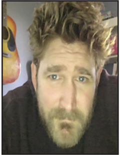
iVoyeur
Flow

DAVE JOSEPHSEN



Dave Josephsen is a book author, code developer, and monitoring expert who works for Sparkpost. His continuing mission: to help engineers worldwide close the feedback loop.
dave-usenix@skeptech.org

Little Mission Creek roars and tumbles and thrashes against its banks, along with, it seems, every watershed in all of western Montana. The Clark Fork in Missoula, the Gallatin in Bozeman, the Great Missouri River in Helena, and the Yellowstone River a dozen miles south from where I sit—they've all crested their banks and tested their spillways in the last several weeks.

But Little Mission Creek is my home, though I barely recognize the violent torrent it has become this spring. Watching it churn impatiently about my legs, it's easy to forget what an arid place this is. The notion of water-rights and violent disputes over creeks like this one have shaped the landscape here every bit as much as the flowing water itself.

I like to sit here at the bank and attempt to imagine how the water flowing past me now will, in roughly 11 minutes, make its way to the head of our valley and join forces with Mission Creek proper—itself busily running north out of the foothills. How in another 20 minutes the water below me will spill crashing into the Yellowstone River and turn east, running for 150 miles into Billings before turning back north, and joining the Missouri just past the North Dakota border.

That's about the extent of my imagination. I can't really wrap my head around the scope of the journey these H_2O molecules are about to make, but that doesn't stop the water in Little Mission Creek. On it flows, heedless of my cognition and indifferent to my doubts, winding halfway across North Dakota before veering back down south to Kansas City, where it turns east again to join the Mississippi in St. Louis before finally making a 700-mile beeline for the Gulf of Mexico at New Orleans.

That's just inconceivable to me. It seems mythical, otherworldly. Someday I'm going to drive it. I'll plan it carefully, taking small roads as necessary to remain as close to the water as possible. Hopefully, I'll get a tangible sense of it—a concrete understanding of what it means to flow like the water in my creek. When I go, I will take some of my creek water with me in a bottle. I'll carry it to the Gulf like a riverbed on wheels and, like an orphan reunited, return it to the Atlantic at the end of my own journey. I wonder if I'll be giving it a head start or delaying its arrival; or maybe it's the journey that matters, not the destination. Maybe when I get there I'll understand.

Data Lake

At work I'm helping out on a project called "The Data Lake." We're all very excited about it. For example, the other day I got a meeting invite whose description read (I promise I'm not making this up): "Break out your data-paddles, because it's time to go data-canoeing in the data-lake." That's how excited we are. Just absolutely dancing-away-with-the-metaphor-in-public excited.

What on earth is a data-lake?

Great question, I'm glad you asked. The data-lake is just a colorful name for a series of S3 buckets. S3 buckets?! What's so great about a bunch of S3 buckets, you ask? Well, it's not so much the storage medium that's cool as what's stored there, how it's stored, and what we can do with it later by way of a few Python scripts and AWS's Athena service. Have you read about schema-on-read, columnar data storage formats, and the rise of the SQL query engines? If not, prepare yourself, because these are the substrate into which the data-lake is carved.

For the entire length of the history of people interacting with databases so far, we have been mapping our data to a schema at write time. Like anal-retentive scriveners whose very nature prevents us from just writing anything down all willy-nilly, we take the raw data in one hand and a description of what the data should look like in the other, and we combine the two, writing the result to disk in a binary, pre-formatted way. Users can subsequently make queries against the data because we have it stored in a schemafied, normalized, queryable format.

Schema-on-read systems, by comparison, map the schema to raw data at query time. That is, the data is not preformatted—it is not "queryable" in the database-sense. It's just bytes sitting somewhere on disk in its native format (JSON, newline-separated lines of text, whatever...). The schema itself is stored as a set of ETL-like (extract, transform, load) instructions (or even a regular expression), which the query engine can use to map the at-rest data into named fields on-demand. So really, there is no "database" in a schema-on-read system. There is just some metadata linking the location of some at-rest data to a schema we can use to parse it when we want to.

Bereft of a proper database to pamper and worship, users instead interact with a *query engine*. When a user makes a query, the query engine finds the data, maps it to the schema in memory, executes the query on the resultant in-memory data blob, and returns the result. When the query engine speaks SQL (most of them speak a dialect of SQL like Presto (https://prestodb.io/)), we call it, unimaginatively, an "SQL query engine."

Why would you *ever* do that?

I know, if you want a database, use a database, right? Well, databases have their own suite of problems, related mostly to getting data *into them*. Engineers often turn to schema-on-read systems to provide an SQL interface to some vast quantity of already at-rest data that would otherwise be too onerous for a traditional database to ingest.

Say, for example, that there's an S3 bucket containing a yottabyte of raw (un-summarized) monitoring check output for every computer ever owned by some corporation since the beginning of time, and you need to query it. You could spend the better part of a month writing custom ETL and using it to get all that data into MySQL, all just to run a couple of queries and then throw it all away, or you could just point your Apache Drill (https://drill.apache.org/) SQL query engine directly at the data.

This sort of ad hoc access to analyze ponderously huge data sets stored across a widely distributed medium is the bread-and-butter use-case for schema-on-read, but those of us who struggle with a preponderance of monitoring data might also find a compelling story herein.

Imagine that you could just flip a switch and enable SQL querying of all of your organizational Nginx logs. What a treasure trove it would suddenly become for managers, engineers, account managers, support personnel...anyone able to formulate an SQL query. A common, self-service interface for anyone with questions like, "What was the 99th percentile response time on the /accounts endpoint this morning?" or "How many people signed up last month?" And you can make it happen without any of the headache of ETL, provisioning, scaling, and managing a proper database or rolling an ELK-style log analysis system.

That's pretty much the data-lake concept in a nutshell: a low-maintenance, self-service SQL interface into timely operational data that you just happen to have lying around anyway.

Keeping Things Low Cost

For the data-lake, our chosen SQL query engine is an AWS-hosted service called Athena (https://aws.amazon.com/athena/). It's easy to use, wholly hosted, and it obviously works flawlessly with data stored in S3. You only pay for the queries you make, but here's the rub: it costs $5 per terabyte of data scanned by each query.

How is THAT going to work?!

I know. You have a LOT of data (so do I). So the game becomes a process of getting the answers you need from the data set with the minimum amount of actual reading data. There are two hacks that make our data-lake cheap enough that so far, we aren't worried about restricting access to it.

The first is data partitioning. It's possible to write the data to S3 in chunks, labeling these in such a way that Athena can intuit the chunk names. A very common partitioning scheme, which many log-writers support without even knowing it, is partitioning by year/month/day. Simply write the data to S3 using year/month/day "directories" (there aren't really any directories in S3), and identify these to Athena as partitions. Then make queries like this one, which I just used to count the number of API calls made by a customer in the first 10 days of April:

```
SELECT COUNT(*) FROM "data-lake.nginx-json" where
"customer_id"='1234' AND "partition_0"='2018' AND
"partition_1"='04' AND "partition_2"
in('01','02','03','04','05','06','07','08','09','10');
```

It seems stupid-obvious, but without partitioning you'll too often find yourself reading more data than you want. You can read more about data-partitioning for Athena at the AWS support site (https://docs.aws.amazon.com/athena/latest/ug/partitions.html).

The second hack to minimize the amount of data you scan is to use a columnar data storage format. I know I said you didn't need to pre-format your data, and you don't. But if you're building a semi-permanent log-query solution on Athena, like we are, and want to save a considerable amount of money, I'd highly recommend running your queries against a columnar-transformed copy of your logs.

So what's a columnar data store? Well, start by imagining a typical database as a spreadsheet, where you have a row of headers followed by rows of data records and where each column represents a schema entry in that data record. You know what it looks like:

```
first,  last,  middle, num, street, state,  pet
dave, josephsen, j, 11, may street, MT, cat
jill, gomez, f, 114, epic road, CA, goldfish
jose, cardona, r, 210, turbine ct, TX, hedgehog
```

A columnar data store is pretty much a broken spreadsheet. We take all the column entries and store them on top of each other, along with a small header which maps the line numbers of each column. I'm simplifying things for instructional purposes but it basically looks like this:

```
first 1, last 4, middle 7, num 10, street 13, state 16, pet 19
dave
jill
jose
josephsen
gomez
cardona,
j
f
r
11
114
210
may street
epic road
turbine ct
MT
CA
TX
cat
goldfish
hedgehog
```

Now imagine what happens when I make a query like

```
select * where middle="j"
```

In a traditionally laid out record-per-line text file, the query engine needs to traverse and parse essentially the entire file, ingesting each record to search for the `middle` field, string compare it against j, and then return the whole line if it matches.

With a columnar format, we can use index numbers to look around rather than scanning the data itself. First, we parse the header for the line-number of the `middle` field, and then we simply seek directly down to line 7, comparing just the individual bytes from each record's middle column, and return the matching records. The records we can also reconstruct from line-number offsets without having to actually scan the data (e.g., the offset between line 7 and the matching record (line 7) is 0. So we reconstruct the entire record by walking the header and forming the union of lines 1+0, 4+0, 7+0, and so on...).

This is a *way* more efficient means of querying data, which translates to both faster responses and smaller Amazon bills.

Flows

Okay, so what have we learned?

- Schema-at-read query engines can effectively query at-rest data using SQL-like syntax.
- Most watershed from western Montana winds up in the Gulf of Mexico.
- You can roll your own query-engine with something like Apache Drill or use a hosted one like Amazon Athena.
- You can query raw data but it's expensive and slow.
- If you transform it into structured data and store it in a columnar format like Parquet (https://parquet.apache.org/), things get orders-of-magnitude faster and cheaper.

But how do we get our log data from text files on individual server instances into Parquet-formatted data in the data-lake? Well, a detailed description of our ingestion pipeline will have to wait until next time, but the short answer is—rather obviously—it *flows* there. Nginx to Rsyslogd to Fluentd to Kinesis to EMR, like rivers winding, maybe our data-lake metaphor isn't really as absurd as it sounds at first. Deeper than a pond and yet perhaps not so final a destination as an ocean, our humble Data Lake is already solving pretty big observability conundrums for us internally, so maybe our excited overuse of metaphor is similarly justifiable. Anyway, grab your hip-waders, because next time we'll wade into the stream and measure the flow.

Take it easy.