STREAMSCOPE: Continuous Reliable Distributed Processing of Big Data Streams

Wei Lin* *Microsoft*
Haochuan Fan* *Microsoft*
Zhengping Qian* *Microsoft Research*
Junwei Xu *Microsoft*
Sen Yang *Microsoft*
Jingren Zhou* *Microsoft*
Lidong Zhou *Microsoft Research*

Abstract

STREAMSCOPE (or STREAMS) is a reliable distributed stream computation engine that has been deployed in shared 20,000-server production clusters at Microsoft. STREAMS provides a continuous temporal stream model that allows users to express complex stream processing logic naturally and declaratively. STREAMS supports business-critical streaming applications that can process tens of billions (or tens of terabytes) of input events per day continuously with complex logic involving tens of temporal joins, aggregations, and sophisticated user-defined functions, while maintaining tens of terabytes in-memory computation states on thousands of machines.

STREAMS introduces two abstractions, rVertex and rStream, to manage the complexity in distributed stream computation systems. The abstractions allow efficient and flexible distributed execution and failure recovery, make it easy to reason about correctness even with failures, and facilitate the development, debugging, and deployment of complex multi-stage streaming applications.

1 Introduction

An emerging trend in big data processing is to extract timely insights from continuous big data streams with distributed computation running on a large cluster of machines. Examples of such data streams include those from sensors, mobile devices, and on-line social media such as Twitter and Facebook. Such stream computations process infinite sequences of input events and produce timely output events continuously. Events are often processed in multiple stages that are organized into a directed acyclic graph (DAG), where a vertex corresponds to the continuous and often stateful computation in a stage and an edge indicates an event stream flowing downstream from the producing vertex to the consuming vertex. In contrast to batch processing, also often modeled as a DAG [27], a defining characteristic of cloud-scale stream computation is its ability to process potentially *infinite* input events *continuously* with delays in seconds and minutes, rather than processing a static data set in hours and days. The continuous, transient, and latency-sensitive nature of stream computation makes it challenging to cope with failures and variations that are typical in a large-scale distributed system, and makes stream applications hard to develop, debug, and deploy.

This paper presents the design and implementation of STREAMSCOPE (or STREAMS), a cloud-scale reliable stream computation engine that has been deployed in shared production clusters, each containing over 20,000 commodity servers. STREAMS adopts a declarative language that supports a continuous stream computation model, extended with the ability to allow user-defined functions to customize stream computation at each step.

STREAMS has been designed for business-critical stream applications desiring a strong guarantee that each event is processed exactly once despite server failures and message losses. Failure recovery in cloud-scale stream computation is particularly challenging because of two types of dependencies: the dependency between upstream and downstream vertices, and the dependency introduced by vertex computation states. An upstream vertex failure affects downstream vertices directly, while the recovery of a downstream vertex would depend on the output events from the upstream vertices. Failure recovery of a vertex would require rebuilding the state before the vertex can continue processing new events. STREAMS therefore introduces two new abstractions, rVertex and rStream, to manage the complexity of cloud-scale stream computation by addressing the two types of dependencies through decoupling. rVertex models continuous computation on each vertex, introduces the notion of *snapshots* along the time dimension, and allows the computation to restart from a snapshot. rStream abstracts out the data and communication aspects of distributed stream computation, provides the illusion of reli-

*Now with Alibaba Group.

able and asynchronous communication channels, and decouples upstream and downstream vertices. Combined, rVertex and rStream offer well-defined semantics to replay computation and to rewind streams, as needed during failure recovery, thereby making it easy to develop different failure recovery strategies while ensuring correctness. The power of this abstraction also comes from the separation of its properties from its actual implementation that achieves those properties. That, for example, allows a different implementation of rStream specifically for development and debugging.

Our evaluation shows that STREAMS can support complex production streaming applications deployed on thousands of machines, processing tens of billions of events per day with complex computation logic to deliver business-critical results continuously despite unexpected failures and planned maintenance, while at the same time demonstrating good scalability and capability of achieving 10-millisecond latencies on simple applications.

STREAMS's key contributions are as follows. First, STREAMS shows that a cloud-scale distributed fault-tolerant stream engine can support a continuous stream computation model without having to converting a stream computation unnaturally to a series of mini-batch jobs [42]. Second, STREAMS introduces two new abstractions, rVertex and rStream, to simplify cloud-scale stream computation engines through separation of concerns, making it easy to understand and reason about correctness despite failures. The abstractions are also effective in addressing challenges on debugging and deployment of stream applications in STREAMS. Support for debugging and deployment is critical in practice from our experiences, but has not received sufficient attention. Finally, STREAMS is deployed in production and runs critical stream applications continuously on thousands of machines while coping well with failures and variations.

The rest of the paper is organized as follows. Section 2 describes STREAMS's continuous stream model and declarative language. Section 3 defines STREAMS's new abstractions: rVertex and rStream. Section 4 describes STREAMS's design and implementation in detail, followed by a discussion of several design choices in Section 5. Engineering experiences are the topic of Section 6. Section 7 presents the evaluation results of STREAMS in a production environment. We survey related work in Section 8 and conclude in Section 9.

2 Programming Model

In this section, we provide a high-level overview of the programming model, highlighting the key concepts including the data model and query language.

Continuous event streams. In STREAMS, data is represented as event streams, each describing a potentially infinite collection of events that changes over time. Each event stream has a well-defined *schema*. In addition, each event has a time interval $[V_s, V_e)$, describing the start and end time for which the event is valid.

```
AlertWithUserID =
    SELECT Alert.Name AS Name, Process.UserID AS UserID
    FROM Process INNER JOIN Alert
    ON Process.ProcessID == Alert.ProcessID;

CountAlerts =
    SELECT UserID, COUNT(*) AS AlertCount
    FROM AlertWithUserID
    GROUP BY UserID
    WITH HOPPING(5s, 5s);
```

Figure 1: A simplified STREAMS program.

Like other stream processing engines [9, 19], STREAMS supports *Current Time Increments* (`CTI`) events that assert the completeness of event delivery up to start time V_s of the `CTI` event; that is, there will be no events with a timestamp lower than V_s in the stream after this `CTI` event. Stream operators rely on `CTI` events to determine the current processing time in order to make progress and to retire obsolete state information.

Declarative query language. STREAMS provides a declarative language for users to program their applications without having to worry about distributed system details such as scalability or fault tolerance. Specifically, we extend the SCOPE [43] query language to support a full temporal relational algebra [15], extensible through user-defined functions, aggregators, and operators.

STREAMS supports a comprehensive set of relational operators including projection, filters, grouping, and joins, adapted for temporal semantics. For example, a temporal inner join applies to events with overlapping time intervals only. Windowing is another key concept in stream processing. A window specification defines time windows and consequently defines a subset of events in a window, to which aggregations can be applied. STREAMS supports several types of time-based windows, such as hopping, tumbling, and snapshot windows. For example, hopping windows are windows (of size `S`) that "jump" forward in time by a fixed size `H`: a new window of size `S` is created for every `H` units of time.

Example. Figure 1 shows an example program that performs continuous activity diagnosis on `Process` and `Alert` event streams. A STREAMS program consists of a sequence of declarative queries operating on event streams. `Process` events record information about every process and its associated user, while `Alert` events record information about every alert, including which process generated the alert. The program first joins the two streams to attach user information to alerts, and then calculates for each user the number of alerts every 5 seconds using a hopping window.

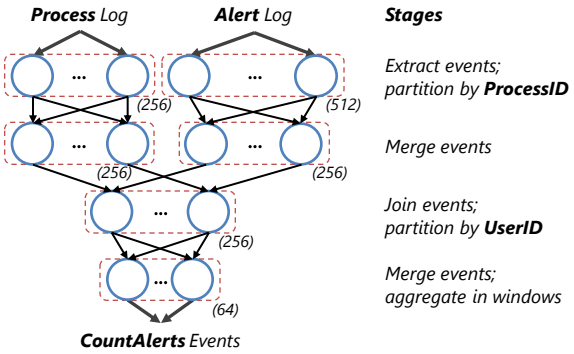


Figure 2: An execution DAG for the example in Figure 1.

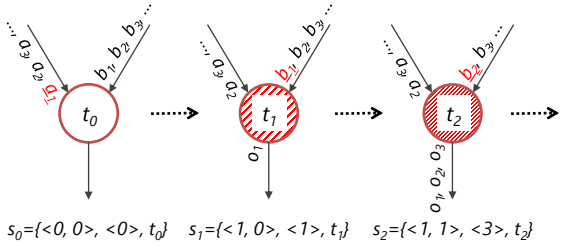


Figure 3: Vertex execution from snapshot s_0 to s_2.

3 STREAMS Abstractions

The execution of a STREAMS program can be modeled as a directed acyclic graph (DAG), where each *vertex* performs local computation on *input streams* from its in-edges and produces *output streams* as its out-edges. Each stream is modeled as an infinite sequence of events, each with a continuously incremented *sequence number*. Figure 2 shows an example DAG corresponding to Figure 1, where each stage of computation is partitioned into multiple vertices to execute in parallel. STREAMS determines the degree of parallelism for each stage (marked in parentheses) based on data rate and computation cost.

A vertex can maintain a local state. Its execution starts with its initial state and proceeds in steps. In each step, the vertex consumes the next events from its input streams, updates its state, and possibly produces events to its output streams. The execution of a vertex is tracked through a series of *snapshots*, where each snapshot is a triplet containing the current sequence numbers of its input streams, the current sequence numbers of its output streams, and its current state. Figure 3 illustrates the progression of a vertex execution from snapshot s_0, to s_1 (after processing a_1), and then to s_2 (after processing b_1). STREAMS introduces two abstractions rStream and rVertex to implement streams and vertices, respectively.

3.1 The rStream Abstraction

Rather than having vertices communicate directly through the network, STREAMS introduces an rStream abstraction to decouple upstream and downstream vertices with properties to facilitate failure recovery.

Conceptually, rStream reliably maintains a sequence of events with continuous and monotonically increasing sequence numbers, supporting multiple writers and readers. A writer issues `Write(seq,e)` to add event `e` with sequence number `seq`. rStream supports multiple writers mainly to allow two instances of the same vertex, which is useful when handling failures and stragglers via duplicated execution, as described in Section 4.

A reader can issue `Register(seq)` to indicate its interest in receiving events starting from sequence number `seq` and start reading from the stream using `ReadNext()`, which returns the next batch of events with their sequence numbers and advances the reading position accordingly. In the implementation, events can be pushed to a registered reader rather than pulled. With rStream each reader can proceed asynchronously from the same stream without synchronizing with other readers or writers. A reader can also rewind a stream by re-registering with an earlier sequence number (e.g., for failure recovery). rStream also supports `GarbageCollect(seq)` to indicate that all events less than sequence number `seq` will not be requested any more and therefore can be discarded. rStream maintains the following properties.

Uniqueness. There is a unique value associated with each sequence number. After the first write for each sequence number `seq` succeeds, any subsequent write that associates `seq` will be discarded.

Validity. If a `ReadNext()` returns an event `e` with sequence number `seq`, there must have been a `Write(seq,e)` that has returned successfully.

Reliability. If `write(seq,e)` succeeds, then, for any `ReadNext()` reaching position `seq`, eventually the read returns `(seq,e)`.

Uniqueness ensures consistency for each sequence number, Validity ensures correctness of the event value returned for each sequence number, while Reliability ensures that, all events written to the stream are always available to readers whenever requested. rStream could simply be implemented by a reliable pub/sub system backed by reliable and persistent store. But STREAMS adopts a more efficient implementation that avoids paying the latency cost of going to persistent and reliable store in the critical path, with the additional mechanism of reconstructing the requested events through recomputation [38, 42], as detailed in Section 4.

3.2 The rVertex Abstraction

The rVertex abstraction supports the following operations for a vertex. `Load(s)` starts an instance of the vertex at snapshot `s`. `Execute()` executes a step from the current snapshot. `GetSnapshot()` returns the current snap-

shot. A vertex can then be started with `Load`(s_0), where s_0 is the initial snapshot with an initial state and with all streams at starting positions. The vertex can then execute a series of `Execute`() operations, which read the input events, update the state, and produce output events. At any point, one can issue `GetSnapshot`() to retrieve and save the snapshot. When the vertex fails, it can be restarted with `Load`(`s`), where `s` is a saved snapshot.

Determinism. For a vertex with its given input streams, running `Execute`() on the same snapshot will always cause the vertex to transition into the same new snapshot and produce the same output events.

Determinism ensures correctness when replaying an execution during failure recovery. It implies that (i) the order in which the execution takes the next event from multiple input streams is deterministic; we explain how this order determinism is enforced naturally in STREAMS without introducing unnecessary delay in Section 4, and (ii) the execution of the processing logic is deterministic. Determinism greatly simplifies the reasoning of correctness in STREAMS and makes streaming applications easier to develop and debug. In Section 5, we discuss how to mask non-determinism when needed.

3.3 Failure Recovery

The rStream abstraction decouples upstream and downstream vertices to allow individual vertices to recover from failures separately. When a vertex fails, we can simply restart its execution by calling `Load`(`s`) from a most recently saved snapshot `s` to continue executing. The rVertex abstraction ensures that execution after recovery is the same as the continuation of the original execution as if no failures occurred. The rStream abstraction ensures that the restarted vertex is able to (re-)read the input streams. Section 4 describes how rVertex and rStream are implemented and how different failure recovery strategies can achieve different tradeoffs.

4 Architecture and Implementation

STREAMS is designed and implemented as a streaming extension of the SCOPE [43] batch-processing system. As a result, STREAMS heavily leverages the architecture, compiler, optimizer, and job manager in SCOPE, adapted or re-designed to support stream processing at scale. This approach expedites the development of STREAMS; the integration of batch and stream processing also offers significant benefits in practice, as elaborated in Section 6.

In STREAMS, a user programs a stream application declaratively as described in Section 2. The program is compiled into a streaming DAG for distributed execution as shown in Figure 4. To generate such a DAG, the STREAMS *compiler* performs the following steps: (1)

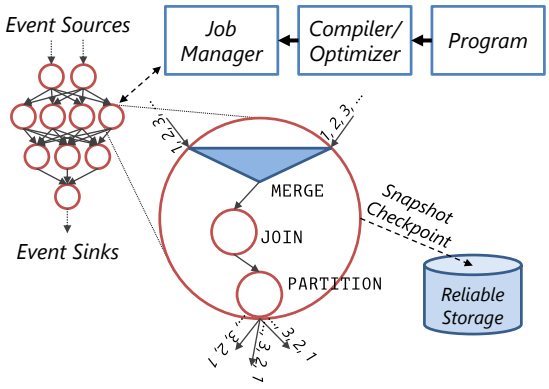


Figure 4: An overview of a STREAMS program.

the program is first converted into a *logical* plan (DAG) of STREAMS runtime operators, which include temporal joins, window aggregates, and user-defined functions; (2) the STREAMS *optimizer* then evaluates various plans, choosing the one with the lowest estimated cost based on available resource, data statistics such as the incoming rate, and an internal cost model; and (3) a *physical* plan (DAG) is finally created by mapping a logical vertex into an appropriate number of physical vertices for parallel execution and scaling, with code generated for each vertex to be deployed in a cluster and process input events continuously at runtime. We omit the details of these steps as they are similar to those for SCOPE [43].

The entire execution is orchestrated by a streaming *job manager* that is responsible for: (1) scheduling vertices to and establishing channels (edges) in the DAG among different machines; (2) monitoring progress and tracking snapshots; (3) providing fault tolerance by detecting failures/stragglers and initiating recovery actions. Unlike a batch-oriented job manager that schedules vertices at different times on demand, a streaming job manager schedules all vertices in a DAG at the beginning of job execution. To provide fault tolerance and to cope with runtime dynamics, rVertex and rStream are used to implement vertices and channels in a streaming DAG, working in coordination with the job manager.

4.1 Implementing rVertex

The key to implementing rVertex is to ensure Determinism as defined in Section 3, which requires both *function determinism* and *input determinism*. In STREAMS, all operators and user-defined functions must be deterministic. We also assume that the input streams for a job are deterministic, both in terms of order and event values. The only remaining input-related non-determinism is the ordering of events across multiple input streams. Because STREAMS uses `CTI` events (Section 2) as markers, we insert a special `MERGE` operator at the beginning of a

vertex that takes multiple input streams, which produces a deterministic order of events for subsequent processing in the vertex. It does so by waiting for the corresponding `CTI` events across input streams to show up, ordering them deterministically, and emitting them in that deterministic order. Because the processing logic of vertices tends to wait for the `CTI` events in the same way, this solution does not introduce additional noticeable delay.

StreamS labels events in each stream with consecutive monotonically increasing sequence numbers. A vertex uses sequence numbers to track the last consumed/produced events from all streams. At each step of the execution, a vertex consumes the next event(s) of the input streams, invokes `Execute()`, which might change its internal state, and generates new events into the output streams, thereby reaching a new snapshot. `GetSnapshot()` returns such a snapshot, which can be implemented by pausing the execution after a step or some copy-on-write data structures so that a consistent snapshot can be retrieved while running uninterrupted. `Load(c)` starts a vertex and loads `c` as the current snapshot before resuming execution. To be able to resume execution from a snapshot, a vertex can periodically create a *checkpoint* and store it reliably and persistently.

4.2 Implementing rStream

The rStream abstraction provides reliable channels that allow receivers to read from any written position. One straightforward implementation is for producing vertices to write events persistently and reliably into the underlying Cosmos distributed file system. Those synchronous writes introduce significant latencies in the critical path of stream processing. StreamS instead uses a hybrid scheme that moves those writes out of the critical path while providing the illusion of reliable channels: the events being written are first buffered in memory co-located with the producing vertex and can be transmitted directly to consuming vertices. The in-memory buffer is asynchronously flushed to Cosmos to survive server failures. Events that are only kept in memory might be lost on a failure, but can be recomputed when requested.

To be able to recompute lost events in case of failures, StreamS tracks how each event is computed, similar to dependency tracking in TimeStream [38] or lineage in D-Streams [42]. In particular, during execution, the job manager tracks vertex snapshots (through `GetSnapshot()`), which it can use later to infer how to reproduce events in the output streams. A vertex decides for itself when to capture a snapshot, save it (e.g., checkpointing to a reliable persistent store), and report progress to the job manager. For example, in Figure 5, vertex v_4 sends two updates to the job manager. The first update reports a snapshot $s_1 = \{\langle 2,7\rangle, \langle 12\rangle, t_1\}$, which

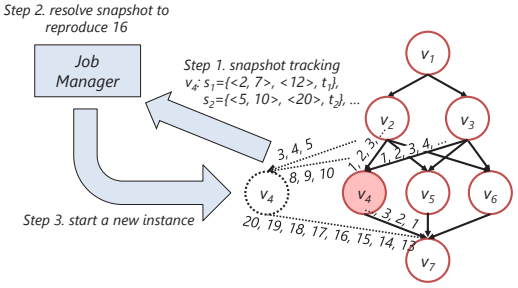


Figure 5: Snapshot tracking and recovery for rStream.

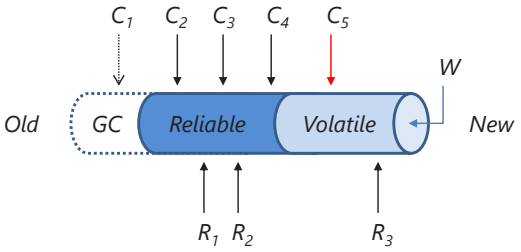


Figure 6: The StreamS implementation of rStream.

indicates that the vertex consumed up to event 2 in the first input stream and up to event 7 in the second, and produced output event 12, while at state t_1. The second update $s_2 = \{\langle 5,10\rangle, \langle 20\rangle, t_2\}$ reports that it has reached event 5 in the first input stream, event 10 in the second, while at state t_2. This tracking is completely transparent to users. Now if event 16 in the output stream needs to be recomputed, the job manager can simply scan the snapshots and find the highest output sequence number that is lower than 16, which is s_1 in this case. It then starts a new instance of the vertex, loads snapshot s_1, and continues executing until event 16 is produced. The execution of that new instance would require events 3-5 from the first input stream and events 8-10 in the second, which might trigger recomputation in upstream vertices if those events were no longer available. This process eventually terminates as the original input events are always assumed to be reliably persisted. Overall, such a design moves the flush to the reliable persistent store out of the critical path in normal execution, while at the same time reduces the number of events that need to be recomputed during failure recovery. While rStream is conceptually infinite, in a real implementation, garbage collection is necessary to remove obsolete events and related tracking information that are no longer needed for producing output events or for failure recovery, which we describe in Section 4.3.

Figure 6 illustrates rStream's implementation. In this example, there is one writer W (the upstream vertex) and three readers R_1, R_2, and R_3 (downstream vertices). The stream grows over time from left to right. The prefix of the stream (marked as GC) includes events that are

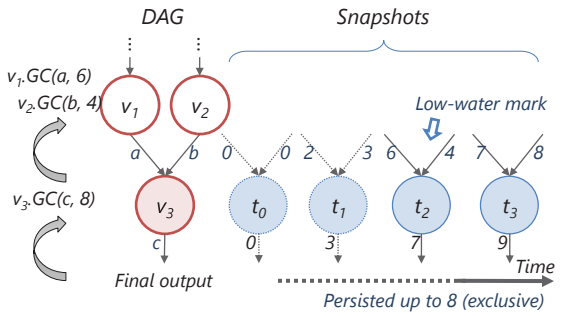


Figure 7: Garbage collection: a recursive view.

obsolete and can be garbage collected, followed by a sequence of events that have been reliably persisted. The tail of the stream (i.e., the most recent events) is *volatile* and could get lost when failures happen. Checkpoints can be created periodically (e.g., C_1, C_2, C_3, C_4, and C_5) for snapshots of the upstream vertex. When the volatile portion of the stream is lost due to failure, it can be recomputed from snapshot C_4. Events in the reliable portion can be served to R_1 and R_2 without recomputation. Replaying from a checkpoint in the reliable portion (e.g., C_4) fully recovers the transient portion; as a result, no further cascading recomputation is needed for recovery.

4.3 Garbage Collection

STREAMS persists checkpoints of snapshots, streams, and other tracking information reliably for failure recovery, and must determine when such information can be garbage collected. STREAMS maintains *low-water marks* for vertices and streams during the execution of a streaming application. For a stream, the low-water mark points to the lowest sequence number of the events that are needed; for a vertex, the low-water mark points to the lowest snapshot of the vertex that are needed.

For each vertex, snapshots are totally ordered by the vector of sequence numbers of its input and output streams. Because snapshots capture the linear progress of deterministic vertex execution, all sequence numbers only move forward. For example, consider a vertex with two input streams and one output stream, a snapshot s with a vector of sequence numbers at $(\langle 7,12\rangle,\langle 5\rangle)$ will be lower than a snapshot at $(\langle 7,20\rangle,\langle 8\rangle)$. There cannot be another snapshot at $(\langle 6,16\rangle,\langle 4\rangle)$.

Consider a vertex v with I as its set of input streams and O as its set of output streams. Vertex v maintains a low-water mark sequence number lm_o for each output stream $o \in O$, initialized to 0. Vertex v implements $\texttt{GC}(o,m)$ to perform garbage collection, indicating that any sequence number lower than m will no longer be requested by the downstream vertex consuming output stream o. For simplicity, we assume that each stream is consumed by a single downstream vertex, but it is straightforward to support the general case, where a stream is shared among multiple downstream vertices.

1. If $m \leq lm_o$, return; // no further GC needed.

2. Set lm_o to m. Let s be the highest checkpointed snapshot s satisfying the condition that the sequence number for output stream o in s is no higher than lm_o for every $o \in O$. Discard any snapshot lower than s.

3. For each input stream $i \in I$, let v_i be the upstream vertex producing input stream i and let s_i be the sequence number corresponding to input stream i in s, call $v_i.\texttt{GarbageCollect}\ (s_i)$ to discard events lower than s_i in input stream i. Recursively call $v_i.\texttt{GC}(i,s_i)$.

Intuitively, $\texttt{GC}(o,m)$ figures out which information is no longer needed if the downstream vertex (connected to the output stream o) will not request any events with a sequence number lower than m. It is called when the final output events are persisted or consumed, or when any output events in a stream is persisted reliably. Figure 7 shows an example of low-water marks. Although the algorithm is specified recursively, it can be implemented efficiently through a reverse topological order traversal.

4.4 Failure Recovery Strategies

STREAMS must recover from failures to keep streaming applications running. The rVertex and rStream abstractions decouple downstream vertices in a DAG from their upstream counterparts, making it easier to reason about and deal with runtime failures. In addition, they abstract away underlying implementation details, and allow them to share a common mechanism for fault tolerance.

Different failure recovery strategies can be developed; the choice can be decided by a combination of factors: normal-case cost (in terms of resources required), normal-case overhead (in terms of latency), recovery cost (in terms of resources required for recovery), and recovery time. We highlight three strategies that represent different tradeoffs that are appropriate in different scenarios. With rStream and rVertex, each vertex can recover from failures independently. As a result, those strategies can be applied at the vertex granularity and even different vertices in the same job could potentially use different strategies due to their different characteristics.

Checkpoint-based recovery. In this strategy, a vertex checkpoints its snapshot periodically into a reliable persistent store. When the vertex fails, it will load the most recent checkpoint and resume execution. A straightforward implementation of checkpointing introduces overhead in normal execution that is not ideal for vertices that maintain a large internal state. Advanced checkpointing techniques [33, 34] often require specific data structures, which introduces complexity and overhead.

Replay-based recovery. Quite often stream computation is either stateless or has a *short-term memory* due to its use of window operators; that is, its current internal state depends only on the events in the most recent window of a certain duration (say the last 5 minutes). In those cases, a vertex can get away with not explicitly checkpointing state, and instead reloading that window of events to rebuild state from an initial one. While this is a special case, it is common enough to be useful. Leveraging this property, StreamS can simply track the sequence numbers of the input/output streams without having to store the local states of a vertex. This strategy might need to reload a possibly large window of input events during recovery, but it avoids the upfront cost of checkpointing in the normal case.

This strategy has a subtle implication on garbage collection. Instead of loading a state in a snapshot, a vertex must recover it from earlier events in the input streams. Those events must be retained along with the snapshot.

Replication-based recovery. Yet another strategy is to have multiple instances of the same vertex run at the same time: they can be connected to the same input streams and output streams. Our rStream implementation allows multiple readers and writers, deduplicating automatically based on sequence numbers. The Determinism property of rVertex also makes replication a viable approach because those instances will behave consistently. With replication, a vertex can have instances take checkpoints in turn without affecting latency observed by readers because other instances are running at a normal pace. When one instance fails, it can also get the current snapshot from another instance directly to speed up recovery. All those benefits come at the cost of having multiple instances running at the same time.

5 Discussion

StreamS makes different choices from existing distributed stream processing engines on stream model, non-determinism, and out-of-order event processing.

Mini-batch stream processing with RDD. Instead of supporting a continuous stream model, D-Streams [42] models a stream computation as a series of mini-batch computations in small time intervals and leverages immutable RDDs [41] for failure recovery. Modeling a stream computation as a series of mini-batches could be cumbersome because many stream operators, such as windowing, joins, and aggregations, maintain states to process event streams efficiently. Breaking such operations into separate mini-batch computation requires rebuilding computation state from the previous batches before processing new events in the current batch. A good example is the inner join in Figure 1. The join operator needs to track all the events that might generate matching results for the current or future batches in efficient data structures and can only retire them on `CTI` events. Regardless of how mini-batches are generated, such a join state is potentially big for complex join types and needs to be rebuilt in each batch or passed on between consecutive mini-batches. Furthermore, D-Streams unnecessarily couples low latency and fault tolerance: a mini-batch defines the granularity at which vertex computation is triggered and therefore dictates latency, while an immutable RDD, mainly for failure recovery, is created for each mini-batch. The low-latency requirements demand small batch sizes even though there is no need to enable failure recovery at that granularity.

Non-determinism. Determinism is required in rVertex for correctness and also makes debugging easier. Non-determinism could introduce inconsistency when a vertex re-executes during failure recovery. Non-determinism might cause re-execution to deviate from the initial execution and lead to a situation where downstream vertices use two inconsistent versions of the output event streams from this vertex. StreamS can be extended to support non-determinism, but at a cost.

One way to avoid inconsistency due to non-determinism is to make sure that any output events produced by a vertex do not need to be recomputed. This can be achieved, for example, by checkpointing the snapshot to a reliable and persistent store before making the output events visible to downstream vertices. This is in essence the choice that MillWheel [7] makes in its design. This proposal introduces significant overhead because the expensive checkpointing is on the critical path. An alternative approach is to log non-deterministic decisions during execution for faithful replay [10, 22, 23, 32, 35]. Logging non-deterministic decisions is often less costly than checkpointing snapshots, but this approach requires that all sources of non-determinism be identified, appropriately logged, and replayed. StreamS does not support such mechanisms in the current implementation.

Out-of-order event ordering. Events could arrive in an order that does not correspond to their application timestamps, for example, when the events come from multiple sources. To allow out-of-order event processing, systems such as Storm [3] and MillWheel [7] assign a unique but unordered `ID` to each event. A downstream vertex sends `ACK`s with those `ID`s to an upstream vertex to track progress and handle failures. StreamS decouples the logical order of events from their physical delivery and consumption. It borrows the idea of CTI events as discussed in Section 2 from stream databases to achieve out-of-order event processing at the language and operator level. At the system level, StreamS assigns unique and ordered sequence numbers to events, making it easy to track progress and handle failures, while avoiding explicit `ACK`s that could incur performance overhead.

6 Production Experiences

STREAMS has been deployed in production. This section highlights our experiences with developing STREAMS and with supporting the life cycle of streaming applications, from development, debugging, to deployment.

From batch to streaming. STREAMS has been developed as an extension to an existing large-scale batch processing system and benefited greatly from reusing the existing components, such as the compiler, optimizer, and DAG/job manager, with adaptation and changes to support streaming. For example, the compiler is extended to handle streaming operators and the optimizer has a revised cost function to evaluate streaming plans.

Quite a few streaming applications were migrated from recurring batch jobs to achieve better efficiency and low latency. STREAMS provides supports for such migration in the compiler and allows the use of a batch version to validate the results of a streaming counterpart.

Scaling and robustness to fluctuation. STREAMS creates a physical plan to handle scaling based on estimated peak input rates and operator costs, ensuring that a sufficient number of vertices in each stage execute in parallel to handle the peak load. We find STREAMS's design robust to fluctuations caused by load spikes or server failures thanks to the decoupling between vertices using rStream. When one vertex falls behind temporarily, the input events to the vertex are queued in essentially an infinite buffer in the underlying distributed storage system. The queuing also allows effective event batching to allow the vertex to catch up quickly. When the peak load increases over time, STREAMS provides the support to move to a new configuration with increased degrees of paralleliem, without any interruption. This is done by initiating new vertices with derived states from checkpoints in the current job, and retires the corresponding vertices when the new ones catch up. The flexibility of rStream makes it easy to support such transitions. We decided not to support dynamic reconfiguration [38] as the additional complexity was not justified.

Distributed streaming made easy. In STREAMS, the declarative programming language and the stream data model makes it easy to program a streaming application without worrying about the distributed system details. STREAMS extends the simplicity to development and debugging via a different instantiation of rStream and a different scheduling policy in the job manager. Specifically, STREAMS introduces an *off-line* mode, where finite datasets, usually persistently stored, can be read to simulate on-line event streams, through a special instantiation of rStream. The job manager also uses a special *off-line* mode that favors ease of debugging over latency by executing one vertex at a time, instead of running all vertices concurrently, thereby significantly reducing the required resources. The off-line mode is completely *transparent* to the user code, which behaves the same way as in the on-line version except for latency.

Traveling back in time. A streaming application typically progresses forward in time, but we have encountered cases where traveling back in time is needed. For example, a user might request to re-examine a segment of execution in the past in response to an audit request. As a result of the investigation, the user needs to apply adjustments to the past results because the learning algorithm used is imperfect and needs correction in this particular case. To improve the algorithm, the user further conducts experiments with new algorithms and compares them with the current one. To handle such requirements, we maintain all past checkpoints and input channels in a global *repository* that implements a retention policy. Our rVertex and rStream abstractions support time travel to the past as is the case for failure recovery.

Continuous operation during system maintenance. Cluster-wide maintenance and updates, e.g., to apply patches, occur regularly in data centers. For batch jobs, the maintenance can be done systematically by not assigning new tasks to the ones to be patched and waiting for the existing tasks to finish on those machines. This is unfortunately not sufficient for streaming applications as they run continuously. Instead, STREAMS leverages duplicate execution (as used to handle stragglers) to minimize the effect: after receiving a notification that certain machines are scheduled for maintenance, the job manager replicates the execution of each affected vertex and schedules another instance in a different safe machine. Once the new instance catches up in terms of events processing, the job manager can safely kill the affected vertex and allow maintenance to proceed.

Straggler handling. Stragglers are vertices that make progress at a slower rate than other vertices in the same stage. Preventing and mitigating stragglers is particularly important for streaming applications where stragglers can have more severe and long-lasting performance impact. First, STREAMS continuously monitors machine health and only considers healthy machines for running streaming vertices. This significantly reduces the likelihood of introducing stragglers at runtime. Second, for each vertex, STREAMS tracks its resulting `CTI` events, normalized by the number of its input events, to estimate roughly its progress. If one vertex has a processing speed that is significantly slower than the others in the same stage that execute the same query logic, a duplicate copy is started from the most recent checkpoint. They execute in parallel until either one catches up with the others, at which point the slower one is killed.

A streaming application might encounter anomalies during its execution; for example, when hitting unexpected input events. We have encountered cases where

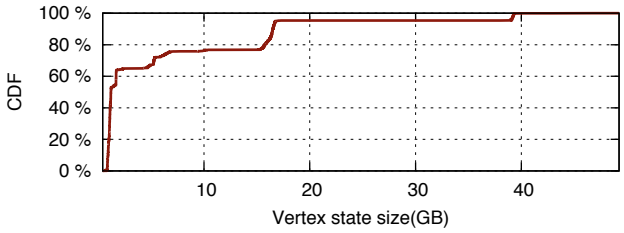


Figure 8: Distribution of vertex state sizes.

certain rare input events take a lot of time to process. The vertex hitting such an event is often considered a straggler, but such a straggler cannot be fixed by duplicate execution as the computation is always expensive. We extend STREAMS with an *alert* mechanism to supply users with various alerts, including event processing speeds and on-line statistics, and provide a flexible mechanism that allows users to specify a *filter* to weed out such events to keep the streaming application running smoothly (before a new solution is ready to be deployed). To ensure determinism, before a filter is applied, STREAMS creates a checkpoint for the vertex, flushes the volatile part of the streams, and records the filter.

7 Evaluation

STREAMS has been deployed since late 2014 in shared 20,000-server production clusters, running concurrently a few hundred thousand jobs daily, including a variety of batch, interactive, machine-learning, and streaming applications. Our evaluation starts with an in-depth study of a large business-critical production streaming application. Next, we perform extensive experiments using three simple streaming applications to demonstrate the scalability and performance with STREAMS, as well as the tradeoff between latency and throughput. Finally, we evaluate different failure recovery strategies. All the experiments are carried out in our shared production environment to perform real and practical evaluation.

7.1 A Production Streaming Application

For our evaluation, we study a production streaming application that supports a core on-line advertisement service. The application detects fraud clicks of on-line transactions in near-real time and refunds affected customers accordingly. Because the application is related to accounting, strong guarantees are needed. Latency is important for the application because lower latency allows customers to adjust their selling strategies more quickly, leading to higher revenues for the service. The previous implementation as a batch job introduced a latency of

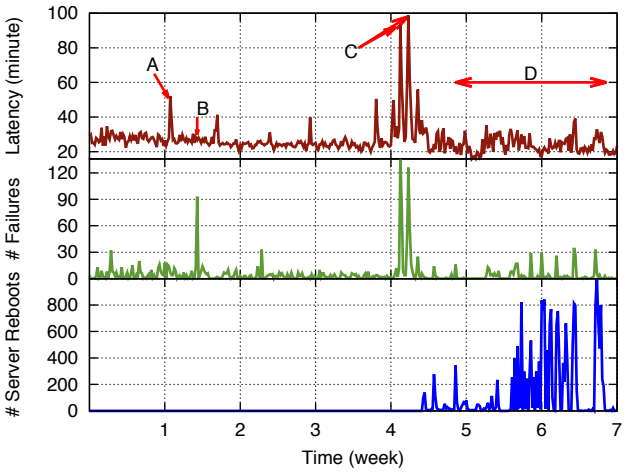


Figure 9: Application performance, failures, and server reboots over a 7-week period.

around 6 hours: it had to wait for the data to accumulate (every 3 hours) and to rebuild the state every time it ran.

The application has a complex processing logic that involves a total of 48 stages, containing 18 joins of 5 different types (specifically, left semi, left anti semi, left outer, inner, and clip [9]). During the period of evaluation, the application executes on 3,220 vertices, processing tens of billions of input events (around 9.5 TB in size) and resulting in around 180 TB of I/O per day. Figure 8 shows the distribution of in-memory vertex state sizes in the application. There are about 25% "heavy" vertices that maintain a huge in-memory state because the application extracts millions of features from raw events and maintains a large number of complicated statistics in memory before feeding them into a sophisticated on-line prediction engine for fraud detection. The aggregated in-memory state of all the vertices is around 21.3 TB.

7.2 STREAMS in Production

In a shared production environment, failures, variations, and system maintenance are the norm. We cover several key aspects of the production streaming application, including performance, failures, variations, and stragglers.

Performance and failure impact. Figure 9 shows the end-to-end latency of this application over a 7-week period (top figure), along with the number of server failures (middle figure), and the number of servers brought down for planned maintenance (bottom figure), all aligned on time. We observed random server failures from time to time, impacting the application latency in various ways. We also experienced a major planned system maintenance that systematically rebooted machines.

We highlight four interesting periods, labeled *A*, *B*, *C*, and *D*. In case *A*, although the number of failures was not

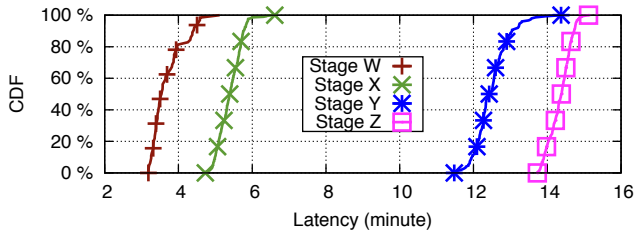


Figure 10: Distribution of vertex latency in four representative stages.

high, some of the failed vertices held a relatively large in-memory state and took a long time to recover, thereby leading to a significant latency spike. In case *B*, however, a majority of failures occurred to vertices with a relatively small state. The recovery was therefore fast and its latency impact was hardly visible. Case *C* corresponds to a "live site" triggered by an *unplanned* mis-configuration that caused a significant number of machines to reboot. The issue led to a significant latency spike, but the application survived this massive failure. In Case *D*, a *planned* system maintenance rebooted machines in batches to apply OS patches and software upgrades. The entire application had to be migrated to run on a different set of machines. STREAMS used duplicate execution for each affected vertex to migrate gracefully.

The average end-to-end latency for the application is around 20 minutes. The original timestamps of the input events are included in the final output events and are used to compute end-to-end latency. The input delay, which is the interval between when events are generated/timestamped and when they appear in the input streams of the application, is also included. Window aggregations semantically introduce delay in latency and are the dominant factor in the overall latency for this application.

Variations. Performance variations are common in distributed computation, even among vertices in the same stage. Figure 10 shows the latency distribution of all the vertices in four representative stages, respectively. Stage *W* is responsible for extracting input events from raw event logs: the latency observed at that stage is mostly due to input delay. The variations observed in that stage are also consistently observed in later stages. Stage *X* contains window aggregations, which intrinsically introduce delays that are comparable to those of the window sizes to the downstream stage *Y*. Stage *Z* represents the application's final computation stage. Variations are the result of various factors: load fluctuation and interference on servers, or changing data characteristics and their impact on computation complexity and efficiency.

Concurrent channels. Interestingly, noticeable performance variations are observed on vertices that process the same data from the same upstream vertex. We examine a vertex whose output events are broadcast to 150

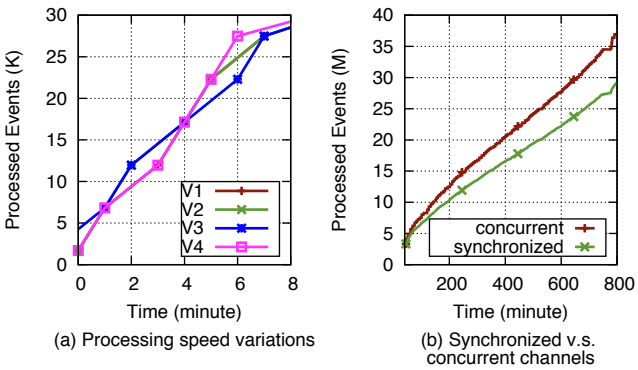


Figure 11: Benefits of concurrent channels.

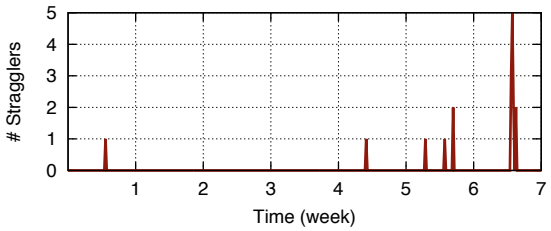


Figure 12: Stragglers in production.

downstream vertices in the application we study. Figure 11(a) shows a detailed 8-minute view of processing speed variations on four selected vertices. A difference of several thousand events in processing speeds shows up from time to time, mainly due to computation variations in individual vertices: one vertex might outperform others for a period of time and then lag behind in the next. This observation argues against a naive *synchronized* design (e.g., using TCP directly) that forces all downstream vertices to proceed in lock steps, causing the slowest vertex to dictate in each step. STREAMS employs a *concurrent* design, allowing individual downstream vertices to advance at different speeds. Figure 11(b) compares the projected progress of such a synchronized design with the actual execution that uses a concurrent channel. The performance of using concurrent channels noticeably outperforms that of using synchronized ones.

Stragglers. Stragglers do appear in production even with mechanisms to prevent them. A straggler cannot recover by itself, unlike performance variation, and actions such as duplicate execution might be needed to resolve it. Figure 12 shows the number of stragglers we detected and successfully recovered during the 7-week period. We are conservative in classifying a vertex as a straggler because we observe that in most cases a vertex that falls behind temporarily can catch up by processing at large batches. The detected ones are those with persistent issues.

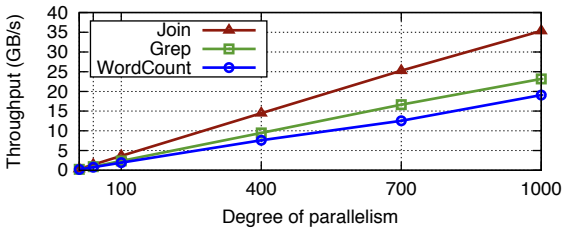


Figure 13: Scalability.

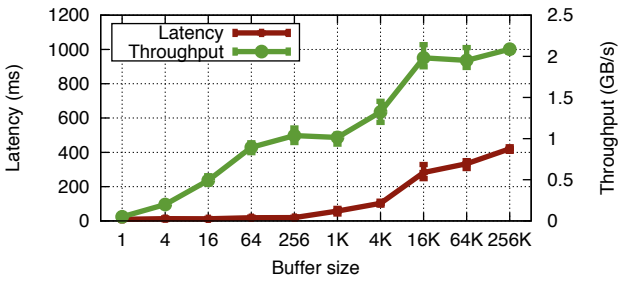
7.3 Scalability

To evaluate scalability and study performance tradeoff, we run three simple streaming applications in production. (1) **Grep** scans input events (strings) for a matching pattern; (2) **WordCount** counts the number of words in an input stream over 1-minute hopping windows, and (3) **Join** joins two input streams to return matching pairs. Each event in the first input stream has a 2-min window $[t, t+2]$, while a matching event with the same join key appears in the second stream in a 1-min window $[t+0.5, t+1.5]$, so that each event appears in the join result, allowing the application to produce a steady output stream. In all cases, each event is 100 bytes.

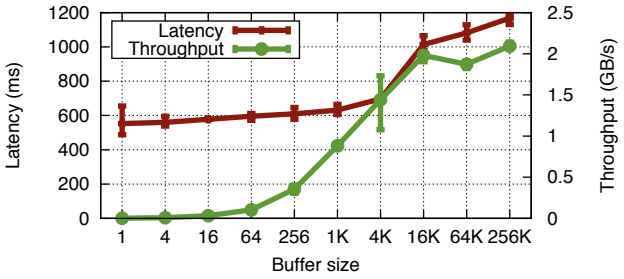
To evaluate scalability, we run each application with different numbers of vertices (degrees of parallelism) up to 1,000. We constrain each vertex to use one CPU core and limit I/O bandwidth to avoid significantly impacting production activities. Figure 13 reports the maximum throughput that STREAMS can sustain under a 1-second latency bound for each application with different numbers of vertices. STREAMS scales linearly to 1000 vertices, achieving a throughput of up to 35 GB/s. We also repeat the same experiments on a small dedicated 20-machine test cluster without any vertex resource constraint. STREAMS is able to saturate the network and the maximum throughput is bounded by network bandwidth.

7.4 Tradeoff: Latency vs. Throughput

We further study the tradeoff between latency and throughput by varying event buffer size using Grep with 100 vertices. We choose Grep because it has no window or join constructs that could introduce application-level delays. We repeat each experiment 3 times and report the average, minimum, and maximum values. As shown in Figure 14(a), STREAMS achieves a latency around 10 msec using a small buffer size at the cost of lower throughput. As the buffer size grows, the throughput improves but the latency also increases. STREAMS achieves a stable maximum throughout when buffering every 16K events, where the latency is around 280 msec. Our default production setting triggers computation either when accumulated events fill a 2MB buffer or every



(a) STREAMS (Asynchronous writes)



(b) Synchronous writes

Figure 14: Latency and throughput tradeoff using Grep with 100 vertices.

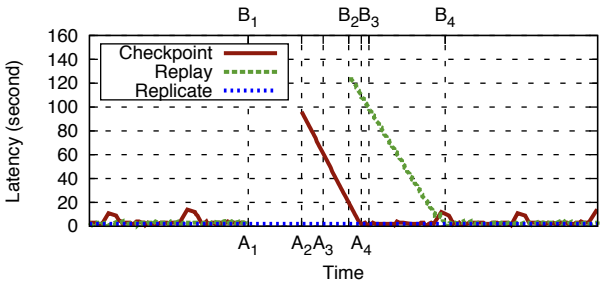


Figure 15: Comparing failure recovery strategies.

500 msec, but it is configurable for each application.

In STREAMS, events are first buffered in memory before *asynchronously* flushed to a reliable persistent store. To compare, we repeat the same Grep experiment by *synchronously* storing every event persistently, as done in MillWheel [7]. Figure 14(b) shows a similar tradeoff. However, its latency is always worse than that of STREAMS, while its throughput is worse when the buffer size is smaller than 1KB and is comparable otherwise.

7.5 Failure Recovery Strategies

We compare three failure recovery strategies using Join in Figure 15. The experiment is conducted in a production environment, where we inject a vertex failure manually, apply different recovery strategies, and observe the latency impact during failure and recovery. We align the time lines of the executions for ease of comparison.

The replication-based strategy has no impact on the latency because it always has at least two instances of the same vertex running. For the checkpoint-based strategy, each checkpointing introduces a small latency spike. After a failure, a new instance of the failed vertex reloads the latest checkpoint and continues executing. From A_1 to A_2, the checkpointed snapshot is reloaded. From A_2 to A_3, the vertex re-produces events that were already generated before failure. Those are discarded. After A_3, the vertex starts to produce new output events. The latency is high at this point because input events have been buffered during failure/recovery. The vertex catches up at A_4. Replay-based recovery does not have checkpointing overhead. The last snapshot is reconstructed by replaying the input events, which corresponds to the period between B_1 to B_2. There is a longer delay in replay-based recovery because the state in checkpoint is more condensed than the input events (a common case). Once the state is reconstructed, it follows the same steps as in the checkpoint-based recovery: it reproduces some duplicate output from B_2 to B_3 and then catches up at B_4. The actual shape of the curves depends on many factors, such as the sizes of the states, the number of events that must be replayed, the replay speed, and the catch-up speed.

As a rule of thumb, the checkpoint-based strategy is preferable if the checkpointing cost is low. The replay-based strategy is favored if the checkpointing cost is high, but the replay cost is comparable to that of recovery from a checkpoint. In our production application, 25% of the vertices use replay-based recovery (manually configured) to avoid the normal-case latency penalty, while the remaining use checkpoint-based recovery for fast fail-over. Replication is used for duplicate execution to handle stragglers or to enable migration.

8 Related Work

The key concepts in StreamS, such as declarative SQL-like language, temporal relational algebra, compilation and optimization to streaming DAG, and scheduling, have been inherited from the design of stream database systems [6, 13, 20, 5, 19], which extensively studied stream semantics [29, 18, 15, 28] and distributed processing [11, 25, 17, 14, 26]. StreamS's novelty is in the new rStream and rVertex abstractions designed for high scalability and fault tolerance through decoupling, representing a different and increasingly important design point that favors scalability at the expense of somewhat loosened latency requirements. MapReduce Online [21], S4 [37], and Storm [3, 31] extend a DAG model in batch processing systems like Hadoop [1] to streaming.

StreamS is designed to achieve the exactly-once semantics despite failures; such strong consistency is required by many production streaming applications and makes it easy to reason about correctness. Other systems such as Trident [4] (over Storm), MillWheel [7, 8], TimeStream [38], D-Streams [42], and Samza [2] also embrace strong consistency, but make different design choices. The abstractions in StreamS separate the key requirements of fault tolerant streaming processing from the different approaches in satisfying those requirements. For example, state management and tracking in Trident, MillWheel, and Samza can be considered a way to realize rVertex. TimeStream's dependency tracking and D-Streams' lineage tracking can be used to implement rStream with on-demand recomputation, while Kafka [30]-based channel implementation in Samza implements rStream with all events persisted reliably. StreamS's rStream implementation moves the cost of reliable persistence out of the critical path (unlike MillWheel [7] and Samza [2]), while keeping the probability of on-demand recomputation low and avoiding cascading recovery in practice. Other noteworthy technical differences, such as continuous vs. mini-batch models, non-determinism, and out-of-order event processing, have been discussed in Section 5.

Several other systems focus on other design dimensions. For example, Photon [12] and JetStream [39] address the geo-distribution aspect of streaming to achieve consistency and efficiency over a wide area network. Naiad [36] and Flink [16] handle dataflows with cycles for incremental computation. SEEP [24] and ChronoStream [40] address the resource elasticity for streaming by dynamically adjusting the degree of parallelism. Heron [31] improves on Storm to run in shared production cluster efficiently and introduces backpressure.

9 Conclusion

StreamS takes a principled approach to distributed fault-tolerant cloud scale stream computation with new abstractions rVertex and rStream. Its implementation and deployment in production not only provide the insights that validate the design choices, but also offer valuable engineering experiences that are key to the success of such a cloud scale stream computation system.

10 Acknowledgments

We would like to thank the anomymous reviewers, as well as our shepherd Hari Balakrishnan, for their valuable comments and suggestions. We are grateful to Michael Levin, Andrew Baumann, Geoff Voelker, and Jay Lorch for providing insightful feedback on our early draft. We would also like to thank Microsoft Big Data team members for their support and collaboration.

References

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

[2] Apache Samza. http://samza.apache.org/.

[3] Apache Storm. http://storm.incubator.apache.org/.

[4] Trident. https://storm.apache.org/documentation/trident-tutorial.html.

[5] ABADI, D. J., AHMAD, Y., BALAZINSKA, M., ÇETINTEMEL, U., CHERNIACK, M., HWANG, J., LINDNER, W., MASKEY, A., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. B. The design of the Borealis stream processing engine. In *CIDR* (2005), pp. 277–289.

[6] ABADI, D. J., CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. B. Aurora: A new model and architecture for data stream management. *VLDB J. 12*, 2 (2003), 120–139.

[7] AKIDAU, T., BALIKOV, A., BEKIROGLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., AND WHITTLE, S. MillWheel: Fault-tolerant stream processing at Internet scale. *PVLDB 6*, 11 (2013), 1033–1044.

[8] AKIDAU, T., BRADSHAW, R., CHAMBERS, C., CHERNYAK, S., FERNÁNDEZ-MOCTEZUMA, R. J., LAX, R., MCVEETY, S., MILLS, D., PERRY, F., SCHMIDT, E., 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 (2015), 1792–1803.

[9] ALI, M. H., CHANDRAMOULI, B., GOLDSTEIN, J., AND SCHINDLAUER, R. The extensibility framework in Microsoft StreamInsight. In *ICDE* (2011), pp. 1242–1253.

[10] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009* (2009), pp. 193–206.

[11] AMINI, L., ANDRADE, H., BHAGWAN, R., ESKESEN, F., KING, R., SELO, P., PARK, Y., AND VENKATRAMANI, C. SPC: A distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms* (2006), ACM, pp. 27–37.

[12] ANANTHANARAYANAN, R., BASKER, V., DAS, S., GUPTA, A., JIANG, H., QIU, T., REZNICHENKO, A., RYABKOV, D., SINGH, M., AND VENKATARAMAN, S. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), pp. 577–588.

[13] ARASU, A., BABCOCK, B., BABU, S., DATAR, M., ITO, K., NISHIZAWA, I., ROSENSTEIN, J., AND WIDOM, J. STREAM: The Stanford stream data manager. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003* (2003), p. 665.

[14] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S., AND STONEBRAKER, M. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *ACM SIGMOD Conf.* (Baltimore, MD, June 2005).

[15] BARGA, R. S., GOLDSTEIN, J., ALI, M. H., AND HONG, M. Consistent streaming through time: A vision for event stream processing. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings* (2007), pp. 363–374.

[16] CARBONE, P., FÓRA, G., EWEN, S., HARIDI, S., AND TZOUMAS, K. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603* (2015).

[17] CETINTEMEL, U. The Aurora and Medusa projects. *Data Engineering 51*, 3 (2003).

[18] CHAKRAVARTHY, S., KRISHNAPRASAD, V., ANWAR, E., AND KIM, S. Composite events for active databases: Semantics, contexts and detection. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile* (1994), pp. 606–617.

[19] CHANDRAMOULI, B., GOLDSTEIN, J., BARNETT, M., DELINE, R., PLATT, J. C., TERWILLIGER, J. F., AND WERNSING, J. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB 8*, 4 (2014), 401–412.

[20] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., REISS, F., AND SHAH, M. A. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), ACM, pp. 668–668.

[21] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. Mapreduce online. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA* (2010), pp. 313–328.

[22] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay.

[23] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008, Seattle, WA, USA, March 5-7, 2008* (2008), pp. 121–130.

[24] FERNANDEZ, R. C., MIGLIAVACCA, M., KALYVIANAKI, E., AND PIETZUCH, P. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), pp. 725–736.

[25] FRANKLIN, M. J., JEFFERY, S. R., KRISHNAMURTHY, S., REISS, F., RIZVI, S., WU, E., COOPER, O., EDAKKUNNI, A., AND HONG, W. Design considerations for high fan-in systems: The HiFi approach. In *CIDR* (2005), pp. 290–304.

[26] HWANG, J.-H., BALAZINSKA, M., RASIN, A., CETINTEMEL, U., STONEBRAKER, M., AND ZDONIK, S. High-Availability Algorithms for Distributed Stream Processing. In *The 21st International Conference on Data Engineering (ICDE 2005)* (Tokyo, Japan, April 2005).

[27] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007* (2007), pp. 59–72.

[28] JAIN, N., MISHRA, S., SRINIVASAN, A., GEHRKE, J., WIDOM, J., BALAKRISHNAN, H., ÇETINTEMEL, U., CHERNIACK, M., TIBBETTS, R., AND ZDONIK, S. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1379–1390.

[29] JENSEN, C. S., AND SNODGRASS, R. T. Temporal specialization. In *Proceedings of the Eighth International Conference on Data Engineering, February 3-7, 1992, Tempe, Arizona* (1992), pp. 594–603.

[30] KREPS, J., NARKHEDE, N., AND RAO, J. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece* (2011).

[31] KULKARNI, S., BHAGAT, N., FU, M., KEDIGEHALLI, V., KELLOGG, C., MITTAL, S., PATEL, J. M., RAMASAMY, K., AND TANEJA, S. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 239–250.

[32] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *SIGMETRICS 2010, Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, New York, USA, 14-18 June 2010* (2010), pp. 155–166.

[33] LI, K., NAUGHTON, J. F., AND PLANK, J. S. *Real-time, concurrent checkpoint for parallel programs*, vol. 25. ACM, 1990.

[34] LI, K., NAUGHTON, J. F., AND PLANK, J. S. Low-latency, concurrent checkpointing for parallel programs. *Parallel and Distributed Systems, IEEE Transactions on 5*, 8 (1994), 874–879.

[35] MONTESINOS, P., HICKS, M., KING, S. T., AND TORRELLAS, J. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009* (2009), pp. 73–84.

[36] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *ACM SIGOPS*

24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013 (2013), pp. 439–455.

[37] NEUMEYER, L., ROBBINS, B., NAIR, A., AND KESARI, A. S4: Distributed stream computing platform. In *ICDMW 2010, The 10th IEEE International Conference on Data Mining Workshops, Sydney, Australia, 13 December 2010* (2010), pp. 170–177.

[38] QIAN, Z., HE, Y., SU, C., WU, Z., ZHU, H., ZHANG, T., ZHOU, L., YU, Y., AND ZHANG, Z. TimeStream: Reliable stream computation in the cloud. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013* (2013), pp. 1–14.

[39] RABKIN, A., ARYE, M., SEN, S., PAI, V. S., AND FREEDMAN, M. J. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), pp. 275–288.

[40] WU, Y., AND TAN, K.-L. ChronoStream: Elastic stateful stream computation in the cloud. In *Data Engineering (ICDE), 2015 IEEE 31th International Conference on* (2015), IEEE.

[41] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012* (2012), pp. 15–28.

[42] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized Streams: Fault-tolerant streaming computation at scale. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 423–438.

[43] ZHOU, J., BRUNO, N., WU, M.-C., LARSON, P.-Å., CHAIKEN, R., AND SHAKIB, D. SCOPE: Parallel databases meet MapReduce. *VLDB J. 21*, 5 (2012), 611–636.