CICompass
TECH NOTES
ci-compass.org

A Review of Streaming Systems and Architectures

Authors: Erik Scott and Steve Petruzza
Date Published: April 30, 2024
DOI: 10.5281/zenodo.11087313

Sometimes science happens slowly, sometimes it comes at us like a firehose. Experiments that produce huge volumes of data, e.g. scientific instruments producing streaming data, are becoming more common with time. Capturing, processing, storing, and sharing high-volume, high-speed scientific data is a challenging problem and one that often requires a mix of off-the-shelf tools and custom, bespoke solutions.

In this technical note, we will take a short look at the challenges in more detail, highlight some existing tools that are useful to address these challenges and discuss some general strategies and design principles for those times that you have to take data matters into your own hands.

To Save Everything Or Not

One of the first decisions to make when designing a streaming data system is whether to store every possible data point or not. In some cases, there may be far too much data to store all of it. Experiments with lots of high-resolution video, for instance, can quickly grow into petabytes. Saving this in a long-term archive is not an attractive prospect. Despite scientists' proclivity to "save everything," sometimes there is no choice but to process the incoming stream of data as rapidly as possible and only save the interesting events and results. On the other hand, experiments that produce still-impressive, but more manageable data flows, can benefit from high-throughput storage techniques. Some relevant methods include "NoSQL" databases, in-memory databases, ring buffer architectures, and message queues.

Traditionally, database management systems were not the best way to handle high volumes of incoming data. These systems are designed with transaction processing in mind; hence, they offer strong guarantees on what would happen when two or more reads and writes occur at the same time. To a large degree, this is less relevant for scientific streaming workflows - reading is usually done later under fully specified circumstances. This shifts the burden of ensuring correctness under concurrency away from the database and onto the application code. So-called "NoSQL" databases were invented precisely for this purpose.

Examples of NoSQL databases appropriate for streaming scientific data flows include "key-value" stores and "document" databases. Key-value stores save records with precisely two pieces of information. The key component is an identifier unique to that record and is the only way it can be looked up. The value is the actual "payload" of the record. This is where information such as the actual sensor value or measurement is stored. Both the key and the value are generally treated as simple buckets of binary data and are usually accessed as though they are strings. Because data can only be retrieved by the specific key used to store it, attention needs to be paid to how that key is formed. Often, the key will contain information on the sensor name and the time. Note that it is (usually) impossible to look up a record based on any arbitrary sub-part of the key. For instance, there is no way to query for all sensor readings between Tuesday and Thursday short of generating every possible key and performing a brute-force solution. Examples of key-value stores include Cassandra [1] (nearly the oldest of the parallel NoSQL stores) and Riak [2] (notable for having a version, RiakTS, specifically for Time Series data). ClickHouse [3] is interesting for its ability to handle large numbers of incoming

This project is supported by the National Science Foundation Office of Advanced Cyberinfrastructure in the Directorate for Computer Information Science under Grant #2127548.

records per second by making efficient use of memory pools spread among large clusters of servers.

Document Stores, on the other hand, offer a much richer set of capabilities at the expense of often less compact storage. The name is a bit misleading - the only type of “document” these systems can store is a JSON format record. Searching for records is much more flexible than a key-value store. Any field of the JSON records can be used as a key, and in fact, there can be many indexed fields for each record resulting in many different ways to search for data. Multiple records can be returned if there are multiple matches. Unlike key-value stores, document databases can do “range” queries (e.g., “show all data points captured between 3:30 am and 5:00 pm”). Finally, document databases can search fields that are not indexed. Keys are indexed for fast lookup, but searching based on non-indexed fields requires the database to read every record stored looking for matching data. An example document database (and by far the leader of this type) is MongoDB [4].

Ruthless Curation

Saving everything you run across is not always a good idea. It is easy for scientific experiments to produce high rates of data for very long periods of time, building to colossal volumes very quickly, as in streaming data generated by the LIGO detectors [5] The price of computing capability has fallen faster than the price of storage, and it is not unusual to see storage costs exceed those of the compute costs. This is especially true in cloud environments, where a system architecture is needed that can accept incoming data at a very high rate, do some processing to classify each record, and save only the interesting ones. Admittedly, scientists may be unnerved by this strategy - as they are trained to preserve all of their observations. However, when faced with sufficient challenges, a pragmatic solution might be the only way forward.

“Stream Processing” systems are not new - they have been used in the finance world for decades. A classic example is arbitrage - exploiting tiny, simultaneous imbalances in pricing and thus profiting from the difference. When exchange rates between two currencies are not precisely reciprocal to each other, for instance, the difference can be pocketed. Such imbalances are usually very short-lived, on the order of milliseconds. Specialized solution strategies are needed for this to work.

The most simple way to handle high incoming data rates is to write your analysis code in such a way that it reads its inputs from a network socket. A few extra lines of Python are all it takes. In many cases, this might be enough. For example, tiny sensors for environmental studies may have serious limitations in power budget and communications ability. In a case like this, even simple filtering can reduce the communications burden dramatically, and the radio is often the dominant power consumer in these applications.

Sometimes, the problem deserves a better solution. There may be a need to at least temporarily store the incoming data, perhaps for disaster recovery. There might be a significant mismatch between incoming data rates and processing needs for classification. It could be the case that the incoming data is “bursty” (lots of data at once, followed by a period of little or no flow) and there needs to be some “buffering” to prevent having to over-provision the computing resources. It could also be the case that a large number of data points have to be compared with each other, likely in some sort of time series analysis, and it is necessary to temporarily store a “sliding window” of entries. In cases like this, it is common to employ a message queue or a specialized database-like system suitable for transient storage and analysis.

Message Queues can be thought of as “program-to-program email”. Of course, it is more efficient than email, but the premise is the same. Some process sends messages off into some liminal space with no concern about what happens to them after that. Some other processes, this one on the analysis side, read these messages and act on them. The two processes may read and write messages at the same rate, in which case the queue is effectively equivalent to a socket connection, or one task might be much faster than the other. In this latter case, the queue serves as a buffer to temporarily hold the buildup for messages. It is important to realize this works only as long as the slower element is able to eventually catch up. It might be fine if your experiment produces a million data points per second and your analysis takes a millisecond each - if your experiment runs for one second and you have enough temporary storage for a million points worth of readings. On the other hand, if the

This project is supported by the U.S. National Science Foundation Office of Advanced Cyberinfrastructure in the Directorate for Computer Information Science under Grant #2127548.

experiment runs constantly for months then the queue is likely to overflow and data will be lost. Commonly used message queue software includes Kafka and RabbitMQ [6]. Worth noting is the difference between throughput and latency. Throughput is a measure of the amount of data handled per unit of time, while latency is the amount of time between data first arriving and computed data flowing out the other end of the pipeline. Latency can be thought of as “reaction time”, of a sort, for a streaming system.

As a representative example of a message queue-based streaming platform, let us consider Apache Kafka [7]. Kafka is a distributed system that can run across many servers. Adding servers can increase throughput, reliability, or both. Architecturally, Kafka's dataflow is organized as a collection of "topics", which can be conceptualized as a "queue name". Messages consist of key-value pairs; keys are allowed to be empty, but if there is a valid key then it can be used to distribute workload among servers.

Kafka is a specific kind of message queueing system - a publish-subscribe (universally called pub-sub) model. Many producers (code that creates messages and sends them) can write into a given topic. Similarly, many subscribers (processes that read the messages) can connect to a queue and read messages from it. Reading from a topic does not automatically remove the message from the queue - a given message can be read by many (perhaps thousands) of subscribers [7]. Kafka can be configured to support guaranteed "exactly once" semantics, implementing distributed transactions.

Specialized databases exist for streaming data. Aerospike [8], for instance, can ingest data at high rates and temporarily store it in memory. Records are marked with an “expiration time” after which they are automatically deleted and the space is recovered. It can also write records to persistent storage (flash, phase change memory, spinning disk, or similar) and thus blurs the lines between a specialty streaming analysis system, a streaming database, and a message queue. Indeed, streaming databases can be pressed into service as message queues.

Finally, the ultimate brute-force solution to high-rate data reduction is Custom Hardware. Field Programmable Gate Arrays (FPGAs) [9] are hardware devices that can be “reprogrammed” to act like any arbitrary digital system (up to size constraints of the device, of course). It is not uncommon to use these devices to perform computation on data coming into some of their ports. It is not even necessary to synthesize something resembling a processor. Quite often the equivalent of a dedicated array of gates can do enough computation to perform data reduction. This sort of strategy is fairly common in radio astronomy and space communications, and of course, Wall Street has found its own uses for the technology.

Conclusions

Dealing with high-rate scientific data is the domain of Streaming Systems. The major dichotomy is between systems that store incoming data for long periods of time versus those that take a transient approach to storage. A key design point is knowing if the incoming data will arrive for a long time or only in short bursts, and another is knowing the frequency of incoming records versus the time to process them. Finally, all such systems need to be aware of throughput (how much data can be handled per unit time) versus latency (how much time elapses between data coming in and results flowing out.

References

[1] Wahid, Abdul, and Kanupriya Kashyap. "Cassandra—A distributed database system: An overview." Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2018, Volume 1 (2019): 519-526.

[2] Klophaus, Rusty. "Riak core: Building distributed applications without shared state." In ACM SIGPLAN Commercial Users of Functional Programming, pp. 1-1. 2010.

[3] Fast Open-Source OLAP DBMS - ClickHouse". ClickHouse. Accessed Mar 28, 2024. https://clickhouse.com/

[4] Banker, Kyle, Douglas Garrett, Peter Bakkum, and Shaun Verch. MongoDB in action: covers MongoDB version 3.0. Simon and Schuster, 2016.

[5] Laser Interferometer Gravitational-wave Observatory. Accessed Mar 28, 2024. https://www.ligo.caltech.edu/page/what-is-ligo

[6] Dobbelaere, Philippe, and Kyumars Sheykh Esmaili. "Kafka versus RabbitMQ: A comparative

This project is supported by the U.S. National Science Foundation Office of Advanced Cyberinfrastructure in the Directorate for Computer Information Science under Grant #2127548.

study of two industry reference publish/subscribe implementations: Industry Paper." In Proceedings of the 11th ACM International Conference on distributed and event-based systems, pp. 227-238. 2017.

[7] What is Kafka, and How Does It Work? A Tutorial for Beginners". Confluent, Inc. Accessed Mar 28, 2024. https://developer.confluent.io/what-is-apache-kafka/

[8] Srinivasan, V., Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. "Aerospike: Architecture of a real-time operational DBMS." Proceedings of the VLDB Endowment 9, no. 13 (2016): 1389-1400.

[9] Farooq, Umer, Zied Marrakchi, Habib Mehrez, Umer Farooq, Zied Marrakchi, and Habib Mehrez. "FPGA architectures: An overview." Tree-Based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization (2012): 7-48.

This project is supported by the U.S. National Science Foundation Office of Advanced Cyberinfrastructure in the Directorate for Computer Information Science under Grant #2127548.