Kafka, Samza and the Unix Philosophy of Distributed Data

Martin Kleppmann
University of Cambridge
Computer Laboratory

Jay Kreps
Confluent, Inc.

Abstract

Apache Kafka is a scalable message broker, and Apache Samza is a stream processing framework built upon Kafka. They are widely used as infrastructure for implementing personalized online services and real-time predictive analytics. Besides providing high throughput and low latency, Kafka and Samza are designed with operational robustness and long-term maintenance of applications in mind. In this paper we explain the reasoning behind the design of Kafka and Samza, which allow complex applications to be built by composing a small number of simple primitives – replicated logs and stream operators. We draw parallels between the design of Kafka and Samza, batch processing pipelines, database architecture, and the design philosophy of Unix.

1 Introduction

In recent years, online services have become increasingly personalized. For example, in a service such as LinkedIn there are many activity-based feedback loops, automatically adapting the site to make it more relevant to individual users: recommendation systems such as "people you may know" or "jobs you might be interested in" [30], collaborative filtering [33] or ranking of search results [23, 26] are personalized based on analyses of user behavior (e.g. click-through rates of links) and user-profile information. Other feedback loops include abuse prevention (e.g. blocking spammers, fraudsters and other users who violate the terms of service), A/B tests and user-facing analytics (e.g. "who viewed your profile").

Such personalization makes a service better for users, as they are likely to find what they need faster than if the service presented them with static information. However, personalization has also opened new challenges: a huge amount of data about user activity needs to be collected, aggregated and analyzed [8]. Timeliness is important: after the service learns a new fact, the personalized recommendations and rankings should be swiftly updated to reflect the new fact, otherwise their utility is diminished.

In this paper we describe Kafka and Samza, two related projects that were originally developed at LinkedIn as infrastructure for solving these data collection and processing problems. The projects are now open source, and maintained within the Apache Software Foundation as Apache Kafka[1] and Apache Samza[2], respectively.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] `http://kafka.apache.org/`
[2] `http://samza.apache.org/`

1.1 Implementing Large-Scale Personalized Services

In a large-scale service with many features, the maintainability and the operational robustness of an implementation are of paramount importance. The system should have the following properties:

System scalability: Supporting an online service with hundreds of millions of registered users, handling millions of requests per second.

Organizational scalability: Allowing hundreds or even thousands of software engineers to work on the system without excessive coordination overhead.

Operational robustness: If one part of the system is slow or unavailable, the rest of the system should continue working normally as much as possible.

Large-scale personalized services have been successfully implemented as batch jobs [30], for example using MapReduce [6]. Performing a recommendation system's computations in offline batch jobs decouples them from the online systems that serve user requests, making them easier to maintain and less operationally sensitive.

The main downside of batch jobs is that they introduce a delay between the time the data is collected and the time its effects are visible. The length of the delay depends on the frequency with which the job is run, but it is often on the order of hours or days.

Even though MapReduce is a lowest-common-denominator programming model, and has fairly poor performance compared to specialized massively parallel database engines [2], it has been a remarkably successful tool for implementing recommendation systems [30]. Systems such as Spark [34] overcome some of the performance problems of MapReduce, although they remain batch-oriented.

1.2 Batch Workflows

A recommendation and personalization system can be built as a *workflow*, a directed graph of MapReduce jobs [30]. Each job reads one or more input datasets (typically directories on the Hadoop Distributed Filesystem, HDFS), and produces one or more output datasets (in other directories). A job treats its input as immutable and completely replaces its output. Jobs are chained by directory name: the same name is configured as output directory for the first job and input directory for the second job.

This method of chaining jobs by directory name is simple, and is expensive in terms of I/O, but it provides several important benefits:

Multi-consumer. Several different jobs can read the same input directory without affecting each other. Adding a slow or unreliable consumer affects neither the producer of the dataset, nor other consumers.

Visibility. Every job's input and output can be inspected by ad-hoc debugging jobs for tracking down the cause of an error. Inspection of inputs and outputs is also valuable for audit and capacity planning purposes, and monitoring whether jobs are providing the required level of service.

Team interface. A job operated by one team of people can produce a dataset, and jobs operated by other teams can consume the dataset. The directory name thus acts as interface between the teams, and it can be reinforced with a contract (e.g. prescribing the data format, schema, field semantics, partitioning scheme, and frequency of updates). This arrangement helps organizational scalability.

Loose coupling. Different jobs can be written in different programming languages, using different libraries, but they can still communicate as long as they can read and write the same file format for inputs and outputs. A job does not need to know which jobs produce its inputs and consume its outputs. Different jobs can be run on different schedules, at different priorities, by different users.

Data provenance. With explicitly named inputs and outputs for each job, the flow of data can be tracked through the system. A producer can identify the consumers of its dataset (e.g. when making forward-incompatible changes), and a consumer can identify its transitive data sources (e.g. in order to ensure regulatory compliance).

Failure recovery. If the 46th job in a chain of 50 jobs failed due to a bug in the code, it can be fixed and restarted at the 46th job. There is no need to re-run the entire workflow.

Friendly to experimentation. Most jobs modify only to their designated output directories, and have no other externally visible side-effects such as writing to external databases. Therefore, a new version of a job can easily be run with a temporary output directory for testing purposes, without affecting the rest of the system.

1.3 From Batch to Streaming

When moving from a high-latency batch system to a low-latency streaming system, we wish to preserve the attractive properties listed in Section 1.2.

By analogy, consider how Unix tools are composed into complex programs using shell scripts [21]. A workflow of batch jobs is comparable to a shell script in which there is no pipe operator, so each program must read its input from a file on disk, and write its output to a different (temporary) file on disk. In this scenario, one program must finish writing its output file before another program can start reading that file.

To move from a batch workflow to a streaming data pipeline, the temporary files would need to be replaced with something more like Unix pipes, which support incrementally passing one program's output to another program's input without fully materializing the intermediate result [1]. However, Unix pipes do not have all the properties we want: they connect exactly one output to exactly one input (not multi-consumer), and they cannot be repaired if one of the processes crashes and restarts (no failure recovery).

Kafka and Samza provide infrastructure for low-latency *distributed stream processing* in a style that resembles a chain of Unix tools connected by pipes, while also preserving the aforementioned benefits of chained batch jobs. In the following sections we will discuss the design decisions that this approach entails.

1.4 Relationship of Kafka and Samza

Kafka and Samza are two separate projects with a symbiotic relationship. Kafka provides a message broker service, and Samza provides a framework for processing messages. A Samza job uses the Kafka client library to consume input streams from the Kafka message broker, and to produce output streams back to Kafka. Although either system can be used without the other, they work best together. We introduce Kafka in more detail in Section 2, and Samza in Section 3.

At the time of writing, there is an effort underway to add a feature called *Kafka Streams* to the Kafka client library [31]. This feature provides a stream processing capability similar to Samza, but it differs in that Kafka Streams does not prescribe a deployment mechanism, whereas Samza currently relies on Hadoop YARN. Most other high-level architecture choices are similar in Samza and Kafka Streams; for purposes of this paper, they can be regarded as equivalent.

2 Apache Kafka

Kafka has been described in detail in prior work [8, 16, 19, 32]. In this section we present a brief high-level overview of the principles behind Kafka's design.

Kafka provides a publish-subscribe messaging service, as illustrated in Figure 1. Producer (publisher) clients write messages to a named *topic*, and consumer (subscriber) clients read messages in a topic.

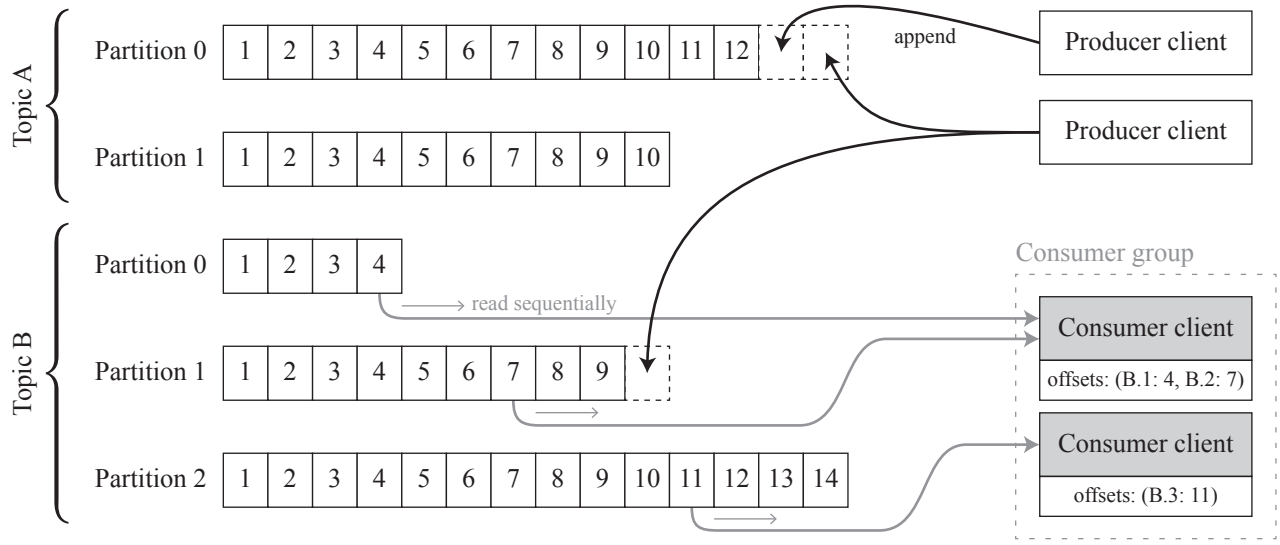

Figure 1: A Kafka *topic* is divided into *partitions*, and each partition is a totally ordered sequence of *messages*.

A topic is divided into *partitions*, and messages within a partition are totally ordered. There is no ordering guarantee across different partitions. The purpose of partitioning is to provide horizontal scalability: different partitions can reside on different machines, and no coordination across partitions is required. The assignment of messages to partitions may be random, or it may deterministic based on a key, as described in Section 3.2.

Broker nodes (Kafka servers) store all messages on disk. Each partition is physically stored as a series of segment files that are written in an append-only manner. A Kafka partition is also known as a *log*, since it resembles a database's transaction commit log [12]: whenever a new message is published to a topic, it is appended to the end of the log. The Kafka broker assigns an *offset* to the message, which is a per-partition monotonically increasing sequence number.

A message in Kafka consists of a key and a value, which are untyped variable-length byte strings. For richer datatypes, any encoding can be used. A common choice is Apache Avro,[3] a binary encoding that uses explicit schemas to describe the structure of messages in a topic, providing a statically typed (but evolvable) interface between producers and consumers [10, 15].

A Kafka consumer client reads all messages in a topic-partition sequentially. For each partition, the client tracks the offset up to which it has seen messages, and it polls the brokers to await the arrival of messages with a greater offset (akin to the Unix tool `tail -f`, which watches a file for appended data). The offset is periodically checkpointed to stable storage; if a consumer client crashes and restarts, it resumes reading from its most recently checkpointed offset.

Each partition is replicated across multiple Kafka broker nodes, so that the system can tolerate the failure of nodes without unavailability or data loss. One of a partition's replicas is chosen as *leader*, and the leader handles all reads and writes of messages in that partition. Writes are serialized by the leader and synchronously replicated to a configurable number of replicas. On leader failure, one of the in-sync replicas is chosen as the new leader.

2.1 Performance and Scalability

Kafka can write millions of messages per second on modest commodity hardware [14], and the deployment at LinkedIn handles over 1 trillion unique messages per day [20]. Message length is typically low hundreds of

[3] `http://avro.apache.org/`

bytes, although smaller or larger messages are also supported.

In many deployments, Kafka is configured to retain messages for a week or longer, limited only by available disk space. Segments of the log are discarded when they are older than a configurable threshold. Alternatively, Kafka supports a *log compaction* mode, in which the latest message with a given key is retained indefinitely, but earlier messages with the same key are garbage-collected. Similar ideas are found in log-structured filesystems [25] and database storage engines [18].

When multiple producers write to the same topic-partition, their messages are interleaved, so there is no inherent limit to the number of producers. The throughput of a single topic-partition is limited by the computing resources of a single broker node – the bottleneck is usually either its NIC bandwidth or the sequential write throughput of the broker's disks. Higher throughput can be achieved by creating more partitions and assigning them to different broker nodes. As there is no coordination between partitions, Kafka scales linearly.

It is common to configure a Kafka cluster with approximately 100 topic-partitions per broker node [22]. When adding nodes to a Kafka cluster, some partitions can be reassigned to the new nodes, without changing the number of partitions in a topic. This rebalancing technique allows the cluster's computing resources to be increased or decreased without affecting partitioning semantics.

On the consumer side, the work of consuming a topic can be shared between a group of consumer clients (illustrated in Figure 1). One consumer client can read several topic-partitions, but any one topic-partition must be read sequentially by a consumer process – it is not possible to consume only a subset of messages in a partition. Thus, the maximum number of processes in a consumer group equals the number of partitions of the topic being consumed.

Different consumer groups maintain their offsets independently, so they can each read the messages at their own pace. Thus, like multiple batch jobs reading the same input directory, multiple groups of consumers can independently read the same Kafka topic without affecting each other.

3 Apache Samza

Samza is a framework that helps application developers write code to consume streams, process messages, and produce derived output streams. In essence, a Samza job consists of a Kafka consumer, an event loop that calls application code to process incoming messages, and a Kafka producer that sends output messages back to Kafka. In addition, the framework provides packaging, cluster deployment (using Hadoop YARN), automatically restarting failed processes, state management (Section 3.1), metrics and monitoring.

For processing messages, Samza provides a Java interface `StreamTask` that is implemented by application code. Figure 2 shows how to implement a streaming word counter with Samza: the first operator splits every input string into words, and the second operator counts how many times each word has been seen.

For a Samza job with one input topic, the framework instantiates one `StreamTask` for each partition of the input topic. Each task instance independently consumes one partition, no matter whether the instances are running in the same process, or distributed across multiple machines. As processing is always logically partitioned by input partition, even if several partitions are physically processed on the same node, a job's allocated computing resources can be scaled up or down without affecting partitioning semantics.

The framework calls the `process()` method for each input message, and the application code may emit any number of output messages as a result. Output messages can be sent to any partition, which allows re-partitioning data between jobs. For example, Figure 3 illustrates the use of partitions in the word-count example: by using the word as message key, the `SplitWords` task ensures that all occurrences of the same word are routed to the same partition of the `words` topic (analogous to the shuffle phase of MapReduce [6]).

Unlike many other stream-processing frameworks, Samza does not implement its own network protocol for transporting messages from one operator to another. Instead, a job usually uses one or more named Kafka topics as input, and other named Kafka topics as output. We discuss the implications of this design in Section 4.

```
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);
  }
}
```

Figure 2: The two operators of a streaming word-frequency counter using Samza's *StreamTask* API.

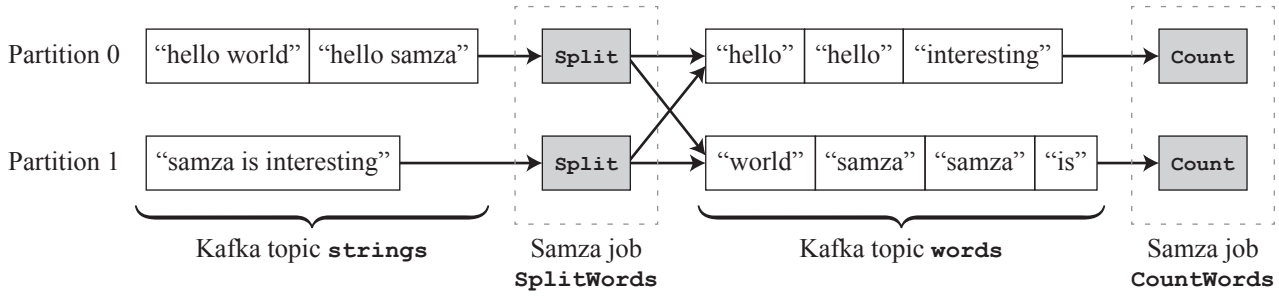

Figure 3: An instance of a Samza task consumes input from one partition, but can send output to any partition.

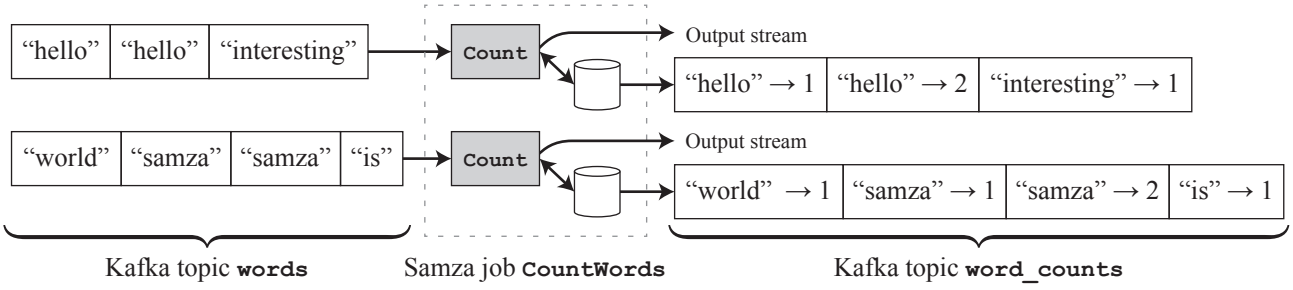

Figure 4: A task's local state is made durable by emitting a changelog to Kafka.

3.1 State Management

Many stream-processing jobs need to maintain state, e.g. in order to perform joins (Section 3.2) or aggregations (such as the counters in `CountWords`, Figure 2). Any transient state can simply be maintained in instance variables of the StreamTask; since messages of a partition are processed sequentially on a single thread, these data structures need not be thread-safe. However, any state that must survive the crash of a stream processor must be written to durable storage.

Samza implements durable state through the `KeyValueStore` abstraction, exemplified in Figure 2. Each StreamTask instance has a separate store that it can read and write as required. Samza uses the RocksDB[4] embedded key-value store, which provides low-latency, high-throughput access to data on local disk. To make the embedded store durable in the face of disk and node failures, every write to the store is also sent to a dedicated topic-partition in Kafka, as illustrated in Figure 4.

This changelog topic acts as a durable replication log for the store: when recovering after a failure, a task can rebuild its store contents by replaying its partition of the changelog from the beginning. Kafka's log compaction mode (see Section 2.1) prevents unbounded growth of the changelog topic: if the same key is repeatedly overwritten (as with a counter), Kafka eventually garbage-collects overwritten values, and retains the most recent value for any given key indefinitely. Rebuilding a store from the log is only necessary if the RocksDB database is lost or corrupted.

Writing the changelog to Kafka is not merely an efficient way of achieving durability, it can also be a useful feature for applications: other stream processing jobs can consume the changelog topic like any other stream, and use it to perform further computations. For example, the `word_counts` topic of Figure 4 could be consumed by another job to determine trending keywords (in this case, the changelog stream is also the `CountWords` operator's output – no separate output topic is required).

3.2 Stream Joins

One characteristic form of stateful processing is a join of two or more input streams, most commonly an equi-join on a key (e.g. user ID). One type of join is a *window join*, in which messages from input streams A and B are matched if they have the same key, and occur within some time interval Δt of one another. Alternatively, a stream may be joined against *tabular data*: for example, user clickstream events could be joined with user profile data, producing a stream of clickstream events with embedded information about the user.

Stream-table joins can be implemented by querying an external database within a StreamTask, but the network round-trip time for database queries soon becomes a bottleneck, and this approach can easily overload the external database [13]. A better option is to make the table data available in the form of a log-compacted stream. Processing tasks can consume this stream to build an in-process replica of a database table partition, using the same approach as the recovery of durable local state (Section 3.1), and then query it with low latency.

For example, in the case of a database of user profiles, the log-compacted stream would contain a snapshot of all user profiles as of some point in time, and an update message every time a user subsequently changes their profile information. Such a stream can be extracted from an existing database using change data capture [5, 32].

When joining partitioned streams, Samza expects that all input streams are partitioned in the same way, with the same number of partitions n, and deterministic assignment of messages to partitions based on the same join key. The Samza job then *co-partitions* its input streams: for any partition k (with $0 \leq k < n$), messages from partition k of input stream A and from partition k of input stream B are delivered to the same StreamTask instance. The task can then use local state to maintain the data that is required to perform the join.

Multi-way joins on several different keys may require different partitioning for each join. Such joins can be implemented with a multi-stage pipeline, where the output of each job partitions messages according to the next stage's join key. The same approach is used in MapReduce workflows.

[4] `http://rocksdb.org/`

4 Discussion

In Sections 2 and 3 we outlined the architecture of Kafka and Samza. We now examine the design decisions behind that architecture in the light of our goals discussed in Section 1, namely creating large-scale personalized services in a way that is scalable, maintainable and operationally robust.

4.1 Use of Replicated Logs

Stream processing with Samza relies heavily on fault-tolerant, partitioned logs as implemented by Kafka. Kafka topics are used for input, output, messaging between operators, durability of local state, replicating database tables, checkpointing consumer offsets, collecting metrics, and disseminating configuration information.

An append-only log with optional compaction is one of the simplest data structures that is useful in practice [12]. Kafka focuses on implementing logs in a fault-tolerant and scalable way. Since the only access methods supported by a log are an appending write and a sequential read from a given offset, Kafka avoids the complexity of implementing random-access indexes. By doing less work, Kafka is able to provide much better performance than systems with richer access methods [14, 16]. Kafka's focus on the log abstraction is reminiscent of the Unix philosophy [17]: "Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new 'features'."

Real systems do require indexes and caches, but these can be derived from the log by a Kafka consumer that writes messages to an indexed store, either in-process (for local access) or to a remote database (for access by other applications). Because all consumers see messages in the same partition in the same order, deterministic consumers can independently construct views that are consistent with each other – an approach known as state machine replication [27]. The truth is in the log, and a database is a cached subset of the log [9].

4.2 Composing Stream Operators

Each Samza job is structurally simple: it is just one step in a data processing pipeline, with Kafka topics as inputs and outputs. If Kafka is like a streaming version of HDFS, then Samza is like a streaming version of MapReduce. The pipeline is loosely coupled, since a job does not know the identity of the jobs upstream or downstream from it, only the topic names. This principle again evokes a Unix maxim [17]: "Expect the output of every program to become the input to another, as yet unknown, program."

However, there are some key differences between Kafka topics and Unix pipes. In particular, Kafka preserves the advantages of batch workflows discussed in Section 1.2: a topic can have any number of consumers that do not interfere with each other (including consumers operated by different teams, or special consumers for debugging or monitoring), it tolerates failure of producers, consumers or brokers, and a topic is a named entity that can be used for tracing data provenance.

Kafka topics deliberately do not provide backpressure: the on-disk log acts as an almost-unbounded buffer of messages. If a slow consumer falls behind the producer, the producers and other consumers continue operating at full speed. Thus, one faulty process does not disrupt the rest of the system, which improves operational reliability. Since Kafka stores all messages on disk anyway, buffering messages for a slow consumer does not incur additional overhead. The slow consumer can catch up without missing messages, as long as it does not fall behind further than Kafka's retention period of log segments, which is usually on the order of days or weeks.

Moreover, Kafka offers the ability for a consumer to jump back to an earlier point in the log, or to rebuild the entire state of a database replica by consuming from the beginning of a log-compacted topic. This facility makes it feasible to use stream processors not only for ephemeral event data, but also for database-like use cases.

Even though the intermediate state between two Samza stream processing operators is always materialized to disk, Samza is able to provide good performance: a simple stream processing job can process over 1 million messages per second on one machine, and saturate a gigabit Ethernet NIC [7].

4.3 Unix as a Role Model

Unix and databases are both data management systems [24], allowing data to be stored (in files or tables) and processed (through command-line tools or queries). Unix tools are famously well suited for implementing ad-hoc, experimental, short-running data processing tasks [21], whereas databases have traditionally been the tool of choice for building complex, long-lived applications. If our goal is to build stream processing applications that will run reliably for many years, is Unix really a good role model?

The database tradition favors clean high-level semantics (the relational model) and declarative query languages. While this approach has been very successful in many domains, it has not worked well in the context of building large-scale personalized services, because the algorithms required for these use cases (such as statistical machine learning and information retrieval methods) are not amenable to implementation using relational operators [28, 29].

Moreover, different use cases have different access patterns, which require different indexing and storage methods. It may be necessary to store the same data in both a traditional row-oriented fashion with indexes, as well as columnar storage, pre-aggregated OLAP cubes, inverted full-text search indexes, sparse matrices or array storage. Rather than trying to implement everything in a single product, most databases specialize in implementing one of these storage methods well (which is hard enough already).

In the absence of a single database system that can provide all the necessary functionality, application developers are forced to combine several data storage and processing systems that each provide a portion of the required application functionality. However, many traditional database systems are not designed for such composition: they focus on providing strong semantics internally, rather than integration with external systems. Mechanisms for integrating with external systems, such as change data capture, are often ad-hoc and retrofitted [5].

By contrast, the log-oriented model of Kafka and Samza is fundamentally built on the idea of composing heterogeneous systems through the uniform interface of a replicated, partitioned log. Individual systems for data storage and processing are encouraged to do one thing well, and to use logs as input and output. Even though Kafka's logs are not the same as Unix pipes, they encourage composability, and thus Unix-style thinking.

4.4 Limitations

Kafka guarantees a total ordering of messages per partition, even in the face of crashes and network failures. This guarantee is stronger than most "eventually consistent" datastores provide, but not as strong as serializable database transactions.

The stream-processing model of computation is fundamentally asynchronous: if a client issues a write to the log, and then reads from a datastore that is maintained by consuming the log, the read may return a stale value. This decoupling is desirable, as it prevents a slow consumer from disrupting a producer or other consumers (Section 4.2). If linearizable data structures are required, they can fairly easily be implemented on top of a totally ordered log [3].

If a Kafka consumer or Samza job crashes and restarts, it resumes consuming messages from the most recently checkpointed offset. Thus, any messages processed between the last checkpoint and the crash are processed twice, and any non-idempotent operations (such as the counter increment in `CountWords`, Figure 2) may yield non-exact results. There is work in progress to add a multi-partition atomic commit protocol to Kafka [11], which will allow exactly-once semantics to be achieved.

Samza uses a low-level one-message-at-a-time programming model, which is very flexible, but also harder to use, more error-prone and less amenable to automatic optimization than a high-level declarative query language. Work is currently in progress in the Kafka project to implement a high-level dataflow API called *Kafka Streams*, and the Samza project is developing a SQL query interface, with relational operators implemented as stream processing tasks. These higher-level programming models enable easier development of applications that fit the model, while retaining the freedom for applications to use the lower-level APIs when required.

5 Conclusion

We present the design philosophy behind Kafka and Samza, which implement stream processing by composing a small number of general-purpose abstractions. We draw analogies to the design of Unix, and batch processing pipelines. The approach reflects broader trends: the convergence between batch and stream processing [1, 4], and the decomposition of monolithic data infrastructure into a collection of specialized services [12, 28].

In particular, we advocate a style of application development in which each data storage and processing component focuses on "doing one thing well". Heterogeneous systems can be built by composing such specialised tools through the simple, general-purpose interface of a log. Compared to monolithic systems, such composable systems provide better scalability properties thanks to loose coupling, and allow easier adaptation of a system to a wide range of different workloads, such as recommendation systems.

Acknowledgements

Large portions of the development of Kafka and Samza were funded by LinkedIn. Many people have contributed, and the authors would like to thank the committers on both projects: David Arthur, Sriharsha Chintalapani, Yan Fang, Jakob Homan, Joel Koshy, Prashanth Menon, Neha Narkhede, Yi Pan, Navina Ramesh, Jun Rao, Chris Riccomini, Gwen Shapira, Zhijie Shen, Chinmay Soman, Joe Stein, Sriram Subramanian, Garry Turkington, and Guozhang Wang. Thank you to Garry Turkington, Yan Fang and Alastair Beresford for feedback on a draft of this article.

References

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, August 2015. doi:10.14778/2824032.2824076.

[2] Shivnath Babu and Herodotos Herodotou. Massively parallel databases and MapReduce systems. *Foundations and Trends in Databases*, 5(1):1–104, November 2013. doi:10.1561/1900000036.

[3] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, et al. Tango: Distributed data structures over a shared log. In *24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–340, November 2013. doi:10.1145/2517349.2522732.

[4] Raul Castro Fernandez, Peter Pietzuch, Jay Kreps, Neha Narkhede, et al. Liquid: Unifying nearline and offline big data integration. In *7th Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2015.

[5] Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh, et al. All aboard the Databus! In *3rd ACM Symposium on Cloud Computing (SoCC)*, October 2012.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[7] Tao Feng. Benchmarking Apache Samza: 1.2 million messages per second on a single node, August 2015. URL `http://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node`.

[8] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, et al. Building LinkedIn's real-time activity data pipeline. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 35(2):33–45, June 2012.

[9] Pat Helland. Immutability changes everything. In *7th Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2015.

[10] Martin Kleppmann. Schema evolution in Avro, Protocol Buffers and Thrift, December 2012. URL `http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html`.

[11] Joel Koshy. Transactional messaging in Kafka, July 2014. URL `https://cwiki.apache.org/confluence/display/KAFKA/Transactional+Messaging+in+Kafka`.

[12] Jay Kreps. *I Heart Logs*. O'Reilly Media, September 2014. ISBN 978-1-4919-0932-4.

[13] Jay Kreps. Why local state is a fundamental primitive in stream processing, July 2014. URL http://radar.oreilly.com/2014/07/why-local-state-is-a-fundamental-primitive-in-stream-processing.html.

[14] Jay Kreps. Benchmarking Apache Kafka: 2 million writes per second (on three cheap machines), April 2014. URL https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[15] Jay Kreps. Putting Apache Kafka to use: a practical guide to building a stream data platform (part 2), February 2015. URL http://blog.confluent.io/2015/02/25/stream-data-platform-2/.

[16] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *6th International Workshop on Networking Meets Databases (NetDB)*, June 2011.

[17] M D McIlroy, E N Pinson, and B A Tague. UNIX time-sharing system: Foreword. *The Bell System Technical Journal*, 57(6):1899–1904, July 1978.

[18] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, June 1996. doi:10.1007/s002360050048.

[19] Todd Palino. Running Kafka at scale, March 2015. URL https://engineering.linkedin.com/kafka/running-kafka-scale.

[20] Kartik Paramasivam. How we're improving and advancing Kafka at LinkedIn, September 2015. URL http://engineering.linkedin.com/apache-kafka/how-we%E2%80%99re-improving-and-advancing-kafka-linkedin.

[21] Rob Pike and Brian W Kernighan. Program design in the UNIX environment. *AT&T Bell Laboratories Technical Journal*, 63(8):1595–1605, October 1984. doi:10.1002/j.1538-7305.1984.tb00055.x.

[22] Jun Rao. How to choose the number of topics/partitions in a Kafka cluster?, March 2015. URL http://www.confluent.io/blog/how-to-choose-the-number-of-topicspartitions-in-a-kafka-cluster/.

[23] Azarias Reda, Yubin Park, Mitul Tiwari, Christian Posse, and Sam Shah. Metaphor: A system for related search recommendations. In *21st ACM International Conference on Information and Knowledge Management (CIKM)*, October 2012.

[24] Dennis M Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7), July 1974. doi:10.1145/361011.361061.

[25] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, February 1992. doi:10.1145/146941.146943.

[26] Sriram Sankar. Did you mean "Galene"?, June 2014. URL https://engineering.linkedin.com/search/did-you-mean-galene.

[27] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[28] Margo Seltzer. Beyond relational databases. *Communications of the ACM*, 51(7):52–58, July 2008. doi:10.1145/1364782.1364797.

[29] Michael Stonebraker and Uğur Çetintemel. "One size fits all": An idea whose time has come and gone. In *21st International Conference on Data Engineering (ICDE)*, April 2005.

[30] Roshan Sumbaly, Jay Kreps, and Sam Shah. The "Big Data" ecosystem at LinkedIn. In *ACM International Conference on Management of Data (SIGMOD)*, July 2013.

[31] Guozhang Wang. KIP-28 — add a processor client, July 2015. URL https://cwiki.apache.org/confluence/display/KAFKA/KIP-28+-+Add+a+processor+client.

[32] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, et al. Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655, August 2015. doi:10.14778/2824032.2824063.

[33] Lili Wu, Sam Shah, Sean Choi, Mitul Tiwari, and Christian Posse. The browsemaps: Collaborative filtering at LinkedIn. In *6th Workshop on Recommender Systems and the Social Web (RSWeb)*, October 2014.

[34] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2012.