Apache Samza

Martin Kleppmann

Definition

Apache Samza is an open source framework for distributed processing of high-volume event streams. Its primary design goal is to support high throughput for a wide range of processing patterns, while providing operational robustness at the massive scale required by Internet companies. Samza achieves this goal through a small number of carefully designed abstractions: partitioned logs for messaging, fault-tolerant local state, and cluster-based task scheduling.

Overview

Stream processing is playing an increasingly important part of the data management needs of many organizations. Event streams can represent many kinds of data, for example, the activity of users on a website, the movement of goods or vehicles, or the writes of records to a database.

Stream processing jobs are long-running processes that continuously consume one or more event streams, invoking some application logic on every event, producing derived output streams, and potentially writing output to databases for subsequent querying. While a batch process or a database query typically reads the state of a dataset at one point in time, and then finishes, a stream processor is never finished: it continually awaits the arrival of new events, and it only shuts down when terminated by an administrator.

Many tasks can be naturally expressed as stream processing jobs, for example:

- aggregating occurrences of events, e.g., counting how many times a particular item has been viewed;
- computing the rate of certain events, e.g., for system diagnostics, reporting, and abuse prevention;

- enriching events with information from a database, e.g., extending user click events with information about the user who performed the action;
- joining related events, e.g., joining an event describing an email that was sent with any events describing the user clicking links in that email;
- updating caches, materialized views, and search indexes, e.g., maintaining an external full-text search index over text in a database;
- using machine learning systems to classify events, e.g., for spam filtering.

Apache Samza, an open source stream processing framework, can be used for any of the above applications (Kleppmann and Kreps, 2015; Noghabi et al, 2017). It was originally developed at LinkedIn, then donated to the Apache Software Foundation in 2013, and became a top-level Apache project in 2015. Samza is now used in production at many Internet companies, including LinkedIn (Paramasivam, 2016), Netflix (Netflix Technology Blog, 2016), Uber (Chen, 2016; Hermann and Del Balso, 2017), and TripAdvisor (Calisi, 2016).

Samza is designed for usage scenarios that require very high throughput: in some production settings, it processes millions of messages per second or trillions of events per day (Feng, 2015; Paramasivam, 2016; Noghabi et al, 2017). Consequently, the design of Samza prioritizes scalability and operational robustness above most other concerns.

The core of Samza consists of several fairly low-level abstractions, on top of which high-level operators have been built (Pathirage et al, 2016). However, the core abstractions have been carefully designed for operational robustness, and the scalability of Samza is directly attributable to the choice of these foundational abstractions. The remainder of this article provides further detail on those design decisions and their practical consequences.

Partitioned Log Processing

A Samza job consists of a set of Java Virtual Machine (JVM) instances, called *tasks*, that each processes a subset of the input data. The code running in each JVM comprises the Samza framework and user code that implements the required application-specific functionality. The primary API for user code is the Java interface `StreamTask`, which defines a method `process()`. Figure 1 shows two examples of user classes implementing the `StreamTask` interface.

Once a Samza job is deployed and initialized, the framework calls the `process()` method once for every message in any of the input streams. The execution of this method may have various effects, including querying or updating local state and sending messages to output streams. This model of computation is closely analogous to a map task in the well-known MapReduce programming model (Dean and Ghemawat, 2004), with the difference that a Samza job's input is typically never-ending (*unbounded*).

Similarly to MapReduce, each Samza task is a single-threaded process that iterates over a sequence of input records. The inputs to a Samza job are partitioned into disjoint subsets, and each input partition is assigned to exactly one processing task. More than one partition may be assigned to the same processing task, in

```
class SplitWords implements StreamTask {

  static final SystemStream WORD_STREAM =
    new SystemStream("kafka", "words");

  public void process(
        IncomingMessageEnvelope in,
        MessageCollector out,
        TaskCoordinator _) {

    String str = (String) in.getMessage();

    for (String word : str.split(" ")) {
      out.send(
        new OutgoingMessageEnvelope(
          WORD_STREAM, word, 1));
    }
  }
}
```

```
class CountWords implements StreamTask,
                            InitableTask {

  private KeyValueStore<String, Integer> store;

  public void init(Config config,
                   TaskContext context) {
    store = (KeyValueStore<String, Integer>)
      context.getStore("word-counts");
  }

  public void process(
        IncomingMessageEnvelope in,
        MessageCollector out,
        TaskCoordinator _) {

    String word = (String) in.getKey();
    Integer inc = (Integer) in.getMessage();

    Integer count = store.get(word);
    if (count == null) count = 0;
    store.put(word, count + inc);
  }
}
```

Fig. 1 The two operators of a streaming word-frequency counter using Samza's *StreamTask* API (Image source: Kleppmann and Kreps (2015), © 2015 IEEE, reused with permission)

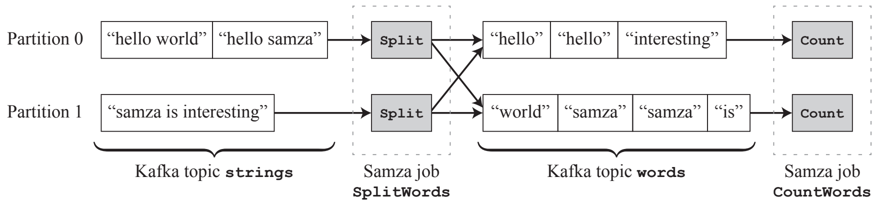

Fig. 2 A Samza task consumes input from one partition, but can send output to any partition (Image source: Kleppmann and Kreps (2015), © 2015 IEEE, reused with permission)

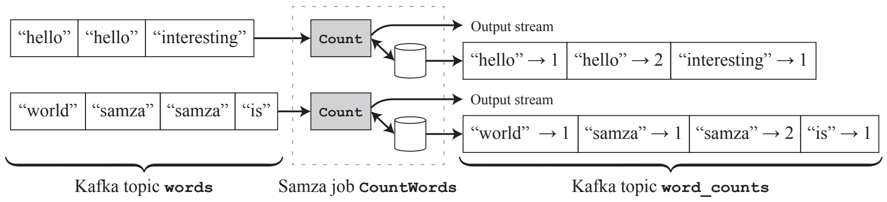

Fig. 3 A task's local state is made durable by emitting a changelog of key-value pairs to Kafka (Image source: Kleppmann and Kreps (2015), © 2015 IEEE, reused with permission)

which case the processing of those partitions is interleaved on the task thread. However, the number of partitions in the input determines the job's maximum degree of parallelism.

The log interface assumes that each partition of the input is a totally ordered sequence of records and that each record is associated with a monotonically increasing sequence number or identifier (known as *offset*). Since the records in each partition are read sequentially, a job can track its progress by periodically writing the offset of the last read record to durable storage. If a stream processing task is restarted, it resumes consuming the input from the last recorded offset.

Most commonly, Samza is used in conjunction with Apache Kafka (see separate article on Kafka). Kafka provides a partitioned, fault-tolerant log that allows publishers to append messages to a log partition, and consumers (subscribers) to sequentially read the messages in a log partition (Wang et al, 2015; Kreps et al, 2011; Goodhope et al, 2012). Kafka also allows stream processing jobs to reprocess previously seen records by resetting the consumer offset to an earlier position, a fact that is useful during recovery from failures.

However, Samza's stream interface is pluggable: besides Kafka, it can use any storage or messaging system as input, provided that the system can adhere to the partitioned log interface. By default, Samza can also read files from the Hadoop Distributed Filesystem (HDFS) as input, in a way that parallels MapReduce jobs, at competitive performance (Noghabi et al, 2017). At LinkedIn, Samza is commonly deployed with *Databus* inputs: Databus is a change data capture technology that records the log of writes to a database and makes this log available for applications to consume (Das et al, 2012; Qiao et al, 2013). Processing the stream of writes to a database enables jobs to maintain external indexes or materialized views onto data in a database and is especially relevant in conjunction with Samza's support for local state (see Section "Fault-tolerant Local State").

While every partition of an input stream is assigned to one particular task of a Samza job, the output partitions are not bound to tasks. That is, when a `StreamTask` emits output messages, it can assign them to any partition of the output stream. This fact can be used to group related data items into the same partition: for example, in the word-counting application illustrated in Figure 2, the `SplitWords` task chooses the output partition for each word based on a hash of the word. This ensures that when different tasks encounter occurrences of the same word, they are all written to the same output partition, from where a downstream job can read and aggregate the occurrences.

When stream tasks are composed into multistage processing pipelines, the output of one task becomes the input to another task. Unlike many other stream processing frameworks, Samza does not implement its own message transport layer to deliver messages between stream operators. Instead, Kafka is used for this purpose; since Kafka writes all messages to disk, it provides a large buffer between stages of the processing pipeline, limited only by the available disk space on the Kafka brokers.

Typically, Kafka is configured to retain several days or weeks worth of messages in each topic. Thus, if one stage of a processing pipeline fails or begins to run slow, Kafka can simply buffer the input to that stage while leaving ample time for the problem to be resolved. Unlike system designs based on backpressure, which require a producer to slow down if the consumer cannot keep up, the failure of one Samza job does not affect any upstream jobs that produce its inputs. This fact is crucial for the robust operation of large-scale systems, since it provides fault containment: as far as possible, a fault in one part of the system does not negatively impact other parts of the system.

Messages are dropped only if the failed or slow processing stage is not repaired within the retention period of the Kafka topic. In this case, dropping messages is desirable because it isolates the fault: the alternative – retaining messages indefinitely until the job is repaired – would lead to resource exhaustion (running out of memory or disk space), which would cause a cascading failure affecting unrelated parts of the system.

Thus, Samza's design of using Kafka's on-disk logs for message transport is a crucial factor in its scalability: in a large organization, it is often the case that an event stream produced by one team's job is consumed by one or more jobs that are administered by other teams. The jobs may be operating at different levels of maturity: for example, a stream produced by an important production job may be consumed by several unreliable experimental jobs. Using Kafka as a buffer between jobs ensures that adding an unreliable consumer does not negatively impact the more important jobs in the system.

Finally, an additional benefit of using Kafka for message transport is that every message stream in the system is accessible for debugging and monitoring: at any point, an additional consumer can be attached to inspect the message flow.

Fault-tolerant Local State

Stateless stream processing, in which any message can be processed independently from any other message, is easy to implement and scale. However, many important applications require that stream processing tasks maintain state. For example:

- when performing a join between two streams, a task must maintain an index of messages seen on each input within some time window, in order to find messages matching the join condition when they arrive;
- when computing a rate (number of events per time interval) or aggregation (e.g., sum of a particular field), a task must maintain the current aggregate value and update it based on incoming events;
- when processing an event requires a database query to look up some related data (e.g., looking up a user record for the user who performed the action in the event), the database can also be regarded as stream processor state.

Many stream processing frameworks use transient state that is kept in memory in the processing task, for example, in a hash table. However, such state is lost when a task crashes or when a processing job is restarted (e.g., to deploy a new version). To make the state fault-tolerant, some frameworks such as Apache Flink periodically write checkpoints of the in-memory state to durable storage (Carbone et al, 2015); this approach is reasonable when the state is small, but it becomes expensive as the state grows (Noghabi et al, 2017).

Another approach, used, for example, by Apache Storm, is to use an external database or key-value store for any processor state that needs to be fault-tolerant. This approach carries a severe performance penalty: due to network latency, accessing a database on another node is orders of magnitude slower than accessing local in-process state (Noghabi et al, 2017). Moreover, a high-throughput stream processor can easily overwhelm the external database with queries; if the database is shared with other applications, such overload risks harming the performance of other applications to the point that they become unavailable (Kreps, 2014).

In response to these problems, Samza pioneered an approach to managing state in a stream task that avoids the problems of both checkpointing and remote databases. Samza's approach to providing fault-tolerant local state has subsequently been adopted in the Kafka Streams framework (see article on Apache Kafka).

Samza allows each task to maintain state on the local disk of the processing node, with an in-memory cache for frequently accessed items. By default, Samza uses RocksDB, an embedded key-value store that is loaded into the JVM process of the stream task, but other storage engines can also be plugged in its place. In Figure 1, the `CountWords` task accesses this managed state through the `KeyValueStore` interface. For workloads with good locality, Samza's RocksDB with cache provides performance close to in-memory stores; for random-access workloads on large state, it remains significantly faster than accessing a remote database (Noghabi et al, 2017).

If a job is cleanly shut down and restarted, for example, to deploy a new version, Samza's host affinity feature tries to launch each StreamTask instance on the machine that has the appropriate RocksDB store on its local disk (subject to available resources). Thus, in most cases the state survives task restart without any further action. However, in some cases – for example, if a processing node suffers a full system failure – the state on the local disk may be lost or rendered inaccessible.

In order to survive the loss of local disk storage, Samza again relies on Kafka. For each store containing state of a stream task, Samza creates a Kafka topic called a *changelog* that serves as a replication log for the store. Every write to the local RocksDB store is also encoded as a message and published to this topic, as illustrated in Figure 3. These writes can be performed asynchronously in batches, enabling much greater throughput than synchronous random-access requests to a remote data store. The write queue only needs to be flushed when the offsets of input streams are written to durable storage, as described in the last section.

When a Samza task needs to recover its state after the loss of local storage, it reads all messages in the appropriate partition of the changelog topic and applies them to a new RocksDB store. When this process completes, the result is a new copy of the store that contains the same data as the store that was lost. Since Kafka replicates all data across multiple nodes, it is suitable for fault-tolerant durable storage of this changelog.

If a stream task repeatedly writes new values for the same key in its local storage, the changelog contains many redundant messages, since only the most recent value for a given key is required in order to restore local storage. To remove this redundancy, Samza uses a Kafka feature called *log compaction* on the changelog topic. With log compaction enabled, Kafka runs a background process that searches for messages with the same key and discards all but the most recent of those messages. Thus, whenever a key in the store is overwritten with a new value, the old value is eventually removed from the changelog. However, any key that is not overwritten is retained indefinitely by Kafka. This compaction process, which is very similar to internal processes in log-structured storage engines, ensures that the storage cost and recovery time from a changelog corresponds to the size of the state, independently of the total number of messages ever sent to the changelog (Kleppmann, 2017).

Cluster-based Task Scheduling

When a new stream processing job is started, it must be allocated computing resources: CPU cores, RAM, disk space, and network bandwidth. Those resources may need to be adjusted from time to time as load varies and reclaimed when a job is shut down.

At large organizations, hundreds or thousands of jobs need to run concurrently. At such scale, it is not practical to manually assign resources: task scheduling and resource allocation must be automated. To maximize hardware utilization, many jobs and applications are deployed to a shared pool of machines, with each multi-core machine typically running a mixture of tasks from several different jobs.

This architecture requires infrastructure for managing resources and for deploying the code of processing jobs to the machines on which it is to be run. Some frameworks, such as Storm and Flink, have built-in mechanisms for resource management and deployment. However, frameworks that perform their own task scheduling and cluster management generally require a static assignment of computing resources – potentially even dedicated machines – before any jobs can be deployed to the cluster. This static resource allocation leads to inefficiencies in machine utilization and limits the ability to scale on demand (Kulkarni et al, 2015).

By contrast, Samza relies on existing cluster management software, which allows Samza jobs to share a pool of machines with non-Samza applications. Samza supports two modes of distributed operation:

- A job can be deployed to a cluster managed by Apache Hadoop YARN (Vavilapalli et al, 2013). YARN is a general-purpose resource scheduler and cluster manager that can run stream processors, MapReduce batch jobs, data analytics engines, and various other applications on a shared cluster. Samza jobs can be deployed to existing YARN clusters without requiring any special cluster-level configuration or resource allocation.
- Samza also supports a *stand-alone* mode in which a job's JVM instances are deployed and executed through some external process that is not under Samza's control. In this case, the instances use Apache ZooKeeper

(Junqueira et al, 2011) to coordinate their work, such as assigning partitions of the input streams.

The stand-alone mode allows Samza to be integrated with an organization's existing deployment and cluster management tools, or with cloud computing platforms: for example, Netflix runs Samza jobs directly as EC2 instances on Amazon Web Services (AWS), relying on the existing cloud facilities for resource allocation (Paramasivam, 2016). Moreover, Samza's cluster management interface is pluggable, enabling further integrations with other technologies such as Mesos (Hindman et al, 2011).

With large deployments, an important concern is resource isolation, that is, ensuring that each process receives the resources it requested and that a misbehaving process cannot starve colocated processes of resources. When running in YARN, Samza supports the Linux cgroups feature to enforce limits on the CPU and memory use of stream processing tasks. In virtual machine environments such as EC2, resource isolation is enforced by the hypervisor.

Summary

Apache Samza is a stream processing framework that is designed to provide high throughput and operational robustness at very large scale. Efficient resource utilization requires a mixture of different jobs to share a multi-tenant computing infrastructure. In such an environment, the primary challenge in providing robust operation is fault isolation, that is, ensuring that a faulty process cannot disrupt correctly running processes and that a resource-intensive process cannot starve others.

Samza isolates stream processing jobs from each other in several different ways. By using Kafka's on-disk logs as a large buffer between producers and consumers of a stream, instead of backpressure, Samza ensures that a slow or failed consumer does not affect upstream jobs. By providing fault-tolerant local state as a common abstraction, Samza improves performance and avoids reliance on external databases that might be overloaded by high query volume. Finally, by integrating with YARN and other cluster managers, Samza builds upon existing resource scheduling and isolation technology that allows a cluster to be shared between many different applications without risking resource starvation.

References

Calisi L (2016) How to convert legacy Hadoop Map/Reduce ETL systems to Samza Streaming. URL https://www.youtube.com/watch?v=KQ5OnL2hMBY

Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K (2015) Apache Flink: Stream and batch processing in a single engine. IEEE Data Engineering Bulletin 38(4):28–38, URL http://sites.computer.org/debull/A15dec/p28.pdf

Chen S (2016) Scalable complex event processing on Samza @Uber. URL https://www.slideshare.net/ShuyiChen2/scalable-complex-event-processing-on-samza-uber

Das S, Botev C, Surlaker K, Ghosh B, Varadarajan B, Nagaraj S, Zhang D, Gao L, Westerman J, Ganti P, Shkolnik B, Topiwala S, Pachev A, Somasundaram N, Subramaniam S (2012) All aboard the Databus! LinkedIn's scalable consistent change data capture platform. In: 3rd ACM Symposium on Cloud Computing (SoCC), DOI 10.1145/2391229.2391247

Dean J, Ghemawat S (2004) MapReduce: Simplified data processing on large clusters. In: 6th USENIX Symposium on Operating System Design and Implementation (OSDI)

Feng T (2015) Benchmarking Apache Samza: 1.2 million messages per second on a single node. URL http://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node

Goodhope K, Koshy J, Kreps J, Narkhede N, Park R, Rao J, Ye VY (2012) Building LinkedIn's real-time activity data pipeline. IEEE Data Engineering Bulletin 35(2):33–45, URL http://sites.computer.org/debull/A12june/A12JUN-CD.pdf

Hermann J, Del Balso M (2017) Meet Michelangelo: Uber's machine learning platform. URL https://eng.uber.com/michelangelo/

Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, Shenker S, Stoica I (2011) Mesos: A platform for fine-grained resource sharing in the data center. In: 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)

Junqueira FP, Reed BC, Serafini M (2011) Zab: High-performance broadcast for primary-backup systems. In: 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp 245–256, DOI 10.1109/DSN.2011.5958223

Kleppmann M (2017) Designing Data-Intensive Applications. O'Reilly Media, ISBN 978-1-4493-7332-0

Kleppmann M, Kreps J (2015) Kafka, Samza and the Unix philosophy of distributed data. IEEE Data Engineering Bulletin 38(4):4–14, URL http://sites.computer.org/debull/A15dec/p4.pdf

Kreps J (2014) Why local state is a fundamental primitive in stream processing. URL https://www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing

Kreps J, Narkhede N, Rao J (2011) Kafka: a distributed messaging system for log processing. In: 6th International Workshop on Networking Meets Databases (NetDB)

Kulkarni S, Bhagat N, Fu M, Kedigehalli V, Kellogg C, Mittal S, Patel JM, Ramasamy K, Taneja S (2015) Twitter Heron: Stream processing at scale. In: ACM International Conference on Management of Data (SIGMOD), pp 239–250, DOI 10.1145/2723372.2723374

Netflix Technology Blog (2016) Kafka inside Keystone pipeline. URL http://techblog.netflix.com/2016/04/kafka-inside-keystone-pipeline.html

Noghabi SA, Paramasivam K, Pan Y, Ramesh N, Bringhurst J, Gupta I, Campbell RH (2017) Samza: Stateful scalable stream processing at LinkedIn. Proceedings of the VLDB Endowment 10(12):1634–1645, DOI 10.14778/3137765.3137770

Paramasivam K (2016) Stream processing with Apache Samza – current and future. URL https://engineering.linkedin.com/blog/2016/01/whats-new-samza

Pathirage M, Hyde J, Pan Y, Plale B (2016) SamzaSQL: Scalable fast data management with streaming sql. In: IEEE International Workshop on High-Performance Big Data Computing (HPBDC), pp 1627–1636, DOI 10.1109/IPDPSW.2016.141

Qiao L, Auradar A, Beaver C, Brandt G, Gandhi M, Gopalakrishna K, Ip W, Jgadish S, Lu S, Pachev A, Ramesh A, Surlaker K, Sebastian A, Shanbhag R, Subramaniam S, Sun Y, Topiwala S, Tran C, Westerman J, Zhang D, Das S, Quiggle T, Schulman B, Ghosh B, Curtis A, Seeliger O, Zhang Z (2013) On brewing fresh Espresso: LinkedIn's distributed data serving platform. In: ACM International Conference on Management of Data (SIGMOD), pp 1135–1146, DOI 10.1145/2463676.2465298

Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E (2013) Apache Hadoop YARN: Yet another resource negotiator. In: 4th ACM Symposium on Cloud Computing (SoCC), DOI 10.1145/2523616.2523633

Wang G, Koshy J, Subramanian S, Paramasivam K, Zadeh M, Narkhede N, Rao J, Kreps J, Stein J (2015) Building a replicated logging system with Apache Kafka. Proceedings of the VLDB Endowment 8(12):1654–1655, DOI 10.14778/2824032.2824063

Cross-References

Apache Flink
Apache Heron
Apache Kafka
Continuous Queries
Publish/Subscribe