Efficient Incremental Computation of Aggregations over Sliding Windows

Chao Zhang, Reza Akbarinia, Farouk Toumani

Efficient Incremental Computation of Aggregations over Sliding Windows

Chao Zhang
LIMOS, CNRS, University of Clermont Auvergne
France
chao.zhang@uca.fr

Reza Akbarinia
LIRMM, INRIA, University of Montpellier
France
reza.akbarinia@inria.fr

Farouk Toumani
LIMOS, CNRS, University of Clermont Auvergne
France
farouk.toumani@uca.fr

ABSTRACT

Computing aggregation over sliding windows, *i.e.*, finite subsets of an unbounded stream, is a core operation in streaming analytics. We propose PBA (Parallel Boundary Aggregator), a novel parallel algorithm that groups continuous slices of streaming values into chunks and exploits two buffers, cumulative slice aggregations and left cumulative slice aggregations, to compute sliding window aggregations efficiently. PBA runs in $O(1)$ time, performing at most 3 merging operations per slide while consuming $O(n)$ space for windows with n partial aggregations. Our empirical experiments demonstrate that PBA can improve throughput up to 4× while reducing latency, compared to state-of-the-art algorithms.

CCS CONCEPTS

• **Information systems → Data streaming**.

KEYWORDS

Data Stream; Streaming Algorithm; Sliding Window Aggregation

ACM Reference Format:
Chao Zhang, Reza Akbarinia, and Farouk Toumani. 2021. Efficient Incremental Computation of Aggregations over Sliding Windows. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3447548.3467360

1 INTRODUCTION

Nowadays, we are witnessing the production of large volumes of continuous or real-time data in many application domains like traffic monitoring, medical monitoring, social networks, weather forecasting, network monitoring, etc. For example, every day around one trillion messages are processed through Uber data analytics infrastructure[1] while more than 500 million tweets are posted on Twitter [11]. Efficient streaming algorithms are needed for analyzing data streams in such applications. In particular, aggregations [23], having the inherent property of summarizing information from data, constitute a fundamental operator to compute real-time statistics in this context. In the streaming setting, aggregations are typically computed over finite subsets of a stream, called *windows*. In particular, Sliding-Window Aggregation (SWAG) [20, 21, 26, 28] continuously computes a summary of the most recent data items in a given range r (aka window size) and using a given slide s. If the range and slide parameters are given in time units (*e.g.*, seconds), then the sliding window is *time-based*, otherwise, *i.e.*, if these parameters are given as the number of values, it is *count-based*. Fig. 1 presents an example of computing *sum* over the count-based sliding window with a range of 10 values and a slide of 2 values.

Stream processing systems (SPSs) [2, 6, 19, 30, 32] are ubiquitous for analyzing continuously unbounded data. However, one of the challenges faced by SPSs is to efficiently compute aggregations over sliding windows. This requires the ability of such systems to incrementally aggregate moving data, *i.e.*, inserting new data items and evicting old data items when a window is sliding without recomputing the aggregation from scratch. High throughput and low latency are essential requirements as SPSs are typically designed for real-time applications [13].

Two orthogonal techniques have been proposed to meet such requirements: *slicing* (aka partial aggregation) [7, 18, 20, 31], and *merging* (aka incremental final aggregation) [25–28]. Slicing techniques focus on slicing the windows into smaller subsets, called *slices*, and then computing partial aggregation over slices, called *slice aggregations*. The benefit of the slicing technique is that slice aggregations (*i.e.*, partial aggregations over slices) can be shared by different window instances. For example, in Fig. 1, a slice aggregation over a slice of 2 values can be shared by 5 windows. On the basis of a slicing technique, final aggregations over sliding windows are computed by merging slice aggregations, *e.g.*, in Fig. 1, SWAGs are computed by merging 5 slice aggregations. During the merging phase, each insertion or eviction is processed over a slice aggregation rather than a value. In modern SPSs, practical sliding windows can be very large [28], thereby making the merging phase non-trivial. To efficiently merge slice aggregation, *incremental computation* is necessary because an approach that recalculate slice aggregations from scratch (hereafter called Recal) is very inefficient [20, 21, 26, 28].

The difficulty of incremental merging depends on the considered class of aggregations: *invertible* or *non-invertible*. An aggregation is invertible if the merging function for its partial aggregations has an inverse function. For example, *sum* is clearly invertible, because it has the arithmetical subtraction as the inverse function. Similarly, *avg* and *std* are invertible because they use *addition* as merging function. Partial aggregations of invertible aggregations can be

[1] AthenaX: SQL-based streaming analytics platform, https://eng.uber.com/athenax

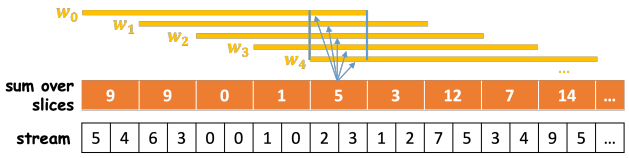


Figure 1: Example of computing *sum* over a count-based sliding window with a range of 10 values and a slide of 2 values.

efficiently merged using the *subtract-on-evict* algorithm [13, 27], *i.e.*, maintaining a running *sum* over a sliding window by subtracting evicted values and adding inserted values. However, this is not the case for non-invertible aggregations, *e.g.*, *max*, *min*, and *bloom filter*, where the *subtract-on-evict* algorithm cannot be applied at the merging phase. Incremental merging of continuous data in the context of non-invertible aggregations is challenging and requires more sophisticated algorithms (*e.g.*, see [5, 25–28]).

This paper focuses on merging slice aggregations for computing non-invertible aggregations over FIFO sliding windows with an arbitrary range and an arbitrary slide. We propose PBA (Parallel Boundary Aggregator), a novel algorithm that computes incremental aggregations in parallel. PBA groups continuous slices into *chunks*, and maintains two buffers for each chunk containing, respectively, the *cumulative slice aggregations* (denoted as *csa*) and the *left cumulative slice aggregations* (denoted as *lcs*) of the chunk's slices. Using such a model, SWAGs can be computed in constant time bounded by 3 for both amortized and worst-case time. Interestingly, the required *csa* and *lcs* for each window aggregation are completely independent from each other, so the incremental computation of *csa* and *lcs* can be achieved in parallel. These salient features put PBA ahead of state-of-the-art algorithms in terms of throughput and latency.

In this paper, we make the following main contributions:

- We propose a novel algorithm, PBA, to efficiently compute SWAGs. PBA uses chunks to divide final aggregations of window instances into sub-aggregations that are elements of *csa* and *lcs* buffers. These two buffers can be computed incrementally and in parallel.
- We analyze the latency caused by SWAG computations with different chunk sizes in PBA, and propose an approach to optimize the chunk size leading to the minimum latency.
- We conduct extensive empirical experiments, using both synthetic and real-world datasets. Our experiments show that PBA behaves very well for average and large sliding windows (*e.g.*, with sizes higher than 1024 values), improving throughput up to 4× against state-of-the-art algorithms while reducing latency. For small-size windows, the results show the superiority of the non-parallel version of PBA (denoted as SBA) that outperforms other algorithms in terms of throughput.
- To show the benefit of our approach in modern stream-processing frameworks, we implemented PBA on top of Apache Flink [3, 6], called FPBA. Our empirical evaluation shows that FPBA scales well as increasing the parallelism of Flink in both local and cluster modes.

The rest of this paper is organized as follows. In Section 2, we present the background related to SWAG. We discuss state-of-the-art algorithms for merging slice aggregations, and parallelization in SPSs in Section 3. PBA is presented in Section 4. Our experimental evaluation is reported in Section 5. We conclude in Section 6. All related proofs and pseudo-codes are included in our technical report, which is available online [1].

2 BACKGROUND

2.1 Sliding Window Aggregations

A sliding window $W[r, s]$, $r > s$, with a range r and a slide s is an infinite sequence of window instances $(w_0, w_1, w_2, ...)$. Sliding window aggregation means computing an aggregate function α over each window instance of a sliding window $W[r, s]$, *i.e.*, $\alpha(W[r, s]) = (\alpha(w_0), \alpha(w_1), \alpha(w_2), ...)$.

We assume that the sliding window respects the FIFO semantic, where values that are first inserted into the window will be first evicted from it. In this case, SWAG is computed by aggregating streaming values according to their arrival order.

Two stream-processing models are widely used by SPSs. One is the *micro-batch processing model*, where a micro batch containing a finite subset of values is processed at a time, *e.g.*, Apache Spark [4]. The other is the *continuous processing model*, where streaming values are processed one by one, *e.g.*, Apache Flink [3]. We consider the *continuous processing model* in this paper. In this case, the computation of SWAGs is triggered immediately at the boundary of window instances.

2.2 Stream Slicing

To share partial aggregations, a stream is sliced to generate disjoint subsets of streaming values, called slices. SWAGs can be computed by merging partial aggregations of slices. The state-of-the-art slicing technique is Cutty [7], in which a partial aggregate is incrementally computed for each slice. Given a sliding window $W[r, s]$, the size of each slice determined by Cutty is s which equals to the slide size. For example, for the sliding window $W[10, 3]$, each slice has 3 streaming values, and an instance of $W[10, 3]$ will cover 3 slices and 1 streaming value. We consider Cutty as the underlying slicing technique in this paper.

2.3 Aggregation Functions

Modern frameworks provide a uniform interface to capture general aggregations. In the streaming context, aggregation functions can be decomposed into three functions: lift, combine, and lower [13, 28], called hereafter the LCL framework. We use the geometric mean, *i.e.*, $gm(X) = (\prod_{i=1}^{n} x_i)^{1/n}$, as an example to illustrate them:

- lift maps a streaming value to a tuple of one or several values, *e.g.*, $gm.lift(v) = (v, 1)$;
- combine continuously merges two outputs of lift, *e.g.*, $gm.combine((v_1, n_1), (v_2, n_2)) = ((v_1 \times v_2), (n_1 + n_2))$;
- lower maps a tuple to a final aggregation result, *e.g.*, $gm.lower(v, n) = v^{1/n}$.

The combine function is also known as *merging function* and its outputs are called *partial aggregations*. We use $\oplus$ to denote a combine function through this paper.

An aggregation is invertible if its combine function is invertible. Let x, y be any values from the domain of a combine function $\oplus$. Then $\oplus$ is invertible if there exists an operation $\ominus$, such that $x \oplus y \ominus y = x$, *e.g.*, the combine function of geometric mean is invertible as its inverse function is a pair of division and subtraction.

Aggregations can also be classified in *distributive, algebraic* and *holistic* [10]. Distributive aggregations, *e.g.*, *sum* and *max*, have an identical constant size for both partial and final aggregation. Algebraic aggregations have a bounded size for partial aggregation, *e.g.*, *average* and *geometric mean*. The other aggregations are holistic, which have an unbounded size for partial aggregations, *e.g.*, *median* and *percentile*. In this paper, we consider non-invertible aggregations that are distributive or algebraic. Note that, such consideration is consistent with state-of-the-art works [25–27].

3 STATE OF THE ART

This section reviews state-of-the-art algorithms devoted to merging slice aggregations, and discusses slicing techniques and parallelization in SPSs. A more broad vision on SWAG can be found in [13] while existing surveys review various facets of SPSs [8, 12, 14].

3.1 Incremental Merging of Slice Aggregations

To merge slice aggregations for non-invertible aggregations, efficient algorithms are needed. The performance of such algorithms can impact the overall aggregation time, because latency caused by computing SWAG has the nature of propagation towards computations over the following ones. To our best knowledge, TwoStack [27], DABA [27], FlatFIT [25], SlickDeque (Non-Inv) [26] and SlideSide (Non-Inv) [29] are the most efficient state-of-the-art algorithms for computing non-invertible aggregations over FIFO windows. Consider each instance of a sliding window that requires merging n slice (partial) aggregations. In Table 1, we present the algorithmic complexities of the state-of-the-art algorithms computed originally in [26, 29], and also our algorithm PBA (see Section 4.4 for explanation). TwoStack pushes pairs of a partial aggregation and a streaming value into a back stack and pops such pairs from a front stack. Computing an aggregation over a sliding window instance only needs to peek and merge partial aggregations from the front and back stack. However, when the front stack is empty, TwoStack flips all items in the back stack onto the front stack requiring $O(n)$ time. DABA [27] leverages TwoStack to gradually perform the flipping operation of TwoStack which amortizes its time to $O(1)$. As a consequence, DABA is in $O(1)$ worst-case time. FlatFIT [25] only recalculates window aggregation one time per $n+1$ slides. Intermediate aggregation results are maintained in an appropriate index structure and reused to avoid redundant computations. Hence, on average, each of $n+1$ slides can be computed by 3 merging steps. SlickDeque (Non-Inv) uses a deque to store streaming values, and not every streaming value will be added into the deque. For example, when computing *max* over sliding windows, SlickDeque continuously replaces the end of a deque v' using a new value v if $v > v'$. Adding a value v will depend on the rank of v w.r.t the number of values in the deque which could be n. The worst time complexity of SlickDeque is $O(n)$ while its amortized time is less than 2 because a new value will never be compared more than twice. SlideSide (Non-Inv) uses two stacks to store partial aggregations, like TwoStack, but shares values between them. While, an operation similar to the flipping in TwoStack is still required, resulting in $O(n)$ worst-case time.

Table 1: Algorithmic Complexities (Non-Invertible SWAG).

Algorithm	Time (amortized)	Time (worst)	Space
TwoStack	3	n	$2n$
DABA	5	8	$2n$
FlatFIT	3	n	$2n$
SlickDeque	< 2	n	$2n$
SlideSide	3	n	$2n$
PBA	< 3	3	$\frac{3n+13}{2}$

As shown in Table 1, which contains complexity analysis of PBA described in Section 4.4, PBA is the best for time and space complexity, except the comparison with SlickDeque in amortized time. However, unlike SlickDeque, PBA runs in constant time, which is also experimentally demonstrated in Section 5.1.1. Such a feature of PBA guarantees its performance in the worst case. Note that amortized and worst-case time determine throughput and latency, respectively.

In addition to its low time and space complexities, one of the main advantages of PBA compared to the state-of-the-art algorithms, *e.g.*, TwoStack [27], DABA [27], FlatFIT [25], SlickDeque (Non-Inv) [26], and SlideSide (Non-Inv) [29], is that the latter algorithms cannot use additional threads to get performance gains, *i.e.*, computing intermediate results in parallel. For instance, in TwoStack, the two stacks are interdependent. In SlickDeque, the deque needs to be traversed in the worst case. However, the *lcs* and *csa* buffers used for computing SWAGs in PBA are completely independent of each other, allowing their incremental computation to be done in parallel.

3.2 Combining Slicing and Merging

Various slicing techniques have been proposed in the literature, among which we could mention Panes [20], Pairs [18], Cutty [7], and Scotty [31]. The main goal is to determine sizes of intervals over the stream called *slices* which are then used to split each window into several slices in order to allow sharing the aggregations computed over slices. Slicing techniques can improve throughput by amortizing the total time of merging evenly over each stream value in each slice. However, having only a slicing technique cannot reduce latency caused by merging slice aggregations, which is especially important for a continuous stream-processing model.

A merging technique on top of a slicing one can deal with the latency issues, and additionally improves the throughput. First of all, latency can be reduced, as demonstrated in Cutty and Scotty through combining their slicing techniques with the FlatFAT [28] merging algorithm. *Eager slicing* of Scotty (using FlatFAT for merging) shows up to two orders of magnitudes reduction of latency, compared to *lazy slicing* (with Recal as the merging algorithm). In addition, an advanced merging algorithm can also improve throughput. Specifically, [27] shows that TwoStack outperforms Recal, except in the case of very small window sizes, and TwoStack is always faster, up to 6×, than FlatFAT. In this paper, we adopt the Cutty slicing technique and design a constant-time merging algorithm to

reduce latency and improve throughput. It is worth noting that our merging approach remains independent from slicing techniques.

3.3 Parallelization in SPSs

Parallelization in SPSs is an active research area. A large piece of work in this field focuses on *inter-operator* parallelism with a clear dominance of key-based splitting techniques [24] in shared nothing cluster architecture. Regarding *intra-operator* parallelism, [22] presents parallel patterns to handle generic stateful operators (*e.g.*, sorting, join, grouping, ...). PBA fits into the category of exploiting *intra-operator* parallelism to incrementally compute SWAGs.

Novel SPSs are designed to exploit multi-core processors to process streams in parallel [17, 34]. In this line of work, SABER [17] processes fixed-size batches in parallel using a hybrid architecture of CPUs and GPGPUs (*i.e.*, modern GPUs), and merges local results of batches to produce the final result. The parallelism of SABER is suitable for the micro-batch model, giving the opportunity to process each micro batch by separate tasks (one task for each batch [17]). PBA is designed for the continuous processing model. In addition, PBA merges at most 3 sub-aggregations to produce the final aggregation for arbitrary cases, *i.e.*, requiring only constant time for merging, which is achieved by using the optimal chunk size computed based on the range and slide parameters in a continuous aggregate query.

Modern distributed SPSs [2–4, 19, 30, 32] compute SWAGs over a stream of key-value pairs in parallel. For example, in Apache Flink [3, 6], one can transform a *DataStream* into a *KeyedStream* with disjoint partitions, where a key will be assigned to a specific partition. SWAGs over different partitions can be computed in parallel using multiple cores of nodes in a cluster. The parallel computation in PBA is orthogonal to such distributed SPSs, since PBA computes SWAGs over values with the same key in parallel, which makes PBA suitable to extend partition-oriented parallelism in distributed SPSs.

4 PARALLEL BOUNDARY AGGREGATOR

In PBA, a stream is considered as a sequence of chunks having an identical number of slices. Non-invertible aggregations over window instances are computed by merging two buffers of cumulative aggregations over slice aggregations in chunks. To efficiently compute SWAG, buffers are computed in parallel using two threads. As SPSs are usually running on modern CPUs [33], in PBA we assume that the system is equipped with at least two cores allowing to run two tasks in parallel.

4.1 Main Ideas

Let us consider a query with the non-invertible aggregation *max* over a range of 10 values and a slide of 2 values, *i.e.*, $W[10, 2]$. Slices of 2 values can be created, and thus each window instance needs to merge 5 slice aggregations. In general, the final aggregation needs to merge $\lfloor r/s \rfloor$ slice aggregations for each instance of $W[r, s]$.

PBA is an $O(1)$ time solution for $W[r, s]$, which is achieved by maintaining two buffers: *csa* (*cumulative slice aggregations*) and *lcs* (*left cumulative slice aggregations*). Generally, a *csa* buffer is computed through accumulating slice aggregations from left to right inside a *chunk* (an array of slices), and a *lcs* buffer from right to

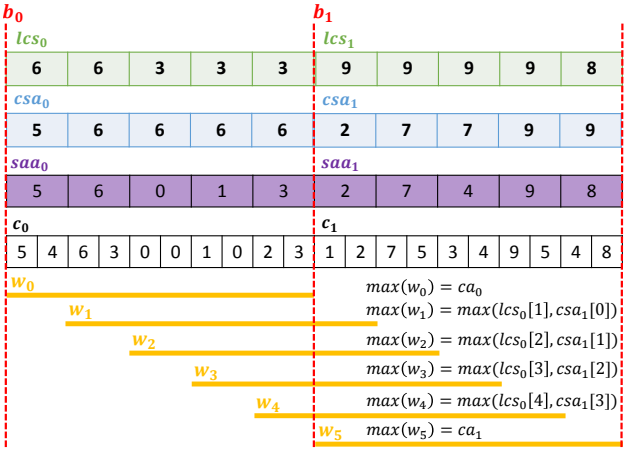


Figure 2: Example of computing the count-based $W[10, 2]$ with the aggregation *max* over a stream using PBA with a chunk size of 5 slices.

left. Then SWAG can be computed by merging 2 elements respectively from *lcs* and *csa* for this example. The two buffers of PBA for computing *max* over $W[10, 2]$ are shown in Fig. 2, *e.g.*, csa_0 and lcs_0 for chunk c_0. Let us explain in detail how the two buffers are computed. Streaming values are partitioned into chunks having an identical number of slices, *e.g.*, c_0 and c_1 in Fig. 2 have 5 slices, and chunks are separated by boundaries, *e.g.*, b_1 between c_0 and c_1. Slice aggregations inside a chunk are sequentially stored in a *saa* (slice aggregation array), *e.g.*, saa_0 and saa_1 in Fig. 2. Then, *csa* and *lcs* are computed over *saa*, which are two arrays having the same length as a *saa* array. For example, csa_1 and lcs_0 are computed by memorization of each intermediate result during accumulating elements of saa_1 and saa_0, illustrated as follows.

- csa_1 is obtained by accumulating from $saa_1[0]$ to $saa_1[4]$,
- lcs_0 is obtained by accumulating from $saa_0[4]$ to $saa_0[0]$.

The last element in a *csa* array is denoted as a *ca* (chunk aggregation), *e.g.*, $ca_1 = csa_1[4] = 9$. Aggregation over a window instance is computed using elements in *lcs* and *csa* arrays, *e.g.*, in Fig. 2, $max(w_3) = max(lcs_0[3], csa_1[2])$. Consequently, *max* over each window instance, *e.g.*, each of $(w_0, ..., w_4)$, only requires at most 1 *max* operation.

Challenges in PBA: In order to obtain the constant-time solution described above, the two buffers, *csa* and *lcs*, must be efficiently computed as streaming values keep arriving. *csa* can be computed by accumulating streaming values and slice aggregations. The main difficulty is how to compute *lcs* efficiently, since computing a *lcs* array requires traversing a *saa* array inversely, which means having to stop computing *csa*, resulting in an eventual high delay. We use a parallel strategy to deal with this issue. Our key observation is that SWAG never uses *csa* and *lcs* from the same chunk, so the inputs required to compute *csa* and *lcs* are completely independent. For example, in Fig. 2, computing $max(w_0)$ requires lcs_0 and csa_1. lcs_0 is computed with saa_0 as input, which is complete at the boundary b_1. While csa_1 is computed with saa_1 as input, which is independent of saa_0. The independence between the inputs of

lcs_0 and csa_1 allows the task of computing lcs_0 to be carried out when streaming values reach the boundary b_1. This task runs in parallel and simultaneously with the task of computing csa_1. Thus, PBA can keep receiving new streaming values after b_1 and compute csa_1. Another issue is whether the task of computing lcs_0 can be finished before the end of w_1, *i.e.*, whether PBA needs to wait for the result of $lcs_0[1]$ to compute $max(w_1)$. Such an issue is dealt with in Section 4.3, where we identify the optimal chunk size minimize the waiting time.

4.2 PBA Model

In PBA, a stream is considered as a sequence of non-overlapping *chunks* of slices. Each chunk has an identical number of slices and starts at a specific time denoted as *boundary*.

- b_i (*boundary*): the start of the chunks, *e.g.*, b_1 in Fig. 2;
- c_i (*chunk*): a sequence of slices from b_i to b_{i+1}, *e.g.*, c_0 in Fig. 2;
- b (*chunk size*): the number of slices in a chunk, *e.g.*, 5 in Fig. 2.

We incrementally compute partial aggregations for every chunk to compute SWAGs. PBA applies two kinds of incremental aggregations: *(i)* computing *csa* and *(ii)* computing *lcs*. To compute *csa*, PBA adopts a two-level incremental computation inside each chunk: *(i.1) accumulating streaming values* inside each slice to have a *slice aggregation*, and *(i.2) accumulating slice aggregations* inside each chunk to have a *chunk aggregation*. To compute *lcs*, PBA maintains intermediate results during accumulating slice aggregations from right to left in a slice aggregation array saa_i.

Dividing SWAG into Sub-Aggregations: As the window instances, whose starts coincide with boundaries in every chunk, are trivial cases requiring only *csa* elements *e.g.*, w_0 in Fig. 2, we focus on the other cases in the sequel. To illustrate the PBA model, we first define $k = \lfloor \frac{\lfloor r/s \rfloor}{b} \rfloor$, which denotes the number of chunks a window instance can fully cover, *e.g.*, $k = \lfloor \frac{\lfloor 10/2 \rfloor}{5} \rfloor = 1$ in Fig. 2. In the PBA model, we use boundaries to divide a window instance w of $W[r, s]$ into several sub-parts and merge partial aggregations of sub-parts to have a final aggregation of w. The number of sub-parts can be $k+1$ or $k+2$, depending on $W[r, s]$ and the used chunk size. We denote a partial aggregation of each sub-part as a *sub-aggregation*, and all sub-aggregations of w as $(SA_0, SA_1, ...)$, *i.e.*, SA_0 is the first one (see Fig. 3 for an example). Then the first sub-aggregation is an element in *lcs*, and the last one in *csa*. Each of the others is a chunk aggregation, *i.e.*, the last element in a *csa* buffer. An example of $k = 1$ is shown in Fig. 2, where each window instance is divided into 2 sub-parts and SWAGs are computed by merging 2 sub-aggregations. To illustrate the case of $k > 1$, Fig. 3 presents the example of computing a count-based $W[16, 1]$ with a chunk size of 4, where we have $k = 4$ and each window instance is divided into 5 sub-parts, *e.g.*, $max(w_1) = max(lcs_0[1], ca_1, ca_2, ca_3, csa_4[0])$.

Efficient Computation of *lcs*: In the final aggregation stage, since *lcs* and *csa* buffers used for merging are computed over completely independent *saa* arrays, we compute *lcs* and *csa* buffers in parallel, *i.e.*, two tasks run simultaneously and each task focus on computing a single buffer. Specifically, we maintain two threads in PBA. The main thread is used to compute *csa* and the final aggregation, and the other for processing the task of computing *lcs* launched at each boundary. Using a multi-core CPU, new streaming values can be immediately accumulated to *csa* without any interruption.

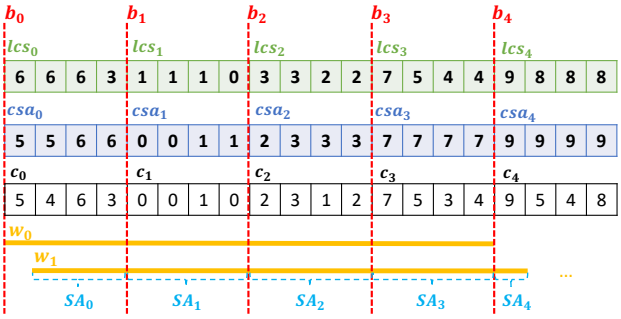


Figure 3: Example of using a chunk size of 4 to compute *max* a count-based $W[16, 1]$, which leads to $k = \lfloor \frac{\lfloor 16/1 \rfloor}{4} \rfloor = 4$ and the window instance w_1 is divided into 5 sub-parts.

4.3 Optimal Chunk Size in PBA

The chunk size is crucial for PBA. Specifically, a carelessly selected chunk size can cause delay to compute final aggregations. For instance, consider the example in Fig. 2, and let 1 T_u be the time required to execute $\oplus$, *i.e.*, a merging function or a *combine* function. Although the two tasks for computing lcs_0 and csa_1 are simultaneously started at the boundary b_1, $lcs_0[1]$ cannot be obtained earlier than $csa_1[0]$ because computing $lcs_0[1]$ (3 T_u) requires two more T_u than computing $csa_1[0]$ (1 T_u). Thus, PBA needs to wait for 2 T_u to use $lcs_0[1]$ and $csa_1[0]$ to compute $max(w_1)$.

The following two theorems identify the *optimal chunk size*, denoted as b^*, for count-based windows and time-based window, respectively. The corresponding proofs are provided in our technical report, which is available online [1].

Theorem 1. *Given a count-based sliding window $W[r, s]$, the optimal chunk size b^* in PBA can be obtained as follows:*

- *if* $\lfloor r/s \rfloor \leqslant s + (r \bmod s) + 1$, *then* $b^* = \lfloor r/s \rfloor$;
- *otherwise* $b^* = \lfloor \frac{s(\lfloor r/s \rfloor + 1) + r \bmod s + 1}{s+1} \rfloor$.

With b^*, we have $k = \lfloor \frac{\lfloor r/s \rfloor}{b^*} \rfloor = 1$.

Theorem 2. *Given a time-based sliding window $W[r, s]$, the optimal chunk size b^* in PBA can be obtained as follows:*

- *if* $\lfloor r/s \rfloor \leqslant \frac{s+2T_u+C}{T_u}$, *then* $b^* = \lfloor r/s \rfloor$;
- *otherwise* $b^* = \lfloor \frac{\lfloor r/s \rfloor (s+T_u)+s+2T_u+C}{s+2T_u} \rfloor$,

where $C = 0$ *if* $r \bmod s = 0$, *otherwise* $C = s(r \bmod s) + 1\ T_u$. *With b^*, we have* $k = \lfloor \frac{\lfloor r/s \rfloor}{b^*} \rfloor = 1$.

The optimal chunk size ensures that *(i)* each *lcs* buffer can be obtained without waiting, and *(ii)* $k = 1$ which leads to the minimum number of merging operations to compute final aggregations.

4.4 Complexity Analysis

As previous works [25–27, 29] present complexities by computing a count-based $W[n, 1]$ with a range of n and a slide of 1, *i.e.*, merging

n slice aggregations for each window instance, we also rest on such a scenario to discuss complexities of PBA.

Given a sliding window $W[n, 1]$, as PBA is used with the optimal chunk size b^*, we have $b^* = \lfloor \frac{n+2}{2} \rfloor$ and $k = \lfloor \frac{\lfloor r/s \rfloor}{b^*} \rfloor = 1$ according to Theorem 1. Thus, a window instance will be divided into either $k + 1 = 2$ or $k + 2 = 3$ sub-parts, *i.e.*, spanning 2 or 3 chunks. Based on this, we discuss the time and space complexity of PBA below.

Time Complexity: In the final aggregation stage, merging 2 or 3 sub-aggregations of sub-parts needs 1 or 2 merging operations. Considering that the last streaming value of a window will be merged with *csa*, the total number of merging operations by PBA is 2 or 3. Therefore, the *worst-case* time is 3, and the *amortized* time is less than 3.

Space Complexity: As a window instance can span at most 3 chunks, w.l.o.g, we denote the three chunks as c_{i-2}, c_{i-1}, c_i, and consider the current streaming value is arriving at a slice of c_i. Thus, 3 *saa* arrays of size b^* are needed. Note that the *lcs* arrays of the three chunks will be computed over *saa* arrays and do not require extra space. In addition, PBA only needs the *csa* array of the current chunk c_i, *i.e.*, csa_i, and maintains one element in csa_i, the current one, rather than the entire csa_i array. For windows spanning 3 chunks, the chunk aggregation ca_{i-1} of chunk c_{i-1} is also needed. Thus, the total space required by PBA is: $3b^* + 2 = 3\lfloor \frac{n+2}{2} \rfloor + 2 \leq \frac{3n+13}{2}$.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of PBA against the state-of-the-art algorithms for computing non-invertible aggregations over sliding windows with FIFO semantics. Moreover, we integrate PBA into Apache Flink and present the corresponding improvement in local and cluster modes, respectively.

5.1 PBA Compared to Alternatives

State-of-the-Art Algorithms: In our experimental evaluation we compare PBA to the two best state-of-the-art algorithms: SlickDeque [26] and TwoStack [27]. SlickDeque shows the best performance in terms of throughput and latency [26] while TwoStack is slightly worse than SlickDeque but behaves better than the others [26, 27, 29]. Note that, although DABA [27] has a constant time complexity in the worst case, it does not show a better performance result compared to SlickDeque in terms of throughput and latency [26], or TwoStack in terms of throughput [27]. We also consider the naive algorithm algorithm and call it Recal (recalculate). As the sliding window moves, it calculates an aggregation from scratch. To study the performance of our algorithm in the case of single-core machines, we implemented a sequential version of PBA, denoted as SBA (sequential boundary aggregator), which uses only one thread to compute SWAG. SBA follows the same procedure as PBA, except that the task of computing *lcs* in SBA is done in the same thread as the one that computes *csa* buffer (SBA uses only one thread). In order to slice a data stream, we set each slice size to be equal to the slide size (as described in Section 2.2 for details). Notice that this is the underlying slice solution for all algorithms tested in our evaluation. We implemented PBA and SBA using Java 8, and the state-of-the-art algorithms based on the pseudo-codes in [27] and [26]. The source codes of PBA, SBA, and the implementation of state-of-the-art algorithms are available online [1].

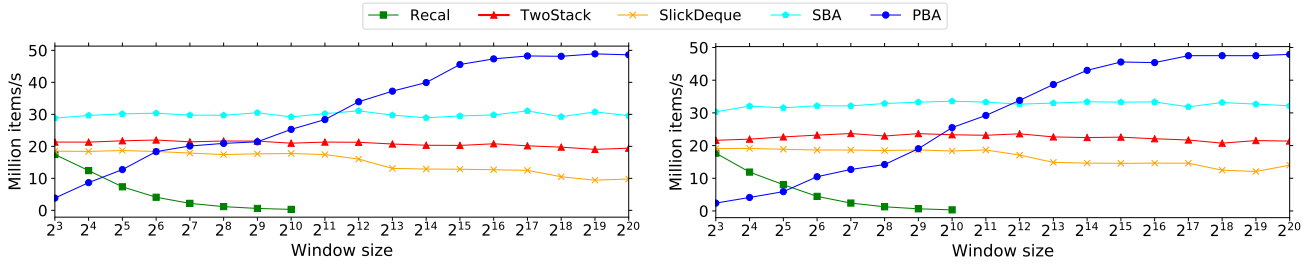
Datasets: We use two datasets, a real-world and a synthetic one, for throughput experiment. The real-world dataset is the DEBS12 Grand Challenge Manufacturing Equipment Dataset [15], which is also used by SlickDeque [26]. This dataset contains energy consumption recorded by sensors of manufacturing equipment with the frequency of 100 records per second, and each tuple is associated with a timestamp, which we will use in our time-based experiments. The original dataset contains 32 million tuples, which we made into 130 million tuples, *i.e.*, 4 copies. The synthetic dataset contains 200 million values generated using the uniform distribution. Note that in our experiments we observed the same tendency for PBA with different datasets, this is why we have not used more than two test datasets. For the latency experiments, we used the synthetic dataset and tested latency results for 1 million window instances.

In our experiments, as aggregation function we use *max*, commonly used in related works [25, 26, 28, 29] as the representative of non-invertible aggregations. We first evaluate throughput for count-based windows with a fixed slide while varying the ranges. Then we evaluate the case with a fixed range while varying the slides. Similarly, we evaluate the throughput for time-based windows in both settings (i.e., with a fixed slide and varying ranges, and then with a fixed range and varying slides). We test latency for both count-based and time-based windows. To further study the latency of different algorithms, we also test a more complex aggregation, MomentSketch = $(min, max, sum, count, \sum x_i, ... \sum x_i^l, \sum logx_i, ... \sum (logx_i)^l)$[9] with $l = 10$, which can approximate percentiles.

Experiment Setup: We run the throughput and latency experiments at a server with Ubuntu 16.01.3 LTS, 6 virtual CPUs of Intel(R) Xeon(R) 2.40GHz, and 16GB main memory, where the heap size of JVM is configured to 14GB.

5.1.1 Throughput Experiments. The workload for evaluating *max* over count-based windows is as follows: 1) range from 2^3 to 2^{20} with the slide size set to one; 2) slide from 1 to 10 with a range of 2^{17}. For the time-based windows the workload is as follows: 1) range from 6 to 60 minutes with a slide of 10 milliseconds; 2) slide from 10 milliseconds to 100 milliseconds with a range of 1 hour. We do not test for larger slide sizes as the overall throughput will be amortized by the time of computing slice aggregations, such that the differences between merging algorithms cannot be observed (see Section 3.2 for detailed discussion). We run the workload 20 times and report the median of all evaluation results. We stop reporting the throughput of the Recal algorithm when the corresponding throughput is too small. Our experimental results are presented in Fig. 4, Fig. 5, and Fig 6. We discuss the results in the sequel.

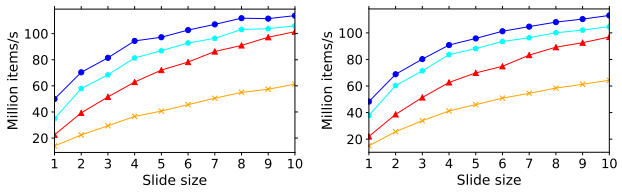
Count-based windows with a fixed slide. Our throughput results for this case are shown in Fig. 4. PBA shows the best throughput with large window sizes, and SBA (the sequential version of PBA) shows a higher throughput than TwoStack and SlickDeque, even for small window sizes. More precisely, the throughput of PBA increases w.r.t to the increase of window size r when $r < 2^{15}$, and keeps consistent when $r \geqslant 2^{15}$. When $r \geqslant 2^{15}$, the throughput of PBA is 2.5× higher than that of TwoStack and 4× higher than that SlickDeque. Note that, the throughput of PBA increases w.r.t to the



(a) DEBS'12 dataset [15]

(b) Synthetic dataset

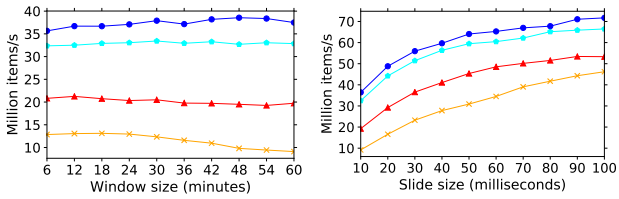
Figure 4: Throughput for count-based windows by varying the window size and with a fixed slide size.



(a) DEBS'12 dataset [16]

(b) Synthetic dataset

Figure 5: Throughput for count-based windows by varying the slide size and with a fixed range.



(a) Fixed-slide experiment

(b) Fixed-range experiment

Figure 6: Throughput for time-based windows.

window size r when $r < 2^{15}$, because a smaller window size will lead to a smaller optimal chunk size in PBA. Thus, the parallel task in PBA, *i.e.*, computing *lcs*, will consequently require less time but will be launched more frequently, compared to a case with a larger window size. In such a case, the constant overhead for launching a parallel task will be relatively more obvious. For example, when $r = 2^5$, the optimal chunk size of PBA is 17. Then, the corresponding parallel computation for each chunk in PBA requires applying only 15 times of the *max* operation over 2 elements, and this cost is much less than the cost of launching a parallel task. But, if $r = 2^{20}$, the optimal chunk size is 5242884, which requires 5242882 times of *max* over 2 elements. In the latter case, launching a parallel task can bring significant benefits.

Count-based windows with various slide sizes. Our throughput results for this case are shown in Fig. 5. We observe that, with a larger slide size, combining slicing with merging can improve throughput. PBA shows better throughput with a small slide shown in Fig. 5, because $\frac{range}{slide}$ is still large in such cases, which means there are still a large number of slice aggregation that need to be merged. Then, as the slide size is getting larger, the throughput of different algorithms are getting closer. There are mainly two reasons that explain such a behavior. The first reason comes from the fact that the ratio $\frac{range}{slide}$ is smaller in these cases, which leads to less slice aggregations to be merged for each window instance. Thus, the required computations are not enough to exploit the full capacity of PBA. The second reason is that the overall difference between various algorithms will be amortized by the cost of computing partial aggregations over each slice (this is due to the large number of $\oplus$ operations performed in each slice).

Time-based windows with various range and slide sizes. Our throughput results of this case are shown in Fig. 6. PBA shows the overall best throughput with large window sizes, and SBA (the sequential version of PBA) shows a higher throughput than TwoStack and SlickDeque. The corresponding reasons are identical to the case of count-based windows. Note that, in Fig. 6 (a), PBA outperforms the others because $\frac{range}{slide}$ is large even for the case of $range = 6$ minutes.

5.1.2 Latency Experiments. The workload consists in: *(i)* computing the aggregation Max over count-based windows with a slide of 1 and ranges of 2^{13} and 2^{14} respectively, and *(ii)* computing the aggregation MS (MomentSketch) over time-based windows with a slide of 1 second (containing around 100 streaming values) and ranges of 1 and 2 hours respectively. In this experiment, we compare TwoStack, SlickDeque, SBA and PBA. We capture the computation time in nanoseconds, corresponding to the time required by an algorithm to compute the aggregation over each window when the ends of windows are reached. We run the experiments for 1 million windows, and report the latency for the last 970 thousand ones.

Our latency results are presented in Fig. 7. In Table 2 and Table 3, we present 7 statistical summaries for the latency experiments, *i.e.*, Min (minimum), Max (maximum), Avg (average), Std (standard deviation), LQ (lower quartile, *i.e.*, 25% percentile), Med (median), HQ (higher quartile, *i.e.*, 75% percentile). On the whole, we observe that both SlickDeque and PBA do not have latency spikes, and PBA performs better than SlickDeque. We discuss detailed results of all tested algorithms in the sequel.

As it can be observed, PBA has the overall best latency results in terms of all statistical summaries, especially for the Max and Std metrics shown in Table 2 and 3, which corresponds to the worst case

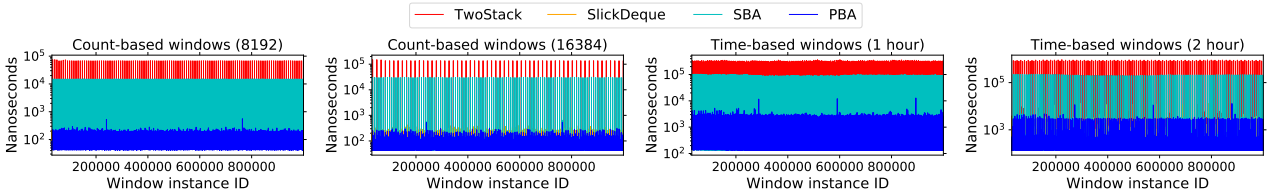


Figure 7: Latency experiments.

Table 2: latency summaries of Max over count-based windows.

Algo.	Range	Min	Max	Avg	Std	LQ, Med, HQ
TS	2^{13}	57	74543	69.21	754.3	60, 60, 61
SD	2^{13}	57	667	81.77	22.12	64, 79, 88
SBA	2^{13}	43	15160	**50.39**	233.64	**45, 46, 47**
PBA	2^{13}	**40**	**559**	53.93	**10.41**	45, 48, 65
TS	2^{14}	63	150024	77.55	1077.86	67, 68, 70
SD	2^{14}	61	647	83.07	21.86	65, 80, 90
SBA	2^{14}	43	31031	50.06	331.25	45, 46, 47
PBA	2^{14}	**42**	**537**	**46.06**	**5.24**	**44, 45, 47**

Table 3: latency summaries of MS over time-based windows.

Algo.	Range	Min	Max	Avg	Std	LQ, Med, HQ
TS	1 h	203	365582	337.65	5348.91	223, 230, 258
SD	1 h	267	13638	385.28	101.5	336, 370, 415
SBA	1 h	131	103716	194.85	1566.49	160, 163, 166
PBA	1 h	**129**	**12837**	**164.24**	**73.35**	**141, 145, 168**
TS	2 h	207	934212	381.1	9999.85	226, 235, 290
SD	2 h	266	13175	392.72	84.86	342, 380, 427
SBA	2 h	**132**	226099	201.77	2471.56	162, 164, 168
PBA	2 h	135	**12971**	**171.75**	**64.59**	**145, 150, 191**

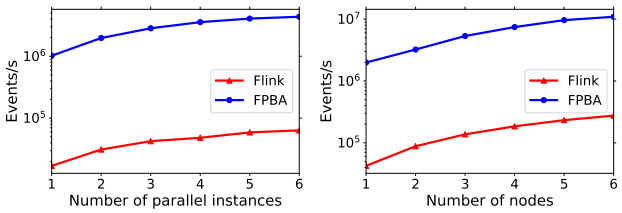
time complexity of PBA shown in Table 1. Moreover, PBA shows better latency results with a larger window size, shown in Fig. 7, and Table 2 and 3 (the reason has been discussed in Section 5.1.1). SBA shows similar results as PBA except for Max and Std in Table 2 and 3. SBA computes a *lcs* buffer at each boundary, which postpones the computation of a *csa* buffer until the computation of *lcs* has been completed. Therefore, SBA has a high latency periodically, *e.g.*, in Fig. 2, at boundary b_1, SBA needs to firstly compute lcs_0, and then csa_1, and this incurs high latency for w_1. Such a high latency of SBA can be observed in Fig. 7. Note that, PBA does not have high latency due to computing *lcs* and *csa* in parallel.

SlickDeque has almost the same latency as TwoStack for Avg, but much better than TwoStack for Max and Std. As shown in Fig. 7, TwoStack has latency spikes periodically, and thus TwoStack does not perform well in terms of Max and Std. Such latency spikes become larger when the window size increases. TwoStack requires moving all elements of one stack to another one periodically, and the corresponding cost is linearly increasing w.r.t to the window size. This is why the periodical red spike of TwoStack for the case of 16384 is higher than the corresponding one for the case of 8192, and similar for time-based window with a range of 2 hours compared to a range of 1 hour.

We also observe that, as aggregation getting more complex, the latency is also getting higher, which can be observed in Table 2 and Table 3. As an aggregation defined in the LCL framework can have an arbitrarily complex $\oplus$ operation, the difference in the latency caused by using a different merging algorithm can be more obvious.

5.2 Apache Flink Integration

We integrate PBA into Apache Flink [3, 6], denoted as FPBA. PBA is integrated into Flink through KeyedProcessFunction, *i.e.*, a low-level operator in Flink. To support parallel tasks of PBA, a thread pool for each task slot in Flink is launched, which will be used to compute the *lcs* buffers for all key-value pairs associated with the



(a) Local mode experiment **(a) Cluster mode experiment**

Figure 8: Throughput on top of Apache Flink.

corresponding task slot. Our goal in this subsection is to show that PBA can be easily integrated into modern distributed and parallel SPSs, and to evaluate FPBA in both local and cluster modes. In this experiment, we consider 2^{10} as the representative of large windows. We compute *max* over count-based sliding windows with a slide of 1 and a window size of 2^{10}, and capture the throughput of Flink and FPBA by using a different degree of parallelism in local mode, and a different number of nodes in cluster mode.

Datasets: We use the DEBS14 Grand Challenge Smart Home Dataset [16], which contains recordings from smart plugs in private households. Each tuple of this dataset includes a power consumption and a plugin ID. The first 10 million tuples contain 14 plugin IDs with a slight data skew, which we made into different sizes with the identical distribution in our experiments, *i.e.*, 60 million tuples with 84 plugin IDs for local execution, and 240 million tuples with 336 plugin IDs for cluster execution.

Experiment Setup: We run the cluster experiment at a Flink cluster having 1 master node and 6 worker nodes running Ubuntu server 16.01.3 LTS. The master node has 6 virtual CPU cores of Intel XEON 3GHz, and 16GB main memory, and the job manager of Flink is configured with 6GB of memory. Every worker node has 4 virtual CPU cores of Intel XEON 3GHz, and 8GB main memory,

and the task manager of Flink is configured with 6GB of memory. We run the local experiment only at the master node of the Flink cluster. We run both local and cluster experiments 10 times, where we consider the first 3 runs as warm-ups, and we repeat the whole execution 3 times. We report the median throughput for Flink and FPBA. We use Flink 1.10.0 and Java 8 in this experiment.

We present the results in Fig. 8. In Fig. 8 (a), we observe the throughput of FPBA or Flink linearly increases up to 3 parallel instances (called parallelism in Flink). Each parallel instance of the execution pipeline of FPBA, or Flink, contains 2 tasks, *i.e.*, reading data and computing SWAG. Therefore, up to 3 parallel instances, the task for computing SWAG can run on a dedicated CPU core at a machine with 6 cores. For more parallel instances, the throughput growth becomes less linear compared to the first phase. In Fig. 8 (b), we observe the throughput of FPBA and Flink linearly scale up w.r.t more nodes used in a cluster.

6 CONCLUSION

In this paper, we presented PBA, a novel algorithm that calculates incremental aggregations in parallel for efficiently computing SWAGs, and also its sequential version SBA. We also proposed an approach to optimize the chunk size, which guarantees the minimum latency for PBA. Our experimental results demonstrate that SBA has higher throughput than state-of-the-art algorithms irrespective of window sizes, and PBA shows best performance for large windows (*i.e.*, improving throughput up to 4× while reducing latency). Thus, a simple strategy for obtaining high performance can be using SBA for small windows and PBA for larger ones. In addition, FPBA, an integration of PBA into Apache Flink, shows significant improvement in both local mode and cluster mode.

As future works, we plan to extend PBA to support both multi-query processing and out-of-order streams. The idea is to replace the array structure of *lcs* buffers with a tree structure, where each leaf node stores a slice aggregation and any intermediate node stores partial aggregation of all its children (e.g., in the spirit of the array-based binary heap implementation of FlatFAT). In such a scenario, the major benefit of using PBA model is that any updates caused by late values can be processed only over *lcs* buffers in parallel, such that the computation for accumulating streaming values will not be delayed. One technical challenge underlying the development of such an approach lies in optimizing the chunk size that ensures the overall minimum latency, while taking into account watermarks and multiple combinations of ranges and slides.

7 ACKNOWLEDGMENTS

This research was financed by the French government IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25).

REFERENCES

[1] 2021. https://github.com/chaozhang-db/PBA.
[2] 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. *Proc. VLDB Endow.* (2013).
[3] Apache Flink. 2019. https://flink.apache.org.
[4] Apache Spark. 2019. https://spark.apache.org/.
[5] Arvind Arasu and Jennifer Widom. 2004. Resource Sharing in Continuous Sliding-Window Aggregates. In *VLDB*.
[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* (2015).
[7] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *CIKM '16*.
[8] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* (2012).
[9] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2018. Moment-Based Quantile Sketches for Efficient High Cardinality Aggregation Queries. *Proc. VLDB Endow.* (2018).
[10] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.* (1997).
[11] Vibhuti Gupta and Rattikorn Hewett. 2020. Real-Time Tweet Analytics Using Hybrid Hashtags on Twitter Big Data Streams. *Inf.* (2020).
[12] Thomas Heinze, Leonardo Aniello, Leonardo Querzoni, and Zbigniew Jerzak. 2014. Cloud-Based Data Stream Processing. In *DEBS*.
[13] Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. 2017. Sliding-Window Aggregation Algorithms: Tutorial. In *DEBS*.
[14] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* (2014).
[15] Zbigniew Jerzak, Thomas Heinze, Matthias Fehr, Daniel Gröber, Raik Hartung, and Nenad Stojanovic. 2012. The DEBS 2012 Grand Challenge. In *DEBS*.
[16] Zbigniew Jerzak and Holger Ziekow. 2014. The DEBS 2014 Grand Challenge. In *DEBS*.
[17] 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 *SIGMOD*.
[18] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-Fly Sharing for Streamed Aggregation. In *SIGMOD*.
[19] 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 *SIGMOD*.
[20] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *SIGMOD Rec.* (2005).
[21] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *SIGMOD*.
[22] Gabriele Mencagli and Tiziano De Matteis. 2016. Parallel Patterns for Window-Based Stateful Operators on Data Streams: An Algorithmic Skeleton Approach. *International Journal of Parallel Programming* (2016).
[23] Radko Mesiar Michel Grabisch, Jean-Luc Marichal and Endre Pap. 2009. *Aggregation Functions.* Cambridge University Press.
[24] Henriette Röger and Ruben Mayer. 2019. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *ACM Comput. Surv.* (2019).
[25] Anatoli U. Shein, Panos K. Chrysanthis, and Alexandros Labrinidis. 2017. FlatFIT: Accelerated Incremental Sliding-Window Aggregation For Real-Time Analytics. In *SSDBM*.
[26] Anatoli U. Shein, Panos K. Chrysanthis, and Alexandros Labrinidis. 2018. SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation. In *EDBT*.
[27] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2017. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. In *DEBS*.
[28] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-Window Aggregation. *PVLDB* (2015).
[29] Georgios Theodorakis, Peter Pietzuch, and Holger Pirk. 2020. SlideSide: A Fast Incremental Stream Processing Algorithm for Multiple Queries. In *EDBT*.
[30] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@twitter. In *SIGMOD*.
[31] Jonas Traub, Philipp M. Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing. In *EDBT*.
[32] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP*.
[33] 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. *Proc. VLDB Endow.* (2019).
[34] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. 2017. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors. In *ICDE*.