Leveraging Watermarks to Improve Performance of Streaming Systems

by

Omar Farhat

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I would like to acknowledge the names of my co-authors who contributed to the research described in this thesis, these include:

- Dr. Khuzaima Daudjee
- Dr. Leonardo Querzoni
- Harsh Bindra

Abstract

Modern stream processing engines (SPEs) process large volumes of events propagated at high velocity through multiple queries. By continuously receiving watermarks, which are marker events injected into the stream to signify that no further events are expected beyond a designated timestamp, SPEs can infer stream progress to correctly process window operators. While stream progress is useful information for query execution, it is only utilized to ensure input completion. We argue that to improve performance, stream progress should be leveraged in the design of SPE subsystems. In this thesis, we demonstrate the significant advantages of leveraging stream progress to solve two important SPE problems: query scheduling, and query sample processing.

First, existing SPE schedulers generally aim to minimize query output latency by minimizing, in turn, the mean propagation delay of events in query pipelines. However, for queries containing commonly used blocking operators such as windows, we show that a superior approach would be to prioritize the queries based on stream progress. Through the design and development of Klink, we leverage stream progress to unblock window operators and to rapidly propagate the events to output operators. Secondly, sample query processing limits input to only a subset of events such that the sample is statistically representative of the input while ensuring output accuracy guarantees. However, output latency can be significantly increased because relevant watermarks can suffer from large ingestion delay due to long or bursty network latencies. Window computations that account for stragglers can add significant latency while providing inconsequential accuracy improvement. We propose Aion, an algorithm that utilizes sampling to provide approximate answers with low latency by minimizing the effect of stragglers through leveraging control over stream progress.

We integrate Klink and Aion into the popular open-source SPE Apache Flink. We demonstrate that Klink delivers hefty performance gains on benchmark workloads, reducing mean and tail query latencies by up to 60% over existing scheduling policies. Similarly, using different benchmark workloads, we demonstrate that Aion reduces stream output latency by up to 85% while providing 95% accuracy guarantees.

Acknowledgements

I would like to thank my supervisor Dr. Khuzaima for his direct guidance in both an academic and professional capacity. His distinct diligence, rigorous commitment, and extensive experience have improved my research, writing, and analytical skills. Thank you to Dr. Tamer Özsu and Dr. Samer Al-Kiswany for reading my thesis and providing helpful feedback.

I wish to express my gratitude and appreciation to my friends who supported me through my capricious experiences. I owe many thanks to my brother, my mother, and my father for their unconditional love and support. Finally, for those who transcended fear, defied boundaries, and galvanized change, your resilience has echoed a Thawra miles and hearts apart.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Streaming systems are a popular choice for use by applications driven by the need to process large volumes of data at high velocity [20]. Modern applications increasingly rely on instantaneous responses to speed up decisions and actions [99]. Examples of such application domains include real-time analytics, anomaly detection, and real-time object recognition [104] that leverage streaming systems to deliver on their data processing requirements [99, 78, 22, 58]. For instance, to significantly improve on driver safety, Glimpse [28] needs to process road signs and traffic lights within very low latencies not exceeding 33ms. Google's MillWheel streaming system supports ads customers who require low latency updates to customer-visible dashboards [7]. Other applications that rely on low-latency streaming services include news reporting from large volumes of Twitter feeds to be analyzed within seconds to detect latest news topics and events [60, 61]. The Google Trends service (formerly Zeitgeist) pipeline ingests a continuous input of search queries and detects anomalous queries within seconds [7]. Existing stream processing engines (SPEs) such as Apache's Flink [18] and Spark [102] are designed to provide sub-second processing latencies to respond instantaneously to demands of such applications [97, 99].

SPEs provide event processing semantics where streaming *queries* are defined by multiple processing units called *operators* deployed over one or multiple nodes [44, 92]. Applications interested in monitoring events such as in anomaly detection typically use *window* operators based on the event's timestamp generated at the source, called *event-time*. However, the order of events received by event-time windows does not necessarily mirror the order of events generated at the source, as these may arrive at the window out-of-order due to network latencies or parallel execution of prior operators. Events disorder is fundamentally challenging for SPEs because they make window input completeness hard to guarantee. Window operators cannot ensure that all the relevant events were collected as

events can be arbitrarily delayed up to an unspecified amount of time, while not accounting for these events can lead to incorrect output. To resolve this problem, SPEs typically support out-of-order processing (OOP) architecture through leveraging *watermarks*.

Watermarks are timestamped events denoting that no events preceding their timestamp should be further expected. They essentially represent a contract between the user and the SPE to ensure input completeness and output correctness. Watermarks also hold another significance: the progress of the stream in event-time can be interpreted from their frequency of propagation. Specifically, by continuously receiving watermarks, window operators can reason about their progress in terms of input completion. The stream progress property provides important insights into the current state of execution that can be used to improve the general design and performance of SPEs. In this thesis, we demonstrate the significant advantages of leveraging stream progress to solve two important SPE problems: query scheduling, and query sample processing.

1.1 Runtime Query Scheduling

In SPEs, the problem of stream query scheduling takes on significant importance as the number of active operators far outstrips the number of CPU threads that are available to run. Since SPEs typically rely on the operating system (OS) on which they run to schedule applications' queries, the OS scheduler is agnostic of the characteristics of these queries and their operator pipelines over which the streaming data is processed. That is, the OS does not attempt to leverage query or operator characteristics to optimize query execution so that *query latency* is minimized. For example, OS schedulers such as the Linux scheduler implement standard policies like "fair scheduling" that do not consider how to take advantage of a window operator's semantics to minimize query latency.

Operator scheduling policies such as Highest Rate [88] aim to reduce query latency by minimizing the mean propagation delay of events in the pipeline. That is, the idle time spent by each event in the input queues of each operator is minimized. The main intuition behind these policies is that minimizing the propagation delay of events will yield better performance. However, for applications demanding low processing latencies [99], these strategies do not perform well if queries contain complex operators such as window and *join* [49]. Specifically, window operators block the stream from flowing until their input is complete. Thus, minimizing the mean propagation delay of each event in the pipeline does not necessarily translate to minimized output latency if window operators are not expected to complete their input in the near interim.

Efficient operator scheduling for queries containing window operators, which are commonly and routinely used in streaming query pipelines, is a challenging problem to solve. This is because:

(i) Identifying events relevant to window operators is difficult. Events can be arbitrarily delayed for ingestion due to, e.g., network delays or parallel and distributed executions, while windows claim events based on the generation order of events at the source. Consequently, the performance of the SPE is prone to suffer greatly.

(ii) SPEs are unaware of the time at which windows are due to be processed. For example, a time-window of five seconds may need to wait for additional time stretching to as far as thirty seconds to account for any arbitrary delays (e.g. network delays, event processing delays). Hence, it is challenging to correctly identify and prioritize queries that are due to be processed.

Through leveraging stream progress, we show that the best strategy to minimize output latency is to prioritize the propagation of relevant events to window operators that are due to be processed first and then propagate their output downstream. This minimizes the blocking of window operators, which in turn minimizes the output latency of the stream, resulting in faster stream progress.

To illustrate using an example with multiple query operators, consider Fig. 1.1 showing two window operators *W1* and *W2* each with their queued events. *W1* and *W2* aggregate all events generated within a time range (window) of seven seconds and six seconds, respectively. The queue size for *W1* is 5 events and 4 events for *W2*. Consider two scheduling algorithms deployed on a standard[a] processor core: First-Come-First-Serve (FCFS) [88, 80], and Klink, the algorithm that this thesis presents, that implements the aforementioned rationale. FCFS, the latency optimized policy that is shown first, is used to process ingested events across *W1* and *W2*. FCFS initially consumes event 1 of *W1* and then event 2 of *W2* since they are the two events at the head of the queue. Then, FCFS alternates between the two windows processing events 2 of *W2*, 3 of *W1*, and 3 of *W2*. The subsequent 2 events of *W1* are processed next. After processing *W2*'s event 6 that represents the final event in the window, output is emitted at time 8 seconds, incurring a two second window latency as *W1*'s output was emitted two seconds after the window's deadline. FCFS incurs the same window output latency for *W1* as for *W2*. Thus, FCFS achieves an average output latency of two seconds. Klink, the algorithm proposed in this thesis, is primarily derived from the observation that prioritizing the completion of the window input significantly reduces

output latency. Specifically, Klink prioritizes events while taking into consideration window deadlines so that the output latency is minimized. Initially, Klink processes the first two events 1 and 2 of *W2*. Next, despite *W2*'s deadline being more imminent than *W1*, the small difference in their deadlines combined with the longer queue of *W1* leads Klink to schedule events 2 and 3 of *W1*. Then, Klink processes all of *W2*'s events within the window's deadline achieving zero window output latency. Finally, Klink processes all remaining events of *W1*, achieving a latency of 2 seconds. Therefore, Klink achieves an average window latency reduction of 50% over FCFS.

[a] A commodity, off-the-shelf, processor that de facto uses time slicing for multithreading [48].

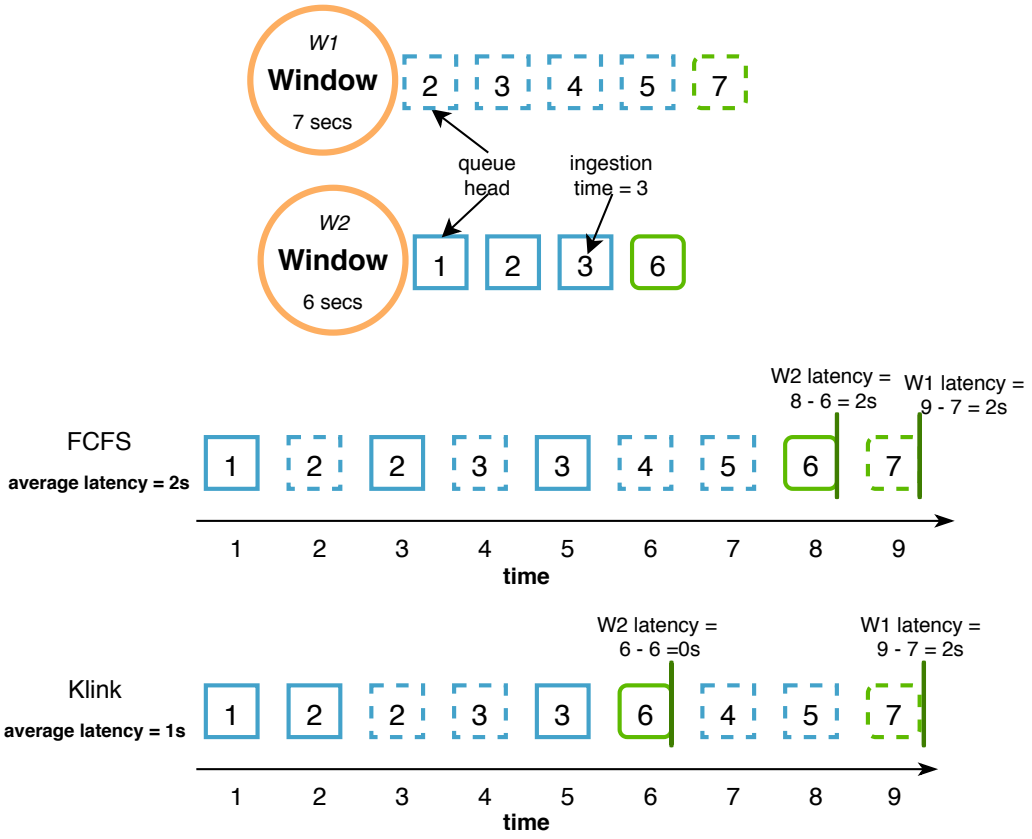


Figure 1.1: Example demonstrating Klink's superior scheduling policy over the sub-optimal First-Come-First-Serve (FCFS) policy that does not prioritize window deadlines.

This simple example shows that to be efficient, scheduling policies aimed at reducing output latency must account for input completion progress of the window operators.

Furthermore, for the many types of applications that are latency-conscious or require low-latency performance (as described earlier in this chapter), latency reduction is an important performance goal even when SPE throughput is not a consideration. To further demonstrate the performance impact of this scheduling problem, we measured the output latency while varying the number of input events per second for two different query workloads and complexities, namely the Yahoo! Streaming Benchmark (YSB) [29] and the Linear Road Benchmark (LRB) [10].

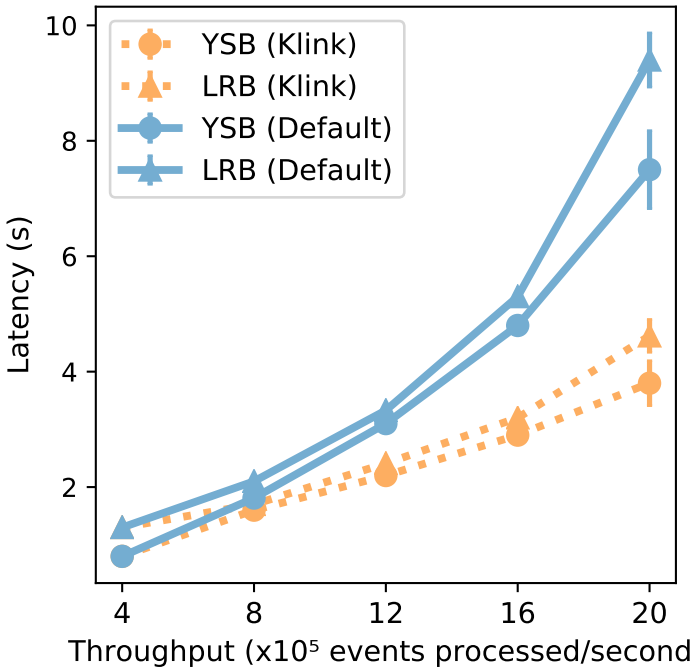


Figure 1.2: Average output latency vs. SPE throughput (number of events processed per second)

We deployed Flink on a machine to process these two workloads, using two different schedulers: Flink's *Default* (the OS scheduler) as well as our proposed scheduler that we call **Klink**. Fig. 1.2 shows that, considering a desired throughput level (on the x-axis), significant extra output latency of 50% for both YSB and LRB is incurred by Default over Klink. This overhead is further exacerbated for SPEs processing higher number of events as is typical in real deployments where network delay is also variable, imposing the aforementioned challenges (i) and (ii). These significant performance gains achieved by Klink demonstrate that schedulers can be designed to deliver significant reductions in

output latencies even at comparable throughput performance.

In this thesis, we present the design and implementation of our scheduler, called Klink, that optimizes for stream progress to reduce the output latency for queries running window operators. Klink leverage stream progress information through utilizing watermarks thereby solving challenges (i) and (ii). We utilize watermarks in our solution to develop a high performance scheduler aware of the progress of window operators.

As introduced earlier, we also explore leveraging the stream progress property in query sample processing.

1.2 Query Sample Processing

The problem of query processing is of significant importance as the substantial cost of processing high volume of events violates the real-time requirement [92]. *Sample processing* is a computing paradigm proposed to enforce this requirement by efficiently processing queries via limiting the input size to a subset of events [82, 57]. Fundamentally, it achieves efficiency by trading-off output accuracy for lower latency. This trade-off is viable for many streaming applications as timely generated output with accuracy guarantees is often much more useful than latent or delayed output with exact accuracy [53, 52, 44, 6, 13]. In this context, bounding output accuracy is an essential factor that cannot be overstated. Thus, sample processing strives to achieve minimal latency within output accuracy requirements.

Streaming queries popularly utilize sample processing to reduce the processing cost of events. For queries exhibiting window operators, sample processing initially selects a subset of incoming events such that processing them would satisfy the output accuracy requirements. At the time of window completion, that is, after all the events relevant to the window operator have been observed by the SPE, the sample is then processed downstream to the output operator.

In query sample processing, the output is propagated downstream only *after* input completion. SPEs can experience long wait times for input completion for a window due to network delays [76, 84]. For instance, in the example illustrated in Fig.1.3, the SPE waits for the arrival of events e_3 and e_5 even after the window's deadline. Consequently, *stragglers* delay input completion thereby increasing output latency. Consider Fig. 1.4 that illustrates the impact of stragglers on output latency for a time-window of size 1.5s where the network delay of events varies based on distributions that are modeled from real-world traces[1] [16, 84]. For a network delay that is Gamma distributed (light-tail), stragglers

[1]Network delay behavior can be modeled by the Exponential and the Gamma distributions.

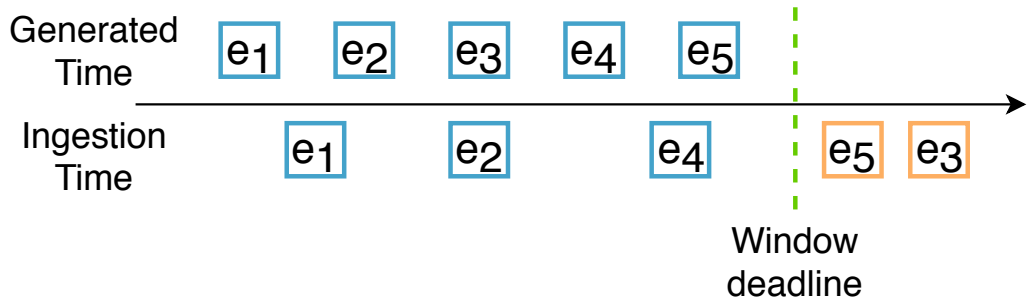


Figure 1.3: Example illustrating the difference between generation and ingestion time for each event. On-time events are highlighted in blue, while stragglers are highlighted in orange.

impose a significant 25% additional wait-time to guarantee input completion. As for Exponentially distributed delays (heavy tail), the imposed latency is exacerbated by more than 99%, effectively delaying the input completion way beyond the window's deadline. The impact of stragglers on output latency is significantly high so as to overshadow the acquired benefits from sample processing. To the best of our knowledge, none of the existing sample processing techniques mitigate the impact of stragglers on the output latency [36, 41, 53].

Mitigating the problem of stragglers on sample processing while striking a balance between accuracy and latency is a challenging problem to solve. This is because:

(i) Straggler count and delay patterns vary based on an application's environment. To choose a sample that satisfies the specified accuracy guarantees requires constructing reliable estimations of the straggler events. However, network delays, and in particular the case of exponential delay distributions, adds high variability to any estimation technique.

(ii) Choosing a sample that satisfies the accuracy guarantees is not a trivial task. The sampled events need to be statistically representative of the original input that includes the stragglers. Furthermore, the size of the sample needs to be intelligently chosen based on the functionality of the window operator.

(iii) Determining the minimum number of stragglers to include in the sample can have a large impact on the output latency. As illustrated in Fig. 1.4, stragglers impose a significant delay penalty on the output latency. Hence, the number of included stragglers needs to be minimized.

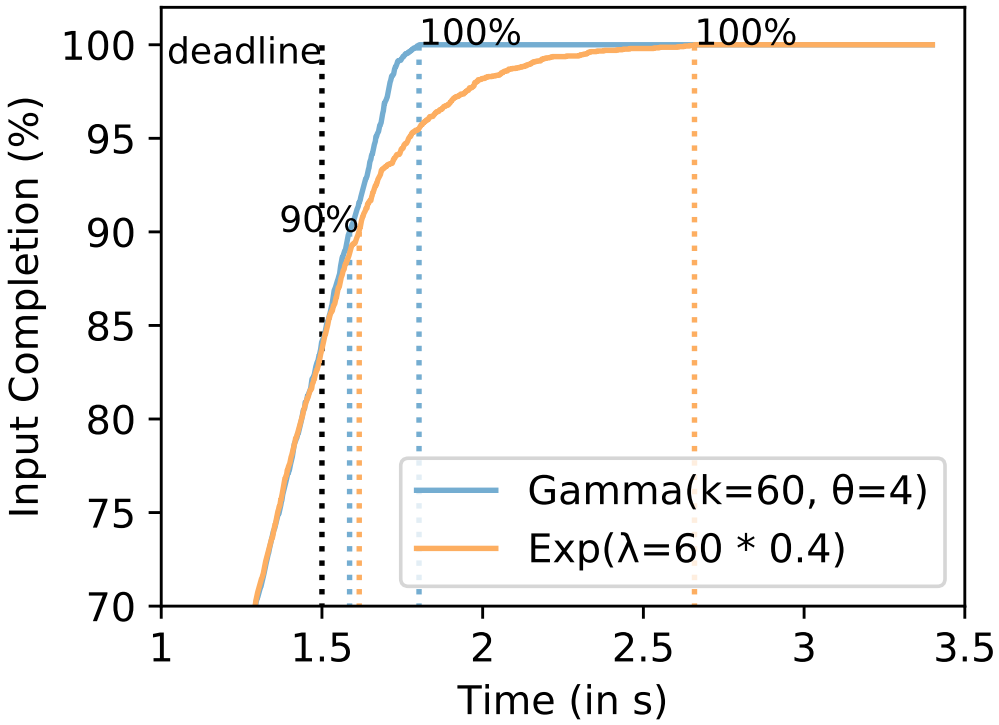


Figure 1.4: Input completion rate for window operators whereby network delay for events is modeled by different empirically verified distributions.

In this thesis, we demonstrate that sample processing does not need to wait for input completion to process its sampled input. Specifically, existing sample processing approaches do not need to add a *slack* delay to account for stragglers [52]. Instead, windows can propagate their output downstream as soon as output accuracy requirements are satisfied thereby circumventing the costly slack delay, and, consequently, reducing the output latency significantly.

To illustrate the impact of stragglers on output results, consider Fig. 1.5 that shows output accuracy results obtained running a sample processing algorithm in two settings: with and without stragglers. The first setting takes stragglers into account thereby waiting for input completion while the second circumvents stragglers by processing the window's input as soon the deadline is due and the accuracy requirements are satisfied. In both settings, the algorithm ran with a target output accuracy of 95%

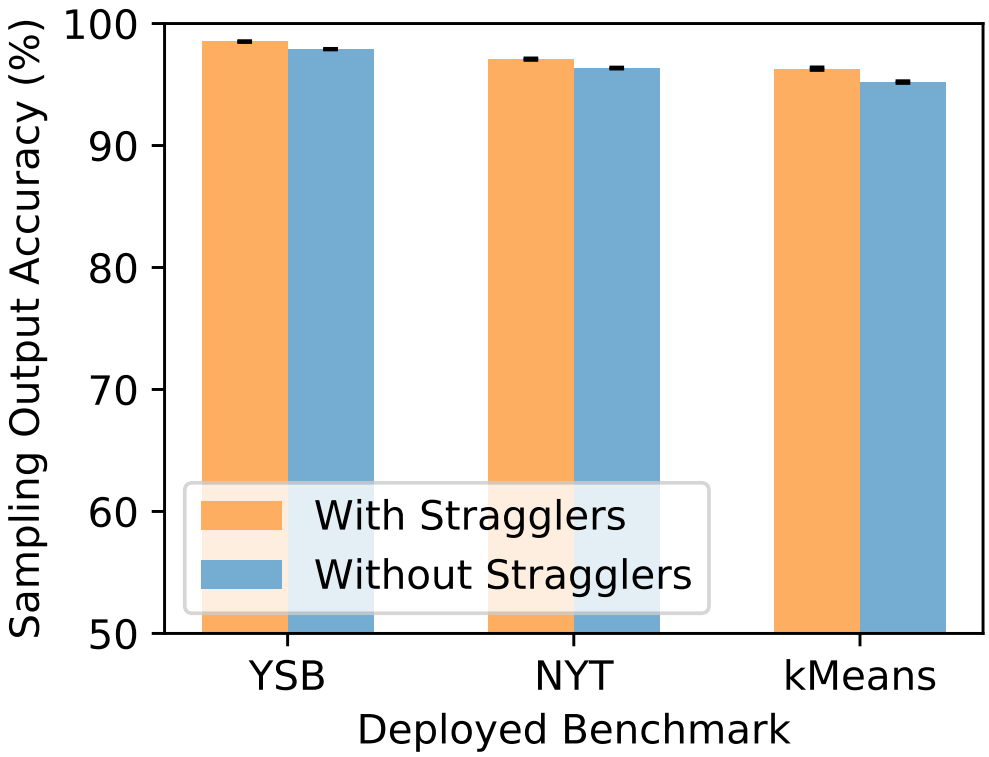


Figure 1.5: Relative accuracy obtained of sample processing window operators with and without stragglers.

over two popular streaming benchmarks that include windowed operators of different functionalities. The first popular benchmark is the Yahoo! Streaming Benchmark [29] (YSB) and the second is the New York Taxi (NYT) benchmark [77]. The figure shows that sampling with stragglers provides an insignificant improvement of *less than* 1% compared to sampling without stragglers for both YSB and NYT benchmarks. We also ran the kMeans benchmark as an example of a query with a windowed operator of higher complexity. The accuracy difference between the two samples was only about 1% indicating that the two samples shared identical statistical significance. These results demonstrate that not only do stragglers impose *large* output latency on input completion, they contribute *insignificantly* towards achieving higher output accuracy.

This observation motivates the work in which we present the design and implemen-

tation of our sample processing algorithm called Aion[2]. Aion continuously monitors and samples important patterns in the workload such as network and inter-event generation delays to estimate the pattern of events and stragglers. Aion then utilizes these estimations, in addition to the type of the window operator, to compute the minimum sample size that achieves the accuracy guarantees. Aion also exploits straggler patterns by *intelligently* processing the sample before input completion such that the impact of stragglers is mitigated. As we demonstrate in our experiments, Aion delivers significant performance gains to reduce latency by as much as 80%.

1.3 Thesis Contributions & Outline

To summarize, the contribution of this thesis is two-folds:

- We present Klink, a scheduler optimized for running multiple queries delivering up to 60% *mean* and *tail* output latency reduction over state-of-the-art techniques. Klink represents a breakthrough over existing scheduling policies in that it *intelligently* optimizes for minimizing the delay in generating output by processing the necessary events to progress the stream into materializing its results. Klink's general design is leveraged through integrating it into Apache Flink [18], a popular state-of-the-art SPE.
- We propose Aion, an algorithm that utilizes sampling to provide approximate answers with low latency by controlling stream progress and minimizing the impact of stragglers. Aion quickly processes the window to minimize output latency while still achieving high accuracy guarantees. Similarly, we implement Aion in Apache Flink and show using benchmark workloads that Aion reduces stream output latency by up to 85% while providing 95% accuracy guarantees.

The rest of this thesis is organized as follows. Chapter 2 surveys related work and provides background on scheduling and sample processing. Chapter 3 presents Klink scheduling policy and provides experimental results to verify its efficiency. Chapter 4 introduces Aion design and its algorithmic details. In addition, Chapter 4 also presents experimental results for Aion. Finally, Chapter 5 concludes the thesis and outlines future directions for this work.

[2]Aion is the ancient Greek god of ages representing unbounded time.

Chapter 2

Background & Related Work

In this chapter, we describe window processing semantics and watermarks. We also discuss past work on query scheduling algorithms and sample query processing techniques.

2.1 Window Processing Semantics

To process computation on a stream of data, SPEs provide semantics for grouping events that exhibit common properties into structures called windows. Windows provide flexibility and can host complex grouping selections [46]. Each window is characterized by a *deadline* that identifies when the windowed operation is expected to output a new result based on the current group of events collected in the window. As an example, the deadline of a five-second event-time tumbling window starting from timestamp zero is the sequence of multiples of five seconds, while a deadline for a five-event count-window elapses every 5th event collected by the window. After production of each new output, the window is emptied to receive new events until the next deadline. Windows are important as query operators such as join, aggregation, and selection are executed on a per window basis [56].

Applications interested in monitoring events such as in anomaly detection typically use windows based on the event's timestamp generated at the source, called *event-time*. However, the order of events received by event-time windows does not necessarily mirror the order of events generated at the source, as these may arrive at the window out-of-order due to network latencies or parallel execution of prior operators. Out of order events occur in SPEs for multiple reasons such as:

- Packets holding events generated by multiple transmissions from a remote origin can propagate through different paths incurring different delays. Consequently, events can be frequently ingested out-of-order relative to their generation order at the source.
- Windows that group or join several streams running in parallel or distributed executions on multiple machines can incur varying latencies associated with communication and coordination within, and across, machines. This would give rise to out-of-order ingestion of events.

Stragglers are challenging for SPEs because they make window input completeness hard to guarantee. Window operators cannot ensure that all the relevant events were collected as events can be arbitrarily delayed up to an unspecified amount of time, while not accounting for these events can lead to incorrect output. For example, consider a count-window that processes groups of five events at a time. Ideally, the window should process every five events in the same order as they were generated at the source. However, because of stragglers, count-windows need to wait and collect every five events with the lowest event-time and then process them to guarantee equivalent ordering. Otherwise, the windowed operator will produce incorrect output. Furthermore, in the example of Fig. 2.1, the first watermark (3) acts as a progress indicator, hinting that about half of the input for the window has been seen.

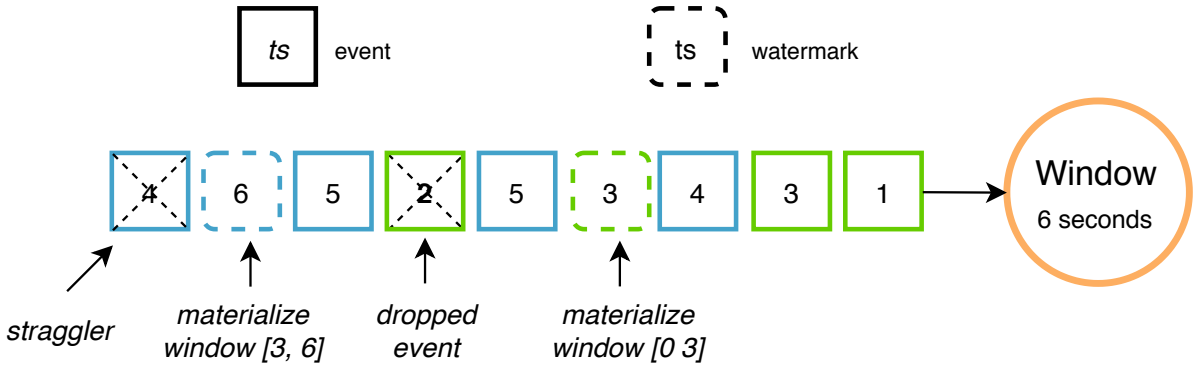


Figure 2.1: Example illustrating the concept of watermarks in SPEs. Events are consumed in order starting from right to left and each event holds its generation timestamp at the source. Events that are of the same colour as a watermark are processed with the ingestion of that watermark.

These challenges pushed SPEs designers to adopt techniques to enforce *in-order* processing semantics, which impose large performance overheads [62] as in-order processing

perilously delays the execution of events until they are adequately re-ordered to guarantee correctness.

Conversely, *out-of-order* processing (OOP) semantics allows window operators to be unblocked without imposing any ordering constraints on the stream [62, 52]. The OOP semantics can be implemented in multiple ways:

- Stragglers can be dropped after a given time threshold. Once the threshold is exceeded to guarantee input completeness, windows can be safely processed. However, this approach burdens the user with the difficult task of requiring conservative estimates about anticipated lateness, as significant extra output latency will be incurred otherwise.
- Periodically generating *watermarks* [62] that are special events injected in the stream to signify that no events earlier than their timestamp are expected anymore. Therefore, windows can be safely processed once the SPE ingests a watermark timestamp elapsing the window's deadline. Watermarks are commonly generated by the application, alleviating the uncertainty concern of receiving late events from the engine.

Most modern SPEs such as Flink, Spark and Storm support out-of-order processing by using watermarks [7, 18, 8].

2.2 Watermarks

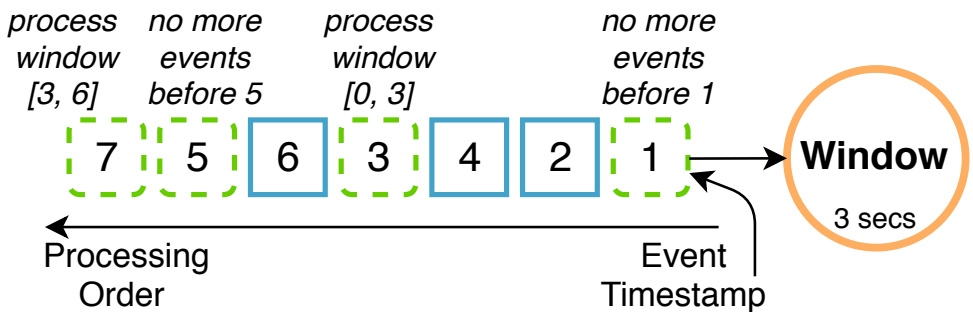


Figure 2.2: Example reflecting the progress property of watermarks.

Watermarks are timestamped events denoting that no events preceding their timestamp should be further expected. They essentially represent a contract between the user and

the SPE to ensure input completeness and output correctness. Watermarks also hold another significance: the progress of the stream in event-time can be interpreted from their frequency of propagation. More specifically, by continuously receiving watermarks, window operators can reason about their progress in terms of input completion.

Watermarks are injected into the stream either (i) at the source, or by (ii) a specific operator that periodically emits watermarks [18, 7]. In both cases, applications decide on their implementation logic for generating watermarks, but typically, they are injected periodically. For example, a periodic watermark can be generated every five seconds holding a timestamp of the current time minus five seconds. In such cases, each watermark can be interpreted to mean that events can be delayed up to five seconds at most. The watermark injection frequency generally is not tied to the input data rate and does not depend on the pipeline size or on the characteristics of its operators (e.g. their window sizes).

To illustrate this with an example, consider Fig. 2.2 where a window operator spanning three seconds starts operating at 0. The stream sequence contains both watermarks (green dashed boxes) and events (blue boxes). The window operator initially receives watermark 1, indicating that no events before 1 are marked to arrive and further indicating window $[0, 3]$ is due for processing next. The window then receives events 2 and 4, each of which is sorted into the appropriate window. Watermark 3 is then processed indicating that it is safe to process the first window to generate its output. The window $[3, 6]$ with event 6 and watermarks 5 and 7 is processed similarly.

In the example of Fig. 2.2, note that different watermarks have different meanings for the window operator. Watermarks 1 and 5 act as a progress indicator for the window, hinting at stream progress. While watermarks 3 and 7 act as progress indicators, they also signal input completion by the window's deadline consequently triggering the operator to compute the corresponding output.

Given a window, we define the first ingested watermark that signals input completion to push the window to produce output as a *sweeping watermark* (SWM). In Fig. 2.2, watermark 3 is the SWM that signals input completion for window $[0, 3]$. Since 5 arrives after watermark 3, 5 is not an SWM for window $[0, 3]$. Watermark 7 is also an SWM as it is the first to signal input completion for window $[3, 6]$. A significant advantage of leveraging OOP and watermarks is that an SWM is inferred based on its arrival at the window. For example, if due to network and pipeline delays watermark 3 was ingested after watermark 5, then 5 is inferred as the SWM for window $[0, 3]$ as it would be the first to signal input completion. Therefore, applications do not need to be concerned with knowing or identifying if any of their generated watermarks are SWMs.

SWMs are important for SPEs since processing them pushes window operators to emit their contents. Two key invariants hold in processing SWMs: (i) propagating an SWM to a window operator implies that all the relevant events have already been collected by the window operator, and (ii) propagating an SWM to the output operator guarantees that all events produced by the relevant window were flushed as output. The first invariant guarantees that all relevant events that precede an SWM have already been ingested and that SPEs do not need to re-order events to deal with input incompleteness. The second invariant is enforced by the order of execution of window operators. More specifically, window operators emit their output events followed by SWMs, which are received by the output operator after the window output events. Therefore, all the relevant events are guaranteed to have been processed beforehand. The aforementioned invariants give rise to the two following important observations:

- Minimizing the end-to-end propagation delay of SWMs implies that the output latency is minimized since the SWM delay is a function of the delay of completing the window's input and the propagation delay of the emitted events to the output operator.
- The propagation delay of SWMs is a factor of the number of events in the stream at the time of ingestion. More specifically, the *cost* of propagating an SWM to a window operator is a function of the number of queued events in the stream. Therefore, it is necessary to process all the events stowed forth in the stream before the ingestion of the SWM to achieve minimized propagation delay.

Thus, it is essential to minimize the propagation delay of SWMs to the output operator to minimize output latency. The aforementioned observations are at the basis of our runtime scheduler design.

2.3 Query Scheduling

Scheduling policies have been proposed in the context of stream processing. The First-Come-First-Serve (FCFS) policy initially proposed for databases and web servers [15] optimizes for the maximum output latency for each request. An adaptation for streaming engines was studied in [88, 87]. Shortest-Remaining-Processing-Time (SRPT) [75] was initially also proposed for web servers to minimize mean output latency but was adapted to streaming engines by estimating the processing time as the cost of processing a single

event times queue size of each operator. This policy laid the foundation for later streaming specific scheduling policies such as the Rate-Based (RB) policy [96] which was later generalized to Highest Rate (HR) to function across multiple queries in [88]. HR delivered better performance than Aurora's scheduler [2] that was expected to function across multiple queries using Round-Robin for inter-query scheduling.

All of these algorithms optimize for output latency by exploiting particular properties of operators. However, in contrast to Klink, they are not optimized for queries employing window operators and thus are effectively agnostic of the semantics of queries and the progress of the stream. As we will demonstrate in Chapter 3, Klink delivers better performance than the aforementioned policies as

With recent interest in both single node (scale-up) and multi-node (scale-out) streaming systems, the importance of effective scale-up [56, 71, 103] through prudent scheduling cannot be overstated; being able to deliver high performance per node adds to the total compute capacity of any system. StreamBox's scheduler proposed in [71] is one example in this vein. StreamBox leverages watermarks to process windows as fast as possible. However, as we demonstrated in the experimentation section, StreamBox does not prioritize streams efficiently since it is agnostic of the load size, is performance-sensitive to watermark frequencies, and does not scale well when a back-pressure mechanism is implemented.

Haren [80] is a portable scheduling framework that distills an abstract API sufficiently flexible to implement multiple scheduling policies. Haren is orthogonal to our work where scheduling policies can be implemented into Haren. However, Haren has not been shown to run on an industrial-strength SPE like Flink. Moreover, the design requirements of Klink, i.e., distributed, and watermark-aware are not supported by Haren.

Real-Time scheduling policies were also studied in the context of stream processing [86, 14]. Such schedulers rely on understanding the performance of the SPE in the context of meeting execution deadlines [64, 55, 21] and then apply algorithms from real-time scheduling theory. However, this class of algorithms suffers from the same issues as the algorithms that are not concerned with real-time scheduling. That is, they do not consider windowed operators and thus impose very high output latencies. Since scheduling operators in data streams requires orchestrating the use of computational resources across the deployed operators in an SPE, one problem is how to deploy the operators across multiple nodes running the SPE [9, 85, 67]. As this focuses on how to effectively distribute the operators across nodes, Klink's design is not targeted to address this orthogonal problem though Klink can function in any deployment.

2.4 Query Sample Processing

Approximate Query Processing (AQP) techniques are generally applied on windowed operators in streaming queries to offer a trade-off on accuracy to optimize for specific performance goals. There exist multiple ways in which AQP achieves this balance.

AQP has been applied in multiple systems. For relational databases, AQP has been studied extensively [63, 5, 27] with techniques based on sampling [6], sketches [42], and aggregations [47]. These techniques try to expedite query processing latency and memory utilization by approximating the results of the query. Such AQP techniques inherently assume that (i) databases have their data stored on disk and can, therefore, sample offline [25], and (ii) read-heavy databases can be exploited to build and store high-quality samples [63]. However, these assumptions do not apply to SPEs as events are ingested continuously, are not stored, and are then processed on the fly. As such, SPEs require different sampling techniques than the ones designed for databases.

AQP has also been studied in the context of SPEs [53, 36]. Sketches has been applied in the context of reducing memory footprint of stateful operators [24, 35, 89, 40]. Sketches utilize various data structures to store statistically significant information about the original input. However, sketches are designed to reduce memory utilization by running expensive algorithms to summarize the input. As such, these techniques that are geared towards reducing memory utilization cannot be efficiently adopted to reducing output latency at the expense of accuracy.

Another form of AQP is sample processing (i.e. processing a sample of the input). Earlier research work [53] introduced and formalized a framework to implement sample processing algorithms on the Gigascope SPE. Examples of common sample processing approaches discussed in [53] include [52, 94, 83, 93, 74] that optimize for different goals in the system. For instance, [12] gracefully degrades performance by dropping events when the system is overloaded. The proposed algorithm computes the minimum shedding rate such that system resources are not over-utilized while still maintaining adequate query accuracy. Load-Aware Shedding (LAS) presented by [83] is a proactive load shedding mechanism that aims to limit queue latencies. LAS utilizes learning techniques thereby advancing a model-free algorithm to assess the cost of processing events. Another approach is presented in [52] whereby the sample size is estimated for aggregations with dynamic slack time for window processing. However, these methods generally wait for the arrival of watermarks before processing the sample. As previously illustrated, methods that include stragglers for their input are prone to suffer from extremely high output latency.

Sample processing has also been studied in different contexts. IncApprox [57] is an

approximation method that studies the problem of having the deployed SPE ingest events from multiple sources, each of which is emitting at a different rate. IncApprox presents a solution that represents the data received from each source as *strata* through stratified sampling. The weight of each stream is then computed based on the arrival rate. Reservoir sampling is then applied to each source which is used for approximation. However, IncApprox is designed for systems relying on emitting mini-batches as in Apache Spark [102]. StreamApprox [82], an evolution of IncApprox, relaxes this assumption by presenting a system designed for both mini-batches and event-at-a-time processing paradigms. While Aion does not discriminate between stream sources, the algorithm can be easily extended by stratifying each stream source as described in StreamApprox and IncApprox. These papers assume that the user is knowledgeable enough to provide a static sample size that minimizes the error rate, raising practicality concerns. Additionally, all of these algorithms suffer from the stragglers problem.

There exists a multitude of techniques proposed to include stragglers by imposing extra latency to ensure input completion [36, 41, 91, 76, 101, 52, 84]. Heartbeats [91] are watermark-like events injected in the stream to signify end of sub-streams. This algorithm assumes static maximum latency delay and consequently is prone to suffer from high latency. Similar work [76] proposed a watermark generation algorithm that processes an event only if has been idle in the stream for k seconds where k is set to the maximum observed network delay. More recently, [84] proposed an algorithm with probability guarantees on slacking for stragglers by monitoring both network and inter-event generation delays. However, these algorithms present a solution against the uncertainty of the number of stragglers by imposing slack delay. In contrast, Aion's novelty lies in its observation that stragglers are unproductive to sample processing, thereby circumventing them and minimizing slack delay to deliver minimized output latency.

Chapter 3

Query Scheduling

Through the design and development of Klink, we leverage ingested watermarks to robustly infer stream progress based on window deadlines and network delay, and to assign and schedule query pipeline execution that reflects stream progress. Klink aims to unblock window operators and to rapidly propagate events to output operators while performing judicious memory management. In this chapter, we introduce the design of Klink (Sections. 3.1 and 3.2). We also present Klink integration into the popular open source SPE Apache Flink (Section. 3.3) and demonstrate that Klink delivers hefty performance gains on benchmark workloads, reducing mean and tail query latencies by up to 60% over existing scheduling policies (Section. 3.4).

3.1 Klink: Design and Algorithms

We now present the design of Klink including its algorithmic details. Fundamentally, Klink is composed of (i) a stream analysis algorithm that computes a priority for each stream such that SWM propagation delay is minimized (Sections 3.1.1 and 3.1.2), (ii) a memory management algorithm that ensures Klink has sufficient memory resources to run its scheduling policy (Section 3.1.4), and (iii) a distributed design that enables the priority to be computed at each node and then propagated in a fully decentralized manner, enabling scaling with the SPE infrastructure (Section 3.2).

Initially, and before receiving any events as shown in Algorithm 1, Klink examines the semantics of the set of deployed queries (including parallelization of operators) denoted by $\mathcal{Q}$. Query-level scheduling is performed to reduce the complexity of the scheduling

Algorithm 1 Klink's inter-query scheduler main loop

```
1: procedure KlinkLoop(𝒬, r)
2:     while true do
3:         ℐ ← acquireRuntimeData(𝒬)
4:         q ← klinkEvaluator(𝒬, ℐ)
5:         executeQuery(q)
6:         sleep(r)
7:         if memory_utilization ≥ b then
8:             runMemoryManagement(𝒬)
9:         end if
10:    end while
11: end procedure
```

algorithm, and to yield a schedule of subsequent operators capable of processing the flow of events end-to-end.

Calculating the priority of each query requires the scheduler to maintain runtime characteristics that include the network delay, queue size, cost, and selectivity. The window(s) with the latest deadline in query q at time t that will unblock the stream are selected by the algorithm to yield output events. Based on the deadline of each query q, Klink logically divides the stream into *epochs* whereby each epoch is demarcated by an SWM. Specifically, the $(n+1)^{th}$ epoch starts after the ingestion of the n^{th} SWM. For example, Fig. 2.2 contains two epochs: The first defined between time 0 and SWM 3, and the second between SWMs 3 and 7. Note that the second epoch overlapped with two windows, i.e., tumbling window $[3, 6]$, and the next $[6, 9]$. Similarly for a sliding window of size 5 seconds and slide 1 second, the first epoch is defined between time 0 and SWM 5, the second and the third epochs between SWMs 5 and 7. Thus, with each converging deadline, Klink progresses the query to the next epoch. Klink also groups the collected information on a per-epoch basis to enrich future estimations with temporal context.

The collected information is continuously updated to infer the expected ingestion time of the next SWM and the expected emission time of the window operator. For query q at epoch n, we denote by $\mathcal{D}_n^q$ the set of cumulative network delays incurred by each event. The network delay can be estimated simply by the ingestion timestamp of the event minus its generation timestamp. In addition, Klink also maintains the total number of events queued, and the cost of executing them end-to-end, represented by $cost^q(t)$. As in [64], cost is estimated by considering both *selectivity* (ratio of output events to an input event per operator) and operator processing time (time taken to process a single event per operator).

As discussed in [64], the cost of processing an event can be accurately represented by using the standard measures of per operator mean processing latency, queueing delays, and communication latencies between operators. These factors for estimation of the processing cost are encapsulated into the tuple $\mathcal{I}$ (line 3 in Algorithm 1). The priority of each query is then computed based on the retrieved information (line 4). The query with the highest priority is returned and is then scheduled for execution (line 5).

To keep the overhead to a minimum, Klink is inactive while the operators are executing, and recommences only when the operators finish their planned execution. Then, Klink re-evaluates the priorities and selects new queries to execute (lines 2-10). Every such evaluation round, or *cycle*, runs for r milliseconds. In general, a small value of r is expected to incur higher overhead while a large value implicates missing the deadlines for idle queries. As we demonstrate in Section 3.4, Klink's scheduling overhead is meager.

Klink's priority evaluator (line 4) aims to minimize the propagation delay of SWMs. This is achieved by selecting queries with lowest *slack*, defined by the idle time a query can mask without processing its queued events to avoid missing deadlines. For an epoch n, we express the slack time for query q at time t by:

$$sl^q(t) = (w^q_{n+1} - t) - cost^q(t) \tag{3.1}$$

where w^q_{n+1} represents the ingestion time of the $(n+1)^{th}$ SWM. Thus, as the stream progresses towards ingestion time, the query's slack value attenuates. The query with least slack is then selected for execution.

To estimate the slack time (Eq. 3.1), it is essential to determine w^q_{n+1}. We describe Klink's robust estimation technique next.

3.1.1 Estimating SWM Ingestion

Klink estimates the ingestion of SWM for query q at epoch n by two important factors: the expected network delay denoted by the random variable d^q_n, and the periodicity of SWMs p^q. The estimated ingestion time for the $(n+1)^{th}$ SWM is represented by:

$$E[w^q_{n+1}] = E[d^q_n + p^q] \tag{3.2}$$

To proactively compute $E[d^q_n]$ before the collection of all events pertaining to the n^{th} epoch, Klink relies on historical data captured during the previous epochs to profile the newest epoch. This allows Klink to estimate the arrival times of the SWM at the beginning

of each new epoch. The accuracy of the estimation then increases with the progression of the stream as long as the query is continuously ingesting events and is monitoring the network delay. Once the epoch is finalized, d_n^q as a random variable is then represented by only events constituting the n^{th} epoch. We compile this definition into the following equation:

$$\mu_n^q = \begin{cases} \frac{1}{|\mathcal{D}_n^q|} \times \sum_{d \in \mathcal{D}_n^q} d & \text{if } t \geq w_n^q, \\ \frac{1}{n-1} \times \sum_{i=0}^{n-1} \mu_i^q & \text{otherwise.} \end{cases} \tag{3.3}$$

We define χ_n^q as the square of each distribution in the following equation:

$$\chi_n^q = \begin{cases} \frac{1}{|\mathcal{D}_n^q|} \times \sum_{d \in \mathcal{D}_n^q} d^2 & \text{if } t \geq w_n^q, \\ \frac{1}{n-1} \times \sum_{i=0}^{n-1} \chi_i^q & \text{otherwise.} \end{cases} \tag{3.4}$$

For the case of $t < w_n^q$, we observe that the random variable d_n^q is a function of the expected delay over the previous epochs. Hence, by the Central Limit Theorem, d_n^q is normally distributed, and in turn, w_{n+1}^q is also normally distributed. To better understand this distribution, we calculate the mean (Eq. 3.5) and variance (Eq. 3.6) of w_{n+1}^q:

$$\begin{aligned} E[w_{n+1}^q] &= E[d_n^q + p^q] \\ &= \frac{1}{n-1} \sum_{i=0}^{n-1} E[d_i^q] + E[p^q] = \mu_n^q + p^q \end{aligned} \tag{3.5}$$

$$\begin{aligned} Var[w_{n+1}^q] &= E[(w_{n+1}^q)^2] - E[w_{n+1}^q]^2 \\ &= \frac{1}{n-1}[\chi_n^q + \frac{1}{n-1} \sum_{0 \leq i \neq j}^{n-1} \mu_i^q \mu_j^q] - (\mu_n^q)^2 \end{aligned} \tag{3.6}$$

Thus, w_{n+1}^q is normally distributed with mean $\mu_n^q + p^q$ and variance denoted by Eq. 3.6. By exploiting this distribution, we can estimate a time-interval to conclude the range of SWM's ingestion timestamp. This is discussed further in the next section (Sec 3.1.2).

Algorithm 2 presents an efficient algorithm to compute the mean and variance of each epoch. The procedure ProcessEvent is triggered for each event processed. First, the delay expressed by event e is added to $\mathcal{D}_n^q$ (line 2). Then, the arrays that maintain information over epoch n are updated to include the new entry (line 3-4). Note here that μ_n^q and χ_n^q hold temporary values as the stream is still progressing. The second procedure, NextEpoch, is triggered at the arrival of each SWM (line 7). To reduce memory consumption, the presented algorithm is optimized to express $\mathcal{D}_n^q$ over only the last m epochs (line 8). Then,

Algorithm 2 Klink's scheduler epoch-control

1: **procedure** ProcessEvent(q, e)
2: $\mathcal{D}_n^q \leftarrow \mathcal{D}_n^q \cup \{e\}$
3: $\mu_n^q = \mu_n^q + d$
4: $\chi_n^q = \chi_n^q + d^2$
5: **end procedure**
6:
7: **procedure** NextEpoch(q)
8: $\mathcal{D}^q = \mathcal{D}^q \setminus \mathcal{D}_{n-m}^q$
9: $\mu_n^q = \mu_n^q \ / \ |\mathcal{D}_n^q|$
10: $\chi_n^q = \chi_n^q \ / \ |\mathcal{D}_n^q|$
11: $n = n + 1$
12: **end procedure**
13:
14: **procedure** ProcessSWM(q, e)
15: ProcessEvent(q, e)
16: NextEpoch(q)
17: **end procedure**

μ_n^q and χ_n^q are computed and finalized (line 9-10). Finally, the last procedure ProcessSWM signals the arrival of a new event with delay e (line 15), and then q is progressed to the next epoch. Studying the distribution of w_{n+1}^q allows us to estimate the arrival of the SWM with high confidence. We present in the next section an algorithm that exploits this characteristic to compute the slack time sl^q for query q.

3.1.2 Estimating Slack Time

To compute the slack time of each query, Klink initially computes the likelihood of SWM ingestion for each possible time-range that spans the execution duration. That is, a sliding window of size r is passed through the time-range at which the SWM can be ingested. Then, for every window spanning r milliseconds, the likelihood of SWM ingestion at that time is computed. The slack value is then computed based on the likelihood of each range. We discuss the details of Algorithm 3 in the rest of this section.

The algorithm starts first by sliding the window of size r over a time-range at which the SWM is expected to be ingested. However, since this range may span a lengthy duration, we optimize the algorithm's performance by limiting the range to a provided confidence

value f. This is represented by the following equation:

$$P(t^q_{n,min} \leq w^q_{n+1} \leq t^q_{n,max}) = f \tag{3.7}$$

where $t^q_{n,min}$ and $t^q_{n,max}$ encloses a time-range where the probability of w^q_n falling in this range is f. Note here that selecting a small interval would yield less accurate slack estimations while selecting a large interval would lead to performance inefficiencies. Thus, an f value needs to be selected (discussed further in Sec 3.4).

After delimiting the search space, Klink slides a window of size r over the time-range computed by (Eq. 3.7). Then, for each window slide, the slack value is computed as:

$$\begin{aligned} sl^q(t) = \sum_{x=t^q_{n,min}}^{t^q_{n,max}} & P(x \leq w^q_{n+1} \leq x + r|t) \\ & \times((x + r - t) - cost^q(t)) \end{aligned} \tag{3.8}$$

where x is incremented by r milliseconds.

Eq. (3.8) illustrates that the slack time for query q is calculated by computing the likelihood of ingesting the SWM in each time-range, then calculating the slack value for that ingestion time-range. The conditional probability (in Eq. 3.8) can be computed as:

$$P(x \leq w^q_{n+1} \leq x + r|t) = \begin{cases} \frac{P(x \leq w^q_{n+1} \leq x+r)}{P(w^q_{n+1} \geq t)} & \text{if } x \leq t, \\ 0 & \text{otherwise.} \end{cases} \tag{3.9}$$

Since w^q_{n+1} is normally distributed, the probabilities can be approximated by the Guassian Q-function as in Eq. 3.9:

$$\begin{aligned} & P(x \leq w^q_{n+1} \leq x + r) \\ & = Q(\frac{E[w^q_{n+1}] - (x + r)}{Var(w^q_{n+1})}) - Q(\frac{E[w^q_{n+1}] - x}{Var(w^q_{n+1})}) \end{aligned} \tag{3.10}$$

We detail the aforementioned slack time estimation in Algorithm. 3. Initially, the procedure KlinkEvaluator (line 19) is invoked by Algorithm. 1 (line 4) where the set of queries and collected information are passed as parameters. The algorithm then unpacks the collected runtime information (line 20). The set contains important information per query q: upcoming window deadline, last watermark processed, experienced network delays $\mathcal{D}^q$, and the data collected over previous m epochs μ^q_n and χ^q_n. The set also contains

Algorithm 3 Klink slack computation

procedure ComputeConfInterval(q)
  $\mu_n^q = \frac{1}{m} \sum_{i=n-m}^{n-1} \mu_i^q$; $\chi_n^q = \frac{1}{m} \sum_{i=n-m}^{n-1} \chi_i^q$
  $E[w_{n+1}^q] = \mu_n^q + p^q$; $\sigma[w_{n+1}^q] = \sqrt{(Eq.\ 3.6)}$
  /* *Compute* $\geq 95\%$ *interval* */
  $t_{n,min}^q = E[w_{n+1}^q] - 2\sigma[w_{n+1}^q]$
  $t_{n,max}^q = E[w_{n+1}^q] + 2\sigma[w_{n+1}^q]$
  return $t_{n,min}^q$,$t_{n,max}^q$
end procedure
procedure ComputeExpectedSlack(q, t) ▷ Eq. (3.8)
  $t_{n,min}^q$, $t_{n,max}^q$ = ComputeConfInterval(q)
  $sl^q = 0$; $x = max(t, t_{n,min}^q)$
  for $x \leq t_{n,max}^q$ **do**
    $pr = \frac{P(x \leq w_{n+1}^q \leq x+r)}{P(w_{n+1}^q > t)}$
    $sl^q = sl^q + pr \times [(x + r - t) - cost^q(t)]$
    $x = x + r$
  end for
  return sl^q
end procedure
procedure KlinkEvaluator($\mathcal{Q}$, $\mathcal{I}$)
  unpack($\mathcal{I}$); $min\_sl = 0$; $min\_q = null$
  for each query q **do**
    sl^q = ComputeExpectedSlack(q, t)
    if $sl^q \leq min\_sl$ **then**
      $min\_sl = sl^q$; $min\_q = q$
    end if
  end for
  return min_q
end procedure

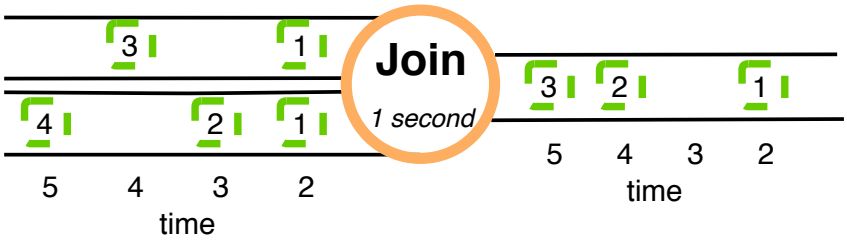


Figure 3.1: Example illustrating a window operator joining two input streams of SWMs into an output stream of SWMs.

information per operator including: mean selectivity, mean processing cost, current queue size, and current memory utilization. The procedure then loops over all deployed queries to compute the slack for each (line 24). The query with the minimum slack time is then selected and returned for execution (line 24 and line 27). To compute the estimated slack, first the confidence value is calculated (line 1). The confidence value in this function is calculated by estimating the mean, and the standard deviation for SWM ingestion (line 2). For high accuracy, this pseudo-code runs for a default $f = 95\%$ confidence value. After computing $t^q_{n,min}$ and $t^q_{n,max}$, the slack is then computed as in Eq. 3.8 (lines 12-16). At first, the algorithm divides the time-range into smaller ranges whereby each small range probability is calculated (line 13). Then, sl^q is updated (line 14), and the smaller-range is then translated by r milliseconds. Finally, the slack value is returned (line 17).

3.1.3 Handling Join Operators

Klink is designed to rapidly unblock window operators by prioritizing the input streams based on the anticipated arrival of SWMs. For *join* operators that perform windowed joins over multiple input streams, the join operator is unblocked once all input streams propagate an SWM elapsing the window deadline. This is done to ensure correctness – by guaranteeing that the relevant elements were propagated through all input streams – and is typically achieved by (i) maintaining the last watermark propagated by each stream, then (ii) computing the minimum watermark timestamp, and (iii) comparing it to the window deadline to evaluate if it unblocks the operator. In this section, we discuss a mechanism implemented by Klink to efficiently handle join operators.

To illustrate joins, consider the Fig. 3.1 example of a 1-second window operator joining two input streams into one output stream. At time $t = 2$, two SWMs of equivalent

timestamp of 1 were ingested effectively unblocking the operator and pushing the SWM to the output stream. Note here that the value of the SWM in the output stream implies that no event before 1 is further expected by the two input streams. At time $t = 3$, an SWM of timestamp 2 was ingested. However, the window of $ddl = 2$ was not unblocked until the ingestion of SWM 3 at the top stream. Although SWM 3 unblocked the window of $ddl = 2$, it did not unblock the window of $ddl = 3$ since no SWMs were propagated by the bottom stream. The window of $ddl = 3$ is then unblocked at the ingestion of SWM 4 from the bottom stream.

Although watermarks propagating through each input stream can be perceived as SWMs in the context of window deadlines, they do not necessarily unblock the window operator. This is problematic as an input stream can be scheduled for execution yet the deadline can be extended well beyond the anticipated time of the SWM in the other input streams. Consequently, computations are hindered, translating to delayed SWM propagation in other queries. As such, it is important to maintain accurate prioritization of each input stream and account for the different rates of watermarks' propagation.

Klink solves this problem by computing multiple different slack values, one for each input stream that has different SWM period rate and network delay. The slack of the query is then calculated as the minimum of each stream. Thus, the procedure call in Algorithm 3 (line 27) returns the minimum slack for query q. This design ensures that accurate prioritization is determined and maintained.

3.1.4 Klink's Memory Management

While latency optimization is a prominent goal in stream processing applications, workloads can impose heavy memory constraints on SPEs from a resource utilization viewpoint. For example, operators that store transient state while processing events require extra memory that may not be available. Operator instances will likely contend on scarce memory resources, thereby blocking stream flow and increasing output latency. These effects can degrade performance by increasing latency [11, 23, 38]. The common approach used in many SPEs is to implement a backpressure mechanism that throttles the input rate to ease memory utilization on the system. Unfortunately, this simple heuristic approach negatively impacts output latency by slowing down the whole stream.

Klink introduces a new approach to address memory utilization stress that, differently from existing backpressure mechanisms, prudently exploits information on how the running application works. The fundamental idea is to prioritize the flow of events toward low

selectivity operators to reduce the number of events "in flight" in the application and thus alleviate the overall memory usage. For example, a filter operator that has the likelihood of filtering one every four observed events can reduce memory utilization by 25%. Window operators that support partial computations (e.g. aggregations that can be computed online, or online joins [62]) can reduce memory utilization before emitting their output, and exhibit low selectivity when they ingest SWMs. Klink leverages these characteristics through prioritizing the execution of queries that contain operators with large queue size, have low selectivity, and support partial computations, i.e., the queries that provide the largest potential reduction in memory utilization.

Initially, for each query q, Klink looks up operators having selectivity[1] values lower than 1, such as filter and window operators. Then, Klink computes the number of queued events that would be reduced by processing all events queued in ancestor operators downstream to the k^{th} operator. To express this mathematically, we denote by sz_k^q the number of queued events from the first non-source operator of q until the downstream operator k, and by S_i^q the selectivity of the i^{th} operator in query q. Then, the number of events processed can be expressed by $p_k^q = sz^q \times (1 - \prod_{i=1}^{k} S_i^q)$, where p_k^q refers to the number of processed events obtained from scheduling pipeline q downstream to the k^{th} operator.

The intuition behind this metric is to schedule the sequence of operators that would provide the largest reduction in the number of events. However, since Klink runs queries for r milliseconds before commencing the subsequent scheduling cycle (Alg. 1, line 6), Klink computes the number of events that can be processed within r by factoring in the cost of each operator. After identifying all pipelines that maximize the value p_k^q, Klink selects the query with the least slack, thereby optimizing also for output latency. The selected sequence of operators are then scheduled for execution.

Klink runs its memory management algorithm only when memory utilization reaches a level at which Klink's least slack policy experiences lessened effect. When a memory usage threshold b is reached such that operators would start contending on memory (Alg. 1, lines 7-9), Klink activates its memory management algorithm. Klink guarantees that each cycle of the algorithm runs for only a targeted period specified by a set memory availability percentage or by a specified time interval. For example, the memory manager can run until half of the consumed memory has been freed or after three seconds have elapsed. We discuss sensitivity of these values in the evaluation section (Sec. 3.4). While reducing memory utilization instead of slack may not always reduce output latency, it ensures that Klink can continue to apply its least slack policy while attaining robust low mean and tail

[1]Selectivity value is retrieved from the SPE or can be computed through maintaining the ratio of output events to an input event per operator.

output latencies, as demonstrated in Sec. 3.4.

3.2 Distributed Klink Design

SPEs leverage distributed computing to achieve greater scalability and meet performance goals [79]. Distribution is typically attained at the granularity of operators where SPEs disseminate them across the available resources. Each compute node would then have a subset of the operators that, collectively, would execute the query exactly as they had been deployed on a single node. Klink is designed to be decentralized by running autonomously and limiting its scope to the deployed queries. Klink, however, maintains common prioritization targets shared across all nodes.

The goal of distributed deployment is to distribute query operators across nodes running the SPE. Initially, applications deploy their *logical* queries to the SPE that accordingly devises a *physical* plan that establishes one-to-one mappings between the operators and the compute nodes [9]. Klink functions orthogonally to the deployment problem and is designed to work with any physical plan.

Fundamentally, Klink's primary goal is to minimize the propagation delay of SWMs through minimizing the number of events queued in the stream before the arrival of SWM. It does so by scheduling the sub-parts of the query that is localized on all nodes. However, distribution imposes the challenge of not having all the necessary information to compute the global priority of the query. To achieve this goal, Klink initially identifies operators that constitute the logical query and maintains the deployment location of each operator over all Klink instances. This is supported in the distributed design by line 9 of Algorithm 1. Klink forwards the necessary information to the corresponding nodes including the network delay, its frequency, and the cost at each node. However, not all information is needed by all the nodes. We illustrate this in the following example.

Consider Fig. 3.2 that shows a query partitioned over two different nodes (machines) A and B. Node A contains the first two operators of the query while Node B contains the last two operators. Once node B attempts to assess the priority of the localized query subset, the results will not be optimal since the necessary information such as network delay stats from the first two operators are unavailable. However, the two nodes have sufficient information to assess the cost of executing the query starting from $O3$ to the output operator. Hence, only the watermark related information needs to be forwarded from node A. On the other hand, the Klink instance running on

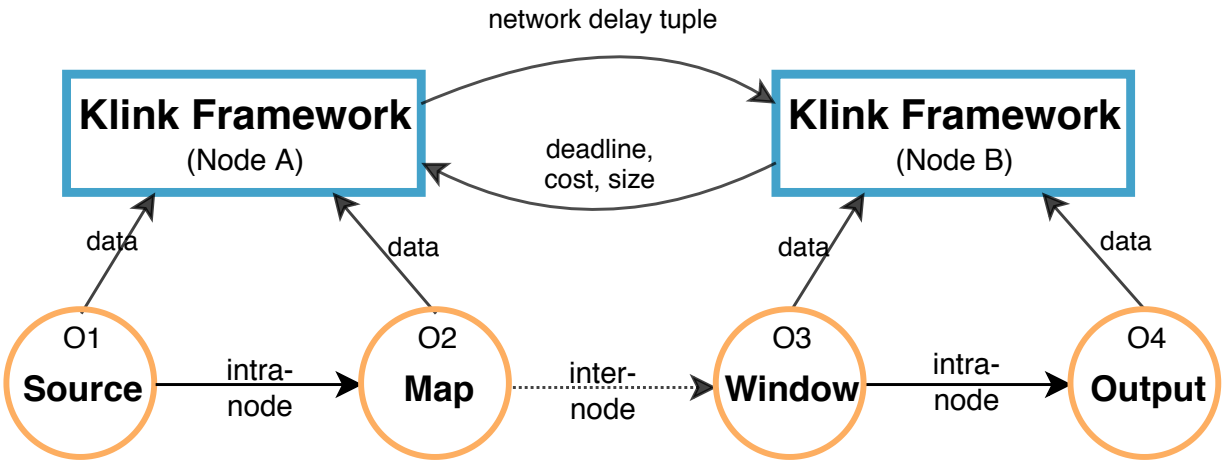


Figure 3.2: Example showing how Klink forwards information in a distributed environment.

node A needs to gauge the true processing of the query by acknowledging the status of queued events downstream. To achieve this, Klink ensures that all nodes that contain subsequent operators (which in this case is only B) send their cost information to A so that A can safely measure the priority value of the query.

To generalize Klink's *information forwarding*, consider the following cases:

- Network delay forwarding: All nodes require estimated network delays value to estimate the priority of the sub-query. However, to reduce the scheduler overhead and data transmission, we consider the following optimizations. If the application query generates watermarks at the source, then they will be observed first by the source operator. Hence, w^q_{n+1} is guaranteed to be localized on the node containing the source operator, so information should be forwarded to all nodes that hold the subsequent operators. If it is the case that the watermark is generated by an operator in the pipeline, then the node that contains that operator forwards this information to all nodes containing subsequent and previous operators. Otherwise, all watermark timestamps are shared and the maximum value is selected.

- Cost forwarding: Nodes need to assess the cost of executing the information only downstream. Hence, prior knowledge of earlier nodes in the query is not required. As such, every node transmits the necessary information to the nodes containing any of the downstream operators. The cost of the operators is then incorporated into the cost function.

Information about network delay and cost forwarding is sent between nodes where any one node receives local information from only one other node. Thus, this limits and reduces dependencies – no node needs to rely on multiple nodes for information, avoiding both synchronization with multiple nodes and global dependencies. Once information are forwarded to nodes, each Klink instance computes its priorities and executes accordingly. The scheduler collects information from other Klink instances as part of line 9 in Algorithm. 1. We discuss Klink's implementation details in the following section.

3.3 System Implementation

Klink is implemented within the open-source SPE Apache Flink [18]. Flink relies on the OS scheduler to resolve the operator scheduling problem. To provide support for developers to implement custom scheduling policies, we added into Flink a framework to support the implementation of other scheduling policies at the runtime level. This section describes the integration of Klink into Flink and Klink's generalization to other SPEs.

Flink is a distributed stream processing engine that is efficient, scalable, and fault-tolerant. Klink is implemented in JAVA as a layered system where a logical separation is exhibited between the different layers. The three main layers are named Deploy, Core and API. The top layer, API, contains two segregated engines namely DataStream and DataSet pertaining to stream and batch processing, respectively. At the Core layer, Flink operates at the granularity of *Tasks* where each task is a standalone thread. Tasks are then transformed to either operators or a chain of operators at the API layer. Flink provides no infrastructural support to design or implement a runtime scheduling policy. In fact, Flink solely relies on the OS scheduler to orchestrate the execution of Tasks. To implement the Klink algorithm, we integrated a custom runtime scheduler into Flink.

Several SPE architectures that include runtime schedulers have been proposed [19, 80, 1]. In contrast to a thread-based execution model where each incoming event is allocated a thread, Aurora [19] and Borealis [1] use a *state-based* execution model for its runtime scheduler. Specifically, a single scheduler thread that tracks system state is deployed to orchestrate the execution of threads. In this design, each operator instance is mapped to a thread. The scheduler design adopted by these two systems showed that state-based schedulers are better suited for SPEs than thread-based schedulers. Thus, we integrated a state-based scheduler infrastructure into Flink to maximize efficiency.

Fig. 3.3 illustrates the architecture of the Klink framework. All created tasks have to register with the *RuntimeScheduler* instance, which is a standalone thread initially

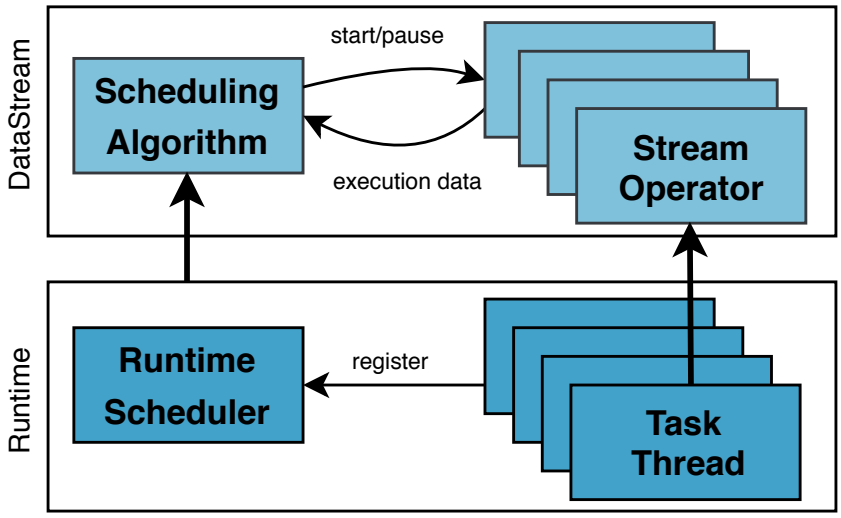


Figure 3.3: Klink architecture and API within Flink

launched prior to the creation of any task. After analyzing the semantics of the deployed queries,[2] the RuntimeScheduler component invokes the *Scheduling Algorithm* component that controls the execution of operators via the two methods *start* and *pause*. Stream operators continuously provide the Scheduling Algorithm component with the necessary execution data for prioritization purposes as described in Section 3.1.

Through implementing two additional components into Flink's Core layer, we integrated a state-based scheduling framework capable of running any scheduling policy. Specifically, we implemented (i) a scheduler responsible for orchestrating operator execution and retrieving runtime information i.e., $\mathcal{I}$ (line 25, Algorithm. 3), and (ii) an independent policy component that leverages the collected information to determine a scheduling execution order. The first component is designed with four main API calls: *register*, *collect*, *start*, and *pause*. Initially, each Task must invoke the *register* API call to inform the scheduler of its existence. Then, the scheduler will continuously invoke the *collect* API call with each operator to collect the necessary runtime information (Algorithm 1, line 9). The runtime information then will be passed to the second implemented component, where the Klink scheduling algorithm is deployed to determine the set of new Tasks to be executed (line 11). Finally, the runtime scheduler will *pause* the current Tasks running, and will *start* the execution of those new Tasks (line 13).

Klink's distributed design exchanges collected runtime information across nodes, as

[2]Per Section 3.1.

described in Section 3.2. We implement this functionality by running a remote procedure call (RPC) service as a background thread on each node that serves to transfer data between nodes. The RPC service is instantiated by the *JobMaster* (the master component of Flink's distributed architecture) to facilitate communication between different nodes. The runtime scheduler provides to the JobMaster information to be sent.

Since modern SPEs share similar design architectures, the design of our runtime scheduler can be easily ported into those engines. For instance, Klink can be integrated into Apache Storm [95] by implementing the four aforementioned system calls into Storm *Bolts*. Specifically, after a topology has been submitted and a Storm *Supervisor* has been created, a scheduler runtime instance can be instantiated that Bolts would register with. As in Flink Metrics API, Klink could retrieve operators' information through accessing stored information on each *Worker*. Klink could also be implemented in distributed mode over Storm through implementing the same RPC service over each Worker.

3.4 Performance Evaluation

In this section, we present the results of a series of experiments we conducted to demonstrate the performance advantage that Klink possesses over the algorithms from prior related work on single and multi-node environments, and then analyze its overhead.

3.4.1 Experimental Setup

Our experiments are run on a cluster of nodes each having an Intel Xeon processor with 24 cores (using hyper-threading) and 32 GB of memory. Each machine is running Java OpenJDK implementation v1.8.0_191 on top of Ubuntu 16.04 LTS. Test are performed on an implementation of Klink based on Apache Flink v1.8. One machine is dedicated to workload generation. Input data is then transmitted to the SPE nodes via Kafka v2.2.1.

Benchmarks

We conduct our evaluation using three well-known streaming benchmarks: the Yahoo! Streaming Benchmark (**YSB**) [29], Linear Road Benchmark (**LRB**) [10], and the New York City Taxi (**NYT**) benchmark [51]. We implement these benchmarks on Apache Flink and evaluate performance by running different scheduling policies in each experiment.

LRB simulates a highway toll system [10]. We use the streaming variation [50] of LRB that has a complex pipeline structure that includes a mix of tumbling windows, sliding windows, and join operators. The sliding window is of size five seconds with a slide of three seconds. It also contains a join operator that joins three streams. The workload is generated using the original driver data from [10]. We implement the accident detection and toll calculation queries that utilize windows. To study performance when the pipeline is stressed, we ran LRB with the deadline of the last window operators to be one third of the earlier window deadlines so that the pressure on the query pipeline will be intensified at SWM ingestion. NYT covers a large dataset of taxi trips in New York spanning six years. The dataset is rich with information such as the number of passengers, distances, and fares. This NYT aggregation query over real-world data is composed of a sequence of stateless operators and a sliding window of size two seconds and a slide of one second.

Performance Metrics

We compare the algorithms using mean latency, tail latency, throughput, and slowdown. Output latency reflects the time taken by the SPE to materialize results. To measure SPE latency with minimal overhead, we inject into the stream events called *latency markers* that are specially used to measure the propagation delay from the source to the output operators. Latency markers are originally generated by the source operators, queued with the other events, and are then processed by the stream operators. To reflect the actual output latency incurred, we measure the propagation delay of SWM as indicative of the latency at which an SPE is able to produce output events. Latency is measured by subtracting the SWM timestamp at which it was received by the output operator from the timestamp of its ingestion. In our experiments, we emit a latency marker from each source every 200 ms as our tests showed that a lower frequency is an inadequate representation of the SPE's performance while a higher frequency imposes extra overhead without additional benefit. We also measure throughput by the aggregate number of events processed per second by each operator. Finally, slowdown [88] is a metric used to extract the overhead portion from latency by dividing it by the ideal processing time. This metric is measured by the propagation delay of SWMs divided by the aggregation of the execution cost of processing a single event at each operator.

Scheduling Algorithms

In addition to the default Flink scheduler (Default), we compare against two other state-of-the-art algorithms that we implemented into the runtime scheduler (Sec 3.3):

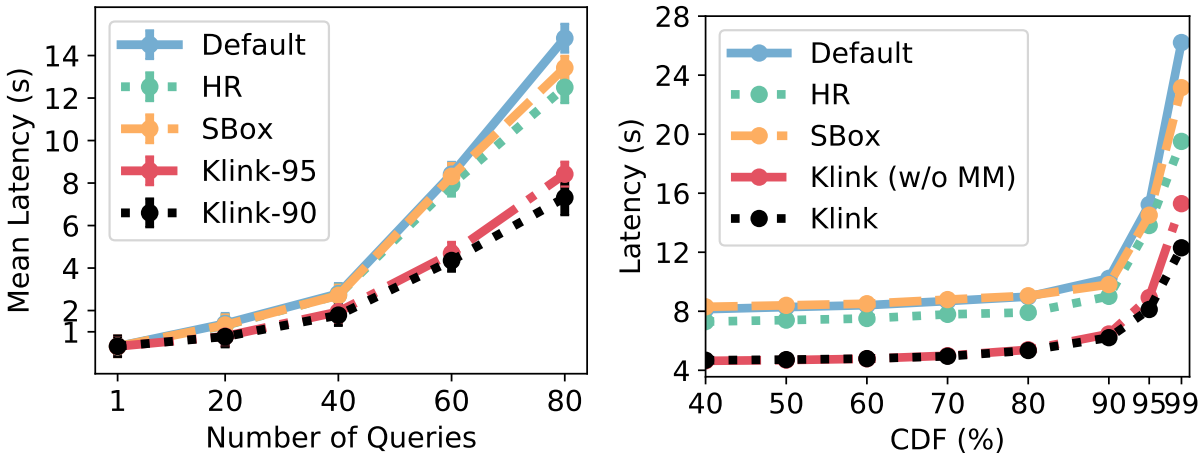
- Highest Rate (HR) [88] aims to minimize the average propagation delay of events across multiple queries running in the system. HR assigns priority based on the granularity of paths. The priority of each path is equal to the global output rate, which is represented by the selectivity of the operator (number of output events per a single input event) and the execution cost (duration of execution of a single input event). This policy prioritizes paths with higher productivity.
- The StreamBox (SBox) algorithm [71] strives to minimize the output latency of scale-up systems. The algorithm initially looks up the query with the closest window deadline, then schedules the query for execution until a watermark is processed. Hence, queries that are expected to emit their content are scheduled.

3.4.2 Results

To evaluate the performance of Klink, we extensively test its performance over five different experiments. Each experiment lasts 20 minutes. Each data point on the graph is an average over at least 10 independent runs (unless stated otherwise) with 95% confidence intervals shown as error bars around the means. We also generate Zipf distributed network delays with a distribution constant of 0.99 [84, 16]. Based on our empirical experimentation, we set Klink's size of epochs history m to 400 and the scheduling cycle duration r to 120 ms for robust performance.

YSB Benchmark

The first experiment compares the performance of the default Flink scheduler, denoted by Default, as well as the three scheduling algorithms (HR, SBox, Klink) on a single node running the YSB benchmark with Uniformly distributed network delays. Each query instance is deployed at a randomized time in the first 20 seconds of the experiment to randomize the uniform distribution of the window deadlines. We measure the latency cumulative distribution function (CDF) to study the tail latency differences obtained by setting the number of events generated to $10,000$ per second per query, and the number of deployed queries to 60. We also include results of Klink without our memory management algorithm but with the back pressure mechanism being able to kick in, which we name 'Klink (w/o MM)' and study its impact on tail latency in comparison to Klink (with memory management), per (Sec 3.1.4).



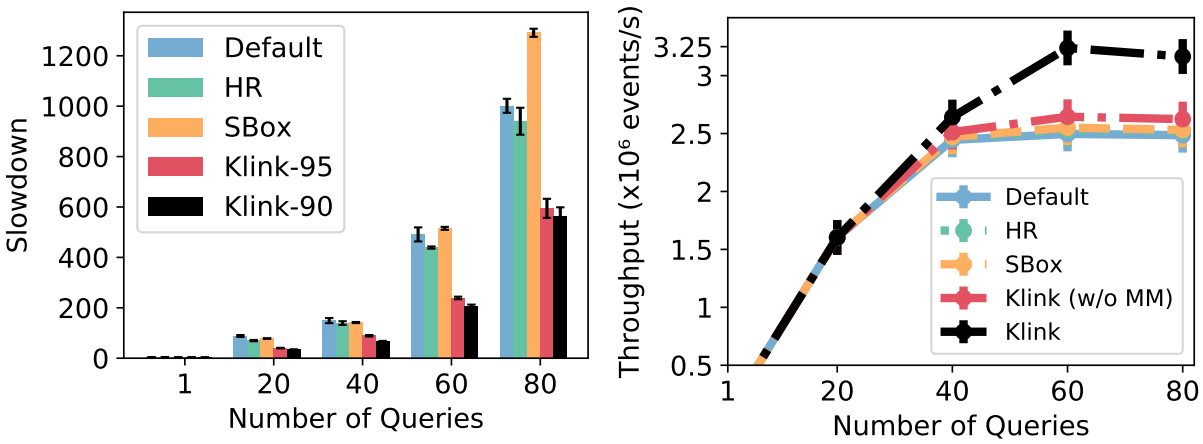
(a) Mean latency

(b) Latency CDF

Figure 3.4: Mean latency and CDF for YSB workload

With increase in the number of deployed queries in Fig. 3.4a, the mean latency for Klink is capped at 7.3s, halving the delay over Default to provide large performance improvements of about 50% over Default and SBox, and 45% over HR. SBox and HR only marginally increase their performance over Default by reaching an output latency of 12.8s and 13.5s compared to Default's 15s for 80 concurrent queries. Since none of these scheduling algorithms factor in window deadlines, they do not exhibit much of a performance difference with each other. Interestingly, Klink with 90% confidence value (Klink-90) achieves higher performance than Klink with 95% confidence value (Klink-95). While Klink-90 and Klink-95 achieve similar levels of estimation accuracy, the overhead of 90% is lower as discussed later in Fig. 3.9a.

Fig. 3.4b compares the output latency CDF of Klink with the other scheduling algorithms and, in particular, shows their tail latency performance. All scheduling algorithms maintained consistent latency performance between the 40^{th} and 90^{th} percentiles with a significant gap between Klink and the other algorithms. For the tail latency (90^{th} – 99^{th}), Default's performance degraded from 9s at the 90^{th} percentile to 26 at the 99^{th} percentile indicating a heavy tail latency. Specifically, because Default does not prioritize queries that have due window deadlines, it suffers from high latencies especially under high memory utilization. The two Klink algorithms presented achieved significantly better latency performance across all percentiles. For instance, at the tail latency of 99^{th} percentile, Klink significantly reduced latency by 55% over Default. Interestingly, Klink equipped with the



(a) Slowdown

(b) Throughput

Figure 3.5: Slowdown and Throughput for YSB Workload

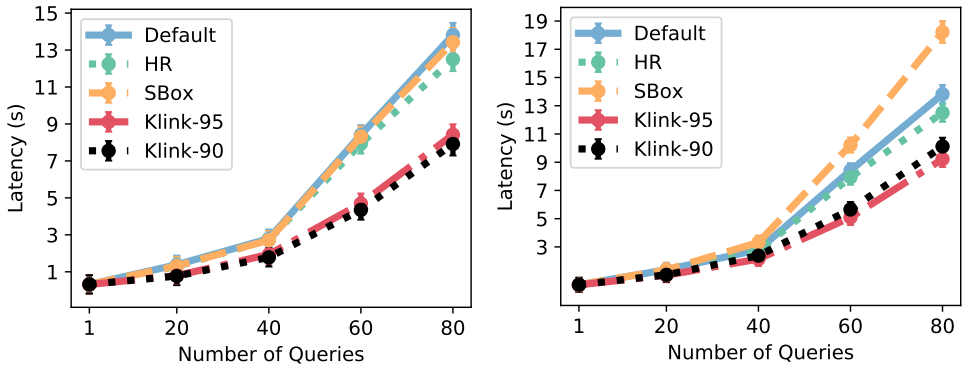
memory management technique (Sec 3.1.4) reduced tail latency over its counterpart by 20%. This demonstrates that while it is challenging to deliver consistent performance when SPEs are under memory stress, Klink's memory management technique allows it to deliver robust performance even under this challenging environment.

We measure the slowdown (Sec 3.4.1) incurred by each algorithm in Fig. 3.5a under the same workload settings as for last YSB experiment. The results mirror the prior trend in output latency and show that Klink delivers significantly better performance than the other algorithms.

Fig. 3.5b shows throughput while varying the number of deployed YSB queries. Default, HR, and SBox all achieve the same throughput of 2.5M events processed per second. Klink (w/o MM) delivers 2.65M throughput increase over the other algorithms. Interestingly, the non-Klink algorithms fail to scale their throughput performance past 40 deployed queries. The performance of these algorithms and their scalability is capped by a lack of timely processing of windowed queries together with inefficient memory utilization that queued events induce. In all cases where the output latency escalates, this is because the offered input load outstrips the SPE capacity of processing events, causing the latency to climb more quickly (e.g., past 40 deployed queries in Fig. 3.4a). Klink with its memory management algorithm (Section 3.1.4) demonstrates better scalability by achieving a throughput of 3.25M events processed per second delivering a 25% throughput improvement over its competitors. These results confirm that Klink's sound memory management technique

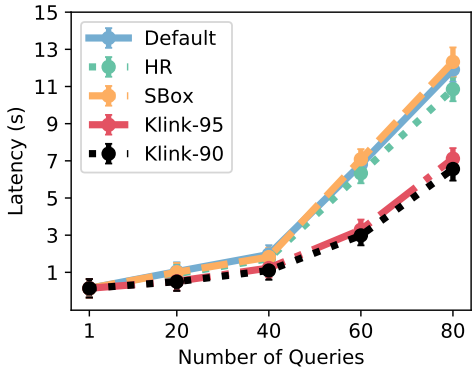
allows it to attain scalable system performance.

LRB and NYT Benchmarks

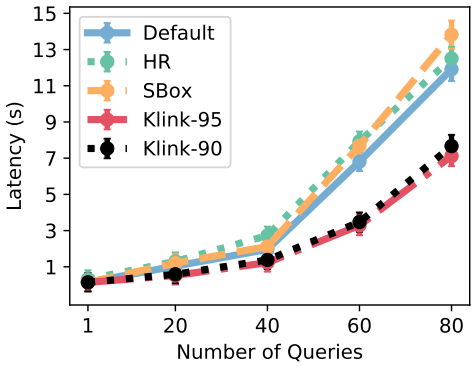


(a) LRB - Uniform

(b) LRB - Zipf



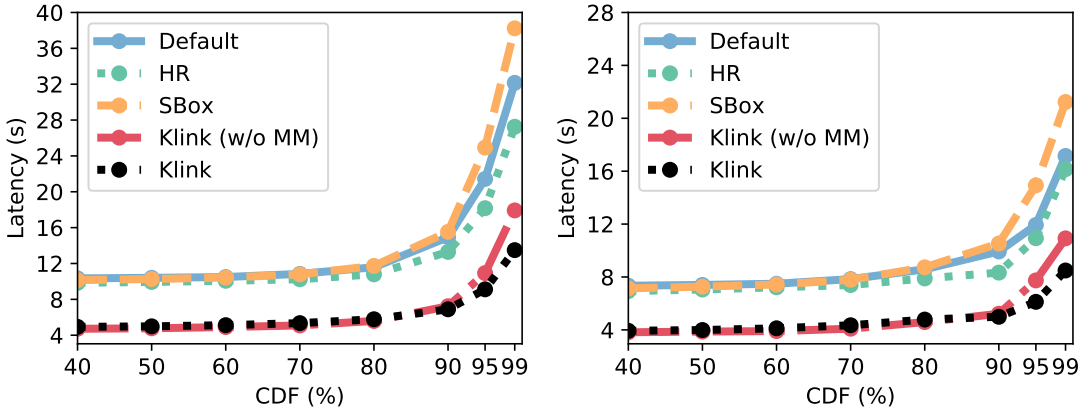
(c) NYT - Uniform



(d) NYT - Zipf

Figure 3.6: Mean latency vs. Number of queries running LRB & NYT benchmarks for different delay distributions

Our third experiment runs LRB and NYT with network delays under the Uniform and Zipf distributions. Fig. 3.6 shows these results. For the Uniform network delay scenario, Default Flink, HR, and SBox perform similarly whereby the latency almost reaches 15s for all algorithms. Klink delivers large latency reductions of 45% over Default and 40% over both LRB and NYT. As in YSB, Klink with 90% confidence value offers a 7% performance advantage over Klink with 95% confidence value. Default and HR maintain the same performance for Zipf distribution as in Uniform distribution. However, this is not the case for



(a) LRB CDF

(b) NYT CDF

Figure 3.7: Latency CDF for LRB and NYT workloads at 10,000 events/s per 60 deployed queries

SBox, which performs much worse than in the Uniform case. From our experiments, we noticed that SBox is sensitive to the periodicity of watermarks and to network delay variation. Furthermore, SBox also suffers from a heavy penalty due to lack of a back-pressure protection mechanism. Klink maintains similar performance advantage over Default for both LRB and NYT. However, in contrast to the Uniform scenario, Klink with $f = 95\%$ confidence value achieves a performance advantage of 5% over Klink with $f = 90\%$. This implies that running Klink with $f = 90\%$ is prone to lower accuracy rates than $f = 95\%$ to the extent that the extra overhead imposed by $f = 95\%$ is masked. Hence, the Zipf variability of network delay elicits higher accuracy. As in YSB (Fig. 3.4a), the latency performance in Fig. 3.6 over all environment variables exacerbated past 40 queries due to the SPE inability to scale its throughput.

Fig. 3.7 presents the latency CDF obtained for LRB and NYT. Default's latency in LRB increased by a significant 53% from 15 seconds at the 90^{th} percentile to 32 seconds at the 99^{th} percentile. As for NYT, Default's tail latency increased by 45% from 10 to 17 seconds. Similarly to YSB, these algorithms' tail latency performance scales poorly due to inefficient query scheduling under high memory utilization. The two Klink algorithms achieve significantly better latency performance across all percentiles. Specifically, the tail latency of Klink over Default for LRB and NYT experienced significant reductions of 60% and 50%, respectively. Furthermore, Klink (w/o MM) presented a heavier tail latency compared to its counterpart. This difference shows that tail latencies are affected when the SPE is running under high memory pressure and that Klink is effective in mitigating

its impact on latency. Finally, these results demonstrate Klink's robustness at achieving better mean and tail latencies over other scheduling algorithms regardless of the workload.

Distributed Experiments

We evaluate the distributed performance of Klink by deploying it on up to 8 nodes (machines) and running YSB. We ran HR in standalone mode since HR's design is not decentralized by default. SBox is unable to run without complete knowledge of the query pipeline so we run it in standalone mode for single-node experiments since it cannot operate in a distributed setting.

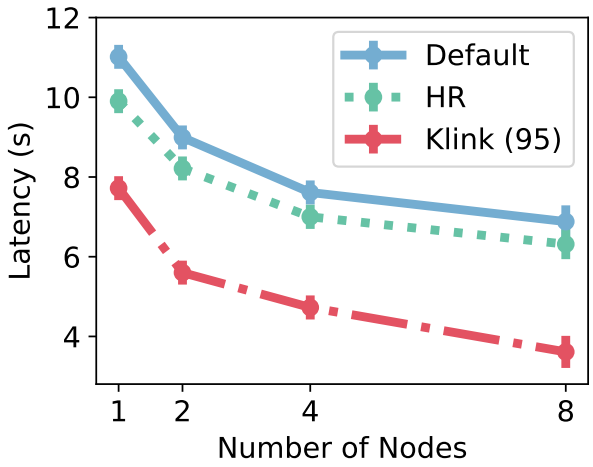
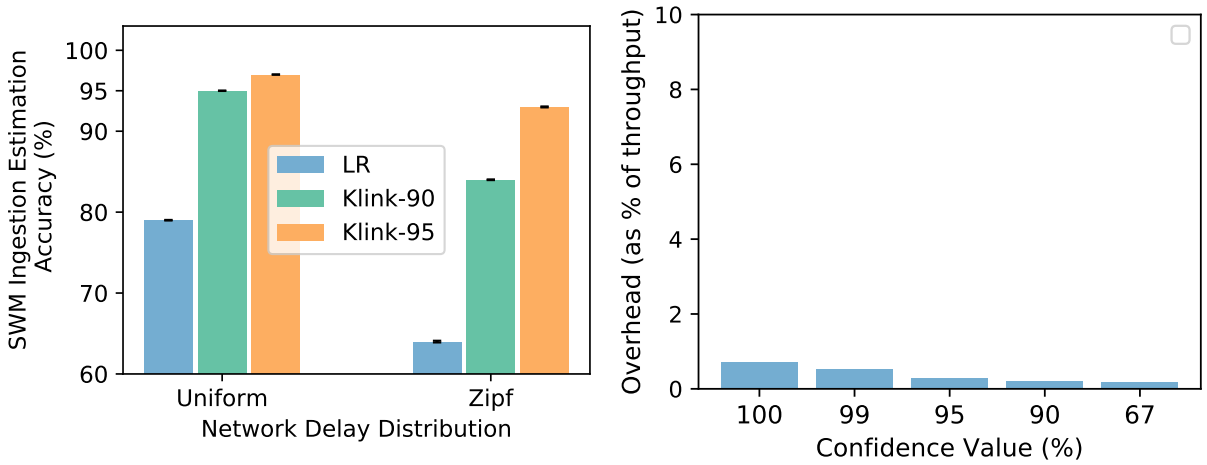


Figure 3.8: Distributed experiments running 80 YSB queries each emitting $10,000$ events/s

Since streaming systems are designed at their core to take advantage of distributed data processing, Klink embeds this design by continuously propagating relevant information across the SPE nodes.

Fig. 3.8 shows the performance of the algorithms running 80 YSB queries (each emitting $10,000$ events/s) while varying the number of nodes (machines) from one to eight. In these experiments, we utilize Flink's built-in mechanism that considers the type of operators and memory locality to minimize data mobility and parallelism levels to divide query pipelines across the compute nodes. For latency, we see a continuous decrease for all algorithms. Klink's distributed design allows it to lower its latency in comparison to the other scheduling algorithms with Klink maintaining a 40% performance improvement.



(a) (b)

Figure 3.9: (a) Klink's accuracy at estimating SWM ingestion time and (b) Klink's overhead while running at different confidence values of f.

Sensitivity and Overhead

Our last experiment measures the sensitivity of the watermark ingestion estimator based on the two widely occurring delay distributions of Uniform and Zipf. The purpose of this test is to measure the robustness of Klink's SWM ingestion estimation approach against network variability. The accuracy rate is measured by the fraction of times an SWM is ingested within Klink's estimated time range (Sec 3.1.1). The experiment is conducted with multiple values of f (Sec 3.1.2). We also provide a scheduler overhead analysis in these tests.

Fig. 3.9a presents Klink's accuracy at estimating SWM ingestion time. The figure shows accuracy performance for two confidence values of f, 95, and 90, under the different network delay distributionls. We also implemented gradient descent, a simple linear regression technique (LR) to show the performance advantage Klink possesses. For both network delays, Klink-95 provides marginally higher estimation accuracies than Klink-90, which is significantly more accurate than LR. While Klink-95 and Klink-90 provide 98% and 95% accuracy rates respectively, LR guarantees only 80% accuracy. The performance of LR exacerbates under the Zipf distribution with accuracy reaching only 62%. In comparison, Klink-95 and Klink-90 maintain higher SWM ingestion accuracy rate reaching 95% and 85%, respectively. The is expected as the Zipf distribution injects higher unpredictability

into the network delay.

Fig. 3.9b shows Klink's runtime overhead incurred from (Algorithm. 1), statistics collection, SWM estimation (Algorithm. 3, memory management, and orchestration with other operators. The overhead is measured as a percentage impact on throughput. That is, had the SPE runtime been allocated to processing events instead of running the scheduling algorithm, the figure presents the throughput loss that Klink would incur. While running Klink achieves higher throughput (Fig. 3.5b), its overhead should also be minimal to better utilize resources for processing events. Fig. 3.9b confirms Klink's efficiency as the algorithm incurs negligible overhead. The figure shows a marginal drop in overhead as the confidence values decrease, but the overhead difference between the highest confidence value and the lowest is meager. Klink's scheduler overhead impacts throughput by a negligible 0.5%. Since Klink's performance is hardly affected when varying the confidence values, Klink should be used with high confidence values.

Considering both Fig. 3.9a and Fig. 3.9b, an interesting percentile is the 90% confidence value as it delivers a similar accuracy rate to that of 95% and 99% confidence values while having much lower overhead. Klink's overhead is independent of the delay distribution, demonstrating robustness.

Klink's memory management reduces memory utilization after reaching a particular threshold. Our results reported that the scheduler algorithm is prone to suffer from overhead if the threshold is high, and is likely to introduce significant latency if the threshold is low. We empirically found that setting the threshold to 80% gave the best balance between both objectives. Similarly, to avoid extra latencies and to overcome deadlines, we configured Klink's memory management technique to run for the least slack time computed across all queries.

Chapter 4

Query Sample Processing

In this chapter, we present Aion, an algorithm that utilizes sampling to provide approximate answers with low latency by minimizing the effect of stragglers. Aion quickly processes the window to minimize output latency while still achieving high accuracy guarantees (Section. 4.1). We implement Aion in Apache Flink and show using benchmark workloads that Aion reduces stream output latency by up to 85% while providing 95% accuracy guarantees (Section. 4.2).

4.1 Aion: Straggler-Free Sampling

We now present the design of Aion including its algorithmic details. Fundamentally, in the context of windowed streams, Aion's main objective is to collect a sample of minimal size such that processing this sample produces an output that is within a specified error threshold r_{thr} of the exact output. We formally express the error by $P(r \leq r_{thr}) \geq 1 - \delta$, where r refers to the relative error discrepancy obtained by processing the original and the sampled inputs, and 1 - δ represents the probability of obtaining an error less than r_{thr}. The sample needs to be carefully chosen such that its size is minimal so as to reduce its processing cost. Importantly, the sample size and distribution of its values need to be sufficiently representative of the original input to satisfy the accuracy requirements.

Traditional sample processing techniques complete sampling their input only after the stream consumes a watermark. However, since watermarks signal input completion thereby accounting for stragglers, a significant output latency can be imposed. For instance, [76] presented an algorithm that mandates all events to *slack* for k-seconds before being processed, where k is continuously adjusted to the maximum observed network delay

value. Stragglers, however, contribute minimally in improving the accuracy of the sample (Fig. 1.5).

To circumvent the effects of stragglers on output latency, Aion leverages control over stream progress by automating the generation of watermarks. Watermarks divide a stream into *sub-streams*, each of which is defined over a periodic time-range. After watermark ingestion by the window operator, all events of prior timestamps consumed as part of sub-streams can then be safely processed. Therefore, Aion generates watermarks frequently to ensure incremental processing by window operators [62, 7]. Aion minimizes the impact of stragglers on output latency by generating a watermark as soon as the sampling requirements over each sub-stream are satisfied, even if all stragglers have not yet been ingested.

Symbols	Definitions
r_{thr}	User defined error margin
r	True error margin
δ	Probability of obtaining r below r_{thr}
f	Defined length for every sub-stream
n^w	Target Sample size over the windowed stream w
n_i^w	Number of events sampled over sub-stream i in stream w
d_i^w	Collected network delays over the i^{th} sub-stream in the windowed stream w
g_i^w	Collected inter-event generation delays over the i^{th} sub-stream in stream w
v_i^w	Collected event values over the i^{th} sub-stream in stream w
D_i^w	Random variable defined over the distribution of d_i^w
G_i^w	Random variable defined over the distribution of g_i^w
V_i^w	Random variable defined over the distribution of v_i^w
m	History size considered by the random distributions D_i^w, G_i^w, and V_i^w
N^w, N_i^w	Number of events received in stream w, and sub-stream i in stream w, respectively
n_i^w	Total number of events sampled over sub-stream i in stream w
s_i^w	Sampled events over sub-stream i in stream w
$N_{i,t}^w$	Number of events observed in sub-stream i in stream w at time t
θ_i^w	Sample rate defined by Aion over sub-stream i in stream w
ddl_i^w	deadline for the ith sub-stream in stream w

Table 4.1: Symbols used in Chapter. 4

Aion's design inherently supports incremental processing by sampling from each sub-stream at a customized rate. This strategy ensures more accurate estimations (for network and inter-event generation delays) since the workload data distribution has a lower likelihood of changing within each sub-stream. As soon as the accuracy requirements are

achieved, potentially before input completion, Aion generates a watermark to process the sub-stream at the window operator, effectively circumventing the effects of stragglers. The length of each sub-stream f (in milliseconds), also defined as the periodicity of watermarks, is essential to the algorithm's performance. A smaller value of f ensures higher uniformity for the input rate of each sub-stream at the expense of higher algorithm overhead. On the other hand, a larger value of f benefits from a lower overhead but imposes higher likelihood of input rate fluctuation. Aion is designed to leverage the granularity of f to proactively fine tune its input rate anticipation over the upcoming sub-streams. In our experimentation section, we choose values of f that empirically struck the best balance.

Aion is composed of (i) an algorithm that monitors properties of the workload including the network delay, the inter-event generation delay, and the distribution of the event values (Section 4.1.1), (ii) a proactive algorithm that estimates the sample size such that the error margin is bounded by r_{thr} (Sections 4.1.2 and 4.1.3), and (iii) a sampling algorithm that effectively samples the input based on the computed sample size (Section 4.1.3).

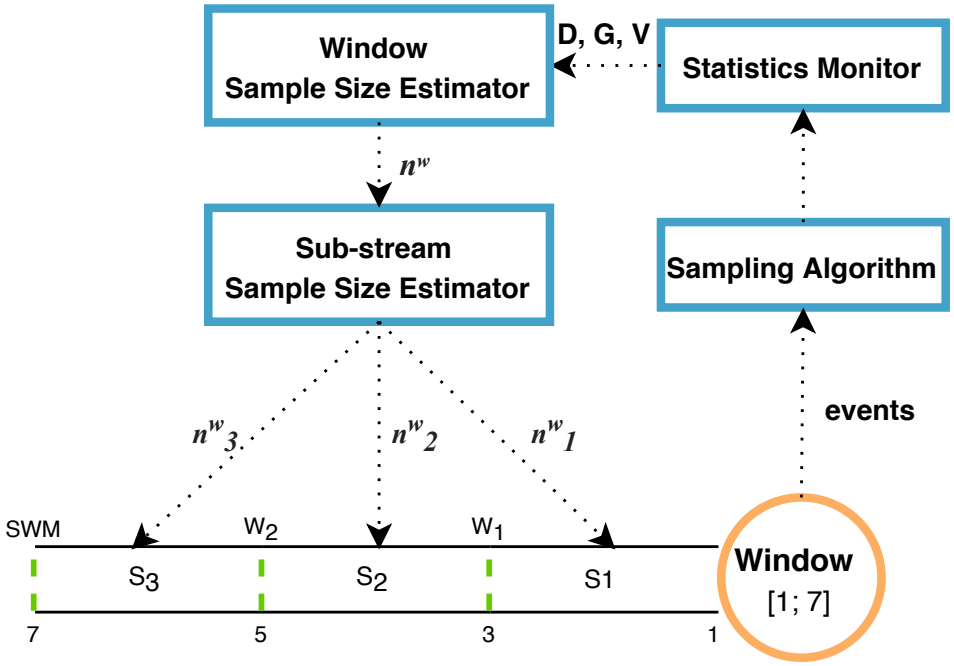


Figure 4.1: Example illustrating Aion components' interaction over a logically divided stream.

To illustrate Aion's functionality, consider Fig. 4.1 of a window operator encompassing all events generated between 1 and 7, with watermark frequency of $f = 2$. Initially, the Statistics Monitor collects and stores information such as the network delay (D), inter-event generation delay (G), and the event values (V). Then, the Window Sample Size Estimator is invoked either to build an estimation on the targeted windowed sample size or to update an existing one (Sec. 4.1.2). The sample size is estimated based on the specified error margin r_{thr} and the type of the windowed operator. For a window stream w, this part of the algorithm outputs n^w as the desired sample size estimate for the entire windowed stream. Finally, n^w is forwarded to the Sub-Stream Sample Size Estimator that computes the desired sample size for each sub-stream. Aion computes the sample size as a function of n^w and the expected ingestion delay of stragglers. Then, at each sub-stream, Aion runs its sampling algorithm to choose a sample with the target sampling size that is fair and representative.

Each of the aforementioned components in Fig. 4.1 are described in detail in the following sections.

4.1.1 Monitoring the Workload

Aion collects necessary information from the stream to quantify the target sampling size that achieves the output accuracy guarantees $P(r \leq r_{thr}) \geq 1-\delta$ (Fig. 4.1). In this section, we discuss in detail the collected information, their statistical representation, and their role in Aion.

Aion collects its information on a sub-stream basis to ascertain high output accuracy and to leverage incremental window processing. For the i^{th} sub-stream in the windowed stream w, Aion collects from the stream three main pieces of information: the network delay d_i^w, the inter-event generation delay g_i^w, and the event values distribution v_i^w. We denote by $d_i^w = \{d_{i,0}^w, d_{i,1}^w, ...\}$ the set of observed network delays by the SPE over the i^{th} sub-stream. For an event e, the network delay can be computed by $e.rts - e.gts$, where $e.rts$ refers to e's ingestion time by the SPE, and $e.gts$ to its generation time at the source. Similarly, for every two consecutive events e_k and e_{k+1}, $g_i^w = \{g_{i,0}^w, g_{i,1}^w, ...\}$ encompasses the set of all inter-event generation delays computed by $e_{k+1}.gts - e_k.gts$. Finally, $v_i^w = \{v_{i,0}^w, v_{i,1}^w, ...\}$ contains the set of event values observed in the corresponding sub-stream i in stream w. In Aion, this information is collected at the ingestion of every new event (Algorithm 4). More specifically, after identifying the windowed stream and the sub-stream to which event

Algorithm 4 Aion: Processing Events

```
1: procedure processEvent(e)
2:     /* Identify the stream and sub-stream event i belongs to */
3:     w ← getWindowIndex(e)
4:     i ← getSubStreamIndex(e)
5:     /* Examples of the collected statistics */
6:     d_i^w ← d_i^w ∪ (e.rts − e.gts)
7:     g_i^w ← g_i^w ∪ e.gts
8:     v_i^w ← v_i^w ∪ e.val
9:     w.observedEvents ← w.observedEvents + 1
10:    i.observedEvents ← i.observedEvents + 1
11:    /* If watermark has already been emitted */
12:    if i.isProcessed then
13:        dropEvent(e)
14:        return
15:    end if
16:    n_i^w ← GetSampleSize(w, i)
17:    s_i^w ← getSubsample(w, i)
18:    /* Choose to sample e or drop it */
19:    isSampled ← runSamplingAlgorithm(s_i^w, θ_i^w)
20:    if !isSampled then
21:        dropEvent(e)
22:    else
23:        s_i^w ← s_i^w ∪ e
24:    end if
25:    /* Check if its safe to generate a watermark */
26:    if SafeToProcess(s_i^w) then
27:        genWatermark(w, i)
28:        processSample(s_i^w)
29:        i.isProcessed ← True
30:    end if
31: end procedure
32: procedure SafeToProcess(w, i, n_i^w, s_i^w)
33:    minNeededSize ← getTargetSize(w, i)
34:    return currTime ≥ i.ddl && n_i^w ≥ minNeededSize
35: end procedure
```

e belongs based on its generation timestamp (lines 3–4), the necessary information is then extracted (lines 6–10).

Aion utilizes the collected information over the processed sub-streams to proactively estimate the patterns of the upcoming sub-streams. For a processed sub-stream i whose watermark has been emitted, we capture the statistical significance of the collected information distribution by the mean and the standard deviation. More specifically, we denote mean network delay by $\mu(d_i^w) = \frac{1}{|d_i^w|}\sum_{j=0}^{|d_i^w|} d_{i,j}^w$ and $\sigma(d_i^w)$ as the standard deviation. Similarly, We define $\mu(g_i^w)$, $\sigma(g_i^w)$, and $\mu(v_i^w)$, $\sigma(v_i^w)$, for inter-event generation delays and event values distribution to follow the above definitions. Aion computes the mean delay and the standard deviation on-the-fly, imposing no storage or computational overhead. Furthermore, Aion does not assume any underlying distribution over the collected information as information patterns can vary over multiple distributions [91, 16]. As for an upcoming sub-stream i, Aion estimates the statistical representation of the needed information i.e., d^w, v^w, g^w based on the historically processed sub-streams. We denote by the random variables D_i^w, G_i^w, and V_i^w the network delay, the inter-event generation delay, and the event values, respectively. Then, for the upcoming sub-stream i, we estimate the mean for D_i^w by:

$$E[D_i^w] = \frac{1}{m}\sum_{j=i-1-m}^{i-1} E[D_j^w] = \frac{1}{m}\sum_{j=i-1-m}^{i-1} \mu(d_j^w) = \mu(d_i^w) \tag{4.1}$$

Note that our estimations are limited to the last m sub-streams to reduce the storage overhead. .

Since D_i^w is the result of a summation of means, D_i^w follows a normal distribution through the central limit theorem. Having known distributions, specifically normal distribution, provides reliability on calculations using the aforementioned random variables. These reliability properties are also shared by G_i^w and V_i^w since we define them similarly to Eq. 4.1.

Aion leverages these random variables for key calculations. That is, in Algorithm 5, Aion utilizes the network delay and the inter-event generation delay to collect the sub-stream size (line 4), and it utilizes the event values to estimate the sample size in Algorithm 4 (line 16) and Algorithm 5 (line 10). We discuss the estimations further in the next section.

4.1.2 Window and Sample Size Estimators

Aion intelligently selects and processes a sample which delivers a result within r_{thr} of the true result. Initially, Aion quantifies the projected number of events in the windowed stream. Then, based on the estimated number of events, r_{thr}, and the type of the window operator, Aion estimates a target sample size. This section discusses Aion's estimation techniques for the window and sample sizes.

Initially, Aion quantifies the projected number of events in the windowed stream based on the collected information (Section 4.1.1). For a windowed stream w, we denote the size of the window by N^w representing the total number of events including stragglers of the window. Intuitively, the size of the windowed stream is a function of the size of each of its sub-stream constituents, that is, $N^w = \sum_{i=1}^{k} N_i^w$, where k represents the number of sub-streams in the windowed stream w, and N_i^w refers to the number of events in sub-stream i (Algorithm 5, lines 3–6). The size for each sub-stream is then estimated based on the inter-event generation delays observed over previously processed sub-streams. Note that a workload with a high frequency of event generation would entail low values of g_i^w, while sparser event generation would mean higher values of g_i^w. The size estimation of sub-stream i can be expressed by:

$$E[N_i^w] = \frac{f}{E[g_i^w]} = \frac{f}{\mu(g_i^w)} \tag{4.2}$$

Aion continuously adjusts its estimations of the size of the windowed stream and its sub-stream constituents as earlier sub-streams are processed (i.e., as corresponding watermarks are emitted). In doing so, it guarantees more accurate estimations as time progresses towards the window's deadline. It is important to note that regardless of the distribution of g_i^w which can vary depending on the application type [84], Aion makes no assumptions on the input's arrival rate.

Aion computes the sample size based on the statistics monitor's estimation of N^w. However, since estimation elicits uncertainty thereby affecting the accuracy of the sample size, Aion seeks to overestimate N^w based on the level of uncertainty quantified by the standard deviation. In doing so, the sample size is marginally augmented to achieve higher accuracy guarantees while keeping it small enough to maintain low processing cost. Overestimation of N^w is extremely helpful in the case of ingesting more events than the anticipated size. As for underestimation, the processing cost is marginally increased, thereby hardly affecting it. Hence, marginally overestimating N^w helps Aion to consistently achieve robust performance in the face of workload fluctuation. In our experiments, we overestimate the window length based on two degrees of the standard deviation.

Algorithm 5 Aion: Estimations & Sampling

1: **procedure** GETSAMPLESIZE(w, i)
2: /* *Estimate the window size for each sub-stream (Sec. 4.1.2)* */
3: **for** j in range $(0, \frac{ddl^w - ddl^{w-1}}{f})$ **do**
4: $N_j^w \leftarrow estmSubstreamSize(w, j)$ // (Eq. 4.2)
5: $N^w \leftarrow N^w + N_j^w$
6: **end for**
7: /* *Estimate the sample size for each sub-stream (Sec. 4.1.3)* */
8: $n^w \leftarrow estmWindowSampleSize(N^w)$ // (Eq. 4.4
9: **for** j in range $(0, \frac{ddl^w - ddl^{w-1}}{f})$ **do**
10: $n_j^w \leftarrow estmSubstreamSampleSize(n^w, j)$ // (Eq. 4.8)
11: **end for**
12: **return** n_i^w
13: **end procedure**
14: **procedure** RUNSAMPLINGALGORITHM(n_i^w, e)
15: $\theta_i^w \leftarrow getSamplingRate(n_i^w)$ // (Eq. 4.8)
16: $r \leftarrow genNumber(0, 1)$
17: **return** $r \leq \theta_i^w$
18: **end procedure**

After estimating the windowed stream size N^w, we quantify the desired sample size (Algorithm 5, line 8) based on r_{thr}, and the type of the window operator. From the related literature, there has been work on sample processing that did not account for the functionality of the window operator and therefore limited their sample selection to achieving similar statistical properties to that of the original input [53]. However, as illustrated and recommended in [27], specifying an error function based on the window functionality yields consistent higher accuracy. Aion adopts the latter approach to achieve the highest accuracy possible. We provide two examples of error function derivations for the two commonly used window functionalities: events-mean computation, and summation. For each functionality, we derive a formula relating the error bound r_{thr}, the windowed stream size N^w, and its corresponding sample size n^w.

We consider the first case of events-mean computation, where the window is computing the average of observed events. The relative error function can be expressed as $\frac{|\mu(s^w) - \mu(v^w)|}{\mu(v^w)}$, where $\mu(s^w)$ represents the mean of the sampled input, and $\mu(v^w)$ denotes the mean of the original input. To maintain the processing cost at a minimum, we are interested in finding the *minimum* sample size n^w that satisfies $P(|\frac{\mu(s^w) - \mu(v^w)}{\mu(s^w)}| \leq r_{thr}) \geq 1 - \delta$. Per [66], the

error r_{thr} is tightly related to sample size n^w, and the original input size N^w by:

$$r_{thr} = \frac{z_{\delta/2}\sqrt{(1 - \frac{n^w}{N^w}) \times \frac{\sigma(s^w)}{\sqrt{n^w}}}}{\mu(N^{w-1})} \tag{4.3}$$

where $z_{\delta/2}$ refers to the confidence interval matching the z-value[1] with the specified probability δ. Then, solving for n^w, we have:

$$n^w = \frac{z^2_{\delta/2}\sigma^2(s^w)}{r^2_{thr}\mu(v^{w-1})^2 + \frac{z^2_{\delta/2}(\sigma(s^w)^2}{N^w}} \tag{4.4}$$

By solving for the minimum sample size in Eq. 4.3, n^w can be derived as in Eq. 4.4. Thus, Aion collects at least n^w events in the windowed stream w to achieve the accuracy guarantees. The $\mu(v^{w-1})$ is a historical mean.

n^w/N^w is the proportion of events that were sampled from the original input. Therefore, taking the estimate for the sample total and scaling it up by N^w/n^w accounts for events that are not in the sample. Using the derived equations for events-mean computation, a formula for estimating the input summation is given by:

$$\begin{aligned} \frac{N^w}{n^w}\sum_{i=1}^{n^w} v_i &= N^w \frac{\sum_{i=1}^{n^w} v_i}{n_w} \\ &= N^w \mu(s^w) \end{aligned} \tag{4.5}$$

Eq. 4.5 expresses that an estimate for the original input can be taken by scaling $\mu(s^w)$ up by the known N^w. This method requires finding an estimate for $\mu(s^w)$ and therefore, the same sample size estimate for determining $\mu(s^w)$ as seen in Eq. 4.4 can be used to get an accurate estimate for the total input summation.

As for the summation operator, that is, computing the sum over all elements considered, the relative error function is defined by:

$$r = \frac{|\sum_{i=1}^{N^w} v_i^w - (N^w \times \mu(s^w))|}{|\sum_{i=1}^{N^w} v_i^w|} \tag{4.6}$$

There exist multiple approaches to define error bounds over a window functionality. For instance, a different error function for the summation functionality is used by AQ-K-Slack [52]. This approach imposes an unnecessarily large sample size. The literature also

[1]Further information on z-values can be found in [66].

includes work on other types of scalar window functions like MAX, MIN, and quantiles [34, 54]. Other event-based vector operations can be adapted for Aion by utilizing the error functions derived as in [54, 4] for grouping. They can be incorporated into Aion following similar derivations from Sec. 4.1.2.

After estimating the window and sample sizes, Aion utilizes these estimations in its "Sub-stream Sample Size Estimator" component (Fig. 4.1). Aion leverages these estimations over each sub-stream and then executes its sampling algorithm accordingly, as described in the next section.

4.1.3 Sampling over Sub-streams

Aion is optimized to select a sample free of stragglers that is representative of the original input. In this section, we describe Aion's sampling algorithm that minimizes the impact of stragglers.

On the arrival of each event, Aion runs its sampling algorithm (line 16 in Algorithm 4, procedure call defined in Algorithm 5 line 14) to select its sample. Based on the computed sample size n^w (Section 4.1.1, and line 10 in Algorithm 5), an event can be added to the sample based on the rate $\frac{n^w}{N^w}$. However, this approach suffers from the stragglers' problem as the sample is unlikely to be complete by the window's deadline. Aion, therefore, optimizes the sampling rate over each sub-stream i so that at the window's deadline, the sample would be complete or near completion. To express this formally, we denote $N^w_{i,t}$ to be the total number of events ingested by time t. By using the inter-event generation and network delays, we can estimate $N^w_{i,t}$ by:

$$N^w_{i,t=ddl^w_i} = \frac{(ddl^w_i - E[D^w_i]) - ddl^w_{i-1}}{E[G^w_i]} \tag{4.7}$$

Then, the updated sampling rate at sub-stream i is computed by:

$$\theta^w_i = \frac{n^w}{N^w_{i,t=ddl^w_i}} \tag{4.8}$$

Aion then samples the incoming events according to θ^w_i thereby mitigating the presence of stragglers. Note that if θ^w_i is greater than 1, Aion slacks for minimal time to include the least number of stragglers. The algorithm is shown in Algorithm 5 (lines 14–17).

As such, Aion minimizes the numbers of stragglers in the sample by prioritizing sample completion before the window's deadline. Then, Aion generates a watermark as soon as the

window's deadline is due and the accuracy guarantees are achieved (Algorithm 4, lines 32–34). These conditions ensure that a watermark is generated when the accuracy guarantees are met while minimizing the output latency

4.2 Performance Evaluation

In this section, we present the results of a series of experiments conducted on multiple benchmark workloads to demonstrate the performance advantage that Aion possesses over representative algorithms from prior work.

4.2.1 Experimental Setup

We describe our experimental setup and methodology, including machine configurations, the sampling algorithms that we compare Aion against, benchmarks, and the delay distribution settings. Our experiments are run on a machine having an Intel Xeon processor consisting of 24 cores (using hyper-threading) and 32 GB of memory. The machine is running Java OpenJDK implementation v1.8.0_191 on top of Ubuntu 16.04 LTS. The implementation of Aion is on Apache Flink v1.8. We dedicate a different machine with the same configuration for generating the workload, which is transmitted to the SPE nodes via Kafka v2.2.1.

Benchmarks

We conduct our evaluation using three well-known streaming benchmarks: the Yahoo! Streaming Benchmark (**YSB**) [45], the New York City Taxi (**NYT**) [77], and the **kMeans** benchmark. We implement these benchmarks on Apache Flink and evaluate performance by running different sampling algorithms in each experiment. YSB emulates an advertisement tracking system where users launch ad campaigns, each of which is composed of multiple ads. The YSB query handles a stream of ad clicks and outputs the interest in each ad campaign. We use the code-base provided by [45] with the addition of generating periodic watermarks from the source. NYT covers a large dataset of taxi trips in New York, spanning six years. The dataset is rich with information such as the number of passengers, distances, and fares. The query measures the average distance of each trip ride in sliding windows. KMeans query is an algorithm that partitions the dataset into k clusters. The dataset originally at the source is filtered and processed in the pipeline before running the kMeans algorithm in a windowed operator.

Algorithms

Sample processing algorithms generally need to run in conjunction with a deployed stream progress control algorithm. Specifically, sample processing algorithms continuously update their sample until the accompanying stream progress control algorithm generates a watermark. Popularly used stream progress control algorithms include slacking techniques [76, 84]. Slack algorithms compute the minimum slack delay needed to guarantee input completion, then generating a watermark as soon as the slack delay expires. We implemented onto our system the K-Slack [76] algorithm to automate the generation of watermarks. K-Slack generates a watermark every k seconds, where k is set to the maximum observed network delay. Since Aion uniquely combines sampling and stream progress control, it overrides K-Slack by running its own stream progress control algorithm proposed in this thesis.

To demonstrate Aion's efficacy, we compare Aion against *Default*, which is the baseline approach that constitutes running Flink with no sampling. In Default, windows process their input as soon as the deployed stream progress control algorithm emits a watermark. We also compare against the following sample processing algorithms:

- AQ-K-Slack [52]: Similarly to Aion, this algorithm samples events as they are ingested by the SPE. It computes the sampling ratio needed to achieve the necessary output accuracy guarantees. Using an error function specific to summation windowed functionality, AQ-K-Slack ties the relative error function with the target accuracy while accounting for stragglers in processing output.

- Aion-: denotes Aion without (minus) the stragglers' circumvention technique employed (Sec 4.1.3). The Aion- algorithm applies Aion's sampling technique but without the stream progress control algorithm. Specifically, the sample size is estimated as a fraction of N_i^w and not $N_{i,t=ddl_i^w}^w$. We implemented Aion- to study the impact of circumventing stragglers when sampling.

Delay Distributions

As in [84, 16], we vary two main types of delay for our experiments, namely the network delay, and the inter-event generation delay. Similarly to [84, 16], we refer to these as follows in our experiments;

- CC: Network and inter-event generation delays are constant at 150ms and 1ms, respectively.

- GG: Network delay and inter-event generation delay are distributed with $Gamma(k = 60, \theta = 4)$ and $Gamma(k = 2, \theta = 0.5)$ respectively.
- EC: Network delay is exponentially distributed with mean delay of 240ms while inter-event generation delay is constant at 1ms.
- EG: Network delay is exponentially distributed with mean delay of 240ms while inter-event generation delay is Gamma distributed with $Gamma(k = 2, \theta = 0.5)$.

We generate ten different streams for each combination. Each data point on the graph is an average over at least 10 independent runs (unless stated otherwise). Ideally, Aion's watermark periodicity f should be a divisor of the window deadline so that a watermark can be emitted at the window's deadline effectively minimizing latency. Based on empirical evidence collected through our experiments, we set f to 600ms for deployed windows of size 3s and 6s. This value strikes a balance between the estimation granularity and the overhead of the algorithm.

4.2.2 Results

YSB

Figure 4.2 shows the performance of the sampling algorithms running the YSB workload under different delay distributions. For the first distribution of UU, Aion delivers lower latency than the other algorithms as it reduces both the stragglers' impact on output latency and the processing cost of events. More specifically, while Default, AQ-K-Slack, and Aion- algorithms impose $150ms$ delay to account for stragglers, Aion circumvents the slack delay by emitting a watermark as soon as the output accuracy guarantees are achieved. For the processing cost, since Default processes all of the events, its processing overhead dominates the output latency. As for Aion- and Aion, both algorithms sampled at a rate reaching 30% of the original input, while the AQ-K-Slack sampling mechanism is more restrictive, pushing its sampling rate to exceed 60%. Aion minimizes both the slack delay and the processing cost yielding significant latency reduction over the other algorithms. For the GG delay distribution, the maximum observed network delay reached 350ms adding significant overhead for both Default and AQ-K-Slack. However, as in the case of UU, the processing cost dominated the output latency for both Default and AQ-K-Slack. Aion delivers lower output latency over Default and AQ-K-Slack by 80% and 70%, respectively.

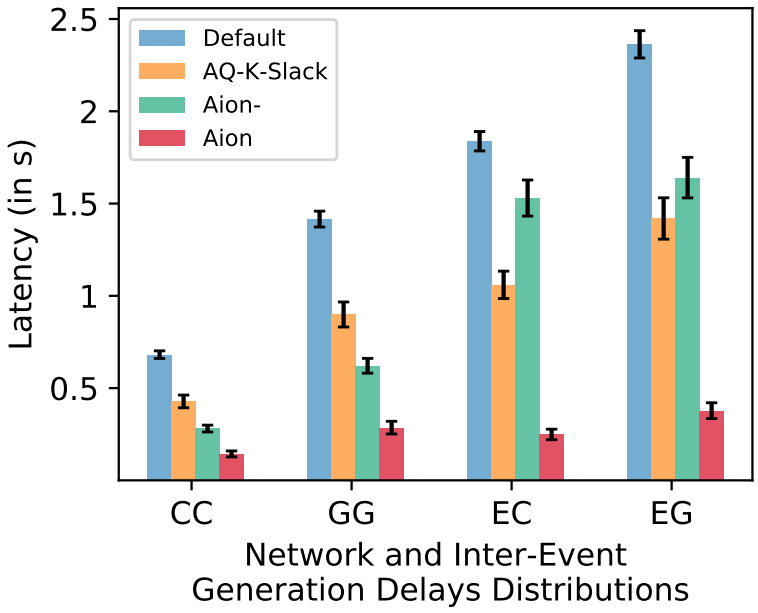
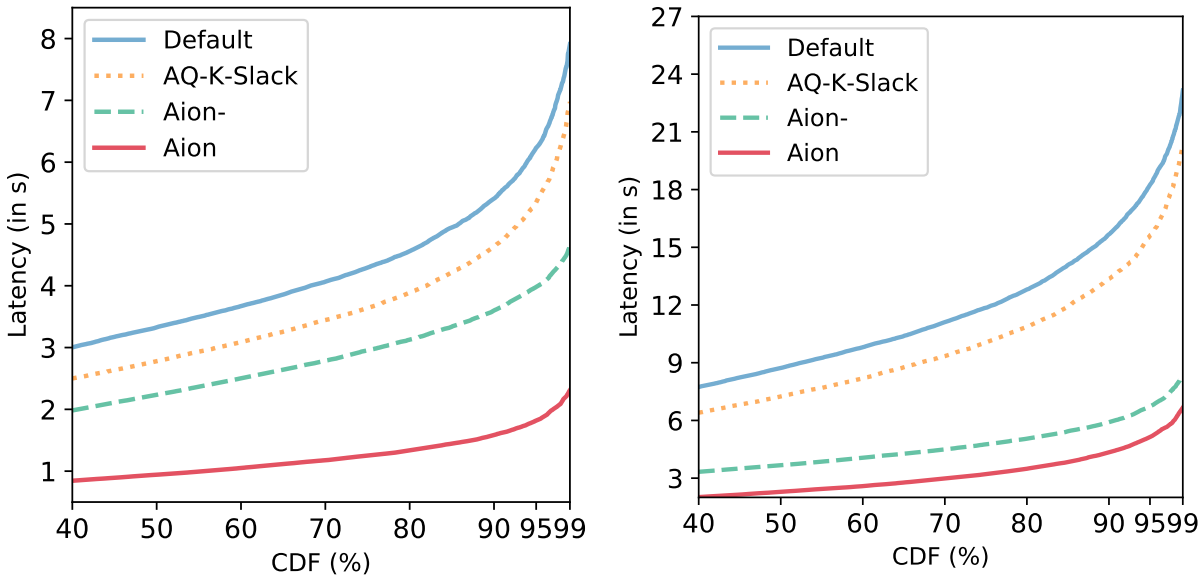


Figure 4.2: Mean latency vs. different environment distributions of network delay and inter-event generation delay running YSB benchmark

When network delay is exponentially distributed, the performance of Default and Aion- worsens as the maximum observed network delay exceeds 1500ms. That is, the two algorithms process windowed data long after generating a watermark that is past the window's deadline. Note that Aion- quickly processes its data after slacking for 1500ms while Default processes the entire input. As for AQ-K-Slack, it delivers better performance over these two algorithms since its slack function uses randomization to mitigate the delay. Since AQ-K-Slack's sampling function is costly, its processing cost dominates output latency. Aion outperforms Default and AQ-K-Slack with latency reductions of 85% and 78%, respectively. Aion significantly outperforms the two algorithms by mitigating the effect of stragglers and significantly reducing the processing cost.

We also compared Aion's latency to the other algorithms for different input loads. Fig. 4.3 presents the cumulative distribution function (CDF) of recorded latencies for the range of 40^{th} to 99^{th} percentile tail latency under two input load levels of number of events generated: $5,000$ and $25,000$ (5x) events/s. The experiment is run with distribution EG where the network delay is exponentially distributed and the inter-event generation delay is gamma-distributed. For the 90^{th} percentile, Aion achieves latency reductions over Default



(a) 5,000 events/s

(b) 10,000 events/s

Figure 4.3: Recorded CDF latency running YSB benchmark for delay distribution EG with 5,000 and 25,000 number of input events generated per second.

and AQ-K-Slack of 76% and 70% respectively for both 5,000 and 25,000 events generated per second. As for tail latencies, and specifically 95^{th} to 99^{th} percentiles, Aion maintains a low latency level delivering consistent latency performance as with smaller percentiles. However, this was not the case for other algorithms as Aion reduced latency over Default and AQ-K-Slack by 80% and 75% respectively for 5,000 events generated per second, and 84% and 80% respectively for 25,000 events generated per second. Interestingly, Aion- performed similarly to Aion for 25,000 events/s but not for 5,000 events/s. This is due to the output latency for 25,000 events/s being dominated by the processing cost and not the slack cost, while for 5,000 events/s it was dominated by the slack delay. Aion optimizes for both of these costs to attain the large aforementioned latency reductions.

Aion optimizes for reduced latency while still achieving the specified accuracy guarantees. Figs. 4.2 and 4.3 show that Aion outperforms the other algorithms significantly in terms of reducing output latency. Next, we show the accuracy guarantees delivered by the sampling algorithms Aion, Aion-, and AQ-K-Slack through experiments with parameters $r_{thr} = 5\%$ and $\delta = 0.95$.

Fig. 4.4 presents the distribution of error obtained while running YSB. The experiments

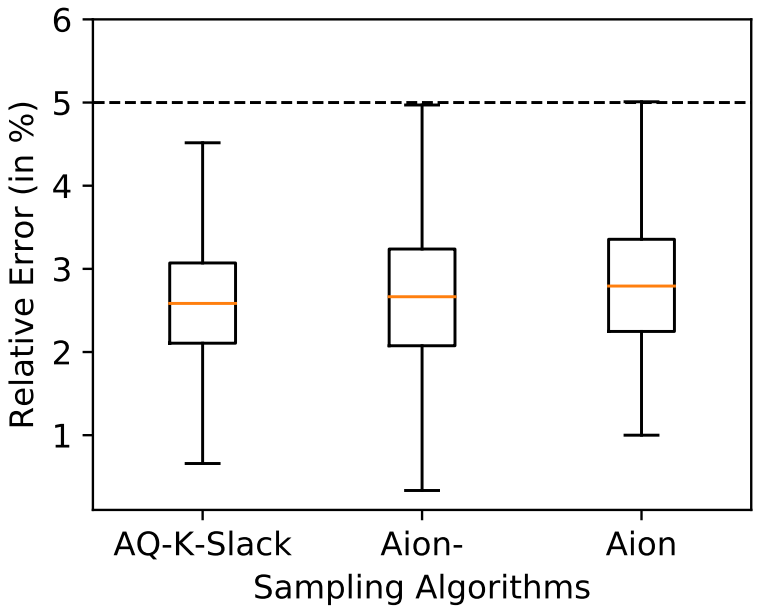


Figure 4.4: Error plot showing statistical significance obtained by AQ-K-Slack, Aion-, and Aion running YSB benchmark for delay distribution EG with 25,000 input events generated per second

were executed for EG with 25,000 events generated per second. For the three tested algorithms, although the relative error margin was set to 5%, the algorithms recorded significantly lower error margins. Specifically, the mean relative error (Sec. 4.1.2) reached 3% for all algorithms with a relatively small standard deviation. Interestingly, for all algorithms, the maximum experienced error was less than the threshold, thereby delivering excellent accuracy exceeding the required guarantees. As such, this figure proves that circumventing stragglers have little to no impact on finding a sample with high accuracy guarantees.

NYT

The NYT benchmark is a relatively expensive query compared to YSB as it includes a longer pipeline and more processing intensive operators. As such, the processing cost in NYT factors higher in output latency compared to the processing cost factor in YSB's output latency. Fig. 4.5 shows the performance of the sampling algorithms running the NYT workload under different delay distributions. For UU, Aion achieves significantly

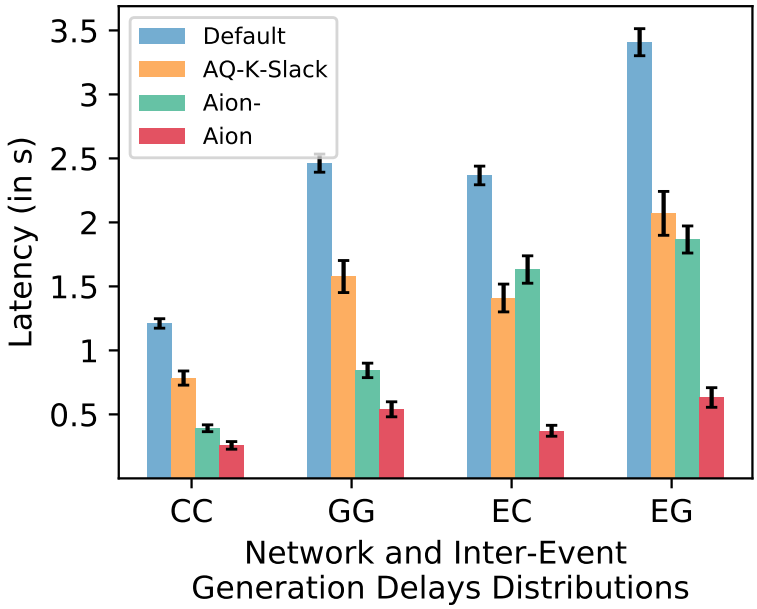
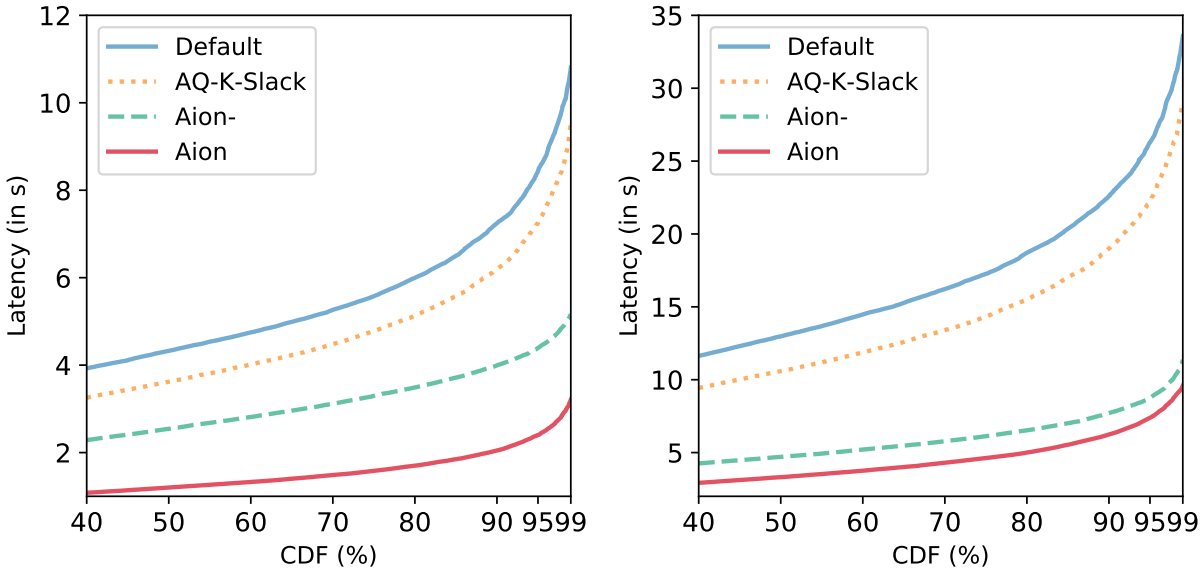


Figure 4.5: Mean latency vs. different environment distributions of network delay and inter-event generation delay running NYT benchmark

lower output latency over Default and AQ-K-Slack mainly due to Aion's superior sampling mechanism as it samples at a lower rate. Since the slack delay in UU is minimal and processing cost of NYT is expensive, Aion achieves a latency reduction of 75% over Default that processes all events, and 60& reduction over AQ-K-Slack as it suffers from higher processing cost and a high slack delay. For other delay distributions, and specifically EU and EG, where the experienced network delay is significantly higher, Aion's mechanism of circumventing stragglers significantly reduced the output latency. Specifically, Aion achieved an 80% output latency reduction over Default and 72% over AQ-K-Slack. Furthermore, the difference between Aion and Aion- in EU and EG prove that it is essential to minimize the straggler count in the sample to minimize the output latency, regardless of the query complexity.

Aion's latency CDF for NYT is shown in Fig. 4.6. As explained earlier, this experiment delivered different results than that for YSB in Fig. 4.3 due to the higher complexity of NYT benchmark. At the lighter workload of $5,000$ events per second, the output latency of Aion- was dominated by slack delay while Default and AQ-K-Slack suffered from both the slack delay and processing cost. Consequently, Aion's tail latency, 95^{th} to 99^{th} percentile,



(a) 5,000 events/s

(b) 10,000 events/s

Figure 4.6: Recorded CDF latency running NYT benchmark for delay distribution EG with 5,000 and 25,000 number of input events generated per second.

is significantly reduced over Default and AQ-K-Slack by 80% and 75%, respectively. As for the heavier workload of 25,000 events per second, the tail latency reduction attained by Aion over Default and AQ-K-Slack reaches 87% and 85%, respectively. Aion- performs similarly to Aion as output latency is dominated by the processing cost in both of these Aion-based algorithms. Aion incurs lower processing cost because its sampling mechanism requires a significantly smaller sample size. Therefore, Aion can attain greater scalability than other algorithms as it avoids slack delays while reducing processing costs.

We studied the distribution of errors obtained when running the NYT benchmark in Fig. 4.7 with parameters $r_{thr} = 5\%$ and $\delta = 0.95$. In this case, since the NYT window operator is a mean estimation and the workload entails a higher standard deviation of V, both Aion- and Aion experience error greater than 5%. However, the error crossed the threshold only three times from one-hundred data points collected, therefore achieving a high confidence level of 97%. Furthermore, while AQ-K-Slack delivered a significantly higher latency, by 80%, the error for both algorithms was very similar. Finally, Aion's stringent accuracy requirements are a function of δ as Aion guarantees $P(r \leq r_{thr}) \geq \delta$. But in the cases where the obtained error exceeded r_{thr}, the max error is relatively close

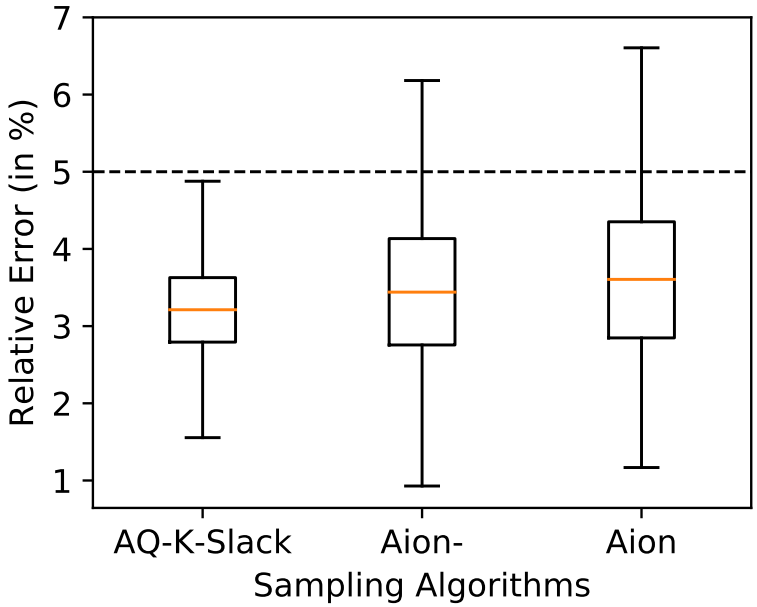


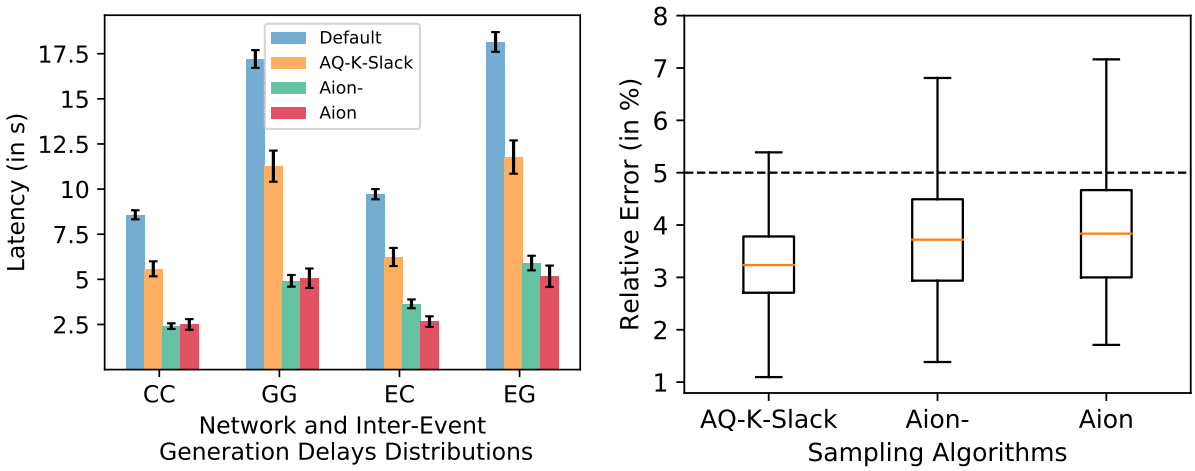
Figure 4.7: Error plot showing statistical significance obtained by AQ-K-Slack, Aion-, and Aion running NYT benchmark for delay distribution EG with 25,000 input events generated per second

to r_{thr} indicating that Aion delivers consistent accuracy performance.s

kMeans

The benchmark kMeans is the third and most expensive query for which we compare Aion against the other algorithms. Furthermore, the sampling rate in kMeans reached 45% for the Aion algorithms, whereas it reached 85% for AQ-K-Slack. In this case, the processing cost dominates the output latency for all tested configurations. This is shown in Fig. 4.8a as the two Aion algorithms delivered significantly lower output latency results over Default and AQ-K-Slack for all environment settings. Aion's sampling mechanism helped it to reduce the output latency significantly. Aion minimized both the processing cost and the network penalty over Default and AQ-K-Slack by up to 75% and 65%, respectively.

Aion's latency CDF for kMeans is shown in Fig. 4.9. The benchmark kMeans is even more expensive than NYT, making processing cost to be the dominating factor in output latency. For 5,000 events generated per second, Aion achieves significantly lower 95^{th} to



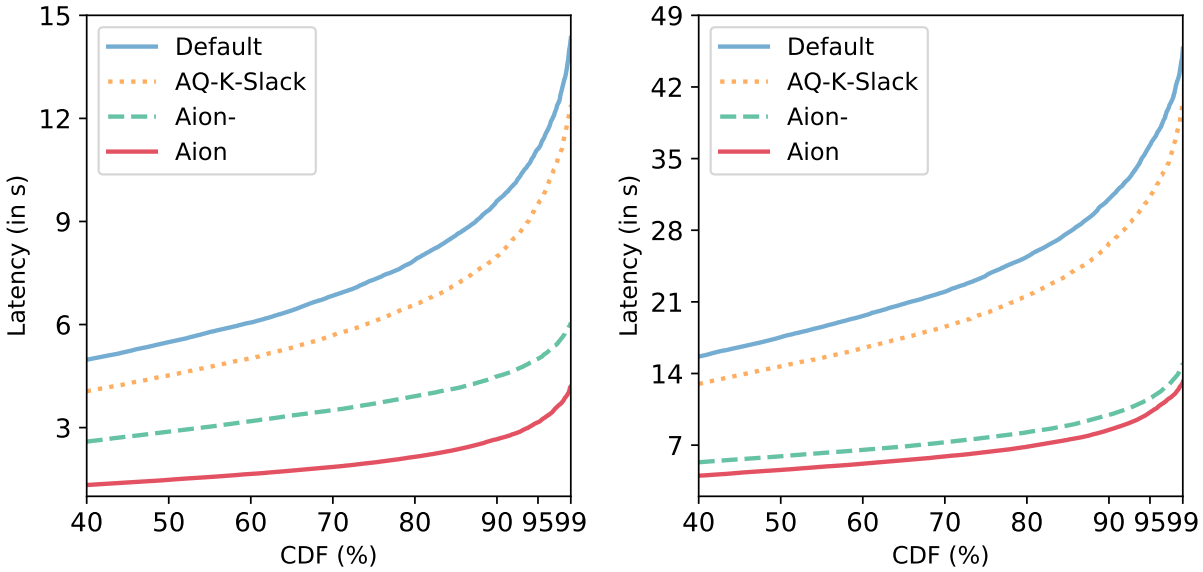
(a) Mean latency

(b) Error plot

Figure 4.8: (a) Mean latency vs. different distributions of network delay and inter-event generation delay running kMeans benchmark, and (b) error plot showing statistical significance obtained running kMeans benchmark for delay distribution EG with 25,000 input events generated per second

99^{th} percentile tail latencies, reducing them over Default and AQ-K-Slack by about 80% and 72%, respectively. AQ-K-Slack, which suffers from a significantly higher sampling rate, incurs high output latency thereby displaying similar tail latency growth to that of Default. As for the 25,000 events generated per second input load, we observe that Aion and Aion- achieve very similar performance, further indicating that the processing cost dominated the output latencies in this experiment. As for Default and AQ-K-Slack, both algorithms suffered significant performance degradation, adding 82% and 78% latency overhead, respectively, at the tail. Aion delivers slightly better performance over Aion- since Aion exhibits no slack delay while Aion- waited for an additional 1.5s to ensure input completion.

Similar to previous experiments, we ran experiments with the following parameters: $r_{thr} = 5\%$, $\delta = 0.05$. Fig. 4.8b presents the distribution of error obtained while running kMeans. In this case, since the window operator is computing the k-means algorithm that involves more complex processing, the resulting error was higher. While all algorithms achieved a maximum latency higher than that of the threshold, all of them achieved this within the constraint $P(r \leq r_{thr}) \geq 1 - \delta$. Furthermore, while AQ-K-Slack delivered significantly higher latency by 80%, the mean error for both algorithms was very similar.



(a) 5,000 events/s

(b) 10,000 events/s

Figure 4.9: Recorded CDF latency running kMeans benchmark for delay distribution EG with 5,000 and 25,000 number of input events generated per second.

Empirical verification and exhaustive experimentation consistently demonstrated that Aion error values rarely stretched over the specified error threshold.

Finally, we measured Aion's runtime overhead incurred by statistics collection, window and sample sizes estimation, and sampling incoming events. Aion's overhead represents 0.15% of the obtained output latency per window. The low overhead is due to Aion's minimal monitoring of workload, and efficient sampling technique. Due to lack of space, we omit the overhead graphs.

Chapter 5

Conclusion

We began by motivating the use of stream progress property in SPE runtime to improve the performance of streaming engines. Through leveraging the stream progress property, we presented and studied problems that are of significant importance for SPEs: (1) the problem of multi-query scheduling, and (2) the problem of sample query processing. Before presenting our algorithms, we first illustrated the advantage of utilizing stream progress in each of the two problems. Then, in Chapter 2, we introduced the necessary background material and covered the relevant work.

In Chapter 3, we addressed problem (1) by introducing our scheduling algorithm that aims to unblock stateful operators and to rapidly propagate the events to output operators. Specifically, Klink assesses the progress of each stream over multiple queries by analyzing watermarks and appropriately assigning a priority value designed to minimize the output latency. We presented and analyzed Klink's technique in analyzing and estimating incoming watermarks. Through the integration of Klink into Apache Flink and extensive experimentation, we demonstrated that Klink outperforms existing scheduling algorithms. Our experiments demonstrated that Klink delivers large output latency reductions of up to 65% over existing techniques.

In Chapter 4, we addressed problem (2) by circumventing stragglers and significantly reducing output latency. Specifically, we introduced Aion, a query sample processing algorithm that approximates answers with low latency by minimizing the effect of stragglers. Specifically, by leveraging control over stream progress and automating the generation of watermarks, Aion mitigates the impact of stragglers on output latency. Similarly, Aion was integrated into Apache Flink along with other sample processing algorithms. Our results demonstrated that Aion reduces stream output latency by up to 85% while providing 95%

accuracy guarantees.

There remain several interesting SPE problems where the stream progress property could be leveraged. Specifically, similar to Klink's approach to runtime scheduling, the problem of elasticity can be studied in the context of minimizing the propagation delay of SWMs. The key insight is that SPEs need not to scale horizontally, even if currently overloaded, if they can process the queued events by the time the SWM is ingested. Otherwise, if the window deadline is not overdue, the SPE would be adding unnecessary resources to process queued events without impacting output latency.

References

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[2] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *the VLDB Journal*, 12(2):120–139, 2003.

[3] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming graph partitioning: an experimental study. In *PVLDB*, volume 11, pages 1590–1603, 2018.

[4] Swarup Acharya et al. Congressional samples for approximate answering of group-by queries. In *SIGMOD*, pages 487–498. ACM, 2000.

[5] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *SIGMOD*, pages 574–576. ACM, 1999.

[6] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EUROSYS*, pages 29–42. ACM, 2013.

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

[8] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[9] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems (DEBS)*, pages 207–218. ACM, 2013.

[10] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the VLDB Endowment*, volume 30, pages 480–491. ACM, 2004.

[11] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD International conference on Management of Data*, pages 253–264. ACM, 2003.

[12] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361. IEEE, 2004.

[13] Magdalena Balazinska, YongChul Kwon, Nathan Kuchta, and Dennis Lee. Moirae: History-enhanced monitoring. In *CIDR*, pages 375–386, 2007.

[14] Pablo Basanta-Val, Norberto Fernández-García, Andy J Wellings, and Neil C Audsley. Improving the predictability of distributed stream processors. *Future Generation Computer Systems*, 52:22–36, 2015.

[15] Michael A Bender, Soumen Chakrabarti, and Sambavi Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *SODA*, volume 98, pages 270–279, 1998.

[16] Jean-Chrysostome Bolot. Characterizing end-to-end packet delay and loss in the internet. In *Journal of High Speed Networks (JHSN)*, pages 305–323, 1993.

[17] Wei Cai, Ryan Shea, Chun-Ying Huang, Kuan-Ta Chen, Jiangchuan Liu, Victor CM Leung, and Cheng-Hsin Hsu. A survey on cloud gaming: Future of computer games. *IEEE Access*, 4:7605–7620, 2016.

[18] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[19] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings 2003 VLDB Conference*, pages 838–849. Elsevier, 2003.

[20] Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, Erik Sutherland, Nesime Tatbul, et al. S-store: a streaming newsql system for big velocity applications. *Proceedings of the VLDB Endowment*, 7(13):1633–1636, 2014.

[21] Badrish Chandramouli, Jonathan Goldstein, Roger Barga, Mirek Riedewald, and Ivo Santos. Accurate latency estimation in a distributed event processing system. In *2011 IEEE 27th International Conference on Data Engineering*, pages 255–266. IEEE, 2011.

[22] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[23] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: an embedded concurrent key-value store for state management. *Proceedings of the VLDB Endowment*, 11(12):1930–1933, 2018.

[24] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[25] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. In *TODS*, volume 32, pages 9–es. ACM, 2007.

[26] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)*, 32(2):9–es, 2007.

[27] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. Approximate query processing: No silver bullet. In *SIGMOD*, pages 511–519. ACM, 2017.

[28] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, 2015.

[29] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 1789–1792. IEEE, 2016.

[30] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *11th Annual Workshop on Network and Systems Support for Games, NetGames 2012, Venice, Italy, November 22-23, 2012*, pages 1–6. IEEE, 2012.

[31] Mark Claypool and David Finkel. The effects of latency on player performance in cloud-based games. In *2014 13th Annual Workshop on Network and Systems Support for Games*, pages 1–6. IEEE, 2014.

[32] Graham Cormode. Sketch techniques for approximate query processing.

[33] Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. *Synopses for massive data: Samples, histograms, wavelets, sketches*. Now Publishers, Inc., 2011.

[34] Graham Cormode, Flip Korn, Shanmugavelayutham Muthukrishnan, and Divesh Srivastava. Space-and time-efficient deterministic algorithms for biased quantiles over data streams. In *PODS*, pages 263–272. ACM, 2006.

[35] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. In *Journal of Algorithms*, pages 58–75. Elsevier, 2005.

[36] Miyuru Dayarathna and Srinath Perera. Recent advancements in event processing. pages 1–36. ACM, 2018.

[37] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644. Springer, 2006.

[38] Sergio Esteves, Gianmarco De Francisci Morales, Rodrigo Rodrigues, Marco Serafini, and Luís Veiga. Aion: Better late than never in event-time streams. *arXiv preprint arXiv:2003.03604*, 2020.

[39] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. Edgewise: a better stream processing engine for the edge. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 929–946, 2019.

[40] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. Moment-based quantile sketches for efficient high cardinality aggregation queries. In *PVLDB*, volume 11, page 1647–1660. ACM, 2018.

[41] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. Complex event recognition in the big data era: a survey. In *VLDBJ*, pages 313–352. Springer, 2020.

[42] Phillip B. Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD*, page 331–342. ACM, 1998.

[43] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 383–397. ACM, 2015.

[44] Lukasz Golab and M Tamer Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[45] Jamie Grier. Extending the yahoo! streaming benchmark. *URL hhttps://www.ververica.com/blog/extending-the-yahoo-streaming-benchmark*, 2016.

[46] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tufte. Frames: data-driven windows. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS)*, pages 13–24. ACM, 2016.

[47] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. Online aggregation. In *SIGMOD*, pages 171–182. ACM, 1997.

[48] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[49] Gabriela Jacques-Silva, Ran Lei, Luwei Cheng, Guoqiang Jerry Chen, Kuen Ching, Tanji Hu, Yuan Mei, Kevin Wilfong, Rithin Shetty, Serhat Yilmaz, et al. Providing streaming joins as a service at facebook. *Proceedings of the VLDB Endowment*, 11(12):1809–1821, 2018.

[50] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *Proceedings of the VLDB Endowment*, pages 431–442. ACM, 2006.

[51] Zbigniew Jerzak and Holger Ziekow. The debs 2015 grand challenge. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS)*, page 266–268. ACM, 2015.

[52] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. Quality-driven continuous query execution over out-of-order data streams. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 889–894. ACM, 2015.

[53] Theodore Johnson, Shanmugavelayutham Muthukrishnan, and Irina Rozenbaum. Sampling algorithms in a stream operator. In *SIGMOD*, pages 1–12. ACM, 2005.

[54] Nikos R Katsipoulakis et al. Spear: Expediting stream processing with accuracy guarantees. In *ICDE*. IEEE, 2020.

[55] Nikos R Katsipoulakis, Alexandros Labrinidis, and Panos K Chrysanthis. A holistic view of stream partitioning costs. *Proceedings of the VLDB Endowment*, 10(11):1286–1297, 2017.

[56] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pages 555–569. ACM, 2016.

[57] Dhanya R Krishnan, Do Le Quoc, Pramod Bhatotia, Christof Fetzer, and Rodrigo Rodrigues. Incapprox: A data analytics system for incremental approximate computing. In *WWW*, pages 1133–1144, 2016.

[58] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[59] Santoshkumar Kulkarni and Joarder Kamal. Amazon aws, Sep 2018.

[60] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: Mapreduce-style processing of fast data. *arXiv preprint arXiv:1208.4175*, 2012.

[61] Boduo Li, Yanlei Diao, and Prashant Shenoy. Supporting scalable analytics with latency constraints. *Proceedings of the VLDB Endowment*, 8(11):1166–1177, 2015.

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

[63] Kaiyu Li and Guoliang Li. Approximate query processing: What is new and where to go? In *Data Science and Engineering*, pages 379–397. Springer, 2018.

[64] Teng Li, Jian Tang, and Jielong Xu. Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Transactions on Big Data*, 2(4):353–364, 2016.

[65] Ling Liu, Calton Pu, Roger Barga, and Tong Zhou. Differential evaluation of continual queries. In *ICDCS*, pages 458–465. IEEE, 1996.

[66] S.L. Lohr. *Sampling: Design and Analysis*. Brooks/Cole, 2010.

[67] Federico Lombardi, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni. Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):572–585, 2017.

[68] R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *UCC*, pages 69–78. IEEE/ACM, 2015.

[69] Gurmeet Singh Manku et al. Approximate medians and other quantiles in one pass and with limited memory. *SIGMOD*, 27(2):426–435, 1998.

[70] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. In *International Conference on Database Theory (ICDT)*, pages 398–412, 2005.

[71] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. Streambox: Modern stream processing on a multi-core machine. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 617–629, 2017.

[72] Lory Al Moakar, Thao N Pham, Panayiotis Neophytou, Panos K Chrysanthis, Alexandros Labrinidis, and Mohamed Sharaf. Class-based continuous query scheduling for data streams. In *Proceedings of the Sixth International Workshop on Data Management for Sensor Networks*, page 9. ACM, 2009.

[73] Aloysius Ka-Lau Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[74] Barzan Mozafari and Carlo Zaniolo. Optimal load shedding with aggregates and mining queries. In *ICDE*, pages 76–88. IEEE, 2010.

[75] Shanmugavelayutham Muthukrishnan, Rajmohan Rajaraman, Anthony Shaheen, and Johannes E Gehrke. Online scheduling to minimize average stretch. In *40th Symp. Foundations of Computer Science Science (FOCS)*, pages 433–443. IEEE, 1999.

[76] C. Mutschler and M. Philippsen. Distributed low-latency out-of-order event processing for high data rate sensor streams. In *IPDPS*, pages 1133–1144. IEEE, 2013.

[77] Christopher Mutschler, Holger Ziekow, and Zbigniew Jerzak. The debs 2013 grand challenge. In *DEBS*, page 289–294. ACM, 2013.

[78] Snehal Nagmote and Pallavi Phadnis. Massive scale data processing at netflix using flink. *Flink Forward Conference*, 2019.

[79] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, Nicolas Kourtellis, and Marco Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 589–600. IEEE, 2016.

[80] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. Haren: A framework for ad-hoc thread scheduling policies for data streaming applications. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems (DEBS)*, pages 19–30. ACM, 2019.

[81] Thao N Pham, Panos K Chrysanthis, and Alexandros Labrinidis. Avoiding class warfare: managing continuous queries with differentiated classes of service. *Proceedings of the VLDB Endowment*, 25(2):197–221, 2016.

[82] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. Streamapprox: approximate computing for stream analytics. In *Middleware*, pages 185–197. USENIX, 2017.

[83] Nicoló Rivetti, Yann Busnel, and Leonardo Querzoni. Load-aware shedding in stream processing systems. In *DEBS*, page 61–68. ACM, 2016.

[84] Nicolo Rivetti, Nikos Zacheilas, Avigdor Gal, and Vana Kalogeraki. Probabilistic management of late arrival of events. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS)*, pages 52–63. ACM, 2018.

[85] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. Reinforcement learning based policies for elastic stream processing on heterogeneous resources. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems (DEBS)*, pages 31–42. ACM, 2019.

[86] Sven Schmidt, Thomas Legler, Daniel Schaller, and Wolfgang Lehner. Real-time scheduling for data stream management systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 167–176. IEEE, 2005.

[87] Mohamed A Sharaf, Panos K Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. Efficient scheduling of heterogeneous continuous queries. In *Proceedings of the VLDB Endowment*, volume 32, pages 511–522. ACM, 2006.

[88] Mohamed A Sharaf, Panos K Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Transactions on Database Systems (TODS)*, 33(1):5, 2008.

[89] Anshumali Shrivastava, Arnd Christian Konig, and Mikhail Bilenko. Time adaptive sketches (ada-sketches) for summarizing data streams. In *SIGMOD*, pages 1417–1432. ACM, 2016.

[90] Ethan Siegel. Forbes, Sep 2017.

[91] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *PODS*, page 263–274. ACM, 2004.

[92] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[93] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *PVLDB*, pages 309–320. ACM, 2003.

[94] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *PVLDB*, pages 159–170. ACM, 2007.

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

[96] Tolga Urhan and Michael J Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *Proceedings of the VLDB Endowment*, volume 1, pages 501–510. ACM, 2001.

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

[98] Jeffrey S Vitter. Random sampling with a reservoir. In *TOMS*, volume 11, pages 37–57. ACM, 1985.

[99] Vanish Talwar Michael Y. Levin Gabriela Jacques da Silva Nikhil Simha Anirban Banerjee Brian Smith Tim Williamson Serhat Yilmaz Weitao Chen Guoqiang Jerry Chen Yuan Mei, Luwei Cheng. Turbine: Facebook's service management platform for stream processing. In *2016 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 589–600. IEEE, 2020.

[100] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos. Elastic complex event processing exploiting prediction. In *BigData*, pages 213–222. IEEE, 2015.

[101] Nikos Zacheilas, Vana Kalogeraki, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. Maximizing determinism in stream processing under latency constraints. In *DEBS*, pages 112–123. ACM, 2017.

[102] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[103] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. Analyzing efficient stream processing on modern hardware. *Proceedings of the VLDB Endowment*, 12(5):516–530, 2019.

[104] Tan Zhang, Aakanksha Chowdhery, Paramvir Bahl, Kyle Jamieson, and Suman Banerjee. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 426–438, 2015.