And then we posted it to Hacker News.

If you listen to, well, pretty much anyone rational, they will tell you in no uncertain terms that the last thing you ever want to do is put your SQL Database on the internet. And even if you're crazy enough to do that, you certainly would never go post the address to it on a place like Hacker News. Not unless you were a masochist anyway.

We did it though, and we're not even a little bit sorry about it.

The Idea

QuestDB has built what we think is the fastest Open Source SQL Database in existence. We really do. And we're pretty proud of it, in fact. So much so that we wanted to give anyone that wanted the opportunity a chance to take it for a spin. With real data. Doing real queries. Almost anyone can pull together a demo that performs great under just the right conditions, with all the parameters tightly controlled.

But what happens if you unleash the hordes on it? What happens if you let anyone run queries against it? Well, we can tell you, now.

What It Is

First off, it's a SQL-based Time Series database, built from the ground up for performance. It's built to store and query verylarge amounts of data.

We deployed it on an AWS `c5.metal` server in the London, UK datacenter (sorry all you North Americans, there's some built-in latency due to the laws of physics). It was configured with 196GB of RAM, but we were only using 40GB at peak usage. The `c5.metal` instance provides 2 24-core CPUs (48 cores), but we only used one of them (24 cores) on 23 threads. We really weren't using anywhere *close* to the full potential of this AWS instance.

The data is stored on an AWS EBS volume that provides SSD access to the data. It's not all in memory.

The data is the entire NYC Taxi Database plus associated weather data. It amounts to 1.6 billion records, weighing in at about 350GB of data. That's a lot. And it's too much to store in-memory. It's too much to cache.

We provided some clickable queries to get people started (none of the results were cached or pre-calculated) but we essentially didn't restrict the kinds of queries that users could run.

The Hacker News Post

A few weeks ago, we wrote about how we implemented SIMD instructions to aggregate a billion rows in milliseconds [1] thanks in great part to Agner Fog's VCL library [2]. Although the initial scope was limited to table-wide aggregates into a unique scalar value, this was a first step towards very promising results on more complex aggregations. With the latest release of QuestDB, we are extending this level of performance to key-based aggregations.

To do this, we implemented Google's fast hash table aka "Swisstable" [3] which can be found in the Abseil library [4]. In all modesty, we also found room to slightly accelerate it for our use case. Our version of Swisstable is dubbed "rosti", after the traditional Swiss dish [5]. There were also a number of improvements thanks to techniques suggested by the community such as prefetch (which interestingly turned out to have no effect in the map code itself) [6]. Besides C++, we used our very own queue system written in Java to parallelise the execution [7].

The results are remarkable: millisecond latency on keyed aggregations that span over billions of rows.

We thought it could be a good occasion to show our progress by making this latest release available to try online with a pre-loaded dataset. It runs on an AWS instance using 23 threads. The data is stored on disk and includes a 1.6billion row NYC taxi dataset, 10 years of weather data with around 30-minute resolution and weekly gas prices over the last decade. The instance is located in London, so folks outside of Europe may experience different network latencies. The server-side time is reported as "Execute".

We provide sample queries to get started, but you are encouraged to modify them. However, please be aware that not every type of query is fast yet. Some are still running under an old single-threaded model. If you find one of these, you'll know: it will take minutes instead of milliseconds. But bear with us, this is just a matter of time before we make these instantaneous as well. Next in our crosshairs is time-bucket aggregations using the SAMPLE BY clause.

If you are interested in checking out how we did this, our code is available open-source [8]. We look forward to receiving your feedback on our work so far. Even better, we would love to hear more ideas to further improve performance. Even after decades in high performance computing, we are still learning something new every day.

[1] https://questdb.io/blog/2020/04/02/using-simd-to-aggregate-b…

[2] https://www.agner.org/optimize/vectorclass.pdf

[3] https://www.youtube.com/watch?v=ncHmEUmJZf4

[4] https://github.com/abseil/abseil-cpp

[5] https://github.com/questdb/questdb/blob/master/core/src/main…

[6] https://github.com/questdb/questdb/blob/master/core/src/main…

[7] https://questdb.io/blog/2020/03/15/interthread

[8] https://github.com/questdb/questdb

And then we posted the link to the live database.

And sat back.

And watched the incoming traffic.

And tried not to panic.

What Happened

We saw a lot of traffic. I mean a *lot* of traffic.

Somewhere over 20,000 unique IP addresses.

Excluding simple `show` queries, we saw 17,128 SQL queries with 2,485 syntax errors in the queries. We sent back almost 40GB of data in response to the queries.

Someone in the HN comments suggested that column stores are bad at joins, which led to someone coming in and trying to join the table to itself. Ordinarily, this would be a stunningly bad decision.

The result was … not what they were expecting:

Yeah, that's 2,671,914,914,060,072,000 rows. In 69ms. That's a lot of results in a very short amount of time. Definitely not what they were expecting.

We saw only a couple of bad actors try something cute:

```
2020-06-23T20:59:02.958813Z I i.q.c.h.p.JsonQueryProcessorState [8536] exec [q='drop table trips']
2020-06-24T02:56:55.782072Z I i.q.c.h.p.JsonQueryProcessorState [6318] exec [q='drop *']
```

But those didn't work. We may be crazy, but we're not naive.

What we learned

It turns out that when you do something like this, you learn a lot. You learn about your strengths and your weaknesses. And you learn about what users want to do with your product as well as what they will do to try to break it.

Joining the table to itself was one such lesson. But we also saw a lot of folks use a `where` clause, which caused fairly slow results. We were not entirely surprised by this result, as we are aware that we have not done the optimizations on that path to yield the fast results we want. But it was a great insight into how often it is used, and how people use it.

We saw a number of people use the `group by` clause as well, and then be surprised that it didn't work. We probably should have warned people about that. In short, `group by` is automatically inferred, so it's not needed. But since it's not implemented at all, it causes an error. So we're looking at ways to handle that.

Conclusions

It seems that the vast majority of the people that tried the demo were very impressed with it. The speed is truly breathtaking.

Here are just a few of the comments we got in the thread:

I abused LEFT JOIN to create a query that produces 224,964,999,650,251 rows. Only 3.68ms execution time, now that's impressive!

Very cool. Major props for putting this out there and accepting arbitrary queries.

Very impressive, i think building your own (performant) database from scratch is one of the most impressive software engineering feats.

Very cool and impressive!! Is full PostreSQL wire compatibility on the roadmap? I like postgres compatibility 😁

(Yes, full PostgreSQL Wire Protocol is on the roadmap!)

Mind blowing, did not know about questDB. The back button seems broken on chrome mobile

Yes, the demo did break the Back button. There's an actual reason for that, but it is true, for now.

Try It Yourself

Want to try it yourself? You've read this far, you really should! got to http://try.questdb.io:9000/ to give it a whirl.

We'd love to have you join us on our Community Slack Channel, give us a Star on GitHub, and Download and try it yourself!