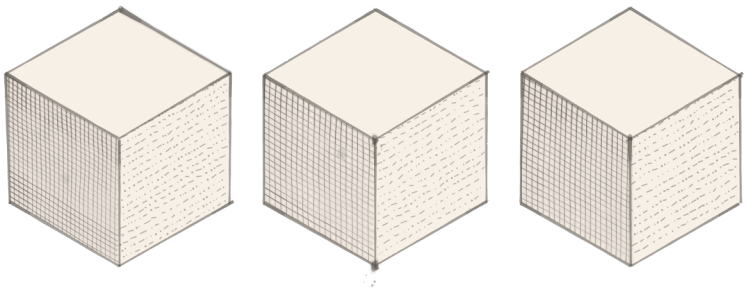
Fast drilldown dashboards from a single Parquet file

One 40MB Parquet data cube,
an 18KB reader, an R2 bucket, and
a few unassuming http range requests.

Aug 21, 2026



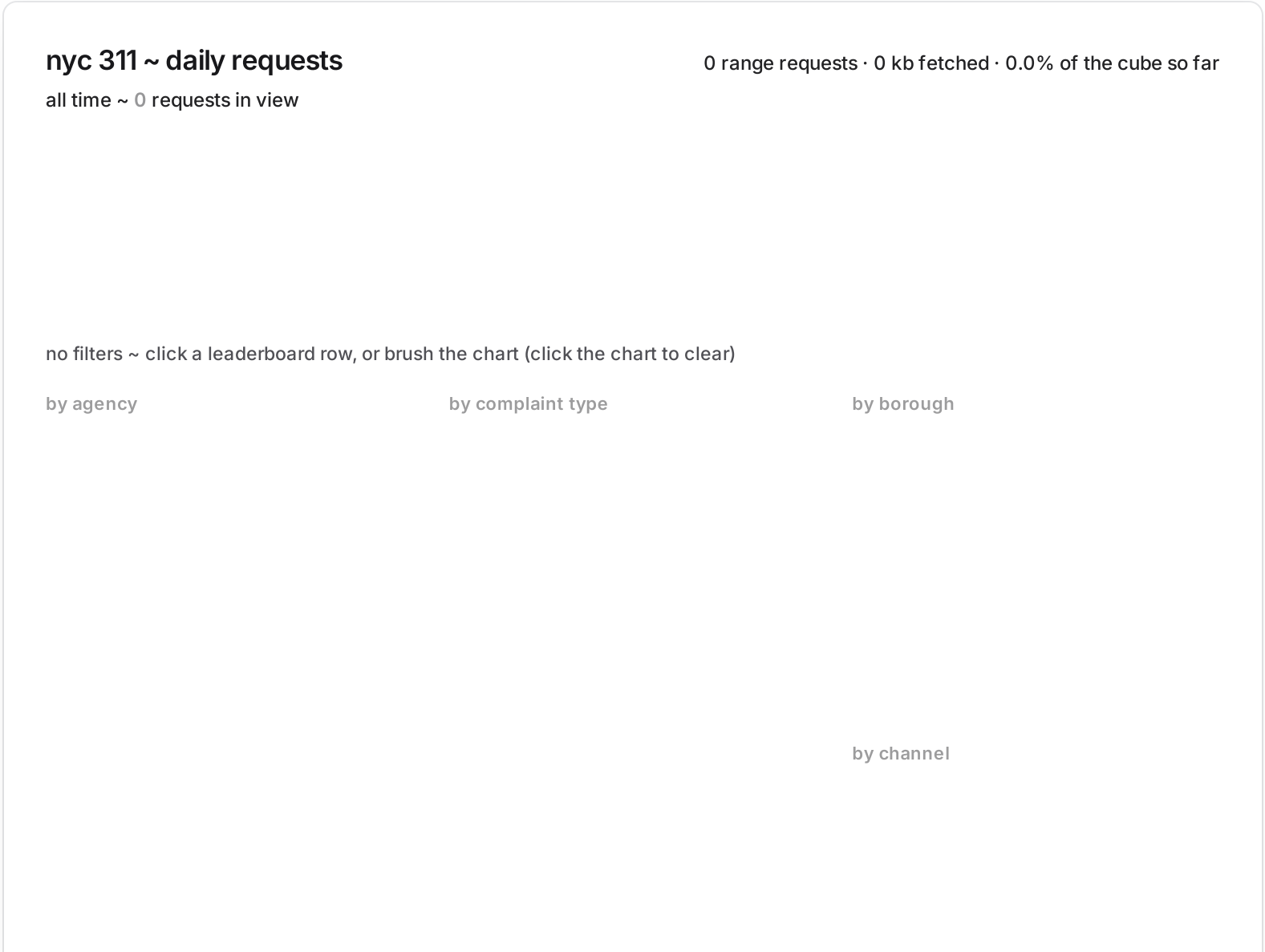
Every month brings a new eruption of clever uses for object storage, easily the most volcanically active corner of non-AI software infrastructure on earth. The most recent lava bomb was Vicent Martí's writeup of Cursor Origin's S3 + WAL approach to managing Git repositories at scale. It's a masterpiece of technical writing, unlike this post. I'll admit that even before reading it, I was daydreaming about a totally different kind of task where object storage probably just *works*, this time for customer-facing analytics dashboards. A friend of mine has customer usage data in Iceberg on R2, and wants to show his users some basic charts with filters. He told me he didn't want to add any more vendors, which ruled out MotherDuck, the cloud-hosted DuckDB database company where I work.

Well, in analytics, when all you have is object storage, everything looks like a range request. We could probably just roll this kind of data up into a Parquet-backed data cube, and use range queries to pull the aggregate rows we need for the dashboard. Normally I'd turn to DuckDB-Wasm for this kind of BI-shaped workload, but then I realized that Hyparquet, a javascript Parquet reader that runs in the browser, can do the same range scans in 18kb rather than multiple mb and a dedicated worker. *With that, you can serve a real*

drilldown dashboard with neither a database nor a query engine. The cube could even be tens (or hundreds) of MB, since a correctly laid-out file means you only ever read a few small slices of it at a time. You just need a data pipeline to produce the cubes ~ which is also, it turns out, where all the actual money goes when you *do* have a real analytical database.

The heresy was too good to pass up. To test it, I took the well-known NYC 311 service requests dataset I had on my computer ~ about 34 million rows at the request level, 15 or so years ~ and rolled it up into a 40MB Parquet cube with filters for city agency, complaint type, submission type, and borough, plus a single creation-time column for the time series. Then I stuck it on R2. 40MB is big enough to feel the pain of downloading the whole thing.

The demo dashboard below reads directly from that file using Hyparquet. The bytes pass through a small Cloudflare Worker on the way, because the free r2.dev URL is rate-limited. To be honest, I was surprised how fast new data loads, given that it forgoes both a real database and a powerful query engine. The UI does all of the actual reading, and it is lightweight enough to embed directly in this post without hurting the page load. The real complexity is almost entirely offloaded to the data cube layout. Try scrubbing the chart or clicking on the rows of the leaderboards.



So, how does this dashboard work?

A dashboard like this one is designed to answer a bounded set of analytical questions ~ requests per day, requests per day for one agency, all-time totals by borough. Each question can be answered by `GROUP BY` queries, so we can precompute them all ahead of time and save each result as its own small table, called a *grouping set*. Stack all of the grouping sets in one Parquet file, one section per set, and you have a *data cube*. A grouping set is only useful if it either enables a question to be answered, or reduces the latency of pulling the data. This file has both kinds. The all-time totals feed the leaderboards, and a daily grouping set for every combination of filters provides the data for the line chart. The weekly and yearly grouping sets reduce the number of rows scanned that results from brushing the chart. The same totals *could* be summed from daily rows, but there are fewer rows to fetch if we precompute by weeks and years.

The file now holds the grouping sets that render the dashboard, but the browser still has to pull out just the rows it needs. Two features of the Parquet format make that possible. A Parquet file is divided into *row groups* of a few tens of thousands of rows, and it ends with a footer that contains metadata about the byte ranges of row groups and the min/max values of each column inside it. The client reads the footer once. Each query then uses the min/max values to pick the row groups that could match, fetches those byte ranges, and aggregates the rows in the browser.

The low latency in the dashboard requests is due to how the rows in the Parquet file are sorted and scanned. If the rows of the file were randomly ordered, each row group's min/max values would span nearly the full range of each column, and a query would have to read most of the file just to fetch a small percentage of rows. Instead, the rows of each grouping set are sorted by the columns its queries filter on. The matching rows thus usually make up a contiguous stretch of the file, and the min/max statistics enable the reader to ignore the rest of the row groups. That is why clicking NYPD in the agency leaderboard reads about 260KB out of the 40MB file rather than the whole file. Below is the actual layout of the file in terms of bytes and grouping sets:

row groups	grouping set	rows	size
	totals feed the "requests in view" total and the four leaderboards	**831.1k**	**1.7mb**
	all time read when no date range is brushed	4.8k	103kb
	by week read when brushed: the leftover weeks at the range's edges	796.6k	1.3mb
	by iso year read when brushed: the whole years in the range's middle	29.7k	272kb
	daily · no dimensions	**5.0k**	**171kb**

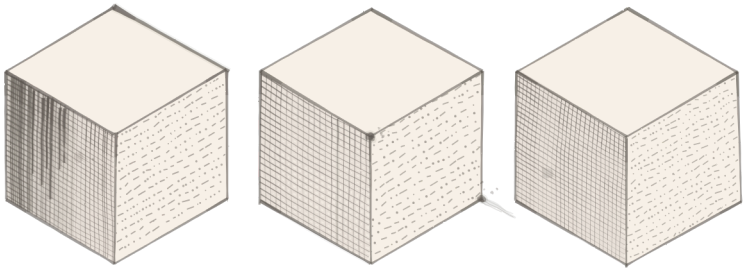
grouping set	rows	size
draws the line chart when no filters are active		
daily · one dimension draws the line chart when one filter is active	**770.3k**	**2.4mb**
channel	22.7k	171kb
borough	30.0k	330kb
complaint	635.6k	1.5mb
agency	82.0k	383kb
daily · two dimensions draws the line chart when two filters are active	**4.5m**	**10.6mb**
borough + channel	127.7k	401kb
complaint + channel	1.1m	2.6mb
complaint + borough	2.1m	4.5mb
agency + channel	198.0k	640kb
agency + borough	358.5k	997kb
agency + complaint	643.0k	1.5mb
daily · three dimensions draws the line chart when three filters are active	**7.3m**	**15.8mb**
complaint + borough + channel	3.3m	6.8mb
agency + borough + channel	804.0k	1.9mb
agency + complaint + channel	1.1m	2.4mb
agency + complaint + borough	2.1m	4.6mb
daily · all four dimensions draws the line chart when all four filters are active	**3.3m**	**6.6mb**
footer · the index byte ranges and min/max statistics for every section; read first, once	—	**195kb**

This setup works under two conditions. The combinatorics of your charts and filters have to stay small, and your pipeline has to rebuild each customer's file fast enough to meet the update cadence. Most usage and billing pages meet both. They are a fixed set of charts ~ events over time, counts or sums by hour or by day, a few filters or leaderboards ~ over data that updates on a coarse schedule rather than in realtime, for their sake as much as yours. From the perspective of latency, the cube size doesn't matter, but you'll want it to be somewhat small anyway since you're regenerating one per customer on a schedule.

The time grain is clearly dominant in my example, since the daily sections account for most of the bytes of the file. Cardinality is the other multiplier ~ complaint type has 485 distinct values, and every large section in the diagram contains it. In fact, choosing a daily grain for the line chart made the file about 7x larger than the weekly equivalent (5.6mb). Still, the daily grain did not meaningfully impact the latency of the range requests, since any interaction only ever reads a few row groups. And for this case, it's nice to see a big

single-day spike, since a big uptick in service requests can happen in a single day because of major events like hurricanes or blizzards.

Dashboards such as the one above work well for *distributive* and *algebraic* aggregations, which can be computed in pieces and then combined before visualizing. Think of sums, counts, maxes, and averages. Making this setup work for *holistic* aggregations (ones that require knowledge of the distribution before achieving a final filtered aggregate) have both exact and approximate solutions. I'll leave that as an exercise to the reader and their favorite agent.



Range requests over a carefully laid-out file have plenty of prior art. **PMTiles** packs a tileset into one file that clients read via range requests over http. It works because the tiles are laid out in the file along a Hilbert curve, so the tiles for a given map view sit near each other in the file and can be fetched in a few coalesced range requests. **And of course, the well-known SQLite-over-HTTP writeup** proved the mechanic works even for B-trees. The only real requirement is a host that serves byte ranges over http ~ R2, S3, GitHub Pages, a plain nginx box.

My favorite part is that this approach shifts the complexity "left" all the way to the data pipeline. The layout is decided beforehand, so by the time a user clicks on a leaderboard or scrubs a time series chart, the client only has to fetch the right rows and sum them. As for the pipeline, for most customer-facing dashboards, a 10mb per-customer cube falls out of a DuckDB `GROUP BY GROUPING SETS` statement. Naturally I'm biased, but if I were building the pipeline, I would go with **MotherDuck** (a cloud-hosted DuckDB database & query engine) and **Flights** (a MotherDuck feature where you can schedule python jobs for this kind of stuff). The setup is pretty straightforward with your agent of choice. Regardless of how you do it, for my friend, who's a data engineer, pushing the complexity to the pipeline is pitch-perfect *déformation professionnelle*.

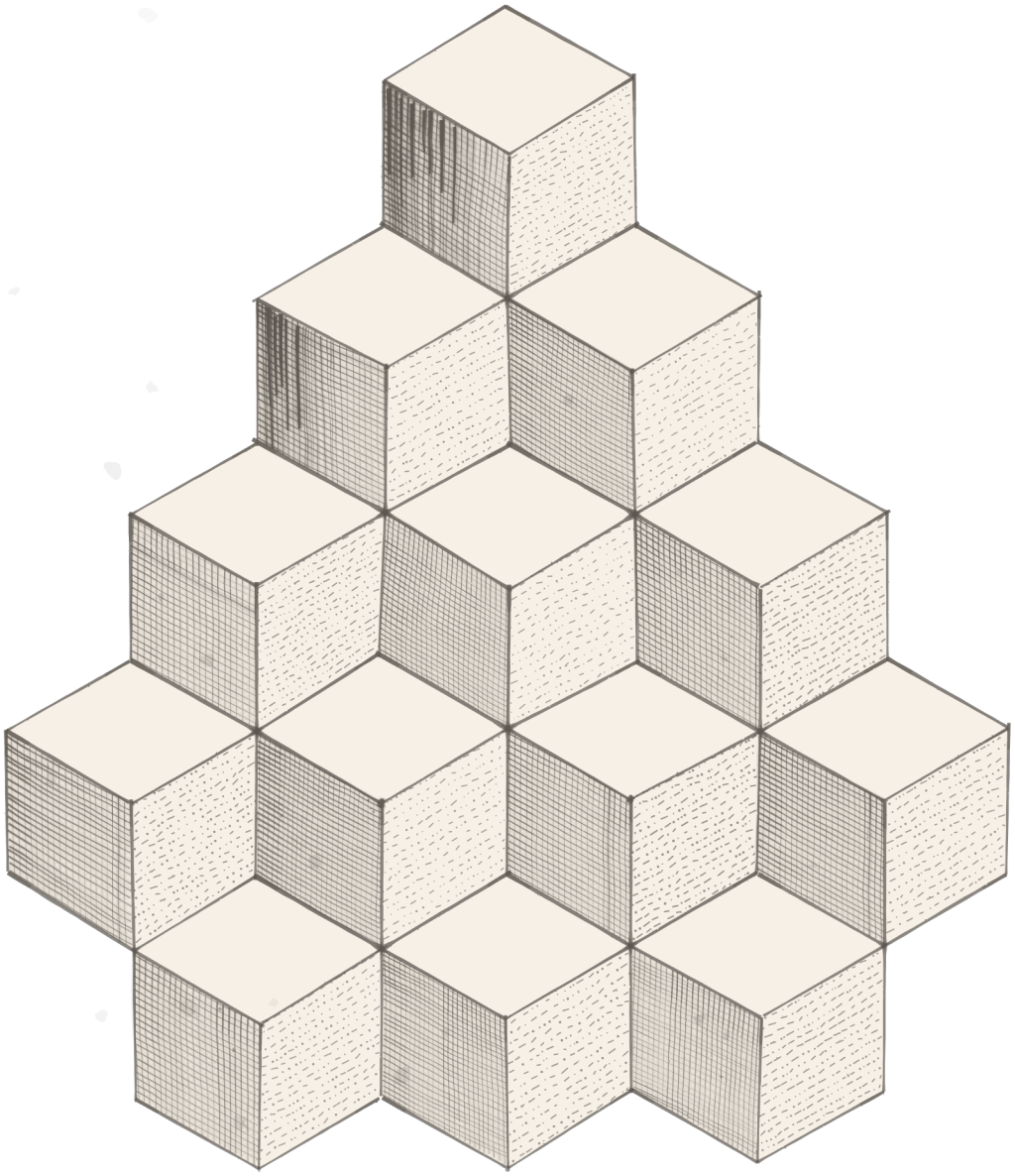
One file per customer also makes auth refreshingly boring. Access control amounts to a signed URL for that customer's file, or a tiny Worker that checks the session.

Given that R2 has free egress, the pipeline is also the thing that costs actual money. Writes cost **$4.50 per million** (12.5x the price of reads), and you pay one write per customer per rebuild, regardless of file size (a 1MB cube and a 40MB cube cost the same to upload). Take 10,000 customers. Each rebuild replaces the files, so storage is flat: 10MB cubes make 100GB, about $1.50/month; 40MB cubes make 400GB, about $6/month. Rebuilding every file once a day is 300k writes a month, or about $1.35; rebuilding hourly is 7.2M writes,

about $32; rebuilding every five minutes for a month is 86M writes, about $389. Thankfully, Iceberg snapshot diffs tell you exactly which customers have new data, so it's easy to only rebuild the cubes with new activity.

Even still, let's say you update every 5 minutes and every customer has activity in that window (again, not very likely). For a single use-case like this one, $389/mo for 10k customers is probably cheaper across the board than standing up new infra, and it is almost certainly simpler. And both the economics and the user experience have changed very recently: egress fees wouldn't have quite killed this idea, but they've probably discouraged people from experimenting this way. The same setup on S3 comes out only about 20% more expensive overall ~ roughly $20 a month of egress at a million queries, and a million queries is more traffic than most customer dashboards will ever see.

My *other* favorite part is the radically thin implementation. An 18kb javascript reader and a byte layout that does the database work for you. What a world!



discuss this post on Hacker News