Falkirk Wheel: Rollback Recovery for Dataflow Systems

Ionel Gog
UC Berkeley
ionel@berkeley.edu

Michael Isard
Google Research
misard@google.com

Martín Abadi
Google Research
abadi@google.com

Abstract

Data processing applications often combine computations with disparate fault-tolerance requirements. For example, batch computations prioritize throughput over recovery latency, and can tolerate recovery delays of up to several minutes, while streaming computations expect recovery latencies of at most a few seconds. However, state-of-the-art data systems each offer a single fault-tolerance regime, so complex applications either: (*i*) suffer performance degradation in steady state and during recovery due to the poor fit of the fault-tolerance regime for parts of the applications, or (*ii*) are difficult to maintain because they are developed using fragile combinations of batch and streaming systems that provide different APIs and schedulers, and evolve independently.

This paper describes Falkirk Wheel, a design for rollback recovery that enables applications to combine different fault-tolerance regimes. Falkirk Wheel provides a design based on logical times, which is expressive enough for general applications including incremental and iterative computations. Our experiments show that an implementation of Falkirk Wheel in Naiad successfully combines fault-tolerance regimes, with an order of magnitude lower response latencies in steady state than Naiad's batch-tuned fault-tolerance. Moreover, Falkirk Wheel is competitive with streaming systems tuned for single fault-tolerance regimes, as it provides 3-5× lower response latencies than Flink and Drizzle in steady state and during failure recovery on the Yahoo! Streaming Benchmark.

CCS Concepts

• **Computer systems organization** → *Reliability*.

Keywords

Fault tolerance, Dataflow systems, Rollback recovery

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3487011

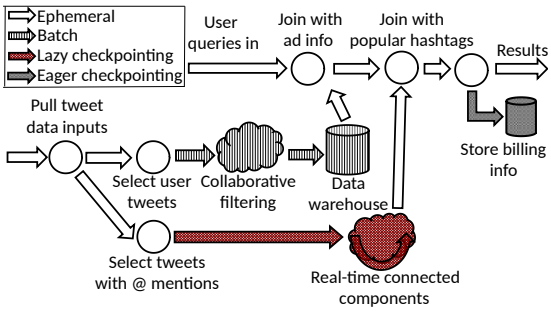


Figure 1: An example ad-serving application comprised of several batch, streaming, and iterative components. Each component uses a different fault-tolerance regime.

ACM Reference Format:
Ionel Gog, Michael Isard, and Martín Abadi. 2021. Falkirk Wheel: Rollback Recovery for Dataflow Systems. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3472883.3487011

1 Introduction

While fault-tolerant systems for data processing have traditionally been designed for either high-throughput batch [18, 25, 37] or real-time streaming [3, 10] operation, many applications combine computations with disparate fault-tolerance requirements. Consider as an example the ad-serving application in Fig. 1. The application receives as input the stream of tweets from the Twitter firehose, and consists of five components that execute in parallel: (*i*) selects tweets that mention user accounts, and computes the weakly connected components in real-time of the Twitter graph of user mentions, (*ii*) selects tweets created by user accounts, and periodically runs collaborative filtering to batch-compute ad recommendations, (*iii*) joins ad recommendations with the output of the connected components computation in order to produce similar ad recommendations to connected users, (*iv*) enables commercial users to query the application to discover popular hashtags within connected components, which can help companies to better target ads, and (*v*) saves statistics and billing information for the served ads in a key-value store.

Four distinct fault-tolerance regimes are appropriate for different parts of such an application:

• **Ephemeral regime**: is used in components that do not store mutable state, require high-throughput processing, and can tolerate slow recovery. Data sources send messages to such components, and these messages are processed without being saved to stable storage. As a result, data sources do not receive an acknowledgment until the messages have flowed through the components, and fault tolerance is attained by requiring data sources to retry on failure. In the example application, the component that joins ad recommendations and hashtags would benefit from this regime.

• **Batch regime**: is used in components that persist outputs and can tolerate re-execution in the case of a failure. This introduces a high increase in latency, possibly of minutes. For example, the data-intensive collaborative filtering computation is a candidate for this regime because it runs periodically and its results are not required to be current.

• **Lazy checkpointing regime**: is used in components that maintain complex state, which must be regularly checkpointed (e.g., the computation of weakly connected components). In this regime, it is acceptable to re-execute a few seconds' worth of work in the event of a failure, so checkpoints need not be taken every time the state is updated.

• **Eager checkpointing regime**: is used in components that must persist messages and state updates as soon as they are processed in order to guarantee consistency with delivered results (e.g., saving ad billing data).

Existing fault-tolerance designs fit several of these regimes, but no current data processing system can support all of them at once [4, 13, 31, 32, 36, 37]. Batch processing systems [18, 37] optimize for throughput and only support the batch regime, which does not offer low recovery time. By contrast, stream processing systems [3, 4, 10] support the eager checkpointing regime, which offers low recovery time, but reduces throughput. As a result, an application like the one in Fig. 1 is executed in one of two ways: (*i*) a single system and fault-tolerance regime, which might introduce performance penalties if the fault-tolerance regime is not well-suited for all components, or (*ii*) several stand-alone systems, each with its own fault-tolerance regime. For example, the application might run on Naiad [31], which provides a batch-tuned fault-tolerance regime, but its real-time weakly connect components computation would suffer from high-latency recovery. Alternatively, the application might use multiple systems: Spark [37] to execute the collaborative filtering component, and Storm [10] to run the the weakly connected components computation. The output of each application component might be saved on a distributed file system (e.g., HDFS), from where dependent components read it [7]. However, this approach has two main drawbacks:

(1) Performance may degrade because all components must output on the distributed file system to communicate with dependent components, even when a component and its fault-tolerance regime do not need to persist outputs.

(2) Ensuring that the application recovers in a globally consistent state in the event of a failure is difficult as all the systems have to coordinate to decide how far components need to rollback and what messages must be replayed.

In this paper we describe the Falkirk Wheel[1] design which subsumes previous, more specialized fault-tolerance regimes (§2). With the help of Falkirk Wheel, applications can execute components in different fault-tolerance regimes, and benefit from the properties of each regime. Falkirk Wheel enables the execution of complex applications by tagging (implicitly or explicitly) all events with partially ordered *logical times* (§3.1), while allowing each application component to use its own *logical time domain* (i.e., its own definition of logical time) suited for its own fault-tolerance regime (§3.2). Moreover, Falkirk Wheel provides an algorithm for choosing a set of logical times to roll back to such that the application remains in a globally consistent state after a failure (§3.3). Falkirk Wheel restores from saved state the effects of events at logical times in the chosen set, and re-executes events with logical times outside the set. Finally, unlike previous fault-tolerance solutions [13, 31], Falkirk Wheel supports *selective rollback* (§3.4), whereby a component that has consumed messages at two logical times t_1 and t_2 may be able to preserve its work for time t_1 after rollback, but undo and re-execute its work for t_2, independently of the order it performed the work.

Falkirk Wheel builds on a large body of prior work on rollback recovery [19]. In particular, Falkirk Wheel is inspired by theoretical research on *timely rollback* [2]. While that research defines the mathematical framework underlying our design in terms of logical times, this paper shows for the first time how to map the abstractions used by timely rollback to fault-tolerance implementation techniques (§4). Furthermore, the paper extends prior work by addressing concerns that are typically of less interest from a theoretical perspective: (*i*) commit of input/output data in order to offer exactly-once semantics, (*ii*) garbage collection of persisted state in order to support long-running applications, and (*iii*) efficient tracking of application progress to support consistent rollbacks[2].

To demonstrate the efficacy of our design, we apply it to Naiad [31], a general dataflow system that can execute complex applications that combine batch, graph, low-latency incremental, and iterative stream processing. We show that Naiad with Falkirk Wheel is both practical and high-performance: (*i*) Falkirk Wheel enables the execution of application components in different fault-tolerance regimes, and thus offers low-latency recovery for some components

[1]Named after a prior solution for high-throughput streaming rollback [12].
[2]We also developed technical extensions to the prior theoretical work, in particular to deal with logical time domains for sequence numbers (§3.1).

(e.g., real-time incremental components) and high throughput for others (e.g., batch components) (§5.1), (*ii*) Falkirk Wheel combines fault-tolerance regimes and provides an order of magnitude lower response latencies than Naiad's current batch-tuned fault-tolerance regimes (§5.2), and (*iii*) Falkirk Wheel can be set to checkpoint at desirable times, and thus offers 3-5× lower response latencies than stream processing systems (Flink [6] and Drizzle [36]) on the Yahoo! Streaming Benchmark (§5.3).

The key contributions of this paper are:

- a design based on logical time (§3.1) for combining fault-tolerance regimes (§3.2), and an algorithm for choosing a globally consistent state to roll back to after a failure (§3.3);
- selective rollback, which enables interleaved processing of messages with distinct logical times (§3.4);
- an implementation of Falkirk Wheel that includes support for long-running applications, provides input/output commit guarantees, and garbage collection of state (§4); and
- an evaluation confirming the practicality and performance benefits of Falkirk Wheel (§5).

2 Background

Setting the stage, in this section we summarize a few fault-tolerance regimes and rollback-recovery protocols, and comment on the design and performance trade-offs they embody. The intention is not to criticize existing techniques, but to explain their different strengths and motivate the Falkirk Wheel approach of combining them in a single system.

In our discussion we refer to a processing operator in a dataflow graph as a *processor*. Each processor p receives messages from other processors on a set of point-to-point streams, which we call edges ($E(p)$). Moreover, we suppose that one or more processors may share a physical CPU, and a single network connection may be used to transmit messages between multiple processors on different machines. Finally, we assume that processors fail-stop, and that network failures are modeled as though the receiving processor(s) have failed and messages in transit were lost.

Lazy checkpointing regime. Chandy and Lamport described a general protocol for checkpointing *distributed snapshots* of an arbitrary distributed system [15]. In this protocol, the system performs a periodic global checkpoint by sending a marker message to source processors, which forward the marker to downstream processors, and so forth until all processors receive the marker. Each processor p takes a checkpoint C_p of its state upon the receipt of a marker message. Besides the processor state, the checkpoint also includes a sequence of undelivered messages M_e on each edge $e \in E(p)$. The design of the protocol ensures that the sets C_p and M_e form a globally consistent system state such that following a failure all processors can be restored to the state at the most recently saved checkpoint.

This protocol for distributed snapshots is similar to our lazy checkpointing regime since processors do not checkpoint every state update. The regime offers good throughput in steady state, and thus is used in the Naiad and Flink [13, 31], but it has several drawbacks: (*i*) each processor must have additional logic to save a checkpoint at an arbitrary moment chosen by the system (logic which must be provided by users), (*ii*) processors can incur checkpointing overhead in the non-failure case, (*iii*) processors may checkpoint at arbitrary times, which may affect application latency in undesirable moments and (*iv*) all processors, even non-failed ones, must roll back to a checkpoint following a failure.

Eager checkpointing regime. Streaming systems such as Storm [10] and MillWheel [3] guarantee exactly-once message delivery to stateful processors, corresponding to the eager checkpointing regime of Fig. 1. In these systems, when a processor receives a message, it persists its updated state and any resulting outgoing messages before it acknowledges the processed message. If a processor fails, the system restores the processor to its most recently persisted state, which includes the effect of all acknowledged messages.

This fault-tolerance regime has four benefits: (*i*) it allows processors to choose independently when to checkpoint, (*ii*) it can guarantee high availability since processors do not have to replay many messages, (*iii*) non-failed processors do not need to be interrupted, and (*iv*) processors may join and leave the computation with low overhead since the system does not need to keep track of the dataflow graph. Drawbacks include a throughput penalty because all state mutations must be persisted, and a latency penalty because a processor's output messages must be acknowledged by their recipient processors before a processor can acknowledge the next input message. While this requirement to acknowledge messages ensures exactly-once semantics, it may limit the complexity of applications due to long chains of dependent acknowledgments (e.g., upstream processors waiting for acknowledgments from downstream processors). Such dependency chains prevent streaming components from processing time windows in parallel, and iterative components from pipelining iterations.

Ephemeral regime. Both Storm [10] and MillWheel [3] also allow processors to execute in a relaxed fault-tolerance regime. In this regime, the processors do not eagerly checkpoint each state update before proceeding to the next, but instead lazily and independently checkpoint state and acknowledge messages. Upon a failure, affected processors restart from their latest checkpoint, replay unacknowledged messages, and thus might duplicate messages. Therefore, this regime guarantees at-least-once semantics and offers better performance than the eager checkpointing regime, but

Fault-tolerance regime	*Processors that must roll back*	*Throughput*	*Steady state latency*	*Recovery latency*
Lazy checkpointing/distributed snapshots	all	medium	medium	medium
Eager checkpointing	failed	low	high	low
Batch	failed or consumed input from failed	high	low	high
Ephemeral	failed or consumed input from failed	high	low	very high

Table 1: Properties of fault-tolerance regimes. We show desirable properties in green, and undesirable properties in red.

should be used only for processors with idempotent computations, or in applications that tolerate inconsistent states.

Batch regime. Many acyclic batch dataflow systems [18, 25, 37] share a fault-tolerance regime pioneered by MapReduce [18]. In these systems, each application processor proceeds through a set of steps. First, a processor reads all of its input messages, followed by an implicit notification after the input is complete. Next it executes a user-specified computation, updates its state, and writes output. Finally, the processor empties its state, and quiesces. Therefore, in case of a failure, these systems choose to restore each failed processor either to a state where it has processed no input data, or to a state where it has processed all inputs.

This regime has the appealing property that processors are always restored to an empty state after failure, so the (user-supplied) processor logic does not need to include any checkpointing code. On the other hand, any computation in progress at the time of a failure is lost and must be re-executed. Nonetheless, non-failed processors do not need to be interrupted, unless they have consumed messages from processors that were restored to the empty state. As a result, the model is well suited to offline data-parallel applications, where throughput in the absence of failures is paramount and high recovery latency is tolerable.

3 Falkirk Wheel: Hybrid Fault Tolerance

Complex applications would benefit from solutions that combine the four broad classes of fault-tolerance regimes described in §2 (see Table 1). Where it leads to acceptable performance, developers can write processors that execute in the batch regime and never checkpoint, sidestepping the overhead and complexity of serializing and restoring state. By contrast, developers can write dedicated checkpointing code for processors that benefit from keeping large amounts of state in memory (perhaps indexed in sophisticated ways), and thus avoid rebuilding the state every time the processor fails by using the lazy checkpointing regime, or by persisting each state update and using the eager checkpointing regime.

In order to enable applications to combine fault-tolerance regimes, Falkirk Wheel must ensure that processors roll back to a globally consistent state after one or more processor failures. Informally, we express the requirements of a globally consistent state rollback in terms of pairwise constraints on processors. If processor p sends to processor q then:

(1) q may roll back only to the extent that it will not need to receive again any message that p has not logged; and
(2) q must roll back far enough to undo the effect of any messages invalidated by p's rollback.

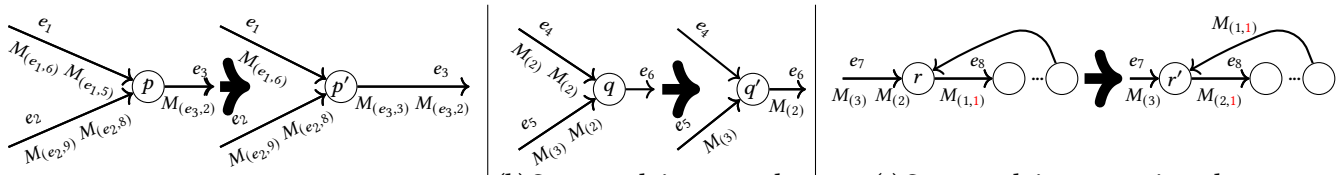
Concretely, when a processor p in state S fails, Falkirk Wheel must make p consistent with its upstream and downstream neighbor processors. Consistency can be achieved with upstream neighbors either by re-delivering lost messages from upstream processors to p, rolling back upstream processors to match p's latest checkpointed state S^*, or a mixture of both. Moreover, if processor p sent messages while transitioning from state S^* to S, the downstream recipients of those messages must typically be rolled back in case some source of non-determinism causes processor p to emit different messages after processor p is restarted in state S^*.

Therefore, Falkirk Wheel has to be able to reason about the relationship between the events at a processor, the messages sent by the processor, and the states of its neighboring processors. Falkirk Wheel enables such reasoning and captures the above constraints in terms of two central concepts: *logical times* (§3.1) and bridges between *logical time domains* (§3.2). In Falkirk Wheel each set of processors can operate in their own logical time domain, with logical times evolving differently in each domain. For example, a processor that tags each output message with a different logical time might execute in the eager checkpointing regime, and thus log each state and output message. By contrast, a processor that tags all output messages with one logical time during a wall-clock time window operates in a different logical time domain, might use the lazy checkpointing regime, and thus checkpoint only once per time window.

In this section, we focus on the case where logical times are totally ordered on each edge, and messages on an edge with logical time t_1 are delivered before messages with logical time $t_2 > t_1$. While this case is sufficient to support the fault-tolerance regimes and the systems we describe in §2, we relax these constraints in §3.4 to enable the parallel execution of different logical times and selective rollback of state corresponding to different logical times. We describe the concepts informally, but formalize them in §3.3, and provide a regime-agnostic algorithm for globally consistent rollback.

3.1 Logical Times

Falkirk Wheel tags messages between processors with partially ordered logical times of two broad categories:



(a) Sequence numbers. (b) Structured times: epochs. (c) Structured times: entering a loop.

Figure 2: Example of processors that operate in different logical time domains. Tuples $(\cdot)$ show the logical time of each message. In Fig. 2a, the logical time of a message with sequence number n on edge e is (e, n). p has processed the first four messages on edge e_1 and the first seven on e_2, and has sent two messages on e_3. p' shows p's next state in which it has processed another message on e_1 and has issued a message on e_3 in response. Fig. 2b uses epoch numbers as logical times; all messages in a given epoch have the same logical time. q' has processed and sent all messages of the first two epochs. Fig. 2c uses structured logical times, generalizing epochs. r forwards incoming messages into a loop that operates in a different logical time domain, which includes an additional loop iteration counter (shown in red). r' has started the loop for the second epoch, and is about to complete the first loop iteration of the first epoch.

• **Sequence numbers** assign a message counter to every input and output of a processor. They are useful for unstructured computations where little is known about the semantics of the processor, since they track computation in a fine-grained way.

• **Structured times** may be shared by many input and output messages. They are used for structured computations, where processors may associate each input message with a particular *epoch* of input, and may tag all consequent state updates and output messages with an epoch. We use the term epoch for a batch of data that shares fate for the purposes of fault tolerance, but more generally, structured logical times can include other coordinates beyond an epoch, for example to indicate nested loop iteration counts.

Sequence numbers are system-defined and dynamic: a processor with several inputs may produce outputs with different sequence numbers based on the non-deterministic arrival times of messages on its input edges (see Fig. 2a for an example of a processor with two inputs and two outputs, which uses sequence numbers). By contrast, structured logical times are application-defined, and are used when each message is associated with a particular batch or epoch of input (see Fig. 2b). Moreover, structured logical times are used in applications that include loops implemented as distributed sets of processors, and that benefit from overlapping computations from different loop iterations (e.g., weakly connected components). Each structured logical time indicates an input epoch along with loop counters tracking progress through (possibly-nested) iteration. Fig. 2c shows an example of structured logical times in the context of such a loop.

Falkirk Wheel uses logical times to determine a globally consistent rollback after a failure. It chooses a set of logical times at each processor, which we call a *time span* (or *span*), and restores the processor to a state that includes only the effect of the delivered messages with logical times in the chosen span. We note that Falkirk Wheel may be able to "roll back" a non-failed processor to a special span $\top$, which includes all logical times for processed messages (i.e., the processor continues working unaffected by the failure).

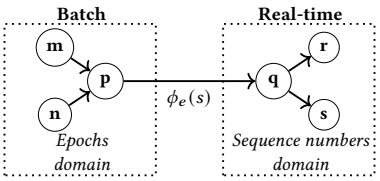


Figure 3: Application composed of a batch and a real-time component. The components have different fault tolerance requirements, and thus use different logical time domains. ϕ bridges between the domains.

3.2 Bridging between Logical-Time Domains

One of the key features of Falkirk Wheel is its ability to execute applications comprised of components executing under different fault-tolerance regimes (i.e., in different logical time domains). Fig. 3 shows a simplified example of such an application, which consists of batch and real-time components that execute in different logical time domains. The batch component is optimized for throughput, tags messages with epochs, and restarts an entire epoch in case of a failure. The real-time component is optimized for low-latency recovery, uses sequence numbers, and checkpoints each state change.

In order to compute globally consistent rollbacks for this application, Falkirk Wheel must bridge between the two logical time domains. Therefore, for each edge e which connects two logical time domains (from processor p to q), Falkirk Wheel uses a *transformer* function $\phi_e(s)$ that maps a span s at p to a span at q. The function ϕ_e summarizes the effects that p caused at q, and encodes information about the semantics of p's behavior to ensure that neighboring processors' spans are consistent at rollback. Given a processor p and an outgoing edge e, the span $\phi_e(s)$ is a conservative estimate of

the logical times that were "fixed" on e given the messages in s at p. In particular, p is guaranteed not to have produced any messages with logical times in $\phi_e(s)$ as a result of processing a message with a logical time outside (later than) s. This means that it is "safe" to roll q back to $\phi_e(s)$ whenever p rolls back to a span at least as large as s. As long as the guarantee is met, any choice of ϕ will lead to correct behavior. If we do not know anything about the semantics of p we can always set $\phi_e(s) = \emptyset$ (i.e., roll q back to its initial state), but we would like to choose it as large as possible, since a larger ϕ will allow us to preserve more work during rollbacks.

Only processors that manipulate logical time (e.g., control-flow processors) and those that bridge between time domains need non-trivial ϕ functions. Nonetheless, we discuss several instances of bridging between logical time domains:

Transformers for sequence number spans. When logical times are pairs of the form (e, n) where e is an edge and n is the sequence number of a message on that edge, we let $(e_1, n_1) \leq (e_2, n_2)$ if and only if $e_1 = e_2$ and $n_1 \leq n_2$. Thus, logical times form a partial order and are comparable only if they correspond to messages on the same edge, and within an edge sequence numbers indicate the natural ordering. The span at a given state of a processor is the set of sequence numbers of messages it has processed to reach that state. The transformed span along its outgoing edge e is the set of message sequence numbers it has sent along that edge in response to the incoming messages it has processed in a particular program execution. The transformer may be a function not only of the logical times of input messages but also delivery order and other non-deterministic choices.

Connecting epochs and sequence numbers. Fig. 3 shows a processor p that receives messages from batch processors that tag messages with epochs, and sends messages to processor q that eagerly checkpoints according to sequence numbers. In this case the application might require p to forward all epoch t_1 data before sending any epoch t_2 data, if necessary buffering epoch t_2 data. If, during an execution, p sends n epoch t_1 messages on edge e, then we would let $\phi_e(\{t_1\})$ be the smallest span that contains logical time (e, n).

Entering a loop. Fig. 2c shows a processor r, which receives messages tagged with epochs and sends them to a processor in a logical time domain that includes a loop iteration counter. Output messages have logical times (t, c) where t is the epoch of the incoming message and c is the iteration counter. In this case we would choose $\phi_e(s)$ to be $\{(t, c) : t \in s\}$.

3.3 Globally Consistent Rollback

Connecting logical times and transformers, we now detail the constraints necessary for rolling back to a globally consistent state, and an algorithm for choosing such a state. We first introduce notation for additional metadata and outputs Falkirk Wheel must capture for each edge e:

- L_e: the sequence of messages sent on edge e that were logged (if any);
- T_{D_e}: the set of logical times of any messages send on edge e but not logged (i.e., discarded in case of a failure); and
- T_{M_e}: the set of logical times of any messages that arrived and were processed on edge e.

We write $T_{D_e}(s)$ for the set of times in T_{D_e} that correspond to messages generated in response to events in a span s. Even if processors do not log messages or save metadata we indicate conservative estimates of the missing quantities that still allow the application to roll back to a consistent state.

The constraints that must be satisfied between a pair of processors, where p sends to q on edge e, are:

No discarded messages are lost. The first constraint is $T_{D_e}(s(p)) \subseteq s(q)$ and ensures that no discarded messages are lost. If p logs all the messages it sends from events in $s(p)$ then there are no discarded messages (i.e., $T_{D_e}(s(p)) = \emptyset$) and the constraint is trivially satisfied. However, if p does not log output messages, then it is often the case that $T_{D_e}(s(p)) \subseteq s(p)$ because many processors only ever send messages at a logical time t in response to processing events at t. Such processors need not keep track of T_{D_e} and can conservatively assume $T_{D_e}(s(p)) = s(p)$. By contrast, if processors send "into the future", they must keep track of T_{D_e}, or roll back completely on every failure (e.g., Differential Dataflow [29] processors that output messages with time t_2 in response to messages with logical time t_1, where $t_1 < t_2$).

No messages are duplicated. This constraint ensures that processor q rolls back far enough that any messages it received are within the span "fixed" by p's rollback (i.e., $\phi_e(s(p))$), in the sense described in §3.2. If q has not explicitly kept track of $T_{M_e} \cap s(q)$ then it is safe to proceed conservatively as if $T_{M_e} \cap s(q) = s(q)$.

No transmitted messages are lost. This is a technical constraint, which ensures that no messages in transmission are lost in case of a failure. The constraint is that p and q can only roll back to $s(p)$ and $s(q)$ as long as no message with a time in $s(q)$ sent by an event at p with a time in $s(p)$ was lost in transmission during the failure. Our implementation satisfies this constraint by saving a checkpoint for span $s(q)$ only after it is certain it will not send any more messages to q with times in $s(q)$; we say that the times in $s(q)$ are *complete* at q when this condition is satisfied. In order for q to "roll back" to $s(q) = \top$ then either $s(p) = \top$ in which case we know that no messages from p to q were lost, or there must be some span s such that all messages sent by p in s have times in s, and all times in s are complete in q.

Falkirk Wheel ensures that a rollback satisfies the above-mentioned constraints at all processors by choosing a maximal span for each p. Each span is chosen from $\hat{S}(p) = \{s_1, \ldots, s_K\}$, where $s_1 \subset s_2 \subset \ldots \subset s_K$, and denotes the

sequence of spans for which p has saved enough information to roll back to. Falkirk Wheel chooses the maximal spans for rollback using the following fixed-point algorithm, where $In(p)$ and $Out(p)$ are the incoming and outgoing edges of p.
Initially: $\forall p,\ s(p) = \max\{s \in \hat{S}(p)\}$.
Continue until fixed point:

$$s'(p) = \max\{x \in \hat{S}(p) \text{ such that } x \subseteq s(p)$$
$$\wedge\ \forall e \in Out(p),\ T_{D_e}(x) \subseteq s(dst(e))$$
$$\wedge\ \forall d \in In(p),\ T_{M_d} \cap x \subseteq \phi_d(s(src(d)))\}$$

After the algorithm finds the solution, Falkirk Wheel restores p, from checkpoint and/or logs, to the state corresponding to the events in $s(p)$. Following, it removes any spans greater than $s(p)$ from $\hat{S}(p)$, and discards any logged messages that were sent by events with times outside $s(p)$. Finally, it sets the message queue on an outgoing edge e to processor q to contain the messages in L_e whose times are not in $s(q)$.

We note that as long as $\emptyset \in \hat{S}(p)$ for all p, the algorithm converges since s never increases, and $s(p) = \emptyset$ for all p satisfies all constraints. While the algorithm always finds a solution, applications might suffer cascading rollbacks in pathological situations. Falkirk Wheel offers a flexible design to support many choices of where and when to checkpoint and log, but it is the application developer's duty to ensure that logging/checkpointing happens in appropriate places such that the application does not suffer cascading rollbacks. For example, a developer could choose a more restrictive regime (e.g., eager checkpointing) to execute the application.

3.4 Selective Rollback

In the previous subsections, we implicitly assumed that the events in a span $s(p)$ are a prefix of the events that had been delivered to p, so rolling p back to s is equivalent to restoring p to its state at the moment when it had processed that prefix of events. However, this restriction requires a processor to suspend delivery of a message until all messages with earlier logical times are processed. As a result, the processor stalls until no earlier messages remain in the application, introducing additional latency and requiring messages to be buffered that could otherwise be eagerly delivered. For example, a processor that computes analytics over sliding or rolling time windows cannot process multiple time windows in parallel. Furthermore, if input messages arrive after the end of their time window, which is common [4], the processing of other time windows is further delayed.

To address this restriction a developer could shard logical times across processors. However, this approach would reduce the performance of processors that need to simultaneously update state for multiple logical times (e.g., incremental computations). As a result, we introduce *selective rollback*, which enables Falkirk Wheel to meet the following twin performance requirements: processors must be able to interleave the delivery of messages with different logical times, and also checkpoint only state corresponding to completed logical times. Selective rollback allows a processor that has consumed messages at two logical times t_1 and t_2 to preserve its work for time t_1 after rollback, but undo and re-execute its work for t_2, independently of the order it performed the work. However, not all processors can selectively roll back because they must produce correct results under certain message re-orderings on their input edges. Concretely, a processor cannot selectively roll back if its input messages with a given logical time are not delivered in FIFO order.

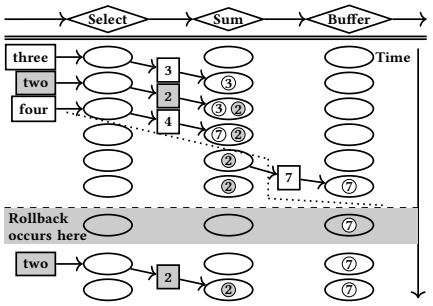


Figure 4: Selective rollback. Rectangles show messages and ovals processor state. A message or state with a white background corresponds to logical time t_1, while a gray background corresponds to logical time t_2. The dotted line indicates when a processor will not receive any more messages with time t_1; at this point the Sum processor sends an output message and discards its state related to t_1. Lastly, the dashed line indicates the time when a processor failure occurs. Due to selective rollback, processors can process logical times in parallel, and upon failure can roll back to a state where they have consumed all messages at t_1 and none at t_2.

While not all processors can selectively roll back, we believe selective rollback is practical because most applications deliver messages in FIFO order, and either keep no state at a processor or partition a processor's state by logical time (e.g., an aggregate counter keeping an intermediate counter for a batch of input messages, state stored during a time window). Moreover, many processors can safely delete the state corresponding to a logical time once that logical time is complete (e.g., at the end of a time window, or after an aggregate counter has processed inputs and released output). As a consequence, if a processor can wait until logical time t_1 is complete and its portion of local state is deleted, then the processor does not need to checkpoint.

To exemplify selective rollback, Fig. 4 shows a fragment of a simple application made up of Select, Sum, and Buffer processors, as well as a timeline of message deliveries and corresponding updates to the processor state, shaded according to logical times. The Select processor translates a word

into its numeric representation, and is stateless. The Sum processor accumulates a sum for each logical time. When notified that there will be no more messages at a given logical time (i.e., time is completed), Sum outputs the accumulated sum for that time and removes the sum from its local state. The Buffer processor records all messages it has seen.

In this application, each processor makes a selective checkpoint after seeing the final message tagged with logical time t_1 (shown with the dotted line). The selective checkpoint does not include a processor's full state, but only the state it would contain having only seen all messages from logical time t_1, but no messages from t_2. Upon a processor failure, the processors first roll back (shown as a shaded rectangle) to their selective checkpointed state. Subsequently, an upstream processor re-executes, causing the time t_2 message to be re-sent, and eventually the application state returns to that before the rollback. We note that executing the application without selective rollback would prevent it from interleaving the delivery of messages at different logical times, and would demand checkpointing of non-empty state for its Sum processor, either of which would introduce a substantial performance penalty.

Falkirk Wheel supports such applications, and is correct even if a processor p delivers messages out of order (e.g., some events outside of span s are processed before some events inside s). Then restoring to span s corresponds to rolling p back to a *selective* checkpoint that remembers only the events in span s. Falkirk Wheel achieves this by using the fixed-point algorithm (§3.3), which computes consistent spans even when processors selectively checkpoint.

4 Falkirk Wheel Implementation

In order to demonstrate the practicability of our design, we apply Falkirk Wheel to Naiad (FW-N), a general dataflow system, which combines batch, graph, incremental and iterative stream processing [31]. We describe the practical challenges in applying Falkirk Wheel to data processing systems, but also the changes we made to Naiad. We focus on issues that would be common to any implementation, such as: failure recovery (§4.1), logging and checkpointing for different fault-tolerance regimes (§4.2), garbage collection of checkpoints and management of inputs/outputs in long-running applications (§4.3), updating processors to use Falkirk Wheel (§4.4), and applying Falkirk Wheel to other systems (§4.5).

4.1 Failure recovery

FW-N provides a **span monitoring service** that keeps track of progress across processors, and executes the algorithm for computing globally consistent spans to roll back to in case of a failure (see §3.3). For each processor p, the span monitor tracks the following metadata: (*i*) which spans p can restore to, (*ii*) the maximal times of processed events at those spans, and (*iii*) the times of messages p sent but did not log. FW-N updates this metadata after a processor has persisted a checkpoint for its state corresponding to a span s, and has logged any messages it sent from times in s. Despite the regular updates, the in-memory metadata is small, at most a few Kb per span. Moreover, the monitor is simple to replicate for reliability because it is deterministic and monotonic, and does not have a large memory footprint.

In our FW-N implementation, a processor p discovers the failure of another processor q by the failure of a network connection to a remote machine. When this happens p continues to work, buffering output to q in case the connection is re-established. However, when a failure detector confirms q's failure, FW-N initiates recovery.

During recovery processors must pause execution before rollback. However, before processors can pause they must finish executing the user code for any event being processed. Because this may be slow, we allow developers to divide processors into three types: latency-insensitive, normal, and low-latency. *Latency-insensitive processors* are asked to pause as soon as a failure is detected, and drain their event processing in parallel with the restart of failed processors. Once failed processors have restarted, FW-N tells *normal processors* to pause, and finally FW-N asks *low-latency processors* to pause after the normal processors have stopped. Thus, our implementation minimizes the time that a low-latency component is suspended after a failure.

Pausing processors send up-to-date metadata to the span monitor, which temporarily also adds $\top$ to the available rollback spans for any non-failed processor. At this point the monitor executes the algorithm for computing the globally consistent spans to roll back given the failed processors (see §3.3), and transmits the computed spans to the paused processors. Following, the application restarts, and the processors that restored to $\top$ continue unaffected, while other processors restore to their rollback spans, and lazily re-read and retransmit logged messages.

By default a stateful processor that is rolling back is reinitialized to an empty state and presented with streams from which to restore. However, non-failed processors may optionally maintain data structures to allow *in-memory rollback* to a span s by discarding state corresponding to events outside s instead of restoring from checkpointed storage. This substantially reduces recovery time, and the aggregate load on distributed storage as many processors roll back at once.

4.2 Logging and Checkpointing

The support for logging and checkpointing we added to Naiad is similar to that which would be required for any data processing system. Moreover, it was easy to make FW-N automatically keep track of metadata for all processors. Naiad already requires that messages are serializable in order to

support distributed operation so we were able to log sent messages, where indicated by the application, without changing the implementation of any processor. We next describe the logging and checkpointing required for processors to run in the fault-tolerance regimes we introduced in §1.

Ephemeral regime. A processor can request the logging of all of its delivered messages, which gives any deterministic processor without external side effects fault-tolerance with no coding effort. FW-N can roll back the processor to any span by replaying the sequence of messages the processor has processed with times in the span. This regime is a good fallback, but the sequence of messages grows without bound so it is not suitable for long-running applications.

Batch regime. A processor can use this regime if it does not keep state between logical times. We call such a processor "stateless" even though it may accumulate state within a logical time. FW-N periodically flushes logged messages and persists metadata for such processors, and informs the span monitor so that garbage collection can take place. In our implementation, we did not have to change the user-provided code of any stateless processor while adding fault-tolerance.

Lazy checkpointing regime. A processor can run in this regime by asking FW-N to trigger periodic checkpoint callbacks when times are completed. Processors use local policy to decide when to write checkpoints in response to callbacks, and can write either *full* or *incremental* checkpoints. FW-N buffers writes to stable storage and delivers an asynchronous notification once the writes have been persisted. As a result, applications are not stalled by fault-tolerance traffic to stable storage unless the buffer fills up, which happens only when writing large full checkpoints.

Eager checkpointing regime. A processor uses this regime in order to offer low-latency recovery. The processor persists its state and messages, and sends metadata updates to the span monitor after it completes each logical time.

4.3 Executing Long-Running Applications

Applications that run continuously impose constraints on Falkirk Wheel's design: (*i*) they must run without exhausting physical memory or amassing unbounded state in stable storage, and (*ii*) they must communicate reliably with external input sources and output consumers, to receive and send unbounded streams of data. We now describe mechanisms that enable FW-N to support such applications.

Garbage collection. FW-N garbage collects persisted state of applications with the help of the monitor service. The monitor receives metadata describing a new persisted checkpoint, and continuously runs an incremental version of the algorithm outlined in §3.3 to track, for each processor p, a *low-watermark* of the earliest span p would have to roll back to in the worst case that all processors failed at once. The monitor informs p every time its low-watermark span increases, so p can garbage collect superfluous state. Moreover, it also notifies processors that send to p, so that they can discard logged messages that no longer need to be re-sent. As a result, the redundant persisted state and logged messages are garbage collected as soon as the incremental algorithm computes a new low-watermark span (usually several milliseconds after the monitor receives the metadata).

Inputs and outputs. To be able to provide exactly-once semantics, we require that services producing and consuming input and output support fault tolerance via acknowledgment and retry. Services producing input keep a batch of data available, and re-send it if requested, until the batch is acknowledged by the application. The application must be prepared to re-send a batch of output until it is acknowledged by the consuming service. These assumptions are compatible with services such as Kafka [8] and Azure Event Hubs [30].

FW-N uses the span monitor to handle input and output acknowledgments. An input processor acknowledges all inputs received at times in a span s when it is informed by the monitor that it will never need to roll back beyond s. Similarly, an output processor buffers messages and does not send them until told their times will never be rolled back.

4.4 Adding Fault Tolerance to Naiad Processors

Most users leverage existing libraries of operators (e.g., Spark transformations, Naiad Differential Dataflow) to develop applications. Such libraries will ship with default operator-specific fault-tolerance, which will need to be tuned only by power users. We now discuss the changes we made to apply Falkirk Wheel to Naiad's main libraries: (*i*) Lindi, a library of processors that keep no state between logical times, with similar functionality to Spark [37] plus support for iterations, and (*ii*) Differential Dataflow [29], a library designed for incremental iterative computation, comprised of processors that keep state in order to respond quickly to updates.

All Lindi processors are stateless, and thus we suppress logging of sent messages (L_e) and assume that each processor receives a message in each epoch (i.e., the worst case). As a result, FW-N can reconstruct processors' metadata on failure without persisted state, and update the span monitor on new completed spans without waiting for stable storage. Thus, Lindi processors incur no fault-tolerance overhead beyond periodically sending metadata updates to the span monitor. We note that while most Lindi processors do not log sent messages, some might be instructed by developers to log sent messages in order to prevent cascading rollbacks in upstream processors (like a persisted Spark RDD).

Differential Dataflow processors store state differentiated by logical time, and use a common library for maintaining in-memory state. Processors partition state in order to support low-latency incremental updates with iteration [29], and we

believe that any system that has the capability to overlap processing at different logical times is likely to adopt a similar state partitioned by time. Applying Falkirk Wheel to Differential Dataflow required adding persistence to the library for maintaining state, but no additional effort for processors. Moreover, given the time-partitioned state, it was straightforward to implement selective checkpointing/restoring (full or incremental) of the state corresponding to a set of times.

Most of the complexity of adding fault-tolerance to Differential Dataflow processors arose from our choice of implementing *in-memory rollback*. The main challenge was that in order to prevent state from growing, processors can coalesce state corresponding to distinct logical times in the past as logical times become complete. Thus, FW-N must take care not to compact state that would need to remain distinct if a processor were to roll back. To ensure correctness, FW-N divides logical times into three regions: (*i*) *stable times* beyond which the processor will not roll back, (*ii*) *recent times* that have not yet been checkpointed, and (*iii*) all other times. FW-N allows processors to coalesce times only in the stable and recent regions.

4.5 Applying Falkirk Wheel to Other Systems

In order to benefit from Falkirk Wheel, users of existing data processing systems would not be required to migrate their applications to Naiad, but developers would have to integrate Falkirk Wheel into their systems. The changes would be minimal, similar to the ones we made to Naiad (e.g., sending metadata to the monitor, pausing processors for rollback, choosing fault-tolerance regimes for libraries). Moreover, these systems could reuse the span monitor we implemented, and one could imagine a setup with data processing systems subscribing to the monitor and communicating via transformer functions, much as systems share Zookeeper [22].

5 Evaluation

We use our FW-N implementation to evaluate the efficacy of our Falkirk Wheel design. We seek to answer:

(1) Does FW-N offer low-latency responses in steady state and during failure recovery for applications that combine batch, real-time, and graph processing? (§5.1)

(2) How does FW-N compare to Naiad's batch-tuned fault-tolerance regime? (§5.2)

(3) How does FW-N compare to fault-tolerance regimes offered by stream processing systems? (§5.3)

(4) Does selective rollback reduce latency? (§5.4)

(5) How does the system span monitor scale? (§5.5)

Setup. We performed all experiments on a cluster of 25 machines. Each machine has a Xeon E5-2430Lv2 CPU (12× 2.40GHz), 64 GB RAM, and a 30 GB SSD drive.

5.1 Latency on the Ad-Serving Application

To demonstrate the benefits of combining different fault-tolerance regimes, we developed an application that comprises of batch analytics, real-time graph processing, and real-time analytics. The application has the structure of Fig. 1, and contains contains 107 stages in the dataflow graph, where each stage is replicated as multiple data-parallel processors.

The application receives input data in batches of around 10 MB once a second. The first stages select data and reduce each batch to around 1.2 MB. These reduced batches are then persisted so that the application can acknowledge inputs without having to persist the non-reduced input batches.

Following, the real-time component computes the weakly connected components of a graph derived from a sliding window of the input data, running on 15 machines. The component updates the graph every 20 seconds using data from the newly arrived batches, and selectively checkpoints each processor at the span corresponding to every update, at which point data from the next window are flowing around the loop of the connected components algorithm (i.e., the connected computation has started for the next window).

Simultaneously, the batch component emulates a collaborative filtering algorithm by executing a slow pipeline of three MapReduce steps running on five machines. The component runs once every minute on newly arrived data, and logs its final output, but no intermediate state.

Lastly, the application implements user analytics queries using a distributed join running on four machines. This component matches the most recent persisted output of the batch collaborative filtering component and the output of the real-time connected components computation as reported by the span monitor. Thus, its processors are guaranteed never to be rolled back in the event of a failure in the other components.

In Fig. 5, we show the externally visible consequences of emulated processor failures. The triangle data points show query latencies, plotted at the times the queries complete. The vertical dash data points show the latencies of input batches being persisted (so they can be acknowledged to the data source), plotted at the time the batches have persisted. The first dashed vertical line shows the time when three processors fail. One processor is mapping input batches, and the other two are part of the real-time connected components computation. The second line shows the time when all failed processors are ready to restore their state, and all processors have been paused (see §4.1). The third line shows when the monitor has computed rollback spans, and the fourth shows when the failed processors have restored their state and resumed. At around 263 seconds all the batches that were queued during the failure can be seen being persisted, having been processed in parallel by the system.

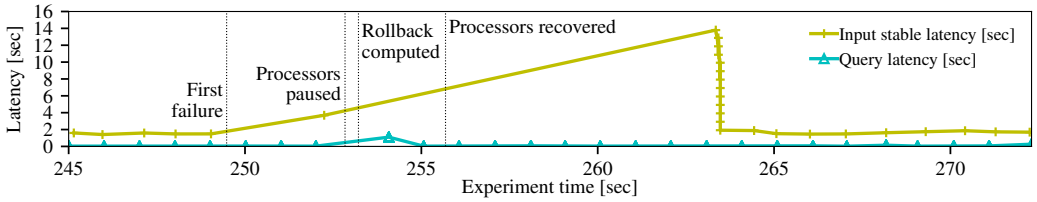


Figure 5: Latency of different application components during failure recovery. FW-N allows query latency to remain low despite three machine failures (at around 250 seconds), masking the longer recovery time required for the latency of batch input processing to return to its normal levels.

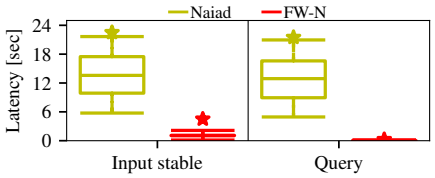
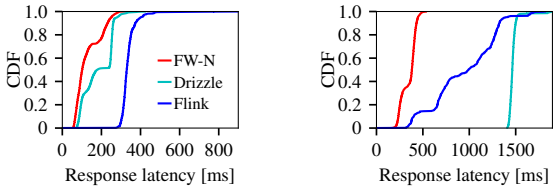


Figure 6: Latency in the absence of failures. Unlike Naiad's synchronous coordinated checkpointing, FW-N offers low latency for both components because it combines multiple fault-tolerance regimes.



(a) At-least-once semantics. (b) Exactly-once semantics.

Figure 7: Latency in steady state. FW-N has lower response latency than Flink and Drizzle, both with at-least-once (a) and exactly-once (b) semantics. Results on YSB executed on 25 machines.

Takeaway: the experiment illustrates that: (*i*) the latency of queries is not significantly affected by failures of other application components, and (*ii*) components can recover independently from failures. These two benefits are due to FW-N's ability to combine fault-tolerance regimes.

5.2 FW-N *vs.* Naiad

We now use the previously described application to compare FW-N with the batch-tuned synchronous coordinated checkpointing fault-tolerance regime provided in Naiad by default [31]. In the experiment, we set Naiad to pause all processes every 10 seconds in order to take a checkpoint of their state. This checkpointing frequency represents a sweet spot. Checkpointing too often increases the latency of the batch components, while checkpointing infrequently adds latency to the user queries, which require exactly-once semantics and cannot be released until state is checkpointed (i.e., Naiad has an output latency greater than the time between checkpoints). Fig. 6 shows the latency between batch inputs arriving and being stably persisted (on the left-hand side), and the latency from query arrival to response (on the right-hand side). The plots show median, 25th and 75th percentile (box), 1st and 99th percentile (whiskers), and max (star) latencies over 300 input batches and 1,000 queries.

Takeaway: Naiad offers a single fault-tolerance regime, and thus it cannot offer both high throughput for the batch component, and low latency for the query component. By contrast, FW-N combines fault-tolerance regimes, logs records where appropriate, and releases messages as soon as the rollback low-watermark is updated by the span monitor.

5.3 Response Latency on Streaming Applications

We now compare FW-N with the fault-tolerance solutions of two state-of-the-art streaming systems: (*i*) distributed snapshots [13] for Flink [6], and (*ii*) Drizzle [36] for Spark [37]. We investigate the response latency both in steady state and during failure recovery by executing the Yahoo! Streaming Benchmark (YSB) [16]. The YSB application computes ad campaign statistics. It receives a stream of JSON ad view events as input from Kafka, selects ads of a certain type, associates them with a campaign, and computes total ads for each campaign for every 10 second window.

We replicate Drizzle's streaming workload latency experiment [36, Fig. 8], and we measure *response latency*, defined as the time from the end of a window until the results are output for the window. In the experiment, we compare with Drizzle's published YSB implementation, and an optimized Flink implementation [21]. Moreover, in order to isolate the performance of the fault-tolerance solution from the performance of other systems used in the YSB application, we produce input data on the fly rather than ingest it from Kafka. Otherwise, Kafka would become a bottleneck at around 300,000 events per second per machine [28]. Finally, we ensure that all our implementations do not deserialize JSON strings, but directly produce input data, as otherwise the application would be bottlenecked on JSON deserialization [28].

Steady state. In Fig. 7a, we show response latencies in steady state while processing 10 million events per second. We configured the implementations to provide at-least-once semantics in order to match Drizzle's experiment [36, Fig. 8]. For each system, we report the numbers for the best configuration (Drizzle uses 200 ms long batches and a group size of 10; Flink checkpoints every second).

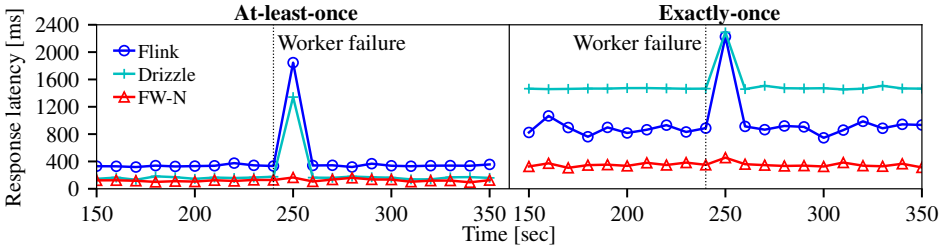


Figure 8: Latency during failure recovery. FW-N has up to 10× lower mean response latency during failure recovery, both with at-least-once and exactly-once semantics. Results on YSB executed on 25 machines.

Following, we modify the implementations to provide exactly-once semantics, and we show the response latencies of the three systems in Fig. 7b. In this setup, FW-N offers approximately 3× lower response latency than Drizzle, and around 5× lower than Flink beyond 50th percentile. These latency differences are fundamental. Flink checkpoints at pre-defined processing time intervals, which might happen at unsuitable times (e.g., during a time window), and thus force the application to checkpoint large states. Moreover, Flink can only release outputs after it checkpoints state (i.e., each second after it is certain output will not to be rolled back). Similarly, Drizzle can only release output after it checkpoints state at the end of each group of tasks. While one could increase checkpointing frequency, this would affect throughput and in many cases cancel the latency reductions obtained from being able to release outputs more frequently. By contrast, FW-N tracks fine-grained computation progress, and releases output as soon as it updates spans.

Failure recovery. In Fig. 8, we show response latency during recovery from an emulated machine failure at 240 seconds experiment time. On the left we execute with at-least-once semantics to replicate Drizzle's failure recovery experiment [36, Fig. 7][3]. On the right we show response latencies when providing exactly-once semantics. In both setups, FW-N recovers quickly and offers a mean response latency several times smaller than Drizzle and Flink.

Takeaway: The monitor tracks fine-grained spans and enables FW-N to offer lower response latency than other systems in steady state and during recovery.

5.4 Benefits of Selective Rollback

To explore the benefits of selective rollback, we develop a more challenging version of the Yahoo! Streaming Benchmark that stresses the performance of FW-N. In this experiment, we increase the workload 12× (i.e., 120 million events per second), we reduce the window to one second, and we delay some ad events such that they arrive outside of the window in which they are created. We note that we only show results for FW-N because the other systems could not keep up with this high-throughput workload. Drizzle and Flink scaled up to 60 and 90 million events per second, respectively.

In the presence of delayed events, FW-N without selective rollback can either use Chandy-Lamport to coordinate checkpointing (similar to Flink), or block processing of other windows until it finishes processing the window with the delayed events. The former approach increases latency, whereas the latter approach causes a backlog of events to accumulate while Naiad waits for the delayed events. Since the former approach does not outperform FW-N on a less challenging benchmark (see Flink results in Fig. 7b), we choose to compare the latter approach with FW-N with selective rollback enabled. In Fig. 9a, we show a timeline of mean response latency during the experiment. At 30 seconds in the experiment, we delay several ad events, which we submit 400 ms after the end of their time window. FW-N with selective rollback can process subsequent time windows, and thus offers low response latency in the presence of delayed events.

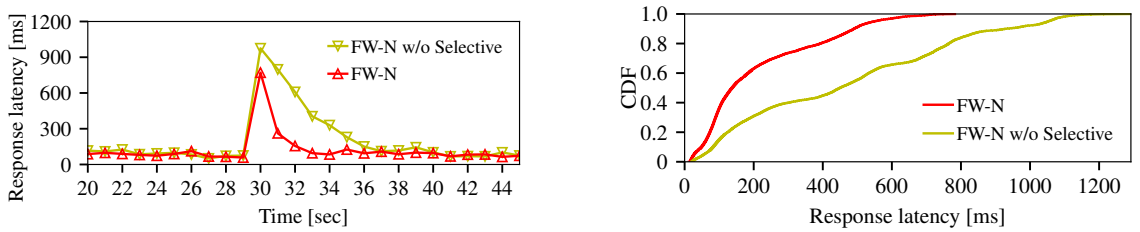
Moreover, in Fig. 9a we highlight the differences in response latencies when events are delayed often. We show results as a CDF of response latencies when several events are delayed by 400 ms every five seconds. While not shown in figure, it is also important to note that the difference in response latency of the configurations grows as the fraction of time windows with delayed events increases.

Takeaway: FW-N with selective rollback reduces response latency by enabling parallel processing of time windows in the presence of delayed events.

5.5 Scalability of the Span Monitoring Service

We study the scalability of the span monitor by measuring the runtime of its slowest component, the algorithm that computes a globally consistent state to roll back to (see §3.3). In the experiment, we vary the number of processors by running the ad-serving application with an increasing number of processors (i.e., higher parallelism). In Fig. 10 we show

[3]Our initial Flink results were similar to prior results [36, Fig. 7], but after an investigation we found that reducing "akka.ask.timeout" to 1500 ms, "gate-invalid-addresses-for" and "retry-gate-closed-for" to 100 ms, decreases latency during recovery by 10×. These flags control the timeout Flink uses to mark workers and tasks dead. We set the values as low as possible while ensuring that workers and tasks are not incorrectly marked dead.



(a) Several events are delayed by 400 ms at 30 seconds. **(b) Several events are delayed by 400 ms, every 5 seconds.**

Figure 9: Benefits of selective rollback. FW-N with selective rollback offers lower response latency than FW-N in the presence of delayed ad events.

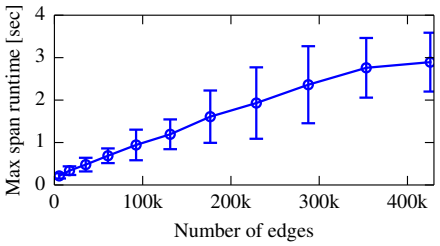


Figure 10: Span monitor scalability. The slowest component of the monitor, the algorithm for choosing maximal spans for rollback, scales linearly with the number of edges.

the runtime as the number of processors increases, and implicitly the number of edges between them (i.e., connections between processors). Since the algorithm uses pairwise constraints between processors, as expected the runtime scales linearly with the number of edges.

Takeaway: The algorithm completes within seconds even for large applications that have hundreds of thousands of edges between processors. We anticipate few applications to have as many edges in practice. Nevertheless, the runtime could be further reduced by applying divide and conquer, and by splitting the monitor into a hierarchy of span monitors.

6 Related Work

Dataflow systems such as Apache Samza [9], Storm + Trident [10, 35], MillWheel [3], Google Dataflow [4], StreamScope [27], and TimeStream [33] support stateful processors and a fault-tolerance regime that guarantees exactly-once semantics. TimeStream uses sequence numbers to track progress, and is similar to our solution for processors where little is known about their semantics. MillWheel, Storm and Samza persist messages and state updates as soon as they are processed (corresponding to our eager checkpointing regime). By contrast, StreamScope asynchronously checkpoints state to remove the overhead of reliable persistence from the critical path, while keeping the probability of rollback low (akin to our lazy checkpointing regime).

Special cases of Falkirk Wheel have been demonstrated for batch [17, 20] and streaming [6, 23, 26] systems. These data processing systems have adopted fault-tolerance regimes that are subsumed by Falkirk Wheel. Going beyond previous solutions, Falkirk Wheel offers uniform support for multiple existing fault-tolerance regimes and permits selective rollback, which enables fault tolerance without compromising performance even in the presence of delayed events.

Other designs exploit the knowledge that some processors are deterministic to reduce rollback [14, 20]. Falkirk Wheel could integrate such techniques by extending the algorithm that computes rollback spans to account for determinism.

Several other streaming systems (e.g., Borealis [11], Aurora [1]) achieve fault tolerance by actively replicating processors [5, 24, 34]. These systems are designed for special-case applications that can be implemented in acyclic streaming dataflows, and thus cannot execute complex applications (e.g., our ad-serving application). However, extending Falkirk Wheel to support general forms of active replication is interesting future work. Such a fault-tolerance regime would provide another point in the trade-off between performance overhead in steady state and recovery latency. Actively replicated components would suffer little overhead in the steady state and would recover quickly, but would waste resources.

7 Conclusions

State-of-the-art data processing systems offer a single fault-tolerance regime so complex applications that combine different types of computation (e.g., batch, streaming, incremental) suffer performance degradation in steady state and during recovery due to the poor fit of the fault-tolerance regime.

In this paper, we propose Falkirk Wheel, a design for combining existing fault-tolerance regimes, and also accommodating new ones. In particular, our design permits selective rollback, which enables fault tolerance for incremental and iterative systems without compromising performance. By building an implementation of Falkirk Wheel, and through experiments that combine high-throughput batch computation, iterative incremental analytics, and low-latency queries, we show that Falkirk Wheel is able to reconcile fault-tolerance regimes, which reduces latency in steady state and during recovery.

References

[1] Daniel J Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *Proceedings of the VLDB Endowment* 12, 2 (2003), 120–139.

[2] Martín Abadi and Michael Isard. 2015. Timely Rollback: Specification and Verification. In *Proceedings of the NASA Formal Methods Symposium*.

[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. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. 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 (Aug. 2015), 1792–1803.

[5] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. 2013. Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD)*. New York, NY, USA, 577–588.

[6] Apache Software Foundation. Apache Flink. http://flink.apache.org.

[7] Apache Software Foundation. Apache Hudi. https://hudi.apache.org.

[8] Apache Software Foundation. Apache Kafka. http://kafka.apache.org.

[9] Apache Software Foundation. Apache Samza: A Distributed Stream Processing Framework. https://samza.apache.org.

[10] Apache Software Foundation. Storm: Distributed and Fault-Tolerant Realtime Computation. https://storm.apache.org.

[11] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. 2008. Fault-tolerance in the Borealis Distributed Stream Processing System. *ACM Trans. Database Syst.* 33, 1 (2008).

[12] Scottish Canals. The Falkirk Wheel. http://www.thefalkirkwheel.co.uk/.

[13] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.

[14] 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 (SIGMOD)*. 725–736.

[15] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems* 3, 1 (Feb. 1985), 63–75.

[16] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Tom Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. Benchmarking Streaming Computation Engines at Yahoo! https://yahooeng.tumblr.com/post/135321837876.

[17] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce Online. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*.

[18] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI) (OSDI)*.

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

[20] 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)*. 49–60.

[21] Jamie Grier. Extending the Yahoo! Streaming Benchmark. https://data-artisans.com/blog/extending-the-yahoo-streaming-benchmark.

[22] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC, Vol. 8)*.

[23] 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)*. IEEE Computer Society, Washington, DC, USA, 779–790.

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

[25] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys)*.

[26] Richard Koo and Sam Toueg. 1986. Checkpointing and Rollback-recovery for Distributed Systems. In *Proceedings of 1986 ACM Fall Joint Computer Conference* (Dallas, Texas, USA). IEEE Computer Society Press, Los Alamitos, CA, USA, 1150–1158.

[27] Wei Lin, 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 Symposium on Networked Systems Design and Implementation* (Santa Clara, California) *(NSDI)*. USENIX Association, 439–453.

[28] Frank McSherry. The Yahoo Streaming Benchmark. https://github.com/frankmcsherry/blog/blob/master/posts/2018-02-11.md.

[29] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Proceedings of the 6th Conference on Innovative Data Systems Research (CIDR)*.

[30] Microsoft. Azure Event Hubs. http://azure.microsoft.com/en-us/services/event-hubs/.

[31] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM.

[32] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: a Universal Execution Engine for Distributed Data-Flow Computing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[33] 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 (EuroSys)*. ACM.

[34] 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)*. ACM, New York, NY, USA, 827–838.

[35] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 147–156.

[36] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (Shanghai, China). ACM, 374–389.

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. 2012. 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)*.