Klink: Progress-Aware Scheduling for Streaming Data Systems

Omar Farhat, Khuzaima Daudjee
Cheriton School of Computer Science
University of Waterloo
{obfarhat,kdaudjee}@uwaterloo.ca

Leonardo Querzoni
DIAG
Sapienza University of Rome
querzoni@diag.uniroma1.it

ABSTRACT

Modern stream processing engines (SPEs) process large volumes of events propagated at high velocity through multiple queries. To improve performance, existing SPEs generally aim to minimize query output latency by minimizing, in turn, the propagation delay of events in query pipelines. However, for queries containing commonly used blocking operators such as windows, this scheduling approach can be inefficient. Watermarks are events popularly utilized by SPEs to correctly process window operators. Watermarks are injected into the stream to signify that no events preceding their timestamp should be further expected. Through the design and development of Klink, we leverage these watermarks to robustly infer stream progress based on window deadlines and network delay, and to 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. We integrate Klink into the popular open source SPE Apache Flink and demonstrate that Klink delivers significant performance gains over existing scheduling policies on benchmark workloads for both scale-up and scale-out deployments.

KEYWORDS

Stream processing; Scheduling; Windows; Watermarks

ACM Reference Format:
Omar Farhat, Khuzaima Daudjee and Leonardo Querzoni. 2021. Klink: Progress-Aware Scheduling for Streaming Data Systems. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3452794

1 INTRODUCTION

Streaming systems are popularly used by modern applications driven by the need to process quickly [50] large volumes of high-velocity data [14]. Application domains including real-time analytics, anomaly detection, and real-time object recognition [57] leverage streaming systems to deliver on their data processing requirements [16, 31, 35, 39]. Existing stream processing engines (SPEs) such as Apache's Flink [12] and Spark [55] are designed to provide high throughput with low-latency processing in responding to demands of such applications [35, 50, 54].

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452794

SPEs provide event processing semantics where streaming *queries* are defined by multiple processing units called *operators* deployed over one or more computing nodes [22, 50]. Operators are then scheduled for execution at run-time on the available CPUs. System designers carefully provision streaming applications such that the SPE can schedule operators from all queries while limiting contention and maximizing resource utilization. However, many applications have heavy input loads, often with fluctuating or unpredictable load spikes, degrading the SPE's performance. In such resource-challenged settings, the SPE runs application(s) under high resource contention among the concurrent queries. This contention typically causes events to be buffered for relatively long periods of time in the input queues of operators that are waiting to be executed, degrading throughput and overall end-to-end latency. In particular, tail latencies can be significantly affected resulting in high output latencies. This performance degradation can be severe, leading to violation of application service requirements.

Modern SPEs maximize throughput [24] by performing micro-batching [55], query scheduling [6] and elastic resource provisioning [21, 34, 45]. Some proposals strive to reduce latency in resource-challenged settings. In particular, operator scheduling policies such as Highest Rate [48] aim to reduce output latency by minimizing the mean propagation delay of events in the pipeline, i.e., 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, these strategies do not perform well if queries contain latency-sensitive operators such as *window* and *join* [25]. Specifically, window operators block the stream from flowing until their input is complete. Thus, minimizing the mean event propagation delay does not necessarily translate to minimized output latency if deadlines of window operators are not properly taken into account. For instance, a window operator that is not ready to produce its output may be scheduled before one whose deadline has already elapsed yet is still waiting in the scheduler queue for a CPU time slice.

To exemplify the problem, consider an SPE deployed on a standard processor core running two queries q_i and q_j. Each query contains a window operator with q_j's window having its upcoming deadline elapse before q_i's deadline. If the SPE is oblivious of q_j's input stream progress with respect to its window deadline, the SPE can process q_i over q_j, thereby inducing unwarranted output latency. Processing q_j first would allow the SPE to prioritize (earlier) production of output. 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.

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

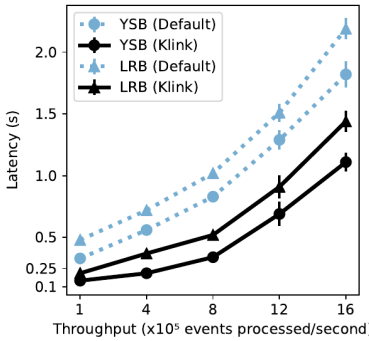


Figure 1: Average output latency vs. SPE throughput (number of events processed per second)

(i) Identifying events *relevant* to window operators is difficult, i.e., events that the operator must consider when computing the next windowed output. Events can be arbitrarily delayed for ingestion due to, e.g., network delays or parallel and distributed executions, while windows claim events based on their generation order at the source.

(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 as much as twenty seconds to account for any arbitrary delays (e.g., network or event processing delays). Hence, it is challenging to correctly identify and prioritize queries that are due to be processed.

In this paper, we show that the best *strategy* to minimize output latency is to prioritize the propagation of 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 further demonstrate the performance impact of this scheduling problem, we measured the output latency while varying the throughput in terms of number of input events processed per second for two different query workloads and complexities, namely the Yahoo! Streaming Benchmark (YSB) [18] and the Linear Road Benchmark (LRB) [7].

We deployed Flink on a machine[1] to process these two workloads using Flink's *Default* scheduler as well as Klink, the scheduler proposed in this paper that implements the aforementioned strategy. Figure 1 shows that given a target throughput level (x-axis), significant extra output latency of about 50% for both YSB and LRB is incurred by Default over Klink. This overhead is further exacerbated for SPEs under significant resource contention as is typical in real deployments where network delay is variable, imposing the aforementioned challenges (i) and (ii). Output latency is small under light loads but increases exponentially as the workload grows, as expected from a system under resource stress. The significant performance gains by Klink demonstrates that SPEs can benefit greatly if schedulers are designed to minimize output latencies even at comparable throughput performance.

[1]See Sec. 6 for the machine's hardware specs.

In this paper, we present the design and implementation of **Klink**, our scheduler that optimizes for stream progress to reduce output latency for queries running window operators including joins.

We leverage *watermarks* [32] that are special events commonly generated and injected in SPEs to solve aforementioned challenges (i) and (ii). Watermarks are in widespread use in pipelines encompassing window operators [4, 5, 12, 51] and indicate that no further events are expected beyond a certain timestamp. The periodic propagation of watermarks in the stream encapsulates an indicator of progress vital to recognizing the input completion level for window operators. Klink uses this indicator to prioritize the execution of queries that include window operators that are due to be processed first. Klink also takes into account memory usage of window operators so as to avoid the SPE from becoming a bottleneck. Furthermore, Klink's scheduling strategy is compatible with both stream processing and micro-batching SPE architectures.

To summarize, the contributions of this paper are five-fold:

(1) We present Klink, a scheduler optimized for running multiple queries, delivering up to 60% mean and tail output latency reductions over state-of-the-art techniques.
(2) Klink presents a significant advancement over existing scheduling policies in that it optimizes to minimize output generation latency by processing the relevant events to progress window operators into materializing their results.
(3) Klink's design is robust to workload changes and is adept at handling network delay variability and at managing memory to optimize for performance.
(4) We present a distributed design for Klink capable of achieving its goal for stream queries running on multiple nodes.
(5) We leverage the generality of Klink's design to present our system implementation of Klink that is integrated into Apache Flink [12], a popular state-of-the-art SPE.

2 BACKGROUND

This section discusses windows, out-of-order processing and watermarks, after which we present our design of Klink.

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 for queries through hosting complex grouping selections [23].

Following [30], we assume that each *event* e is an ordered set of values and includes a timestamp $\tau(e)$ generated at the source, called *event-time*. An infinite sequence of events $S = \langle e_1, e_2, \dots \rangle$ is called a *stream* and S^* is the set of all streams. A window is a subset of events in a stream and the set of all windows is $\hat{S}^* \subset S^*$, while $(\hat{S}^*)^*$ is the set of potentially infinite window sequences. A query q based on a window and implemented by an operator with n input streams is defined by (i) n *window functions* $\langle \omega_1^q, \dots, \omega_n^q \rangle$ where $\omega_i^q : S^* \rightarrow (\hat{S}^*)^*$, (ii) an n-ary *operator function* $\mathcal{F}^q : (\hat{S}^*)^n \rightarrow \hat{S}^*$ that, applied to n windows from input streams, produces a window result and (iii) a *stream function* $\phi^q : (\hat{S}^*)^* \rightarrow S^*$ that transforms window results into output stream events.

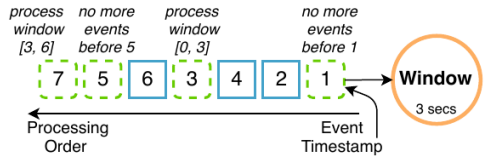


Figure 2: Exemplifying progress property of watermarks.

Window functions can be *count-based* or *time-based*, if the windows they produce are defined on the number of included events or the event-time frames they span, respectively. Window functions on a stream are each characterized by a size s and a slide l; these two parameters, combined, define deadlines; a window's deadline identifies when that window contains all events needed to produce its output. More formally, consider a window function $\omega_{(s,l)}$ that, when applied on a stream S, produces a sequence of windows $\langle w_1, w_2, \dots \rangle$. A count-based ω function will produce a window $w_i = \langle e_k, \dots, e_m \rangle$ where $m = k + s - 1$ and whose deadline is the event e_m. Conversely, a time-based ω function will produce a window w_i and next window $w_{i+1} = \langle e_{k'}, \dots, e_{m'} \rangle$ such that $\tau(e_m) - \tau(e_k) \leq s$, $\tau(e_{m+1}) - \tau(e_k) > s$, $\tau(e_{k'}) - \tau(e_k) \leq l$ and $\tau(e_{k'+1}) - \tau(e_k) > l$ where the deadline is met every l time units. Output generation for a given window w_i can be triggered on the windowed operator as soon as the corresponding deadline is met.

Windows are important as query operators such as join, aggregation, and selection are executed on a per window basis [30]. However, the order of events received by event-time windows may not necessarily mirror the order of events generated at the source. Such out-of-order events occur in SPEs for multiple reasons: (i) events generated from different remote origins, transmissions or channels may take different paths incurring different delays, and (ii) operators 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.

For SPEs, late events make window input completeness difficult to attain as deadlines cannot be guaranteed. Window operators cannot ascertain that all events to be included for a window's computation to proceed 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. SPEs deal with these challenges by adopting one of the following approaches:

• In-order processing (IOP): the SPE uses mechanisms that enforce events to be processed in the order defined by their event-time. This approach typically imposes large performance overheads [32] as in-order processing can perilously delay the processing of events until they are appropriately re-ordered to guarantee correctness.

• Out-of-order processing (OOP): the SPE allows window operators to be unblocked without imposing any ordering constraints on the stream [28, 32]. Late events can be discarded or pushed as output that may be considered by the receiving application to correct previous output.

Most modern SPEs such as Flink, Spark and Storm [4, 5, 12] support both IOP and OOP processing by using watermarks.

2.2 Watermarks

Watermarks [32] are timestamped events denoting that no events preceding their timestamp should be further expected. If we consider a watermark event $\dot{e}_k$ in a stream S such that $S = \langle e_1, \dots, e_{k-1}, \dot{e}_k, e_{k+1}, \dots \rangle$, an operator O processing $\dot{e}_k$ may expect that any event e_i following $\dot{e}_k$ has a larger timestamp, i.e. $\tau(e_i) > \tau(\dot{e}_k)\ \forall i > k$.

Watermarks 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 (even in cases where no event is injected by the sources). Watermarks are crucial to correctly implement OOP with streams where events may not be ordered by their timestamps. Once a watermark reaches an operator, the operator can advance its (internal) event time clock to the value of the watermark, and check which window deadlines have been met. Recall that the watermark's semantics imply that all events relevant for these windows have already been received by the operator [5]. In particular, a windowed operator implementing a query q that receives a watermark with timestamp $\tau(\dot{e})$ will check which windows have a deadline that elapsed by $\tau(\dot{e})$ and will trigger the computation of the corresponding outputs using its function ϕ^q.

To illustrate this with an example, consider Fig. 2 where a window operator with window function $\omega_{(3,3)}$ spanning three seconds starts operating at event time 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 and generate the output of the first window $[0, 3]$. The window $[3, 6]$ with event 6 and watermarks 5 and 7 is similarly processed next.

Watermarks take the form of *punctuations* [49, 52] and are injected into the stream either (i) at the source, or (ii) by a specific operator that periodically emits them [4, 12, 20]. In both cases, applications decide on their implementation logic for generating watermarks. 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 means that events can be delayed at most five seconds. 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).

Watermarks are propagated by the SPE in the application pipeline and are expected to be received with monotonically increasing timestamps by operators. Operators are required to completely process events that precede a given watermark before forwarding it downstream. SPEs implement watermark propagation using various approaches (e.g., dropping late events, buffering, reordering). For example, the Flink SPE [12] drops late watermarks, i.e., when they arrive out-of-order at the SPE.

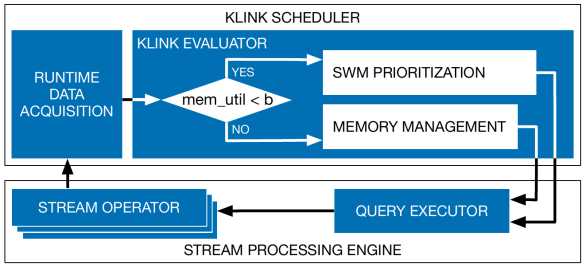


Figure 3: Klink Architecture

In the example of Fig. 2, note that different watermarks have different meanings for the window operator. Watermarks 1 and 5 act as progress indicators 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, 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 $[3, 6]$. Note that 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 output. Two key invariants hold in processing SWMs: (i) propagating an SWM to a window operator implies that all the events to be included in windows whose deadlines precede the SWM timestamp 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 events to be included in a window's computation 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 that are received by the output operator after the window output events. Therefore, all the events to be included in a window's computation are guaranteed to have been processed beforehand.

The aforementioned invariants give rise to the following two important observations:

(i) Minimizing 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.

(ii) 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 queued events 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 form the basis of Klink's design.

3 KLINK: DESIGN AND ALGORITHMS

We now present the design of Klink including its algorithmic details. Klink (Fig. 3) is composed of (i) a module that acquires runtime data, and (ii) a stream analysis algorithm, Klink Evaluator, that implements two query prioritization policies: SWM Prioritization that prioritizes queries to minimize output latency (Sections 3.1 and 3.2), and Memory Management that prioritizes queries to minimize memory utilization stress (Sec. 3.4). Klink uses the former policy unless the measured runtime memory utilization meets or exeeds a configurable bound b, which makes Klink transiently switch to the latter policy. The Klink architecture also includes a third component, which is a distributed design that enables the priority to be computed at each node and then propagated in a decentralized manner, enabling scaling with the SPE infrastructure (Sec. 4).

In a nutshell, Klink's main loop periodically acquires runtime data, computes a priority for each query, and schedules execution of the query with the largest priority. Priorities are assigned to speed up the propagation of events to window operators that are due to be processed first. If the memory stress condition is met (i.e., $mem\_util \geq b$ in Fig. 3), Klink's strategy prioritizes scheduling of those operators whose execution will release the most memory.

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, 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 network delay, queue size, cost, and selectivity. The window operators 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 deadlines 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 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 overlaps with two windows, i.e., tumbling window $[3, 6]$ followed by $[6, 9]$. Similarly, for a sliding window of size five seconds and slide one second, the first epoch is defined between time 0 and SWM 5, the second epoch between SWMs 5 and 7. Thus, with each processed window, Klink progresses the query to the next epoch. Klink also groups the collected information on a per-epoch basis to impart temporal context to future estimations.

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 [33],

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 [33], the cost of processing an event can be represented by using the standard measures of per operator mean processing latency, queuing delays, and communication latencies between operators. These factors for estimation of the processing cost are encapsulated in a tuple $\mathcal{I}$ provided by the runtime data acquisition module. The priority of each query is then computed based on the retrieved information, and on the currently applied policy. The query with the highest priority is returned and is then scheduled for execution. Klink does not impose requirements on the query execution model: operators are deployed by the SPE (and per its standard behavior) as threads/processes on available computing cores and are then executed with priorities defined per Klink's policy.

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. 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 implies missing the deadlines for idle queries. As shown in Section 6, Klink's scheduling overhead is low.

Klink's priority evaluator 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:

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

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. 1), it is essential to determine w^q_{n+1}. We describe Klink's robust estimation technique next.

3.1 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:

$$E[w^q_{n+1}] = E[d^q_n + p^q] \tag{2}$$

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 stream progress as long as the query is continuously ingesting events and is monitoring the network delay. Once the epoch is finalized, d^q_n as a random variable is then represented by only events constituting the n^{th} epoch. We compile this definition into the following equation:

$$\mu^q_n = \begin{cases} \frac{1}{|\mathcal{D}^q_n|} \times \sum_{d \in \mathcal{D}^q_n} d & \text{if } t \geq w^q_n, \\ \frac{1}{n-1} \times \sum_{i=0}^{n-1} \mu^q_i & \text{otherwise.} \end{cases} \tag{3}$$

We define χ^q_n as the square of each distribution in the following equation:

$$\chi^q_n = \begin{cases} \frac{1}{|\mathcal{D}^q_n|} \times \sum_{d \in \mathcal{D}^q_n} d^2 & \text{if } t \geq w^q_n, \\ \frac{1}{n-1} \times \sum_{i=0}^{n-1} \chi^q_i & \text{otherwise.} \end{cases} \tag{4}$$

For the case of $t < w^q_n$, we observe that the random variable d^q_n is a function of the expected delay over the previous epochs. Hence, by the Central Limit Theorem, d^q_n is normally distributed, and in turn, w^q_{n+1} is also normally distributed. To better understand this distribution, we calculate the mean (Eq. 5) and variance (Eq. 6) of w^q_{n+1}:

$$\begin{aligned} E[w^q_{n+1}] &= E[d^q_n + p^q] \\ &= \frac{1}{n-1} \sum_{i=0}^{n-1} E[d^q_i] + E[p^q] = \mu^q_n + p^q \end{aligned} \tag{5}$$

$$\begin{aligned} Var[w^q_{n+1}] &= E[(w^q_{n+1})^2] - E[w^q_{n+1}]^2 \\ &= \frac{1}{n-1}[\chi^q_n + \frac{1}{n-1} \sum_{0 \leq i \neq j}^{n-1} \mu^q_i \mu^q_j] - (\mu^q_n)^2 \end{aligned} \tag{6}$$

Thus, w^q_{n+1} is normally distributed with mean $\mu^q_n + p^q$ and variance denoted by Eq. 6. By exploiting this distribution, we can estimate a time-interval to infer the range of SWM's ingestion timestamp. This is discussed further in the next section (Sec. 3.2).

Studying the distribution of w^q_{n+1} allows us to estimate the arrival of the SWM with high confidence. We present next an algorithm that exploits this characteristic to compute the slack time sl^q for query q.

3.2 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 1 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:

$$P(t^q_{n,min} \leq w^q_{n+1} \leq t^q_{n,max}) = f \tag{7}$$

where $t^q_{n,min}$ and $t^q_{n,max}$ enclose 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. 6).

After delimiting the search space, Klink slides a window of size r over the time-range computed by Eq. 7. Then, for each window

slide, the slack value is computed as:

$$sl^q(t) = \sum_{x=t^q_{n,min}}^{t^q_{n,max}} P(x \le w^q_{n+1} \le x+r|t) \times((x+r-t) - cost^q(t)) \quad (8)$$

where x is incremented by r milliseconds.

Eq. (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. 8) can be computed as:

$$P(x \le w^q_{n+1} \le x+r|t) = \begin{cases} \frac{P(x \le w^q_{n+1} \le x+r)}{P(w^q_{n+1} \ge t)} & \text{if } x \le t, \\ 0 & \text{otherwise.} \end{cases} \quad (9)$$

Since w^q_{n+1} is normally distributed, the probabilities can be approximated by the Guassian Q-function as in Eq. 9:

$$P(x \le w^q_{n+1} \le x+r) = Q(\frac{E[w^q_{n+1}] - (x+r)}{Var(w^q_{n+1})}) - Q(\frac{E[w^q_{n+1}] - x}{Var(w^q_{n+1})}) \quad (10)$$

Algorithm 1 Klink's Slack Computation

```
1:  procedure ComputeConfInterval(q)
2:      μ^q_n = (1/h) Σ_{i=n-h}^{n-1} μ^q_i; χ^q_n = (1/h) Σ_{i=n-h}^{n-1} χ^q_i
3:      E[w^q_{n+1}] = μ^q_n + p^q; σ[w^q_{n+1}] = √(Eq. 6)
4:      /* Compute ≥ 95% interval */
5:      t^q_{n,min} = E[w^q_{n+1}] − 2σ[w^q_{n+1}]
6:      t^q_{n,max} = E[w^q_{n+1}] + 2σ[w^q_{n+1}]
7:      return t^q_{n,min}, t^q_{n,max}
8:  end procedure
9:  procedure ComputeExpectedSlack(q, t)                ▷ Eq. (8)
10:     t^q_{n,min}, t^q_{n,max} = ComputeConfInterval(q)
11:     sl^q = 0; x = max(t, t^q_{n,min})
12:     for x ≤ t^q_{n,max} do
13:         pr = P(x ≤ w^q_{n+1} ≤ x+r) / P(w^q_{n+1} > t)
14:         sl^q = sl^q + pr × [(x + r − t) − cost^q(t)]
15:         x = x + r
16:     end for
17:     return sl^q
18: end procedure
19: procedure KlinkEvaluator(Q, I)
20:     unpack(I); min_sl = 0; min_q = null
21:     for each query q do
22:         sl^q = ComputeExpectedSlack(q, t)
23:         if sl^q ≤ min_sl then
24:             min_sl = sl^q; min_q = q
25:         end if
26:     end for
27:     return min_q
28: end procedure
```

We detail the aforementioned slack time estimation in Algorithm. 1. Initially, the procedure KlinkEvaluator (line 19) is invoked by Klink's main loop where the set of queries and collected information are passed as parameters. The algorithm then unpacks the collected runtime information (line 20). The set $\mathcal{I}$ contains important information per query q: upcoming window deadline, last watermark processed, experienced network delays $\mathcal{D}^q$, and the data collected over previous h epochs μ^q_n and χ^q_n. The set also contains 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. 8 (lines 12-16). At first, the algorithm divides the time-range into smaller ranges whereby each small range's 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).

Figure 4: Example illustrating a window operator joining two input streams of SWMs into an output stream of SWMs.

3.3 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 Klink's mechanism to efficiently handle join operators.

To illustrate joins, consider the Fig. 4 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 with deadline $ddl = 2$ was not unblocked until the ingestion of SWM 3 at the top stream. Although SWM 3 unblocked the window with $ddl = 2$, it did not unblock the

window with $ddl = 3$ since no SWMs were propagated by the bottom stream. The window with $ddl = 3$ 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 windowed join 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 a different watermark propagation rate and network delay. The slack of the query is then calculated as the minimum of each stream. Thus, the procedure call in Algorithm 1 (line 27) returns the minimum slack for query q. This design ensures that accurate prioritization is determined and maintained.

3.4 Klink's Memory Management

While latency optimization is a prominent goal in stream processing, application 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 can contend on scarce memory resources, thereby blocking stream flow and increasing output latency [8, 17, 19]. Many SPEs 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, independently 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 discards 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 [32]) 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[2] 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 q's pipeline 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. Since Klink runs queries for r milliseconds before commencing the subsequent scheduling cycle, 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, 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. 6). 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 output latencies, as demonstrated in Section 6.

4 DISTRIBUTED KLINK DESIGN

SPEs leverage distributed computing to achieve greater scalability and meet performance goals [40]. 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 while maintaining 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 nodes [6, 42]. 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 query operators on available nodes to which operators have been assigned by the SPE. 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.

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

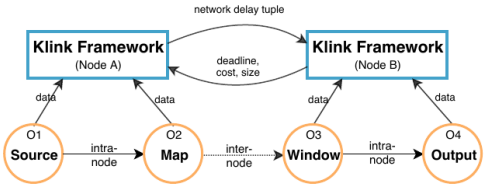


Figure 5: Klink forwarding information in distributed environment.

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. 5 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 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 running the source operator, so information should be forwarded to all nodes running downstream 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 running down- and upstream 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 is forwarded to nodes, each Klink instance computes its priorities and executes accordingly. The scheduler collects information from other Klink instances as part of its main loop. We discuss Klink's implementation details in the following section.

5 SYSTEM IMPLEMENTATION

This section details our implementation of Klink [1] within the open-source distributed SPE Apache Flink [12]. Flink is implemented in Java as a layered system with logical separation between different layers. The three main layers are named Deploy, Core and API. The top layer, API, contains two segregated engines 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 schedules the execution of submitted applications in two stages. At deployment time, the application is analyzed and its operators (possibly aggregated in chains) assigned for execution on task slots located on worker nodes. Then, at runtime a task scheduler on each worker node executes operators as threads in a Java Virtual Machine (JVM) instance. The JVM's runtime scheduling of these Tasks is performed in conjunction with the OS scheduler.

Flink provides no infrastructural support to design or implement different runtime scheduling policies. 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. Here we describe this framework and how we leveraged its capabilities to implement Klink within Flink.

Several SPE architectures that include runtime schedulers have been proposed [2, 13, 41]. In contrast to a thread-based execution model in which each incoming event is allocated a thread, Aurora [13] and Borealis [2] 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.

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 $\mathcal{I}$ (line 20, Algorithm. 1), 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. 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. Finally, every r seconds, the runtime scheduler will *pause* the currently running Tasks, and will *start* the execution of those newly scheduled Tasks.

Klink's distributed design exchanges collected runtime information across nodes, as described in Section 4. 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.

Klink requires the underlying SPE to provide support for OOP, watermarks, and runtime data acquisition; since modern SPEs share similar design architectures and provide such support, the design of our runtime scheduler can be easily ported into those engines. For instance, Klink can be integrated into Apache Storm [51] 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.

6 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.

6.1 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. Our implementation of Klink is based on Apache Flink v1.8. One machine is dedicated to workload generation. Input data is transmitted to the SPE nodes via Kafka v2.2.1.

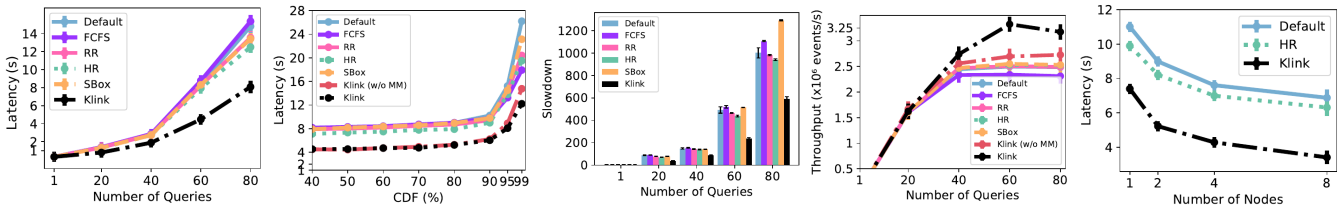
6.1.1 Benchmarks. We conduct our evaluation using three well-known streaming benchmarks: the Yahoo! Streaming Benchmark (**YSB**) [18], Linear Road Benchmark (**LRB**) [7], and the New York City Taxi (**NYT**) benchmark [27]. We implement these benchmarks on Apache Flink and evaluate performance by running different scheduling policies in each experiment.

LRB simulates a highway toll system [7]. We use the streaming variation [26] of LRB that has a complex pipeline 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. LRB contains a join (group by) over 3 sub-streams of 6.5K events produced every two seconds per sliding window per query. The workload is generated using the original driver data from [7]. 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 operator to be 1/3 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 NYC 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 complex pipeline that includes a sequence of many stateless operators and a sliding aggregation window of size two seconds and a slide of one second. NYT generates aggregation of 7K events produced every second per sliding window per query. YSB contains a simple pipeline with aggregation of 10K events produced every three seconds per window per query.

6.1.2 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 SWMs as indicative of the latency at which an SPE is able to produce output events. Latency is measured by subtracting the SWM event-time from the timestamp of its processing at the output operator. We empirically use the lowest latency marker frequency to achieve 99% similarity to the actual event latency without affecting performance. In our experiments, this amounted to emitting a latency marker from each source every 200 ms. We also measure throughput, by measuring the aggregate number of events processed per second by each operator. Finally, slowdown [48] 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.

6.1.3 Scheduling Algorithms. In addition to the default Flink scheduler (Default), we compare Klink against two standard heuristic algorithms and two state-of-the-art algorithms that we implemented into the runtime scheduler (Sec. 5):

- Round-Robin (RR) : cycles over the set of active operators and schedules the first operator that is ready to execute for a fixed time quantum. Notably, RR avoids starvation.
- First-Come-First-Served (FCFS): processes an input stream in event arrival order, optimizing for maximum output latency of requests.
- Highest Rate (HR) [48] 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, 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.
- StreamBox (SBox) [36] 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.



(a) Mean latency (b) Latency CDF (c) Slowdown (d) Throughput (e) Distributed

Figure 6: Mean latency, CDF, Slowdown, and Throughput for YSB workload

6.2 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 [11, 44]. Based on our empirical experimentation, we set Klink's size of epochs history h to 400 and the scheduling cycle duration r to 120 ms for robust performance. We use a default confidence value f of 95 as the accuracy for estimating SWM ingestion time (we study the sensitivity of f in Section 6.2.5).

6.2.1 YSB Benchmark. The first experiment compares the performance of the default Flink scheduler, denoted by Default, as well as the five other scheduling algorithms (FCFS, RR, HR, SBox and Klink) on a single node running the YSB benchmark with uniformly distributed network delays. We also study the performance of Klink without our memory management policy but with Flink's backpressure mechanism being able to kick in. We use this policy, which we name 'Klink (w/o MM)', to study the impact on tail latency in comparison to Klink with memory management.

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 ranging from 1–80.

With increase in the number of deployed queries (Fig. 6a), the mean latency for Klink is capped at 7.3s, reducing the delay to provide large performance improvements of about 50% over Default, SBox, FCFS and RR, and 45% over HR. FCFS incurs the highest latency of 15.5s at 80 concurrent queries. HR and SBox only marginally increase their performance over Default by incurring an output latency of 12.8s and 13.5s, respectively, compared to Default's 15s, with RR's performance being equivalent to SBox. Since the other algorithms perform limited or no prioritization of window deadlines, they do not exhibit much of a performance difference with each other.

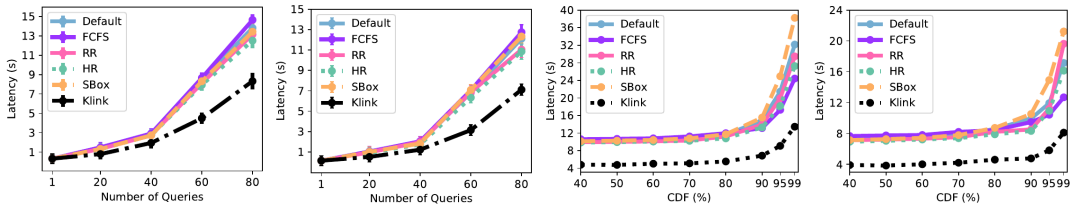
Fig. 6b compares the output latency CDF of Klink with the other scheduling algorithms and, in particular, shows their tail latency performance when running 60 concurrent queries. All scheduling algorithms maintained consistent latency performance between the 40^{th} and 90^{th} percentiles with a significant gap between Klink and the other algorithms. For the tail latency (90^{th} – 99^{th} percentile), Default's performance degraded from 9s at the 90^{th} percentile to 26s at the 99^{th} percentile indicating 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. FCFS improves tail latencies, providing marginally better performance than other non-Klink algorithms.

Klink achieves significantly better latency performance across all percentiles. For instance, at the tail latency of 99^{th} percentile, Klink significantly reduces latency by 55% over Default. Interestingly, Klink equipped with the memory management technique (Sec. 3.4) 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 approach to memory management allows it to deliver robust performance even under this challenging environment.

We measure the slowdown (Sec. 6.1.2) incurred by each algorithm in Fig. 6c under the same workload settings as in the 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. 6d shows throughput while varying the number of deployed YSB queries. Default, FCFS, RR, HR, and SBox all achieve the same throughput of 2.5M events processed per second. Klink (w/o MM) delivers throughput of 2.65M. 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. 6a). Klink with its memory management algorithm (Section 3.4) 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 approach attains scalable system performance.

6.2.2 LRB and NYT Benchmarks. Our third experiment runs LRB and NYT with network delays under the Uniform distribution. Fig. 7 shows these results. In this experiment, Default, FCFS, RR, HR, and SBox perform similarly with the latency at 80 queries ranging from 12 – 15s for all algorithms. Klink delivers large latency reductions of at least 45% over these algorithms for both LRB and NYT. As in



(a) LRB mean latency (b) NYT mean latency (c) LRB CDF (d) NYT CDF

Figure 7: Mean latency and CDF for LRB and NYT workloads

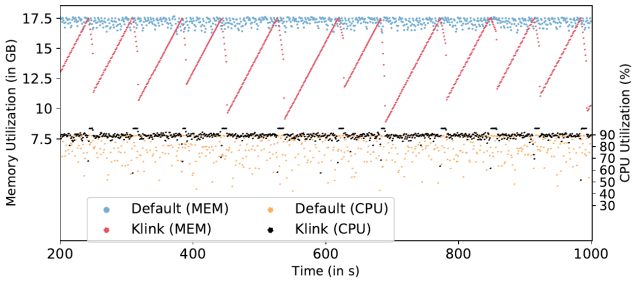


Figure 8: Memory & CPU utilization over time

YSB (Fig. 6a), the latency in Fig. 7 over all environment variables worsens past 40 queries due to the SPE's inability to scale with larger loads. Latency performance under Zipf distribution for YSB, LRB and NYT is similar to Figs. 6a, 7a and 7b so we omit its graph due to space constraints.

Figure 7 presents the latency CDF 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. In both tests Klink achieves 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. 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.

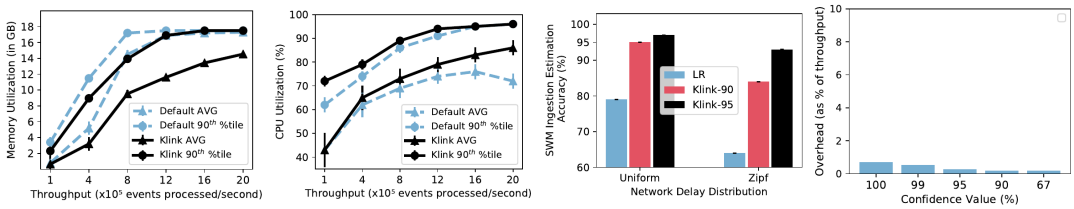
6.2.3 Resource Utilization. As an insight into Klink's performance advantages, we measured Klink's memory and CPU utilization running the YSB workload at 60 concurrent queries. Figure 8 shows the observed utilizations over time where each data point is an aggregate of values sampled every 200ms. The Default scheduler causes the SPE to continually run close to the maximum memory threshold (17.5GB in our experiments), as shown in the upper half of Fig. 8. Conversely, Klink manages memory by periodically increasing and decreasing its usage according to its active scheduling policy. Overall, Klink maintains significantly lower memory utilization. Figure 9a shows Klink's average and tail memory utilization compared to Default. Over the throughput range from 8 to 16 ($\times 10^5$ events/sec), Klink consumes between 60% to 25% less memory over Default. As for tail (90^{th} percentile) utilization, Default hits the memory threshold much earlier at 8×10^5 events/sec while Klink is able to extend this reach to 16×10^5 events/sec, thus doubling performance.

As shown in the lower half of Fig. 8, Klink consistently maintains high CPU utilization, a result of its superior memory management and scheduling strategies over Default. Due to high memory consumption, Default's performance is hindered. Lower CPU utilization levels are a manifestation of the SPE not being able to process events efficiently. Fig. 9b shows that Klink's average and tail CPU utilization is consistently higher than Default's with Klink able to reach, and sustain, higher tail CPU utilization earlier than Default. Klink's CPU utilization scales with throughput while Default fails to scale its CPU utilization with higher throughput levels.

6.2.4 Distributed Experiments. We evaluate the distributed performance of Klink by deploying it on up to 8 nodes (machines) and running YSB. We ran both HR and SBox in standalone mode. 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.

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. Figure 6e 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 when distributing 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.

6.2.5 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



(a) Memory **(b) CPU** **(c) Sensitivity** **(d) Overhead**

Figure 9: Memory utilization, CPU utilization, watermark sensitivity, and scheduler overhead

rate is measured by the fraction of times an SWM is ingested within Klink's estimated time range (Sec. 3.1). The experiment is conducted with multiple values of f (Sec. 3.2). We also provide a scheduler overhead analysis in these tests.

Figure 9c presents Klink's accuracy at estimating SWM ingestion time. The figure shows accuracy performance for two confidence values of f, 95 and 90 represented as Klink-95 and Klink-90 respectively, under the different network delay distributions. 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 degrades under the Zipf distribution with accuracy reaching only 62% while both Klink-95 and Klink-90 maintain higher SWM ingestion accuracy rate reaching 95% and 85%, respectively. The overall performance impact is as expected since the Zipf distribution injects higher unpredictability into network delay.

Figure 9d shows Klink's runtime overhead due to runtime data collection, SWM estimation, memory management, and orchestration with other operators. The overhead is measured as a percentage of throughput, i.e., the throughput loss that Klink would incur had the SPE runtime been allocated to processing events instead of running the scheduling algorithm. Fig. 9d confirms Klink's efficiency as it incurs negligible overhead when utilizing resources while delivering higher throughput (Fig. 6d). The figure shows a drop in overhead as the confidence values decrease, but the difference between the highest confidence value and the lowest is small. 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.

7 RELATED WORK

Scheduling policies such as First-Come-First-Served (FCFS) [10, 47, 48] and Shortest-Remaining-Processing-Time (SRPT) [38] were studied for streaming engines. SRPT led to rate-based policies [53] such as Highest Rate (HR) [48] that delivered better performance than Aurora's scheduler [3]. All of these algorithms optimize for output latency by exploiting particular properties of operators. In contrast to Klink, they are not optimized for queries employing window operators and are agnostic of stream progress. Queries can be prioritized by using user-defined values or by specifying an SLA [37, 43]. These policies expect as input differentiated heterogeneous queries and assume that different performance requirements are fixed for each query. Klink's algorithm can be complementarily used with such policies.

Effective scale-up [30, 36, 56] through prudent scheduling can deliver high performance per node to add to a system's compute capacity. StreamBox [36] uses watermarks to demarcate and parallelize substream processing. The substream with the earliest watermark is allocated more resources for effective parallelization. StreamBox is agnostic of load size, performance-sensitive to watermark frequencies, and does not scale with backpressure mechanism. Haren [41] is a scheduling framework that provides an abstract API to study multiple scheduling policies. Real-time stream scheduling [9, 46] relies on understanding the performance of the SPE for meeting execution deadlines [15, 29, 33] and then applying algorithms from real-time scheduling theory. This class of algorithms does not consider specifically windowed operators and how they affect output latency.

Strategies to deploy a query on a distributed cluster [6, 34, 42, 45] are orthogonal to the problem we address of scheduling deployed query operators at run time for execution on the available processing cores. The former type of scheduling happens at deployment time while the latter type happens at runtime. Klink is a solution to the latter type, integrating nicely into a system in which a query deployment scheduler works at deployment time.

8 CONCLUSION

We presented Klink, a state-of-the-art SPE scheduler optimized for executing queries employing window operators. Klink assesses the progress of each stream over multiple queries by analyzing watermarks and appropriately prioritizing execution to minimize output latency. Through integration into industrial-strength Apache Flink and extensive experimentation, we demonstrated that Klink outperforms key prior scheduling techniques. Our experiments show that Klink can deliver large output latency reductions of up to 50% over comparable techniques. These results demonstrate the effectiveness of Klink's design in delivering excellent performance.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Canada Foundation for Innovation (CFI) and Ontario Research Fund (ORF). L. Querzoni was partly funded by "FogAware" grant from Sapienza University of Rome, Prot.nr. PH120172B230B4D7.

REFERENCES

[1] 2020. Klink Codebase. https://github.com/klink-scheduler/klink.

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

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

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

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

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

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

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

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

[10] Michael A Bender, Soumen Chakrabarti, and Sambavi Muthukrishnan. 1998. Flow and Stretch Metrics for Scheduling Continuous Job Streams.. In *SODA*, Vol. 98. 270–279.

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

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

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

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

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

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

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

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

[19] Sergio Esteves, Gianmarco De Francisci Morales, Rodrigo Rodrigues, Marco Serafini, and Luís Veiga. 2020. Aion: Better Late than Never in Event-Time Streams. *arXiv preprint arXiv:2003.03604* (2020).

[20] Omar Farhat, Harsh Bindra, and Khuzaima Daudjee. 2020. Leaving Stragglers at the Window: Low-Latency Stream Sampling with Accuracy Guarantees. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems* (Montreal, Quebec, Canada) *(DEBS '20)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/3401025.3401732

[21] B. Gedik, S. Schneider, M. Hirzel, and K. Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1447–1463.

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

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

[24] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–34.

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

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

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

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

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

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

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

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

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

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

[35] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques da Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, and Guoqiang Jerry Chen. 2020. Turbine: Facebook's Service Management Platform for Stream Processing. In *2016 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 589–600.

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

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

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

[39] Snehal Nagmote and Pallavi Phadnis. 2019. Massive Scale Data Processing at Netflix using Flink. *Flink Forward Conference* (2019).

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

[41] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. 2019. 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)*. ACM, 19–30.

[42] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. 2015. R-Storm: Resource-Aware Scheduling in Storm. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada) *(Middleware '15)*. Association for Computing Machinery, New York, NY, USA, 149–161. https://doi.org/10.1145/2814576.2814808

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

[44] Nicolo Rivetti, Nikos Zacheilas, Avigdor Gal, and Vana Kalogeraki. 2018. Probabilistic Management of Late Arrival of Events. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS)*. ACM, 52–63.

[45] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. 2019. 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)*. ACM, 31–42.

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

[47] Mohamed A Sharaf, Panos K Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. 2006. Efficient scheduling of heterogeneous continuous queries. In *VLDB*. 511–522.

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

[49] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 263–274.

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

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

[52] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568.

[53] Tolga Urhan and Michael J Franklin. 2001. Dynamic pipeline scheduling for improving interactive query performance. In *PVLDB*, Vol. 1. ACM, 501–510.

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

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

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

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