Railgun: managing large streaming windows under MAD requirements

Ana Sofia Gomes
sofia.gomes@feedzai.com
Feedzai

João Oliveirinha
joao.oliveirinha@feedzai.com
Feedzai

Pedro Cardoso
pedro.cardoso@feedzai.com
Feedzai

Pedro Bizarro
pedro.bizarro@feedzai.com
Feedzai

ABSTRACT

Some mission critical systems, e.g., fraud detection, require accurate, real-time metrics over long time sliding windows on applications that demand high throughput and low latencies. As these applications need to run "forever" and cope with large, spiky data loads, they further require to be run in a distributed setting. We are unaware of any streaming system that provides all those properties. Instead, existing systems take large simplifications, such as implementing sliding windows as a fixed set of overlapping windows, jeopardizing metric accuracy (violating regulatory rules) or latency (breaching service agreements). In this paper, we propose Railgun, a fault-tolerant, elastic, and distributed streaming system supporting real-time sliding windows for scenarios requiring high loads and millisecond-level latencies. We benchmarked an initial prototype of Railgun using real data, showing significant lower latency than Flink and low memory usage independent of window size. Further, we show that Railgun scales nearly linearly, respecting our msec-level latencies at high percentiles (<250ms @ 99.9%) even under a load of 1 million events per second.

PVLDB Reference Format:
Ana Sofia Gomes, João Oliveirinha, Pedro Cardoso, and Pedro Bizarro. Railgun: managing large streaming windows under MAD requirements. PVLDB, 14(12): 3069 - 3082, 2021.
doi:10.14778/3476311.3476384

1 INTRODUCTION

In some mission critical systems, e.g., financial fraud detection, it is desirable that the underlying streaming engines fulfill our proposed M-A-D requirements:

Msec-level latencies at high percentiles (<250ms @ 99.9%);
Accurate sliding window aggregations event-by-event;
Distributed, scalable and fault-tolerant.

However, to the best of our knowledge, no streaming engine today delivers all three MAD requirements. Initial streaming engines such as STREAM [6], NiagaraCQ [16] or Siddhi [44] provide accurate sliding window aggregations per event, but do not scale beyond one node, nor do they comply with millisecond-level latencies. In opposition, state-of-the-art streaming engines such as Flink [11], Kafka Streams [18], and others [7, 33, 36, 39] provide scalability and fault-tolerance with low latencies, but at the expense of inaccurate sliding windows aggregations due to their window choices or load shedding [1], failing to meet **A**.

A critical decision is how to handle a *large streaming state* while delivering *low latency*. In low throughput and small windows, events can fit in-memory of a single node, and accurate aggregations can be computed for every new event over sliding windows. However, for large windows or high throughput (where **D** is required) handling the incoming *and* expiring events becomes such a problem that streaming engines either shed load, or use hopping windows as an *approximation* of real-time sliding windows, computing aggregations and expiring events only so often.

For instance, using hopping windows, a 5-min sliding window can be approximated, e.g., using five fixed physical 5-min windows, each offset by 1 minute (the *hop*) and where, as time passes, new windows (and their aggregations) are created and expired. Figure 1 illustrates this behavior of hopping windows, and how they might lead to inaccurate aggregations. At timestamp 5, there are exactly 5 active physical windows, h1-h5. When e_5 arrives, still within a 5-min window of e_1, window h1 is already expired and h6 created. Since h1 expires at timestamp 5, and h2 only starts at timestamp 2, both h1 and h2 only count 4 events, e1-e4 and e2-e5, respectively. This shows an example where a true 5-min sliding window (s0)

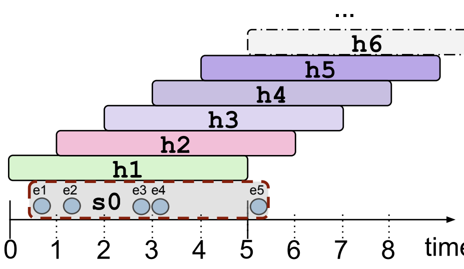


Figure 1: A 5-min hopping window with 1-min hop uses physical windows (h1-h6) but none capture the 5 events (circles) together, unlike a real-time sliding window (s0).

takes into account all 5 events for its aggregations when e_5 arrives, but a hopping window with 1-min hop does not. The hop could be made smaller, e.g., 1 second, but that would imply concurrently

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476384

managing 300 5-min physical windows, instead of 5. Additionally, since the frequency on which time slides is still fixed, a 1 second hop window might still not capture all 5 events together.

To keep latency low while dealing with high event throughput, state-of-the-art streaming engines use hopping windows in an attempt to save memory. Aggregations over real-time sliding windows require accessing all events to compute accurate aggregations, making these solutions low scale. Approximate aggregations can be done over hopping windows by discarding event tuples, but require handling the multiple aggregation window states, where the number of window states is defined by a ratio between the window size, and the hop size. As we shall see in Section 5.1, this tradeoff works until the hop size is much smaller than the window size. When windows start to span over multiple minutes, or hours, the window aggregation state becomes so large, that streaming engines either choose to reduce aggregation precision further (by using larger hops of minutes or hours), or deploy lambda architectures, to combine delayed results computed in batch with small real-time windows (see Figure 2).

This state of affairs presents a challenge for modern fraud detection systems. Responsible for processing trillions of dollars per year worldwide, these mission-critical systems have demanding latency requirements (e.g., <250 ms for 99.9% percentile), and still require accurate aggregation metrics event-by-event (for regulatory and adversarial reasons, see Section 2.1) over long windows. To address this need, we propose Railgun, a novel distributed streaming engine based on low-memory-footprint, disk-backed sliding windows (an improvement on the sliding windows and SlideM algorithm [34]) on top of which, we built state-management and distributed communication layers to fulfill all of the MAD requirements. Our contributions are as follows:

(1) We formulate the MAD requirements, supporting why they are needed in use-cases such as fraud-detection (Section 2.1);
(2) We present our proposal, Railgun, with an overview of the architecture, components and decisions (Sections 3 and 4);
(3) We illustrate how Flink degrades when small hops are used to approximate real-time sliding windows (Section 5.1);
(4) We show that Railgun computes real-time metrics over large windows in an efficient and scalable way (Section 5.2).

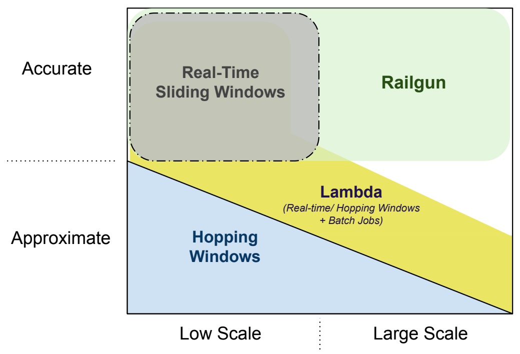


Figure 2: Approaches to manage *very large* streaming state to fulfill tail low latency.

(5) We show that Railgun scales nearly linearly, up to 50 nodes, even under a load of 1 million events per second, while respecting our latency requirement at high percentiles (Section 5.3).

2 BACKGROUND

A data stream S is an unbounded sequence of events $e_1, e_2, \ldots$, each with a timestamp. Aggregations over streams are computed using windows. A *window* w is a sequence of contiguous events of S with a certain size w_s (defined by a number of events, time interval, or start-stop conditions as in a user session). In this paper, we focus on time-based windows, henceforth referred simply as windows. As time passes, a window over a stream is evaluated often, at a specific, and changing, timepoint T_{eval}. T_{eval} determines the events to include for the aggregations, where an event with timestamp t_i belongs to a window evaluation iff $T_{eval} - w_s \leq t_i < T_{eval}$.

Hopping windows are windows where T_{eval} changes according to a step of length s. This step s, or *hop*, marks *when* new windows are created. If s is smaller than w_s, then the windows overlap, i.e., an event may belong to more than one hopping window[1]. When s is equal to w_s, hopping windows do not overlap, and events belong to exactly one window. This case is frequently given the name of *tumbling windows*. Step s is generally not bigger than w_s.

Real-time sliding windows, or just *sliding windows*, are windows where T_{eval} is the moment right after a new event has arrived. This frequent evaluation is computationally expensive as, for each new event e_i, the system has to expire events and (re-)compute aggregations, but on the other hand, aggregations are always accurate.

2.1 Fraud-detection requires MAD systems

Fraud-detection systems are responsible for, e.g., approve or block transactions, or raise alarms when money laundering is suspected.

As subcomponents of financial ecosystems, fraud-detection systems have very strict and demanding requirements, including all the MAD requirements defined above. It is easy to see why MD are required. Good customer experience implies a service that replies almost instantaneously (i.e., with sub 250ms latency at the 99.9% percentile, **M**), and which is available at all times even when processing several thousand of requests per second [5] (i.e., that is scalable and resilient to high-loads and failures, by being distributed, **D**).

Let's address accuracy (**A**) of sliding window aggregations. To make decisions, modern fraud-detection systems use machine learning models and rule based-systems, both fueled by streaming aggregations [10]. For instance, queries such as Q1 and Q2 below can be used to *profile* the common behaviors of card holders or merchants, and detect suspicious behavior.

Profiles computed over hopping windows are weaker as they are vulnerable to adversary attacks. Sophisticated fraudsters use many techniques to understand the best possible timings, and exploit attacks to occur at specific times, or follow a specific cadence, taking advantage of the predictable hop size.

```
Q1: SELECT SUM(amount), COUNT(*) FROM payments
    GROUP BY cardId [RANGE 5 MINUTES]
Q2: SELECT AVG(amount) FROM payments
```

[1]Hopping windows are often called sliding windows by systems such as Flink because they *approximate* the behavior of real-time sliding windows.

```
    GROUP BY merchantId [RANGE 5 MINUTES]
```

Example 1: Streaming 5-min metrics per card and merchant.

Additionally, profiles over hopping windows lead to inaccurate and counter intuitive results, compromising rule compliance – either from internal bylaws, or from external regulators. As illustration, consider the following business rule: "*if* the number of transactions of a card in the last 5 minutes is higher than 4, *then* block the transaction". If the window is implemented using 1-min hops, then the situation in Figure 1 can happen: the rule should trigger on the fifth event since it arrives within 5 minutes of the first one, but there is no hopping window including all 5 events in its boundaries using a 1-min hop.

To avoid this, one could argue that the hop could be adjusted to catch the intended behavior. However, the solution is not a panacea. First, the problem in Figure 1 can happen regardless of the hop size. Second, if the hop is reduced to a size where hopping windows behave almost like real-time sliding windows (e.g., 1-ms or even 1-sec step) then most stream processing engines systems crash or significantly degrade performance (cf. Section 5.1). This problem worsens with long windows. Fraud profiles use windows spanning over days, weeks, months and sometimes years. These include, e.g., the number of distinct addresses used in the last 6 months, or the average user's expenditure of the past year. As we shall see in Section 2.2, the performance of hopping windows depends on a ratio between the window size and the hop size. Hence, the longer windows are, the lower the precision must be, to achieve the same performance in terms of CPU, memory consumption and latency.

If streaming aggregations use only hopping windows and have metrics over long windows, then fraud systems need to use batch jobs and lambda-architectures [29]. In that case, imprecise but real-time aggregations are combined with precise but outdated aggregations over complex pipelines which are costly to maintain and hard to debug. Because of this, a lot of work has been committed to integrate batch and streaming in the same language and platform [4, 11, 13]. Nevertheless, in these systems, compliance is not achieved in real-time, limiting the possibility of preventing fraud from happening, and be restricted to use-cases where a post-mortem alarm is useful. Real-time 100% compliance (i.e., accurate metrics per-event, **A**) is only possible using real-time sliding windows.

2.2 Related Work

The first generation of stream processing engines has risen from the database community. These include seminal systems such as STREAM [6], TelegraphCQ [14], NiagaraCQ [16], Aurora [3], and later, Borealis [2], Coral8 [45], Event Insights [35], Siddhi [44], Spade [23], Trill [13], Truviso [22], and SAP ESP [53]. While a few of these systems implement real-time sliding windows [6, 13, 16, 44], almost all of them run as a single-node (Borealis and Spade are noteworthy exceptions), and they do not address the aggressive low-latency requirements we have.

On the other hand, the latest generation of stream processing engines focus on processing high throughputs under low latency requirements, by building scalable, fault-tolerant, distributed systems. Examples of these include Flink [11], Kafka Streaming [18], Spark Streaming [7], and others [32, 33, 36, 37, 39, 47]. However, to achieve high levels of performance, these systems need to limit what windows are possible. Neither Flink, Kafka Streaming, Spark Streaming, or to the best of our knowledge, any known, distributed streaming engine implements real-time sliding windows, restricting the window slide movement to fixed hops.

So why are hopping windows so largely used? Since the window size and hop size do not change during run-time, the number of physical window states active at any given time is fixed and exactly $\frac{\texttt{windowSize}}{\texttt{hopSize}}$. This allows streaming engines to do important optimizations, such as *avoiding* to store events. Since the number of active window states is fixed, arriving events can be discarded once their contribution has been applied to all the active windows states. Hence, besides saving storage, these solutions also avoid processing event expiration. As an example, recall Q1 and Figure 1: the sum and count payments made by a card in the last 5 minutes with 1-min hop. Any event for this window affects 5 window states, and in this case, two variables [2] per window state. Every minute and, for every card active in the last 5 minutes, two new variables are created and the oldest two, expired.

Since their memory requirements are independent of throughput, this property makes hopping windows interesting as long as the ratio $\frac{\texttt{windowSize}}{\texttt{hopSize}}$ is low. When the ratio is higher, hopping windows bring extra problems with respect to latency, CPU usage and state scalability. For instance, a 60-min window with a 5-min hop implies 12 *active* window states per metric; but if the hop is decreased to 1 second, for the same window size, the number of active window states becomes 3600. All of which must be updated per arriving event, and per metric. The situation becomes unsustainable especially on large windows ranging hours or days, unless the hop is adjusted accordingly, with severe consequences on the usefulness of the aggregations computed. The impact of using small hops on large windows is further explored in Section 5.1.

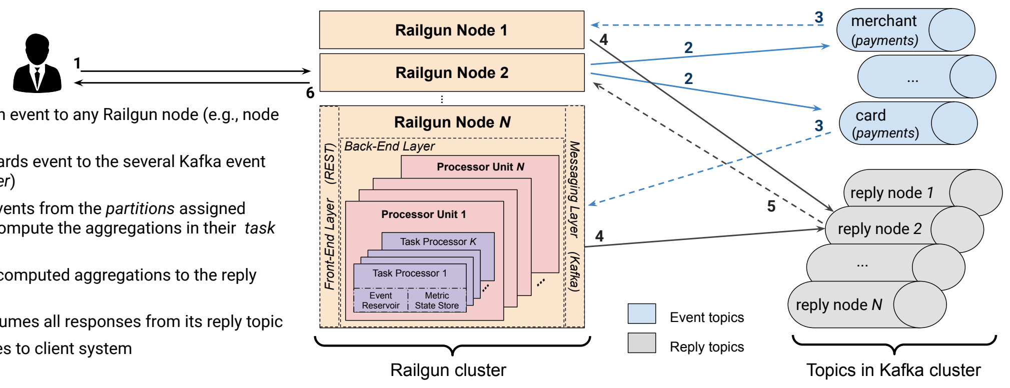
Despite these drawbacks, hopping windows' wide usage and characteristics have driven substantial research and optimizations such as Cutty [12], Scotty [48] and others [9, 46] that contribute to its popularity by delivering reduced hardware costs, support for out-of-order events, and distributed computations of a single query.

Unlike hopping windows, real-time sliding windows cannot discard events and therefore require storing and accessing events. For each event, metrics must be updated with the events exiting and entering the window. Since the window slides for every event, and not just at a specific hop, the optimizations discussed above for hopping windows are no longer possible.

Flink acknowledges the issue of high-precision metrics over time windows for low latency fraud-detection, with a customized solution [21]: for each event, the solution computes each aggregation from scratch by iterating over all stored events (persisted in RocksDB) for those matching the window interval. This approach has quadratic performance, and since Flink was not designed to store events and manage event expiration, few optimizations are possible and performance degrades with long windows. Hence, this solution has much worse performance than Flink's standard hopping windows (compared in Section 5.1), failing our **M** requirement.

The interested reader can further explore related issues with stream processing engines elsewhere [8, 24, 25].

[2] one variable for the sum and another for the count



1. client system sends an event to any Railgun node (e.g., node 2 *front-end layer*)
2. node 2 front-end forwards event to the several Kafka event topics (*messaging layer*)
3. processor units poll events from the *partitions* assigned (*back-end layer*) and compute the aggregations in their *task processors*
4. processor units send computed aggregations to the reply topic of node 2
5. node 2 front-end consumes all responses from its reply topic
6. node 2 front-end replies to client system

Figure 3: Tale of an event traversing Railgun.

3 RAILGUN

Railgun is the paper's main contribution, and takes different design decisions when compared to the alternatives above: 1) works with real-time sliding windows to achieve aggregation correctness at all times (and not just at every hop); 2) uses an *event reservoir* to efficiently store and access events under low latency and optimal memory usage; 3) manages an embedded aggregation store (persisted in RocksDB) for holding aggregation states and auxiliary data; and 4) takes advantage of a messaging layer (using Kafka) for distributed processing, fault-tolerance and recovery.

The event reservoir, described in Section 4.1.1, exploits the predictable, time access pattern of events to optimize transfers between memory and stable storage, accessing nearly all events from memory using an eager caching. Plus, by optimizing the computation and storage of aggregation states, Railgun can deliver accurate results, per-event, with low latency. This event reservoir is an evolution of previous work [34]. Here, we use a locally-attached storage in each Railgun node to minimize latency, a schema registry to support event schema evolution, and define a data format and compression for efficient storage, both in terms of deserialization time and size.

To distribute work and achieve scalability, Railgun uses Kafka topics, further split into *(topic, partition)*. Each stream can have multiple topics, depending on the combination of the metrics' *group by*s, and each topic has multiple partitions which are distributed among the several nodes' processor units. A *(topic, partition)* maps to a *task* in Railgun, and is its minimal unit of work. Inside a task, we compute all aggregation metrics for a data stream subset, following a task *plan* optimized to reuse computations. To support high-availability and fault-tolerance, tasks have multiple *replicas*, and a Railgun node processor concurrently handles a set of active tasks and a set of replica tasks. In addition, Railgun relies on Kafka's consumer group guarantees to ensure that tasks are always assigned to nodes, and provides a custom rebalance strategy (see Section 4.2) to optimize task recovery, while safeguarding a balanced assignment from tasks to nodes.

Railgun delivers event-by-event accurate results, by supporting real-time sliding windows, while still providing millisecond-level latencies at high percentiles. As we shall see in Section 5, Railgun can preserve these consistent tail latency results even when its scaled to achieve throughputs of one million events per second, and its performance is independent of the window size.

Railgun's high-level architecture is presented in Figure 3, showing what happens when an event traverses the system. At this stage, and to simplify development, all Railgun nodes are equal and composed by layers: a *front-end* layer to communicate with the client; a *back-end* layer to compute aggregations and access storage; and a *messaging* layer to handle distribution of tasks, detect failures, and communication between processor unit workers. This design could be revised in the future, with different nodes split by function.

3.1 Front-End Layer

The front-end layer is the entry point for client requests, including events, requests for new metrics/streams, or deletions. Besides communicating with the client, the front-end layer distributes events and manages the overall cluster state (both using Kafka).

When a new stream is registered by the client, the front-end creates a set of partitioned topics to support it. The number of topics needed per stream depends on the number of distinct *group by* fields of the stream. As further detailed in Section 4, a stream is mapped to one or more topics to support work distribution across the several processing units. Hence, when a new event arrives (step 1 of Figure 3), it is the front-end layer responsibility to route events to all of its topics (step 2). E.g., in case of Example 1 and Figure 3, to simultaneously publish any event of stream `payments` to topics `merchant` and `card`, as they are *group by* aggregations for both `merchantId` and `cardId` (`Q1` and `Q2` of Example 1).

The metrics of a stream are computed by one or more back-end instances possibly residing in different Railgun nodes (step 3), which reply to the node originally posting the event in its dedicated reply topic (step 4). The front-end is also in charge of collecting the several computations (step 5) from its reply topic, and responding to the client with all the aggregations computed for that particular event in a single message (step 6).

3.2 Back-End Layer

The back-end layer is responsible for the computation of metrics. Each back-end instance has one or more *processor units*, each with its own dedicated thread. A processor unit manages a set of *tasks*, all computed within a single thread, to reduce context switching and synchronization, thereby optimizing for latency.

Importantly, each processor unit is completely independent of each other, and two processor units deployed on the same physical Railgun node are logically equivalent to two Railgun nodes with one processor unit each. As such, by having many processor units inside a single node, we can exploit multi-core machines efficiently. Notwithstanding, and as we shall see in Section 4.2, distributing processor units among multiple physical nodes can bring advantages in terms of fault-tolerance, work rebalance and high availability.

Algorithm 1 Processor Unit Logical Loop

```
1:  while running do
2:      processOperationalRequests(requests)
3:      activeMessages ← consumerActiveTasks.poll()
4:      replicaMessages ← consumerReplicaTasks.poll()
5:      for message ← activeMessages ∪ replicaMessages do
6:          t ← message.topic
7:          p ← message.partition
8:          taskProcessor ← taskProcessors.get(t, p)
9:          answer ← taskProcessor.processMessage(message)
10:         if (message ∈ activeMessages) then
11:             sendReply(answer)
12:         end if
13:     end for
14: end while
```

The processor unit duties are listed in Algorithm 1. While running, the processor handles operational requests (such as adding/removing new streams or metrics); consumes message events for its assigned (active and replica) *tasks*; forwards the events to their appropriate *task processors* which handle event storage and task computation; and replies with the computation answer to a dedicated reply topic, for active tasks. Each processor unit is then both a consumer of event topics (inbound stream events), and a producer for reply topics (outbound aggregation results).

A task encompasses the computation of all metrics associated with a given *(topic, partition)*. A *(topic, partition)* is the unit of work distribution among nodes and processor units. As we shall see in Section 3.3, events are routed, consumed and processed according to their *(topic, partition)*. Processor units have *active* tasks, for which they are the leaders, and *replica* tasks for which they are hot standbys. To poll messages from the messaging layer, the processor unit has two consumers, for each type of task. Separating the consumers allows us to prioritize active tasks, and better exploit Kafka rebalance protocol and consumer group guarantees (cf. Section 4.2).

Finally, processing message events and computing metrics for any task happens within a *task processor*. Each processor unit has as many task processors as (active or replica) tasks it has assigned, all computed within a single thread. Thus, while the number of processors units sets the cluster's level of parallelism, the number of task processors in Railgun establishes the cluster's level of concurrency.

3.3 Messaging Layer

The goal of the messaging layer is many-fold: 1) serve as the communication layer between different Railgun nodes, including between the front-end and back-end layer of the same physical node; 2) detect Railgun node failures, during the polling of messages; 3) support the recovery of Railgun node failures, by reliably storing events and aggregation replies which can be rewinded upon request.

Currently, Railgun uses Kafka [31]. Kafka is a distributed, highly scalable and fault tolerant messaging system, with high throughput and low latency guarantees. In opposition to push-based systems such as RabbitMQ [43], Kafka follows a pull-based approach where consumers continuously poll for new messages by providing their individual offset since the last poll. This is important since it allows a Railgun node to recover by rewinding the stream and replaying unprocessed messages without degrading the end-to-end latency of the overall system. Kafka stores messages in *topics* and provides built-in capabilities to split topics into several *partitions*, for higher throughput and parallelism. As described further in Section 4, Railgun takes advantage of Kafka's partitions to distribute work among the several Railgun processor units and their task processors.

Except with the client, all communication happens using Kafka, and Railgun nodes are, simultaneously, Kafka producers and consumers. Railgun nodes communicate for many reasons: 1) to broadcast operational requests triggered by the client, such as creating/deleting a stream, metric or partitioner; 2) to propagate events to all of their topics; 3) to share aggregation partial results which are collected before answering the client; 3) to handle cluster maintenance tasks such as node removal/addition and the corresponding rebalance of tasks in the cluster.

Although we could have chosen any other messaging system (as long as it implements the same pull-based and partition concepts), we choose Kafka due to its proven performance, and for providing us with many features that simplify our development. First, Kafka is actively monitoring what consumers enter or leave the cluster, requiring every consumer to send heartbeats periodically and assuming consumer failure, otherwise. Therefore, whenever the consumer landscape changes, Kafka detects this and, at step 3 of Algorithm 1, triggers a callback to rebalance the cluster. At this moment, Railgun's assignment strategy (described in Section 4.2) takes over, e.g., to decide how to reassign the multiple *(topic, partition)* – or tasks – of a failed node.

Second, in Kafka, consumers can be organized in *consumer groups*, to allow load distribution among a group of several consumers, with important guarantees. Namely, by design, Kafka ensures that there is exactly one consumer within a consumer group subscribing and consuming messages from a *(topic, partition)*. We take advantage of this property to ensure that there is exactly one processor actively responsible for a task, by configuring all Railgun active task consumers to belong to the same consumer group. On the other hand, replica task consumers all have different consumer groups to enable multiple Railgun processors to subscribe to the same *(topic, partition)*. Therefore, we ensure high-availability by having multiple hot replicas available in the cluster.

Finally, we ensure exactly-once semantics by combining Kafka's at-least-once guarantees, with an event deduplication logic on the application back-end layer (see Section 4.1.1).

3.4 Railgun Operators

Railgun's language is summarized on Figure 4. We support SQL-like query statements, where each statement can include multiple aggregations over a single stream.

Currently, Railgun does not natively support stream joins. In practice, we implement joins (e.g., between a stream and a lookup table) prior to the streaming engine, in an enrichment stage.

Besides sliding and tumbling windows, we support infinite windows, i.e., windows where events never expire (e.g., the count of all distinct addresses of a client). Any window can be delayed, i.e., where instead of considering the window against the latest arriving event, we can delay its starting by a specific delay offset. Delayed windows are especially useful in bot-attacks scenarios. We choose not to support hopping windows, since we see them as an approximation of our sliding windows. In practice, we never found a use-case where hopping windows are functionally preferable to sliding windows. Although, at this time, we only support time windows, the system could be easily extended to support windows whose size is based on the number of event.

Finally, we use jexel expressions [17] as our filter expression language to support additional flexibility, using Java.

4 COMPUTATION AND DISTRIBUTION

The *(topic, partition)* combinations affect how work is distributed among the several Railgun nodes, and processor units. Each data stream has a topic for each configured top-level entity, which we call *partitioner*. For instance, Figure 3 shows two topics for two partitioners - `merchant` and `card` - over the same `payments` stream (corresponding to the *group by*s for `merchantId` and `cardId` of Example 1). Currently, the set of partitioners is manually provided by an administrator when a stream is created, depending on the possible *group by*s of the metrics. Since computation is contained within a task processor, to provide accurate metrics, we need to ensure that whenever a task processor is computing a metric for an entity (e.g., of a particular card, or merchant), it receives all that entity's events. As a result, metrics with multiple *group by*s over a stream, as in `Q1` and `Q2` of Example 1, may cause the event to be forwarded to more than one topic (step 2 of Figure 3). Events are, in fact, replicated as many times as the number of partitioners (i.e., top-level *group by*s such as `cardId` or `merchantId`) needed for a stream, resulting in a few topics per stream.

Notwithstanding, the number of topics needed is usually small, and it is not necessarily equal to the number of distinct group by keys of all stream metrics defined (which could lead to dozens of topics). Accurate metrics, only need events to be hashed by a *subset* of their group by keys. E.g., two metrics, one grouping by `card` and `merchant`, and the other by `card`, could both use topic `card`. This reduces Kafka's storage, and thus the front-end receives, from configuration, the partitioners for a given a stream upon stream creation. Partitioners can also be set after a stream is created, but this causes the creation of new *(topic, partition)* and a consequent cluster rebalance, which can be an expensive operation. However, and as we shall see in Section 4.2, our rebalance strategy is sticky, i.e., it preserves task assignment to their previous processor as much as possible. As a result, the processing of the existing *(topic, partition)* of the cluster is generally unaffected when a rebalance is triggered for adding new topics. Plus, adding a new partitioner is done only when a new top-level *group by* is needed which, in practice, is rarely required after a stream is created.

Finally, a *partition* is a Kafka concept that further allows us to distribute work among several consumers (i.e., processor units). For partitioning, Kafka allows producers to provide a key when publishing a message, which is hashed according to the number of partitions defined for a topic. When a key is provided, it is guaranteed that messages with the same key will always be delivered to the same *(topic, partition)*. In Railgun, we configure the message key for each topic to be the *partitioner*. When a new event arrives for a stream, the front-end layer node receiving the event publishes as many messages as partitioners defined for that stream.

The number of partitions for each topic is defined according to the expected load of each stream-partitioner. Recall that the *(topic, partition)* is the minimal work unit, and the distinct number of *(topic, partition)* establishes the number of task processors created in Railgun, where each task processor handles a single pair of *(topic, partition)*. Hence, by increasing the number of partitions, we increase the cluster's level of concurrency. As described, by exploiting Kafka's guarantees over consumer groups, we ensure there is exactly one *active* task processor for each existing *(topic, partition)*. However, to support high-availability, the number of task processors is multiplied by the replication factor. If there are n distinct *(topic, partition)*, and r is the replication factor, there are exactly $n \times r$ task processors working in the cluster.

```
    SELECT AggExpression FROM streamName
    WHERE filterExpression
    GROUP BY fields
    OVER WindowExprESSION

AggExpression    ::= Aggregation(field)      |
                   Aggregation(field), AggExpression

Aggregation      ::= count                   |
                     sum                     |
                     avg                     |
                     stdDev                  |
                     max                     |
                     min                     |
                     last                    |
                     prev                    |
                     countDistinct

WindowExpression ::= TimeWindowExpr          |
                     TimeWindowExpr delayed by offset

TimeWindowExpr   ::= sliding windowSize      |
                     tumbling windowSize     |
                     infinite
```

Figure 4: Railgun Operators.

4.1 Task Processors

The computation of *all* metrics for a given *(topic, partition)* encapsulates a *task*, which is done within a *task processor*. Each task processor is designed to share nothing, and work independently of other task processors, without the need to synchronize or access shared storage. To support this, each task processor is further composed of: an *event reservoir* that stores its own events; a *state store* holding aggregation states of each configured metric; and an execution task *plan* – i.e., a directed acyclic graph (DAG) defining how metrics will be executed.

4.1.1 Event Reservoir. The event reservoir is a structure that stores all the events of a task processor, and allows efficient access of the events as they are needed by windows to update the aggregations. The event reservoir is an evolution of previous work [34], and has two parts: a very small memory part holding the tail and head of each window, and a potentially large part stored in the node's local disk holding the full set of events.

Processing an event starts with the event reservoir, where events are persisted to and loaded from disk as needed. Before persistence, events are serialized and compressed into groups of contiguous *chunks*. Grouping events into chunks helps to reduce the number of I/O operations needed. In a reservoir, all I/O operations are asynchronous, to not affect event processing latency. Chunks hold multiple events and are kept in-memory until they reach a fixed size, after which they are *closed*, serialized, compressed, and persisted to disk over *ordered* and append-only files. Similarly, files hold multiple chunks of events, until they reach a fixed sized, after which they become *immutable*. Since files are immutable and events follow a monotonic order given by their timestamp, we can efficiently support random reads by maintaining an auxiliary index in-memory, from timestamps to files. Supporting random reads is especially useful when adding metrics with new windows to the system.

Since chunks are frequently persisted to disk, recovery is simplified, as only the most recent events can be lost, and quickly recovered from Kafka broker nodes.

Out-of-order events are supported until the closure of a chunk, i.e., as long as the event timestamp occurs after the last closed chunk timestamp. After that moment, and depending on the configuration, events are either discarded, or have their timestamp rewritten to the first timestamp of the chunk. For scenarios requiring extensive support for out-of-orders events, we can delay the chunk closure by a time period provided by configuration. This keeps chunks in a transition state in-memory for a threshold period, on which they are closed for recent event, but are still open for late events. In a way, this configuration can be seen as a watermark [49], which should be used with care, as it may cause an increased memory consumption, and recovery delays. However, since latency is a prime goal of our system, we might delay the closure of a chunk, but we never delay the answer and computation of event metrics, as opposed to systems such as Spark Streaming or Flink. Events are also deduplicated based on an *id*, against the chunks still in-memory, to avoid processing an event more than once.

The reservoir takes advantage of the predictable event consumption pattern in stream processing where events are *always* consumed by their timestamp order, by advancing windows. Namely, the reservoir provides very efficient *iterators* which transparently load chunks of events into memory as they are needed by windows. Iterators eagerly load adjacent chunks into cache when a new chunk is loaded from disk, and starts to be iterated. Hence, when a window needs events from the next chunk, the chunk is normally already available for iteration. Notwithstanding, if for some reason, chunks are evicted from the application cache right before they are requested, thus resulting in *syscall* to fetch them from disk,

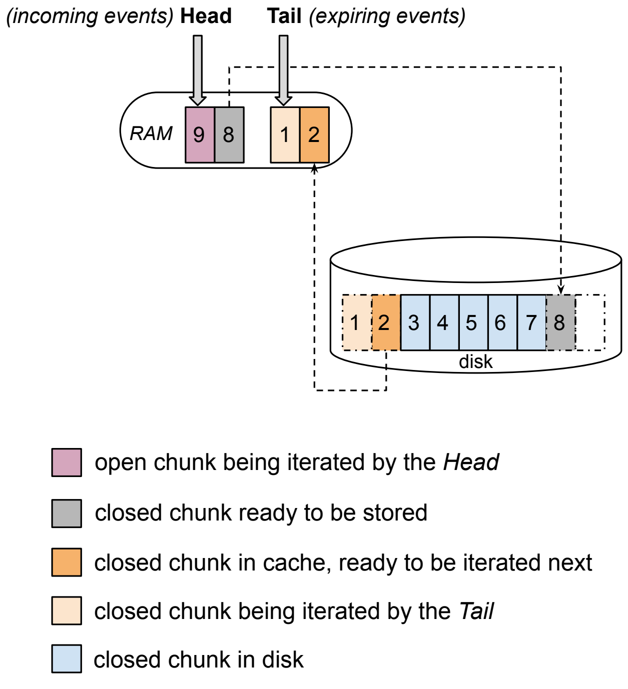

(incoming events) Head
Tail (expiring events)
RAM
9
8
1
2
1
2
3
4
5
6
7
8
disk
open chunk being iterated by the Head
closed chunk ready to be stored
closed chunk in cache, ready to be iterated next
closed chunk being iterated by the Tail
closed chunk in disk

Figure 5: Iterators for a Window in Event Reservoir.

the request will likely not trigger an actual read request to disk. Since chunks are organized as a sequence in a file, the operating system I/O will likely already read ahead the chunk contents into page cache. Thus, when a chunk not in cache is requested, it is likely delivered from the OS page cache, paying only the deserialization cost – a fraction of what it would be if an actual I/O request to disk was required. This predictability helps us relax the hardware demands for the reservoir tremendously, as even for low latency scenarios, we can use a network-attached storage or HDDs, instead of holding all events in memory, which significantly reduces the total cost of ownership.

Along with a reservoir, we keep a *Schema Registry* to support schema evolution of events. Before persisted, chunks are serialized using a specific events' schema and stored referencing their current schema id. Each time the event schema changes, a new entry is added to the schema registry, and the current schema id reference is updated. Whenever we need to deserialize a chunk with an old schema, we just retrieve it from the schema registry. Chunks are also compressed aggressively to guarantee a good compression ratio. This is important to minimize storage overhead, since events can be replicated across multiple task processors.

Regardless of the window type and window size, only a tiny fraction of events need to be kept in-memory, as illustrated by Figure 5. By default, each window has two iterators – one for the head of the window (incoming events), and another for the tail (expiring events) – and each iterator only needs one chunk in-memory[3]. Whenever possible, we reuse iterators among windows.

[3]However, due to eager caching, more chunks may be in-memory.

For instance, over the same reservoir, two real-time sliding windows always share the same head iterator (e.g., a 1-min and a 5-min sliding window share the same head iterator, which points to the most recently arrived event). This design makes the reservoir optimal for I/O [41], and extremely efficient for long windows. Namely, and except for the extra storage needed (minimized by compression and serialization), windows of years are equivalent to windows of seconds – in performance, accuracy, and memory consumption.

4.1.2 Task Plan. The task plan is a DAG of operations that compute all the metrics of a task, following the order: `Window -> Filter -> Group By -> Aggregator`. Since we often have metrics sharing the same `Window`, `Filter`, and `Group By` operators, the plan optimizes these by reusing the DAG's prefix path.

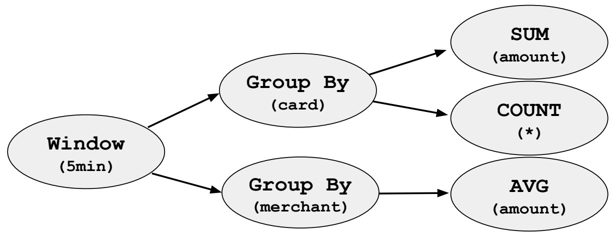


Figure 6: Plan DAG of Example 1.

Figure 6 shows the DAG of Example 1. In it, all metrics share the same window, but `Q1` groups by field `card` while `Q2` by field `merchant`. Optimizing the DAG to reuse operators prevents us from repeating unnecessary computations, especially ones related with windows. Every time a plan advances time, the `Window` operator produces the events that *arrive* and *expire*, to the downstream operators of the DAG. However, to make these optimizations, we restrict Railgun's query expressibility to follow a strict order of operations, defined in Section 3.4. This is in contrast to general solutions such as Flink or Spark Streaming, which provide a more flexible API, harder to optimize.

While the roots of the DAG iterate over the reservoir and push events downstream, the leafs (i.e., `Aggregator` operators) use the *state store* to keep and access the results of the aggregations.

4.1.3 State Store. Similar to other streaming engines such as Flink, Railgun uses RocksDB [20] to store, for each metric key value, the latest aggregations results and auxiliary data. Built on top of LSM-trees [38], RocksDB has proven to be a reliable, memory efficient and low latency embedded key-value store.

The amount of RocksDB keys, and their access pattern is tightly related with the task plan. Namely, each key represents a particular metric entity in a plan, and the amount of keys accessed per event match the number of DAG's leaves of a plan. For instance, for each event, the plan of Figure 6 will access exactly two keys for the card aggregations (`sum` and `count`), and one key for the merchant (`amount`). Each key holds the aggregation current value for the specific window and the specific entity. Depending on the aggregation type, auxiliary data might be stored with the aggregation. For instance, an `average` requires storing also a counter, while a `sum` or a `count`, do not require any extra data other than the current value. Additionally, `max` and `min` store a deque structure [30], the `stdDev` stores the three parameter to compute the Welford's online algorithm [50], whereas the `countDistinct` uses an auxiliary column-family in RocksDB to hold the counts.

To support fault-tolerance, RocksDB provides checkpointing, which forces the flushing of all data in-memory to disk. However, by design, even without checkpoints, RocksDB data is only kept in-memory for a short period of time, and is frequently persisted to disk. This makes checkpoints very efficient, since only a small amount of data needs to be written to disk, at a given time. We synchronize checkpoint triggers among the event reservoir and the state store, and references to the latest event checkpoint offset of each task processor and node are frequently stored in a dedicated Kafka topic, which allows us to ensure that both stores can be easily recovered during a failure.

4.2 Rebalance and Recovery

The assignment of tasks to nodes and processor units is triggered during a Kafka rebalance, which happens whenever nodes or tasks are added/removed from the cluster.

As previously mentioned, Kafka tracks consumers within each consumer group to guarantee load distribution and message delivery. Kafka consumer group protocol ensures each *(topic, partition)* has exactly one consumer assigned in a group. In particular, it is impossible to have a *(topic, partition)* assigned to multiple consumers of the same group, and if there are more consumers in a group than *(topic, partition)* combinations, a consumer might not have any *(topic, partition)* assigned. To achieve this, Kafka is continually tracking what consumers are registered for a consumer group, and is actively receiving heartbeats for each consumer. When a consumer enters or leaves (either due to a failure or graceful shutdown) a Kafka consumer group, a rebalance happens.

When a rebalance is triggered, one of the Railgun nodes (the consumer group coordinator) decides how *(topic, partition)* pairs are distributed among each consumer. While Kafka makes available several different strategies to assign *(topic, partition)* to consumers, Railgun uses a custom assignment strategy, built upon Kafka's sticky assignment implementation. The assignment strategy logic is shown in Figure 7 and is split into two main assignments: *active* tasks, and *replica* tasks.

Recall that a task maps to a specific *(topic, partition)*, and that consumers (located within processor units) can have tasks assigned as active and as replicas. This distinction between active and replica tasks has more to do with Kafka consumer groups, than with computation. A processor unit will process messages from both active and replicas, and compute their aggregations in the same way. The only difference is that messages from replica tasks will not trigger a response from the processor, since this responsibility is exclusive of active consumers. While active consumers share the same consumer group, replica consumers have different consumer groups from active and other replica consumers. This allows us to assign a *(topic, partition)* to a single active consumer in the cluster, but also to multiple replica consumers, simultaneously. Since both active and replicas tasks consume messages from the same *(topic, partition)*, they always consume them on the same order, ensuring consistency on the reservoir and metric state store for the several replicas.

Achieving a perfect assignment of tasks is an NP-complete problem as it implies solving a search over multiple-goals [28]: 1) minimize recovery time by minimizing data shuffling; 2) respect the cluster balance for active/replica tasks; 3) distribute the load fairly among the processors. In practice, our assignment strategy logic, shown in Figure 7, implements a greedy approach that always protects two invariants: 1) tasks are only assigned to a physical node once; 2) the load respects a predefined processor *budget*.

The first invariant aims to avoid the loss of multiple task copies when a node fails, or it is decomissioned. Hence, while metric computation within each task processor is agnostic to where processor units are located, the assignment strategy is not. Consequently, the strategy takes as input the locality of each processor, to ensure that a physical node will never be assigned the same task twice during the same a rebalance iteration assignment.

Another constraint relates with how load is distributed among the several consumers. To ensure that load is fairly distributed among the cluster, for each assignment, the strategy sets the maximum *budget* of each processor unit as: $budget = \frac{tasks}{processor\ units}$. Each time a rebalance is triggered, the available budget of a node is reset to this value. Whenever a task is assigned to a processor, the available budget of a node is decremented by 1. When its budget reaches 0, the processor can no longer receive assignments. At this stage, we consider all tasks as equal, however, in the future we might want to give tasks a different weight, depending on their computational cost (viz., partition load, event reservoir size, etc.).

To ensure these two invariants, upon each new rebalance iteration assignment, the group coordinator collects cluster metadata to understand how many tasks, physical nodes and processor units exist, and how processors are located within each physical node.

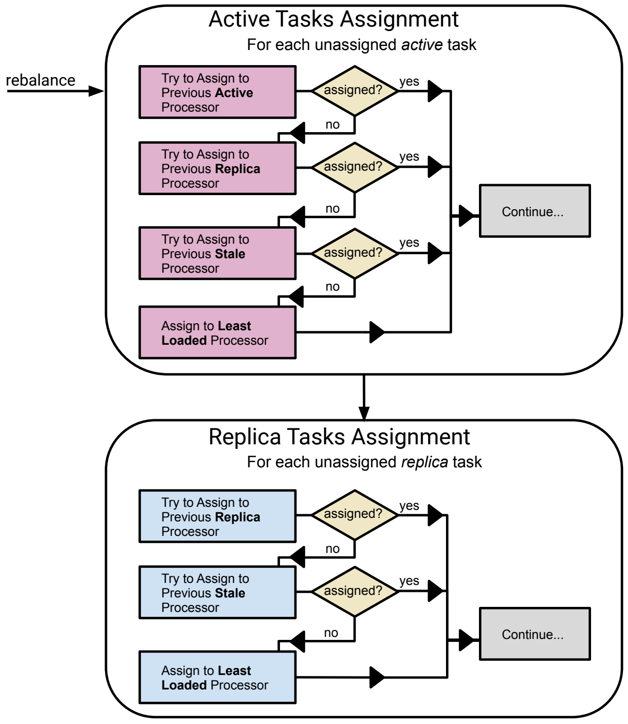

Active Tasks Assignment
For each unassigned *active* task
rebalance
Try to Assign to Previous **Active** Processor
assigned?
yes
no
Try to Assign to Previous **Replica** Processor
assigned?
yes
no
Try to Assign to Previous **Stale** Processor
assigned?
yes
no
Assign to **Least Loaded** Processor
Continue...
Replica Tasks Assignment
For each unassigned *replica* task
Try to Assign to Previous **Replica** Processor
assigned?
yes
no
Try to Assign to Previous **Stale** Processor
assigned?
yes
no
Assign to **Least Loaded** Processor
Continue...

Figure 7: Railgun Sticky Assignment Strategy.

Regardless of the task's type, the goal of the sticky assignment strategy is to avoid data reshuffling as much as possible, while respecting the two invariants above. Therefore, the first step of the algorithm is to always try to maintain the task in the consumer that had the task in the previous iteration. An assignment might fail, if we violate one of the two invariants.

Active tasks that can not be kept in the same processor will be attemptedly assigned to a processor previously holding one of its previous replica tasks. If more than one processor replica is available for assignment, we choose the one with the least load. If a task cannot be assigned to a processor replica (because of the conditions mentioned above), we will try to assign the task to a processor that still has the task as *stale*. A stale task is a task for which the processor used to be assigned in the past (either as active or replica), but lost its assignment during a rebalance. In other words, processors with stale tasks are processors that still have data "leftovers" available for that task. Hence, assigning a task to one of its stale processors, only requires recovering a *subset* of the data, instead of the whole data. Again, in case of ties, we choose the processor with the least load. In the future, we might consider to give priority to the processor that has more data available (i.e., the processor that needs less data shuffle to recover).

Lastly, if none of the assignments is possible, we simply assign the task to the consumer with the most available budget.

This assignment strategy, in combination with a replication factor, allows us to achieve high-availability. When a node fails and a rebalance is triggered, active tasks are always the first tasks assigned, maximizing the probability to be allocated in nodes already holding that task. In this case, the processor does not need to recover any data and the task is recovered immediate.

When a task is assigned to a processor that was not actively processing it before (either as active or replica), a recovery process happens within that processor, which might affect these tasks' immediate availability. However, since we prioritize the assignment of active tasks over replicas, this is extremely unlikely to happen for active tasks. As usual, the replication factor is set according to the number of failures we want to tolerate before affecting a task's availability. In practice, in most of our deployments we use a replication factor of three.

To perform recovery, the processor triggers a request to another processor unit that still has data available – to copy the event reservoir, the state store, and the last event offset since its last checkpoint. After data is transferred, the processor starts its execution by consuming messages from Kafka since the last checkpointed offset. Importantly, a processor with stale data, only needs to copy the delta between its own last checkpoint and the newest checkpoint available in the cluster, thereby minimizing the time to recover.

5 EXPERIMENTS

To validate our approach we present three experiments, in order to: 1) measure how Railgun's real-time sliding windows compare with the performance of Flink's hopping windows; 2) assess how Railgun's latencies are affected with different window sizes and a growing number of windows; 3) measure how Railgun can be scaled to handle more load by adding more nodes to the cluster.

All of our experiments focus on tail latency, i.e., the latency of the system at the high percentiles of latency distribution. Tail latency measures how stable and consistent a system is, and it is one of our core Service Level Objectives (SLOs) contractualized with our clients. Addressing tail latency is hard for multiple reasons. First, tail latency is very hard to debug, as it can be affected by multiple types of resources, either individually or by its combination, including disk storage differences, garbage collector configurations, or network bandwith. Second, in a distributed setting, tail latency is impacted by its slowest component [19], which in our case could mean the slowest Kafka broker, or the most loaded Railgun processor responsible for a larger data partition.

To make our experiments representative, in all three of them we used a real fraud dataset from one of our client. This dataset includes 103 fields, and with it, we aim to simulate real-world dictionary cardinalities for the aggregation states, and the expected load differences among the several Railgun processors.

In all of our experiments we have one or more injectors producing events to a single Kafka input topic (for computing metrics over a single *group by* over one stream) at a sustained throughput. Each computing node consumes events from the input topic, computes the several aggregations, and sends the results to the injector's dedicated reply topic. Latencies are measured by the injector based on the reply message time. I.e., we compute the *end-to-end* latency since the injector sends the message to Kafka, until the moment it consumes from Kafka the aggregations response. As such, this latency includes the network time, the communication overhead using Kafka, and the processing time of the Railgun (or Flink) computing node. These latencies are corrected to take into account the coordination omission problem [26].

The first two experiments (Section 5.1 and Section 5.2) aim to validate several of our design decisions, and have a simpler setup. For these, we use 3 *m5.2xlarge* AWS instances, with 8 vCPUs, 32GB of RAM, using only EBS storage. We use Kubernetes to deploy 1 Kafka *pod* (with Zookeeper), 1 injector, and 1 computing engine – either Railgun or Flink (v1.11.0) – with a JVM heap of 10GB all in separate VMs (by using Kubernetes anti-affinity rules). We use two Kafka topics – one to publish events with 10 partitions; and another to consume responses with 1 partition. Since we only use one Kafka node, replication is set to 1. In both these experiments the throughput is fixed at 500 ev/sec.

For the third experiment (Section 5.3), we simulate a more realistic scenario with multiple Railgun nodes and Kafka brokers. For this, we use multiple *m5.4xlarge* AWS instances, with 16 vCPUs, 64GB of RAM, with only EBS storage. Again, we use Kubernetes with anti-affinity rules to deploy multiple injector nodes, 30 Kafka brokers, and between 1 and 50 Railgun nodes - each with a JVM heap of 32GB - all in separate VMs. In all runs of this experiment, each Railgun node is configured with 8 processor units. Injectors publish events to a single topic, configured with a number of partitions that match the number of Railgun consumers of each run, viz., # processor units $\times$ # Railgun nodes. In this case, Kafka replication is set to 3, and the injectors are configured with *ack=all* to ensure delivery guarantees. We have one reply topic dedicated to each producer with 6 partitions. For aggregation replies, since they are usually discarded by the upstream systems if the reply arrives after our latency requirements, we set the topic replication to 1. In opposition to the previous two experiments, here we vary the injection throughput rate from a minimum of 25 thousand ev/sec to a maximum of 1 million ev/sec, according to the number of Railgun nodes in the cluster.

5.1 Comparing Flink with Railgun

On the first experiment we show the limitations stemming from using hopping windows, as described in Section 2.2, and how they compare with Railgun's real-time sliding window. For this, we chose Flink as it is one of the most performant [15, 27] and widely-used stream processing systems, and the closest to our functional needs.

As mentioned above, we use a single-node deployment for this experiment. The goal is to more fairly compare Railgun with Flink, using a simple setup, where, under a sustained throughput of 500 ev/s, we compute a single metric – the `sum(amount)` per card over a 60-min window. In Railgun we use a 60-min real-time sliding window, while for Flink we use hopping windows, and vary their hop size from 5 minutes to 1 second. Our goal is to show how Flink latency distributions behave when we attempt to approximate hopping windows to sliding windows. All experiment runs are of 35 minutes, where the first 5 minutes are for warmup, and ignored for latency purposes. To optimize Flink for latency rather than throughput, we set Flink's Kafka Client to use a batch timeout of 0.

The results are shown in Figure 8, where we include Railgun's latency for the same query using its real-time sliding window.

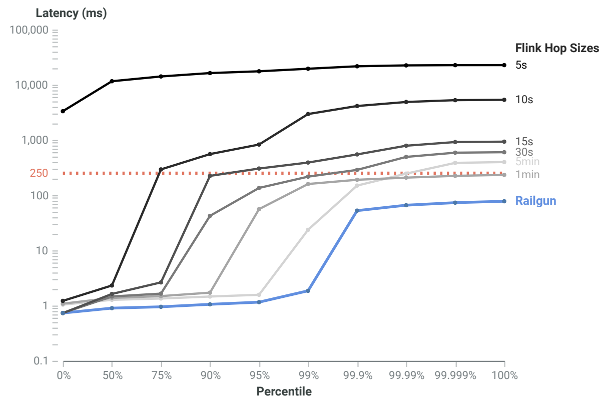


Figure 8: Distribution of Flink's latencies using hopping windows vs. Railgun's latencies using real-time sliding windows, at a fixed throughput of 500 ev/sec.

5.1.1 Discussion. Figure 8 shows how Flink latencies are affected when we increase the hop's granularity, i.e., when we increase the accuracy of the hopping window. Clearly, in this setup, with hops of 10s or less, Flink is unable to keep with a 500 ev/s throughput. Furthermore, recall that in most of our setups, we are required to score events in less than 250ms in the 99.9% percentile (cf. **M** requirement of Section 2.1). For those, we need hops of at least 1 minute, which would severely compromise accuracy, and violate rules for our clients (cf. **A** requirement of Section 2.1). We would expect that with larger hops, e.g., at 10-min, or 30-min hops, Flink would have lower latencies than Railgun, but those hops would produce even bigger aggregation inaccuracies. Railgun solves all these MAD requirements by using real-time sliding windows, accurate for every event, with lower latencies than Flink on all percentiles, on all windows using 1-min hops or less.

5.2 Scaling Railgun Windows

In our second set of experiments, we aim to demonstrate how Railgun performs when we scale the size of the window, and the number of windows, within a single machine. For this we designed two different experiments. For the first experiment **(a)**, we aim to show our claim that the window length is irrelevant for Railgun's performance. For that, we compute the same metric as in Section 5.1, but vary the window size from 5 minutes to 7 days. Since our experiment runs are of 35 minutes (with 5 minute of warmup), and our largest window has 7 days, we start these experiments after a data checkpoint load, to ensure that windows are always iterating events for both its head and tail iterator.

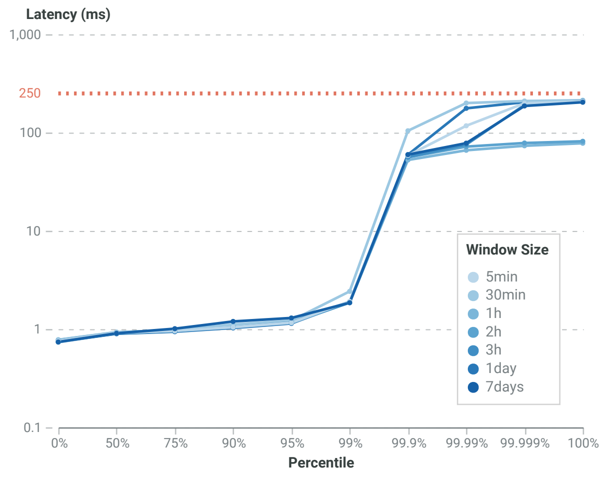
For the second experiment **(b)**, we compute three different metrics: `sum, average` and `count`, over the `amount` field grouped by `card`. Then, we vary the number of windows on which we compute these three metrics, to enforce a different number of reservoir iterators. Recall from Section 4.1.1 that a reservoir iterator is what hold event chunks in-memory for a given window. As depicted in Figure 5, normally, each iterator will hold two chunks in-memory. The window's head iterator, receiving incoming events, holds the open chunk on which new events are being appended, and might still hold one[4] closed chunk which is being written to disk asynchronously. Likewise, the window's tail iterator, iterating over the window's expiring events, necessarily holds the current chunk being iterated in-memory, and if possible, preemptively requests the following chunk to be loaded in the reservoir cache. Chunks might not be loaded if the reservoir cache is full. This might happen, if there are many iterators simultaneously reading reservoir chunks. Accessing a chunk not from cache can cause tail latency spikes. In the best scenario, these chunks are in the OS page cache and we only pay the cost for decompressing and deserializing. In the worst case, we also pay the full I/O seek cost.

The number of unique iterators depends on the number of windows configured in the system and how they are aligned. When two windows are aligned, either at the beginning or at the end, they share the same iterator. As such, for experiment (**b**), we force iterator misalignment by using windows with different window sizes and window delays. Namely, to vary from 20 to 240 iterators, in this experiment, we vary from 10 to 120 misaligned windows.

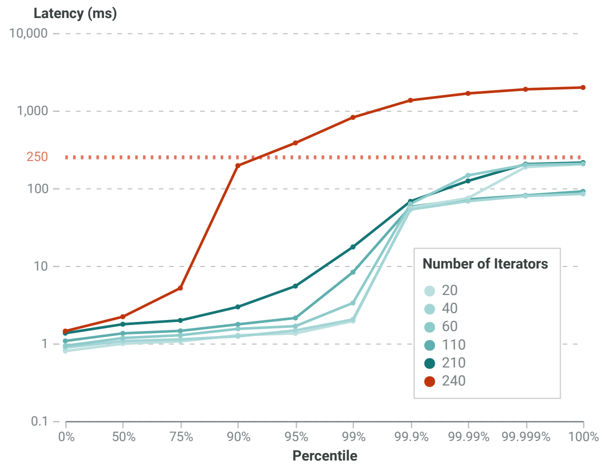
The results from both experiments can be seen in Figure 9.

5.2.1 Discussion. On experiment **(a)**, we clearly show that the window size is irrelevant to Railgun's latency performance. This is expected since for any window we have two iterators, independently of the window size. Additional benchmarks have shown us that variations in the higher percentiles (i.e., >99.9%), are due to Kafka communication, rather than Railgun (and something that

[4]more closed chunks might still be in-memory, if the reservoir is configured for extensive support of out-of-order events, or if there is contention on the disk I/O.



(a) Vary Window Size



(b) Vary Number of Iterators

Figure 9: Distribution of Railgun's latencies when scaling the window size and the number of windows, within a single machine, at a fixed throughput of 500 ev/s.

also affects Flink in Figure 8). Hence, in some runs we have 150ms in 99.99% percentile, while in others 75ms.

For experiment **(b)**, we show that as long as the iterators can retrieve the next chunk from cache, the impact on latencies is almost irrelevant. Each iterator requires a chunk in-memory, and, in this experiment, we used 220 chunk elements in Railgun's cache. This means that for most values used in this experiment (viz. between 20-210 iterators), whenever the iterator requests the next chunk, it is already available for iteration in the cache. Hence, we only start to see some latency degradation when we have almost the same elements in cache as the number of iterators, i.e., when we increase the probability of a cache-miss. On the run where we have 240 iterators, we also start to see Garbage Collection (GC) problems due to memory pressure, which then leads to higher latencies. This is as expected since the actual heap usage is very close to the maximum JVM heap (10GB).

5.3 Scaling Railgun Nodes

In the last set of experiments, we aim to demonstrate how Railgun scales to address higher throughputs in multi-node setup, while still respecting our target Msec latency requirement at high percentiles (<250ms @ 99.9%). Here we compute the same three metrics: `sum`, `average` and `count` of `amount` field grouped by `card` over a 5-min window, and configured Railgun node to process as much load as possible, in a sustained way, without breaching the **M** requirement. For these machines (AWS *m5.4xlarge* with 16 vCPUs and 64GB of RAM) we found the best performance using 8 Railgun processors per node and a JVM heap of 32GB (to take best advantage of compressed object pointers [40]), where we could comfortably handle loads of 25 thousand ev/sec. Afterwards, we set our target to 1 million ev/sec, and increased our Railgun nodes gradually to achieve this target. The results are shown in Figure 10.

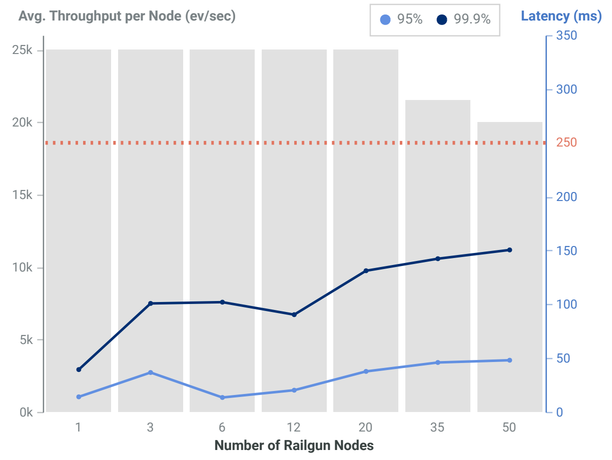


Figure 10: Evolution of the average throughput per node, when varying the throughput from 25 thousand ev/sec to 1 million ev/sec, and the number of nodes from 1 to 50.

5.3.1 Discussion. On this experiment, we show that Railgun scales almost linearly, where we only start to see some small degradation with 35 nodes at a combined throughput of 750 thousand of ev/sec. Our target load of 1 million ev/sec can be achieved using a Railgun cluster with 50 nodes, where each node is processing, on average, 20 thousand ev/sec.

When analyzing this experiment results for a single node, we understood that our main bottleneck when handling these high loads is memory pressure and GC performance. Namely, at 25 thousand ev/sec, we are creating objects at a rate of about 5GB/sec. Although our heap of live objects occupies less than 7GB, at this creation rate, the GC struggles to keep up. To achieve higher throughputs per node we need to change our system to use off-heap memory management optimizations, as frequently done in other streaming engines [42, 51, 52]. Moreover, when scaling the cluster with multiple nodes, we start to see a bottleneck in Kafka, probably caused by the increased number of partitions needed to support the concurrent consumption of messages from the multiple nodes' processors. This is something to be improved in the future with a more careful tuning of Kafka configurations, and broker setup.

6 CONCLUSIONS

In this paper we propose Railgun, a novel distributed streaming engine that supports aggregations over real-time sliding windows, while providing crucial non-functional requirements: high throughput, tail low latency, horizontal scalability and fault tolerance.

One of Railgun's most important enablers is the event reservoir. Since accurate metrics require considering all events, the reservoir efficiently persists them to disk, while fetching chunks of events ahead of time as they are needed by windows. This allows Railgun to support time-windows spanning years with the same memory usage as windows of seconds. To reduce storage costs, the reservoir uses cheap local HDDs or network-attached disks, and exploits the events' immutability to aggressively compress and serialize them.

Another central piece of Railgun's performance is how it takes advantage of Kafka to achieve a distributed, scalable and fault-tolerant system. In particular, we use the concept of *(topic, partition)* to delegate tasks among the several Railgun processors and exploit Kafka consumer's group guarantees to safeguard tasks assignment to a Railgun processors. Finally, to optimize recovery, we also provide a custom rebalance assignment strategy that minimizes data shuffle and maximizes load distribution across the cluster.

Railgun is still under development, and some work lies ahead to validate some components of our design. Namely, we need to certify that: 1) rebalance and recovery can be done respecting our latency requirements over the 99.9 percentile, especially when a task is assigned to a processor that has no previous data for that task; 2) we can efficiently support metrics backfill, i.e., the ability to add a new metric and fill it from old event data.

Although still a prototype, our experiments provide sound promises for Railgun. Particularly, we show that Railgun has lower latencies per event than Flink, even when Flink is configured for a low metric accuracy (e.g., a 5-min hop size for a 60-min window). In addition, we show that our performance is unaffected by the window size, and that Railgun scales reasonably well with the number of windows and metrics, as we are able to prevent I/O calls on the critical path of event processing, by pro-actively loading reservoir chunks into memory, ahead of time.

Lastly, we demonstrate that Railgun can scale nearly linearly and process throughputs of million of events per second. Surely, there are many streaming engine systems achieving higher throughputs per node than Railgun, including Flink [11], Spade [23], Kafka Streaming [7] or Spark Streaming [7]. However, to achieve it, these systems have to degrade latency at high percentiles, or make significant compromises on sliding window aggregation precision. To the best of our knowledge, Railgun is the first distributed streaming engine able to deliver accurate real-time sliding window aggregations, with millisecond-level latencies at high percentiles, thereby making it the first of MAD systems.

REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Ç., M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. 2005. The

Design of the Borealis Stream Processing Engine. In *CIDR 2005*. 277–289.

[2] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 277–289. http://cidrdb.org/cidr2005/papers/P23.pdf

[3] D. J. Abadi, D. Carney, U. Ç., M. Cherniack, C. Convey, C. Erwin, E. F. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. B. Zdonik. 2003. Aurora: A Data Stream Management System. In *ACM SIGMOD 2003*. ACM, 666. https://doi.org/10.1145/872757.872855

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael 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. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[5] Alibaba. 2020. Alibaba Cloud Supported 583,000 Orders/Second for 2020 Double 11. https://www.alibabacloud.com/blog/alibaba-cloud-supported-583000-orderssecond-for-2020-double-11---the-highest-traffic-peak-in-the-world_596884.

[6] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. 2003. STREAM: The Stanford Stream Data Manager. In *SIGMOD 2003*.

[7] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 601–613. https://doi.org/10.1145/3183713.3190664

[8] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis (Eds.). ACM, 1–16. https://doi.org/10.1145/543613.543615

[9] L. Benson, P. M. Grulich, S. Zeuch, V. Markl, and T. Rabl. 2020. Disco: Efficient Distributed Window Aggregation. In *EDBT'20*. OpenProceedings.org, 423–426.

[10] Bernardo Branco, Pedro Abreu, Ana Sofia Gomes, Mariana S. C. Almeida, João Tiago Ascensão, and Pedro Bizarro. 2020. Interleaved Sequence RNNs for Fraud Detection. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash (Eds.). ACM, 3101–3109. https://doi.org/10.1145/3394486

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[12] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *CIKM'16*. ACM.

[13] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, and James F. Terwilliger. 2015. Trill: Engineering a Library for Diverse Analytics. *IEEE Data Eng. Bull.* 38, 4 (2015), 51–60. http://sites.computer.org/debull/A15dec/p51.pdf

[14] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing for Uncertain World. In *CIDR'03*.

[15] Subarna Chatterjee and Christine Morin. 2018. Experimental Study on the Performance and Resource Utilization of Data Streaming Frameworks. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*, Esam El-Araby, Dhabaleswar K. Panda, Sandra Gesing, Amy W. Apon, Volodymyr V. Kindratenko, Massimo Cafaro, and Alfredo Cuzzocrea (Eds.). IEEE Computer Society, 143–152. https://doi.org/10.1109/CCGRID.2018.00029

[16] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (Eds.). ACM, 379–390. https://doi.org/10.1145/342009.335432

[17] Apache Commons. 2011. JEXEL Expressions. https://commons.apache.org/proper/commons-jexl/.

[18] Confluent. 2016. Kafka Streams. https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/.

[19] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80. https://doi.org/10.1145/2408776.2408794

[20] Facebook. 2012. RocksDB. https://rocksdb.org/.

[21] A. Fedulov. 2020. Advanced Flink Appl. Patterns Vol.3: Custom Window Processing. https://flink.apache.org/news/2020/07/30/demo-fraud-detection-3.html.

[22] Michael J. Franklin, Sailesh Krishnamurthy, Neil Conway, Alan Li, Alex Russakovsky, and Neil Thombre. 2009. Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In *Fourth Biennial Conference on Innovative Data Systems Research, CIDR 2009, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*. www.cidrdb.org. http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_122.pdf

[23] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. 2008. SPADE: the system s declarative stream processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 1123–1134. https://doi.org/10.1145/1376616.1376729

[24] Lukasz Golab and M. Tamer Özsu. 2003. Issues in data stream management. *SIGMOD Rec.* 32, 2 (2003), 5–14. https://doi.org/10.1145/776985.776986

[25] Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. 2013. A catalog of stream processing optimizations. *ACM Comput. Surv.* 46, 4 (2013), 46:1–46:34. https://doi.org/10.1145/2528412

[26] T. Hoff. 2015. Your Load Generator Is Probably Lying To You - Take The Red Pill And Find Out Why. http://highscalability.com/blog/2015/10/5/your-load-generator-is-probably-lying-to-you-take-the-red-pi.html.

[27] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[28] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA (The IBM Research Symposia Series)*, Raymond E. Miller and James W. Thatcher (Eds.). Plenum Press, New York, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

[29] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. 2015. Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*. IEEE Computer Society, 2785–2792. https://doi.org/10.1109/BigData.2015.7364082

[30] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA.

[31] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. 1–7.

[32] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 239–250. https://doi.org/10.1145/2723372.2742788

[33] Lightbend. 2011. Akka Streams. https://doc.akka.io/docs/akka/current/stream/index.html.

[34] Marcelo R. N. Mendes, Pedro Bizarro, and Paulo Marques. 2013. Overcoming memory limitations in high-throughput event-based applications. In *ACM/SPEC International Conference on Performance Engineering, ICPE'13, Prague, Czech Republic - April 21 - 24, 2013*, Seetharami Seelam, Petr Tuma, Giuliano Casale, Tony Field, and José Nelson Amaral (Eds.). ACM, 399–410. https://doi.org/10.1145/2479871.2479933

[35] Julio J. Navas. 2010. Insight into Events: Event and Data Management for the Extended Enterprise. In *Enabling Real-Time Business Intelligence - 4th International Workshop, BIRTE 2010, Held at the 36th International Conference on Very Large Databases, VLDB 2010, Singapore, September 13, 2010, Revised Selected Papers (Lecture Notes in Business Information Processing)*, Malú Castellanos, Umeshwar Dayal, and Volker Markl (Eds.), Vol. 84. Springer, 24–35. https://doi.org/10.1007/978-3-642-22970-1_3

[36] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Stateful Scalable Stream Processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[37] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Stateful Scalable Stream Processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[38] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[39] Karthik Ramasamy. 2019. Unifying Messaging, Queuing, Streaming and Light Weight Compute for Online Event Processing. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, DEBS 2019, Darmstadt, Germany, June 24-28, 2019*. ACM, 5. https://doi.org/10.1145/3328905.3338224

[40] John Rose. 2012. CompressedOops. https://wiki.openjdk.java.net/display/HotSpot/CompressedOops.

[41] B. V: Roy. 2007. A short proof of optimality for the MIN cache replacement algorithm. *Inf. Process. Lett.* 102, 2-3 (2007), 72–73.

[42] Xuanhua Shi, Zhixiang Ke, Yongluan Zhou, Hai Jin, Lu Lu, Xiong Zhang, Ligang He, Zhenyu Hu, and Fei Wang. 2019. Deca: A Garbage Collection Optimizer for In-Memory Data Processing. *ACM Trans. Comput. Syst.* 36, 1 (2019), 3:1–3:47. https://doi.org/10.1145/3310361

[43] Pivotal Software. 2007. RabbitMQ. https://www.rabbitmq.com/.

[44] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. 2011. Siddhi: a second look at complex event processing architectures. In *Proceedings of the 2011 ACM SC Workshop on Gateway Computing Environments, GCE 2011, Seattle, WA, USA, November 18, 2011*, Rion Dooley, Sandro Fiore, Mark L. Green, Cameron Kiddle, Suresh Marru, Marlon E. Pierce, Mary Thomas, and Nancy Wilkins-Diehr (Eds.). ACM, 43–50. https://doi.org/10.1145/2110486.2110493

[45] Sybase. 2009. Coral8 Integration Guide. http://infocenter-archive.sybase.com/help/topic/com.sybase.infocenter.dc01030.0200/pdf/cep-IntegrationGuide.pdf.

[46] Georgios Theodorakis, Alexandros Koliousis, Peter R. Pietzuch, and Holger Pirk. 2020. LightSaber: Efficient Window Aggregation on Multi-core Processors. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2505–2521. https://doi.org/10.1145/3318464.3389753

[47] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. 2014. Storm@twitter. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 147–156. https://doi.org/10.1145/2588555.2595641

[48] Jonas Traub, Philipp Marian Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2018. Scotty: Efficient Window Aggregation for Out-of-Order Stream Processing. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1300–1303. https://doi.org/10.1109/ICDE.2018.00135

[49] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. Knowl. Data Eng.* 15, 3 (2003), 555–568. https://doi.org/10.1109/TKDE.2003.1198390

[50] Author(s) B. P. Welford and B. P. Welford. 1962. Note on a method for calculating corrected sums of squares and products. *Technometrics* (1962), 419–420.

[51] Reynold Xin and Josh Rosen. 28 April 2015. Project Tungsten: Bringing Apache Spark Closer to Bare Metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[52] Andrey Zagrebin. 21 April 2020. Memory Management Improvements with Apache Flink 1.10. https://flink.apache.org/news/2020/04/21/memory-management-improvements-flink-1.10.html.

[53] Shuhao Zhang, Hoang Tam Vo, Daniel Dahlmeier, and Bingsheng He. 2017. Multi-Query Optimization for Complex Event Processing in SAP ESP. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 1213–1224. https://doi.org/10.1109/ICDE.2017.166