ReStream: Accelerating Backtesting and Stream Replay with Serial-Equivalent Parallel Processing

Johann Schleier-Smith Erik T. Krogen Joseph M. Hellerstein
University of California, Berkeley
{jssmith, etk, hellerstein}@berkeley.edu

Abstract

Real-time predictive applications can demand continuous and agile development, with new models constantly being trained, tested, and then deployed. Training and testing are done by replaying stored event logs, running new models in the context of historical data in a form of backtesting or "what if?" analysis. To replay weeks or months of logs while developers wait, we need systems that can stream event logs through prediction logic many times faster than the real-time rate. A challenge with high-speed replay is preserving sequential semantics while harnessing parallel processing power. The crux of the problem lies with causal dependencies inherent in the sequential semantics of log replay.

We introduce an execution engine that produces serial-equivalent output while accelerating throughput with pipelining and distributed parallelism. This is made possible by optimizing for high throughput rather than the traditional stream processing goal of low latency, and by aggressive sharing of versioned state, a technique we term Multi-Versioned Parallel Streaming (MVPS). In experiments we see that this engine, which we call ReStream, performs as well as batch processing and more than an order of magnitude better than a single-threaded implementation.

Categories and Subject Descriptors H.2.4 [*Database Management*]: Systems Concurrency; H.3.4 [*Information Storage and Retrieval*]: Systems and Software—Distributed systems

Keywords Stream replay, backtesting, distributed stream processing

SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
 ISBN 978-1-4503-4525-5/16/10…$15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987573

1. Introduction

Stream processing is a common computing paradigm for numerous applications. Security systems, recommender systems, financial trading systems, advertising, business intelligence and monitoring applications all benefit from its ability to produce real-time or near-real-time insights and actions from unbounded data sources. Recently, the database community has seen a rekindled interest in streaming, with advancements in scale [2, 49], consistency guarantees [36], and the two in combination [3].

In *replay* we feed a streaming system using stored event logs, rather than real-time data. A prominent need for replay is "what if?" scenario analysis, in which developers revisit a recorded event log to simulate the behavior of a new streaming program ahead of its deployment. For example, we might evaluate a new combination of triggers in a financial fraud prevention application before a production release. Replay also has a special place in tuning machine learning models, especially as applied to real-time recommendations, where it allows training and back-testing with new features (i.e., model inputs derived in new ways from base facts). Our experience in industry relates directly to this challenge [43]. Section 2 provides further context on the need for replay.

A simple calculation shows that the throughput demands for streaming replay are much greater than those for real-time streaming. Because replay scenarios can call for processing weeks- or months-worth of stored events while a developer waits, we desire a speedup of multiple orders of magnitude to bring job durations down to minutes. Providing interactive replay during development requires an efficient processing framework, a capacity for creating parallelism from a sequential log, and a program that operates on its inputs in order. Replay also shifts performance priorities from the latency of event handling required for live streams to the throughput of bulk log processing. In short, we want to combine streaming semantics with batch-processing performance characteristics. This represents a new point in the design space for data processing systems.

The requirements for accelerated replay can be summarized as follows:

- *Timestamped events from an ordered log*: Data arrives pre-ordered according to externally-provided timestamps. The log source may be partitioned, but events are sorted within each partition and timestamps align events across partitions to a total order.
- *Familiar stream programming*: Users provide blocks of familiar imperative code, triggered by the sequence of events in the stream. A simple model is to use Event-Condition-Action (ECA) rules, with the events, conditions and actions written in a modern programming language or a domain-specific language [20, 27].
- *Serial-equivalent deterministic results*: The results of processing a log must be repeatable and deterministic, equivalent to a single fixed ordering of event handling and independent of partitioning or parallelism during replay. The value of deterministic replay is well established in the debugging literature [32], and these motives translate to the development of data-driven applications.
- *Throughput via scale-out*: To provide the throughput required for quick replay of extensive logs, replay has to scale out much more aggressively than the live streaming systems it simulates. Modern batch processing systems are the benchmark here, scaling up to thousands of parallel machines processing petabytes of input data [19, 37].

In this paper we describe ReStream, a system we have built to explore these requirements. ReStream introduces a new dataflow execution model, Multi-Versioned Parallel Streaming (MVPS), that differs substantially from traditional systems for bulk or stream processing. A key theme in our design is *aggressive sharing of data*—referring to both the input stream itself and the state that is accumulated during processing of that stream. Key design points in ReStream include: (1) serial-equivalent parallel execution, (2) a shared scan of the input stream from which all operators can view each input datum, (3) globally accessible versioned state that provides logically sequential access semantics, and (4) dataflow in access to these state objects. We explain and expand on these ideas further in Section 3.

Our evaluation compares ReStream to multiple implementations using Spark [52], showing the value of serial-equivalence provided by MVPS. While throughput is comparable, ReStream provides consistent results for increasing cluster size and processing throughput, whereas Spark Streaming approximates the computation with deteriorating fidelity and Spark batch processing runs into memory limitations. We also compare ReStream to an efficient single-threaded implementation, demonstrating an order of magnitude greater throughput.

Our key contributions are as follows: We identify accelerated replay as a critical advantage for rapid development of applications that require backtesting; we develop the Multi-Versioned Parallel Streaming execution model, which allows for serial equivalent processing of stored logs using distributed compute resources; and we provide an implementation and evaluate it experimentally, demonstrating its practicality as well as its limits.

The remainder of this paper is organized as follows. We introduce replay workloads in Section 2, including a canonical example. Building upon this example, we describe the Multi-Versioned Parallel Streaming execution model in detail in Section 3, and its implementation in ReStream in Section 4. We describe an experimental evaluation of ReStream that probes the limits to scalability of accelerated replay in Section 5. In Section 6, we place this work in the context of the extensive literature on stream processing before looking ahead and summarizing in Section 7 and Section 8.

2. Motivation

We further highlight the need for accelerated replay with applications. We start with a canonical example which we will use in the discussion of ReStream's runtime in Section 4 and in the experimental evaluation in Section 5. We also describe further motivating examples that highlight the breadth of applications for streaming replay before continuing to a discussion of alternatives to ReStream's accelerated replay.

2.1 A Canonical Example: Labeling Spam

We focus this paper on a simplified example designed to be easy to understand while remaining representative of real-world workloads. At scale, any consumer service that accepts user input (e.g., messages or comments) has to deal with spam. An anti-spam system must be able to keep up with a rapidly changing world in which spammers constantly evolve their strategies. As attackers seek to evade blocking they frequently alter potentially identifying characteristics, rotating source IP addresses, sender accounts, or message payload, and modifying the automated scripts designed to imitate natural user activity. Spam filtering's production demands make stream processing a natural solution. It requires low latency because the spam filter may be deployed inline, holding up message delivery until a decision becomes available. It also requires real-time updates to recognize fast-moving patterns.

While an industrial anti-spam system needs many rules, we proceed with an expository example from a social network using one multi-part rule:

> *Message is spam* if
> 1. Sending user has (*messages sent to non-friends*) $> 2 \times$ (*messages sent to friends*)
>
> *and*
> 2. Message is sent from IP address for which $> 20\%$ of messages sent contain an e-mail address

Implementing this heuristic requires processing two types of events: new friendships and messages. The relevant trigger rules are as follows:

A. New friendship → Mark friend connection between user pair.
B. Message → Check whether sender and recipient have a friend connection; increment the appropriate count of messages sent to non-friends/friends.
C. Message → Increment message count for sending IP address. Scan through message content for e-mail address; if found also increment count of messages containing e-mail addresses.
D. Message → Check conditions (a) and (b) from above; if both hold, mark as spam.

It can be hard to say how a heuristic like this will perform once deployed in production. What fraction of spam messages will it catch? How often will it mistakenly label a legitimate message as spam? One also wonders how suitable the numeric constants embedded in the algorithm are. Replay can provide answers to these and other questions.

2.2 Other Motivating Examples

Applications that benefit from backtesting and stand to experience faster development with accelerated replay are numerous. Security-related applications beyond anti-spam include payment fraud detection and money laundering countermeasures [10], and computer network intrusion detection [38]. Financial applications include automated trading algorithms and market monitoring algorithms [50]. In commerce, online retailers often want to adjust prices dynamically as inventories and consumer interest fluctuate [17]; ride-hailing apps may seek to direct additional drivers to locations experiencing high demand [25].

Real-time trained systems, ones such as ad serving systems [34] that employ machine learning rather than simple ECA thresholds, can also reap big benefits from accelerated replay. The anti-spam rule described in Section 2.1 has a number of hard-coded parameters that could instead be learned, but doing so requires computing training data from history with temporal resolution just as fine as that available during real-time streaming. Replay makes this possible, providing a flexible capability for training and back-testing complex machine learning models [43].

2.3 Replay Alternatives

In the absence of serial-equivalent high-throughput parallelized replay introduced by ReStream, developers have had to use alternative, less powerful evaluation strategies.

Serial streaming replay: When data volumes are small, computations are simple, or when development cycles are slow, serial replay may be adequate. In previous industrial work we described the use of replay in developing recommendations for a social network with hundreds of millions of members [43]. In this environment, optimized serial replay allowed processing 1 billion events in a few hours, making it possible to train new machine learning models with dozens of features in the course of an afternoon. Though it proved valuable, the limited scalability of serial replay provided motivation for the development of ReStream.

Production "dark launch": Developers can release experimental code to production after modifying it to replace actions with logging. In our anti-spam example, this means recording the filtering decision rather than blocking spam. Though it provides high-fidelity evaluation, dark launch testing has slow turnaround, especially when rules require some time to reach a steady state, e.g., a spam rule that integrates behavior for one week will take at least one week to test.

Streaming replay without serial equivalence: Parallel partitioned dataflow operators are a common feature of modern streaming implementations. Systems such as Storm [49] and MillWheel [2] allow access to external state such as a shared database. This approach to sharing state presents a problem for serial-equivalent replay because progress between partitions is not perfectly synchronized. In our spam example, a message coming shortly after a friend connection may be misclassified if the two events are processed out of order. Since we contemplate a multiple order of magnitude acceleration relative to the real-time rate, events processed at about the same time during replay can be surprisingly far apart in the original log: events separated by hours may be processed during the same second. Systems such as Samza [5], and MUPD8 [31] maintain internal operator state and are able to guarantee in-order processing. However, the lack of shared state restricts the flexibility of the programming model, e.g., in referencing common subexpressions.

Batch evaluation: Batch processing systems [19, 52] are set-oriented rather than sequence- or stream-oriented. While they materialize state between batches, they do not support the sort of time-resolved state used in our anti-spam example. Summingbird allows one source program to execute in both streaming and batch environments, the former for real-time processing and the latter for off-line analysis, but it remains limited to calculations framed with batch processing semantics. It has full support for aggregation, and other set-oriented operations, but not for the trigger updates to mutable state used in streaming systems.

Spark's discretized streams approach [53] marks an intermediate point in the design space. Its batches are a temporal coarsening, one that grows as higher throughput calls for larger batches, a semantic coupling unsuitable for replay.

3. Execution Model

Most data analysis systems for both stream and batch processing follow a dataflow model, where records are passed among *operators* in a tree or Directed Acyclic Graph (DAG). In our work, we separate the definition of operators from the passing of information between operators. In particular, our operators are stateless: each operator maintains state by reading and writing to a shared global state store. Our operators also do not communicate directly; the communication

graph in our system is entirely implicit, and results from the data dependencies through the global state. This approach is similar of the tuple spaces abstraction [23] in that communicating entities are decoupled, but different in that we use versioned state to provide deterministic execution.

There is also a connection between our work and transaction processing. We enforce a serial execution schedule derived from the log ordering, a constraint appropriate to the needs of backtesting and replay. Transactional serializability also constrains valid execution schedules. It has been implemented for stream processing [36], but can produce non-deterministic results because it allows reorderings. This presents a scalability challenge since these reorderings may be influenced by runtime conditions, e.g., parallelism.

In this section we elaborate on the specifics of our execution model and highlight how it helps us achieve accelerated replay.

3.1 Goals

We set out to build a system that would provide the capability to execute a streaming query over a stored, immutable event stream as if those events were happening in real time (i.e., to *replay* them), and to do so in a way that is not only fast, but also highly scalable. Our specific desiderata were:

1. **Parallel scale-out** for *performance*. Just as parallel databases, big data processing frameworks, and many streaming systems rely on the use of numerous machines to process distinct portions of the input data, we require an execution model which allows us to easily scale up to tens or hundreds of machines.
2. **Shared global state** for *generality*. Though it is common to provide partition-local state semantics in streaming systems (e.g. Apache Flink [4]), it is also common to provide some way to share global state, e.g. through the use of some external database as in MillWheel [2] and Apache Storm [49]. So as not to restrict the programming model we require that the programmer is able to condition the processing of each event on the full system state.
3. **Serial-equivalent processing semantics** for *faithful replay*. We wish to return results identical to those produced by processing each event sequentially; in achieving parallelism we refuse to sacrifice accuracy.

These three goals are somewhat at odds. A single machine processing each event in sequence allows programmer flexibility and achieves serial-equivalence trivially, but scaling this approach to multiple machines involves "lockstep parallelism," performance-degrading coordination following each event. Disallowing communication between partitions allows scale-out with trivially serial semantics, but does not meet our requirement for globally shared state. Dataflow-oriented streaming systems which allow for global state typically do not provide guarantees about serial consistency of that state [2, 49], again meeting only two of our goals. Transaction processing, when applied to streaming [36], permits event reordering and may produce results different from those of in-order computation. To achieve all three goals simultaneously, we develop a novel execution model which we refer to as Multi-Versioned Parallel Streaming.

3.2 Multi-Versioned Parallel Streaming

The traditional dataflow model involves operators, chunks of computation which perform transformations on their inputs, passing data between one another in such a way that a DAG of transformations is formed; this complete graph is referred to as a "dataflow." Multi-Versioned Parallel Streaming (MVPS) builds off of and makes a few key departures from this model. It operates in a partition parallel context where the data is divided into distinct partitions, each processed in parallel by the same operators (as initially described by the exchange operator in [24]). MVPS augments a dataflow that may not be easily parallelizable, providing a greater level of concurrency and independence of scheduling.

All state in MVPS corresponds to an explicitly tracked timestamp, in our case one derived from the ordered log being replayed. To parallelize a dataflow, we execute all of its operators on each machine, using timestamps and versioned state to preserve serial equivalence, aiming to allow each machine to operate as independently as possible. The key insight in MVPS is to take a dataflow that is semantically very closely coupled—i.e. has tight consistency requirements between subsequent data inputs—and to allow for this coupling to be relaxed without sacrificing any logical consistency.

3.2.1 Comparison to Traditional Dataflow

In a traditional dataflow, operators accept an input record, perform some computation, and may output data to be processed downstream. These operators are linked together by their data dependencies to represent a dataflow graph which, taken as a whole, executes some processing logic. We describe here some of the key points of MVPS that differentiate it from this traditional model:

- *Dataflow via shared state objects*: In a traditional dataflow, data passes directly from operator to operator, e.g. through the use of call stacks or queues. In MVPS, data flows between operators implicitly through accesses on managed state objects.
- *Multi-versioned state*: State is stored in a versioned manner; each time an object's value is written, a new version is created, but the old version remains accessible to timestamped reads. The value of multiple versions is explained further in Section 3.2.2.
- *Shared-state semantics*: In traditional dataflow, operators may maintain some internal state, but typically this state is not accessible by other operators, and in a partition-parallel context, this state is not shared across replicas of the operator, e.g., [5]. In MVPS, operators do not store state; instead, they read from and write to managed state objects, which are globally accessible by all operators

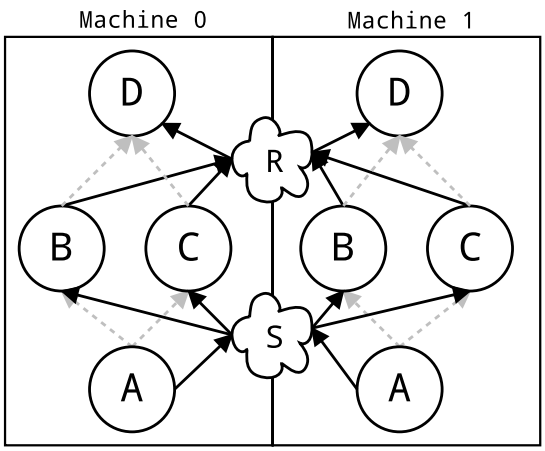

Figure 1: We show here how data dependencies translate between standard dataflow and dataflow via ReplayStates. Circles are operators. Clouds *R* and *S* are ReplayStates; we diagram them between the two machines because their full contents can be accessed by either. Gray dashed arrows denote the dependencies in a traditional dataflow sense. The black arrows show how these translate into MVPS, as a write-read dependency on some ReplayState.

across all partitions. These state objects represent a shared database that retains a history of changes to all values.

- *Full pipeline access to input data*: Traditionally, input data is supplied to the first operator in a pipeline only; subsequent operators are given only the output of their parent operator(s). We allow all operators to access the original input record they are currently processing, in addition to the shared state objects (analogous to data passed in the traditional dataflow sense). Together these last two items embody our theme of aggressively sharing state.
- *Time-ordered pipelined computation*: MVPS requires that each input datum has an associated timestamp to define a total order for the data set.
- *Serial-equivalent semantics*: Despite processing input in parallel on multiple machines, we provide the semantic model that all inputs are processed sequentially in order of their timestamps; this is done by differentiating *logical* processing time from *physical* processing time. Traditionally inputs are processed serially on the portion of a dataflow living on a single machine, but no guarantees are provided about the order of processing on different partitions of the input data.

3.2.2 Operators, Input Data & State

MVPS dataflows are comprised of operators and the state objects that they interact with. Operators themselves are stateless; they may store state only through the use of managed state objects that we refer to as ReplayStates, which are essentially time-aware versioned objects in a key-value store. Each operator is supplied with the input event currently being processed, and may additionally request data from or write data to any ReplayState. The access pattern of operators on ReplayStates implicitly defines the dataflow graph; if in a traditional dataflow operator *B* would consume the output of operator *A*, instead operator *A* will write to some ReplayState *S* and *B* will read from *S* to receive its input data (see Figure 1).

```
A1: W(a,15)  W(b,8)          W(c,19)
A2:                  R(12)             R(17)

B1: W(a,15)  W(b,8)          W(c,19)
B2:                  R(22)             R(17)
```

Listing 1: Two sample access patterns performed by operators $A1$ and $A2$ on some single-value ReplayState and operators $B1$ and $B2$ on another single-value ReplayState. `W(value,time)` denotes a write and `R(time)` denotes a read, where `time` is the logical time at which the operation occurs. Physical time moves from left to right. We show value-oriented writes for brevity but allow update functions expressed as closures (e.g. "add n to this value").

Input data must have a total ordering and must be available in this order within each partition. In our work we assume that the input data are timestamped events which are provided by an in-order scan. We refer to an event's timestamp as its logical time t_L, for simplicity assuming that these timestamps are unique, that any repeated timestamp values have been annotated with order prior to replay. The timestamp ordering defines where causal dependencies may arise; in particular, processing of an event with timestamp t_L may depend on any event with timestamp $t \leq t_L$ but may not depend on other events with timestamp $t > t_L$. This has implications on pipelining; an operator *B* which consumes data produced by operator *A* may only process events with timestamp $t < t_A$, where t_A is the lowest timestamp which any partition of *A* is still processing. This is discussed further in the context of batching in Section 3.2.3.

Since input events contain t_L the operators are always aware of the *logical* time at which their processing is occurring. All operations on ReplayStates are accompanied by a timestamp which specifies the logical time at which the access occurs; for writes, this specifies at what logical time the write goes into effect and for reads this specifies a logical point in time for choosing a version from the ReplayState. By specifying a logical time for accesses to state, we provide the ability to submit reads and writes to shared state that are not necessarily ordered in physical time as they are in logical time, while still achieving serial-equivalent results. For example, consider operators that are accessing ReplayStates using the access patterns shown in Listing 1. The access pattern on state *A* is valid. `R(12)` returns `b` since `W(a,15)` has not yet logically occurred at time 12. `R(17)` returns `a`, since at this point the first write has logically occurred. The access pattern on state *B* is not valid; `R(22)` returns `a`, reflecting the latest information available at its physical time, `W(a,15)`, but `W(c,19)` renders the read inconsistent. We discuss in Section 3.2.3 how we avoid such scenarios. Our use of timestamps in MVPS draws inspiration from timestamp-order concurrency control in databases [12].

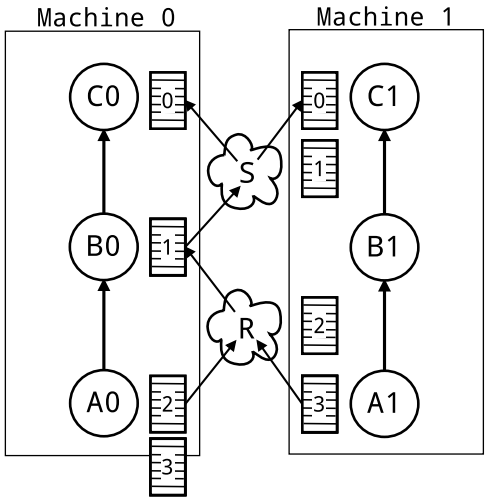

Figure 2: We show here different batches of events (rectangles) passing through the dataflow. The arrows in the center denote the flow of data in or out of the ReplayStates (clouds). Arrows between operators (circles) denote the implicit dataflow dependencies.

Each ReplayState is physically sharded across all machines, e.g. by hash-partitioning data items based on their associated key. We allow global access, so operators on all machines may access any portion of the key space within a ReplayState. We only disallow cyclic chains of write-read dependencies, as discussed further in Section 4.2. Transfer of data between machines is internal to ReplayStates and occurs transparently; each operator treats the ReplayState as a flat mapping without regard for locality.

3.2.3 Batching for Serial-Equivalence & Throughput

A carefully designed batch processing approach allows for high throughput while maintaining serial-equivalent results. Each partition of the input event stream is broken into batches which are defined by start and end points in the ordering, e.g. batch 0 may consist of events between logical time 0 (inclusive) and 10,000 (exclusive). Note that batches are defined by a range over the ordering, *not* by a specific number of events. We number these batches from 0 (the batch containing the events with the lowest timestamps) to N (the batch containing the events with the highest timestamps). Operators process events in these batches, accepting an event as input, reading data from ReplayStates, and performing writes to other ReplayStates to pass data to downstream operators.

Operators can pull data freely from ReplayStates, with two restrictions: (1) An operator A may not read any data from a ReplayState until all operators (on all machines) that write to that ReplayState have finished all batches up to and including the one at which A is attempting to read. That is, for a given ReplayState S and batch B_j, no reads may occur on any machine until all operators that may write to S have finished all batches $B_{i \leq j}$; (2) The timestamps for read and write operations are restricted to be within the boundaries of the current batch.

```
get(timestamp: Long, key: K): V

merge(timestamp: Long, key: K,
      mergeFunction: V => V): Unit
```

Listing 2: The ReplayState object API. `K` and `V` are the key and value types of a ReplayState and `V => V` denotes a function accepting an input of type `V` and returning an output of the same type.

Figure 2 shows an example of how operators may need to wait to process a batch. Operator $A1$ has finished submitting all writes for batch 2 to R, but $A0$ has not; thus $B1$, which may read R, cannot yet begin processing batch 2. However we see that $A1$ can begin processing batch 3. Operators $B0$ and $B1$ have finished submitting all writes for batch 0 to S, so $C0$ and $C1$ are both able to read from S and process the batch. This can happen concurrently with $B0$ continuing to write to S and read from R during the processing of batch 1.

These restrictions combine to ensure that all possible data that may affect the value of a read are present when that read occurs; even with out-of-order processing, operators always have a view of the state that is consistent with a fully serial execution. To see this, consider the case of a single ReplayState which is being read during the processing of batch B_m. Let T_i^S be the logical start time of batch B_i, and T_i^E be the logical end time. Once all writes are complete for batches $B_{i \leq m}$ on all machines (restriction (1) above), all subsequent writes must have a timestamp $t^W \geq T_{m+1}^S = T_m^E$ due to restriction (2). Applying restriction (2) to reads as well, we have $T_m^S \leq t^R < T_m^E$ for all read timestamps t^R within B_m. Since a write occurring logically after a read does not affect the value of the read, a read at t^R can depend only on writes with logical times of $t^W \leq t^R$. We have shown above that $t^W \geq T_m^E$ and $t^R < T_m^E$, thus there is no case where another write may modify the value of any of the reads being satisfied.

To limit memory consumption during long-running replay computations, we perform garbage collection at batch boundaries. When the last operator in the pipeline has finished processing a batch we scan the ReplayStates, retaining for each key only the most recent among versions with write timestamps before the batch end time.

4. ReStream Implementation

ReStream is our implementation of MVPS with batching, the execution model described in Section 3. It comprises approximately 4,000 lines of Scala code. In what follows we describe the design of the ReStream programming model and, with the aid of our spam labeling example, the important aspects of the implementation.

4.1 Programming Model

ReStream programs are specified as a set of "bindings," which are analogous to stateless dataflow operators. Bindings are similar to rule definitions from rule-based sys-

tems [26] in that they map an event type, optionally restricted by some conditional statements, to certain actions. Bindings in ReStream may execute arbitrary Scala code, but are limited to accessing shared state through framework-managed ReplayState objects. An I/O thread reads events from disk, filling a shared buffer that all operators access as they progress. Events that do not match the criteria for a given binding still conceptually flow through the dataflow operator, though no actual action is taken.

Listing 2 shows the ReplayState API. ReplayStates expose `get` (read) and `merge` (write) methods that accept the logical access time of the operation, the key of interest, and for `merge` a function that is applied to the previous value to obtain the new value. ReplayStates maintain the history of a value as it changes, making timestamped reads possible. Requesting a `get` at timestamp t_g will return the value obtained by applying all `merge` operations that have a timestamp t_m such that $t_m \leq t_g$, starting from a specified default value.

Listing 3 shows the implementation of our canonical example, the spam detector presented in Section 2.1.

4.2 Dataflow Analysis

Since a ReStream program leaves dataflow implicit, respecting the write-read ordering needs of MVPS discussed in Section 3.2.2 requires analyzing the code in its collection of bindings. Bindings interface with one another through ReplayStates so we can use Scala metaprogramming techniques to establish data dependencies, drawing a graph with directed edges running from each binding to the states it writes, and to each binding from the states it reads. Figure 3 illustrates this transformation for our spam detection example of Listing 3. We do not consider write-write dependencies, as these are automatically resolved by the multi-versioned state.

We require that the resulting dataflow graph be acyclic. A topological sort of the bindings within it then yields an execution order that guarantees write-read dataflow dependencies will be respected.

4.3 Implementation Details

The general architecture of the system is a driver-worker architecture, shown in Figure 4. There is one driver program and there are n worker nodes, each processing some subsection of the event history logs. The driver coordinates the progress of execution on each worker by sending updates in the form of low-water marks, timestamps that correspond to the least-advanced partition for each operator. The driver is not involved in communication of ReplayState data, which is partitioned across workers and updated by communication between them.

To avoid small network transfers and fine-grained coordination between nodes, communication only occurs once per batch per ReplayState. As an operator processes data within a batch it submits writes to a ReplayState, which stores them in a *local* buffer. Each buffered write eventually

```
bind { nfe: NewFriendshipEvent => // Binding A
  val userPair = (nfe.userIdA, nfe.userIdB)
  friendships.merge(nfe.ts,userPair,_ || true)
}
bind { me: MessageEvent =>        // Binding B
  val userPair = (me.senderId, me.rcvdId)
  if (friendships.get(me.ts, userPair)) {
    friendMsgs.merge(me.ts,me.senderId,_+1)
  } else {
    nonfriendMsgs.merge(me.ts,me.senderId,_+1)
  }
}
bind { me: MessageEvent =>        // Binding C
  ipMsgs.merge(me.ts, me.sendIP, _+1)
  if (SpamUtil.hasEmail(me.content)) {
    ipEmailMsgs.merge(me.ts, me.sendIP, _+1)
  }
}
bind { me: MessageEvent =>        // Binding D
  val ipTotal = ipMsgs.get(me.ts, me.sendIP)
  val ipEmails =
    ipEmailMsgs.get(me.ts, me.sendIP)
  val friendMessages =
    friendMsgs.get(me.ts, me.senderId)
  val nonfriendMessages =
    nonfriendMsgs.get(me.ts, me.senderId)

  if (nonfriendMessages > 2*friendMessages
      && ipEmails > 0.2*ipTotal) {
    // Message is spam; take action
  }
}
```

Listing 3: A sample program for the spam rules in Section 2.1; binding labels here match to the rules shown there. We use Scala's pattern matching syntax (i.e., `varName: MatchingClass => computation`) to specify which bindings are triggered by which events. `friendMsgs`, `nonfriendMsgs`, `ipMsgs`, and `ipEmailMsgs` are ReplayStates mapping `Long` keys to `Long` values, and `friendships` maps `(Long, Long)` tuple keys to `Long` values. We use the Scala lambda syntax in merge operations as a function which accepts an input _ and returns some function of that input (e.g. `_+1`).

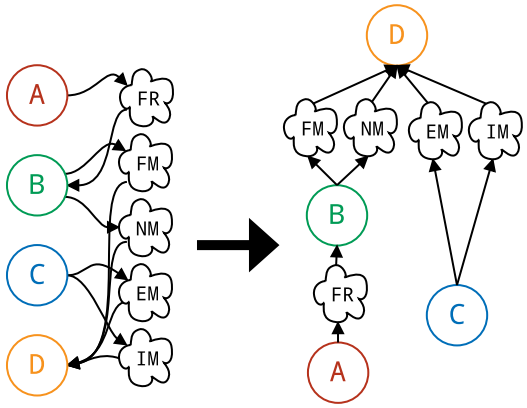

Figure 3: On the left we capture the dependencies in Listing 3; on the right we show the MVPS dataflow which is generated as a result. The colored/labeled circles correspond to colored/labeled bindings; the clouds correspond to the different ReplayStates (FR = `friendships`, FM = `friendMsgs`, NM = `nonfriendMsgs`, EM = `ipEmailMsgs`, IM = `ipMsgs`); and the arrows show which way data flows.

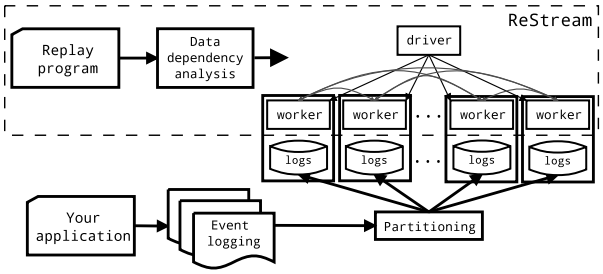

Figure 4: Overview of the ReStream architecture.

makes its way to the worker hosting the partition to which it corresponds. MVPS requires batch sizes expressed in units of time, but for convenience we configure ReStream with a batch size expressed in number of events, then use an estimate of the event rate to derive a corresponding time interval.

Implementing reads efficiently requires a further optimization. Rather than making remote requests while processing an operator, we first issue a batch of "pre-reads," requests to stage locally values that may need to be read. For each read encountered during program analysis (see Section 4.2), we generate a corresponding pre-read. Pre-reads are not fulfilled until all preceding writes are available. Once pre-reads are fulfilled and results are cached at the originating worker the operator can begin execution, now with all reads returning local results. Our present implementation generates pre-reads for all control flow branches, even though some of them may not be used. This presents an opportunity for further optimization.

Conceptually each operator in MVPS proceeds independently, making forward progress whenever allowed. As a practical optimization we group operators that do not have dependencies on each other, either directly or transitively. This results in fewer network transfers (since writes and pre-reads are communicated on a per-group basis instead of on an individual operator basis) and improves temporal locality by allowing multiple operators with a group to process the same event one after the other. For the computation graphs of Figure 3, our greedy algorithm groups together operators A and C (an alternate grouping of B and C would be valid as well). Grouping operators reduces some opportunities for parallelism so one may want to relax this optimization when executing multiple CPU-heavy operations.

We also place an additional restriction on the first operator (e.g., in the case of Figure 2, operator A) limiting how far it can advance ahead of other operators. This limit contains memory demands, both in the buffers that store writes and pre-reads, and in the version history of ReplayStates.

In summary, the key points of the ReStream implementation of MVPS are a compilation step to translate a set of bindings into an MVPS dataflow, limiting communication to batch boundaries to have local-only operations while processing each batch, and submitting pre-reads alongside writes to avoid paying a round-trip latency penalty when attempting to perform reads.

4.4 Fault Tolerance

We introduce here one modification to the scheme described so far: we ensure that messages (e.g. write operations) sent between machines are idempotent; this can be achieved through the use of unique tags on writes (or batches of writes) that allow a destination machine to ignore messages it has already seen. Since in MVPS computation (operators) is completely separate from state (ReplayStates), and since ReplayStates already maintain timestamped versions, this addition causes fault tolerance to fall naturally out of our architecture; as long as a recovering node has a way to access a version of its state which was consistent at some point in the past, it can continue processing forward as normal. Duplicated messages will be ignored at their destination, and eventually the recovering node will catch up to the others. Other nodes resend messages which may have been lost since the last consistent state; this can be achieved by locally storing sent messages until the destination machine confirms that their effect has been durably saved. Thus the key to efficient fault tolerance is efficient checkpointing of state.

It would be trivial, though costly, to implement fault tolerance by storing ReplayStates directly on a fault-tolerant storage system (e.g. Cassandra [30]) rather than in worker memory; a recovering node taking the place of node k would simply resume processing from the stream position at which the last write from node k occurred. However, we can leverage the batched nature of computation in ReStream to create a much more efficient scheme in which ReplayStates are persisted durably once every n batches, with each machine doing so independently without coordination.

This simple approach allows for a variety of trade-offs, notably between the frequency of checkpointing and the time to recover. It also allows for a variety of implementations, including different choices of durable storage.

5. Experimental Evaluation

Our experimental evaluation explores the effectiveness of ReStream's MVPS approach to achieving accelerated replay and probes its limitations. In addition to comparing with an efficient single-threaded implementation, we compare ReStream to multiple implementations built on Apache Spark. We outperform all, surpassing the single-threaded implementation by over an order of magnitude and exceeding the performance of Spark, even though its results sometimes represent an approximation. We also measure how interplay of the input data and the replay program impacts ReStream's ability to achieve distributed parallelism, and find a simple relationship explaining observations.

5.1 Benchmark Configuration

We evaluate the performance of ReStream using our canonical example, the spam detector of Listing 3, preferring this contemporary workload to classic benchmarks such as Linear Road [7]. We generate events which average approx-

imately 125 bytes in size and store the resulting stream, which simulates activity of 100,000 social network users. Most users prefer to send messages to their friends, while a small number of spam users send messages targeting random recipients. Observations of real-world social networks have found power law distributions with a range of coefficients from $\alpha = 1.50$ to $\alpha = 2.67$ [39]. We model individual user popularity and activity levels using $\alpha = 2.0$ in the comparisons of Sections 5.2 and 5.3. In Section 5.4 we evaluate ReStream across the range $\alpha = 1.25$ to $\alpha = 3.0$.

We use clusters of up to 32 Amazon EC2 c3.xlarge compute instances (4 vCPUs running on Intel® Xeon® E5-2680 v2 Ivy Bridge processors with 7.5 GB of memory [8]). We launch servers with enhanced networking into a cluster configuration (rate limited to 700 Mbit/s). We generate input data and save it to local SSD storage, spreading it across hosts evenly without partioning it in a problem-specific way.

5.2 Single-Threaded Comparison

Distributed data processing systems can be notoriously inefficient, paying a high cost for moving data between machines, especially when, like ReStream, they are built upon JVM technologies. As a reference for comparisons we developed a single-threaded implementation of ReStream, one that guarantees serial equivalence through sequential processing and that removes the overhead of coordination and distribution. We provision a separate dedicated thread for I/O, which further boosts the throughput of the processing thread and keeps with the pattern of our distributed implementation.

Using just two hosts, ReStream surpasses the performance of a single-threaded implementation by over 40%, achieving a Configuration that Outperforms a Single Thread (COST) [35] using 8 vCPUs. Beyond this, each doubling of the number of hosts increases throughput by roughly 70%, with 32 hosts performing replay $\sim$16x faster than the single-threaded implementation.

In another experiment, we modified our example, reducing its CPU consumption by eliminating a string search. In this test, where the workload skews heavily towards state maintenance and inter-host communication, ReStream needs four hosts to surpass the single-threaded implementation and achieves just over 2x speedup with 32 hosts. Scaling this modified workload is not practical but serves to show that ReStream manages to accelerate throughput even under challenging circumstances.

5.3 Apache Spark Comparison

Apache Spark provides us with a baseline for distributed system comparisons. It is a popular platform and it is implemented in Scala, as is ReStream. We developed three implementations of our benchmark using Spark.

5.3.1 Approaches

Spark Single Batch: As a first implementation, we built our example spam detector as a traditional batch processing job. It takes full passes over the input data set, one for each operator. State values are paired with timestamps so that logical times can be maintained, thus producing serial-equivalent results. This approach, equivalent to materializing all intermediate states, scales up to a degree and then runs out of memory.

Spark Streaming: In addition to an in-memory batch processing model, Spark offers a streaming API based on discretization and mini-batch processing [53]. Spark Streaming breaks computation into mini-batches to enable stream processing to be carried out in a traditional batch processing manner, buffering incoming events for a specific time interval before firing off a batch to process. To implement our replay workloads, each operator is implemented as a pass over the mini-batch of events, and we maintain global state between each batch. Since the temporal resolution of writes to and reads from global state is limited to the scale of the mini-batch, deviations from serial-equivalence arise. As we show in Figure 5, this approach forces a trade-off between accuracy and throughput; larger batches lead to higher throughput, whereas smaller batches yield greater accuracy. While Spark Streaming may work well in online deployments, pushing it to high throughput requires a temporal coarsening that erodes its ability to capture fine-grained event ordering.

MVPS on Spark: To bring serial-equivalent stream processing to Spark, we emulated the MVPS computation performed by ReStream using Spark APIs and data structures. This involves linking multiple invocations of the Spark Single Batch implementation and carefully managing state moved between them. Programming in this manner on Spark is unnatural, as any simple operation requires integrating past state via complicated join operations that must take timestamp alignment into consideration. We do not consider this style of programming practical. We view our MVPS on Spark implementation as principally useful for comparing performance, though one can also imagine an implementation of ReStream that generates code that executes on Spark.

5.3.2 Discussion

Figure 6 shows throughput of ReStream in comparison to Spark Streaming and MVPS on Spark implementations. We see that Spark Streaming, ReStream, and MVPS on Spark perform similarly for up to 16 hosts. Note, however, that while ReStream and MVPS on Spark produce serial-equivalent results faithfully, Spark Streaming produces *incorrect* results, approximations with throughput-dependent accuracy shown in Figure 5. In comparing to other approaches, we selected the Spark Streaming batch size to maximize throughput (with accuracy at the worst end of the range). When using MVPS on Spark with more than 16

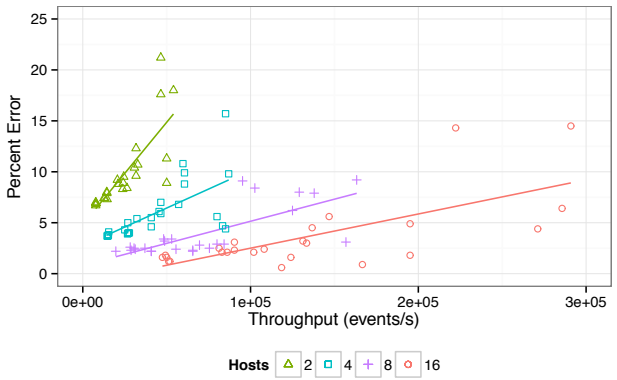

Figure 5: Spark Streaming trades accuracy for throughput. Percent error shows the deviation in the number of spam messages detected by the Spark Streaming implementation relative to a single-threaded implementation. Here we present results for batch sizes ranging from 12,500 to 200,000 events per host. Each trial processes a total of 1 million events per host and each point represents a single trial. The lines on the graph represent a best fit for each number of hosts. Higher error rates at lower host counts are likely a consequence of keeping the number of events per host constant—with more events, representing longer spam detection simulations, the network of users becomes more saturated with friend connections, new connections matter less and the coarsened temporal resolution has less of an effect on accuracy.

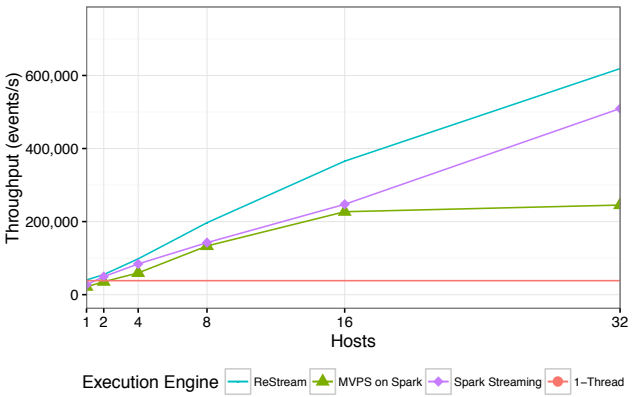

Figure 6: Scalability testing of ReStream and Spark. Note that Spark Streaming produces approximate results. The horizontal line, labeled as *1-Thread*, represents the single-threaded implementation described in Section 5.4. Each point represents an average of five tests, and we process 5 million events per host in all experiments. Our per-host batch size is 10,000 events in ReStream. MVPS on Spark uses very large batches which reduces the amount of state moved between batches; we range from 1 batch at low host counts to 4 batches at high host counts.

hosts we see that performance degrades significantly; this appears to be due to increased memory pressure from the higher overall event count. We note that ReStream does not have this problem, effectively scaling up to 32 hosts without any significant performance degradation.

The Spark Single Batch implementation performs identically to MVPS on Spark so long as the event stream remains short enough (when they reduce to the same program). Holding constant the number of hosts at 16, we found that the Spark Single Batch implementation begins to exhibit a slowdown at 80 million events as it experiences higher memory pressure from attempting to maintain a large amount of state. At and above 100 million events it slows down considerably, demonstrating the limitations of traditional batch processing in this scenario.

In both of these experiments, ReStream outperforms MVPS on Spark, exhibiting ∼50% greater throughput. We additionally note that Spark is substantially more mature than ReStream, and presumably has benefited from much more performance optimization.

5.4 Causality and Limits to Parallelism

ReStream delivers on the promise of parallel speedups with serial-equivalent results. So far, the results we have presented throughout this evaluation appear to belie the underlying tension between these two aims. In order to challenge ReStream's ability to scale we adjusted the parameter α governing the power law distribution of simulated social network activity. Higher α values create more balanced user activity whereas lower α values create greater imbal-

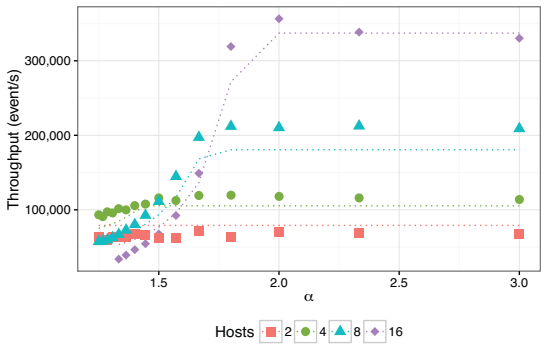

Figure 7: Shown with points is the throughput of ReStream as a function of network scale parameter α, here for per-host batch size 10,000. The lines represent the model of Equation 1.

ances, imbalances representative of the real-world impact of celebrities or "super users."

From a computation perspective, lower α concentrates more activity on individual users and IP addresses, reducing the opportunities for reordering even though the program remains the same. Figure 7 shows that when the number of hosts is small throughput does not suffer much from concentrations of activity. With a large number of hosts ReStream is able to gain more parallelism when α is larger.

We can better understand these behaviors by analyzing the structure of the state access dependencies that ReStream encounters during replay. We instrument ReStream to log a dependency graph, drawing an edge from every timestamp that reads a ReplayState to the timestamp of the most recent preceding write to that same ReplayState. For each batch of inputs processed by ReStream, we compute the longest path in the resulting dependency graph, which we refer to as the critical path. State along this critical path must be pro-

cessed in serial. When scaling out ReStream we maintain a fixed per-host batch size, an approach that ensures consistent amortization of coordination at batch boundaries. This means that the time interval covered by a batch (and, correspondingly, the critical path length) increases as the number of hosts increases.

Defining h as the number of hosts, b as the per-host batch size, and c as the average critical path length in a batch we fit the model

$$throughput \propto h \Big/ \left(\frac{h-1}{h} \times max\left(\frac{c}{b}, a\right) \right) \qquad (1)$$

where a is a free parameter estimated as 1.85 by non-linear least squares regression ($R^2 = 0.94$) for b ranging from 2,500 to 40,000.

The factor $(h-1)/h$ represents likelihood that an access to ReplayState goes to a remote worker. As the critical path length of causal relationships within a batch approaches the average number of inputs per host, imbalances necessarily arise, and some hosts end up doing more work than others. Our measured value of a suggests that this effect dominates when the critical path length starts to reach twice the per-host batch size. We conclude that a key scalability limit for ReStream lies in the causal linking of state required to maintain serial-equivalence.

6. Related Work

We start this survey of related work focusing on modern large-scale systems optimized for streaming. We then consider related ideas in the database, distributed systems, and complex event processing literature.

Modern internet-scale stream processing: Today's internet giants all have developed one or more in-house streaming platforms [2, 5, 31, 41, 49]. Among common applications are ad serving, security, recommendations, and monitoring. Availability and fault tolerance play a large role in this work, and while systems have high capacity and some tolerate out-of-order data arrival, they are not explicitly designed to support replay. A few systems integrate state with the streaming engine to gain fault tolerance [5, 14], while others externalize state in a separate service. Though these approaches can provide shared state, none integrates state management tightly with the computation model as ReStream does. IBM Streams [28] is a modern commercial product, likely among the most robust implementations of traditional dataflow stream processing. Naiad [40] provides a rich model for "timely dataflow" that is particularly clever in incorporating iterative computation with event processing. While Naiad represents a powerful substrate for distributed computation, it does not directly solve the state management challenges of accelerated replay. Trill [15] provides high throughput stream processing by using batching. Like ReStream it maintains the logical semantics of individual events, but does not scale across multiple hosts. Other recent stream processing system include Google's Cloud Dataflow [3]. It introduces powerful abstractions but does not provide a computing model with the aggressive sharing of state developed here, though it could perhaps serve as a substrate for MVPS similar to that which we developed with Spark (see Section 5.3).

Database systems: A number of stream processing systems emerged from the database research community in the early 2000s [1, 6, 16]. Some of this work identified the need for establishing a clear computation model and the need to deal appropriately with timestamps in streams [9]. Interestingly, replay is among the desiderata articulated during this era of research [45] and now our work moves toward making this practical. We also note an important contrast in mindset: whereas much of this research emphasized "load shedding," defined as discarding or postponing events when their arrival rate exceeds system capacity [47], a more appropriate response today would be scaling the computation across more cloud resources, an approach anticipated by Flux [44] and something that ReStream's computation model is particularly well suited to.

More recently, S-Store provides stream processing extensions to a high-performance transactional storage engine [36]. Comparing performance is challenging without standard benchmarks, but in published examples S-Store manages only a few thousands of events per second, two orders of magnitude less than ReStream. This is somewhat surprising as S-Store's serializable semantics allow significant latitude for event reordering (equivalence to *any* serial schedule of transactions), whereas ReStream is restricted to maintain serial-equivalence to history. The S-Stream authors recommend running it on a single core, so we assume it was not written with replay or throughput in mind.

Our approach is more aligned with recent work on deterministic scheduling for transactional systems [21, 48], which shows how establishing a total order ahead of time can improve throughput. Such systems sacrifice latency and require that queries pre-declare their read and write sets. ReStream takes advantage of the same basic trade-offs but introduces the MVPS state abstraction, and again has a much more constrained set of reordering opportunities than a serializable transaction system.

It seems useful to explore connections between ReStream and the transactional literature more deeply. Deterministic transaction systems are much like stream replay systems: they attempt to maximize throughput for processing a predeclared stream of requests. MVPS may offer opportunities for traditional transaction processing to decouple work more dynamically; conversely, the pre-planning and partitioning ideas in [21] may be useful in the ReStream context. More generally, serializability and serial equivalence are only two points in a wide space of consistency and isolation models. It would be useful to better understand the connections be-

tween stream processing and the full breadth of "update processing," transactional or otherwise.

Complex Event Processing (CEP) and other enterprise techniques: There is a well-established body of commercial work on CEP, technology which underlies a variety of business processes and financial applications. Commercial and open source implementations include Esper [20], TIBCO StreamBase [46], JBoss Drools [29], Oracle Stream Explorer [42], and others. Some of these systems scale through shared memory parallelism, but none support distributed operation as does ReStream. Their implementations derive from the literature in production rule systems [22], with limited temporal extensions [13], and they differ most markedly from other streaming systems in their ability to detect complex patterns in input event sequences [18]. Perhaps the MVPS approach can benefit CEP, which continues to make progress towards high performance [51] and consistency guarantees [11].

7. Future Work

Our present implementation of ReStream demonstrates the scalability of MVPS and the advantages of serial equivalence. A natural next step is to deploy ReStream with a team seeking rapid development cycles for real-time predictive applications. Exploring this has exposed the need for added state primitives, e.g., top-k lists, and for windowing, which can be sometimes optimized to reduce memory consumption by replaying events with lagged timestamps. We have already implemented some of these extensions.

Scalability beyond the range of tens of machines requires addressing additional implementation matters. As we increase the cluster size uncoordinated Garbage Collection (GC) across JVMs can cause performance degradation. Since our implementation introduces a synchronization barrier at batch boundaries the GC penalty scales in proportion to the system size, with each JVM's pause translating to a cluster-wide delay. We might alleviate this problem with coordinated GC [33] or by implementing MVPS in a language with explicit memory management.

The fault tolerance mechanisms of Section 4.4 represent a sizable design space, and our implementation in this area remains incomplete. Further work could support our claim that much of the bookkeeping needed for fault tolerance is already provided by ReStream's versioned state maintenance.

Another avenue which merits further investigation is preprocessing and partitioning input log files to reduce intermachine communication. For example, if most ReplayStates were keyed by the same identifier, say a user ID, then partitioning log files by that identifier could significantly reduce communication between machines. In a related vein, an astute reviewer suggests optimization using runtime information from previous replays, perhaps statistics or even event-level dependency information. The success of such optimizations likely depends on the nature of the code being run and on how it evolves during ongoing development.

For some workloads approximate results may be adequate. Among principled approaches we suggest two: sampling and bounded staleness. Sampling techniques estimate a value based on processing only a subset of the data and are well suited to various simple but practical computations. However, devising useful sampling approaches on social networks or other interconnected datasets appears unlikely. Bounded staleness in ReStream might replace the command "read at T" with "read at any time between T_1 and T_2," or "estimate at T to within ± 0.01." These alternative read formulations could provide opportunities for reordering, reduced communication, and partial aggregation.

ReStream demonstrates that serial-equivalence and parallel execution, two notions that might appear to be at odds, are not as incompatible as they seem. Our approach might also benefit live streaming, providing increased throughput in situations where processing latency can be relaxed. Looking beyond streaming, we imagine that the MVPS execution model might apply more broadly to scaling up data-intensive computation.

8. Conclusion

We have developed ReStream, a stream processing system designed for accelerated replay, for parallel processing of stored event logs with throughput much higher than the real-time rate. Such replay can meet the routine needs of developers evaluating new functionality, sparing them production releases and slashing turnaround times. We identified a new system design point in this need: a sequential stream processing programming model delivered with batch-processing throughput.

Our solution is a partitioned and pipelined dataflow with operators that communicate through shared global state. In a model that we call Multi-Versioned Parallel Streaming (MVPS), we use timestamp ordering at individual state elements to guarantee serial equivalence, even as we reorder execution in pursuit of parallelism.

Our implementation outperforms sequential processing by more than an order of magnitude and continues to scale even as well-known alternatives run out of memory or return degraded approximations. Our experiments support the view that aggressive sharing of versioned global state permits throughput bounded in the limit by causal dependencies.

Acknowledgments

The authors would like to thank Ion Stoica for helpful conversations that helped shape this work. We also thank the anonymous reviewers and the many members of the UC Berkeley AMP Lab for valuable feedback. This work was supported by AWS Cloud Credits for Research.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[2] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[3] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, 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, 2015.

[4] Apache Flink - Scalable Batch and Stream Data Processing. `http://flink.apache.org/`. Accessed: 2016-08-22.

[5] Apache Samza. `https://samza.apache.org/`. Accessed: 2016-08-22.

[6] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The Stanford data stream management system. *Book chapter*, 2004.

[7] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, pages 480–491. VLDB Endowment, 2004.

[8] AWS — Amazon EC2 — Instance Types. `http://aws.amazon.com/ec2/instance-types/`. Accessed: 2016-08-22.

[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16. ACM, 2002.

[10] B. Baesens, V. Van Vlasselaer, and W. Verbeke. *Fraud Analytics Using Descriptive, Predictive, and Social Network Techniques: A Guide to Data Science for Fraud Detection*. John Wiley & Sons, 2015.

[11] R. S. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. *arXiv preprint cs/0612115*, 2006.

[12] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[13] B. Berstel. Extending the Rete algorithm for event management. In *Temporal Representation and Reasoning, 2002. TIME 2002. Proceedings. Ninth International Symposium on*, pages 49–51. IEEE, 2002.

[14] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736. ACM, 2013.

[15] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[16] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668. ACM, 2003.

[17] L. Chen, A. Mislove, and C. Wilson. An empirical analysis of algorithmic pricing on Amazon marketplace. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1339–1349. International World Wide Web Conferences Steering Committee, 2016.

[18] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI*, 2004.

[20] EsperTech - Esper - Event Series Analysis and Complex Event Processing for Java. `http://www.espertech.com/products/esper.php`. Accessed: 2016-08-22.

[21] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, July 2015.

[22] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1): 17–37, 1982.

[23] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[24] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 102–111. ACM, 1990.

[25] J. Hall, C. Kendrick, and C. Nosko. The effects of Uber's surge pricing: A case study. *The University of Chicago Booth School of Business*, 2015.

[26] F. Hayes-Roth. Rule-based systems. *Commun. ACM*, 28(9): 921–932, Sept. 1985.

[27] IBM Operational Decision Manager. `https://www.ibm.com/software/products/en/odm`. Accessed: 2016-08-22.

[28] IBM Streams. `https://www.ibm.com/software/products/en/ibm-streams`. Accessed: 2016-08-22.

[29] JBoss Drools - Business Rules Management System. `http://www.drools.org/`. Accessed: 2016-08-22.

[30] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[31] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: MapReduce-style processing of fast

data. *Proceedings of the VLDB Endowment*, 5(12):1814–1825, 2012.

[32] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 100(4):471–482, 1987.

[33] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15. USENIX Association, 2015.

[34] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica. Ad click prediction: A view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1222–1230. ACM, 2013.

[35] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[36] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Çetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, et al. S-Store: Streaming meets transaction processing. *Proceedings of the VLDB Endowment*, 8(13):2134–2145, 2015.

[37] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3 (1-2):330–339, Sept. 2010.

[38] A. Milenkoski, M. Vieira, S. Kounev, A. Avritzer, and B. D. Payne. Evaluating computer intrusion detection systems: A survey of common practices. *ACM Comput. Surv.*, 48(1): 12:1–12:41, Sept. 2015.

[39] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 29–42. ACM, 2007.

[40] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[41] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[42] Oracle Complex Event Processing. `http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html`. Accessed: 2016-08-22.

[43] J. Schleier-Smith. An architecture for Agile machine learning in real-time applications. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 2059–2068. ACM, 2015.

[44] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 25–36. IEEE, 2003.

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

[46] StreamBase Complex Event Processing - TIBCO. `http://www.tibco.com/products/event-processing/complex-event-processing/streambase-complex-event-processing`. Accessed: 2016-08-22.

[47] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment, 2003.

[48] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12. ACM, 2012.

[49] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156. ACM, 2014.

[50] P. Treleaven, M. Galas, and V. Lalchand. Algorithmic trading review. *Commun. ACM*, 56(11):76–85, Nov. 2013.

[51] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 407–418. ACM, 2006.

[52] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, volume 10, 2010.

[53] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.