Samza: Stateful Scalable Stream Processing at LinkedIn

Shadi A. Noghabi*, Kartik Paramasivam†, Yi Pan†, Navina Ramesh†, Jon Bringhurst†,
Indranil Gupta*, and Roy H. Campbell*
*University of Illinois at Urbana-Champaign, †LinkedIn Corp
{abdolla2, indy, rhc}@illinois.edu, {kparamasivam, yipan, nramesh, jon}@linkedin.com

ABSTRACT

Distributed stream processing systems need to support *stateful processing*, *recover quickly from failures* to resume such processing, and *reprocess* an entire data stream quickly. We present Apache Samza, a distributed system for stateful and fault-tolerant stream processing. Samza utilizes a partitioned local state along with a low-overhead background changelog mechanism, allowing it to scale to massive state sizes (hundreds of TB) per application. Recovery from failures is sped up by re-scheduling based on Host Affinity. In addition to processing infinite streams of events, Samza supports processing a finite dataset as a stream, from either a streaming source (e.g., Kafka), a database snapshot (e.g., Databus), or a file system (e.g. HDFS), without having to change the application code (unlike the popular Lambda-based architectures which necessitate maintenance of separate code bases for batch and stream path processing).

Samza is currently in use at LinkedIn by hundreds of production applications with more than 10,000 containers. Samza is an open-source Apache project adopted by many top-tier companies (e.g., LinkedIn, Uber, Netflix, TripAdvisor, etc.). Our experiments show that Samza: a) handles state efficiently, improving latency and throughput by more than 100× compared to using a remote storage; b) provides recovery time independent of state size; c) scales performance linearly with number of containers; and d) supports reprocessing of the data stream quickly and with minimal interference on real-time traffic.

1. INTRODUCTION

Many modern applications require processing large amount of data in a real-time fashion. We expect our websites and mobile apps to be deeply interactive and show us content based on users' most recent activities. We expect social networks to show us current global and local hashtag trends within seconds, ad campaigns to orient ads based on current user activity, and data from IoT (Internet of Things) to be processed within minutes.

Processing these streams of data in a real-time fashion poses some unique challenges. First, at LinkedIn, as a global social network company, trillions of events are fed to our production messaging system (Apache Kafka) and change capture system (Databus) per day. To process this massive amount of data, we need to be able to use resources efficiently and at scale, and to handle failures gracefully. Second, it is common for applications to access and store additional *stateful* data while processing each received event. At LinkedIn, examples of state include (depending on the application): user profiles, email digests, aggregate counts, etc. State computations include aggregations/counts over a window, joining a stream with a database, etc. Thus, we need mechanisms to: i) handle such state efficiently while maintaining performance (high throughput and low latency), and ii) recover quickly after a failure in spite of large state [52].

Third, it is common to require a whole database or the full received stream to be *reprocessed* completely. Such reprocessing is triggered by reasons ranging from software bugs to changes in business logic. This is one of the primary reasons why many companies employ the Lambda architecture. In a Lambda architecture [39], a streaming framework is used to process real-time events, and in a parallel "fork", a batch framework (e.g., Hadoop/Spark [14, 24, 58]) is deployed to process the entire dataset (perhaps periodically). Results from the parallel pipelines are then merged. However, implementing and maintaining two separate frameworks is hard and error-prone. The logic in each fork evolves over time, and keeping them in sync involves duplicated and complex manual effort, often with different languages.

Today, there are many popular distributed stream processing systems including Storm, MillWheel, Heron, Flink [7, 13, 35, 54], etc. These systems either do not support reliable state (Storm, Heron, S4 [35, 43, 54]), or they rely on remote storage (e.g., Millwheel, Trident, Dataflow [5, 7, 8]) to store state. Using external (remote) storage increases latency, consumes resources, and can overwhelm the remote storage. A few systems (Flink, Spark [13,18,59]) try to overcome this issue by using partitioned local stores, along with periodically checkpointing the full application state (snapshot) for fault tolerance. However, full-state checkpointing is known to be prohibitively expensive, and users in many domains disable it as a result [47]. Some systems like Borealis [6] run multiple copies of the same job, but this requires the luxury of extra available resources [18].

In this paper we present Samza, a distributed stream processing system that supports stateful processing, and adopts a *unified* (Lambda-less) design for processing both real-time as well as batch data using the same dataflow structure. Samza interacts with a change capture system (e.g., Databus) and a replayable messaging system (e.g., Apache Kafka, AWS Kinesis, Azure EventHub) [1,10,34,41]. Samza incorporates support for fast failure recovery particularly when stateful operators fail.

The Lambda-less approach is used by Spark and Flink [13, 59]. However, Flink still requires the programmer to access two APIs for streaming and batch processing. We

Proceedings of the VLDB Endowment, Vol. 10, No. 12

present experimental comparisons against Spark. Samza's unique features are:

- **Efficient Support for State:** Many applications need to store/access state along with their processing. For example, to compute the click-through rate of ads, the application has to keep the number of clicks and views for each ad. Samza splits a job into parallel tasks and offers high throughput and low latency by maintaining a local (in-memory or on-disk) state partitioned among tasks, as opposed to using a remote data store. If the task's memory is insufficient to store all its state, Samza stores state on the disk. We couple this with caching mechanisms to provide similar latency to, and better failure recovery than, a memory-only approach. Finally Samza maintains a changelog capturing changes to the state, which can be replayed after a failure. We argue that having a changelog (saving the incremental changes in state) is far more efficient than full state checkpointing, especially when the state is non-trivial in size.
- **Fast Failure Recovery and Job Restart:** When a failure occurs or when the job needs to be explicitly stopped and resumed, Samza is able to restart multiple tasks in parallel. This keeps recovery time low, and makes it independent of the number of affected tasks. To reduce overhead of rebuilding state at a restarted task, Samza uses a mechanism called Host Affinity. This helps us reduce the restart time to a constant value, rather than linearly growing with the state size.
- **Reprocessing and Lambda-less Architecture:** It is very common to reprocess an entire stream or database. Common scenarios are rerunning using a different processing logic or after a bug discovery. Ideally, reprocessing should be done within one system (Lambda-less). Further, the reprocessing often needs to be done alongside processing of streaming data while not interfering with the stream job and without creating conflicting data. Samza provides a common stream-based API that allows the same logic to be used for both stream processing and batch reprocessing (if data is treated as a finite stream). Our architecture reprocesses data without affecting the processing of real-time events, by: a) temporarily scaling the job, b) throttling reprocessing, and c) resolving conflicts and stale data from reprocessing.
- **Scalability:** To handle large data volumes and large numbers of input sources, the system has to scale horizontally. To achieve this goal, Samza: i) splits the input source(s) using consistent hashing into partitions, and ii) maps each partition to a single task. Tasks are identical and independent of each other, with a lightweight coordinator per job. This enables near-linear scaling with number of containers.

Samza has successfully been in production at LinkedIn for the last 4 years, running across multiple datacenters with 100s of TB total data. This deployment spans more than 200 applications on over 10,000 containers processing Trillions of events per day. Samza is open-source and over 15 companies, including Uber, Netflix and TripAdvisor, rely on it today [3].

Our experimental results show that Samza handles state efficiently (improving latency and throughput by more than 100× compared to using remote storage), provides parallel recovery with almost constant time (regardless of the size of the state), scales linearly with adding more containers, and supports reprocessing data with minimal effect on real-time traffic, while outperforming batch systems. We experimentally compare against both variants of our own system (some of which capture other existing systems), and against Spark and Hadoop in both production and test clusters.

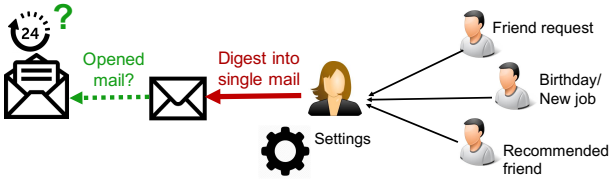


Figure 1: Email Digestion System (EDS).

2. MOTIVATION

2.1 Stateful Processing

Most event processing applications need to access state beyond the mere contents of the events. In this paper, we refer to state as any persistent data-structure defined by the application or used internally in the system. Such state may arise from cumulative or compact aggregates (computed from the stream), or static settings (e.g., user parameters), or stream joins. To illustrate, we describe *Email Digestion System (EDS)*, a production application running at LinkedIn using Samza. EDS controls email delivery to users by digesting all updates into a single email (Figure 1). EDS stores and accesses a large amount of state across multiple users. For each user, EDS accumulates and aggregates updates over a large window of time. To know the window size, EDS looks up the user digestion settings (e.g., every 4 hours) stored in a remote database. Finally, to find the effectiveness of the digested email, it computes if the email was opened in the last couple days, by joining two streams of sent emails and opened emails over a multi-day window.

State is typically categorized into two types:

Read-Only state: Applications look up "adjunct" read-only data, perhaps for each event, to get the necessary information to process it. Examples of such static state include user digestion settings in EDS or the user profile on each ad view event (accessed to find the user's field of expertise).

Read-Write state: Some state is maintained and updated as the stream events continue to be processed. Examples of this type of state include: state required for joins of streams/tables over a windows, aggregations, buffers, and machine learning models. Some applications of this state include rates/counter over a window of time (used for monitoring ads or detecting Denial of Service attacks) and guaranteeing exactly-once semantics by storing all processed message ids to verify uniqueness of incoming message ids.

2.2 Data Reprocessing

As described earlier, it is common to reprocess a stream or database, either in part or entirety. For example, at LinkedIn, we use a critical production job to standardize user profile information in order to offer relevant recommendations and advertisements. This job uses a machine learning model (derived offline) to standardize incoming profile updates in real-time. However, the model continually gets updated (even multiple times per week). Upon each update, all existing user profiles (> 450 millions) have to be reprocessed while still processing incoming updates in real-time and without creating conflicts.

In other scenarios, only a few hours worth of data has to be reprocessed (instead of a whole database). For example, during an application upgrade, a software bug may come up. With proper monitoring, the bug will most likely be detected within minutes or hours. The need after that is to revert the application (or fix the bug), rewind the input stream, and reprocess the data since the upgrade.

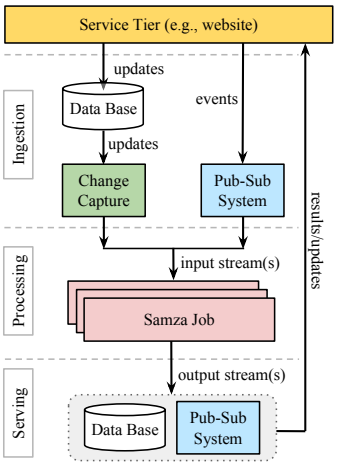


Figure 2: Stream processing pipeline at LinkedIn.

2.3 Application Summary

We summarize 9 major and diverse stream applications built using Samza that are currently running in LinkedIn's production environments across multiple datacenters, in Table 1. The location of the application is determined by the data source, varying from a single cluster to all clusters.

These applications exhibit a wide diversity along several angles: 1) *Scale*: throughput (input messages processed per second) and the number of containers, tasks, and inputs; 2) *State handled*: size and type of state; and 3) *Lifetime*: how long the job has been running.

Scale: The scale of applications varies widely based on the computational need of the application, from 70 containers to more than 500 containers. The higher scale is either due to higher throughput requirements (e.g., Inception) or computation load per event (e.g., EDS and Standardization). Samza supports various input source types (Kafka, Databus, Kinesis, etc.) as well as many input streams. Our applications range from 2 inputs to roughly 900 input streams with $> 27,000$ total partitions. For example, Inception processes 880 input streams (capturing exceptions) from multiple applications.

The number of tasks per container also varies significantly from 1 task per container (Inception) to ≈65 tasks per container (Call graph), with an average value of 10 tasks. A higher task per container ratio provides more flexibility when scaling out/in which is a positive factor for stateful jobs.

State: Applications range widely from stateless jobs (e.g., filtering done by Inception) to ones using a variety of different stores, ranging from fast in-memory stores to on-disk local stores with higher capacity (100s of TB vs. a few TBs) and remote stores with faster failure recovery. The type/size of the store is determined based on application requirements on performance, capacity, and failure recovery.

Lifetime: At LinkedIn, the number of production applications built using Samza has been growing rapidly, with a tenfold growth in the past 2 years (from 20 to 200). While we have focused on only the most mature applications here, newer applications continue to emerge.

3. SYSTEM OVERVIEW

In this section we present our end to end processing pipeline, Samza's high-level architecture, and how jobs are handled.

3.1 Processing Pipeline

Our stream processing pipeline works as a feedback loop (Figure 2). It receives events and updates from the service tier, processes them, and updates the service tier in return. This is a common pattern adopted by many companies.

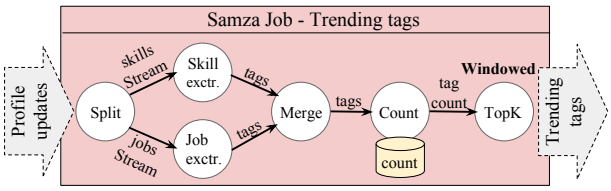


Figure 3: Example Samza job to find trending tags.

The service tier (top of Figure 2, e.g., the website and mobile app, where clients interact) generates two main types of data that need to be processed. First, more than trillions of *events* are generated per day. Use cases vary widely from capturing interactions with the service tier (e.g., viewed ads, and shared articles) to background monitoring and logging (e.g., site latency, exceptions and call tracing). Additionally, an often-overlooked source of data are *updates* occurring on databases (both SQL and NoSQL). Our databases have a transaction log capturing the stream of updates.

At the first phase in Figure 2, called the *Ingestion* layer, these stream of events and updates are ingested into fault-tolerant and replayable messaging systems. We use Apache Kafka [34], a large-scale publish-subscribe system (widely adopted by > 80 other companies [15]), and Databus [1], a database change capture system, as our messaging system for events and updates, respectively. Both these systems have the ability to replay streams to multiple subscribers (or applications), from any offset per subscriber. Databus also supports streaming a whole database from a snapshot.

In the next phase, called the *Processing* layer, these multiple streams of events and updates are fed to one or many Samza jobs. Samza acts as our core processing framework, processing input and generating output in real-time. Finally, in the *Serving* layer, results of the processing layer (e.g., connection recommendations and suggested feeds), are persisted to an external database or a pub-sub system (e.g., Kafka). Results are also returned to the service tier to update the services accordingly.

3.2 Processing Layer Design

A Samza job is an intact stage of computation: one or many input streams are fed to the job; various processing–from simple operations (e.g., filter, join, and aggregation) to complex machine learning algorithms–are performed on the input; and one or many new output streams are generated.

3.2.1 Logical Representation

Samza represents jobs as a directed graph of *operators* (vertices) connected by *streams* of data (edges). Figure 3 shows an example Samza job consuming a stream of user profile updates, splitting the stream into skill and job updates, extracting tags, and computing the top k trending tags (we use this as a running example in our discussion).

A stream is an infinite sequence of messages, each in the form of a $(key, value)$ pair, flowing through the system. A stream is internally divided into multiple partitions, based on a given entry. There are three types of streams: 1) input streams that enter the job, without a source operator (e.g., Profile updates); 2) output streams that exit the job, without a destination operator (e.g., Trending tags); and 3) intermediate streams that connect and carry messages between two operators (e.g., skills, jobs, tags and counts).

An operator is a transformation of one or many streams to another stream(s). Based on the number of input and output streams, Samza supports three types of operators: a)

Table 1: Applications running in LinkedIn's production across multiple datacenters. State size ranges from to 10s of GB to 100s of TBs (actual sizes not shown due to confidentiality). Max values shown in bold.

Name	Definition	Containers	Tasks	Inputs	Throughput msg/s	State type
EDS	Digesting updates into one email (aggregation, look-up, and join).	350	2500	14	40 K	on-disk
Call graph	Generating the graph of the route a service call traverses (aggregation).	150	**9500**	620	**1 Million**	in-mem
Inception	Extracting exception information from error logs (stateless filter).	300	300	**880**	700 K	stateless
Exception Tracing	Enriching exceptions with the source (machine) of the exception (join).	150	450	5	150 K	in-mem
Data Popularity	Calculating the top k most relevant categories of data items (join and machine learning).	70	420	9	3.5 K	on-disk
Data Enriching	Enriching the stream of data items with more detailed information (join).	350	700	2	100 K	on-disk
Site Speed	Computing site speed metrics (such as average and percentiles) from the stream of monitoring events over a 5-minute window (aggregation).	350	600	2	60 K	in-mem
A/B testing	Measuring the impact of a new feature. This application first categorizes input data (by their tag) into new and old versions and then computes various metrics for each category (split and aggregate).	450	900	2	100 K	in-mem
Standardization (>15 jobs)	Standardizing profile updates using machine learning models. This application includes > 15 jobs, each processing a distinct features such as title, gender, and company (join, look-up, machine learning).	**550**	5500	3	60 K	in-mem remote on-disk

1:1 operators (e.g., Count), b) *m:1 operators* (e.g., Merge), and c) *1:m operators* (e.g., Split), as shown in Table 2.

System API: The API of Samza is based on Java 8 Stream package [46] because of its ease of programming and functional programming capabilities. Listing 1 demonstrates the sample code for the Trending Tags job (Figure 3).

Listing 1: Sample API – Trending Tags Job.

```
public void create(StreamGraph graph, Config conf) {
   //initialize the graph
   graph = StreamGraph.fromConfig(conf);
   MsgStream<> updates = graph.createInStream();
   OutputStream<> topTags = graph.createOutStream();

   //create and connect operators
   MsgStream skillTags = updates.filter(SkillFilter f_s)
                                .map(SkillTagExtractor e_s);
   MsgStream JobTags = updates.filter(JobFilter f_j)
                              .map(JobTagExtractor e_j);
   skillTags.merge(jobTags).map(MyCounter)
         .window(10, TopKFinder).sendto(topTags);
         //10 sec window
}

class MyCounter implements Map<In, Out>{
//state definition
 Store<String, int> counts = new Store();
 public Out apply (In msg){
    int cur = counts.get(msg.id) + 1;
    counts.put(msg.id, cur);
    return new Out(msg.id, cur)
 }
}
```

The basic 1:1 operators are: a) `map`: applying a user-defined function on each message (e.g., `SkillTagExtractor` extracting tags using a machine learning model or `MyCounter` updating a local store); b) `filter`: comparing each message against a filter condition (e.g., `SkillFilter`); c) `window`: partitioning a stream into windows and applying a user-defined function on the window (e.g., `TopKFinder` over 10 *s* windows); and d) `partition`: repartitioning and shuffling a stream on a different key. The main m:1 operators are: e) `join`: joining two streams on a user-defined condition, and f) `merge`: merging two streams into one (e.g., merging `skillTags` and `jobTags`). Finally, the 1:m operators are defined by feeding the same stream into different operators (e.g., feeding `update` stream into two different filters).

The combination of diverse operator types and support for arbitrary user-defined functions enables handling a wide range of applications. For example, to perform aggregation (1:1 operator), depending on whether to be done over an entire stream or a window of data, a single aggregation logic (e.g., count) can be used in a `map` or `window` operator.

Table 2: Operators supported in Samza.

Type	Options	Definition
1:1	`map`	applying a defined function on each message.
	`filter`	filtering messages based on a function.
	`window`	splitting a stream into windows and aggregating elements in the window.
	`partition`	repartitioning a stream on a different key.
m:1	`join`	joining ≥ 2 streams into one stream based on a given function
	`merge`	merging ≥ 2 two streams into one stream.
1:m	user-defined	user-defined split or replication of a stream into ≥ 2 streams. This is achieved by allowing multiple operators consume the same stream.

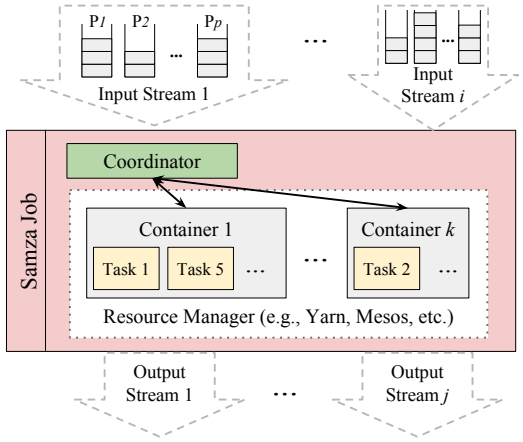


Figure 4: The internal architecture of a job.

3.2.2 *Physical Deployment*

Internally, as depicted in Figure 4, a job is divided into multiple parallel, independent, and identical tasks, and an input stream is divided into partitions (e.g., $\{P_1, ..., P_p\}$). Each task executes the identical logic, but on its own input partition (a data parallelism approach). Each task runs the entire graph of operators. For each incoming message, the task flows the message through the graph (executing operators on the message), until an operator with no output or the final output stream is reached.

Most intermediate stream edges stay local to the task, i.e., they do not cross the task boundary. This keeps most com-

munications local and minimizes network I/O. The only exception is the `partition` operator, where messages are redistributed across all tasks based on the partitioning logic. For non-local streams and the job's input and output streams, Samza utilizes a fault-tolerant (no message loss) and replayable (with large buffering capabilities) communication mechanism. At LinkedIn, we mainly use Kafka, although other communications mechanism supporting partitioning, e.g., Kinesis or Azure EventHub [10,41] can be used instead.

By employing replayable communication with large buffering capabilities, Samza can temporarily overcome congestion. Lagging messages are buffered without impacting upstream jobs, and replayed with the pace of the slow job. This is particularly important at non-local streams with high potential of creating congestion. This gives enough time for a temporary spike to pass, or to scale-out a slow job.

We leverage the partitioning already performed by the input streams in order to split jobs into tasks. The number of partitions of the input streams (configured by the application developer) indicates the number of tasks. For a single stream, each partition is mapped to a single task. However, partitions of different streams (e.g., partition 1 stream A and partition 1 stream B) can be mapped to the same task (used for joining two streams). A higher number of tasks provides more parallelism and finer granularity when scaling. However, too many tasks can create excessive overhead.

Resource Allocation: Tasks are grouped together into containers using a round-robin, random, or user-defined strategy. The number of threads is configurable, ranging from one per container up to one per task[1]. The spectrum of choices defined by these extremes also defines a trade-off between ease of programming (with no race conditions in a single-threaded model) and performance (with potentially higher throughput by exploiting more parallelism).

The application developer configures the number and capacity of containers, which defines the amount of resources assigned to a job[2]. Samza offloads container allocation and placement to a separate Resource Manager layer. This layer manages the available resources in cluster by handling: resource allocation, monitoring, failure detection, and failure recovery (by restarting or reallocation). This layered and modular design provides pluggability and flexibility. Currently, we use Apache YARN [55], one of the most popular resource managers, in our pipeline. Samza is also available as *standalone Samza*, an embeddable client library allowing applications to be hosted in any environment.

Coordinator: Each job has a lightweight *Coordinator* managing and tracking the job. The Coordinator maintains several pieces of metadata pertinent to the job including: i) job configuration (such as the number of containers and input sources); ii) placements (mapping of containers to machines, tasks to containers, and input partitions to tasks). When using YARN, the coordinator is part of YARN's Application Master, and when using standalone Samza, the coordinator uses Zookeeper to elect a singleton leader.

4. SYSTEM DESIGN

In this section we discuss Samza's goal (Section 1), existing ways to address it, and key design techniques in Samza.

[1]Within a tasks users can implement multi-threaded logic.

[2]Configuring the optimal number of containers is a challenging problem, especially in the presence of unpredictable workload changes [57]. As future work, we are working on dynamically and adaptively scaling the number of containers (based on the job's load and requirements).

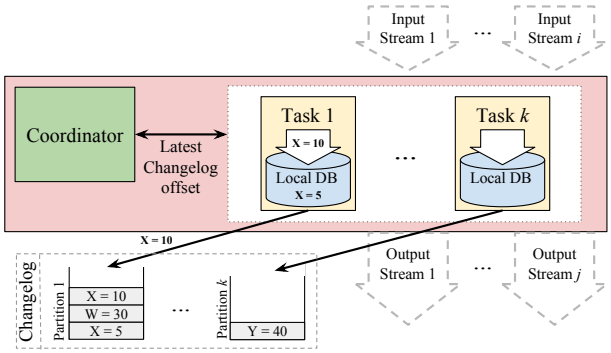


Figure 5: Layout of local state in Samza, and how fault-tolerance is provided.

4.1 Efficient Access to State

Several applications (Section 2.1) access/store large amounts of state along with processing incoming events. Some streaming engines have tackled this problem by using a reliable external *remote store* [5,7], e.g., MillWheel persists data in Bigtable [20]. This remote store has to independently handle fault-tolerance (by replicating data) while still providing some notion of consistency (the weakest requirement is usually read-your-writes consistency per task).

While storing state in an external file system outsources the responsibility of fault-tolerance, this approach is not efficient. It consumes network and CPU resources, increases average and tail latency, and limits throughput. It may also overwhelm the remote store (e.g., in presence of spikes), negatively impacting other applications using the shared store. When a single input message generates multiple remote requests, this is further amplified. For example, Millwheel and Trident provide exactly-once semantics by storing processed message keys (one write per message) along with verifying that incoming messages are unique (one read per message).

Another approach is to keep data local and, for fault-tolerance, use periodic *checkpointing*, i.e., a snapshot of the entire state is periodically stored in a persistent storage [13, 18, 59]. However, full state checkpointing in general slows down the application. It is particularly expensive when state is large, such as 100s of TB (Section 5); users tend to disable full state checkpointing for even smaller state sizes [47].

4.1.1 State in Samza

Samza moves the state from a remote store to instead being local to tasks – the task's *local store* (memory and disk) is used to store that task's state. This is particularly feasible in Samza with independent tasks (Figure 5).

Samza supports both in-memory and on-disk stores as options to trade off performance, failure recovery, and capacity. The in-memory approach is the fastest, especially for applications with random access (poor data locality). The on-disk store can handle state that is orders of magnitude larger while reducing failure recovery time. For our on-disk store we use RocksDB, a high-performance and low-latency single machine storage, widely used [4]. Other embeddable stores, e.g., LevelDB and LMDB, can be used as well.

Samza further improves on-disk stores by leveraging memory as a 3-layer cache. At the deepest layer, each RocksDB instance caches the most popular items using a least recently used (LRU) mechanism. To mitigate the deserialization cost of RocksDB, Samza provides a caching layer of deserialized data in front of RocksDB. Finally, we rely on OS caches to keep the frequently accessed pages around (similar to [44]). Our experiments show that for applications with good-locality workloads, caching mechanisms ensure on-disk

stores perform close to in-memory stores. For random access workloads, on-disk still achieves acceptable performance.

In most cases, state is partitioned across tasks using the same partitioning function and key as used for the input stream. Hash or range partitioning can be used. For instance, in a word-count job, a task is assigned to process words in a specified range (e.g., words starting with $[a-g]$) and stores state for the same range (e.g., counts of words starting with $[a-g]$). Joins, aggregations, metric computation (count/rates), are all supported in this manner.

For applications that absolutely need to use a remote store, Samza supports *asynchronous* processing of remote requests for efficiency and concurrency. When using asynchronous processing, Samza handles out of order event processing while ensuring at-least-once guarantees (even in the event of failures). This may be needed if the input partitioning is different from the state partitioning. In a job enriching place-of-birth updates with country information, the input is a stream of profile updates (key = *userid*) while the store is keyed by country (key = *countryid*). In such cases, if the state is small (tens of GB), Samza can broadcast the state to all tasks and store it locally, otherwise, Samza uses a remote store along with caching (for performance). Another use case is when tasks need to share state, or state needs to be queried from outside the job, where a remote store satisfying the consistency requirements of the job is used.

4.1.2 Fault-Tolerance

Using local state, requires solving a main challenge that arises out of it: *how to provide efficient fault-tolerance?* Samza equips each task with a changelog that captures all updates done on the local store (Figure 5). A key feature of the changelog is that it is only capturing incremental changes rather than the entire state. The changelog is an append-only log maintained outside of the job architecture. Samza stores the changelog in Kafka, enabling fast and easy replays in case of a failure, although, any other durable, replayable and ordered messaging system can be used.

For efficiency, the changelog is kept out of the hot path of computation. Updates are batched and periodically sent to Kafka in the background using the spare network bandwidth. After successfully writing a batch of updates, the *latest offset*–indicating the latest successfully processed input message–is persisted in the Coordinator (Figure 5). After a failure, state is rebuilt by replaying the changelog, then, all messages after the latest offset are reprocessed.

Moreover, to reduce changelog overheads and prevent an indefinitely growing changelog, Samza utilizes Kafka's compaction features. Compaction retains the latest value for each key by removing duplicate updates. Compaction is performed in the background and outside the hot path. Compaction is used in two cases: 1) compacting the batch of updates sent to the changelog (reducing the network overhead); 2) compacting the changelog itself (reducing storage overhead). Right after compaction, the changelog is no larger than a snapshot of the task's most critical state.

Samza guarantees at-least-once processing, preferring performance over consistency. In practice, we observe that at-least-once is sufficient for our applications requirements. For a few cases requiring exactly-once, it is implemented by the application (with low overhead) by leveraging local state.

The changelog approach in Samza provides a *read-your-writes* consistency level on a per task basis. Without failures, data is stored locally on single replica, straightforwardly providing read-your-writes consistency. In presence of a failure, processing and state are rolled back to the point of time where consistency is conserved, i.e., the latest persisted offset, wherein all updates from processed messages up to the latest offset are reliably reflected in the state.

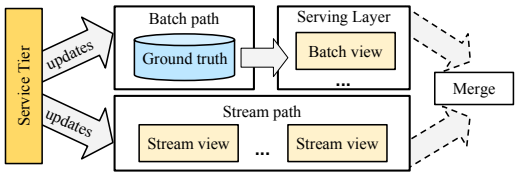


Figure 6: Lambda architecture.

The changelog in Samza adds less than 5% performance overhead. An append-only log has been measured to be far more efficient (2 million op/s with 3 machines in Kafka [25]) compared to accessing a remote store (at most 100K op/s with 3 machines [22, 49]).

4.1.3 Fast state recovery

After a failure, a job or a portion of it needs to be restarted. Failures may include node, container, or disk failures. Restart may also be warranted upon preventive maintenance (either stop-the-world or one container at a time), and configuration updates (due to misconfiguration or workload changes).

Replaying the changelog (even compacted) can still introduce excessive overhead and long pauses, e.g., with 100s of TBs of state. This will be especially pronounced when the changelog is accessed remotely. To mitigate this, Samza uses a fast state recovery mechanism called *Host Affinity (HAff)*. The key idea in HAff is to leverage the state already stored on disk (in RocksDB) by preferring to place a restarting task on the same physical machine where it was running prior to the failure (stored in the Coordinator). This is a best-effort mechanism, and will continually try to optimize placement, even in presence of repeated failures. However, HAff is not effective in case of permanent machine failures, where replaying the changelog is used instead.

To make HAff feasible, Samza stores state in a known directory (in the native file system) outside of the container namespace. This allows state to live independent of the application lifecycle. A garbage collection agent runs in the background, removing state of permanently deleted applications. Since the underlying system cannot distinguish between stopped and deleted applications, we rely on the application developer to manually mark applications as deleted.

In production, we found that HAff is effective in over 85% of restart cases. By using HAff in our large stateful applications ($\approx$ 100 of TBs of state), we were able to reduce recovery time by 60$\times$ (from 30 minutes to 30 seconds).

4.2 Lambda-less

Inevitable software bugs and changes along with inaccuracies (late or out-of-order arrivals) can require parts (or even a whole) stream to be reprocessed. To mitigate this issue, many companies [17] utilize a Lambda architecture, wherein data is dispatched in a parallel "fork" to both an online stream and offline batch path (e.g., Hadoop or Spark), as shown in Figure 6). The stream path processes incoming data in real-time (latency is first-class) while the batch path acts as source-of-truth, periodically generating batch views of accurate results (accuracy is first-class). Final results are computed by merging stream and refined batch views [39]. To reprocess data it is sent via the batch path.

However, the Lambda architecture comes at a high management cost, requiring duplicate development of stream and batch logic and code for the same application, and keeping these logics in sync as the application evolves over time. The Lambda approach also consumes double resources (for

stream and batch processing). In the batch path inaccuracies could still occur–there can be late arrivals at the beginning, and missing data at the end of the batch interval.

Samza instead adopts a unified model supporting both stream and batch. The main challenges are: 1) to process *late events*, and 2) to *reprocess* a stream or database without impacting incoming messages or pressuring the database/service. 3) to support an easy-to-use API (Section 3.2.1) readily available in batch systems [45, 53, 59].

Unified Model: Similar to [8,12], Samza treats batch data as a finite stream of data, with a special token indicating the end of the stream. Application logic is developed and maintained in one place using a unified API. A single application can switch between real-time traffic, batch data from HDFS (integrated with Samza), or a database snapshot.

Processing Late Events: Samza employs a *reactive approach*, i.e., processing and fixing previous results when late or out-of-order results arrive (this bears similarities to Millwheel [7]). To avoid reprocessing the entire stream, the input is split into windows. Upon processing a late message, the impacted windows are found, rolled back, and recomputed [8]. State management is a key element in late event handling. Generally, the whole window of messages should be stored (e.g., a join operation). For some operations, storage can be optimized where a compact final "result" is available (e.g., for a counter or aggregations).

Currently, the application is in charge of implementing the late arrival logic [3]. However, the windowing functionality along with the efficient state handling make Samza an perfect fit for this I/O intensive feature.

Reprocessing: To reprocess an entire stream or database (Section 2.2), Samza leverages: a) Kafka's replaying capability to reprocess a stream, and b) Databus' *bootstrapping* capability to reprocess a database. During bootstrapping, Databus generates a stream from a database snapshot (without impacting the database service) followed by the stream of new updates after the snapshot.

To perform reprocessing, Samza simply switches between different inputs: real-time traffic, replayed stream, or bootstrap stream, in single intact application. Reprocessing can be done in two modes: 1) *blocking* where real-time computation blocks until all reprocessing is complete, or 2) *non-blocking* where reprocessing is done in parallel with real-time processing. Typically, blocking reprocessing is used with small datasets, e.g., rolling-back latest upgrade due to a bug, while non-blocking processing is used with massive datasets, e.g., business logic change requiring reprocessing of whole database. In non-blocking reprocessing, Samza minimizes the impact on the real-time processing via: i) throttling reprocessing, ii) temporary job scale out.

Late events may create conflicts. A merge job is used to resolve conflicts (between the reprocessing and real-time stream) and prioritize the real-time results. This is developer specified logic. For instance, in the Standardization job, the user may change the profile, p_{new}, while the reprocessing will also process the user's old profile, p_{old}. If reprocessing of p_{old} occurs after processing p_{new}, it can override the results of the new profile. Thus, a merge job is needed to merge both updates and prioritize the results of p_{new}.

4.3 Scalable Design

Samza provides scalability via a decentralized design, maximizing independence among its components.

[3] As future work, we are adding late event handling as a built-in support in Samza.

Table 3: Main parameters of data generation in each approach, and the range of values studied.

Approach	Parameter	Definition	Range
Checkpoint	*interval*	time between two consecutive checkpoints.	10 min - 90 min
	state_size	total size of state in Bytes	100 GB - 100 TB
Changelog	*change_rate*	rate of entry changes in the state (msg/s).	10 K - 10 M
	entry_size	size of each entry of the state in Bytes	10 B - 1 KB

1. Scaling resources: As discussed in Section 3.2, a job is split into independent and identical tasks (with input/state partitioning). Then, independently tasks are allocated on containers. This decoupling allows tasks to be flexibly scheduled and migrated if necessary.
2. Scaling state: Samza scales to a massive amount of state, by leveraging independent partitioned local stores. Also, state recovery is done in parallel across tasks and is not impacted by the number of failed containers.
3. Scaling input sources: Samza treats each input stream autonomously from other inputs. This enables scaling to many inputs, e.g., the Inception application (Table 1) processes exceptions from more than 850 different streams sources. Due to its modular design, Samza works with a variety of systems including: Databus, DynamoDB Streams, Kinesis, ZeroMQ and Mongo DB [1, 2, 9, 10, 31, 34], and this set is continuously growing.
4. Scaling number of jobs: Samza utilizes a decentralized design with no system-wide master. Instead, each job has a lightweight Coordinator managing it. Also, jobs maintain their independence from each other, and each job is placed on its own set of containers. This enables Samza to scale to large numbers of jobs. We have seen a 10× growth in the number of applications over the past 2 years.

5. CHECKPOINTING VS. CHANGELOG

To provide fault-tolerance, Samza uses a changelog capturing *changes* to the state in the background. Another popular approach is full state checkpointing, where periodically a snapshot of the *entire state* is taken and stored in an external storage [13, 18, 27, 59]. Checkpointing can be either synchronous (pause, checkpoint, and resume) or asynchronous (in the background)–a more performant but also more complex approach. In both cases, the overhead of checkpointing can be prohibitive especially for large state sizes.

In this section, we quantitatively compare full-state checkpointing vs. Samza's changelog approach, taking into account characteristics of real applications from production.

The average amount of additional data generated (Bytes/s) is the main source of overhead in both checkpointing and changelog. Table 3 summarizes the parameters that affect it. For checkpointing, data generation depends on checkpointing interval (*interval*) and size of each checkpoint (*state_size*). The *interval* trades off checkpointing overhead (less for larger intervals) and the amount of work needed to be redone in the case of a failure (more for larger intervals). On the other hand, changelog depends on the rate of changes (*change_rate*) and the size of each change (*entry_size*). Thus, the average rate of data generation for these approaches are:

$$Data_{checkpoint} = \frac{state\_size}{interval}$$
$$Data_{changelog} = change\_rate \times entry\_size$$

We define the break-even point, *bp*, as where $Data_{checkpoint}$ equals to $Data_{changelog}$. For any *change_rate* value below

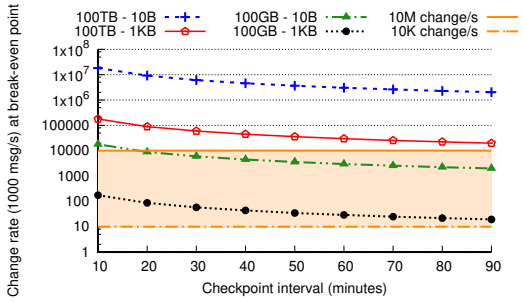


Figure 7: Comparison of checkpointing under various *state_size* (100 TB and 100 GB) and *interval* values with changelog under various *entry_size* (10 B and 1 KB) and *change_rate* values. Shaded region shows 10× typical values from applications (Table 1)

bp, changelog is the preferred approach, and for any value above, checkpointing. For various checkpointing configurations (*interval* and *state_size*) and *entry_size* values, we measure *change_rate* at break-even point. This is depicted as the lines in Figure 7. For example, for a *state_size* of 100 TB, an *interval* of 20 minutes, and *entry_size* of 10 B, *bp* is ≈ 10 Trillion changes/s. For any *change_rate* below 10 Trillion/s, changelog would be a better option.

Based on our production application configurations (Section 2.3) a *change_rate* of Trillions of changes/s is not realistic. As a pessimistic estimate of the *change_rate* (accounting for our application growth over the next few years), we use the throughput achieved in our production applications (Table 1) as a proxy for the *change_rate* and multiply it by 10. This range, is shown by the shaded area in Figure 7.

We observe that for large *state_size* values (100 TB), changelog is clearly a better choice (the shaded area is below the 100 TB lines). A small *state_size* with a large *entry_size* (100 GB - 1 KB, the lowest line in the plot) is also uncommon in production-scale applications, since the state should compose of only a few entries. For a small *state_size* and a small *entry_size* (100 GB–10 B, second lowest line in plot), at a *change_rate* of around 10 M change/s, changelog performs worse than checkpointing. To mitigate this issue, Samza utilizes batching along with a compaction mechanism (removing redundant keys) to reduce the effective *change_rate*. By batching data for a couple of seconds, even with a *change_rate* of 10 M change/s (given the total state has ≈ 10 Million entries in the 100GB–10B case), the effective *change_rate* is reduced significantly, keeping changelog efficient and the more preferable technique.

6. EVALUATION

Our evaluation addresses the following questions:

1. How effective is local state, versus alternative options?
2. What is the effect of failures, and how fast is recovery? How much does Host Affinity help in failure recovery?
3. How fast is reprocessing, especially compared to existing systems?
4. How does Samza scale?

In doing so, we compare Samza with existing systems including Spark and Hadoop, as well as against other alternative Samza-like designs.

6.1 Experimental Setup

We evaluated the system using both production jobs and microbenchmarks, subjecting the system to much higher stress than production workloads. Our experiments were performed on both small (6 nodes) and large (500 nodes) production clusters at LinkedIn.

Microbenchmarks were performed on a test YARN and Kafka cluster. We used a 6 node YARN cluster, with 4 Resource Managers (RMs) and 2 Node Managers (NMs). Each NM was a high-end machine with 64GB RAM, 24 core CPUs, a 1.6 TB SSD, 2 1TB HDDs, and a full-duplex 1 Gbps Ethernet network. We also used an 8 node Kafka cluster of similar machines. We tested the system using two applications: a *ReadWrite* and *ReadOnly* job.

The ReadWrite job contains a map of ids to counters. For each input message, an embedded *id* is extracted, current count for *id* is read, the counter is incremented, and then written back. This job mimics the trend in real-world aggregation and metrics collecting jobs, e.g., EDS, Call Graph, Site Speed, and A/B Testing in Table 1.

The ReadOnly job consists of a join between a database and an input stream. For each message, an embedded *id* is extracted, value *val* for id is read from a database, *val* is joined with (a fraction of) the input message, and outputted as a new message. This follows the pattern used in many real-world enriching jobs, e.g., Data Enriching (enriching a stream of data with additional details) and Exception Tracing (enriching exceptions with source information).

We use a single input stream with infinite tuples (*id*, padding). *id* is a randomly generated number in the range $[1, 10^k]$ and padding is a randomly generated string of size m. We use k and m as tuning knobs of the workload. k trades off state size for locality–a larger k creates more entries (larger state) while decreasing the chance of reading the same data twice. m is used to tune CPU/network usage. Since the serialization/deserialization overhead and header overhead per message is almost constant, m tunes the ratio of overhead to Bytes/s processed. We chose m such that the system is under stress (CPU and network utilization $\geq$ 60%). We found 100 and 130 Bytes padding to be the appropriate values for ReadWrite and ReadOnly, respectively.

Before submitting a job, we pre-populate the input stream, so that no time is spent on waiting for new data (inter-arrival between messages is 0). Additionally, in ReadOnly case, we pre-populate the store with random values for all keys.

6.2 Effectiveness of Local State

In order to evaluate our design of local state, we compare our choice against other alternative designs:

- *in-mem* and *on-disk*: A partitioned in-memory store (our homegrown key-value store) or on-disk store (RocksDB), *without* any fault-tolerance mechanism. Stateless systems, such as Storm and Heron, use these type of stores (typically in-mem). Additionally, without considering the checkpoint overhead which depends on the interval and state size (Section 5), systems using checkpointing, e.g., Flink and Spark [13,18,59], also fall here.
- *in-mem + Clog* and *on-disk + Clog*: Samza's in-mem or on-disk store along with changelog for fault-tolerance.
- *on-disk no cache*: On-disk with no in-memory caching. This mimics the behavior of applications with large state and poor data locality (high cache misses).
- *remote store*: An external remote storage, used in many systems including Millwheel, Trident, Dataflow [5,7,8].

Although our Samza implementation supports all these variants, the default is on-disk + Clog. This variant performs the best, has large state support (hundreds of TBs), and offers low cost failure recovery (close to stateless).

To evaluate state, we used ReadWrite and ReadOnly microbenchmarks. In each test we continuously added containers

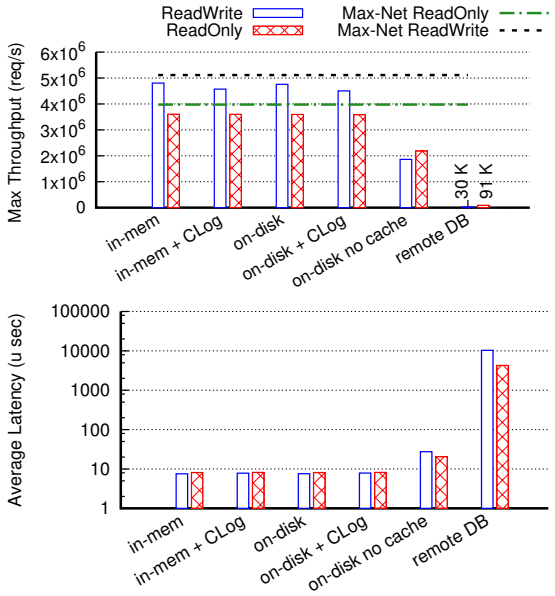


Figure 8: Comparision of storing state using an in-memory structure, local on-disk storage, or a remote database, with and without a changelog (CLog). Theses graph show throughput and latency in a read only and a 50-50 read write workload.

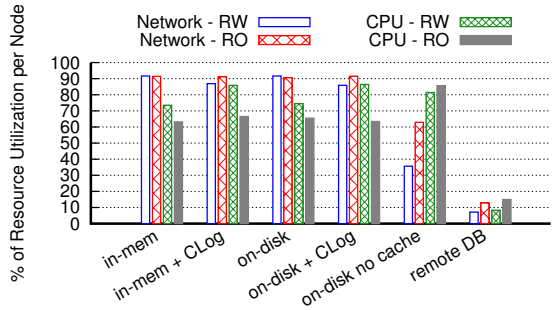


Figure 9: Utilization of network (in bound link) and CPU when using in-mem, on-disk and remote state, with and without a changelog (CLog).

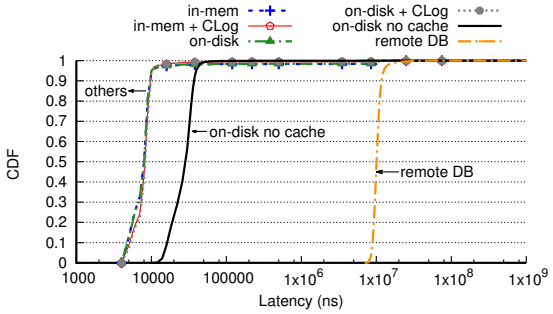


Figure 10: CDF of latency when using local and remote state, with and without a changelog (CLog).

until throughput is saturated. Figure 8 shows maximum achieved throughput and average latency for different stores. We computed the theoretical maximum throughput achievable by the network (Max-Net), i.e., network bandwidth divided by the message size. Since ReadOnly messages are larger than ReadWrite, both maximum and achieved network throughput are smaller.

6.2.1 *In-memory vs. On-disk*

As shown in Figure 8, in-mem and on-disk stores perform similarly, and both approach the network maximum (Max-Net). The in-mem and on-disk stores do not handle fault-tolerance. However, even when we add fault-tolerance using a changelog, the overhead is negligible.

To measure the effect of caches, we also plot numbers from disabling all internal caches, including the caching layer provided by Samza and Rocks DB (on-disk no cache). This reduced throughput by only 50-60%, indicating the caching is not solely responsible for our performance gains.

We conclude that on-disk state coupled with a caching strategy can achieve the same performance as using in-mem store, but it also achieves better fault-tolerance and supports larger state than in-mem (TBs vs tens of GBs).

6.2.2 *Local vs. Remote state*

We compared using local state (in-mem or on-disk) to remote state. As our remote state we used Espresso [48], a scalable key-value store widely used in LinkedIn's production (e.g., storing user profiles). We used an additional 5 node cluster (4 data nodes and a router) with nodes similar to the Kafka cluster. As shown in Figure 8, even with additional resources used for the remote store, latency increases by 3 orders of magnitude (a few μs to a few ms). This is due to traversing multiple hubs (router, data nodes, replication, and back to the user) which each takes hundreds of μs.

Throughput is impacted less than latency, and drops by two orders of magnitude (100-150 $\times$), since requests are issued in parallel. ReadOnly achieves 3$\times$ better throughput than ReadWrite because a) the former issues fewer requests per message (one vs. two), and b) reads have lower overhead vs. writes (no replication needed).

We expect this large difference between local and remote state to hold beyond Espresso. Other studies [22, 49] show that most popular stores, such as Cassandra, HBase, Voldemort, MySQL, Couchbase, and Redis [16, 23, 28, 36, 42, 50], can only reach tens of 1000s of requests/s using of 4 nodes. When using local state we perform millions of requests/s.

6.2.3 *Resource Utilization*

Figure 9 measures the resource utilization (CPU, disk, network) for each test. We elide disk utilization (being $<$ 5% for all except for no-cache case) and outbound network (following the same pattern as inbound link) due to space.

When using in-mem store or on-disk with caching (with or without changelog), we saturate the network (utilizing $\geq$ 85%). Note that our benchmarks are configured to stress-test the system using $\geq$60% of CPU resources, while in production this value is typically below 20%.

Adding changelog has a small impact ($\approx$ 15%) on CPU utilization (additional serialization and deserialization overhead), and less than 2% effect on network. Similarly, removing the internal caches (on-disk no cache) causes a spike in CPU usage, though it is processing fewer messages–this is because of RocksDB's serialization/deserialization overhead.

The remote DB has a low utilization ($<$ 20%) in all resources, since the job is mostly idle–waiting for a response from the database. The resources are used inefficiently as well. For example, using remote store, the amount data transfered over the network for processing a single message is 5-10$\times$ higher than local store.

6.2.4 *Latency Tail and Variance*

We define latency as the total time spent in processing a message (event), including time spent in fetching the message from the input source. Figure 10 shows the Cumulative

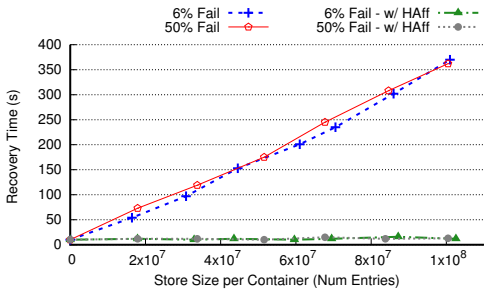


Figure 11: Failure recovery time using different store sizes with and without Host Affinity (HAff). The results are with a ReadWrite workload when 6% and 50% of the containers fail.

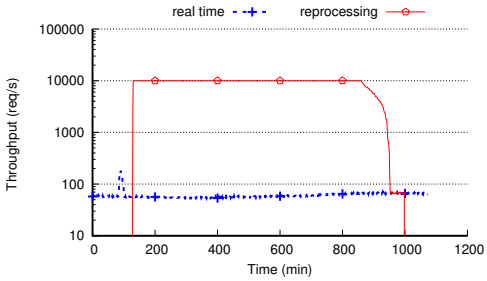


Figure 12: Throughput of Standardization job while performing a reprocessing (starting after 2 hours).

Distribution Function (CDF) of latencies in all cases. Since Samza fetches messages in batches (50 K messages in our test), a few messages incur very long latencies causing long tails in the CDF. However, for the rest, the variance is low and a majority of values are close to the median.

6.3 Failure Recovery

To measure failure recovery overhead when using local state, we randomly killed a percentage of containers (6% to 50%) in a stateful job. We measured the recovery time–the time spent between the first failure until all containers are up and running–both with Host Affinity disabled and enabled (w/ HAff). In Host Affinity, we used a success rate of 100%, i.e., ratio of containers placed on the same machine as before. Although this might seem extreme, it is not far from our production success rate (85-90%). In production, the main reason for misses are permanent node failures, and being a shared cluster with other jobs filling the free capacity.

In this experiment we used the ReadWrite workload, 16 8GB containers, on-disk + Clog store, and an input stream containing all keys in the range $[1 - 10^{12}]$ in order. For each input message processed a new entry was stored locally and added to the changelog.

As Figure 11 shows, without Host Affinity, recovery time increases proportionally with state size. With Host Affinity recovery time becomes near constant independent of the state size. In our production jobs, recovery time reduced from 30 minutes to less than 30 seconds using Host Affinity.

Furthermore, failure recovery time was nearly independent of the percentage of containers failing. This is because tasks are recovered in parallel.

6.4 Reprocessing

We analyzed the impact of reprocessing in our production jobs. We evaluated the Standardization job, our most frequently reprocessed job, over a 24 hour period. Standardization consumes profile updates and using a machine learning model, transforms the update to a standardized text. After 2 hours, we started reprocessing the entire database of user profiles (> 450 Million entries). Simultaneously, we scaled-out the job from 8 containers to 24.

Figure 12 shows the reprocessing throughput. Reprocessing peaks and remains at 10,000 messages per second (due to our throttling mechanism). After all the data is processed (≈16 hours), reprocessing throughput drops and starts catching up with the real-time data. At this point, we stop reprocessing and scale-in the job. Reprocessing time can be reduced, similar to a batch job, by simply allocating more resources. The combination of scale-out and throttling mechanisms ensure that reprocessing does not affect real-time processing performance.

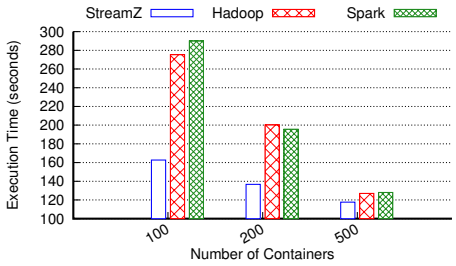


Figure 13: Comparison of Samza with other batch processing frameworks in reprocessing data.

6.4.1 *Batch Processing using Samza*

We compared Samza's reprocessing/batch solution with other mainstream batch processing solutions including Spark and Hadoop [14,58]. Spark offers high similarity of code for batch and stream processing, thus making it a near-Lambda-less architecture. Hadoop is a system that might be used modularly inside the Apache Beam architecture.

In this experiment we used Members Per Country (MPC), a real-world batch job running at LinkedIn, and reimplemented the job in Samza (with HDFS consumer). MPC reads a snapshot of all user profiles, groups them by country (Map), and then counts the members in each country (Reduce). We used 450 million profile records stored across 500 files (250 GB of data) in a production YARN cluster (≈ 500 nodes), and single core containers with 4GB RAM.

Figure 13 shows Samza has better throughput than Spark and Hadoop[4]. This is because it streams data to downstream operators as soon as it becomes available, while Hadoop and Spark (in batch mode) are limited by the barrier between Map and Reduce phases [21] [5].

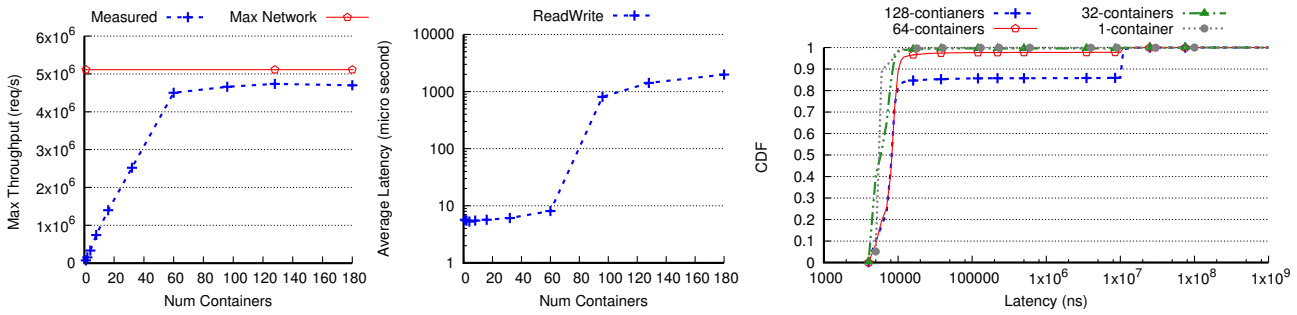
6.5 Scalability

Figure 14 shows maximum throughput and average latency in Samza as the number of containers increases (in the ReadWrite workload). Throughput increases linearly, saturating just beyond 60 containers. The saturation point is very close to the optimum throughput possible in the network. Latency stays low at first and increases thereafter. This knee of increase in latency coincides with the throughput saturation, and thus, can be used as an indicator of when to stop scaling. For maximizing throughput, there is low marginal utility in scaling beyond the saturation point.

Figure 14(c) shows the CDF of the latency. Even with twice more containers than needed, a majority of messages

[4]Latencies are higher in Hadoop due to the barrier and Spark due to micro-batching; these are not plotted.

[5]Samza is also able to exploit more parallelism than the other frameworks, better utilizing CPU cores.



(a) Throughput (b) Latency (c) Latency CDF

Figure 14: Throughput, average latency and CDF of latency in scalability study adding containers, under a 50-50 read write workload using local state. The saturation point of the system is 64 containers.

(> 80%) are processed within a few microseconds, and with small variance. The tail prolongs when containers are beyond the throughput saturation point, primarily because more time is spent waiting for the next events than processing them. We also observe that latencies are higher with more containers (e.g., 128 vs. 32). This is because the latency is calculated from message fetch time to processing completion. With more containers, more outlier messages need to be fetched remotely, and this drives up the average.

7. RELATED WORK

State management: State management varies significantly among stream processing solutions. Many industrial scale systems, such as Apache Storm, Heron and S4 [35, 43, 54] are built with no support for state. Trident and MillWheel [5, 7] manage state by using a combination of "hard state" persisted in an external store along with "soft state" stored in memory as a non-fault-tolerant cache. Thus, they either incur high overhead by relying on a remote storage or accept the chance of losing data.

There has been some work on partitioning state similar to the idea of local state [13, 59]. StreamCloud [30] discusses elastic and load-balanced state partitioning. However, partitioning is only supported for specific operators (join and aggregation) and it does not address fault-tolerance. S-Store [40] proposes transactional state management for stream data that is a potential add-on to Samza.

Fault-tolerance in local state: Upstream backup recovery [6, 54] successfully restore processing, but not the state. One approach to add fault-tolerance is by using replication [6] (as studied in [32]). However, this requires the luxury of extra available resources [18], and approaches like Sweeping checkpointing [29] do not ameliorate this problem.

Another popular approach is continuous full-state checkpointing of state along with input replay in presence of failures. Fernandez et al. [18] discuss scale-out state management for all operators by partitioning state and using checkpoint. Many others [13,33,37,56,59] also employ checkpointing mechanism to ensure fault-tolerance. The SDG approach [27] enables asynchronous checkpointing by locking the state, keeping a dirty buffer for incoming changes during checkpointing, and then applying the dirty buffer on the state. [33] generates a global snapshot by using a blocking variation of Chandy-Lamport snapshot [19] where it blocks on on-the-fly messages before generating the snapshot. Instead of blocking, IBM System S [56] persists checkpoints in the external DBMS (which is slow), and [18] captures pending asynchronous operations as part of the state (which is complex). The excessive overhead of full-state checkpointing, especially with large state sizes, make these approaches prohibitive. Sebepou et al. [51] partition state into smaller chunks, with incremental updates. However, it was only evaluated for aggregation operators, and it is unclear how effective it will be on user-defined logic.

Unified stream and batch: MapReduce Online [21] has explored processing batch jobs in an online barrier-free manner, but they do not fully support stream-processing. Liquid [26] also has a unified integration stack, but still maintains two separate subsystems.

Apache Beam, Dataflow, and Flink [8, 12, 13] have moved toward integrating batch into stream as a unified environment. Dataflow and Borealis [6, 8] have investigated how to handle inaccuracies caused by out-of-order messages occurring in stream frameworks. However, Dataflow relies on a remote store (not handling large state efficiently), and Flink is not fully unified (separate APIs for batch and stream). Samza can be used modularly inside Beam which acts as a wrapper API. Besides, Dataflow and Beam incur extra overhead by not leveraging the inherent partitioning capabilities of systems like Kafka, Kinesis, or EventHub. Spark Streaming [59] also has a unified environment, however, it processes data in micro-batches incurring higher processing latency. Also, Flink and Spark Steaming are not available as a standalone version and lose the deployment flexibility.

Scalability: Scaling to large state necessitates going beyond relying on memory, e.g., by using disk spilling [38]. This is orthogonal to our approach and could be used as an extra optimization in Samza. For better scalability, operators need to work with maximum independence. Thus, many systems have opted to use reliable, replayable communication mechanisms to handle data buffering between operators, e.g., Streamscope and Heron [35, 37]. IBM System S [11, 56] utilizes fault-tolerant replayable communication and distributes operations into a set of independent component-local operators. These systems deploy a similar approach to the scalable design in Samza. However, none of them target large state or reprocessing.

8. ACKNOWLEDGMENTS

We wish to thank the following people for their invaluable input towards this paper: Hassan Eslami, Wei Song, Xinyu Liu, Jagadish Venkatraman, and Jacob Maes. We would like to thank all contributors to Apache Samza with special mention to Chris Riccomini. Their ideas and hard work have been critical to the success of Samza. In addition, we would like to thank Swee Lim and Igor Perisic from LinkedIn for their support.

The UIUC part of this work was supported in part by the following grants: NSF CNS 1319527, and AFOSR/AFRL FA8750-11-2-0084.

9. CONCLUSION

This paper described Samza, a distributed system that supports stateful processing of real-time streams, along with reprocessing of entire data streams. Samza recovers quickly from failures, with recovery time independent of application scale (number of containers). It can support very large scales of state in spite of limited memory, by combining local on-disk storage, an efficient changelog, and caching.

Our experiments showed Samza has higher throughput than existing systems like Spark and Hadoop. Samza runs both batch and stream processing in a unified way while minimizing interference between them. We also described several applications that rely on Samza.

Samza's approach opens up many interesting future directions including: dynamic rebalancing and task re-splitting (changing number of tasks), automatic configuring and scaling of resources (containers), investigating stragglers (not a major issue so far), and handling hot vs. cold partitions.

10. REFERENCES

[1] Databus. https://github.com/linkedin/databus.

[2] MongoDB. https://www.mongodb.com.

[3] Powered by samza. https://cwiki.apache.org/confluence/display/SAMZA/Powered+By.

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

[5] Trident. http://storm.apache.org/Trident-tutorial.html.

[6] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, et al. The design of the Borealis stream processing engine. In *Proc. CIDR*, pages 277–289, 2005.

[7] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, et al. Millwheel: fault-tolerant stream processing at internet scale. *Proc. VLDB*, pages 1033–1044, 2013.

[8] T. Akidau, R. Bradshaw, C. Chambers, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB*, 8(12):1792–1803, 2015.

[9] Amazon. DynamoDB streams. http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html.

[10] Amazon. Kinesis. https://aws.amazon.com/kinesis/.

[11] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, et al. SPC: a distributed, scalable platform for data mining. In *Proc. IWMSSP*, pages 27–37. ACM, 2006.

[12] Apache. Beam. http://beam.incubator.apache.org.

[13] Apache. Flink. https://flink.apache.org.

[14] Apache. Hadoop. http://hadoop.apache.org/.

[15] Apache. Kafaka - powered by. https://cwiki.apache.org/confluence/display/KAFKA/Powered+By.

[16] A. Auradkar, C. Botev, S. Das, et al. Data infrastructure at LinkedIn. In *Proc. ICDE*, pages 1370–1381, 2012.

[17] A. AWS. Lambda. https://aws.amazon.com/lambda/.

[18] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proc. SIGMOD*, pages 725–736. ACM, 2013.

[19] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *TOCS*, pages 63–75, 1985.

[20] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, et al. Bigtable: A distributed storage system for structured data. *TOCS*, 26(2):4:1–4:26, 2008.

[21] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, et al. Mapreduce online. In *Proc. NSDI*, pages 20–25, 2010.

[22] E. P. Corporation. Benchmarking top nosql databases. *Technical Report*, page 19, 2015.

[23] Couchbase. Couchbase. http://www.couchbase.com.

[24] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[25] L. Engineering. Benchmarking apache kafka: 2 million writes per second (on three cheap machines). https://engineering.linkedin.com/kafka.

[26] R. Fernandez, P. Pietzuch, et al. Liquid: Unifying nearline and offline big data integration. In *Proc. CIDR*, page 8.

[27] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *Proc. ATC*, pages 49–60. USENIX, 2014.

[28] T. A. S. Foundation. Apache HBase. http://hbase.apache.org/.

[29] Y. Gu, Z. Zhang, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. An empirical study of high availability in stream processing systems. In *Proc. Middleware*, page 23, 2009.

[30] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *TPDS*, 23(12):2351–2365, 2012.

[31] P. Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[32] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, et al. High-availability algorithms for distributed stream processing. In *Proc. ICDE'05*, pages 779–790, 2005.

[33] G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru, et al. Consistent regions: Guaranteed tuple processing in ibm streams. *Proc. VLDB*, 9(13):1341–1352, 2016.

[34] J. Kreps, N. Narkhede, et al. Kafka: A distributed messaging system for log processing. In *Proc. NetDB*, pages 1–7, 2011.

[35] S. Kulkarni, N. Bhagat, M. Fu, et al. Twitter heron: Stream processing at scale. In *Proc. SIGMOD*, pages 239–250, 2015.

[36] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. In *Proc. SIGOPS OSR*, pages 35–40, 2010.

[37] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: continuous reliable distributed processing of big data streams. In *Proc. NSDI*, pages 439–454, 2016.

[38] B. Liu, Y. Zhu, and E. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *Proc. SIGMOD*, pages 347–358. ACM, 2006.

[39] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., 1st edition, 2015.

[40] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, et al. S-store: Streaming meets transaction processing. *Proc. VLDB*, pages 2134–2145, 2015.

[41] Microsoft. Azure event hub. https://azure.microsoft.com/en-us/services/event-hubs/.

[42] MySQL. Mysql. http://www.mysql.com.

[43] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Prod. ICDM Workshop*, pages 170–177. IEEE, 2010.

[44] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, et al. Ambry: Linkedin's scalable geo-distributed object store. In *Proc. SIGMOD*, pages 253–265. ACM, 2016.

[45] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. SIGMOD*, pages 1099–1110. ACM, 2008.

[46] Oracle. Package java.util.stream. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

[47] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *Proc. SoCC*, pages 195–208. ACM, 2015.

[48] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, et al. On brewing fresh espresso: Linkedin's distributed data serving platform. In *Proc. SIGMOD*, pages 1135–1146. ACM, 2013.

[49] T. Rabl, S. Gómez-Villamor, M. Sadoghi, et al. Solving big data challenges for enterprise application performance management. *Proc. VLDB*, 5(12):1724–1735, 2012.

[50] S. Sanfilippo. Redis. http://redis.io.

[51] Z. Sebepou and K. Magoutis. Cec: Continuous eventual checkpointing for data stream processing operators. In *Proc. IEEE/IFIP DSN*, pages 145–156, 2011.

[52] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[53] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, et al. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB*, 2(2):1626–1629, 2009.

[54] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, et al. Storm@ twitter. In *Proc. SIGMOD*, pages 147–156, 2014.

[55] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, et al. Apache hadoop YARN: Yet Another Resource Negotiator. In *Proc. SOSP*, page 5. ACM, 2013.

[56] R. Wagle, H. Andrade, K. Hildrum, C. Venkatramani, et al. Distributed middleware reliability and fault tolerance support in system s. In *Proc. DEBS*, pages 335–346, 2011.

[57] L. Xu, B. Peng, and I. Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *Proc. IC2E*, pages 22–31, 2016.

[58] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proc. HotCloud*, page 95, 2010.

[59] M. Zaharia, T. Das, H. Li, et al. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proc. HotCloud*, pages 10–10, 2012.