mode where the Postgres session itself remains encrypted. That now relies on a pure-JavaScript TLS 1.3 client using **SubtleCrypto** (courtesy of **subtls**). One advantage of this option is that the WebSocket proxy is much easier to set up: it doesn't need to speak TLS or keep certificates updated, and connections remain secure irrespective of where the proxy lives.

For those keeping score, moving to TLS 1.3 in any of these ways brings the round-trip total down to eight.

Eliminating the `SSLRequest` round-trip

Every TLS-secured Postgres connection **begins with an `SSLRequest` message**. The client sends the magic 4-byte value `0x04 d2 16 2f`, and the server responds with either `0x53` (an 'S', meaning SSL/TLS is supported) or `0x4e` (an 'N', meaning it's not). That's right: **Postgres speaks Spanish** here!

This little dance burns a round-trip, of course. In Neon's case, we **know** the Postgres server at the other end supports TLS, so this particular round-trip is 100% wasted time.

Moving encryption down a level to the WebSocket, as described in the previous section, has the happy side-effect of eliminating this round-trip. We don't have to ask Postgres if it supports TLS, because Postgres no longer even needs to know we're using it.

But what about our alternative mode that does things the original way, running an encrypted Postgres session through an unencrypted WebSocket? It turns out we can save the round-trip there too.

This time, we do it by **pipelining** the `SSLRequest` with the first TLS message, the Client Hello. In other words, we send one message straight after the other, without waiting for a reply.

This works just fine when connecting to Neon's own Rust-based Postgres proxy. But unfortunately this strategy is not applicable more widely, since Postgres itself can't handle it. Postgres reads its TCP socket dry before handing it on to OpenSSL to negotiate encryption, and the connection therefore hangs, waiting on a Client Hello that's already been sent but is now sitting in the wrong buffer. The connection pooler **PgBouncer** behaves the same way.

I have a proof-of-concept patch that changes this, enabling `SSLRequest` pipelining in Postgres. But for now I'm sitting on it, because the **Postgres devs are discussing an even better solution**: omitting the `SSLRequest` entirely, and allowing Postgres connections to begin with an ordinary TLS Client Hello. A point in favor of this solution is that we then won't have to deal with the potentially fiddly business of ignoring the server's 'S' response in the middle of a TLS negotiation.

Fingers crossed this change makes it into a Postgres update soon. In the meantime, our driver's `SSLRequest` pipelining behavior is controlled by the configuration option `pipelineTLS`, which defaults to `true` for Neon hosts and `false` otherwise.

We've now brought the round-trips down to seven.

Faster Postgres authentication

You might have noticed above that it takes a full three round-trips to introduce and identify ourselves to the Postgres server. Even worse, additional latency is caused by having to calculate 4096 SHA-256 hashes along the way. We can definitely speed things up here.

SCRAM-SHA-256 (or from here on in, SCRAM) is a modern authentication scheme designed to raise the time and/or cost of a brute-force password attack, much like **PBKDF2, bcrypt, scrypt or Argon2**. SCRAM is **specifically intended** to take about 100ms of CPU time.

Unfortunately, this just isn't appropriate for connections from a serverless environment. Quite apart from the latency, it will blow your CPU budget out of the water. Currently on Cloudflare Workers, for example, the free plan is limited to 10ms of CPU time while the cheapest paid plan gets 50ms. An authentication scheme that's designed to take 100ms of CPU time is a non-starter.

Happily, Neon needs SCRAM less than many Postgres operators. That's because we generate and support only random passwords, and these are immune to **dictionary attacks**. To make our passwords harder to brute-force, rather than slowing down password verification, we can increase the search space by simply making them longer.

Replacing SCRAM with simple password auth (which is still protected by TLS encryption) saves us the challenge-response round-trip.

And that takes the round-trip count down to six.

Postgres pipelining

Now that we're using password auth, our actual Postgres interactions involve three round-trips: (1) client sends startup message, Postgres requests password auth; (2) client sends password, Postgres says OK, you can send a query; and (3) client sends a query, Postgres returns the result.

For none of these round-trips except the last one — the query and result — is it actually necessary to wait for the server's response before proceeding. We can send these three messages all at once. This pipelining cuts out a further two round-trips, bringing the previous six down to a total of four.

We can compare the round-trips required before and after pipelining in the Network pane of Chrome's developer tools if we run the driver there. (In each case the final outgoing message instructs Postgres to close the connection).

Original

Pipelined

Pipelining is activated when the driver's `pipelineConnect` option is set to `"password"`. This is the default for Neon hosts, where we know password authentication will be offered, and it can be set manually for other hosts.

If you're interested in reproducing this behavior using the standard node-postgres `pg` library, rather than our serverless driver, you can adapt the overridden `connect()` method **in our `Client` subclass**.

TCP_NODELAY

Pipelining these three messages into one very clearly reduced latencies when connecting from southern England to a Neon database in Frankfurt (AWS eu-central-1), which is what I was doing up to now. But to check on any non-network sources of latency, at this point I started running some tests in a Lightsail server in the same AWS region, **using TigerVNC**.

This testing turned up something rather interesting. When connecting to the database from really nearby, pipelining actually seemed to make things worse. Unpipelined, we saw a response to each message arrive within 2 or 3ms. But when the three messages were packaged up together, we saw a pause between the first and second response of up to around 50ms. Something definitely wasn't right here.

My colleagues quickly tracked the issue down to **Nagle's algorithm**, and fixed it by **setting TCP_NODELAY in the WebSocket library** we use within our proxy. Pipelining now gave us a bigger win everywhere, improving things over short distances as well as longer ones.

The final four

The four remaining round-trips are as seen below.

What's next?

Since launching our serverless driver, we've made a number of other speed improvements. Originally, we ran a single WebSocket-to-TCP proxy separate from our main Postgres proxy. That split every network round-trip in two (and potentially sent them halfway round the world in the process).

Instead, our main Postgres proxy now accepts WebSocket connections for itself, which makes things rather snappier. In addition, we've introduced some additional caching in the proxy, which eliminates a hidden round-trip within our back-end on most requests.

Four round-trips looks like the lowest we can go with the technologies we're currently using. But we're hopeful that WebSockets over QUIC in HTTP/3 (**RFC 9220**) and/or **WebTransport** will let us drop that to

three (or even two?) before very long. QUIC effectively combines the TCP and TLS round-trips into one, cutting a round-trip at the start of the interaction.

And at the other end of the interaction, Postgres **has support for pipelining independent queries**. This **isn't yet available in node-postgres**, on which our serverless driver is based, and we're not sure how commonly it will be useful in a serverless context. But if you think that would help you out, please do **let us know**.

George MacKerron
Typescript Developer

Subscribe to Newsletter

Your email...

NEON

Made in SF and the World