Streaming Systems and Architectures

JAYANT SHEKHAR AND AMANDEEP KHURANA

Jayant is Principal Solutions Architect at Cloudera working with various large and small companies in various Verticals on their big data and data science use cases, architecture, algorithms, and deployments. For the past 18 months, his focus has been streaming systems and predictive analytics. Prior to Cloudera, Jayant worked at Yahoo and at eBay building big data and search platforms. jayant@cloudera.com

Amandeep is a Principal Solutions Architect at Cloudera, where he works with customers on strategizing on, architecting, and developing solutions using the Hadoop ecosystem. Amandeep has been involved with several large-scale, complex deployments and has helped customers design applications from the ground up as well as scale and operationalize existing solutions. Prior to Cloudera, Amandeep worked at Amazon Web Services. Amandeep is also the co-author of *HBase in Action*, a book on designing applications on HBase. amansk@gmail.com

Over the last few years, we have seen a disruption in the data management space. It started with innovation in the data warehousing and large-scale computing platform world. Now we are seeing a similar trend in real-time streaming systems. In this article, we survey a few open source stream processing systems and cover a sample architecture that consists of one or more of these systems, depending on the access patterns.

Data Management Systems

Data management systems have existed for decades and form a very mature industry that we all know about. Notable vendors playing in this market include Oracle, Microsoft, Teradata, IBM—some of the most valuable companies on the planet. Data is at the core of a lot of businesses, and they spend millions of dollars for systems that make it possible to ingest, store, analyze, and use data relevant to their customers, channels, and the market. Although mature, the data management industry is going through a disruption right now. This is being caused by the explosion of data being created by humans and machines owing to cheaper and more widespread connectivity. This has given rise to the entire big data movement and a plethora of open source data management frameworks that allow companies to manage data more cheaply and in a more scalable and flexible manner.

Data management systems can be broken down into different categories, depending on the criteria you pick. Databases, file systems, message queues, business intelligence tools are all part of this ecosystem and serve different purposes inside of a larger architecture that solves the business problem. One way to categorize these systems is based on whether they handle data at rest or in motion.

Data at Rest

Systems for data at rest include databases, file systems, processing engines, and grid computing systems. Most architectures for data at rest have a separate storage tier to store raw data, a compute tier to process or clean up data, and a separate database tier to store and analyze structured data sets. In some cases a single system might be performing multiple such functions. That's not necessarily an ideal architecture from a cost, scale, and performance perspective, but they do exist out there in the wild.

Data in Motion

Systems for managing data in motion include things like message queues and stream processing systems. Architectures for data in motion consist of multiple such systems wired and working together toward a desired end state. Some solutions are simply to ingest data from sources that are creating events. Others have a stream processing aspect that writes back into the same ingestion layer, creating multiple data sets that get ingested into the system managing data at rest. Others have the stream processing system as part of the ingestion pipeline so that output is written straight to the system managing data at rest. The stream processing systems could also have different characteristics and design principles.

In this article, we'll survey a few open source systems that deal with streaming data and conclude with a section on architectures that consist of one or more of these systems, depending on the access patterns that the solution is trying to address.

Streaming Systems

There are two types of streaming systems: stream ingestion systems and stream analytics systems. Stream ingestion systems are meant to capture and ingest streaming data as it gets produced, or shortly thereafter, from sources that spew out data. Stream ingestion systems capture individual or small batches of payloads at the source and transport them to the destination. Stream analytics systems, on the other hand, process data as it streams into the system. Work is done on the payloads as they become available. It does not necessarily wait for entire batches, files, or databases to get populated before processing starts. Stream ingestion systems are typically the source for the stream analytics systems. After the stream is analyzed, the output could either be put back into the ingestion system or written to a system that handles data at rest. We'll dive deeper into the following systems:

1. Kafka, a messaging system that falls under the category of stream ingestion systems per the criteria above [1].
2. Spark Streaming, a stream processing system that works with small batches of data as they come in [2].
3. Storm, a stream processing system that works with individual events as they come in [3].
4. Flink, a distributed stream processing system that builds batch processing on top of the streaming engine [4].

Kafka

Apache Kafka [1] is a publish-subscribe messaging system; it is also a distributed, partitioned, replicated commit log service. It has been designed to handle high-throughput for writes and reads of events, handle low-latency delivery of events, and handle machine failures.

Kafka is usually deployed in a cluster. Each node in the cluster is called a *broker*. A single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients. The cluster can be elastically expanded without downtime.

Kafka has a core abstraction called *topics*, and each message coming in belongs to a topic. Clients sending messages to Kafka topics are called *producers*. Clients that consume data from the Kafka topics are called *consumers*. Clients can be implemented in a programming language of your choice.

Communication between the clients and the Kafka brokers is done in a language-agnostic binary TCP protocol. There are six core client request APIs.

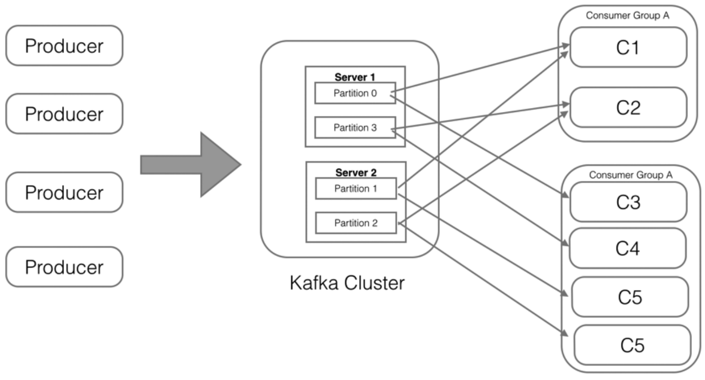


Figure 1: Kafka producers, cluster, partitions, and consumer groups

The topics are split into pre-defined *partitions*. Each partition is an ordered sequence of events that is continuously appended to a commit log. Each message in a partition is assigned a sequential event ID. In Figure 1, we have four partitions for the topic. Partitions can reside on different servers, and hence a topic can scale horizontally. Each partition can be replicated across the brokers for high availability. Messages are assigned to specific partitions by the clients and not the Kafka brokers.

Producers can round-robin between the partitions of the topic when writing to them. If there are too many producers, each producer can just write to one randomly chosen partition, resulting in far fewer connections to each broker.

Partitioning also allows different consumers to process different parts of data from the topic. For simple load balancing, the client can round-robin between the different brokers. Consumers can belong to a consumer group as shown in Figure 1, and each message is delivered to one subscribing consumer in the group.

You can batch events when writing to Kafka. This helps to increase the overall throughput of the system. Batching can also take place across topics and partitions.

Kafka stores the messages it receives to disk and also replicates them for fault-tolerance.

Apache Kafka includes Java clients and Scala clients for communicating with a Kafka cluster. It ships with a library that can be used to implement custom consumers and producers.

There are many tools that integrate with Kafka, including Spark Streaming, Storm, Flume, and Samza.

Spark Streaming

Spark Streaming [2] runs on top of the Spark [5] cluster computing framework. Spark is a batch processing system that can run in standalone mode or on top of resource management frameworks like YARN [7] or Mesos [8]. Spark Streaming is

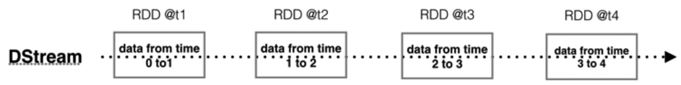


Figure 2: DStreams consists of multiple RDDs based on the time interval.

a subcomponent of the Spark project that supports processing microbatches of streams of events as they come in. Spark Streaming also supports windowing, joining streams with historical data.

Spark Streaming can ingest data from many sources, including Kafka, Flume, Kinesis, Twitter, and TCP sockets. It has inherent parallelism built in for ingesting data. The core abstraction of Spark Streaming is Discretized Streams (DStreams), which represents a continuous stream of events, created either from the incoming source or as a result of processing a source stream. Internally, DStreams consists of multiple Resilient Distributed Datasets (RDDs) [9], which are a core abstraction of the Spark project. These RDDs are created based on the time interval configured in the Spark Streaming application that defines the frequency with which the data from DStreams will be consumed by the application. A visual representation of this is shown in Figure 2.

Spark Streaming processes the data with high-level functions like `map`, `reduce`, `join`, and `window`. After processing, the resulting data can be saved on stores like HDFS, HBase, Solr, and be pushed out to be displayed in a dashboard or written back into a new Kafka topic for consumption later.

When it receives streaming data, Spark Streaming divides the data into small batches (mini batches). Each batch is stored in an RDD, and the RDDs are then processed by Spark to generate new RDDs.

Spark Streaming supports Window Operations, and it allows us to perform transformations over a sliding window of data. It takes in the window duration and the sliding interval in which the window operations are performed.

For Complex Event Processing (CEP), Spark Streaming supports stream-stream joins. Apart from inner-joins, left, right, and full outer-joins are supported. Joins over windows of streams are also supported as are stream-data set joins.

Storm

Apache Storm [3] is an open source project designed for distributed processing of streaming data at an individual event level. A Storm deployment consists of primarily two roles: a master node, called Nimbus, and the worker nodes, called Supervisors. Nimbus is the orchestrator of the work that happens in a Storm deployment. Supervisors spin up workers that execute the tasks on the nodes they are running on. Storm uses Zookeeper under the hood for the purpose of coordination and storing operational

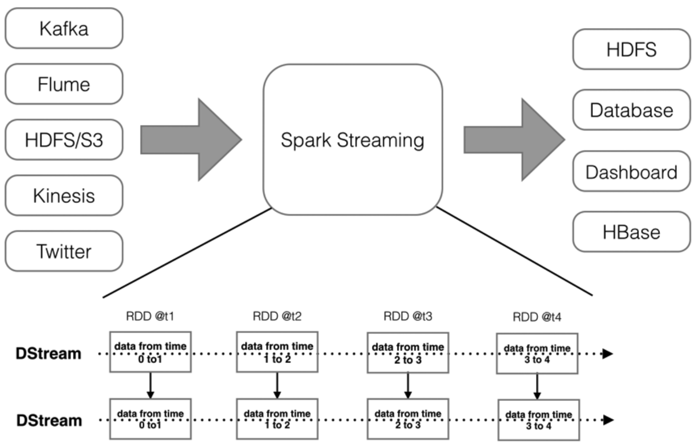


Figure 3: Diagram of Spark Streaming showing Input Data Sources, Spark DStreams, and Output Stores

state. Storing state in Zookeeper allows the Storm processes to be stateless and also have the ability to restart failed processes without affecting the health of the cluster.

Streaming applications in Storm are defined by *topologies*. These are a logical layout of the computation that the application is going to perform for the stream of data coming in. Nodes in the topology define the processing logic on the data, and links between the nodes define the movement of data. The fundamental abstraction in Storm topologies is of a Stream. Streams consist of tuples of data. Fields in a tuple could be of any type. Storm processes streams in a distributed manner. The output of this processing can be one or more streams or be put back into Kafka or a storage system or database. Storm provides two primitives to do the work on these streams—*bolts* and *spouts*. You implement bolts and spouts to create your stream processing application.

A spout is a source of the stream in the Storm topology. It consumes tuples from a stream, which could be a Kafka topic, tweets coming from the Twitter API or any other system that is emitting a stream of events.

A bolt consumes one or more streams from one or more spouts and does work on it based on the logic you've implemented. The output of a bolt could be another stream that goes into another bolt for further processing or could be persisted somewhere. Bolts can do anything from run functions, filter tuples, do streaming aggregations, do streaming joins, talk to databases, and more. A network of bolts and spouts make up a Storm topology (graphically shown in Figure 4) that is deployed on a cluster where it gets executed.

A topology keeps running until you terminate it. For each node, you can set the parallelism and Storm will spawn the required number of threads. When tasks fail, Storm automatically restarts them.

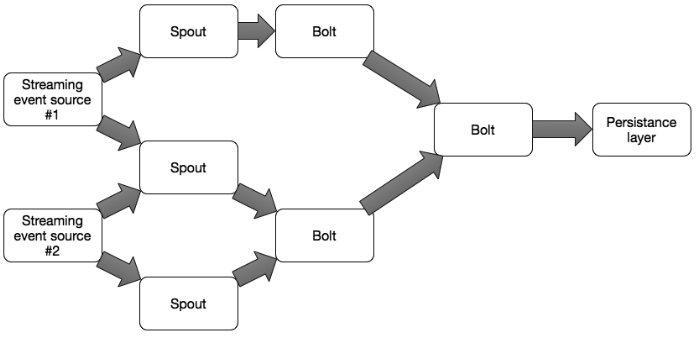


Figure 4: A Storm topology consisting of bolts and spouts

Storm provides three levels of guarantees for tuples in a stream.

- At-most-once processing: this mode is the simplest one and is appropriate in cases where it is required that a tuple be processed not more than once. Zero processing for a tuple is possible, which means message loss is acceptable in this case. If failures happen in this mode, Storm might discard tuples and not process them at all.
- At-least-once processing: this mode is where the application needs tuples to be processed at least one time. This means that more than once is acceptable. If the operations are idempotent or a slight inaccuracy in the results of the processing is acceptable, this mode would work fine.
- Exactly-once processing: this is a more complex and expensive level. Typically, an external system like Trident [6] is used for this guarantee level.

Storm provides users with a simple way to define stream processing topologies with different kinds of configurations. These make for a compelling way to implement a streaming application. Twitter recently announced a new project (Heron [10]) that learns lessons from Storm and is built to be the next generation of Storm.

Apache Flink

Apache Flink, like Spark, is a distributed stream and batch processing platform. Flink's core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams.

Flink uses streams for all workloads—streaming, micro-batch, and batch. Batch is treated as a finite set of streamed data.

Spark is a batch processing framework that can approximate stream processing; Flink is primarily a stream processing framework that can look like a batch processor.

At its core, Flink has an abstraction of DataStreams for streaming applications. These represent a stream of events of the same type created by consuming data from sources like Kafka, Flume, Twitter, and ZeroMQ. DataStream programs in Flink are regular programs that implement transformations on streams. Results may be written out to files, standard output, or sockets. The execution can happen in a local JVM or on clusters of many machines. Transformation operations on DataStreams include Map, FlatMap, Filter, Reduce, Fold, Aggregations, Window, WindowAll, Window Reduce, Window Fold, Window Join, Window CoGroup, Split, and some more.

Data streaming applications are executed with continuous, long-lived operators. Flink provides fault-tolerance via Lightweight Distributed Snapshots. It is based on Chandy-Lamport distributed snapshots. Streaming applications can maintain custom state during their computation. Flink's checkpointing mechanism ensures exactly-once semantics for the state in the presence of failures.

The DataStream API supports functional transformations on data streams with flexible windows. The user can define the size of the window and the frequency of reduction or aggregation calls. Windows can be based on various policies—count, time, and delta. They can also be mixed in their use. When multiple policies are used, the strictest one controls the elements in the window.

As an optimization, Flink chains two subsequent transformations and executes them within the same thread for better performance. This is done by default if it is possible, and the user doesn't have to do anything extra. Flink takes care of finding the best way of executing a program depending on the input and operations. For example, for join operations, it chooses between partitioning and broadcasting the data, between running a sort merge join and a hybrid hash join.

As you can see, Apache Flink has similar objectives as Apache Spark but different design principles. Flink is more powerful based on the design and capabilities since it can handle batch, micro-batch, and individual event-based processing, all in a single system. As it stands today, Flink is not as mature a platform as Spark and doesn't have the same momentum and user community.

Architectural Patterns

Streaming architectures often consist of multiple systems integrated with each other depending on the desired access patterns. Custom integrations happen at the following stages of a streaming pipeline.

1. Ingestion points
2. Stream processing output points

There are typically two ingestion point integrations in a typical architecture: integration of the message queue (Kafka for the context of this article) with the source system, and integration of the message queue with the stream processing system (Storm, Spark Streaming, or Flink for the context of this article).

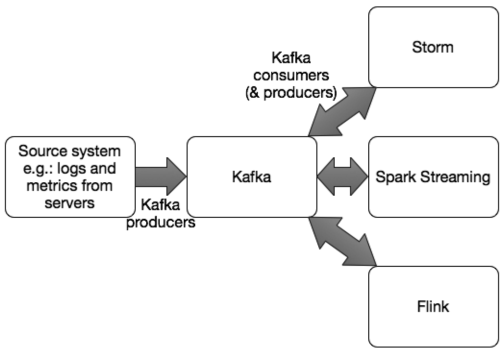

Source system e.g.: logs and metrics from servers
Kafka producers
Kafka
Kafka consumers (& producers)
Storm
Spark Streaming
Flink

Figure 5: Streaming architecture consisting of Kafka, Storm, Spark Streaming, and Flink

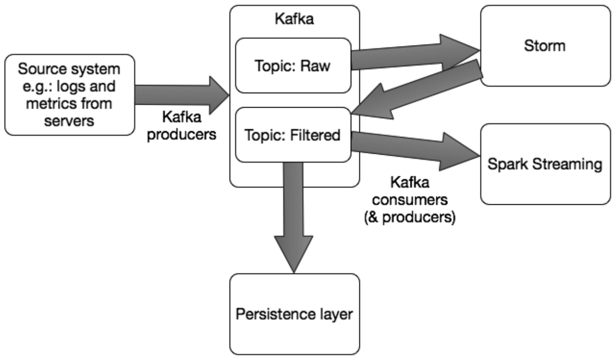


Figure 6: Streaming access pattern showing Storm processing events first, with results then processed by Spark Streaming and also persisted

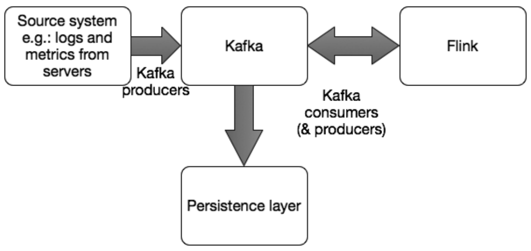

Source system e.g.: logs and metrics from servers
Kafka producers
Kafka
Flink
Kafka consumers (& producers)
Persistence layer

Figure 7: Streaming access pattern showing Flink doing the job of both Storm and Spark Streaming in the use case

As shown in Figure 5, the first level of integration is between the streaming event source and Kafka. This is done by writing Kafka producers that send events to Kafka. The second level of integration is between Kafka and the downstream stream processing systems. The stream processing systems consume events from Kafka, using Kafka consumers, that are written by the user. The processing systems can also write data back into Kafka by implementing Kafka producers. They write data back into Kafka if the output of the stream processing system needs to be put back into the message queue for asynchronous consumption by more than one system thereafter. This approach

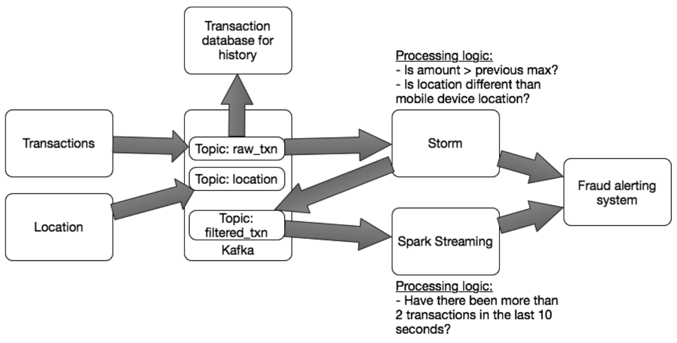

Transaction database for history
Processing logic:
- Is amount > previous max?
- Is location different than mobile device location?
Transactions
Location
Topic: raw_txn
Topic: location
Topic: filtered_txn
Kafka
Storm
Spark Streaming
Fraud alerting system
Processing logic:
- Have there been more than 2 transactions in the last 10 seconds?

Figure 8: Streaming architecture for detecting fraudulent transactions

offers more flexibility and scalability than a tight wiring between the stream processing system and the downstream persistence layer.

In Figure 5, a possible access pattern is that Storm consumes events from Kafka first, does event-level filtering, enrichment, and alerting, with latencies below 100 ms, and writes the processed events back to Kafka in a separate Kafka topic. Thereafter, a windowing function is implemented in Spark Streaming that consumes the output of the Storm topology from Kafka. Kafka becomes the central piece of this architecture where raw data, intermediate data as well as processed data sets land. Kafka makes for a good hub for streaming data. In this case, the output of the windowing function in Spark Streaming is charted onto graphs and not necessarily persisted anywhere. The filtered events (that were output by Storm into Kafka) are what go into a downstream persistence layer like the Hadoop Distributed File System, Apache HBase, etc. That system would look as shown in Figure 6.

Flink can handle both access patterns, and the above architecture could look like Figure 7 with Flink, eliminating the need to have two downstream stream processing engines.

Let's apply this to a specific (hypothetical) use case—detecting and flagging fraudulent credit card transactions. The source streams for this use case would be the following:

- Transaction information coming in from point-of-sale devices of the merchant
- Mobile device location of the customer

For the sake of the discussion, we'll use the following definition of a fraudulent transaction. These make up the rules for our stream processing application.

1. Two or more transactions performed in a span of 10 seconds
2. Transaction amount greater than the previous max done by the given customer
3. If the mobile device location of the customer is different from the location of the transaction

To solve this use case, we need two kinds of access patterns:

1. Transaction-level processing to detect breach of rules 2 and 3
2. Detection of breach of rule 1 over a period of time, potentially across multiple transactions

You could implement this architecture as shown in Figure 8.

Note that this is a hypothetical case to show how the different systems would be used together to solve the complete problem.

Conclusion

More organizations are incorporating streaming in their data pipelines. We discussed Kafka for stream ingestion and Spark, Storm, and Flink for stream analytics. Using the right mix of streaming systems and architectures based on the use case leads to scalable and successful implementations. We hope this article provides enough information for you to select, architect, and start implementing your streaming systems.

References

[1] Apache Kafka—http://kafka.apache.org/.

[2] Apache Spark Streaming—http://spark.apache.org/streaming/.

[3] Apache Storm—http://storm.apache.org/.

[4] Apache Flink—https://flink.apache.org/.

[5] Apache Spark—https://spark.apache.org/.

[6] Trident—http://storm.apache.org/documentation/Trident-tutorial.html.

[7] Apache Hadoop YARN—https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

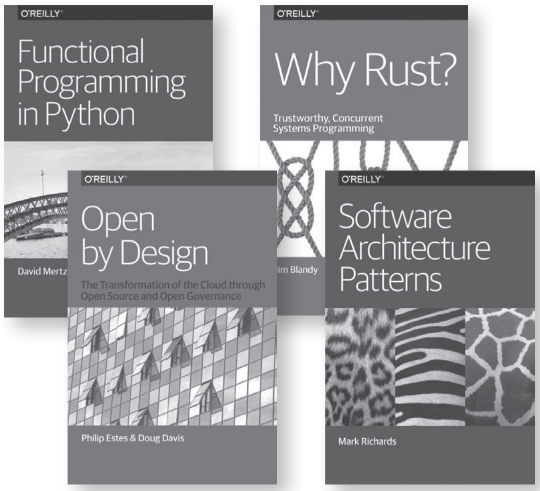
[8] Apache Mesos—http://mesos.apache.org/.

[9] Spark RDDs—http://spark.apache.org/docs/latest/programming-guide.html#resilient-distributed-datasets-rdds.

[10] Heron stream processing system by Twitter—https://blog.twitter.com/2015/flying-faster-with-twitter-heron.

Short. Smart. Seriously Useful.

Free open source and programming ebooks from O'Reilly.


O'REILLY
Functional Programming in Python
David Mertz
Why Rust?
Trustworthy, Concurrent Systems Programming
Jim Blandy
Open by Design
The Transformation of the Cloud through Open Source and Open Governance
Philip Estes & Doug Davis
Software Architecture Patterns
Mark Richards

Looking to stay current with the latest developments in open source, programming, and software engineering? We've got you covered. Get expert insights and industry research on topics like *Functional Programming in Python*, *Open by Design*, *Software Architecture Patterns*, and *Why Rust?* Download a couple—or all of them—today. Did we mention **free**?

Visit **oreilly.com/go/usenix**

O'REILLY®

©2016 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. D1705