Clonos
Consistent Causal Recovery for Highly-Available Streaming Dataflows

Silvestre, Pedro F.; Fragkoulis, Marios; Spinellis, Diomidis; Katsifodimos, Asterios

DOI
10.1145/3448016.3457320

Publication date
2021

Document Version
Final published version

Published in
SIGMOD 2021 - Proceedings of the 2021 International Conference on Management of Data

Citation (APA)
Silvestre, P. F., Fragkoulis, M., Spinellis, D., & Katsifodimos, A. (2021). Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows. In *SIGMOD 2021 - Proceedings of the 2021 International Conference on Management of Data* (pp. 1637-1650). (Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems). ACM. https://doi.org/10.1145/3448016.3457320

Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows

Pedro F. Silvestre Marios Fragkoulis Diomidis Spinellis Asterios Katsifodimos
Delft University of Technology
{P.F.Silvestre,M.Fragkoulis,D.Spinellis,A.Katsifodimos}@tudelft.nl

ABSTRACT

Stream processing lies in the backbone of modern businesses, being employed for mission critical applications such as real-time fraud detection, car-trip fare calculations, traffic management, and stock trading. Large-scale applications are executed by scale-out stream processing systems on thousands of long-lived operators, which are subject to failures. Recovering from failures fast and consistently are both top priorities, yet they are only partly satisfied by existing fault tolerance methods due to the strong assumptions these make. In particular, prior solutions fail to address consistency in the presence of nondeterminism, such as calls to external services, asynchronous timers and processing-time windows.

This paper describes Clonos, a fault tolerance approach that achieves fast, local operator recovery with exactly-once guarantees and high availability by instantly switching to passive standby operators. Clonos enforces causally consistent recovery, including output deduplication, by tracking nondeterminism within the system through causal logging. To implement Clonos we re-engineered many of the internal subsystems of a state of the art stream processor. We evaluate Clonos' overhead and recovery on the Nexmark benchmark against Apache Flink. Clonos achieves instant recovery with negligible overhead and, unlike previous work, does not make assumptions on the deterministic nature of operators.

ACM Reference Format:
Pedro F. Silvestre, Marios Fragkoulis, Diomidis Spinellis and Asterios Katsifodimos. 2021. Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457320

1 INTRODUCTION

Stream processing systems have reached a high level of maturity in the last ten years, rendering them production-grade systems. Apache Flink [12], Apache Kafka [45], Samza [37], Jet [25] and other systems are serving important applications such as fraud detection in transactions, car-trip pricing, demand forecasting, stock trading, and even real-time traffic control.

Making large scale-out deployments fault-tolerant, is the key factor that enabled modern stream processing systems to be used in production settings. Streaming applications require reliable, highly available, and high-performance systems that perform consistent processing. Consistency in the modern streaming systems nomenclature is referred to as *exactly-once processing*, which means that an incoming record will apply its effects to the computation state of the system exactly once, even in the event of failures.

State of the art stream processing systems can provide exactly-once processing and high-availability under failures, but by design they have grown to support specific types of workloads summarized as analytics functions, for instance aggregates and joins. These computations, which are associated with streaming systems since their early times, are mostly deterministic and operate solely within system boundaries. In contrast, emerging classes of applications, such as general event-driven Cloud applications [11, 31], and Stateful Functions [2?] involve custom nondeterministic business logic and frequent interactions with external systems and databases. Because of their event-based nature and performance requirements, such applications are increasingly executed as dataflows on stream processors. To support these applications effectively, dataflow systems need to embrace nondeterminism in their fault tolerance and high availability approaches.

Existing fault tolerance and high-availability approaches [8, 9, 18, 29, 39] fail to address the exactly-once processing guarantees in the presence of nondeterministic computations, mainly because they make very strong assumptions that are not satisfied in modern stream processing workloads. Streamscope [35] and Timestream [38] assume deterministic computations, which restricts their applicability in practical scenarios while SEEP [36] and Rhino [18] additionally assume records to be timestamped with a monotonically increasing logical timestamp, failing to support out-of-order processing [34], which is supported by the majority of modern streaming systems today. Finally, Millwheel [3] is the only system that does not make these assumptions, but it requires a specialized transactional backend, such as Spanner [17], which requires atomic clocks not found in commodity clusters.

In this paper we propose Clonos, a fault tolerance and high-availability method built on top of Apache Flink with the goal of supporting all existing workloads that Flink supports today, i.e., Clonos, as opposed to related work, supports nondeterministic computations. Although Clonos was built and tested on Apache Flink [12], it can be used in any stream processor that simply supports FIFO per-partition channels and coordinated checkpoints [15]. In this paper we make two important contributions. First, we describe a protocol and the associated system components to perform local recovery without the need to restart a complete streaming topology, aiming at high availability and low latency with exactly-once processing guarantees. No existing work has addressed this problem on a feature-rich production-grade system. Second, we deal with the inherent nondeterminism of practical stream processing workloads in a manner transparent to application programmers.

SIGMOD '21, June 20–25, 2021, Virtual Event, China.
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457320

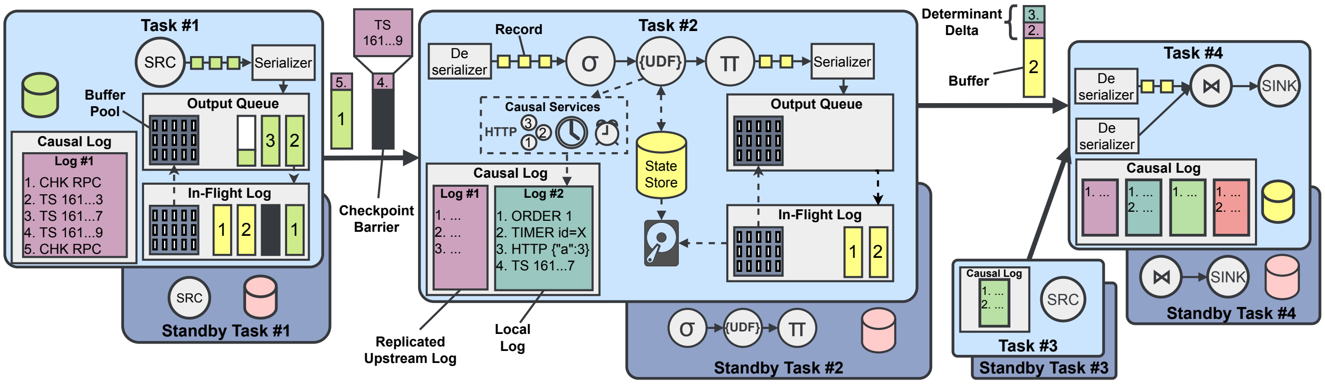


Figure 1: Approach overview.

To build Clonos, we implemented in-flight record logs and lineage-based replay for local recovery, standby tasks and live state transfer for high availability, and causal logging [20] for exactly-once consistent execution of nondeterministic computations and system functions. We present the recovery protocol, the high-availability mechanisms, the means to track nondeterminism, and a set of noteworthy system design and implementation decisions that render Clonos a practical replacement for Flink's fault tolerance mechanism. In short, with this paper we contribute:

- a novel fault tolerance approach that combines checkpointing, standby operators, and causal logging to:
  - provide exactly-once consistent local recovery and high availability on a production-grade system, and
  - support nondeterministic computations and system functions
- an analysis of nondeterminism in stream processing and how Clonos guarantees exactly-once processing
- thorough empirical experiments carried out in a realistic deployment

The rest of the paper is organized as follows. Section 2 offers an overview of our fault tolerance approach. Section 3 outlines the stream processing model used in the paper and includes preliminaries regarding rollback recovery, causal logging, and Apache Flink's execution model. Section 4 analyzes nondeterminism in stream processing and how it is addressed with Clonos, while Section 5 shows how Clonos guarantees exactly-once processing. Section 6 reports important design decisions necessary to make Clonos practically appliable. Finally, Section 7 presents a broad set of experiments and Section 8 presents related work. We conclude in Section 9.

2 APPROACH OVERVIEW

Clonos' main goal is to localize the impact of a failure to the minimum: only failed tasks need to recover from failure, and their upstream and downstream tasks take minimal action towards helping the failed tasks to recover. Recovering locally with exactly-once processing guarantees is challenging: in order to recreate the local state of the failed task, we need to use the most recent checkpoint of that task, and replay all the input records whose effects (on the state) have not been checkpointed. Another difficult problem is record deduplication: because some of the records have already been produced by a failed task, the recovery protocol needs to ensure that those messages are only processed once. The problem becomes even harder for nondeterministic computations that may produce different output (and operator state) for the same input across executions. Achieving all this in a highly-available manner where recovery has to be blazingly fast and the impact to the system's performance minor, is very challenging. Besides local recovery, Clonos features a high availability mode where it uses standby tasks with preloaded state to speed up recovery and further lower the impact of a failure. Below we give an overview of our recovery protocol.

2.1 Normal Operation

Figure 1 depicts a simple job with four tasks and their corresponding standbys. Each task executes a set of operators.

In-Flight Records. These are records that have been produced by an operator since the last successful checkpoint; i.e. their effects have not yet been recorded to the downstream operators' state. Tasks that send their output to downstream tasks (#1, #2, and #3) maintain a log of the in-flight records in memory until the next checkpoint is complete. This practice is the foundation of the upstream backup strategy [29].

Figure 1 captures a snapshot of the execution when the job is processing records of the yellow epoch. When a record reaches Task #2, it is processed and the output record (assuming for simplicity a function that produces a new record for each input record) is put in the output queue. Once the output record is transmitted over the network, it is added in the in-flight log. The in-flight log is segmented into epochs, such that whenever a checkpoint completes, all records in epochs prior to the checkpoint can be removed.

Log of Nondeterministic Events. Tasks maintain a log [20] of determinants for recording information about nondeterministic events and operations. In addition, each task shares its log incrementally with downstream tasks as we describe in Section 4.3. We present the different types of nondeterministic events and operations in Section 4.

Standby Tasks & State Snapshots. In high availability mode Clonos deploys standby tasks that mirror operator state, but remain idle in that they do not take part in data processing. Each standby task receives state snapshots of its corresponding running task after each checkpoint.

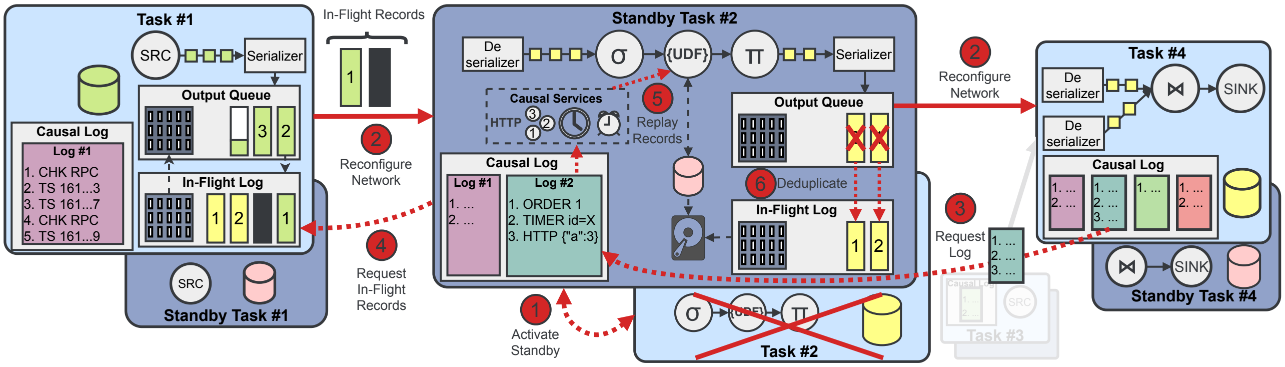


Figure 2: Steps of the fault recovery protocol.

2.2 Recovery protocol

Let’s assume that right after the execution snapshot depicted in Figure 1 a failure kills task #2. Figure 2 highlights the steps of our recovery protocol.

1. Activate New/Standby Task. The job manager initiates the fault recovery procedure, which starts a replacement task. In high availability mode, the topology maintains shadow/standby tasks that already contain the latest checkpointed state and remain idle until they are instructed to run by the job manager.

2. Reconfigure Network Connections. The standby task dynamically connects with the upstream and downstream task(s) of its predecessor in the topology.

3. Retrieve Determinant Log. The recovering task retrieves its predecessor’s determinant log from its downstream task(s).

4. Request In-Flight Records. In parallel to step 3, the standby task sends an in-flight log request to its upstream task(s) which specifies the epochs to replay.

5. Replay In-Flight Records. Each upstream task replays its in-flight records for the requested epoch and channel. In this case task #1 will replay the records of the yellow and green epoch in order. The recovering task (task #2) begins processing these records. Whenever it reaches a nondeterministic operation, the task instead reads from the determinant log the expected result of the operation.

6. Deduplicate Output. In parallel to step 5, the recovering task uses its determinant log to ignore output that its predecessor produced before failing. These output records are instead used to rebuild the in-flight log state.

Clonos’ recovery protocol differs in a number of ways from upstream backup [29] where upstream tasks replay the output records to recovering downstream tasks. Specifically, our protocol:

- uses checkpoints to reduce the duration of replay,
- uses determinants to deduplicate records at the sender following a failure, and to capture many sources of nondeterminism that we describe in the paper, and
- is optimized for the architecture and capabilities of today’s distributed streaming systems, which feature shuffles, asynchronous data transfer, checkpoints, processing-time semantics, out-of-order processing, and communication with external services.

2.3 Applicability & System Requirements

Clonos makes two assumptions. The first is the existence of reliable FIFO channels between a pair of tasks, i.e., for each channel, the downstream task receives all records in the same order that the upstream task has produced them. The second assumption is a checkpoint mechanism that creates snapshots of the system’s global state in regular intervals. Although our concrete implementation of Clonos is in Apache Flink v1.7, both assumptions are satisfied by mainstream streaming systems. For instance, Apache Samza [37], IBM Streams [30], and the latest version of Spark [8] also provide such FIFO channels, while Streams, Jet and Trill support checkpoints. Clonos’ approach can also be easily adapted to systems using uncoordinated checkpoints, through the use of backwards flowing checkpoint complete notifications.

Clonos’ implementation requires extending multiple system components such as the job manager, scheduler, checkpoint & fault tolerance mechanisms, the network stack, and the base stream operators. Clonos’ implementation is available online.[1]

3 PRELIMINARIES

This section provides the necessary background on the concepts used throughput the paper. We focus on current recovery mechanisms for stream processing and how these relate to rollback recovery schemes and causal logging.

3.1 Streaming Model

Stream processing systems [3, 12, 30, 37] process unbounded collections of *records* continuously by ingesting them into a dataflow graph where *edges* denote record streams and *vertices* denote operators. Each *operator*, receives records from an upstream operator, applies a computation on those records, and produces output records that it sends to the next operator(s) *downstream*. Each operator that produces output retains *output buffers* for sending output records downstream efficiently in batches.

3.2 Checkpoint-based Rollback Recovery

The main fault tolerance mechanism in modern scale-out streaming systems such as Apache Flink, IBM Streams, Trill, and Jet, is converging towards periodic Chandy Lamport-style [15] checkpoints

[1] https://github.com/delftdata/Clonos

of the system's global state [10, 12, 14, 30]. To recover from a failure, systems roll back the state of all operators to the latest checkpoint and resume data processing from a specific input offset, possibly replaying part of the computation that was lost during failure. This stop and restart strategy can achieve *exactly-once* processing guarantees [10]: the effects of all input records will affect the system's operator state exactly-once. However, as the execution graph grows, so does the downtime and latency incurred by the restart. In the event of a single failure the complete execution graph needs to be torn down and restarted from the latest global checkpoint.

This can be fixed with local rollback recovery schemes that, in-addition to the checkpoint also store in-flight records: a copy of all records they have produced since their last checkpoint. If a task fails, the system can roll back to its last checkpoint and replay its incoming records from the upstream tasks. Local recovery approaches that use in-flight logs [22, 36] can recover faster, but require two restrictive assumptions: *i*) that operators are deterministic (Section 4), i.e., reprocessing the same record a second time will yield the same output, and *ii*) that each record can be identified uniquely via a logical timestamp. During replay, tasks downstream from the failure can apply dedupication using these timestamps. Clonos lifts those long-standing restrictions using causal logging.

3.3 Log-based Rollback Recovery

Log-based rollback recovery has been extensively studied in the context of distributed systems. A stream processing system can be seen as a message-passing system executing processes that send and receive messages. In the sequel, we will refer to messages as *records*. Log-based approaches rely on the *piecewise deterministic assumption* [21], which states that all nondeterministic events can be identified, and the system can log their determinants. To reproduce a nondeterministic event[2] e (e.g., a timer, a random number, the result of a call to an external service/system), one must store the event and its *determinant*, denoted by $\#e$.

However, having the determinants alone is not enough to replay the nondeterministic events. To replay record reception events, it is required that the record contents be replayed as well. This can be done in one of two ways: *i*) either the receiver can log the record contents together with the determinants or *ii*) the sender can keep a log of the sent messages that are not yet stable in a so-called *in-flight record log*. The second case is more common, because the first requires logging a large number of messages in stable storage. Instead, the in-flight record log can be kept in volatile memory, because after a failure it can be deterministically rebuilt using the input streams and determinants.

3.4 Causal Logging

Causal logging [6, 19] is a log-based rollback recovery approach particularly well-suited to stream processing. Unlike *pessimistic logging*, causal logging maintains the determinant log in-memory and unlike *optimistic logging* it ensures the *always-no-orphans property* [4] (Equation 1), allowing for localized recovery. An orphan process is defined as a process whose state depends on a nondeterministic event e that cannot be reproduced during recovery [21]. If a nondeterministic event cannot be reproduced, then the state of orphaned processes must be rolled back to before that event, in order to ensure consistency.

$$\forall e : \Box(\neg\,\mathrm{Stable}(e) \implies \mathrm{Depend}(e) \subseteq \mathrm{Log}(e)) \tag{1}$$

where Depend(e) is the set of processes whose state was affected by e according to the *happens-before* relationship. Log(e) is the set of processes that have logged e's determinant in volatile memory and Stable(e) is a predicate which becomes true when e's effects are stored in stable storage (i.e. checkpointed). Finally, the operator $\Box$ is the temporal always operator.

Causal logging ensures that either *i*) all processes that depend on e have logged its determinant or *ii*) e is stable. If a set of processes $\mathcal{F}$ fails, then for all non-stable events e either $\mathrm{Depend}(e) \subseteq \mathrm{Log}(e) \subseteq \mathcal{F}$, in which case there is no orphan, or $\mathrm{Depend}(e) \subseteq \mathrm{Log}(e) \nsubseteq \mathcal{F}$ in which case at least one surviving process has the determinant of e, and can share it with the recovering processes.

Causal logging can be optimized by ensuring that no unnecessary determinants are sent to processes that do not depend on them by strengthening the always-no-orphans property as follows.

$$\begin{aligned}\forall e : \Box(\neg\,\mathrm{Stable}(e) \implies \\ ((\mathrm{Depend}(e) \subseteq \mathrm{Log}(e) \land \diamond(\mathrm{Depend}(e) = \mathrm{Log}(e)))))\end{aligned} \tag{2}$$

This property conveys that, while e is not stable, all processes dependent on e must have logged it and – eventually $\diamond$ – the ones that have logged it will be no more than those who depend on it. However, processes only depend on events of other processes if they receive messages from them, because those events happened before the delivery of the message. Thus, there is no need to send extra messages containing determinants, since the determinants a process needs can be *piggybacked* on the message that makes it causally dependent on those determinants.

Finally, in causal logging if the number of possible concurrent failures is bound to be not greater than a value f, it is possible to implement stable storage while avoiding disk access by logging to $f + 1$ processes [5]. In this case, one process may avoid sending its determinants to processes that have not logged them, if enough processes have already logged them for them to be considered stable.

$$\begin{aligned}\forall e : \Box((|\,\mathrm{Log}(e)| \leq f) \implies \\ ((\mathrm{Depend}(e) \subseteq \mathrm{Log}(e) \land \diamond(\mathrm{Depend}(e) = \mathrm{Log}(e)))))\end{aligned} \tag{3}$$

4 DEALING WITH NONDETERMINISM

Nondeterminism causes a lot of issues with the recovery of streaming topologies. The main issue arises when, upon failure and recovery, one needs to deduplicate records which have been generated twice, during replay. If the recovering operator (the one producing the duplicates) is deterministic, downstream operators can simply eliminate the duplicate records, because they know they have received them before. However, if the recovering operator is nondeterministic, it means that upon recovery, it may generate different records and/or in a different order. In that case, the downstream operators cannot correctly eliminate duplicates as they cannot distinguish them from non-duplicates. This is a very simplistic example of the relationship of local recovery schemes and determinism.

[2] Not to be confused with *stream events* which are used interchangeably with records in database research nomenclature.

Clonos is the first local recovery scheme to offer exactly-once processing guarantees in in the lack of determinism by tracking all sources of nondeterminism, and by leveraging causal logging.

Causal Logging for Stream Processing. Clonos leverages causal logging [19] to address the issues of nondeterminism. Unlike message-passing systems, the dataflow operators that process a streaming query are multi-threaded, including threads for data processing, timers, networking, flushing and receiving RPCs. Most of the different threads affect state and generate records at arbitrary system time that affect processing. In addition, a stream processing system offers operations that rely on system or processing time, such as processing time windows. All of these nondeterministic computations and functions need to be controlled in order to provide replayable job executions in a streaming system.

In the rest of this section, we analyze the sources of nondeterminism (Section 4.1), and elaborate how we deal with them (Section 4.2) including what we term *causal services* – a programming abstraction to support nondeterminism for system programmers but also to users authoring UDFs. In Section 4.3 we present the causal log and in Section 5 we discuss how Clonos guarantees exactly-once processing. Figure 3 depicts the concepts discussed this section.

4.1 Sources of Nondeterminism

We now exhaustively list the sources of nondeterminism that can be found in most modern stream processing systems.

Windowing & Time-Sensitive Computations. Streaming computations very often manipulate the inherent time dimension of data, which is based on event-, processing-, or ingestion-time. Of those, processing-time and ingestion-time are nondeterministic because they rely on the local system time at the operator where they are being processed. More specifically, when processing ingestion-time windows, the source operator simply adds a field in the record marking when that record entered the system. Upon a failure and replay, the ingestion time will change (the system time at the sources has changed), and windowing computations may not return the same results. The same holds for processing-time windows, which, instead of taking into account the ingestion time of records, simply trigger in periodic moments in time using *timers*, based on the local clock of the windowing operator.

Event-Time Windows & Out-Of-Order Processing. Event-time is quite different to processing-time. It is the time the records are generated in the input sources (e.g. sensors and mobile devices). In its simple form it is deterministic: no matter how many times one replays a stream, the event-time of each record does not change. However, event-time introduces another complexity: the possibility of records arriving from input sources out-of-order due to network congestion or other reasons [41]. Streaming systems like Google Dataflow, Apache Beam & Flink accept out-of-order events up to a *lateness bound* based on a low-watermark [34]: a marker generated at the input sources according to wall clock time that is then embedded in the data stream. Since low-watermarks are generated according to wall clock time, using timers, they are nondetermistic.

Timers. Timers are programmatic hooks which can be set to execute at some point in the future. Both the system and users can register timers. The triggering of timers is controlled by a timer thread, and the interleaving of operations between two threads is nondeterministic.

User-Defined Functions & External Calls. User-defined functions are not sandboxed: they are allowed to call external services, reach external key-value stores, and also make other asynchronous calls. Every interaction with the outside world is not expected to be deterministic. Consider, for example, a call to an external database that queries the current stock price; this can change at any point in time. As a result, calling external services cannot be considered deterministic and, during recovery, computations can change.

Random Numbers. Users may want to use random numbers in operators. Pseudo-random number generators are typically initialized using the current time producing nondeterministic results.

Keyed Streams & Record Arrival Order. To parallelize and group streams, it is common to partition them using a partitioning key. We refer to such a stream as a keyed stream. Downstream operators (e.g. a `reduce` operator) receive inputs from multiple upstream operators, on a per-key basis. The issue here is that, depending on the network speed, the connection between various operators, etc., the order in which records arrive is not always the same upon recovery. A lot of times, operators are order-sensitive; in that case, the operator will also generate records in a different order than the one before the failure happened. In other words, operators that process multiple inputs are not deterministic. To make it deterministic, we need to fix the order in which records are replayed on recovery (see Section 4.2).

Checkpoints & Received RPCs. Checkpoint-based fault tolerance protocols inject checkpoint barriers into the dataflow graph that instruct operators to checkpoint their state as they pass through them. Those barriers are injected in the dataflow graph through an RPC according to the system time of the job manager (e.g., every 10 seconds). Any RPC received by a task which affects its state is nondeterministic.

Output Buffers. Records are grouped into buffers before they are sent downstream. Buffers are sent either when they are full or when the downstream task demands a buffer.[3] In the latter case, the buffer might not be complete. Thus, the decision of whether the buffer will be split or not is very time-dependent and also depends on the request coming from downstream. This introduces nondeterminism in the size of buffers sent downstream that needs to be taken care of, such that the buffers can be retransmitted in the very same way during recovery.

4.2 Abstracting Nondeterminism with Services

To hide the complexity of causal logging and recovery from users that code user-defined functions (UDFs), operators have access to "causal services" that abstract the complexity away. For instance, assume that in Figure 3 a user-defined function calls the `Timestamp` service, which returns timestamps. Under normal operation, the service generates a nondeterministic timestamp and appends it in the causal log. During recovery, when the user-defined function requests a timestamp from the `Timestamp` service (shown in Listing 1), the service will instead return a timestamp read from the

[3] To handle backpressure, streaming engines allow downstream operators to either pause transmission or force the transmission of a buffer as soon as it contains a record.

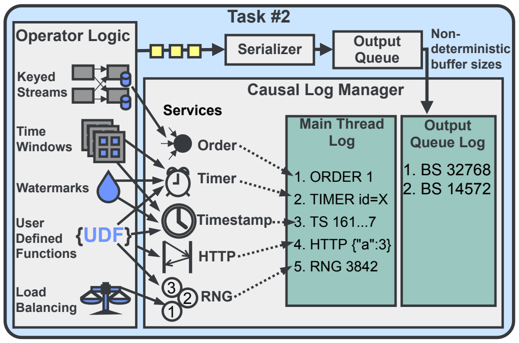


Figure 3: Services offered by the causal log's services API.

causal log. Users can register their own nondeterministic computations in Clonos by providing an anonymous function as in Listing 2. Determinants and causal logging, as well as recovery in all cases are done transparently. Behind the scenes, Clonos applies the anonymous function as Listing 3 shows. We describe the built-in causal services below.

Record Processing Order. The `Order` service is an internal service (not exposed to users), which logs the order in which input records are processed. For performance, this is done at the level of buffers, and each buffer is fully processed before the next is deserialized.

Timers & Received RPCs. Timers fire asynchronously to the main thread, thus their recovery is more complex. We first introduce unique IDs to every timer callback function. Then, we modify timer internals to register a "TimerFired" determinant in the causal log, containing its ID and stream offset at which it fired. During recovery, if a "TimerFired" determinant is encountered, we wait for the same stream offset to be reached. We then use the timer ID to obtain and execute the corresponding callback. RPCs received by an operator are treated similarly.

Wall-Clock Time. When the `Timestamp` service is used to retrieve wall-clock time under normal operation, the service retrieves a timestamp from the system and logs it prior to returning it to the user. During recovery, the same service will return the logged timestamp instead of a fresh wall-clock timestamp. Since this service may be called multiple times per millisecond, if the time granularity allows it (e.g., asking ms-granularity timestamps multiple times within the same ms), instead of generating a new timestamp on every call, this service utilizes timers to only update a stored timestamp periodically (each ms in this case). In between updates, the service simply returns its cached timestamp. This reduces the amount of determinants generated by two orders of magnitude without a large loss in time granularity.

Calls to External Systems. Calls to external systems must be done through causal services (e.g. the `HTTP` service), which persistently record the response in the log. The response can then be deserialized from the log during recovery.

Random Numbers. Instead of storing the numbers generated, the `RNG` service generates a new random seed on every checkpoint and stores it in the log. During recovery, the seed is read from the log and the numbers generated can then be deterministically reproduced.

```
HTTPResponse response = ctx.getHTTPService().get("host:port/path");
long ts = ctx.getTimestampService().currentTimeMillis();
```

Listing 1: Using built-in services.

```
CausalService myService = ctx.buildService(input -> {
  // The user provides any nondeterministic logic and simply returns
  // an object that implements the Serializable interface.
});
//User can later use this logic with any input argument i
Output o = myService.apply(i);
```

Listing 2: Boilerplate to add a new causal service in a UDF.

```
class CausalService<I, O extends Serializable> {
  Function<I, O> f; //defined when service is built
  public apply(I input) {
    Output determinant;
    if (recoveryManager.running())  //Normal operation
      determinant = f.apply(input);
    else                            // Recovery phase
      determinant = recoveryManager.replaySerializable();
    causalLog.append(determinant);
    return determinant;
  } }
```

Listing 3: Internal causal service logic.

4.3 Causal Log

The causal log stores the determinants for every nondeterministic event executed by a task. It is split in two parts. There is a causal log for the main thread of a task and a separate causal log for each of the output channels in that operator.

In a typical message passing system with a single thread of execution, causal logging [19] would require maintaining only one log generated by that single thread of execution. However in a typical scale-out streaming system, the main processing thread is separate from the network threads for performance, and they communicate through shared data structures. As the main thread writes to an output buffer, the output queue may decide to send the non-full (nondeterministically sized) buffer downstream. Thus, each queue has a causal log, where the size of buffers sent is recorded (Figure 3). This log is used during recovery for deduplication.

All Buffers Carry Determinants. Whenever a buffer of data is sent downstream, a causal log delta piggybacks on that buffer. The delta contains all the entries of the output queue logs and the main thread log since the last buffer dispatch. Note that the main thread log is essentially replicated to all downstream operators, as formally required by causal logging [19]. The idea behind this is that whenever a downstream operator receives determinants, those should be able to fully restore the upstream operator.

Replicating Determinants to Downstream Tasks. The downstream task, upon receiving the buffer and the delta of the two logs, appends those updates to the corresponding task causal log. In this way, before data is allowed to affect the state, the causal information necessary to recover it is already stored. In order to be able to afford two successive tasks failing, one might also want to replicate the determinants of each task to a deeper sharing depth.

Truncating Causal Logs. The causal log is organized in segments according to epochs and is truncated whenever a checkpoint completes; the causal log is only needed in the middle of an epoch, when a local recovery has to complete using in-flight logs and the older checkpoint as we describe in the next section.

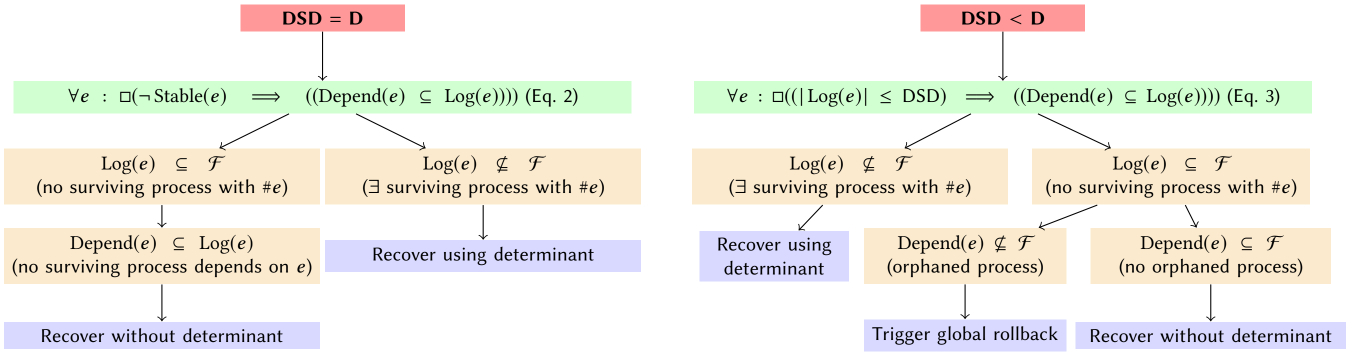

DSD = D
∀e : □(¬Stable(e) ⟹ ((Depend(e) ⊆ Log(e)))) (Eq. 2)
Log(e) ⊆ 𝓕
(no surviving process with #e)
Log(e) ⊈ 𝓕
(∃ surviving process with #e)
Depend(e) ⊆ Log(e)
(no surviving process depends on e)
Recover using determinant
Recover without determinant
DSD < D
∀e : □((|Log(e)| ≤ DSD) ⟹ ((Depend(e) ⊆ Log(e)))) (Eq. 3)
Log(e) ⊈ 𝓕
(∃ surviving process with #e)
Log(e) ⊆ 𝓕
(no surviving process with #e)
Recover using determinant
Depend(e) ⊈ 𝓕
(orphaned process)
Depend(e) ⊆ 𝓕
(no orphaned process)
Trigger global rollback
Recover without determinant

Figure 4: Exhaustive list of failure cases & DSDs with the recovery scenarios that need to be followed in each case.

5 EXACTLY-ONCE RECOVERY

In this section we show how Clonos deals with recovery and how it guarantees *exactly-once processing* with local recovery, using a causal log and in-flight records. In Section 5.5 we describe how we could extend Clonos to guarantee *exactly-once output.*

5.1 Lineage-based Replay

When a new task replaces a failed task it needs to process the records of the current checkpointing epoch. Therefore, it requests from its upstream tasks to replay their in-flight record log. Upon the in-flight log request, upstream tasks start to replay the buffers contained in their in-flight log, in the same order they were dispatched prior to failure. The replay protocol of Clonos is based on lineage. If a task does not have an in-flight record log to replay for a downstream task (typically because itself just recovered from a failure), it will ask its upstream tasks to replay their in-flight record log. This lineage-based process can reach recursively the operator graph all the way up to the input sources, which we assume to be available to provide their input on demand.

5.2 Determinant-based Deduplication

When recovering a task, the task replays the received in-flight records and produces output. Achieving exactly-once processing when performing local recovery requires deduplication after replay. In prior work [22], such deduplication is rather simple: each operator is considered to be deterministic, and all produced records bear a logical timestamp. The downstream operator can simply discard the records bearing the already seen logical timestamps. However, receiver-based deduplication wastes bandwidth.

Instead, deduplication in Clonos is done in two concurrent steps. First, as the main processing thread recovers, it uses its causal log to produce the exact same output records. Concurrently, the network channel threads use their causal logs, which contain only information about the size of buffers received downstream, to reconstruct the same buffers as sent before.

5.3 Correctness of Recovery Scheme

In the following, we analyze the conditions under which recovery can be performed using determinants depending on the depth to which determinants are shared. The correctness of causal logging as a rollback recovery approach has been formally proven in the past [4, 6]. Since Clonos tracks nondeterminism for multiple threads (the main processing thread and one thread per output channel), we model each thread as a process and recover them in unison. Thus, the proofs applicable to pure causal logging trivially extend to Clonos. However, ensuring exactly-once processing when locally recovering a failed operator remains open; we show Clonos guarantees it in the following paragraphs.

We base our reasoning on exhaustively enumerating the different states that the recovery mechanism can reach, depending on the determinant replication strategy and different failure scenarios. Our aim is to show that independently of: *i*) how the determinants are shared with downstream operators, and *ii*) which failure scenario takes place, there is a mechanism to recover the topology with exactly-once processing guarantees. This is done either by retrieving determinants and deduplicating using them or by falling back to restarting the complete dataflow graph as in reference [10].

Assume that in a DAG composed of $\mathcal{N}$ tasks with a maximum depth D (source tasks have a depth of zero) $\mathcal{F} \subseteq \mathcal{N}$ tasks fail. Clonos can be configured to use a determinant sharing depth (DSD) as large as the graph depth or smaller than the graph depth. The determinant sharing depth also defines the number of consecutive tasks that can fail concurrently without creating orphan tasks. For instance, a sharing depth of two, means that the determinants of a task a are sent to the downstream task b directly, and b forwards the same determinants to its downstream tasks c and d. If both a and b fail, we can recover them from the determinants that are stored by c and d. In the following, we analyze the different recovery cases, as depicted in Figure 4.

Case 1: ***DSD* = *D*.** We deal first with the case where the determinant sharing depth equals the depth of the dataflow graph, i.e., $DSD = D$. Note that in this configuration Clonos follows the condition stated in Equation 2. As such, determinants for a nondeterministic event e whose effects have not yet been globally checkpointed, are propagated to all downstream processes. Determinants piggybacked on a buffer are logged by a task (processed by the causal log manager) before the operator state becomes dependent on them (before the operator processes the buffer's records), and as such at no moment do we break the condition that $\text{Depend}(e) \subseteq \text{Log}(e)$. Two failure cases can occur:

- $\text{Log}(e) \subseteq \mathcal{F}$: Since the condition $\text{Depend}(e) \subseteq \text{Log}(e)$ also holds, then no surviving process depends on e, meaning that a different execution path may be taken without breaking consistency or the always no-orphans condition.

- Log(e) $\not\subseteq \mathcal{F}$: At least one surviving process has the determinant of event e, in which case it guides the recovery, either by ensuring the main thread follows the correct execution path or by ensuring an output thread deduplicates a buffer and thus the records it contains.

Translating this to stream processing: this case can only happen when for the failure of a given task, all downstream tasks also fail, as otherwise, downstream tasks will have the necessary determinants to bring the failed tasks into a consistent state with the surviving downstream tasks. The extreme case happens when $\mathcal{F} = \mathcal{N}$, in which case no task is dependent on any other and recovery is effectively equivalent to restoring a global checkpoint and beginning replay from the graph's input sources.

Case 2: $DSD < D$**.** In the case where the determinant sharing depth is less than the depth of the dataflow graph, Clonos follows the condition of Equation 3 by not sharing e's determinant to a depth greater than DSD. In this case, there is the possibility that Log(e) $\subseteq \mathcal{F} \not\subseteq$ Depend(e), meaning that some orphaned process remains. When one of the orphaned processes receives a determinant log request from a recovering task for a log it does not have, it will escalate this to the JobManager, which will trigger a full rollback of the DAG, thus achieving exactly-once processing guarantees. The alternative case is that Log(e) $\not\subseteq \mathcal{F}$, in which case at least one surviving task has the determinants of nondeterministic event e, and can guide the recovery of the failed tasks which depend on it.

Summarizing, the recovery cases depicted in the leafs of the trees in Figure 4, show that there are cases *i*) when the determinants are not required for recovery, *ii*) when determinants are required and can be found in some surviving task, and, finally *iii*) (the worst case) when the topology can recover with a global rollback recovery mechanism.

5.4 Trading Correctness for Performance

Clonos is flexibly configurable in terms of its fault tolerance guarantees. By combining its different building blocks, it can achieve different processing guarantees, as follows.

At-most-once. By disabling both in-flight logging and causal logging/determinants, failed tasks will be recovered with gap recovery [29], leading to inconsistent state with at-most-once processing guarantees, but incurring very little overhead.

At-least-once. By setting the determinant sharing depth $DSD = 0$, only in-flight logging is enabled, and failed tasks are recovered with divergent rollback recovery, achieving at-least-once processing guarantees with very little overhead due to Clonos' no-copy in-flight log (Section 6.1).

Exactly-once. By enabling causal logging it is possible to perform consistent recovery on failed tasks, providing exactly-once processing guarantees, again with little overhead. If the overhead of causal logging becomes a concern, Clonos can also trade-off determinant sharing depth for performance. The determinant sharing depth is set to the depth of the graph by default, but by lowering it to another number f, the determinant sharing overhead is reduced in exchange for supporting at most f concurrent consecutive failures. In this case, if a larger than f number of failures happens, Clonos can again be configured to favour either *i*) availability with at-least-once guarantees (skips deduplication step), or *ii*) consistency by falling back to recovery using the latest global checkpoint [11].

5.5 Achieving Exactly-once Output

There are two common methods for achieving exactly-once output[4] in stream processing systems. The first solution is idempotent sinks [1, 7, 8] and the second is transactional sinks [8, 10]. The idempotent sinks do not work in the face of nondeterminism, while the transactional sinks introduce latency proportional to the checkpoint interval. Clonos, can be trivially extended to achieve *exactly-once output* by piggybacking serialized determinants on records sent to downstream systems (e.g. Kafka). This downstream system has to store these determinants, and be able to return them when requested. The determinants of a previous epoch can be truncated after each checkpoint. In this way, Clonos can achieve very low-latency exactly-once output since the outputs can be consumed already by external systems without having to wait for a checkpoint to complete and the transactional sinks to perform a two-phase commit.

6 SYSTEM DESIGN DECISIONS

In this section we detail the interesting and non-trivial design decisions of the various building blocks comprising Clonos.

6.1 In-flight Record Log

Clonos stores in-flight records in each task that sends its output to other tasks downstream. Because an upstream task may send records to multiple tasks downstream, the records are logged by output channel (partition), which corresponds to a specific connection with a downstream task. To optimize throughput, Flink sends records downstream, serialized in network buffers. Clonos logs these buffers in the in-flight log before they are sent.

Avoiding Buffer Copies. Normally, when a buffer is sent over the network, it needs to return to the buffer pool of the output channels and be recycled. However, the in-flight log also needs to store that buffer. One choice would be to copy it over, and then recycle the buffer. However, to avoid copying buffers, whenever a buffer is dispatched from the network layer downstream, the output channel simply hands over that buffer to the in-flight record log. This, however, can cause deadlocks: the output channels could be waiting for buffers to become available in order to serialize output records, but no buffer would be available if they would all be used by the in-flight log.

Large Buffer Pools & Backpressure Delay. After going through multiple design and implementation iterations optimizing throughput and latency, we opted for the following strategy. As seen in Figure 1, each channel maintains two buffer pools. One buffer pool serves the output channels and the other buffer pool serves the in-flight log. When the network layer hands over a buffer to the in-flight log, in exchange, the in-flight log hands over an empty buffer to the buffer pool of the output channel. Interestingly, in our experiments we have seen that a network connection between

[4] This is also known as the *output commit problem* [21]

two operators needs around 10 buffers per channel - not more. Adding more buffers to output channels might look rational but it has an important side effect. It breaks the natural backpressure mechanism. The more buffers available for output, the slower the reaction of upstream operators to slowdowns from downstream operators, delaying the backpressure messages to propagate back to the sources. That is precisely the reason why Apache Flink, by default, uses a very small buffer pool for output.

Clonos, however, has to address an additional issue owed to the small number of buffers available to the output queue. While a task upstream of a failure replays buffers to the recovering task downstream, its main processing thread continues to produce records that very quickly fill the buffers available to the output queue as those buffers cannot be sent before the replay completes. With no buffers available processing stops for *all output partitions/channels* of the task. This issue conflicts the philosophy of Clonos that the system should never stop making consistent progress. We solved it by placing the buffers at the back of the in-flight log even though they were still unsent. This is allowed because if the downstream is failed, then we are guaranteed to replay them at a later time.

Spilling to Disk. Our in-flight log is segmented into epochs, and whenever a checkpoint completes successfully, the in-flight log is truncated up to that checkpoint, making the data buffers available in its local buffer pool. The in-flight record logs are kept in memory by default. Depending on the checkpoint frequency and input throughput pace, the in-flight log may grow beyond the size of the log's buffer pool leading to blocked processing and backpressure. To counteract this issue, we introduced an asynchronously spilling in-flight log, that persists buffers to disk (Figure 1), recycling them whenever necessary. The spilling in-flight log transitions seemlessly from on-disk buffers to in-memory buffers and prefetches on-disk buffers to speed up the replay process. It functions according to the following four (configurable) policies.

- In-memory: keep all buffers in memory.
- Spill-epoch: spill each epoch as soon as the next one starts.
- Spill-buffer: spill each buffer as it arrives.
- Spill-threshold: Spill all buffers whenever the buffer pool's ratio of available buffers drops below a configurable fraction.

The in-memory and spill-epoch policies both suffer from the possibility of blocking processing when the checkpoint interval is too large. Instead, the spill-buffer approach entails additional synchronous work that creates increased overhead and lacks batching of I/O operations. The spill-threshold approach offers a well-rounded solution to the above issues.

6.2 Network Channel Reconfiguration

Clonos applies reconfiguration of network channels dynamically in order to introduce a new task in the topology. Once the new task receives the acknowledgment from an upstream task, it requests to establish a persistent network connection with its upstream tasks. After a new connection has been setup, the lineage-based replay protocol can begin.

We found it particularly challenging to re-engineer the network stack in order to establish connections of tasks while jobs were executing. The main issue was to align network buffers and counters that match buffer sequence ids. In addition, record deserializers per input channel often keep state from one buffer to the next as they wait to receive the remaining part of a record with the next buffer.

6.3 Standby Tasks

Each standby task mirrors a running task. It contains the same processing logic and stores the same type of state as the one it mirrors. If a running task fails, its corresponding standby task substitutes it. In contrast to a running task, its standby task remains idle unless it is commanded to run.

The allocation strategy of standby tasks underlies an important tradeoff between resource utilization and failure safety, even performance. By controlling the affinity and anti-affinity of standby tasks' allocation, stream processing jobs can tune the amount of compute nodes they utilize for standby tasks. Each saving in resource utilization directly reduces Clonos' safety guarantees since co-locating two or more standby tasks on the same node makes Clonos more susceptible to a potential failure of that node.

Performance is another factor to weigh in when deciding the placement of standby tasks, i.e. their allocation strategy. Depending on a job's processing, co-locating two specific tasks may be critical for performance. If performance optimization is more important than failure safety, a job may choose to co-locate the corresponding standby tasks. By default, Clonos allocates standby tasks using the same allocation strategy provided by a job for the running tasks.

6.4 State Snapshot Dispatch

Similarly to related work [29, 33], Clonos transfers the state snapshot of each running task to its corresponding standby task once a checkpoint is complete. Clonos' state snapshot dispatch can leverage the various approaches offered by the underlying system, such as direct transfer to the local disk of the standby task via a file url or transfer to a shared file system. In addition, if the state backend supports incremental checkpoints then the cost of dispatching state depends on the state's delta instead of its absolute size. By receiving state snapshots regularly, standby tasks are behind their running counterparts only by a checkpoint or less.

It is important to note that the state transfer process is bound by checkpoint frequency and checkpoint duration, which depends on the state size. A state snapshot should not take longer to dispatch to a standby task than the job's checkpoint frequency. In practice, however, this can be avoided if concurrent checkpoints are never performed. Under these assumptions, a checkpoint is guaranteed to complete before the next one begins and state transfer is expected to complete before the next checkpoint's completion when using a distributed file system. Finally, if a standby task is called to run while a state snapshot is in transit Clonos will wait for the transfer to complete before starting the execution of the standby task.

7 EXPERIMENTAL EVALUATION

In this section, we first present our experimental methodology for running two categories of experiments: overhead experiments where we measure the overhead of Clonos in terms of throughput and latency under normal operation, and failure experiments where we study Clonos's fault recovery. In both cases, we compare with Flink, the engine on which our changes were introduced.

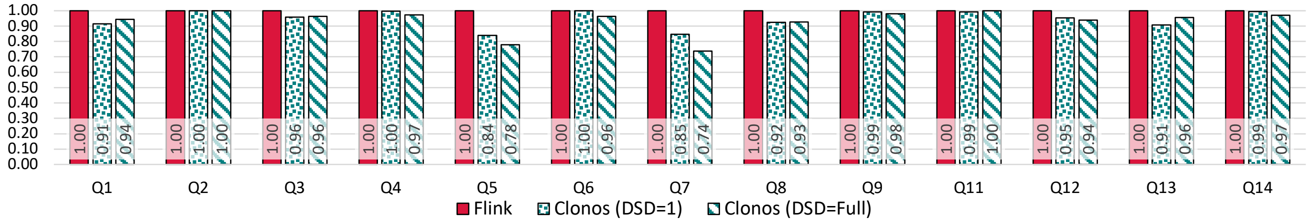


Figure 5: Relative throughput of Clonos (with DSD=1, DSD=Full) compared to "vanilla" Flink's recovery mechanism, under normal operation. The experiment performed on Nexmark queries.

7.1 Setup

We evaluate Clonos on a Kubernetes cluster hosted on a Cloud environment. The Kubernetes cluster hosts a 3-node Kafka cluster, which serves both as the data source and data sink of the failure experiments. An HDFS deployment, with a single NameNode and three datanodes, stores the operators' checkpoints. Finally, the Kubernetes cluster hosts a Flink cluster with 150 TaskManagers, each containing a single task slot. Each TaskManager has access to 2GB of memory, and two processing cores.

A given configuration's throughput is measured by sampling the Kafka cluster three times per second for the number of records in the output topic. Dividing the number of new records by the elapsed time, we obtain real-time throughput. A given configuration's latency is measured by sampling the output Kafka topics from each job and computing the output records' latency. Finally, we configure Flink to offer the fastest possible recovery, so as to provide a fair comparison. This means lowering the failure detection parameters to values not recommended for use in production. In particular, heartbeats are sent every 4 (default: 10) seconds, timing out after 6 (default: 60) seconds.

7.2 Workloads

Nexmark. Since Clonos can be a drop-in replacement for Flink jobs, we used the Nexmark [44] benchmark, along with the extra queries[5] implemented by the Apache Beam project. To enable this we implemented a Clonos runner for Apache Beam. Nexmark includes queries that perform filtering, joins, aggregates, complex windowing, etc. and serves greatly as a representative workload for evaluating stream processing engines. We have excluded Q10 from the benchmark because it requires access to Google's GCP service.

Synthetic. We also use a synthetic workload to be able to evaluate Clonos under configurable scenarios, not found in Nexmark and to avoid optimizations such as operator fusion. This way, for each operator, there is an extra layer of depth for which Clonos pays full network and serialization costs of determinants. For the synthetic experiments presented, we inject to Clonos multiple sequential failures, either concurrently or in intervals. In the interest of space, we only include a subset of our results.

7.3 Overhead Under Normal Operation

In this series of experiments we observe the performance of Clonos under normal operation, i.e., without failures, and quantify runtime overheads. We execute the complete Nexmark benchmark queries setting the degree of parallelism of each operator to 25, meaning that the different jobs occupy between 25 (3 operator stages for the simplest queries such as Q1-2) and up to 150 CPU cores (6 stages for Q7). Operator fusion is turned on.

What is the overhead of Clonos in terms of latency and throughput, under normal operation (no failures)?

In the interest of space, we do not plot latency measurements as we observed those to be stable and comparable to Flink's latency throughout our overhead experiments with a notable difference: the tail latency in the case of DSD=Full can be up to 20% worse (ca. 25ms) than vanilla Flink. For DSD=1 we have noticed an overhead of less than 10% in the worst case.

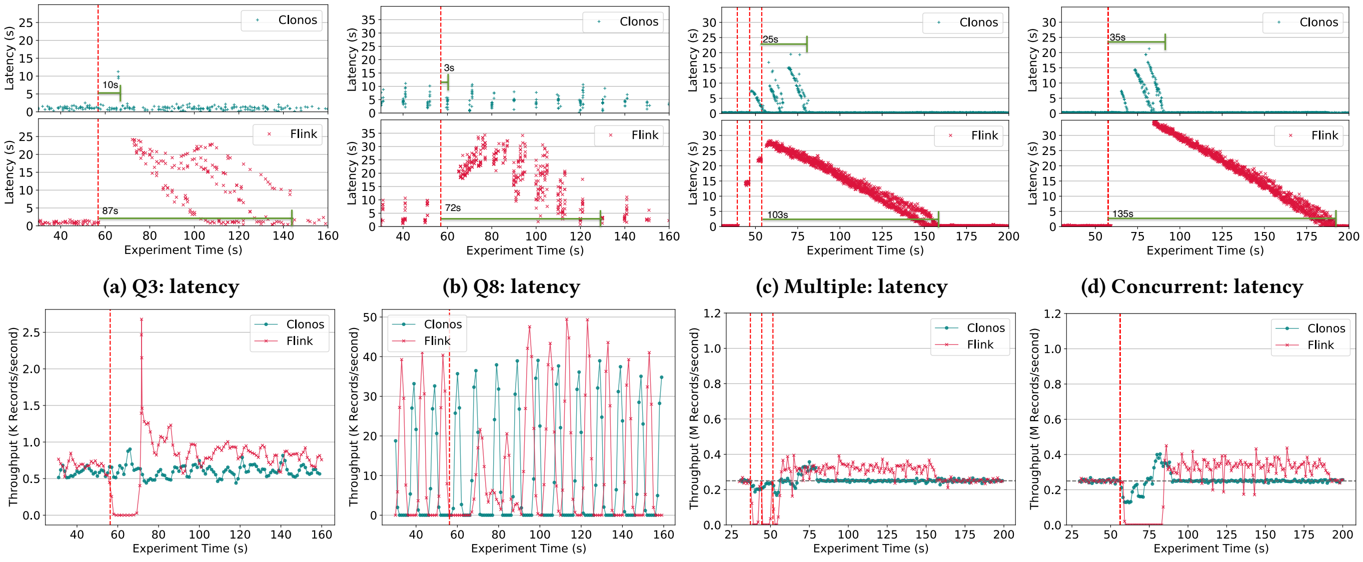
Figure 5 depicts the overhead of Clonos on throughput. First, we see that for simple queries such as Q1-Q2 which are implemented with simple `map` & `filter` operators (D=1) are not affected by the overhead that comes with Clonos, such as in-flight logging. In fact, such a small difference in throughput can easily be also attributed to the effects of the underlying infrastructure. The most complex queries are Q5 and Q7 which are implemented using an aggregation tree to handle skewed keys, and they also perform windowed aggregates. For both queries we observe that, since their depth D=6, the "Full" determinant (i.e., DSD=6) sharing has a high impact on throughput: up to 26%. However, a more reasonable DSD=1 or 2, yields around 15-16% overhead in throughput. We find this penalty in throughput reasonable, considering the benefits of Clonos' fast recovery times (next Section) and its ability to deal with non-deterministic operators. Finally, throughout the whole benchmark, we have observed an average penalty of 7% for DSD=Full and 6% for DSD=1 compared to vanilla Flink.

7.4 Clonos Under Failure Scenarios

For failure experiments we chose to present detailed throughput and latency metrics for two of the most interesting Nexmark queries: Q3, and Q8. In addition, we evaluate Clonos against Flink on multiple and concurrent failure scenarios using a synthetic workload.

Recovery Time. We define *recovery time* to be the time between the instant that a failure takes place and the instant that the recovering system's observed latency has returned to values within 10% of the pre-failure latency. This metric is used to evaluate a mechanism's ability to recover fast from a failure. Note that this metric also includes the time that a system needs to catch up with

[5] `https://beam.apache.org/documentation/sdks/java/testing/nexmark/`



(a) Q3: latency **(b) Q8: latency** **(c) Multiple: latency** **(d) Concurrent: latency**

(e) Q3: throughput **(f) Q8: throughput** **(g) Multiple: throughput** **(h) Concurrent: throughput**

Figure 6: Failure experiments with realistic workload (left) and multiple/concurrent failures (right)

the input stream. Although Clonos is operational in less than a second, a lot of practical use-cases (e.g., credit card fraud detection) require that the system, after recovery, can also catch up with the input stream throughput and get back on track in order to process data as soon as it becomes available.

What is the performance of Clonos with respect to latency and throughput in the presence of single-operator failures?

Nexmark. We focus on Q3 and Q8. Q3 performs a full history join and filtering operations, while Q8 performs a windowed join, which explains the throughput spikes as we measure throughput at the job output sinks. We have also experimented with Q4, Q5, and Q7 since they are the most complex queries, but those produce very few output records and they were inappropriate to plot and exemplify proper recovery times. In order to observe end-to-end latency, a regular amount of output records must be generated.

Figure **??** shows that Clonos recovers within 10s by leveraging standby operators and local recovery. After a sub-second switch to the standby operator, replaying the lost epoch took roughly 10s at which point a small number of queued records were emitted with 10s latency, before the system could catch up. During this time, the alive tasks continue operating under regular latency. Flink, however, loses availability on all tasks and takes at least 87s to recover and catch up. In addition, different output partitions recover at different speeds. This is indicated by the different lines of points visible in the plot. In Figure **??** we inject a failure to the join operator. Clonos recovers within 3s. Note that since we measure latency on the output records (end-to-end latency) the visible points arranged vertically signify records of different arrival times in their respective windows. The window range also explains the empty spots in the figure as the window fires every 10s. Flink, on the other hand, takes more than 72s to fully recover.

In terms of *throughput*, Figure **??** depicts Clonos' ability to instantly recover the job's original throughput, while Flink experiences a downtime of multiple seconds and a turbulent recovery. Notice how Clonos' throughput is barely affected following the failure. We can observe similar behavior in Figure **??**.

What is the performance of Clonos with respect to latency and throughput in the presence of multiple failures?

We perform our multiple and concurrent failure experiments at parallelism 5, operator graph depth 5, checkpoint interval 5 seconds, and per-operator state size of 100 MB. Specifically, Figures **??** and **??** depict an experiment where there are three failures with a 5-second interval, while Figures **??** and **??** depict an experiment with three concurrent failures. The failures are sequenced, meaning the failed operators have connected dataflows. We observe that independently of the frequency of failures (whether they are staggered or concurrent), Clonos' recovery behaves similarly. Before the downstream failures can be recovered, the upstream failures must finish recovering, such that they can replay their in-flight logs. Only partial throughput is lost during recovery, as records continue to flow through causally unaffected paths even though shuffle connections are used. Similarly, latency is only increased on a small subset of records flowing along causally affected paths and latency quickly returns to its pre-failure value.

7.5 Memory Usage

The memory usage of Clonos is completely bound by the size of the buffer pools configured (Section 6.1). We have experimented with different memory sizes and spill strategies for the storage of the in-flight record log as well as determinants. We have observed that while the spill-buffer strategy is much more conservative memory-wise, it leads to poorer and less predictable performance. The spill-threshold strategy presents deteriorating performance under 50MBs of space and has diminishing returns above 80MBs. Thus, all experiments used 80MBs of in-flight log space per task. When the in-flight record log would become larger than the available memory, the log spills buffers to disk. Since both reading and writing to it have a sequential access pattern, the "spill-threshold" strategy (Section 6.1) yielded the best results. The size of the determinant buffer pool has no effect on performance, but too small of a buffer pool may lead to deadlocks. Experimentally, we have found that for DSD=1 a determinant buffer pool of size 5MB is more than sufficient for most workloads. When DSD=Full, this value must be increased as D grows, as more logs are replicated.

8 RELATED WORK

Our contributions are related to fault tolerance, high availability, and causal logging. An elaborate study of fault tolerance and high-availability in stream processing is provided in a survey [24].

8.1 Fault Tolerance

A number of early stream processing systems provided fault tolerance, such as Aurora [16] and Borealis [9]. However, most fault tolerance approaches of the time did not recover a system-wide consistent state with very few exceptions [39]. More recent systems like Apache Flink [12], IBM Streams [30], and Microsoft Trill [14], achieve consistent exactly-once fault tolerance with global rollback recovery as described in Section 3. Other systems, such as Storm [43], Heron [32], and Samza [37], implement at-least-once consistency guarantee. Streaming systems to date increasingly try to handle failures locally, that is, without disrupting a job or regions of it, but only its failed components. Apache Spark [8] performs exactly-once local recovery but in a micro-batch processing model and assuming an idempotent sink that ignores already produced results on recovery.

Consistent local recovery is offered by SEEP [22] and its extension based on stateful dataflow graphs (SDG) [**?**], Timestream [38], Streamscope [35], and Rhino [18]. However, none of these systems supports nondeterministic computations and they make strong assumptions about input order. The only stream processing system that delivers consistent local recovery and can support nondeterministic computations with minimal assumptions is Millwheel [3]. However, Millwheel performs a transaction per record per operator on Spanner [17]. Spanner, to achieve low latency, depends on atomic clocks to operate which do not exist in commodity clusters. Clonos can provide Millwheel's guarantees and consistency on commodity hardware. Table 1 summarizes all systems' determinism assumptions.

Table 1: Assumptions of related work

System	Assumptions
Millwheel [3]	Scalable, transactional backend (Spanner)
Streamscope [35]	Deterministic computations and input
Timestream [38]	Deterministic computations and input
SEEP & SDG [**?**], Rhino [18]	Deterministic computations, monotonically increasing logical clock, records ordered by time.

8.2 High Availability

Existing work on high availability in stream processing [29] proposes active replication [9, 39], passive replication [27, 33], hybrid active-passive replication [26, 42], or models multiple approaches and evaluates them with simulated experiments [13, 29]. These approaches either constrain operator logic or support weaker than exactly-once consistency guarantees. Clonos delivers high availability based on passive replication by substituting only the failed tasks. At the same time Clonos maintains exactly-once consistency guarantees that cover nondeterministic computations using causal logging on a feature-rich production-grade system.

8.3 Causal Logging

We presented causal logging [19, 21] in Section 3. We have elaborated both the system design and implementation aspects of the causal log in Section 6 and the nondeterministic aspects in Section 4.

Among streaming systems, Timestream [38] and Streamscope [35] use an optimistic logging-inspired dependency tracking approach, which records input and output dependencies in computations and uses them to rebuild the state if needed. Instead, Clonos records all nondeterministic events and the order of execution. By additionally respecting the always-no-orphans condition, Clonos can guarantee consistent localized recovery.

Closest to the spirit of Clonos is lineage stash [46], which uses causal logging to provide exactly-once consistency with local recovery for nondeterministic operators. However, it does not support important nondeterministic functions in stream processing, such as timer-based services much needed for processing time windows and watermarks required for progress tracking and out-of-order data. In addition, it uses a micro-batch architecture while Clonos implements continuous data processing. Finally, Clonos also addresses issues of high availability with standby tasks, state shipping and reconfiguration.

9 CONCLUSIONS

In this paper we presented Clonos, a fault-tolerance and high-availability method built in Apache Flink as a replacement for its current fault tolerance mechanisms. Clonos, to the best of our knowledge, is the first fault tolerance mechanism which is applicable to a real, production-grade system and achieves consistent local recovery, high availability, and the flexibility of nondeterministic computations. Clonos has been a substantial engineering effort within our team (more than 20K LOC), which still continues improving the overhead of causal logging. Our experiments so far have shown that Clonos can be competitive (5-24% overhead in throughput and latency) with the current fault-tolerance mechanism of Flink which is industry-proven and serves billions of events every day, in multiple industries. At the moment, we are extending our work into reducing the overhead of causal logging through compressed data structures and extending Clonos' guarantees to low-latency exactly-once output.

Acknowledgements. This work has been partially funded by the H2020 project OpertusMundi No. 870228, and the ICAI "AI for Fintech Lab" project. Experiments were carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

REFERENCES

[1] [n.d.]. Apache Storm project. http://storm.apache.org/.

[2] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful functions as a service in action. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1890–1893.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[4] Lorenzo Alvisi. 1996. *Understanding the message logging paradigm for masking process crashes.* Technical Report. Cornell University.

[5] Lorenzo Alvisi, Bruce Hoppe, and Keith Marzullo. 1993. Nonblocking and orphan-free message logging protocols. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE, 145–154.

[6] Lorenzo Alvisi and Keith Marzullo. 1998. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering* 24, 2 (1998), 149–159.

[7] Apache Storm Trident. 2018. Tutorial. http://storm.apache.org/releases/1.0.6/Trident-tutorial.html. Available online July 2018.

[8] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. ACM, New York, NY, USA, 601–613.

[9] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. 2005. Fault-tolerance in the Borealis Distributed Stream Processing System. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. ACM, New York, NY, USA, 13–24.

[10] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink&Reg;: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1718–1729.

[11] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond Analytics: The Evolution of Stream Processing Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2651–2658. https://doi.org/10.1145/3318464.3383131

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink TM : Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[13] Badrish Chandramouli and Jonathan Goldstein. 2017. Shrink: Prescribing Resiliency Solutions for Streaming. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 505–516.

[14] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, and James F. Terwilliger. 2015. Trill: Engineering a Library for Diverse Analytics. *IEEE Data Eng. Bull.* 38 (2015), 51–60.

[15] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[16] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. 2003. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*. Asilomar, CA.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*.

[18] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2471–2486.

[19] Elmootazbellah Elnozahy. 1994. Manetho: Fault tolerance in distributed systems using rollback-recovery and process replication. (01 1994).

[20] Elmootazbellah N Elnozahy and Willy Zwaenepoel. 1992. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.* 5 (1992), 526–531.

[21] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Comput. Surv.* 34, 3 (Sept. 2002), 375–408.

[22] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. 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* (New York, New York, USA) *(SIGMOD '13)*. ACM, New York, NY, USA, 725–736.

[23] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 49–60.

[24] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. A Survey on the Evolution of Stream Processing Systems. arXiv:2008.00842 [cs.DC]

[25] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, Mehmet Doğan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Hazelcast Jet: Low-latency Stream Processing at the 99.99th Percentile. arXiv:2103.10169 [cs.DC]

[26] Thomas Heinze, Mariam Zia, Robert Krahn, Zbigniew Jerzak, and Christof Fetzer. 2015. An Adaptive Replication Scheme for Elastic Data Stream Processing Systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems* (Oslo, Norway) *(DEBS '15)*. Association for Computing Machinery, New York, NY, USA, 150–161.

[27] Jeong Hwang, Ying Xing, Ugur Cetintemel, and Stan Zdonik. 2007. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. 176–185.

[28] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. 2005. High-Availability Algorithms for Distributed Stream Processing. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, Washington, DC, USA, 779–790.

[29] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. 2005. High-availability algorithms for distributed stream processing. In *21st International Conference on Data Engineering (ICDE'05)*. 779–790. https://doi.org/10.1109/ICDE.2005.72

[30] Gabriela Jacques-Silva, Fang Zheng, Daniel Debrunner, Kun-Lung Wu, Victor Dogaru, Eric Johnson, Michael Spicer, and Ahmet Erdem Sariyüce. 2016. Consistent Regions: Guaranteed Tuple Processing in IBM Streams. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1341–1352.

[31] Asterios Katsifodimos and Marios Fragkoulis. 2019. Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications. In *EDBT*.

[32] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 239–250.

[33] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. 2008. Fault-Tolerant Stream Processing Using a Distributed, Replicated File System. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 574–585.

[34] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment* 1, 1 (2008), 274–288.

[35] Wei Lin, Haochuan Fan, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. STREAMSCOPE: Continuous Reliable Distributed Processing of Big Data Streams. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI'16)*. USENIX Association, Berkeley, CA, USA, 439–453.

[36] Matteo Migliavacca, David Eyers, Jean Bacon, Yiannis Papagiannis, Brian Shand, and Peter Pietzuch. 2010. SEEP: scalable and elastic event processing. In *Middleware'10 Posters and Demos Track*. ACM, 4.

[37] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1634–1645.

[38] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. ACM, New York, NY, USA, 1–14.

[39] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. 2004. Highly Available, Fault-Tolerant, Parallel Dataflows. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 827–838.

[40] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. 2004. Highly Available, Fault-tolerant, Parallel Dataflows. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. ACM, New York, NY, USA, 827–838.

[41] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible Time Management in Data Stream Systems. In *Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Paris, France) *(PODS '04)*. Association for Computing Machinery, New York, NY, USA, 263–274.

[42] L. Su and Y. Zhou. 2016. Tolerating correlated failures in Massively Parallel Stream Processing Engines. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 517–528.

[43] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@Twitter. In

Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (Snowbird, Utah, USA) *(SIGMOD '14)*. ACM, New York, NY, USA, 147–156.

[44] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *NEXMark—A Benchmark for Queries over Data Streams DRAFT*. Technical Report. Technical report, OGI School of Science & Engineering at OHSU, Septembers.

[45] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. Building a Replicated Logging System with Apache Kafka. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1654–1655.

[46] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage Stash: Fault Tolerance off the Critical Path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 338–352.