SPEAr: Expediting Stream Processing with Accuracy Guarantees

Nikos R. Katsipoulakis
Amazon Web Services
Palo Alto, CA, USA
Email: nkatsip@amazon.com

Alexandros Labrinidis, Panos K. Chrysanthis
University of Pittsburgh
Pittsburgh, PA, USA
Email: {labrinid, panos}@cs.pitt.edu

***Abstract*—*Stream Processing Engines* (SPEs) are used for real-time and continuous processing with *stateful* operations. This type of processing poses numerous challenges due to its associated complexity, unpredictable input, and need for timely results. As a result, users tend to overprovision resources, and online scaling is required in order to overcome overloaded situations. Current attempts for expediting *stateful* processing are impractical, due to their inability to uphold the quality of results, maintain performance, and reduce memory requirements.**

In this paper, we present the SPEAr system, which can expedite processing of *stateful* operations automatically by trading accuracy for performance. SPEAr detects when it can accelerate processing by employing online sampling and accuracy estimation at no additional cost. We built SPEAr on top of Storm and our experiments indicate that it can reduce processing times by more than an order of magnitude, use more than an order of magnitude less memory, and offer accuracy guarantees in real-world benchmarks.

1. Introduction

Modern applications process large volumes of data in real-time. Examples of such applications include high-frequency trading, social network analysis [1], targeted advertising [2], click-stream analysis [3], [4], urban analytics, real-time data visualization [5], [6], complex event processing, to name a few. Their goal is to capture exact or approximate patterns to use them for online, time-critical, decision support. *Stream processing* is considered a prime candidate for those applications, due to its focus on continuous execution under a specified delay target, and optimized design for workloads with a single pass over data. As a result, *Stream Processing Engines* (SPEs) have rapidly evolved in the past decade [7], [8], [2], [1], [9], [3], [10], [11], with a number of SPE prototypes being adopted by industry (e.g., Storm, Spark, Flink, Kafka, etc.), and financial reports projecting that the streaming analytics market size will surpass 10 billion US dollars by 2021 [12], [13].

Real-time data ingestion combined with continuous processing of uncharted data streams makes timely processing a challenge for SPEs. Workloads are submitted to SPEs in the form of *continuous queries* (CQs), whose resources are provisioned at submission time and remain constant for the duration of the execution [14], [15], [8], [1], [16], [9], [17]. In order to achieve timely production of results, it is paramount to conduct processing in main memory (i.e., avoid spilling/fetching data to/from secondary storage) [3]. Often, the characteristics of input streams are unknown at every point in time for the lifetime of a CQ, and tend to fluctuate [18], [19]. To ensure Service Level Objectives (SLOs), users overprovision resources based on load estimations from historical data [20], [21], [22], [23], [24], [25]. On the one hand, over-provisioning resources leads to high operational costs. On the other hand, under-provisioning resources leads to missed delay targets, which jeopardizes the usefulness of the results.

In the past, various solutions have been proposed to maintain an SPE's performance at acceptable levels. *Load shedding* enables an SPE to drop tuples when the input load exceeds its processing capacity [20], [22], [21], [24], [26]. Despite the fact that load shedding is able to reduce load, it fails to deliver results of specified accuracy in rapidly fluctuating streams without manual intervention [22]. *Sketches* and *approximation* algorithms reduce the memory usage of a *stateful* operation in a CQ at the expense of accuracy [27], [28], [29], [30]. Such techniques can only be used in specific problems, such as frequency counting [29], [30], [31], membership inclusion [32], etc. Furthermore, *sketches* increase the processing overhead per data point significantly [32], [29], [33]. On top of this, *sketches* require manual tuning to preserve accuracy, which is error-prone, time-consuming, and impractical in real-time. *Elastic* SPEs are able to alter resources without disrupting execution [7], [34], [35], [36]. Nonetheless, the scaling process is demanding in time and resources, and suitable for situations of prolonged load increases [34], [7]. Finally, *incremental processing* solutions have been proposed for SPEs as a method to avoid processing and storing tuples more than once [37], [38], [39], [40]. However, incremental techniques are applicable to a limited number of *stateful* operations and their merits are diminished when holistic *stateful* operations are part of a CQ (e.g., percentiles). In conclusion, each of the existing techniques has its own shortcomings and cannot limit resource usage while delivering acceptable performance without manual intervention.

In this paper, we present the SPEAr system (from SPE

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00100

IEEE computer society

```
rides (time, route, fare)
cq = rides
     .time(x -> x.time)
     .slidingWindowOf(15, 5, MINUTES)
     .percentile(x -> x.fare, 0.95)
```

Figure 1: Example CQ in functional notation.

Accelerator), which represents a proactive approach to expedite processing and reduce memory usage by approximating results automatically. Furthermore, SPEAr's approximate results are within users' accuracy specifications. SPEAr detects opportunities for expedited processing in real-time, without making any assumptions on data patterns, and without the need for manual intervention. It does so by accumulating data information in an online fashion. At the time of processing, SPEAr estimates the accuracy of an approximate result built from an incremental data sample. In the event that the approximate result is within the user's accuracy specification, SPEAr produces the approximate result. Otherwise, SPEAr processes the whole window. To the extent of our knowledge, SPEAr is the only SPE that can (i) detect opportunities for *expediting* execution without manual intervention, and (ii) produce results within an accuracy specification at no additional runtime cost. Our experimental evaluation indicates that SPEAr can reduce processing time, memory usage, and deliver accurate results. In particular, our experiments on real-world datasets show that SPEAr reduces processing times and main memory usage by up to two orders of magnitude compared to a state-of-the-art SPE.

Contributions: The contributions of this work are as follows:

- A novel stream processing model to decrease execution time and main memory usage. Our model is compatible with current SPEs and overcomes the shortcomings of existing techniques.
- The design of the SPEAr system, which adopts our proposed model without incurring additional overheads. Its design consists of incremental sampling, accuracy estimation, and expedited execution.
- Experimental results on Apache Storm using real datasets highlight SPEAr's resource savings, performance gains, and delivery of accurate results.

Outline In Sec. 2 we present background information for our work. Then, in Sec. 3 we analyze the shortcomings of existing solutions, along with the details of our proposed model. In Sec. 4 we present SPEAr's execution details, and in Sec. 5 we discuss our experimental results. Finally, in Sec. 6 we present additional related work, and conclude our paper in Sec. 7.

2. Background on SPEs

In order to aid our presentation, we use the CQ presented in Fig. 1, whose syntax is similar to those offered by many SPEs (e.g., Kafka, Storm, Beam, Flink, Spark etc.). The CQ of Fig. 1 receives the *rides* stream, whose tuples' attributes are *time*, *route* id, and *fare* amount.

CQ definition: A CQ can have one or more input streams S_i, with $i = 1, \dots, N$, and is structured as a sequence of operations, each one receiving an input stream and producing an output stream. An operation is called *stateless* if it does not require previous tuples to apply its transformation (i.e., buffering of tuples is unnecessary). An operation is called *stateful*, if it requires previous tuples to apply its transformation (i.e., buffering of tuples is necessary) [18]. The CQ of Fig. 1 produces the 95 percentile of fares on sliding windows of 15 minutes. The *time* operation is *stateless*, as it annotates each tuple with its corresponding *time* value. The *percentile* operation is a *stateful* operation as it produces the 95 percentile on the set of tuples that constitute a window. A *stateful* operation $g(S_i^w) \rightarrow R_w$ receives a window of tuples S_i^w as input, which is a subset of S_i generated by a windowing function $\mathcal{W} : S_i \rightarrow S_i^w$, with $w = 1, \dots \infty$. For each input window S_i^w, g produces a window result R_w. $\mathcal{W}$ can produce windows based on the number (count-based) or the timestamp (time-based) of tuples. In addition, $\mathcal{W}$ can assign each tuple to one or more windows. In the former case, $\mathcal{W}$ produces *tumbling* (non-overlapping) windows; whereas in the latter case it produces *sliding* (overlapping) windows. *Tumbling* windows require a *range*, whereas *sliding* windows require both a *range* and a *slide*. In the CQ of Fig. 1, *percentile* is defined over a sliding window with a *range* of 15 minutes, and a *slide* of five minutes. In this work, we focus on the performance of *stateful* operators, as they present numerous challenges.

CQ execution: State-of-the-art SPEs are designed to operate either on clusters of multi-core servers or on the Cloud. An SPE turns a CQ into a distributed execution plan, which is modeled by a directed acyclic graph (DAG). Each operation of a CQ has its own execution stage in the topologically-sorted DAG. An SPE allocates resources for each stage, which are (a) the number of $\mathcal{V}$ workers (i.e., number of CPU processes/threads), and (b) the available memory b for each worker [1], [11], [8]. b is pivotal in each worker's performance, since it controls the amount of data that can be processed without spilling to secondary storage [20], [41]. Often, secondary storage $\mathcal{S}$ is independent of workers' contexts, is globally accessible (e.g., S3) [4], and offers two methods: (i) store data, $store(\tau)$, and (ii) retrieve data identified by τ_W, $get(\tau_W)$. The number of workers ($\mathcal{V}$) and the memory budget (b) are defined when a CQ is submitted. The propagation of tuples between execution stages materializes using partitioning techniques [18], [19]. A possible DAG for the example CQ is illustrated in Fig. 2. The left-most stage represents the input and is materialized by a single worker, which distributes tuples to the workers of the *time* stage. The output of the *time* is sent to the *window* stage's workers, which buffer tuples until a window S_i^w is complete. At window completion, each S_i^w is pushed to the workers of the *percentile* stage, which process their input and produce a window result R_w.

SPE stateful processing: S_i^w's tuples are processed when there is a guarantee that S_i^w is complete. To this end, SPEs are equipped with two mechanisms for managing S_i^w's lifecycles: *trigger* that monitors when each window

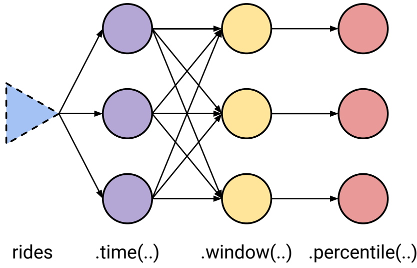


Figure 2: The execution DAG for the CQ of Fig. 1.

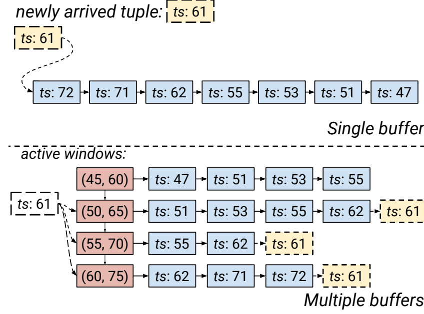


Figure 3: Tuple arrival.

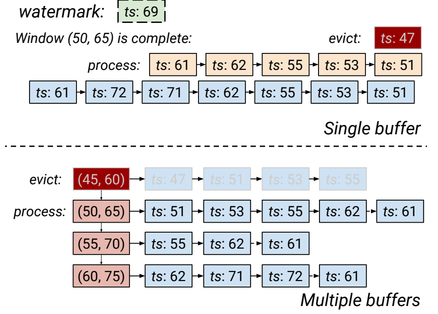


Figure 4: Watermark arrival.

is complete; and *evict* that identifies and discards fully processed tuples. Those two mechanisms are essential for strict delivery semantics, out-of-order tuples' processing, and fault tolerance [9], [42]. Current SPEs support *trigger* with the use of *watermarks*, which are control-tuples carrying a timestamp (τ_W). Those are sent by SPE components periodically, and the receipt of a *watermark* τ_W indicates that all tuples timestamped $\tau \leq \tau_W$ have been observed [2], [9], [10], [42]. In most SPEs, *watermarks* are produced by data source operators, and are propagated by downstream workers accordingly. At tuple arrival, a *stateful* operation worker stores tuples in its memory buffer of size b until a window is complete. This is a requirement in order to guarantee delivery semantics, and fault tolerance [2], [42]. If at any point prior to receipt of a *watermark*, all of a worker's memory budget b is used, then the worker spills consequent tuples to $\mathcal{S}$. At *watermark* arrival, a *stateful* operation worker identifies completed windows, and stages them for processing. Next, we present the design details followed by each worker at *tuple* and *watermark* arrival by current SPEs. For our presentation we will assume the window semantics of the example CQ of Fig. 1.

Tuple arrival: Fig. 3 illustrates the two possible designs for when a tuple arrives among existing SPEs. The *single buffer* design dictates that all tuples are stored in a single buffer based on their order of appearance. This design is adopted by Storm [1] and its advantage is that each tuple is stored only once. In contrast, the *multiple buffers* design dictates that a copy of each tuple is stored in a buffer for each of the windows that it participates in. Flink uses this design, whose merit is that each window is ready for processing at *watermark* arrival [42]. However, the use of multiple buffers requires additional memory per tuple, which leads to low storage utilization. In Fig. 3, the new tuple with timestamp 61 participates in windows: $(50, 65)$, $(55, 70)$, and $(60, 75)$ (according to the CQ of Fig. 1). With the *single buffer* design, the tuple is appended to the buffer and the worker thread becomes dormant until the arrival of the next tuple. In contrast, with the *multiple buffers* design, a copy of the tuple is stored on the buffer for each window.

Watermark arrival: A worker has to prepare the window ending on or before τ_W when the corresponding watermark arrives. In the event that the worker spilled tuples to $\mathcal{S}$, then it has to retrieve them. With the *single buffer* design, the worker scans existing tuples and gathers all the tuples belonging to the window ending on τ_W. At the same time, it evicts expired tuples. Turning to the *multiple buffers* design, the worker picks the buffer corresponding to the window ending in τ_W, and stages it for processing. Fig. 4 presents the window preparation process when the watermark with timestamp 69 arrives and triggers the processing of window $(50, 65)$. With the *single buffer* design, a worker scans its buffer to (i) collect window's $(50, 65)$ tuples, and (ii) evict expired tuples. Then, it sends tuples for processing. Some SPEs that use this design, mark a window's *newly* arrived tuples to offer incremental processing [43], [44]. In the example of Fig. 4, tuples timestamped 61 and 62 are marked as "*new*" since they are processed for the first time. Conversely, with the *multiple buffers* design, the worker picks the window's $(50, 65)$ buffer, and stages it for processing. The full scan of the buffer is avoided with this design at the expense of additional memory space.

3. Our Approach

Incoming stream's properties are unknown and tend to fluctuate overtime [7]. Hence, the exact resources needed for each window cannot be precisely predicted, which leads to situations where $\mathcal{V}$ workers are not capable of fitting all tuples in b. In this case, a worker has to spill data to $\mathcal{S}$ and processing speed drops. In order to avoid this, users either overprovision resources, or scale-out processing [7], [34], [45], [46], [47], which lead to increased operational costs and temporal performance degradation. On top of those, there are no guarantees that data will not end up in $\mathcal{S}$, or that an SPE will avoid constant re-configuration. To this end, SPE processing needs to be expedited in an opportunistic manner. *When possible, an SPE should examine if only part of a window can be processed to produce results within user-defined accuracy levels.* This thesis is conceptualized into the following requirements:

R1 **Maintain quality:** Each window result R_w has to be produced within SLOs and abide to a user-specified level of accuracy.

Algorithm 1 Approximate SPE: Tuple Arrival

```
1: procedure TUPLERECEIVE(τ)
2:     if ¬b.full() then
3:         b.put(τ)
4:     else
5:         b.replace(τ)
6:     S.store(τ)
```

Algorithm 2 Approximate SPE: Watermark Arrival

```
1: procedure WATERMARKRECEIVE(τ_W)
2:     (R̂_w, ε_w) ← g(b)
3:     if ε_w ≤ ε then
4:         return R̂_w
5:     return g(S.get(τ_W))
```

R2 **Process fraction of the data:** When possible, results need to be produced by processing a fraction of S_i^w, since a window's size is the dominating factor in terms of runtime and storage cost.

R3 **React to changes:** An SPE needs to be able to identify opportunities for expedited processing without manual intervention or by relying on historical data patterns.

R4 **Being generally applicable:** Processing a fraction of S_i^w while preserving accuracy needs to be applicable to a plethora of *stateful* operations.

Existing techniques are unable to deliver all of the above requirements. Below, we analyze the shortcomings of the most widely-used alternatives.

Load shedding enables an SPE to drop tuples when load exceeds existing processing capacity [20], [22], [24]. Despite the fact that load shedding can be effective under load, it fails to deliver accuracy guarantees without manual intervention, especially in rapidly fluctuating streams [22], or without relying on historical data (i.e., fails **R1**) [21]. Also, existing techniques, do not react to changes in data (i.e., fail **R3**) [26].

Sketches [48], [28], [29], [33], [30], [6] decrease the memory footprint of a *stateful* operation at the cost of additional processing (due to the use of complex hash functions). For instance, a CountMin sketch requires each tuple to be passed to as many hash functions as is the size of the sketch [29]. Furthermore, to reconstruct the result of the sketch, each distinct group needs to be maintained in memory (a CountMin retains only the estimated frequencies and not the groups themselves). As a result, processing time per tuple is increased and the storage benefit is diminished (i.e., fail **R2**). In addition, *sketches* require fine tuning to adapt to fluctuating streams (i.e., fail **R3**), and can only be used in specific problems (i.e., fail **R4**).

Incremental stream processing offers techniques that avoid processing tuples more than once [49], [38], [39], [40]. Even though those techniques reduce the update cost of a window's result, they can only be applied in associative *stateful* operations (i.e., fail **R4**). In addition, *incremental* processing techniques have to accommodate all tuples in main memory and do not adapt to changes in data (i.e., fail **R2** and **R3**).

Elastic SPE processing offers techniques for adjusting resources in an online fashion [34], [50], [35], which are effective for long-lasting input surges. However, such techniques incur significant overhead on temporal increases in input and fail to produce timely results during short-term spikes (i.e., fail **R1**) [34]. As a result, those techniques are slow to react and cannot keep the operational cost low (i.e., fail **R3**) [7], [34].

3.1. Approximate Stream Query Processing

Approximate Query Processing (AQP) offers techniques for processing a fraction of the data to produce a result with a well-defined accuracy [51], [52], [53], [54]. The fraction of data processed is defined as a *budget*, in terms of either IO operations or response time. As a result, an AQP system takes less time to produce a result by approximating its exact value. Due to AQP's useful functionality, it is offered by many commercial DBMSs (e.g., Oracle 12c, SQL Server, Redshift etc.).

Inspired by the performance benefits of AQP and *incremental processing*, we propose an *approximate stream processing model*, which delivers all of the requirements **R1**-**R4** for faster execution. Due to the unavailability of data (or metadata) *prior to processing* in SPEs, online sampling is the only option for our model, which requires two user-defined parameters: (a) *accuracy* ϵ, and (b) memory *budget* b. The first controls the accuracy of a window result R_w: the lower the error, the higher the accuracy of an approximate window result $\hat{R}_w$. The memory *budget* b controls the processing performance to produce R_w, and ties well with the memory budget b of each SPE worker. Our approximate stream processing model dictates different behavior at *tuple* and *watermark* arrival, which are presented in Alg. 1 and 2.

The core idea of our model is that b is used to accommodate either a *simple random sample* of S_i^w's tuples or S_i^w's metadata, in case S_i^w cannot fit in main memory b. If the memory budget b has not been depleted, then a newly-arrived tuple (τ) is stored in it; otherwise, a stochastic process is used to replace one of the tuples in b (Alg. 1). In any case, τ is stored in $\mathcal{S}$ as is common practice [4]. Alg. 2 indicates that when a watermark τ_W is received, an estimate of the accuracy (ϵ_w) and an approximate result ($\hat{R}_w$) are calculated solely from b's contents. If ϵ_w is within the user-defined accuracy ϵ, then $\hat{R}_w$ is produced as the window's result; otherwise, the whole window is fetched from $\mathcal{S}$ and the exact result R_w is produced. This model abides with the aforementioned requirements, since it (i) delivers results within the user-defined accuracy ϵ (**R1**); (ii) produces a result based on a fraction of the window's data (**R2**); (iii) reacts to data characteristics (**R3**); and (iv) supports a plethora of relational operations (**R4**). SPEAr adopts the proposed model and is designed to not incur overhead when $\epsilon_w > \epsilon$.

4. SPEAr Overview

We built SPEAr on top of Apache Storm v1.2 and adopted the latter's execution engine [1]. A SPEAr cluster consists of a coordinating process, named *Nimbus*. Processing occurs in a distributed set of workers, which are able to accommodate a constant number of worker threads. *Nimbus* is responsible for scheduling operations to available workers, monitoring the execution of CQs, and does not participate in the processing critical path. Worker threads handle execution, which involves reliable tuple delivery, state management, and application of a CQ's operations. We built SPEAr on top of Storm because it follows the *single buffer* design, which results in minimal memory usage per worker.

SPEAr's implementation required changes in Storm's core classes: *TopologyBuilder*, *WindowManager*, and *BaseWindowedBolt*. In terms of CQ definition, we created *SpearTopologyBuilder* which exposes Fig. 5's API. This class enables users to define *stateful* operations with a b and an (ϵ,α) pair. Turning to runtime, we extended each operator's *WindowManager* to operate based on our proposed model. In turn, each manager incrementally gathers information on a window's tuples at tuple arrival (explained in Sec. 4.1), and performs an accuracy estimation at window completion (explained in Sec. 4.2). As far as *BaseWindowedBolt* is concerned, we extended it to a *SpearBolt* that disassociates execution into production and delivery of a result. Based on the estimated accuracy, the appropriate method is called as we explain below.

SPEAr follows our proposed model (Sec. 3) and allows a user to submit CQs with an accuracy (ϵ) and a memory budget (b) specification for each *stateful* operation. The CQ of Fig. 1 is shown in Fig. 5 for SPEAr: each worker-thread of the percentile operation carries an ϵ and a b. b is configured using the *budget()* method and is set to 1MB. This configures each worker-thread to store as many *fare* values as can fit in a 1MB buffer. If at any point a worker-thread requires additional memory to accommodate input, it will be spilled to $\mathcal{S}$. Future versions of SPEAr will be able to accommodate dynamic methods for online budget estimation. ϵ is defined with the *error()* method and indicates that a result can not deviate more than 10% from the exact value, for 95% of the windows. The second argument of *error()* is called *confidence* (α) and controls the confidence intervals for approximate results [51], [52], [53], [54].

SPEAr supports *mean-like stateful* operations, including the most popular aggregate functions (e.g., count, sum, average, quantile, variance, stddev) [5]. Those operations are supported both in *scalar* and *grouped* formats. *Scalar* operations produce a single result per window, whereas *grouped* operations produce a set of results per window. The CQ of Fig. 5 features a *scalar* operation. If the CQ carried a grouping clause, then a percentile value would be produced per distinct group. Furthermore, SPEAr offers an API for defining custom approximate *stateful* operations. A user has to define an accuracy-estimation function, and SPEAr follows the execution workflow dictated by our proposed model. At the moment, relational joins can be implemented using the API for custom *stateful* operations, because a widely-accepted metric for measuring join accuracy does not exist [55], [56], [21], [57], [58]. All of the aforementioned functionality is applicable on time-/count-based *tumbling*/*sliding* windows.

```
cq = rides
     .time(x -> x.time)
     .slidingWindowOf(15, 5, MINUTES)
     .percentile(x -> x.fare, 0.95)
     .budget(1MB)
     .error(10%, 95%)
```

Figure 5: CQ of Fig. 1 for SPEAr.

Depending on the type of the operation (i.e., scalar or grouped), SPEAr's execution steps differ in terms of *sampling*, statistics accumulation, *accuracy estimation*, and *processing*. The novelty of SPEAr emanates from its ability to offer the aforementioned functionality without (i) incurring overhead in the form of additional scans on S_i^w, (ii) without erring on the accuracy specification, and (iii) offering better performance compared to exact processing. Those are achieved by designing SPEAr in order to: sample tuples and accumulate windows' statistics within the *budget*, and estimate accuracy and results without introducing additional scans on S_i^w (**Tuple arrival**). In the event that a $\hat{R}_w$'s $\epsilon_w > \epsilon$, SPEAr's performance is identical to normal execution (**Watermark arrival**). In the following sections we provide SPEAr's execution steps for *scalar* and *grouped* operations.

4.1. Tuple Arrival

Each SPEAr worker uses its b to accommodate information used for $\hat{R}_w$ incrementally. As discussed in [59], approximate results rely on (i) simple random samples (*s.r.s*), and (ii) statistical information on the data distribution. A naive approach to stream approximation would introduce an additional scan of S_i^w at window completion. To avoid this, we adopted the proposed model's tuple arrival algorithm (Alg. 1) to current SPEs' workflows. To this end, SPEAr either accumulates statistical information or updates an incremental sample at tuple arrival. The sample and the statistical data are stored in b. In detail, when a worker receives a tuple, it extracts its timestamp (or unique number in the case of count-based windows). The window(s) that the tuple corresponds to is/are determined using the timestamp. The steps taken by SPEAr differ between *scalar* and *grouped* operations.

Scalar: For each S_i^w, an *s.r.s.* is required to avoid low accuracy due to temporal locality in input [59]. To ensure that an S_i^w's sample is an *s.r.s.*, SPEAr employs *reservoir* sampling, which delivers a sample with size $\leq b$. Moreover, SPEAr uses part of b to maintain S_i^w's statistical estimates incrementally. Those are essential for accuracy estimation, which takes place at *watermark* arrival. For *scalar* operations the aggregated values' *variance* is needed. The incre-

mental sampling and *variance* calculation occur when b's methods *put/replace* are called. Those calls pose negligible overhead compared to IO occurring at tuple arrival. For the CQ of Fig. 5, the *reservoir* sample of each S_i^w carries up to $|\lfloor 10^6 f^{-1} \rfloor - 2|$ values (we assume that each *fare* requires f bytes). In the previous formula, the total number of values stored in b is reduced by 2 because SPEAr maintains *fare* values' *variance* and the size of S_i^w. On non-holistic scalar operations (i.e., incremental), SPEAr incrementally updates R_w at tuple arrival (akin to incremental processing techniques). SPEAr uses b's contents only when an anomaly is detected in tuple delivery (e.g., failure, late tuples, out-of-order delivery). For instance, for the mean *fare* calculation, SPEAr maintains an incremental sum per window. At window completion, SPEAr uses the sum to produce R_w. In the event of a delivery anomaly, SPEAr estimates the accuracy of $\hat{R}_w$ (i.e., ϵ_w). If $\epsilon_w \leq \epsilon$, then SPEAr does not have to scan b to produce the mean. Currently, SPEAr supports only percentile in terms of *holistic* operations by modifying the algorithm presented in [48]. At tuple arrival, b accommodates a *reservoir* sample, and at *watermark* arrival, SPEAr is able to determine if the sample is sufficient to produce $\hat{R}_w$.

Grouped: Previous work on AQP for *grouped* operations advises that $\hat{R}_w$ needs to include every distinct group of S_i^w [59], [51], [54]. For example, if the CQ of Fig. 1 contained a group-by operation on the *route* field, then a percentile value needs to be produced for each distinct route. In this scenario, R_w is a set of (r, v) pairs, where r is a *route* and v the percentile result. $\hat{R}_w$ needs to be a set with the same cardinality (i.e., $|R_w| = |\hat{R}_w|$) and contain the same *route* values (i.e., $\hat{R}_w = [(r, v) | \forall r \in R_w]$). To achieve this, SPEAr uses *stratified* sampling, which dictates that each group's sample size needs to be proportional to its original size [59]. *Stratified* sampling requires two passes over S_i^w, when the number of distinct groups is unknown: (i) one pass to establish each group's frequency; and (ii) one pass to construct the sample. SPEAr's novelty lies in decoupling the two passes in order to avoid scanning S_i^w twice at window completion.

Due to the lack of prior information for S_i^w, SPEAr can solely accumulate each group's frequency while the window is active. Therefore, b is used to track each group's frequency in S_i^w. This constitutes a fundamental difference between SPEAr and AQP systems, which rely on offline sampling that (in turn) enables them to build samples with one scan. SPEAr does not have this privilege and building a *stratified* sample requires an additional scan of S_i^w. As a result, SPEAr constructs the sample only after a window is complete. To achieve this, SPEAr maintains each group's frequency and variance for the value that is used in the *stateful* operation. Those metrics are needed by the *basic congress* sampling technique used by SPEAr [59]. b can accommodate up to $\lfloor b(r+4+f)^{-1} \rfloor$ distinct groups' information, where r is the size of a group's identifier in bytes, 4 is the number of bytes for the frequency count, and f is the bytes needed to accommodate the variance. If b can not accommodate enough values, then SPEAr reverts back to normal processing.

If Fig. 5's CQ required the 95 percentile *fare* value for each *route*, then $\hat{R}_w$ would produce a record for each distinct *route* in S_i^w. In this case, b is used to accommodate each group's frequency in S_i^w. This information is used at *watermark* arrival to derive the size of the sample needed for each group. By modifying *stratified* sampling to be applied this way, SPEAr does not introduce additional scans of S_i^w compared to the *single buffer* design. Finally, when the number of groups is defined by the user at CQ submission, then SPEAr is able to create a *stratified* sample at tuple arrival, by dividing b equally among the distinct groups of S_i^w. In this case, no scans of S_i^w are needed and SPEAr produces $\hat{R}_w$ at a minimal cost.

4.2. Watermark Arrival

At watermark arrival, a SPEAr worker produces an accuracy estimate ϵ_w, and an approximate result $\hat{R}_w$. If $\epsilon_w \leq \epsilon$, then $\hat{R}_w$ is pushed out as S_i^w's result; otherwise, the worker has to process the whole S_i^w and produce an exact result R_w. The latter case entails that tuples might be accessed from secondary storage $\mathcal{S}$. SPEAr's challenge is to make an accurate estimation of ϵ_w, and if $\epsilon_w > \epsilon$, then not introduce significant overhead to the execution critical path.

Only a subset of S_i^w's tuples are used to produce $\hat{R}_w$. Therefore, it is essential to measure the deviation of $\hat{R}_w$ from R_w, which is represented as $\epsilon_w : R_w, \hat{R}_w \rightarrow \mathbb{R}$. ϵ_w differs among *stateful* operations, especially between *distributive/algebraic*, and *holistic* operations [60]. SPEAr carries different ϵ_w functions depending on the *stateful* operation, and allows users to define their own metric. Out-of-the-box, SPEAr uses the relative error for *algebraic* and *distributive stateful* operations [51], [54], [53]. For *grouped stateful* operations, ϵ_w has to aggregate all groups' estimated errors into a single value. To this end, SPEAr calculates the error for each group e_w^g, and then combine all e_w^g values by using one of the error functions presented in [59] (Equations in Def. 3.1). SPEAr uses the same accuracy metric for quantile approximations presented in [48], which is used to define a ϕ-quantile approximate value.

In order to estimate ϵ_w, SPEAr assumes that accuracy estimates are normally distributed. In essence, a window S_i^w of size N is represented by a sample T_w, where $|T_w| = n < N$. Given an operation $g()$, $\hat{g}(T_w)$ is an *estimator* of $g(S_i^w)$. SPEAr's goal is to estimate the expected error of $\hat{g}(T_w)$. For instance, if g is the arithmetic mean of a group of values v, given a user-defined budget $b = n$ (where n is the sample size), then the mean estimate is $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} v_i$, where $v_i \in T_w$. The exact value for the window's mean is $\mu = \frac{1}{N}\sum_{i=1}^{N} v_i$, where $v_i \in S_i^w$. The confidence intervals can be established using the sample's variance s [61], as follows:

$$\bar{Y}_{low} = \bar{y} - \frac{ts}{\sqrt{n}}\sqrt{1 - \frac{n}{N}}, \quad \bar{Y}_{high} = \bar{y} + \frac{ts}{\sqrt{n}}\sqrt{1 - \frac{n}{N}}$$

$\bar{y}$ is the sample mean, t is the value of the normal deviate corresponding to the desired confidence probability (e.g.,

1.96 for $\alpha = 95\%$ and 2.58 for $\alpha = 99\%$), n is the sample size (or budget b), and s is the samples' standard deviation. SPEAr follows the same confidence interval estimation technique to establish the accuracy of $\hat{R}_w$. SPEAr treats the confidence interval of $\hat{R}_w$ as a relative distance to R_w. If the relative distance is within the accuracy specified by the user, then SPEAr returns $\hat{R}_w$ as output; otherwise, the whole window is processed. In order for SPEAr's confidence interval mechanism to provide accurate estimations, b can not be small. This is a limitation imposed by Normal approximation techniques (as a direct implication of the *Central Limit Theorem*) [61]. As a result, the confidence interval will be imprecise with a very small sample on a skewed distribution. However, modern hardware is able to accommodate samples with a size of tens of thousands elements, which is ample for most distribution types. Below, we present the steps taken by a SPEAr worker at *watermark* arrival, for *scalar* and *grouped* operations.

Scalar: If the *scalar* operation is *non-holistic*, a worker has incrementally prepared R_w (this is the case for associative and (non-) invertible operations) [40] and pushes it downstream. Otherwise, each SPEAr worker utilizes the *s.r.s.* of size b. When a *watermark* arrives, the worker performs a constant number of operations (less than ten) to finalize $\hat{R}_w$. Next, it estimates the confidence interval and makes a decision on whether to use the incrementally processed $\hat{R}_w$. In case $\epsilon_w \leq \epsilon$, then the worker produces $\hat{R}_w$; otherwise, it scans S_i^w to produce R_w. The CQ of Fig. 5 contains the *holistic* operation of percentiles. Therefore, at *watermark* arrival, the confidence interval needs to be established. For the case of quantiles, the work of [48] indicates that accuracy is estimated by comparing the sample's size with S_i^w's size. This is done by comparing the allocated budget b for a window with the expected budget documented in [48]. If the allocated budget b is lower than the one required for the approximate quantile algorithm to provide an answer within the expected accuracy, then SPEAr processes the whole window. Otherwise, it scans b and produces the estimated quantile. Overall, SPEAr does not introduce additional S_i^w scans for *scalar* operations: When $\hat{R}_w$ is acceptable (i.e., $\epsilon_w \leq \epsilon$), then SPEAr has a worst-case runtime cost of $O(|b|)$ (this is the case for *holistic* operations); when $\epsilon_w > \epsilon$, then the worst-case runtime cost of becomes $O(|S_i^w| + |b|) \cong O(|S_i^w|)$, since $|b| \ll |S_i^w|$. This is the same complexity as the normal processing of an SPE.

Grouped: Turning to *grouped* operations, a SPEAr worker establishes each group's sample size at *watermark* arrival. This process requires groups' frequencies, which have been incrementally established before the *watermark*'s arrival. In addition, SPEAr calculates each group's accuracy, and aggregates them using the $L1$ accuracy metric [59]. If the estimated $L1$ accuracy is within the specified level (i.e., $\epsilon_w \leq \epsilon$), then SPEAr produces $\hat{R}_w$; otherwise, it has to process all of S_i^w. In any case, S_i^w is either scanned once to form the stratified sample and produce an approximate result for each group; or prepare S_i^w for processing.

TABLE 1: Datasets and Queries Used

	Total Tuples	Win. Size	Win. Slide	Avg. Win. Size
DEBS	56M	30 min.	15 min.	≈10K
GCM	24M	60 min.	30 min.	320K
DEC	4M	45 sec.	15 sec	47K

Similar to *scalar* operations, SPEAr does not introduce additional scans on the window S_i^w. In case $\epsilon_w \leq \epsilon$, a SPEAr worker performs one scan of S_i^w. This scan is already required by the *single buffer* design to evict expired tuples. In case the estimated accuracy $\epsilon_w > \epsilon$, then a SPEAr worker has to perform two scans of S_i^w: one for tuple eviction and one for the actual processing. This is on par with normal processing. To this end, SPEAr's overhead is $O(\|S_i^w\|)$, where $\|S_i^w\|$ indicates the number of distinct groups in S_i^w. This runtime cost is bounded by b, since SPEAr does not allow expediting a window if all the groups' information can not be accommodated in b. Therefore, the worst case overhead is $O(b) \ll O(|S_i^w|)$. Our experiments indicate that this overhead is minimal when examined in the context of continuous execution.

5. Experimental Evaluation

Our goal is to measure SPEAr's ability to improve performance, reduce main memory usage, detect opportunities for faster processing, and preserve accuracy. We conduct the experimental evaluation on an Amazon EC2 cluster with 9 r4.xlarge nodes. Each node runs on Ubuntu Linux 16.04, OpenJDK Java v1.8, and has 4 virtual CPUs of an Intel Xeon E5-2686 v4 with 32GBs of RAM. We set one node as the master, running a single-instance Zookeeper v3.4.10 server and a single *Nimbus* process. We configure each of the remaining nodes to accommodate up to four SPEAr workers. In all experiments, we enable Storm's acknowledgment and back-pressure mechanisms to guarantee in-order delivery of tuples.

For all CQs we set a single source operator that reads data sequentially from a memory-mapped file. In turn, each SPEAr worker applies a *stateful* operation whenever a window is complete. All workers forward their results to a single worker, who persists output to secondary storage for validation. All of the performance and accuracy numbers we report are the arithmetic mean of seven runs, without the maximum and the minimum reported values. In order to measure processing times with high precision, we use Storm's metrics API, which provides periodic reporting of runtime telemetry for each worker thread. Finally, for the experiments that require *sketch* techniques, we use Stream-Lib's popular implementation for CountMin to compare SPEAr with the state-of-the-art sketching technique.

In our evaluation we employ three real datasets. For each one, we form a CQ with a single *stateful* operation on a predefined event-time, time-based sliding window. Table 1 summarizes information on each dataset. Two of the datasets feature *grouped* operations (DEBS and GCM) and one features two *scalar* operations (DEC).

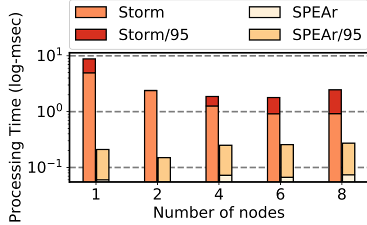


Figure 6: Processing time on Median CQ for DEC.

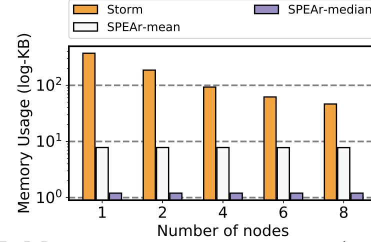


Figure 7: Mean memory usage per worker on DEC.

ACM DEBS 2015 Challenge dataset (DEBS): This dataset contains rides' data from a Taxi Company. For DEBS we use one of the challenge's *grouped* operations, which features a 30 minute sliding window, with a 15 minute slide, for each route's average fare amount [62].
Google Cluster Monitoring dataset (GCM): For this dataset we use part of the *task-events* stream and perform Query 1 from [17], which produces each scheduling class's average CPU time. In order to examine SPEAr's performance on larger windows, we set the window size to 60 minutes and the slide to 30 minutes. Additionally, we investigate the sensitivity of SPEAr on windows' sizes.
DEC Network Monitoring dataset (DEC): For this dataset we use time-based sliding windows of 45 seconds with a slide of 15 seconds [22]. For DEC, we experiment with two *scalar* operations: (i) the average, and (ii) the median TCP packet size.

Unless specified otherwise, we set SPEAr's budget b to a value causes it to accelerate processing in the majority of windows. In order to identify the proper b value for each dataset, we analyzed their data characteristics offline, and then hard-code those values in the CQs. For DEC, GCM, and DEBS, we set b to 1,000 for the mean operation (150 for median), 4,000, and 2,000 tuples, respectively. Those b values amount to 2.1% (0.3% for the median), 5%, and 20% of the average window size for DEC, GCM, and DEBS (Table 1). The reason we set b on DEBS to be a large percentage of the window size is because DEBS's data is sparse and most distinct routes appear once or twice per window. Our analysis indicates that on average, for a window size of $\approx$ 10K tuples, 5K distinct routes appear in it. As a result, SPEAr needs to be able to accommodate at least a tuple from each group to build a stratified sample. We discovered that setting $b = 2$K tuples per worker allows the acceleration of most windows in the dataset (at least 98.2% of the windows). For all CQs we set the relative error to 10% and the confidence to 95% unless specified otherwise.

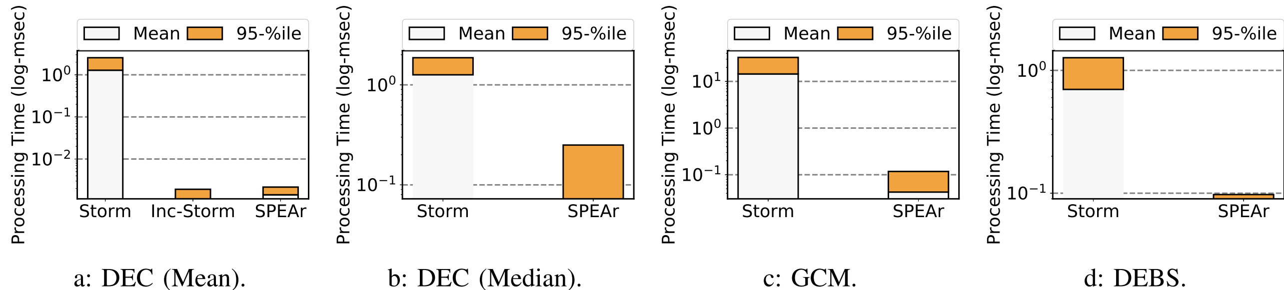
5.1. Scalability (Figures 6-7)

First, we quantified SPEAr's scalability in terms of processing time and amount of memory used. To this end, we measured Storm's and SPEAr's processing time and average memory consumption per worker on five different levels of parallelism for the median operations of DEC. We chose DEC's median for our scalability experiment because it requires maintaining and sorting each window without the interference of groups, and cannot be incrementally processed. Fig. 6 illustrates the window processing time (both mean and 95-percentile) when one, two, four, six, and eight worker nodes are used (the average processing time among all workers is presented). SPEAr is up to *two orders of magnitude* faster compared to Storm in terms of the average window processing time; and up to *one order of magnitude* faster compared to Storm in terms of the 95 percentile window processing time. This happens because SPEAr has to process $b = 150$ tuples per window (which is the budget size) and achieves an error $< 10\%$ for at least 99% of the windows. On the other hand, Storm has to process many more tuples per window (the average window size for DEC is 47K tuples). In addition, we measured the average size of memory used for producing the result by each worker. Fig. 7 illustrates the memory used by Storm and SPEAr on both the average and the median TCP packet size CQs. It is apparent that SPEAr uses a constant amount of memory for both operations, which is equal to the budget set for each CQ, since all the windows are accelerated. SPEAr uses up to *two orders of magnitude* less memory per worker for the median TCP packet size CQ.
Take-away: SPEAr achieves significantly lower processing times compared to Storm (up to two orders of magnitude) with only a fraction of the cluster nodes. Also, SPEAr requires much less memory per worker (up to two orders of magnitude) to produce a result within the specified accuracy (i.e., SPEAr will avoid spilling data on secondary storage in the event that the working set cannot fit in main memory).

5.2. Performance (Figure 8 and Table 2)

For the next experiment, we configure our cluster to use up to four worker threads per CQ, each one running on a single node in isolation. Our goal is to measure the mean and 95-percentile window processing time for all datasets. On top of that, for GCM and DEBS we compare SPEAr with an implementation of the CQ that uses a CountMin sketch. After analyzing each dataset offline, we allocate enough space for the CountMin sketch to achieve a confidence of 95% and an error of up to 10% on all windows (equivalent to SPEAr's accuracy). Fig. 8 illustrates the mean and the 95-percentile window processing times of Storm and SPEAr.

Starting from the mean CQ for DEC, we compared SPEAr not only with Storm, but with an optimized version for processing the mean using incremental techniques. In detail, we modified Storm to incrementally update a running count at tuple arrival. When a watermark arrives, it only



a: DEC (Mean). b: DEC (Median). c: GCM. d: DEBS.

Figure 8: Average and 95-percentile window processing time.

performs a division to produce the mean per window. This is the optimal way for a mean, and we represent it as *Inc-Storm* on Fig. 8a. Both *Inc-Storm* and SPEAr are three orders of magnitude faster than Storm. Even though we do not suggest using SPEAr for an incremental operation such as the scalar mean, we can see that SPEAr is only 11% slower compared to the optimal method for producing the mean. For the median TCP packet size CQ, SPEAr reduces processing time by *almost an order of magnitude*. Next, on GCM, SPEAr reduces window processing time by more than an *order of magnitude* compared to Storm for both the mean and the 95-percentile case respectively (Fig. 8c). This is due to the fact that GCM presents a small number of distinct groups, whose results can be approximated with only a small portion of the window's tuples. In addition, the performance gap on GCM is wider because the number of groups are known for this CQ. As a result, SPEAr is able to perform sampling at tuple arrival. Finally, Fig. 8d illustrates the processing time of Storm and SPEAr on DEBS. For this experiment, we set SPEAr's budget b to 2,000 tuples, which amounts for 20% of the average window size. As mentioned earlier, DEBS is a sparse dataset since a large portion of distinct groups appear on a window less than three times. As a result, b needs to be set at a high enough value to accommodate a stratified sample representing all distinct groups. Our offline analysis of DEBS concluded that a 2,000 tuple budget allows SPEAr to reduce window processing time by 7.77 and 13 times for the mean and the 95-percentile case. With this budget setting, SPEAr is able to accelerate at least 98% of the windows on each worker, and avoid processing at least 25% of the total number of tuples across all windows. To this end, the performance gap compared to Storm is significant.

As far as the CountMin sketch is concerned, we compared SPEAr against a CQ that uses CountMin to produce the results of the grouped aggregate operations of GCM and DEBS. To this end, we used a CountMin sketch for counting the sum of values and the frequency of appearance of each distinct group for the grouped mean operation of GCM and DEBS. Table 2 presents the average and 95-percentile window processing times of SPEAr and Storm with a CountMin sketch. It is apparent that SPEAr offers much lower processing time compared to CountMin on both datasets. In fact, the use of a sketch causes performance degradation, which is justified by the application of the computation-heavy hash functions required by CountMin. SPEAr's processing time is reduced by at least 9.89 times for both the mean and for the 95-percentile case.

TABLE 2: Proc. time (msec): SPEAr vs Storm/CountMin.

	Mean		95-%ile	
	SPEAr	CountMin	SPEAr	CountMin
GCM	.12	40.26	.04	107.6
DEBS	.09	3.7	.098	5.5

Take-away: SPEAr outperforms both Storm and Storm with a CountMin sketch by at least an *order of magnitude* in both the mean and the 95-percentile window processing time. This is due to SPEAr's ability to approximate windows' results by processing less tuples, without introducing additional overhead.

5.3. End-to-end Processing Time (Figure 9)

Next, we measure SPEAr's effects on end-to-end performance. This involves analyzing the end-to-end processing time achieved by Storm and SPEAr on a predetermined dataset. All our datasets' CQs carry time-based sliding windows on event time. As a result, it is infeasible to measure the effect of SPEAr on end-to-end processing time, without including the waiting time for watermarks. This happens because accumulating tuples and generating a watermark dominates execution time, and the merits of the processing speed achieved by SPEAr are dwarfed. However, with a count-based window definition, workers produce each window result by the time the configured number of tuples are met. As a result, the measured time reflects the total processing time required by each SPE. For this experiment, we used DEC's median CQ and set its window semantics to be count-based with a range from 2,500, to 47,000 tuples (DEC's mean window size). Fig. 9 presents the total processing time it takes for Storm and SPEAr to produce DEC's window medians. The reported processing times were measured in our micro-benchmark for which we used only a single worker. We set SPEAr's budget to 150 tuples to achieve $\epsilon \leq 10\%$ and $\alpha \geq 99\%$. Storm's performance remains relatively constant since the total amount of data processed remains the same. On the

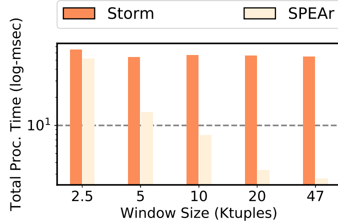


Figure 9: End-to-end Processing time on Median CQ for DEC with count-based windows.

other hand, SPEAr's performance increases as the window size increases. This happens because the bigger the window size becomes, the less tuples SPEAr processes (it carries a constant sample size per window). As a result, the higher the window becomes, the more tuples each window sample represents. In the smallest window size, Storm and SPEAr present comparable results since SPEAr processes a bigger fraction of tuples. However, as the windows' sizes increase, the effect of SPEAr increases and it outperforms Storm.
Take-away: SPEAr is capable of improving end-to-end processing time by more than an order of magnitude.

5.4. Window Size Sensitivity (Figure 10)

Next, we measure the effect of window size on SPEAr's performance. For this set of experiments, we use GCM and measure the average and 95-percentile window processing time with different window definitions. We set SPEAr's budget to 4,000 tuples, and execute the GCM operation on three window size settings: 900, 1,800, and 3,600 sec (with a slide of 450, 900, and 1,800 sec). Those translate to an average window size of 84,000, 164,588, and 320,000 tuples respectively. The reason we choose GCM for measuring SPEAr's sensitivity is because it features a grouped operation, and tuples for each group appear multiple times per window.

Fig. 10 depicts the average and 95-percentile processing time of Storm and SPEAr. When the window size is set to 900 sec, SPEAr achieves at least 2 times better processing time compared to Storm. However, the budget of 4,000 tuples is not enough to expedite all of the windows. In fact, SPEAr expedites only 68% of the total windows. This comes from SPEAr's accuracy estimation mechanism, which identifies that the approximate result is likely to have an average relative error $\geq$ 10%. Hence, the average and the 95-percentile window processing time contains the processing times of windows that are fully processed. When, the window size is set to 1,800 sec, SPEAr is able to accelerate 88% of the total number of windows. Consequently, the improvement in processing time compared to Storm increases further. Finally, when the window size is set to 3,600 sec, SPEAr is able to accelerate all windows and achieve more than *one order of magnitude* lower window processing time compared to Storm, with an average window relative error of less than 10%.
Take-away: SPEAr is able to identify windows that cannot be expedited. In those cases, SPEAr improves performance

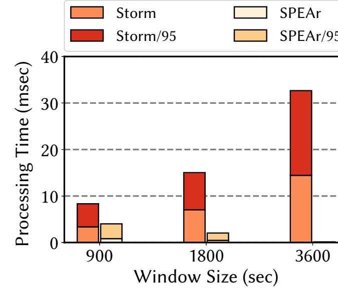


Figure 10: GCM processing time with varying window sizes.

up to 2 times compared to Storm. On larger windows, SPEAr is able to improve performance further. Overall, SPEAr's accuracy estimation mechanism is able to react to data changes and make the right decisions.

5.5. Error (Figure 11-12)

The next batch of experiments focuses on SPEAr's ability to identify opportunities for expediting processing, and the success rate of its accuracy estimation. For this set of experiments we use DEC and set the b to three different values: 250, 500, and 1000 tuples. The rationale for selecting those values, is a setting that is too low and will make SPEAr avoid approximation (250), a setting that is relatively low and will cause SPEAr to make false estimations (500), and a setting that will cause acceleration and acceptable accuracy on all windows (1000). Our expectation is to have SPEAr present the aforementioned behavior and avoid accelerating execution, when it comes at accuracy's expense. We established those b values offline by analyzing the DEC data. For this experiment, error was set to 10% and confidence to 95%, and SPEAr is configured to produce the mean result only at *watermark* arrival (i.e., no incremental optimization). We do this to quantify the performance degradation caused by SPEAr when the accuracy test fails (i.e., SPEAr adds overhead).

Fig. 11 presents the error of SPEAr when compared with the actual result on the DEC dataset. When $b = 250$, SPEAr does not accelerate processing often (an error of 0 indicates that SPEAr performs normal processing). This happens because SPEAr's accuracy estimation mechanism indicates that the interval is wider than 10% of the estimated value. As a result, SPEAr chooses to process the whole window. In fact, only 39.9% of windows are accelerated, and 33 of them produce a result which diverges more than 10% from the actual answer. When $b = 500$, SPEAr accelerates all windows, and its result diverges from the actual answer by more than 10% in 23 windows. Finally, when $b = 1000$ SPEAr accelerates all windows, and only two windows have an error more than 10%. In this case, SPEAr identifies that the budget b is enough to safely accelerate a window.

Fig. 12 presents the average and 95-percentile processing time on the DEC dataset (without the incremental processing optimization). When $b = 250$, SPEAr's performance is worse than Storm's, because SPEAr performs the check to see if b is enough to accelerate processing. However, since

Figure 11: Relative error per Window on DEC: the red (flat) line indicates the user-defined accuracy; the blue line indicates the error achieved by SPEAr.

it is not, SPEAr falls-back to exact processing and scans the whole window. As a result, SPEAr's performance is worse than Storm's. In contrast, when $b \geq 500$ SPEAr outperforms Storm by two orders of magnitude. This happens because b is ample to safely accelerate windows, and SPEAr achieves a significant performance improvement.

Take-away: SPEAr's accuracy estimation mechanism expedites processing without compromising accuracy. It is able to accurately detect situations which are safe to accelerate processing and results to significant performance gains.

6. Related Work

Incremental evaluation of CQs was first presented in [43]. This work along with more contemporary articles on incremental processing do not provide support for general holistic operations [37], [49], [39], [40]. Previous work, improves the amortized cost of updates and lookups for associative and invertible opertions. Only the work of [37] presents techniques for holistic operations, with a logarithmic update time, and a costly lookup time. The work of Wesley et al [60] presents a solution for incremental processing of distinct value quantiles. In comparison, SPEAr features a constant update time, lookup time, and memory cost (b) for holistic operations, and adopts all of the aforementioned techniques for incremental processing.

Previous work on load shedding aims at maintaining an SPEs performance under load without impacting accuracy of results [20], [22], [24], [41]. The work of Babcock et al. [22] focused on identifying shedding rates that maintain a low error, but it does not guarantee that results' accuracy will be sustained on input load fluctuations. *Semantic shedding* limits the error when the user provides a utility graph [20]. But, this approach is not feasible in queries whose goal is to identify patterns in real-time. Finally, [24] presents an optimal shedding algorithm, but without tight accuracy guarantees in rapidly changing input.

A lot of work has been done on sketch-based techniques. *Hyperloglog* [33] and *CountMin* sketch [29] are two of the most widely-used techniques among sketches. Despite the fact that those can provide accuracy guarantees and maintain memory usage low, they do not react to data changes and increase the computation per tuple. Finally, on a grouped operation, the space benefit of a sketch are diminished. This is due to the need to maintain the distinct groups in order to re-construct a sketch's information.

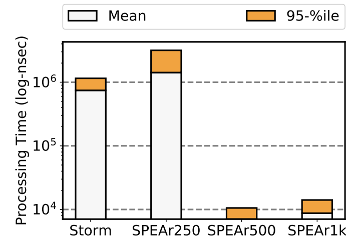


Figure 12: DEC processing time with varying budget.

Finally, a lot of SPEs with the ability to add additional workers have been previously presented [46], [50], [34], [47]. Such SPEs are able to maintain performance, and are geared towards long-term spikes in input. SPEAr's goal is to delay the need to scale-out, and it can complement those techniques. We consider this line of work orthogonal to SPEAr's goals, since SPEAr can be extended to support online reconfiguration of its resources.

7. Conclusion

Current SPEs' challenges emanate from the uncharted nature of stream processing, which dictates that data are not known at CQ submission time. Moreover, continuous execution increases the probability of fluctuating resource needs. SPEAr reduces the effects of the aforementioned characteristics proactively, postpones the need to scale resources in real-time, and mitigates the need for resource overprovisioning. Our evaluation indicates that SPEAr is able to significantly reduce memory usage, and improve performance in a plethora of *stateful* operations compared to a state-of-the-art SPE. Moreover, SPEAr automatically detects situations that allow expediting execution, and preserves results' accuracy. To the extent of our knowledge, SPEAr is the first system of its kind that automatically detects opportunities to accelerate processing, by applying AQP principles in streaming without manual intervention.

References

[1] A. Toshniwal, S. Taneja, A. Shukla *et al.*, "Storm@twitter," in *SIGMOD*, 2014, pp. 147–156.

[2] T. Akidau *et al.*, "Millwheel: Fault-tolerant stream processing at internet scale," *PVLDB*, vol. 6, no. 11, pp. 1033–1044, 2013.

[3] M. Fu, S. Mittal, V. Kedigehalli, K. Ramasamy *et al.*, "Streaming@twitter," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 15–27, 2015.

[4] B. Chandramouli and J. Goldstein, "Shrink - prescribing resiliency solutions for streaming," *PVLDB*, vol. 10, no. 5, pp. 505–516, 2017.

[5] A. Parameswaran, "Visual data exploration: A fertile ground for data management research," http://wp.sigmod.org/?p=2342, 2018.

[6] D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," in *SIGMOD*, 2018, pp. 1129–1140.

[7] V. Gulisano *et al.*, "Streamcloud: An elastic and scalable data streaming system," *TPDS*, vol. 23, no. 12, pp. 2351–2365, 2012.

[8] M. Zaharia *et al.*, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *HotCloud*, 2012.

[9] T. Akidau *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *PVLDB*, vol. 8, no. 12, 2015.

[10] B. Chandramouli *et al.*, "Trill: A high-performance incremental query processor for diverse analytics," *PVLDB*, vol. 8, no. 4, 2015.

[11] P. Carbone *et al.*, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.

[12] Mordor, "Streaming analytics market size-segmented by type, by deployment, end-user industry and region - growth, trends, and forecast (2019-2024)," 2018.

[13] Markets and Markets, "Streaming analytics market by type, applications, vertical, regions - global forecast to 2021," 2016.

[14] D. Abadi *et al.*, "Aurora: A new model and architecture for data stream management," *VLDBJ*, vol. 12, no. 2, 2003.

[15] D. J. Abadi, Y. Ahmad, M. Balazinska *et al.*, "The design of the Borealis stream processing engine," in *CIDR*, 2005.

[16] S. Kulkarni, N. Bhagat, M. Fu *et al.*, "Twitter heron: Stream processing at scale," in *SIGMOD*, 2015, pp. 239–250.

[17] A. Koliousis *et al.*, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *SIGMOD*, 2016, pp. 555–569.

[18] N. R. Katsipoulakis *et al.*, "A holistic view of stream partitioning costs," *PVLDB*, vol. 10, no. 11, pp. 1286–1297, 2017.

[19] A. Pacaci *et al.*, "Distribution-aware stream partitioning for distributed stream processing systems," in *BeyondMR*, 2018.

[20] N. Tatbul, U. Cetintemel, S. Zdonik *et al.*, "Load shedding in a data stream manager," *PVLDB*, vol. 29, pp. 309–320, 2003.

[21] A. Das, J. Gehrke, and M. Riedewald, "Approximate join processing over data streams," in *SIGMOD*, 2003, pp. 40–51.

[22] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *ICDE*, 2004, pp. 350–361.

[23] Y. Xing, S. Zdonik, and J. H. Hwang, "Dynamic load distribution in the borealis stream processor," in *ICDE*, 2005, pp. 791–802.

[24] B. Mozafari and C. Zaniolo, "Optimal load shedding with aggregates and mining queries," in *IEEE ICDE*, 2010, pp. 76–88.

[25] E. Kalyvianaki *et al.*, "Themis: Fairness in federated stream processing under overload," in *SIGMOD*, 2016, pp. 541–553.

[26] N. R. Katsipoulakis *et al.*, "Concept-driven load shedding: Reducing size and error of voluminous and variable data streams," in *IEEE BigData*, 2018.

[27] N. Shrivastava *et al.*, "Medians and beyond: new aggregation techniques for sensor networks," in *SenSys*, 2004.

[28] A. Metwally *et al.*, "Efficient computation of frequent and top-k elements in data streams," in *ICDT*, 2005.

[29] G. Cormode *et al.*, "An improved data stream summary: The count-min sketch and its applications," *JAlg*, vol. 55, no. 1, 2005.

[30] A. Shrivastava *et al.*, "Time adaptive sketches (ada-sketches) for summarizing data streams," in *SIGMOD*, 2016, pp. 1417–1432.

[31] Y. Zhou *et al.*, "Cold filter: A meta-framework for faster and more accurate stream processing," in *SIGMOD*, 2018, pp. 741–756.

[32] F. Bin, D. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *CoNEXT*, 2014, pp. 75–88.

[33] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm," in *EDBT*, 2013, pp. 683–692.

[34] R. Castro Fernandez, M. Migliavacca *et al.*, "Integrating scale out and fault tolerance in stream processing using operator state management," in *SIGMOD*, 2013, pp. 725–736.

[35] N. R. Katsipoulakis *et al.*, "Ce-storm: Confidential elastic processing of data streams," in *ACM SIGMOD*, 2015, pp. 859–864.

[36] Y. Wu and K. L. Tan, "Chronostream: Elastic stateful stream computation in the cloud," in *ICDE*, 2015, pp. 723–734.

[37] A. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates," in *PVLDB*, 2004, pp. 336–347.

[38] P. Carbone *et al.*, "Cutty: Aggregate sharing for user-defined windows," in *CIKM*, 2016, pp. 1201–1210.

[39] K. Tangwongsan *et al.*, "Low-latency sliding-window aggregation in worst-case constant time," in *DEBS*, 2017, pp. 66–77.

[40] A. Shein *et al.*, "Slickdeque: High throughput and low latency incremental sliding-window aggregation," in *EDBT*, 2018, pp. 397–408.

[41] T. Pham *et al.*, "Avoiding class warfare: managing continuous queries with differentiated classes of service," *VLDBJ*, vol. 25, no. 2, 2016.

[42] P. Carbone *et al.*, "State management in apache flink: Consistent stateful distributed stream processing," *PVLDB*, vol. 10, no. 12, 2017.

[43] L. Liu, C. Pu, R. Barga, and T. Zhou, "Differential evaluation of continual queries," in *ICDCS*, 1996, pp. 27–30.

[44] K. Tangwongsan *et al.*, "Optimal and general out-of-order sliding-window aggregation," vol. 12, no. 10, pp. 1167–1180, 2019.

[45] T. Heinze *et al.*, "Auto-scaling techniques for elastic data stream processing," in *DEBS*, 2014, pp. 318–321.

[46] T. Heinze, L. Roediger, A. Meister *et al.*, "Online parameter optimization for elastic data stream processing," in *SoCC*, 2015.

[47] T. N. Pham *et al.*, "Uninterruptible migration of continuous queries without operator state migration," *SIGMOD Rec.*, vol. 46, no. 3, 2017.

[48] G. S. Manku *et al.*, "Approximate medians and other quantiles in one pass and with limited memory," in *SIGMOD*, 1998, pp. 426–435.

[49] K. Tangwongsan *et al.*, "General incremental sliding-window aggregation," *PVLDB*, vol. 8, no. 7, pp. 702–713, 2015.

[50] T. Heinze *et al.*, "An adaptive replication scheme for elastic data stream processing systems," in *DEBS*, 2015, pp. 150–161.

[51] S. Agarwal *et al.*, "Blinkdb: Queries with bounded errors and bounded response times on very large data," in *EuroSys*, 2013, pp. 29–42.

[52] S. Kandula *et al.*, "Quickr: Lazily approximating complex adhoc queries in bigdata clusters," in *SIGMOD*, 2016, pp. 631–646.

[53] A. Galakatos *et al.*, "Revisiting reuse for approximate query processing," in *PVLDB*, 2017, pp. 1142–1153.

[54] Y. Park *et al.*, "Verdictdb: Universalizing approximate query processing," in *SIGMOD*, 2018, pp. 1461–1476.

[55] Y. Ioannidis and V. Poosala, "Histogram-based approximation of set-valued query-answers," in *PVLDB*, 1999, pp. 174–185.

[56] S. Acharya, P. B. Gibbons *et al.*, "Join synopses for approximate query answering," in *ACM SIGMOD*, 1999, pp. 275–286.

[57] J. Kang, J. F. Naughton, and S. D. Viglas, "Evaluating window joins over unbounded streams," in *ICDE*, 2003, pp. 341–352.

[58] U. Srivastava and J. Widom, "Memory-limited execution of windowed stream joins," *PVLDB*, vol. 30, 2004.

[59] S. Acharya *et al.*, "Congressional samples for approximate answering of group-by queries," in *SIGMOD*, 2000, pp. 487–498.

[60] R. Wesley and F. Xu, "Incremental computation of common windowed holistic aggregates," *PVLDB*, vol. 9, no. 12, 2016.

[61] W. G. Cochran, *Sampling Techniques*, 3rd ed. Wiley, 1977.

[62] Z. Jerzak *et al.*, "The debs 2015 grand challenge," in *DEBS*, 2015.