Why local state is a fundamental primitive in stream processing

What do you get if you cross a distributed database with a stream processing system?

By jay kreps. July 31, 2014

One of the concepts that has proven the hardest to explain to people when I talk about Samza is the idea of fault-tolerant local state for stream processing. I think people are so used to the idea of keeping all their data in remote databases that any departure from that seems unusual.

So, I wanted to give a little bit more motivation as to why we think local state is a fundamental primitive in stream processing.

Nottawasaga River (source: BiblioArchives)

What is state and why do you need it?

An easy way to understand state in stream processing is to think about the kinds of operations you might do in SQL. Imagine running SQL queries against a real-time stream of data. If your SQL query contains only filtering and single-row transformations (a simple `select` and `where` clause, say), then it is stateless. That is, you can process a single row at a time without needing to remember anything in between rows. However, if your query involves aggregating many rows (a `group by`) or joining together data from multiple streams, then it must maintain some state in between rows. If you are grouping data by some field and counting, then the state you maintain would be the counts that have accumulated so far in the window you are processing. If you are joining two streams, the state would be the rows in each stream waiting to find a match in the other stream.

Get O'Reilly's weekly data newsletter

I like examples!

Okay, I'll give you some concrete examples.

First, aggregation. Let's say you are a news site and you want to be able to rank articles by click-through rate. You would process the stream of clicks and impressions, and record the count for each article in a given time window.

Now let's talk about joining things. In a stream processing system, there are different types of joins. For example, you might want to join an incoming stream of ad clicks with a stream of ad impressions. These two streams are somewhat time aligned, so you might only need to wait for a few minutes after the impression for the matching click to occur. This is called a streaming join. Another type of join is for "enrichment" of an incoming stream — perhaps you want to take an incoming stream of ad clicks and join on attributes about the user (perhaps for use in downstream aggregations). In this case, you want an index of the user attributes so that when a click arrives you can join on the appropriate user attributes.

O'REILLY
Data Newsletter

1. Predictive modeling in regulated industries
Here's how to strike a balance between accuracy and interpretability when you're using machine learning models in regulated industries.

Related resources:

- How the machine learning wave is changing the way organizations look at analytics (free webcast)
- Finance-related sessions at Strata + Hadoop World in San Jose
- *Data Science, Banking, and Fintech* (free report)

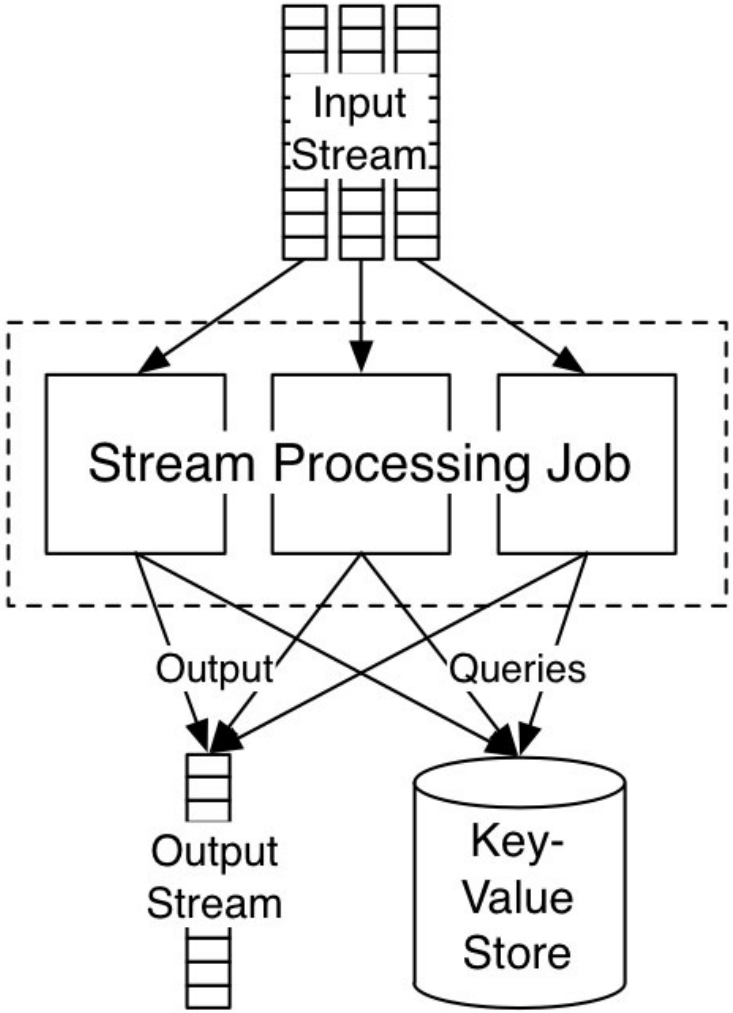
Email Subscribe

We protect your privacy.

Remote state

Let's start by understanding how you can use an external database to handle remote state in stream processing, then I'll contrast this with local state.

A common pattern is for the stream processing job to take input records from its input stream, and for each input record make a remote call to a distributed database. This looks like this:



As you can see, the input stream is partitioned over multiple processors, each of which query a remote database. And, of course, since this is a *distributed* database, it is itself partitioning over multiple machines. One possibility is to *co-partition* the database and

the input processing, and then move the data to be directly co-located with the processing. That is what I am calling "local state," which I will describe next.

Local state

Local state is just data that is kept in memory or on disk on the machines doing the processing. The job can query or modify this data in response to its input. The simplest way of maintaining state might be a simple local key-value store.

We'll talk about some advantages of this in a second, but first let's talk about the obvious problem: what if the machine fails?

Fault-tolerant local state

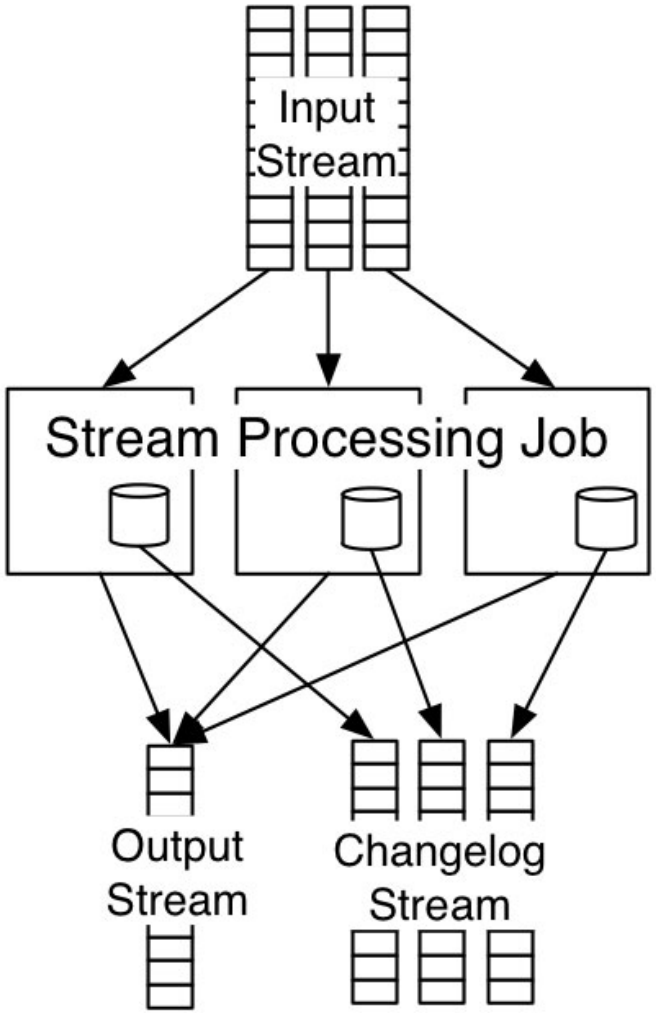
At Samza, we make local state changes fault tolerant by modeling them as just another output log — a commitlog or changelog. This log records each change to the local state of the job. This looks something like this:

PODCAST

O'Reilly Data Show Podcast

The O'Reilly Data Show Podcast explores the opportunities and techniques driving big data and data science.

Subscribe



As the job modifies its local store, it logs out these changes to a Kafka topic. This Kafka topic can be used to replay the changes and restore the local state if the machine fails and the process needs to be restarted on a new host. The log is periodically compacted by

removing duplicate updates for the same key to keep the log from growing too large. This facility allows high-throughput updates as the changelog is just another Kafka topic and has the same write-performance Kafka does. Reads are equally high performance, as data is stored locally in memory or on disk.

The more state you have, the longer the restore time will be, but keeping, say, a few GBs per process is very practical and still reasonably quick to restore. Since you can horizontally scale the number of processes across a shared cluster, this means you can horizontally scale the total state the job is maintaining.

This model of *state changes*, as themselves being a kind of stream or log, is pretty nice. In fact, in many cases where the goal is to join together many different input streams, the changelog is also the only output of the job, and subscribed to by other downstream jobs. I've written more about this kind of use and abuse of logs here.

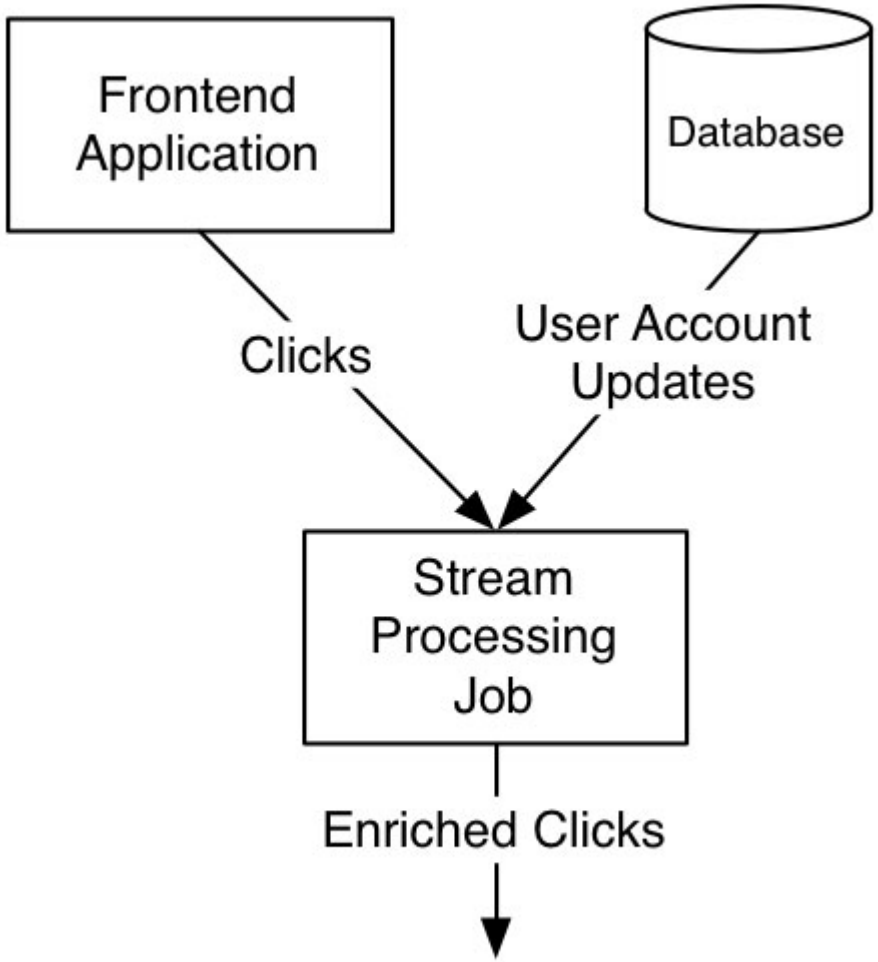
Databases are streams

One aspect of the usage patterns that may not be immediately obvious is that you can treat databases as being a source of a stream: the stream of row updates. MySQL, HBase, Oracle, Postgres, and

MongoDB all give access to the stream of changes in this way. Kafka has good support for this kind of keyed data and can act as a data subscription layer for data coming out of these databases, making it easy to have a lot of processing jobs subscribing to these changes.

Stream processing jobs can then subscribe to these streams and record the subset of data they need to use in their processing into their local state for fast access.

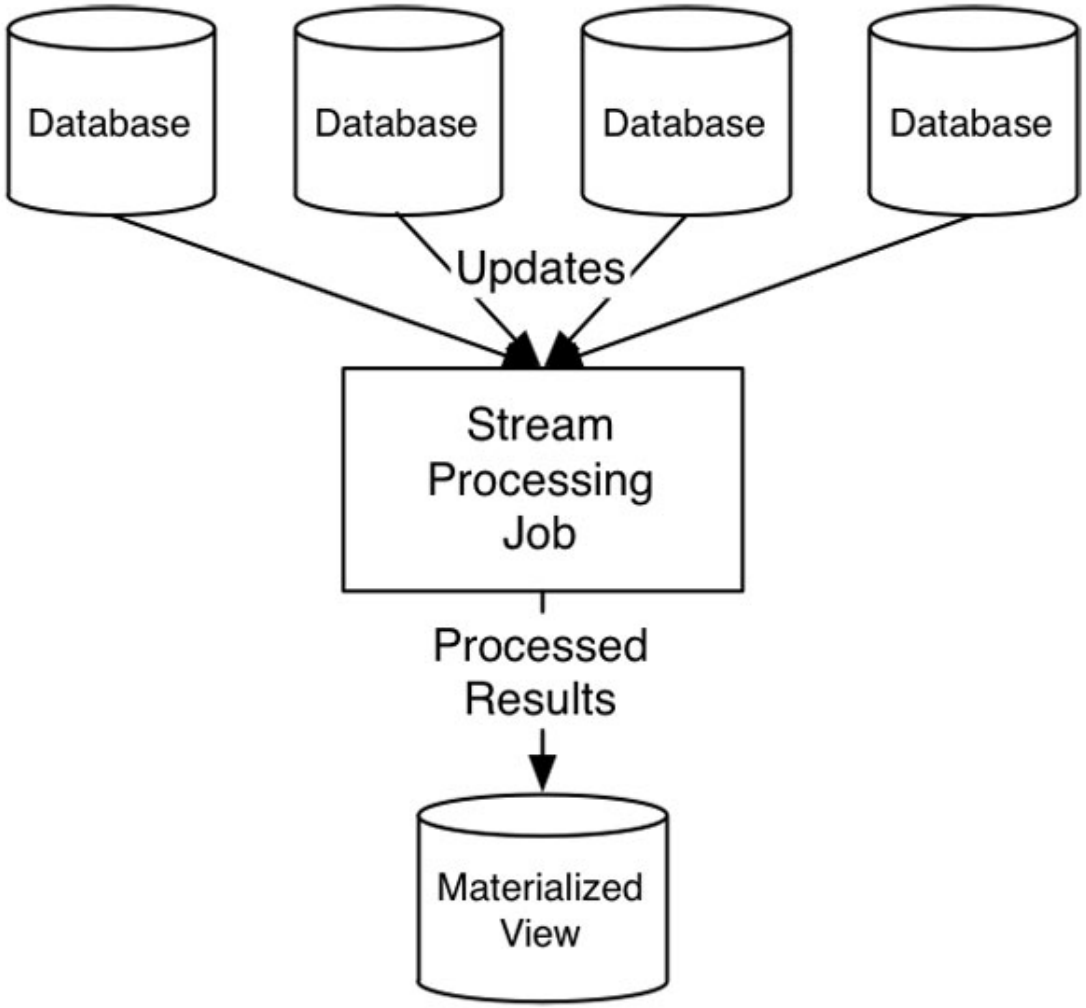
So, our "data enrichment" use case where we wanted to join information about users to a stream of clicks can be implemented by subscribing to the click stream as well as the user account change stream like this:



When the stream processing job gets a user account database update, it pulls out the attributes it uses and writes them to its local key-value store. When it gets a click event, it looks up the user

attributes in its local store and outputs the click event with the additional user attributes included.

In fact, it often happens that *all* the processing is against database change streams coming from different databases. The stream processor acts as a kind of fancy cross-database trigger mechanism to process changes, usually to create some materialized view to be used for serving live queries:



Why would you want this?

Okay, so now I've explained how you can have fault-tolerant local state, but why would you want this? Why not just use a remote database?

There are three primary reasons:

1. Local state can be indexed and accessed in a variety of rich ways that are much harder with remote databases
2. Local, in-process data access can be much, much faster than remote access
3. Local data access is easier to isolate

Rich access patterns

Let's start with why locality allows rich access. Want to have an inverted index? A compressed bitmap index? Key-value access? Want to do full table scans? Want to compute complex aggregations? You can do all of these. Of course, centralized relational databases do allow rich remote access, but to get this, you have to trade horizontal scalability. To get horizontal scalability, you need to partition your data (internally this is what something like HBase or Cassandra would be doing). Since Samza already has a partitioning

model, it makes sense to use this same partitioning model for data that is used for input streams.

Once you have co-located the data with the processing, you can access it in any way you like, as it is local to your machine. If you want to periodically stop processing, scan the full table, compute a complex aggregation on it, and produce output, that will work fine. This is really useful for systems that aggregate inputs like clicks and impressions, and periodically (say every 10 mins), stop and do a brute-force scoring of their accumulated statistics to output the top-N results. Sometimes these kinds of things can be approximated without the periodic batch scan, but often this is hard, and having the option to just apply brute force is nice.

The fundamental idea here is that virtually any kind of data structure can be be backed by a changelog and be made fault tolerant. This means the storage is pluggable — any kind of embeddable data structure can be used. Samza provides key-value storage out of the box, but you can plug in anything you like (an in-memory hash table, a bloom filter, a bitmap index, whatever).

This architecture has the added benefit that the processor is always the single writer for its local partition (if you want another processor

to get data, just repartition the same way you repartition any stream). This means that a lot of the complexity of reasoning about concurrent modifications from different processors disappears.

Local access is faster

To get an idea of why streaming access is faster than remote look-ups, take a look at this benchmark. A single thread can read or write data to Kafka at around 100MB/sec, which for small messages might be on the order of a million messages/sec. This is fundamentally a throughput-bound operation. But now imagine that for each of these incoming messages we need to look up some side data to join to our input. Doing remote lookups against a data store, you might be able to do on the order of 1-10k lookups per second per server on a well-tuned store. This is a huge gap when compared to the performance of streaming access!

The idea that random access is fundamentally slower than sequential access is nothing new, and is discussed wonderfully here. The implication is that jobs that are stateless, and can do their processing without querying an external system, can be very very fast. The challenge is that very few real-world use cases are actually so

simple: in the real world, people want to join on additional side data to their streams, or compute aggregations, etc. Having one processing stage run at streaming speeds of a million records per second won't help you if the next stage needs to do remote random lookups and can only process a thousand records per second. The throughput of a stream processing job will end up being determined by its slowest component, which will almost always end up being the remote random access lookups or updates.

But random access is really useful — we can't just get rid of joins and aggregations. How can we speed these up? To get a feel for the kind of performance that is possible with local embedded stores, take a look at some of the benchmarks for LMDB and RocksDB running on SSDs. The performance of these embedded data stores is amazing, and will generally make any kind of network layer the bottleneck. By having a lot of data locally in memory and the rest on an SSD, a single thread can very often do on the order of hundreds of thousands or even millions of random lookups per second (depending on how cacheable the lookups are).

This observation is what Dhruba Borthakur describes as the motivation for Facebook's choice to invest heavily in the standalone embedded key-value library, RocksDB. They believed that the

differential between SSD performance and network performance would lead them to architectures that co-located data with application processing.

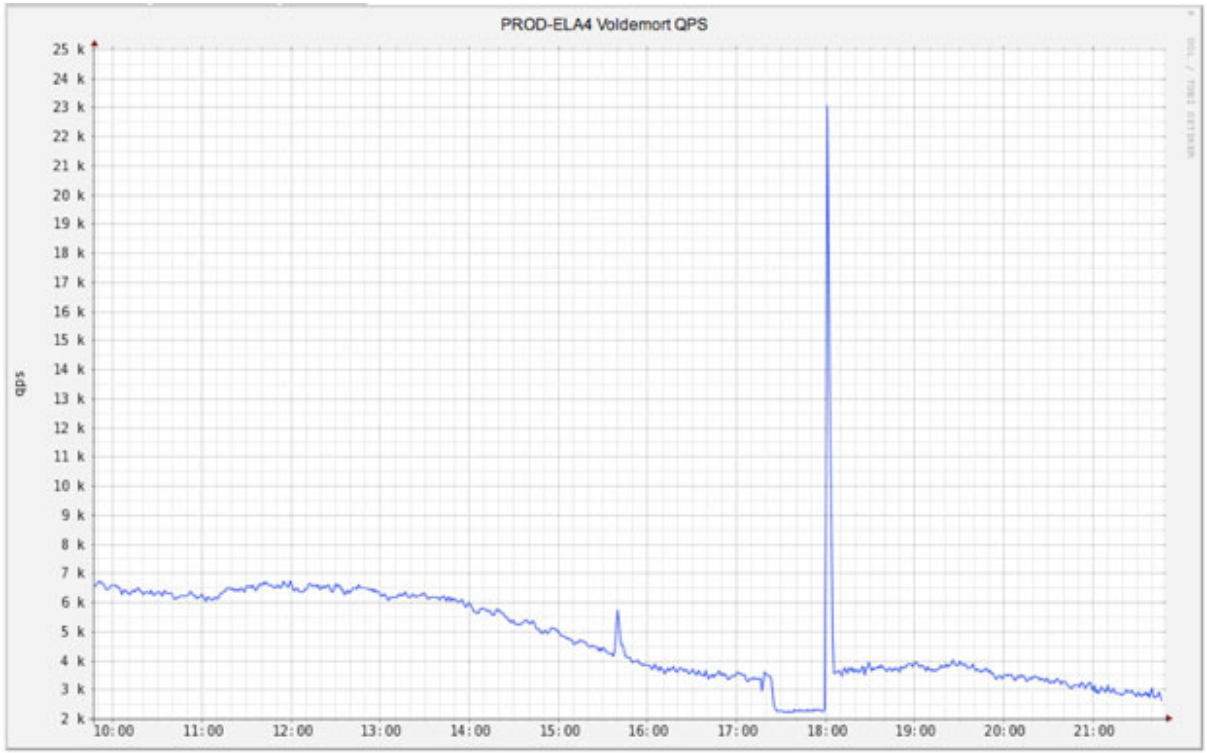
Local access is easier to isolate

One important thing to understand about a stream processing system like Samza is that it will sometimes process at a very high rate. In steady-state, of course, its rate of processing will be controlled by the rate of incoming data. But if a job is brought down or stopped, when it restarts it will catch up on whatever backlog of data has accumulated, and it may do this very quickly. A similar situation arises due to re-processing of data. That is, you will inevitably change your code to produce different, better, results and want to reprocess your input to backfill this new output. Since stream processing systems like Samza let you run your processing partitioned over a cluster of machines, these high-throughput periods can be very high indeed.

This is very different from the load profile of normal websites and services, which generate load on databases proportional to traffic. If you are a website that serves 10,000 page views per second, then

you probably do a number of database queries proportional to that. This rate doesn't increase unless your website suddenly becomes much more popular.

However, because of re-processing and catch-up, stream processing doesn't have this limitation — load can spike arbitrarily high. This is not an academic problem. Here is a production monitoring graph of the query load on a table generated by a stream processing job running against a remote key-value store when the job was stopped for 30 minutes and then restarted, and did its catch-up processing:



This makes sharing this kind of storage a real problem, especially if it is shared with any kind of online usage. These load spikes inevitably lead to corresponding latency spikes for any other users which can, in turn, bring down any online services sharing the same database.

By making data access local, we allow the stream processing job to thrash its own local disk or SSDs without fear of interrupting any online services.

Downsides

This pattern is not without some downsides.

First, there is some duplication of data since the stream processing job indexes the same data that is stored elsewhere in a live store. However, since the jobs often want just a few attributes, this duplication is generally not the full data set but just a small subset. In any case, for isolation purposes, if the primary store is used for serving live queries, you will need to replicate the data to avoid impacting the live use cases.

Second, if the local state per process gets too large, the restore time for a failed job will become slow. For very, very large datasets, this pattern may not make sense.

Finally, if significant logic around data access is encapsulated in a service, then replicating the data rather than accessing the service will not enforce that logic. This issue does arise, but it is somewhat less common than it would seem — and quite similar to the situation that arises with Hadoop and MapReduce processing, which virtually requires the data to be replicated to the local system for processing.

In these cases where any of these issues is a major concern, local state may not make sense. But, of course, having such a facility does not mean that every job needs to use it: nothing prevents a job from making remote calls. However, we have found that once this facility is available, it is very often preferable.

Does random access makes sense for stream processing?

When first introduced to the dilemmas around random access, many people who have experience with MapReduce will wonder whether random access is appropriate for use in stream processing at all.

After all, MapReduce tries to address the disparity between random and sequential access by removing random lookups entirely. A join or aggregation in MapReduce is generally a sort and merge (though, of course, abusing MapReduce to do unnatural things like hash joins is almost a national pastime among Hadoop users). Unfortunately, blocking sorts aren't very convenient in a stream processing system that needs to output data continuously: results would have to be continuously re-sorted as updates arrive. (As an aside, the internals of RocksDB, which essentially queries and merges a bunch of sorted

files, is not that different from what you would get if you tried to imagine a kind of continual, online reducer).

But MapReduce was designed in the age of magnetic hard drives. Traditional spinning hard drives make random access quite expensive. As long as data fits, in-memory access is virtually instantaneous, but as soon as you begin to access disk, you very quickly exhaust your seek capacity, and performance falls off a cliff. An individual drive can generally do around 100-200 random accesses per second. This is very, very far from our million messages per second streaming performance. So, you can see why the designers of MapReduce were so emphatic about avoiding random access, even though it is often very inconvenient to live without.

But the world has changed since MapReduce. Now we have cheap SSDs. SSDs don't remove the gap between random access and linear access, but they do reduce it significantly. Random access, though still expensive, is no longer a deal breaker for throughput-oriented processing.

Building a system from scratch that co-locates application processing with local data storage access can be difficult. You need to have a way to partition data to scale beyond the capacity of a

single machine. Since this is local storage, if a machine dies, that data may well be gone, so you need fault-tolerance mechanisms to let you recover. Doing this as part of building a particular application is a big undertaking.

What Samza provides with its local state facilities gives you a much simpler out-of-the-box mechanism for doing partitioned, fault-tolerant, local state management for asynchronous processing use cases.

To learn a little more, take a look at the documentation and use cases for stateful processing here.

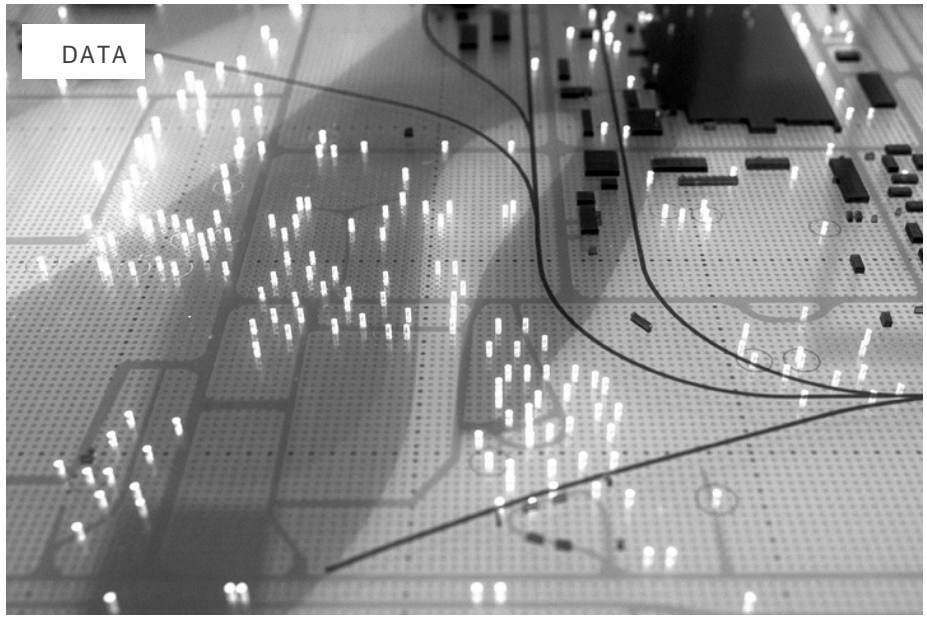
Article image: Nottawasaga River (source: BiblioArchives).

Share Tweet Share

Jay Kreps

Jay Kreps is the cofounder and CEO of Confluent, a company focused on Apache Kafka. Previously, Jay was one of the primary architects for LinkedIn, where he focused on data infrastructure and data-driven products. He was among the original authors of a number of open source projects in the scalable-data-systems space, including "Voldemort":http://project-voldemort.com/, "Azkaban":https://azkaban.github.io/, "Kafka":http://kafka.apache.org, and "Samza":http://samza.incubator.apache.org/.

more

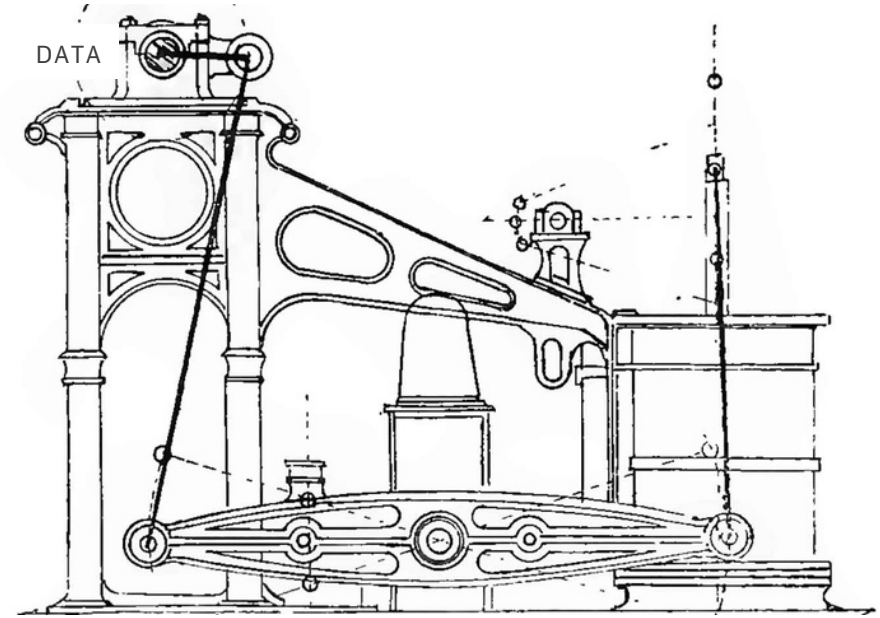


Oil, Gas, and Data

By Daniel Cowles

High-performance data tools in the production of industrial power

DATA



Designing great data products

The Drivetrain Approach: A four-step process for building data products.

DATA

The next 10 years of Apache Hadoop

By Doug Cutting, Tom White, and Ben Lorica

Doug Cutting, Tom White, and Ben Lorica explore Hadoop's role over the coming decade.

Data: Emerging Trends and Technologies

By Alistair Croll

How sensors, fast networks, AI, and distributed computing are affecting the data landscape

ABOUT US

Our Company
Teach/Speak/Write
Careers
Customer Service
Contact Us

SITE MAP

Ideas
Learning
Topics
All

© 2018 O'Reilly Media, Inc. All trademarks and registered trademarks appearing on oreilly.com are the property of their respective owners.

Terms of Service • **Privacy Policy** • **Editorial Independence**