Constant-Time Sliding Window Framework with Reduced Memory Footprint and Efficient Bulk Evictions

Álvaro Villalba, Josep Ll. Berral and David Carrera
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya - BarcelonaTech (UPC)
{alvaro.villalba, josep.berral, david.carrera}@bsc.es

Abstract—The fast evolution of data analytics platforms has resulted in an increasing demand for real-time data stream processing. From Internet of Things applications to the monitoring of telemetry generated in large data centers, a common demand for currently emerging scenarios is the need to process vast amounts of data with low latencies, generally performing the analysis process as close to the data source as possible. Stream processing platforms are required to be malleable and absorb spikes generated by fluctuations of data generation rates. Data is usually produced as time series that have to be aggregated using multiple operators, being sliding windows one of the most common abstractions used to process data in real-time. To satisfy the above-mentioned demands, efficient stream processing techniques that aggregate data with minimal computational cost need to be developed.
In this paper we present the Monoid Tree Aggregator general sliding window aggregation framework, which seamlessly combines the following features: amortized $O(1)$ time complexity and a worst-case of $O(\log n)$ between insertions; it provides both a window aggregation mechanism and a window slide policy that are user programmable; the enforcement of the window sliding policy exhibits amortized $O(1)$ computational cost for single evictions and supports bulk evictions with cost $O(\log n)$; and it requires a local memory space of $O(\log n)$. The framework can compute aggregations over multiple data dimensions, and has been designed to support decoupling computation and data storage through the use of distributed *Key-Value Stores* to keep window elements and partial aggregations.

Index Terms—Internet-of-things, data analytics, big data, stream processing, real-time

1 Introduction

Stream Processing, or processing data on-the-fly, is a critical demand in many environments requiring low latency and reduced data movement. Scenarios like telemetry data analysis in large data centers, or advanced analytics for the Internet of Things (IoT), often require fast processing and aggregation of vast amounts of data. Moreover, processing data close to the source becomes an important factor when data movement is expensive due to high volume of data or poor connectivity. Due to these reasons, over the past five years a number of Stream Processing platforms have emerged, including Apache Storm [1], Apache Flink [2], Apache Samza [3] and Twitter Heron [4] as the most noteworthy open-source solutions. Furthermore, commercial solutions from the most important players in the IT industry are also offered, such as Amazon Kinesis [5], Google MillWheel [6], IBM Streams [7] and Microsoft Azure Stream Analytics [8].

Data streams are unbounded sequences of ordered atomic updates on the same information feature. E.g., a stream associated to the temperature of a physical device D contains a sequence of updates of temperature information coming from device D, each update substituting the previous one. Given that a stream emits updates indefinitely, such sequences of updates can not be traversed upstream as they do not have finite size and lack boundaries. Instead, selecting a limited window on the updates within a data stream is commonly considered the most affordable method for analyzing the data and information coming from a data source. It is for this kind of processing that projecting data from streams into sliding windows becomes a convenient mechanism towards data analysis and aggregation.

More formally, a Sliding Window is an abstraction representing a projection over a data sequence. Sliding windows are usually implemented as FIFO structures containing timestamped data updates, all of the same type. Updates enter the window when they are received from the data source, and are evicted according to a Window Slide Policy (WSP) that defines the criteria that older updates need to meet to leave in the window. Therefore, sliding windows define a contiguous sequence of strictly ordered data updates, whose length is defined by the WSP, and always containing the most recent updates generated to the moment.

Applications that process data streams usually define a set of aggregation operations that when computed produce a result associated to the stream. Due to the unbound nature of streams, sliding windows are a convenient approach to processing such aggregations, by defining the subset of updates to be considered for processing. Therefore, for their computational purpose, sliding windows are associated with at least one aggregation function, that will be computed for the contained values whenever the window content is updated.

There are two key aspects of a Sliding Window aggregation framework that define its applicability and efficiency across different scenarios:

- Firstly, the computational cost associated to the process of adding and evicting values into the structure through the WSP, and recomputing the values of the aggregations represented by the Window. Therefore, the algorithms used to operate the sliding window and the aggregations must be as efficient as possible, avoiding the computing time to grow with the window size. Naive implementations that recompute all the aggregations for every new update, thus having linear cost $O(n)$, are not able to keep up with large window sizes and high arrival rates. The Framework introduced in this paper exhibits amortized constant $O(1)$ time-complexity between updates, and $O(\log n)$ for bulk eviction, positioning itself ahead of the existing state of the art.

- The second aspect is the memory footprint of the Window data structures. Existing time efficient implementations tend to pre-allocate all the needed memory, with space cost $O(N)$ from the pre-defined maximum window size N. While this approach is convenient in terms of computational complexity, it imposes serious limitations in terms of the applicability of the technique across domains. For instance, cloud-based deployments may require extra-large VMs to host them with their associated additional cost, and Fog-based deployments will struggle to accommodate these implementations in memory-constrained edge devices. Furthermore, resizing the maximum window capacity results in a $O(n)$ time complexity operation. The Framework introduced in this paper leverages an efficient decoupling of the computation and data store through the use of a *Key-Value Store*, which results in a local space allocation of only $O(\log N)$ from the pre-defined maximum window size N, which also improves on the existing state of the art.

In this paper we introduce the Monoid Tree Aggregator (MTA) General Window Aggregation Framework, which advances the state of the art in the following aspects:

- Seamlessly combines amortized constant $O(1)$ time-complexity between updates and logarithmic $O(\log n)$ cost in the worst-case scenario

- Its data structures only need to statically preallocate space for $O(\log n)$ elements, being n its maximum capacity.

- The window aggregation mechanism and the Window Slide Policy (WSP) are user-programmable. Aggregations are described as associative operations, based on monoids, and they do not need to be invertible. The WSP, instead, defines the criteria that data to be evicted must meet.

- The WSP enforcement mechanism exhibits amortized $O(1)$ computational cost to perform single evictions on the window and $O(\log n)$ for bulk eviction operations. The mechanism is similar to performing searches on *Binary Search Trees* [9]. This aspect is of paramount importance to implement flexible WSPs, like for instance time ranges (e.g. *data accumulated in the last 5 minutes*) for a source that produces data at variable rates. This situation leads to windows containing a changing number of elements over time and mass evictions at certain moments in time.

The general purpose and efficient Sliding Window aggregation framework leveraged here could be used as an operator for Stream Processing platforms such as Apache Storm or Apache Flink. They would additionally benefit from the fault-tolerance provided by the distributed KVS based data structure.

The rest of the paper is structured as follows: Section 2 discusses the state of the art in the field of efficient Sliding Windows for Stream Processing; Section 3 introduces the main concepts related to Sliding Window frameworks; Section 4 discusses the main characteristics of the MTA Window Framework; Section 5 discusses the implementation details of the framework proposed in this work and provides the results of an experimental evaluation of the MTA Window Framework; Finally Section 6 discusses the conclusions of the work.

2 Related Work

State of the art works in the literature propose to use a FIFO structure and incremental operations to reduce the complexity of the aggregation algorithms to $O(\log n)$ and amortized $O(1)$ for variable-sized windows.

Tangwongsan et al. propose in their prior work two sliding window aggregation frameworks called *Reactive Aggregator* (RA) [10] and *Sliding-Window Aggregation* (SWAG) [11]. Having important differences between them, both approaches follow Boykin et al. [12] method of using associative operations as programmatic aggregators interface. Both RA and SWAG benefit from using associative aggregation, by enabling the computation of partial results and using the neutral element property to evict elements from their FIFO structures.

The main claim for RA is that it is $O(\log n)$ in all its operations with constant-sized sliding windows. RA's sliding window FIFO structure is a flat fixed-sized binary and complete tree called FlatFAT. Similarly to Log MTA, all the leaves are the raw updates to be aggregated, the root node is the result and the intermediate nodes are partial computations. Every update insertion and deletion propagates the aggregation changes from the leaf to the root. Other work in the literature [13]–[16] use tree-like structures in order to keep partial computations in the same way, making use of binary associative operators. They all have a worst-case $O(\log n)$ for all its atomic operations and a complexity $O(n)$ for windows with bulk evictions.

On the other hand, SWAG is a sliding window aggregation framework that runs in worst-case $O(1)$ time for each one of its atomic operations. SWAG's *insert*, *remove* and *query* operations perform in constant time with constant-sized windows. The simplified version of its main algorithm is based on a data structure with two stacks instead of a tree-like structure. One stack receives the new updates, each paired with a partial result generated by aggregating the update with the previous top partial result in the stack. The second stack is generated by reversing the order of the updates from the insertion stack and recomputing the partial results. The *reverse* operation is $O(n)$ but ends up amortized to $O(1)$ during the execution of the window. However, the *reverse* operation can be incrementally performed on *insert* and *remove* operations, turning into a worst-case $O(1)$ process if updates are removed one by one.

The time complexity is better in SWAG than in RA and similar solutions for non-invertible window aggregators, while MTA Window Framework extends major improvements from it. In first place, the window operations are logarithmic for RA-like algorithms and constant for SWAG. However, this is not considering bulk eviction as an atomic operation, and therefore it works in constant time for constant-sized windows. Constant-sized windows remove one update for each one received, keeping always the same number of aggregated updates. As only one element needs to be removed, *remove* operation complies with the $O(\log n)$ time-complexity in RA and $O(1)$ in SWAG. Yet it is a common situation to work with time-based window over a stream with an irregular input frequency. This poses a problem: Variable-sized windows like time-based windows require bulk evictions, and this operation is worst-case $O(n)$ lineal for the state of the art.

Aside from the efficiency issue that the previous point raises, such a situation makes it unfeasible to keep partial results and updates in remote data stores, as n elements might need to be retrieved for a single bulk *remove* operation. Consequently decoupling the majority of data from the local computation is not considered.

Moreover, a general mechanism for framework users to define custom sliding policies is not defined in any of the state of the art solutions.

Table 1 summarizes the comparison of RA and SWAG, with the mechanisms introduced in this work: the *Log MTA* and the more advanced *Amortized MTA*. The parameters compared include the amortized and bulk-eviction worst-case case computational complexity of the frameworks, the size of the data structure, the minimum size to be stored locally for processing the stream in the worst-case scenario, the ability to enforce user-defined Window Slide Policies, and the existence of an efficient design that supports decoupling of data and computation (e.g through the use of external key-value stores to keep part of the data).

	RA	SWAG	*Log MTA*	*Amortized MTA*
Amortized time	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
Bulk eviction time	$O(n)$	$O(n)$	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(\log n)}$
Size	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Min. local size	$O(n)$	$O(n)$	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(\log n)}$
Custom WSP	×	×	✓	✓
Data Decoupling	×	×	✓	✓

TABLE 1
Sliding window frameworks comparison

Alternative approaches to improve efficiency in sliding windows found in the literature [17]–[21] consist on keeping in memory partial aggregations from window updates instead of keeping the original updates. The result is an speed up of the aggregation and removal and also the memory needed is reduced. However, either there is a percentage of error in the number of updates evicted each time, or the algorithm knows the exact number of updates that will be removed in each iteration in order to avoid the error. The most relevant case is the Exponential Histogram from Datar et al. [20], a data structure that maintains an approximation of the number of 1s in a sliding window with logarithmic memory and time complexity. The counting is fragmented over a list, where the number of window updates counted in each list element grows exponentially from tail to head. A general purpose approximate computation similar to Exponential Histogram applied to MTA is a potential subject for future investigation, which would also improve performance and memory consumption.

Bifet & Gavaldà contributed *ADWIN* [17] and *K-ADWIN* [18] mechanisms, which implement a variation of exponential histograms. *ADWIN* is a programmable sliding window framework that automatically adapts its size by detecting changes on the data. When two subwindows have very different average values, the oldest one is evicted. The data kept in the window is considered the currently relevant data from the stream, and *guest algorithms* can perform aggregations from it. *K-ADWIN* combines *ADWIN* with Kalman filter [22], providing better results than both methods separately. *ADWIN* base algorithm can be seen as an adapted MTA WSP that compares the average value between subwindows, and the monoid aggregator as the *guest algorithm*.

Additionally, resource sharing is another methodology discussed in the literature [15], [23] to enhance performance among incremental aggregations. Although our solution is not focused on a resource sharing approach, a basic mechanism to share some resources between aggregations is also present. In this paper we introduced window aggregation multi-dimensionality, which consists in performing several aggregations on different data in the same stream, sharing resources such as the window data structure and the WSP. Experiment 4 from Section 5 shows the benefits from this approach. Tangwongsan et al. [10] already compared RA with the resource sharing focused solution from Arasu & Widom [15] positioning RA as a more advanced solution, and later SWAG [11] [24] as more advanced than RA.

3 Background: Real-Time Sliding Windows

3.1 Sliding Windows: Concept

Sliding Windows are an abstraction representing projections on data sequences, organized as FIFO structures containing elements of the same type (the data updates), and a timestamp associated to each element. Data updates enter the sliding window when they are received from the data source, and are evicted according to a Window Slide Policy defining the conditions to be satisfied by data updates for leaving the projected window. Sliding windows define a contiguous sequence of strictly ordered values, with a length depending on the slide policy, and always containing the most recently generated updates.

Therefore, the three main building blocks used by sliding windows as mechanisms to aggregate streams of data and their features are:

FIFO data structure: An update in the window is not removed until all the older updates are removed too. Contents are a complete and ordered portion of the stream being aggregated. Updates are inserted at the end and removed from the beginning of the structure.

Aggregation algorithm: Aggregations are applied to the data covered by the window in a specific moment. For instance, the aggregation could be a total sum and it could be executed every time a new update is inserted to the window. Some sliding window aggregation frameworks only accept invertible operations as aggregators, with their inverse functions. This way, the eviction of data is done in constant-time easily.

Slide policy: Updates leave the FIFO structure according to a window *slide policy* (WSP). WSP defines the conditions to be satisfied by the older updates in order to be evicted from the window. The result after applying the policy must be a subsequence including the most recent updates in the window. Traditionally WSP in sliding window aggregation frameworks are delimited by maximum window size and time-based windows. Regarding the number of updates to be removed by the policy, it is usually determined by three choices: a single update, a fixed amount of updates and all the window updates. However, a WSP can be expressed in terms of the window aggregation itself and become more rich, customizable and efficient.

3.2 Sliding Windows: Running Example

For the sake of clarity, we include here a running example of a Sliding Window used to compute the *maximum* of the values of the updates that fall within the windows according to a WSP. The WSP is complex enough to need to be expressed in terms of the window aggregation. This will provide an understanding on the need of general purpose and user-programmable WSP based on the aggregation. The WSP dictates that the window will contain the updates that add up to a value which is less or equal than 10, the rest will be removed. From the resulting window we want to extract the maximum value. For this purpose two aggregations will be used over the window: the first one will be *max*, and the other one will *sum*. The former will be used to compute the result of the operation that needs to be calculated for the window. The latter will be used to estimate the updates that have to be removed from the window after an update insertion, according to the WSP. This is also an example of multi-dimensionality of the window in terms of aggregation operations; the window could be used also considering multiple data dimensions across the window elements.

More formally, let S be a stream of ordered data and $(d_i)_{i=1}^{n}$ be the current data updates in S where i is its timestamp and n is the oldest timestamp in S. Then the WSP on the window W is:

$$\forall d_i \in S : d_i \in W \iff \sum_{j=i}^{n} d_j \leq 10$$

In this context, consider a window with the values $[2, 2, 3, 3]$, ordered in ascending order of their timestamp - that is the leftmost 2 is the oldest update in the window while the rightmost 3 is the most recent update (corresponding to d_n following the notation used above). Therefore, when a new update with value 4 is inserted to the window, the policy removes the oldest 2 updates and the result window becomes $[3, 3, 4]$. This slide policy is enforced using the sum aggregation that is calculated over the values in the stream: $3 + 3 + 4 \leq 10$. The max aggregation value would have been 3 before the last insertion, and 4 immediately after.

A naive design of these features can be achieved via the use of a simple queue that aggregates all its contents every time it is queried. The WSP enforcement pops updates until the window contents comply the policy. It clearly entails $O(n)$ to aggregate the window, n being the number of updates that are inside; hence it would quickly struggle to scale with high frequency streams and densely populated windows.

3.3 Sliding Windows: Monoids for Aggregators

A monoid is an algebraic structure with an associative binary operation and a neutral element. They are extensively used in the literature for the implementation of data aggregations, and it is the common choice for state of the art Sliding Window implementations, as it will be discussed in Section 2.

More formally, where S is a set and $\cdot$ is a binary operation, it composes a monoid if it obeys the following principles:

Associativity: $\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$
For all a, b and c in S, the expression $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ is true.

Neutral element: $\exists e \in S : \forall a \in S : e \cdot a = a \cdot e = a$
Exists a value e in S that for all a the expression $e \cdot a = a \cdot e = a$ is true.

Closure: $\forall a, b \in S : a \cdot b \in S$
For all a and b in S, the result of $a \cdot b$ is in S too.

Aggregators are then programmed in a *map* $\rightarrow$ *reduce* $\rightarrow$ *map* structure, where the first *map* transforms the input value to a member of S, the *reduce* stage is a monoid, and the last *map* converts the monoid result to the desired value out of S. For example, an average aggregator could have a monoid with S defined as integer pairs (s, c) where s is the sum of the values and c is the number of values. The monoid operation would be $(s_1, c_1) \cdot (s_2, c_2) = (s_1 + s_2, c_1 + c_2)$. The first map transforms an input value v to the pair $(v, 1)$ in S, where v is the initial sum and 1 the initial count. $(v, 1)$ is then operated by the monoid with another mapped value or a previous monoid result. The last map transforms the monoid reduced result to $\frac{s}{c}$ which is the final average.

4 Framework Design

This section describes the Monoid Tree Aggregator (MTA) Window Framework, which is the main contribution of this paper. The MTA Window Framework is an sliding window framework that aggregates values in amortized constant time between insertions, on par with the most advanced existing solutions in the literature. Additionally, it exhibits logarithmic time complexity in the worst case scenario, which includes bulk element eviction. Efficient bulk eviction is an improvement with respect to the state of the art, and it is of paramount importance for resource-constrained environments and real-time situations, like the ones considered for Edge Computing in emerging IoT scenarios. This time complexity is achieved regardless of whether the aggregation function is invertible or not. Furthermore, it provides programmable aggregation mechanism and Window Slide Policies. All this combined enables the framework to decouple most of the data aggregated from the local memory in which it is being computed, delegating this task to another

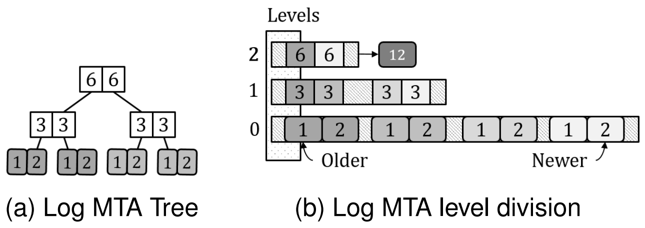


Fig. 1. Log MTA Structure and Element Location Examples

system such as a distributed data store, a local hard drive or an NVMe. For these reasons, the MTA Window Framework positions itself as a significant advance with respect to the existing state of the art solutions.

For the sake of clarity, we present the core algorithms of the MTA Window Framework in two steps: first, we describe a set of algorithms (*Log MTA* in Subsection 4.1), which create a logarithmic-time window aggregation mechanism, less efficient than the concluding MTA solution, but much simpler to explain; later, in Subsection 4.2, we extend the *Log MTA* mechanism to reduce the computation complexity to an amortized constant cost $O(1)$, in the *Amortized MTA* mechanism.

4.1 Log MTA

The Log MTA mechanism is a logarithmic-time aggregation window, used as base for the constant-time solution. It sets the foundations on the main MTA Window Framework features which are discussed in detail in this section, being: general user-programmable aggregation, efficient computation, general user-programmable WSP mechanism, data decoupling and efficient bulk element eviction.

4.1.1 Structure

The FIFO data structure in Log MTA is a binary Tree designed as a list of queues. Each queue is a Tree level, sorted in the main list from leaves (bottom level) to root (top level). The levels contain the Tree nodes grouped by pairs. The elements in the same pair are Tree siblings. A neutral element ($\oslash$) in a pair means an empty branch. The lowest level contains all the window updates in order. The levels above contain the monoid aggregation results from lower level pairs. E.g., Figure 1a shows an abstraction of the full binary Tree of a Log MTA performing a *sum* aggregation, with $[1,2,1,2,1,2]$ as data updates. Figure 1b shows its representation as a list of queues. New Tree nodes are pushed to the level queues and popped when removed, hence the FIFO behaviour. All the leaves of the Tree will be found in the first level, as stated by Invariants 1 and Invariant 2.

Invariant 1. *Let T be the Log MTA binary Tree, l_i the $i-essime$ level from leaves l_1 to root l_h, and h being the height of T. Let $(v_{ij})_{j=1}^{n} \in l_i$ be the nodes of l_i, n be the number of nodes in l_i, $\oslash$ being the monoid neutral element and $*$ any non-neutral element. Then:*

$$v_{i,1} = \langle \oslash, * \rangle \vee v_{i,1} = \langle *, * \rangle$$
$$v_{i,n} = \langle *, \oslash \rangle \vee v_{i,n} = \langle *,* \rangle$$
$$\forall_{j=2}^{n-1} v_{i,j} = \langle *, * \rangle$$

When $n = 1$ then only one of the three statements needs to be satisfied.

Invariant 2. *Having h as the height of T, $n = |l_i|$, and v_{ij} containing a pair of elements:*

$$\forall_{i=1}^{h} \forall_{j=1}^{n} v_{ij} \neq \langle \oslash, \oslash \rangle$$

Theorem 1. *Let $(e_{ijk})_{k=0}^{1}$ be each element in the pair node v_{ij}. Let $children(e_{ijk})$ be a function that returns the children pair of e_{ijk}, $\forall_{i=2}^{h} \forall_{j=1}^{n} v_{ij}$:*

$$v_{i,1} = \langle \oslash, * \rangle \rightarrow children(e_{ijk}) = v_{(i-1),(2j+k-1)}$$
$$v_{i,1} \neq \langle \oslash, * \rangle \rightarrow children(e_{ijk}) = v_{(i-1),(2j+k)}$$

The first pair of a level can be $\langle \oslash, * \rangle$ when the first element has been removed, so the node does not have left child. Also, the last pair of a level can be $\langle *, \oslash \rangle$ when its right child has not been created yet. $\oslash$ can not be found in any other position in the Tree.

Theorem 1 shows how Tree branches can be traversed, derived from Invariants 1 and 2. The greater part of the Tree traversing is performed through the first or last element of every level only, the Tree side branches. However, for bulk eviction we will need random branch traversing from root to leaf in order to find the branches to be removed.

4.1.2 Data Insertion & Aggregation

New updates are inserted to the data structure and aggregated in logarithmic time. Updates in the window are aggregated by grouping them in ordered pairs and applying a user defined monoid on each pair. The results are paired and aggregated again, in a process that is repeated iteratively until a single result is produced. This process is performed incrementally for each insertion using the binary Tree structure, as it can be seen in Algorithm 1.

When an update is inserted to the window, it is pushed at the end of the first level as a new leaf of the Tree. If the last pair in the level is $\langle *, \oslash \rangle$, the new value u is placed as $\langle *, u \rangle$, otherwise a new pair is created and added as $\langle u, \oslash \rangle$. When a new pair is added, it will not have a parent yet.

After adding the new leaf, it is aggregated with its sibling executing the user provided monoid, which Log MTA is oblivious to. The result element will be the parent of both siblings and need to be inserted in the level above. If the pair already had a parent in the level above, the parent's value needs to be updated with the new one. Otherwise, a new pair needs to be added with the aggregation result. The parent pair now needs to be aggregated, propagating the process towards the root level. If the root level now has a pair without $\oslash$, then a new level will be added that will become the new root.

This operation has time complexity $O(\log n)$, as it executes a fixed set of constant-time operations for each level on the Tree, with logarithmic height with respect to the number of updates.

4.1.3 Window Slide Policy Definition

Like the aggregation operation, the WSP is a user-programmable condition that needs to follow some rules. It has access to the total aggregation of the window after the last insertion and to the aggregation of a random subsequence from the head of the window. This aggregated subsequence is the one being checked for removal. If the condition defined by the WSP using these values is met, then *at least* this subsequence needs to be removed from the window aggregation.

Algorithm 1 Log MTA insertion & aggregation. Inserts update u to Tree T

```
1: $L \leftarrow levels(T), agg \leftarrow u$
2: for $l = 1, \ldots, |L|$ do
3:     $P \leftarrow L_{l,|L_l|}$
4:     if $agg \neq \oslash$ then
5:         if $e_1 = \oslash$ then
6:             $e_1 \leftarrow agg$
7:             $agg \leftarrow \oslash$
8:         else
9:             $L_l(enqueue(\langle agg, \oslash \rangle))$
10:        end if
11:    else
12:        $Q \leftarrow L_{l-1,|L_{l-1}|}$
13:        if $e_1 \neq \oslash$ then $e_1 \leftarrow monoid(Q_0, Q_1)$
14:        else $e_0 \leftarrow monoid(Q_0, Q_1)$ end if
15:    end if
16: end for
17: if $agg \neq \oslash$ then
18:     $L(\{\langle agg, \oslash \rangle\})$
19: end if
```

For instance, consider a window with updates that include an ordered timestamp in milliseconds and that it aggregates them with a *max* operation. Therefore, the aggregated result timestamp from a sequence of updates is the latest timestamp. If the WSP example in Listing 1 is applied to this window, its result aggregation will always use updates in the last hour. The condition compares the latest timestamp from the subsequence with the lower boundary of the WSP time frame (one hour before the last update). If the subsequence's latest timestamp is not inside this boundary, then the condition is met and it needs to be subtracted from the window aggregation.

This mechanism enables the user to define from the most basic WSP to complex and dynamic scenarios using sophisticated aggregations.

Listing 1. Window Slide Policy Example

```
function wsp(total, old){
  return
    (total.timestamp - old.timestamp) >= 3_600_000;
}
```

4.1.4 Efficient Bulk Eviction

After inserting a new update, the WSP needs to be enforced to find the longest subsequence of updates that need to be removed from the head of the FIFO structure, so a single result is produced. This process takes advantage from the binary Tree based data structure and it is performed in$O(\log n)$. Furthermore, the time complexity is the same for both removing a single update or performing a bulk update eviction from the window.

The importance of performing efficient bulk update evictions from a window resides in the concept of variable-sized windows in contrast with constant-size windows. Constant-size windows remove one update for each one received, keeping always the same number of aggregated updates. However, it is a common situation to work with time-based window over a stream with an irregular input frequency. This poses a problem: after inserting $k \geq 1$ data updates to the window, k updates might need to be evicted at once, triggered by the time based slide policy. In the state of the art, all the updates need to be traversed and possibly removed, multiplying by n the time complexity of removing a single update. Variable-sized windows like time-based windows make the most of performance improvements on

Algorithm 2 Log MTA WSP enforcement on Tree T with efficient bulk eviction

```
1: $L \leftarrow levels(T), sub \leftarrow \oslash, rm \leftarrow 0$
2: for $l = |L|, \ldots, 1$ do
3:     $P \leftarrow L_{l,1}$
4:     if $rm > 0$ then
5:         $L_l(remove\_pairs(rm))$
6:         $rm \leftarrow 2 \cdot rm$
7:         if $e_0 \neq \oslash$ then $rm \leftarrow rm - 1$ end if
8:         $P \leftarrow L_{l,1}$
9:     end if
10:    if $e_0 \neq \oslash$ then
11:        $subseq \leftarrow monoid(rm\_subseq, e_0)$
12:        if $wsp(subseq, result(T))$ then
13:            if $l = |L| \wedge l \neq 1$ then $L(remove(l))$
14:            else if $e_1 = \oslash$ then $L_l(remove\_pairs(1))$
15:            else $e_0 \leftarrow \oslash$ end if
16:            $rm\_subseq \leftarrow subseq$
17:            $rm \leftarrow rm + 1$
18:        end if
19:    end if
20: end for
21: $P \leftarrow L_{1,1}$
22: $agg \leftarrow monoid(e_0, e_1)$
23: for $l = 2, \ldots, |L|$ do
24:     $P \leftarrow L_{l,1}$
25:     if $e_0 \neq \oslash$ then $e_0 \leftarrow agg$ else $e_1 \leftarrow agg$ end if
26:     $agg \leftarrow monoid(e_0, e_1)$
27: end for
```

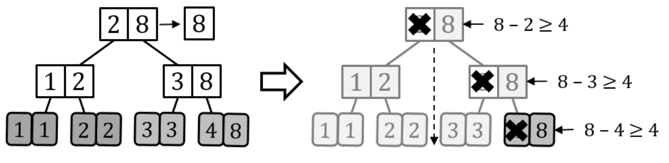


Fig. 2. Log MTA Bulk Eviction. Monoid: $max(x, y)$; WPS: $total - old \geq 4$.

bulk evictions, especially on situations in which real-time aggregation is required.

The WSP enforcement performs a $O(\log n)$ root to leaf search in the binary Tree for the oldest valid update in the window, while pruning invalid branches guided by the user-defined WSP. Algorithm 2 is a detailed specification of this operation. The Tree levels are traversed from root to leaves executing the WSP with the first level value as the aggregated value of its leaves subsequence. If removed, the node's branches will be evicted from the next levels before running the WSP on the new first element. As the *remove_pairs* function in Algorithm 2 only updates the pointer to the first pair of the level queue, it has constant time. The elements can be removed in the background by a garbage collector, minimally affecting the process of getting a result aggregation. When this process is finished, the nodes in the leftmost branch might not be consistent and have aggregated values that have been removed. Therefore, the leftmost branch is recomputed bottom-up, propagating the value changes to the root pair and resulting in a valid aggregation result.

A running example of this process can be found in Figure 2, where timestamps are inserted to the window and the WSP only allows a window of 4 time units. When timestamp 8 is inserted it triggers the WSP enforcement to evict all the other updates in the window in three steps. It checks the first valid element of each level from root to leaves with the WSP. All of them are found out from the window, leaving only the newest update.

Time complexity is $O(\log n)$, as this operation performs a fixed number of constant-time operations for each level of

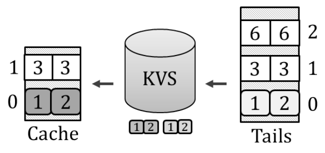


Fig. 3. Log MTA KVS data structure

the Tree, by visiting them twice.

Once the WSP has been enforced, the aggregation result can be queried to the window. This operation returns the aggregation of the window contents in constant time, by returning the monoid result in the root pair. Furthermore, if we require a reactive behaviour from the window, then the following pipeline needs to be executed when a new update arrives: *insert update* $\rightarrow$ *enforce WSP* $\rightarrow$ *query result*. Every time a new update is introduced, it produces the result in logarithmic time.

4.1.5 Reducing Local Memory Footprint

From inserting a new update to generating a new result, Log MTA needs to traverse at most $O(\log n)$ elements: for each level queue, the tail element and probably an element near the head. Therefore, the rest of the data in the window does not need to be waiting in local memory and the resources could be used to run other aggregations. In the worst case, a bulk eviction will need to traverse a Tree branch that is not currently in local memory, which will require an immediate memory retrieval of only $O(\log n)$ elements.

In the proposed data structure, each level queue has three sections between two different memory layers, as it can be seen in Figure 3. The pairs in the tail of the level queues are in the *Tails* list in local memory, the central pairs wait in a shared *Key-Value Store* (KVS), and the pairs in the queues' head can be found in a *Cache* in local memory again.

The rightmost branch of the Tree is found in the Tails list. It receives updates as they are being inserted to the structure, and pushes the replaced pairs to the KVS. The pairs pushed to be sent to the KVS are first kept into a buffer in order to reduce the number of interactions with the data store. When the maximum capacity of the buffer is reached, all its contents are moved to the KVS. The key in each KVS document maps its contents to its Tree level and its position it has inside the queue, so a $O(1)$ single pair retrieval can be achieved. The store can be anything from a local HDD file to a remote and dedicated cluster. Finally, the Cache contains at least the head pair from each level queue, with the exception of the root level. The Cache is refreshed from the KVS and the buffer when its size is under an specific threshold or in a Cache miss situation. Its capacity can be adapted to reduce the interactions with the KVS.

Having the data stored by an external entity, apart from the scalability enhancement it provides, makes it easier to recover aggregation data from a failure.

4.2 Amortized MTA

In this section we present the Amortized MTA (AMTA) sliding window mechanism. It is an approach aimed to reduce Log MTA's time complexity without having an impact deteriorating its other benefits in comparison with the state of the art: space complexity, its user-programmable WSP mechanism, efficient bulk evictions and the reduced local memory footprint.

4.2.1 Structure

Amortized MTA is an sliding window mechanism that inserts, aggregates and removes elements in amortized constant time, with logarithmic time in the worst case. It satisfies Log MTA Invariants 1 and 2, and shares the data structure level division and the memory layers, although the data structure operates differently. In the new data structure, the Tree is replaced by a Forest of binary trees where the rightmost pair of each level is the root of its own tree, as defined in Invariant 3. Figure 4a is an example of an Amortized MTA window performing a *sum* aggregation with the updates $[1, 2, 1, 2, 1, 2, 1, 2]$. The lower level contains the values in the window to be aggregated, while the levels above contain partial aggregations of these values. Considering Invariant 3 now, Theorem 1 is also valid as a tree traversing guide.

The data structure also introduces the Stack and the Result Pair, as it can be seen in Figure 4b example. As an addition to Tails and Cache, they are the parts of the structure required to be local memory at all times. E.g., Figure 4c shows the memory distribution of the data structure.

Result Pair ($R = \langle R_0, R_1 \rangle$) maintains the aggregated result from the leftmost tree in R_0 and the aggregated result from the rest of the Forest in R_1. The Stack contains the aggregated results of the leftmost tree without the first element from each level. The top value from the Stack is always R_0 minus the update in the head of the window. In Figure 4b we can see that the Stack top element is 5, which is R_0 minus the head element in the first level: $6 - 1 = 5$. Likewise, the next element in the Stack is 3, which corresponds to R_0 minus the head element in the second level: $6 - 3 = 3$.

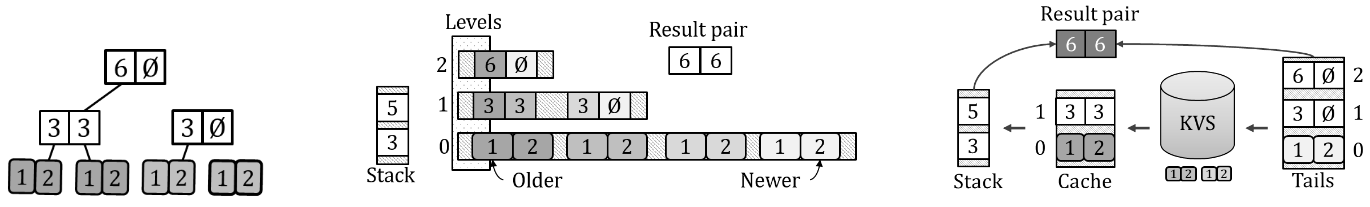
Essentially, AMTA insertions aggregate the new values in R_1, while single update evictions pop values from the Stack onto R_1. For instance, inserting 1 to the data structure in Figure 4 would result on $R_1 = 6 + 1 = 7$, while evicting the first element would pop 5 from the Stack and put it on R_0. Aggregating R always produces the final result value for the window. The Forest is used to keep the Stack updated, to compute R_1 from scratch when necessary and to perform bulk evictions. Therefore, new updates also need to be inserted and removed from the Forest structure.

The main goal of AMTA is to improve its time complexity without giving up its other benefits in comparison with the state of the art: space complexity, its user-programmable WSP mechanism, efficient bulk evictions and the reduced local memory footprint. More details on how this is achieved using this structure can be found in the following sections.

Invariant 3. *Having h as the height of T and l_1 the leaf level for all the binary Forest, $n = |l_i|$, and v_{ij} containing a pair of elements, $\forall_{i=1}^{h} v_{i,n}$ is a tree root.*

4.2.2 Amortized insertion

Log MTA update insertion is $O(\log n)$ because for every new update, the Tree nodes need to be updated from the leaf to the root. This can be avoided by only adding a node to the Tree when its value is definitive. In other words, a pair will only have a parent if both members in the pair have been inserted. In AMTA, a pair will only have a parent if it is not in the tail of its level queue. Figure 4a example shows that the last $\langle 1, 2 \rangle$ pair in level 0 is not aggregated in



(a) AMTA Forest (b) AMTA level division (c) AMTA KVS data structure

Fig. 4. Amortized MTA Structure and Element Location Examples

Algorithm 3 AMTA update insertion. Inserts u to data structure C

```
$L \leftarrow levels(C), R \leftarrow result\_pair(C)$
$agg \leftarrow u, l \leftarrow 1, h \leftarrow max(|L|, 1)$
$R_1 \leftarrow monoid(R_1, u)$
while $l \leq h \wedge agg \neq \oslash$ do
    $P \leftarrow L_{l,|L_l|}$
    $next\_agg \leftarrow \oslash$
    if $e_1 \neq \oslash$ then
        $next\_agg \leftarrow monoid(e_0, e_1)$
        $L(\langle agg, \oslash \rangle)$
    else
        $e_1 \leftarrow agg$
    end if
    $agg \leftarrow next\_agg$
    $l \leftarrow l + 1$
end while
if $agg \neq \oslash$ then
    $L(\{\langle agg, \oslash \rangle\})$
else if $l > h$ then
    $C(compute\_left\_result())$
    $C(compute\_right\_result())$
end if
```

level 1 and there is a $\oslash$ at its tail instead. The consequence in the shape of the data structure is Invariant 3, the tail pair on each level is the root of its own binary tree. This process is amortized $O(1)$ and $O(\log n)$ in the worst case.

However, the pair in the upper level does not contain the full aggregation result, only a part of it. Therefore, every inserted update is aggregated in R_1 ($O(1)$), which contains the aggregation of all the trees except the leftmost one. R_0 contains the aggregated result of the leftmost tree, so the aggregation of R is the full window aggregation result.

As the window grows, R_1 aggregated trees merge with the leftmost tree. In this situation, part of the aggregation moves from R_1 to R_0. Therefore, R and the Stack need to be recomputed from scratch, which has an amortized $O(1)$ time complexity with $O(\log n)$ in the worst case.

Algorithm 3 describes the operation more formally. The new update u is firstly aggregated to R_1, overwriting its value to keep the Result Pair up to date. Then, u is inserted in the Forest's first level queue and the aggregation is propagated up to its tree root. Finally, when an element is inserted to the already existing highest level, R must be recomputed from scratch using *compute_left_result* for R_0 and *compute_right_result* for R_1, both $O(\log n)$.

compute_left_result places into R_0 the aggregation of the leftmost tree while repopulating the Stack, as described in Algorithm 4. The head pairs from each level queue are traversed, from root to leaf. When a pair P is $\langle *, * \rangle$, its element e_1 is aggregated with the top of the Stack (or with $\oslash$ if the stack is empty), and then stacked. Once all levels have been visited, the older update in the window is aggregated with the top of the Stack, and placed in R_0. The contents in

Algorithm 4 Compute AMTA R_0 and Stack S in the data structure C

```
$S \leftarrow stack(C), L \leftarrow levels(C)$
$S(clear()), R \leftarrow result\_pair(C)$
for $l = |L|, ..., 1$ do
    $P \leftarrow l_{l,1}$
    if $e_0 \neq \oslash \wedge e_1 \neq \oslash$ then
        $S(push(monoid(e_1, S(peek()))))$
    end if
end for
if $e_0 \neq \oslash$ then $R_0 \leftarrow monoid(e_0, S(peek()))$
else $R_0 \leftarrow monoid(e_1, S(peek()))$ end if
```

Algorithm 5 Compute AMTA R_1 in the data structure C

```
$L \leftarrow levels(C), R \leftarrow result\_pair(C)$
for $l = 1, ..., |L|$ do
    $P \leftarrow L_{l,|L_l|}$
    $R_1 \leftarrow monoid(monoid(e_0, e_1), R_1)$
end for
```

the Stack will be used to perform eviction of single updates in amortized constant time.

compute_right_result operation aggregates into R_1 all the rightmost pairs (except for leftmost tree) in the Forest, as described in Algorithm 5.

The continuous execution of an update insertion in the Forest makes each element in the data structure to be visited once for the bottom-up propagation. As the space used for the data structure is $O(n)$, the cost of inserting n updates becomes $O(n)$. Then, functions *compute_left_result* and *compute_right_result* affect only $O(\log n)$ in a whole round of n elements insertions, complexity remaining $O(n)$ for inserting n updates. So, the amortized cost for aggregating 1 update to the window becomes $O(1)$.

4.2.3 *Single update evictions*

In Log MTA, performing a bulk eviction is a $O(\log n)$ operation, and it is the only option to remove any number of updates from the window aggregation. Removing a single update from its data structure and propagating the changes on the head of each level would have the same logarithmic cost. To amortize this cost, the solution we followed for AMTA is to find a way to perform amortized constant single update evictions and to save bulk evictions for when the number of elements to be removed is equal or greater than a factor of $\log n$.

The Stack from AMTA's data structure is the key element to achieve an amortized constant time single update eviction. It contains the future R_0 values after removing the head element from each level, being the oldest update removal always in the Stack's top. The rest of the elements will be used at some point, both for single update evictions and to maintain the Stack in amortized constant time.

The single update eviction operation is formally described in Algorithm 6. The Results Pair R is updated by popping an element from S into R_0. At this point, R aggregation already provides the correct aggregation result, but the Forest and the Stack need some maintenance before removing the next update.

The first update is removed from the head of the leaves level in the Forest, and the parent-child relations in the branch are updated. If a pair is removed from the Forest, then its parent element must be replaced by $\oslash$. However, the tree aggregations will not be updated, leaving inconsistent values in the data structure. The main implication of only updating the branch parent-child relations instead of also updating all the values is that, while it still keeps the tree consistent with Theorem 1, the amortized cost is constant and not logarithmic. During this process, all the new head pairs from each traversed level are pushed in the new_heads stack.

The current Stack top element might not be the next R_0. The Stack needs to be updated with new elements, using *update_stack*, which can be found in Algorithm 7. Similarly to *compute_left_result*, it updates the stack using values from new_heads. For every pair P popped from new_heads, its element e_1 is aggregated with the top of the Stack (or with $\oslash$ if the stack is empty), and then stacked.

If the number of levels has decreased after this process, the first tree has been completely removed and the second one took its place. Therefore R_1 needs to be recomputed.

Figure 5 shows an example of this situation. It is a window performing a *sum* aggregation on the sequence $[1, 3, 2, 1, 2, 1, 1, 0, 3, 1]$ with result 15. When the first update is removed, the top of the Stack (6) is moved to R_0 and the update is replaced by $\oslash$ in the Forest. The result is now $6 + 8 = 14$, which corresponds to $15 - 1 = 14$. No further actions are required after this update removal. The same steps are followed for the second update removal, but in this case the head pair from the first level is removed and the head element from the second level is replaced by $\oslash$. Also, $\langle 2, 1\rangle$ is used to update the Stack ($1 + \oslash = 1$).

The continuous usage of this operation results in each element being removed, without updating any value. Furthermore, each pair is traversed once to update the Stack and *computation_right_result* affects only $O(\log n)$. Therefore, the amortized cost for a single removal from the window is $O(1)$.

This process does not make use of the inverse functions of the aggregation operation to subtract the evicted updates, which would run in worst-case $O(1)$ time. For example, if we sum $[1, 2, 3]$ the result would be 6. When evicting 1, we could use the inverse function with result $6 - 1 = 5$ in one step. The problem is that the inverse function does not always exist or is easy to find. AMTA single eviction mechanism provides an equivalent computational cost with a less restrictive aggregation programming interface.

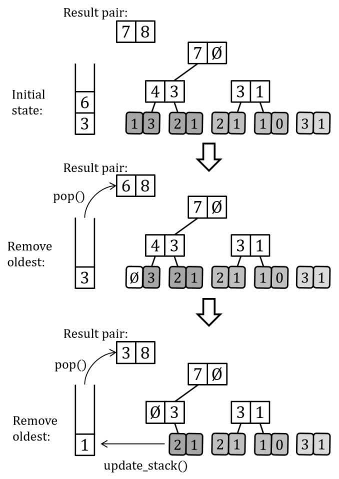


Fig. 5. AMTA single update eviction running example

Algorithm 6 AMTA's single update eviction from the data structure C

```
1:  $S \leftarrow stack(C), L \leftarrow levels(C), l \leftarrow 1$
2:  $removed\_pair \leftarrow true, new\_heads \leftarrow \{\}$
3:  $R_0 = S(pop())$
4:  while $removed\_pair \wedge l \leq |L|$ do
5:      $P \leftarrow L_{l,1}$
6:      if $removed\_pair \leftarrow (e_0 = \oslash \vee e_1 = \oslash)$ then
7:          $L_l(remove\_pairs(1))$
8:          $P \leftarrow L_{l,1}$
9:          if $e_0 \neq \oslash \wedge e_1 \neq \oslash$ then
10:             $new\_heads(push(P))$
11:         end if
12:     else
13:         $e_0 \leftarrow \oslash$
14:     end if
15:     $l \leftarrow l + 1$
16: end while
17: $C(update\_stack(new\_heads))$
18: if $removed\_pair$ then
19:     $C(compute\_right\_result())$
20: end if
```

4.2.4 Amortizing Bulk Evictions

Enforcing the WSP, like in Log MTA, removes updates from oldest to newest while the WSP is satisfied. In this case, the WSP enforcement starts by checking the head update of the window. If the WSP condition is met, the update is removed using the single update eviction operation. For constant-size sliding windows, this solution already runs in amortized constant time with logarithmic time in worst case scenario.

However, the worst time would become linear with variable-size windows. Our solution is to use the bulk eviction after the WSP enforcement removed a factor of $\log n$ elements from the Forest using the single update eviction. Theorem 1 is valid for each tree in the Forest, and each tree is removed from its root. Therefore, Algorithm 2 can be applied to each tree in the Forest until reaching the leaves level and keeping the same cost. The only precondition is to recompute the values from the leftmost branch, in order to make them consistent, which is a $O(\log n)$ process.

5 Evaluation

The evaluation is divided into four experiments concerning different aspects from the MTA Window Framework and state of the art general sliding window solutions.

The analysed algorithms correspond to implementations of Amortized MTA, Log MTA, *DABA* and *Naive* window aggregation. *DABA* is the featured algorithm from the state of the art SWAG framework [11], discussed in Section 2. All *DABA* operations are $O(1)$, but it does not feature a bulk eviction mechanism. Therefore, performing an eviction of n elements is $O(n)$, which is amortized with a higher worst-case than AMTA. On the other hand, the *Naive* approach

Algorithm 7 AMTA Stack update. Updates S from new_heads in the data structure C

```
1: $S \leftarrow stack(C)$,
2: while $|new\_heads| \neq 0$ do
3:     $P \leftarrow new\_heads(pop())$
4:     $S(push(monoid(e_1, S(peek()))))$
5: end while
```

aggregates all the elements in the window every time a new result has to be produced.

All algorithms use monoids as the aggregation mechanism, so we are evaluating sliding window algorithms that do not need to have invertible aggregations. Additionally, we will use MTA's WSP mechanism in all the algorithms, with an adapted WSP enforcement. Both *DABA* and *Naive* will use the head element in the window individually as the subsequence to compare in the WSP, because they don't have efficient bulk eviction mechanisms.

5.1 Implementation

All algorithms are implemented in Java 1.8 and executed as operators in an *Apache Storm* based stream processing runtime called *rapids*. *rapids* processes all data units as objects with a shared class and several data dimensions, meaning that updates and partial results will be objects with multiple values rather than single scalar values. The purpose of running the algorithms in *rapids* rather than isolated is to show how they perform in a production environment.

MTA Window Framework will be evaluated in two different implementations: one where the algorithm's data structure resides in pre-allocated local memory, and another with the KVS-based data structure described in the previous sections. The local memory implementations of MTA replaces each level's Cache-based KVS interaction mechanism by a simple CircularFifoQueue. They compare on equal terms with *DABA* and *Naive* aggregation, as neither of them have a data structure adapted to work with remote data stores. For these algorithms, the data structure is preallocated and never reallocated, to avoid evaluating the latency added by performing incremental memory allocation strategies or static resizing. Furthermore, *DABA* implementation contains the optimizations described by its authors regarding caching results (*Cached DABA*). They have been evaluated on its most favourable implementation for the *rapids* runtime. The MTA local memory implementations will be referred as *Mem. LMTA* and *Mem. AMTA*, while the memory decoupled versions will be *KVS LMTA* and *KVS AMTA*.

All tested algorithm implementations include a WSP enforcement mechanism. For Amortized MTA and Log MTA, the WSP enforcement algorithms are the ones described in Section 4, including the $O(\log n)$ bulk eviction. *DABA* and *Naive* aggregation WSP enforcement check the first elements in the window, one by one, as the algorithms themselves do not have the capability to perform efficient bulk evictions.

KVS LMTA and *AMTA* buffer up to 512 new elements from the data structure before storing them to a distributed data store. Each level have a cache containing up to 512 elements retrieved from the data store. When a level cache size is less than 256, it synchronizes with the data store to fill it up if possible, depending on the size of the level. Moreover, the data store used in the experiments is Couchbase [25]. Couchbase is a KVS based on memcached [26], with a distributed LRU cache in RAM. It prioritizes access in memory over disk for low-latency.

Naive window aggregation consists of a fixed size circular queue. When an update is inserted or removed to the window, it is simply inserted or removed from the queue. Querying the result aggregates all the updates contained in the queue, if it does not have the result already cached.

5.2 Optimizations

On top of the main algorithms that were previously explained, some optimizations were used for the evaluation. Those were not included in the description of the main algorithms for the sake of simplicity.

Aggregation results are cached for all the algorithms evaluated. While a cached result value is valid, no computation needs to be performed to produce a result. After a new insertion or eviction from the window, the cached result is flagged as invalid and the aggregation final result will need to be computed.

In Amortized MTA, both an update insertion and the WSP enforcement might trigger a full Result Pair recomputation. It can happen that the arrival of a new update triggers a full result pair recomputation twice, if both the insertion and WSP enforcement require so. In order to avoid such a situation, result pair recomputations are requested by each stage, but they are executed only once after the operations finished.

KVS LMTA and *AMTA* communication with the data store is done in the background when it is possible. Storing the buffered elements is always a background operation. However, although updating a level cache is also performed by background threads, a cache miss will always require a synchronous update.

DABA contains all the optimizations defined in its corresponding papers (*Cached DABA*).

5.3 Environment

The experiments were run in a cluster with 2-way Xeon E5-2630 (broadwell) v4 clocked at 2.20GHz nodes. Each one features 128GB of DDR4-2400 R ECC RAM. All nodes were interconnected using a non-blocking 10GbE switching fabric. Although an external NFS folder was mounted on the systems, it was not used as a backend for the experiments. Instead, all data was stored locally using four 7.2K rpm 2TB SATA HDDs per nodes, mounted as four independent volumes. Experiments comprising *Naive* aggregation, *DABA*, *Mem. AMTA* and *LMTA* only used a single node. *KVS AMTA* and *LMTA* logic was executed in a single node, but Couchbase ran as a cluster in three extra nodes. Therefore, the contents of both algorithms data structures were distributed between 4 nodes.

Listing 2. Experiments' monoid

```
function monoid(left, right){
  Element result = new Element();
  result.count = left.count + right.count;
  result.maxSize = right.maxSize;
  return result;
}
```

Listing 3. Experiments' WSP

```
function wsp(total, old){
  return total.count - old.count >= total.maxSize;
}
```

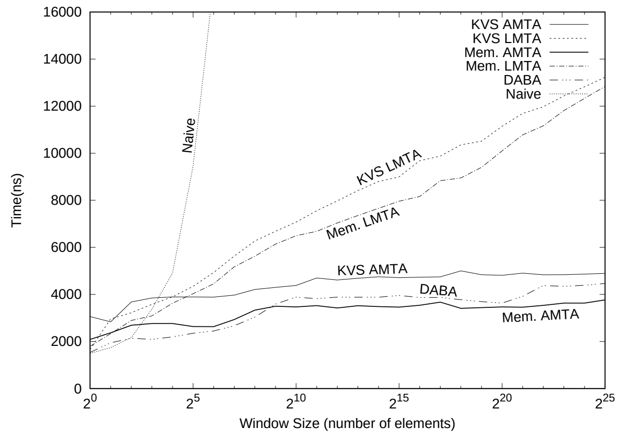


Fig. 6. Average latency for constant-sized windows

5.4 Experiment 1: Constant-sized window latency

In this experiment we analyze the average latency of inserting a new update and generating a result with a constant-sized window. Its aim is to demonstrate the effective time complexity of each algorithm, and how they compare to each other. Each measurement was performed for different window sizes by inserting one update to the window, removing the oldest one, and retrieving the total aggregation. The user defined operations for this experiment are the monoid in Listing 2 and the WSP in Listing 3. Updates and partial results contain two dimensions: *count* and *maxSize*. *count* is always 1 on an update inserted to the window, as it counts itself. *maxSize* establishes the size of the window, and so it is used by the WSP to remove updates from the window when this size is exceeded. The evaluated window sizes go from 1 to 2^{25}. Each iteration of the experiment starts by filling the window up to *maxSize*. Once the window size is *maxSize*, update insertions are performed until all the initial updates from the filling up stage are removed by the WSP, hence traversing all the window possible states. The latencies shown in Figure 6 for each window size correspond to the average latency of the process triggered by an update insertion, including aggregation, WSP check and update removal. The chart is drawn in a logarithmic scale for the x-axis for clarity.

As it can be observed, *Naive* aggregation initially has the lowest latency, but it grows linearly with the window size and rapidly becoming the obvious worst-performant algorithm in terms of time complexity.

As it was expected, *AMTA* and *DABA* show a constant time complexity behaviour. Being *Mem. AMTA* the algorithm with the lowest latency with a window size 2^9 or greater, its distance with *KVS AMTA* is relatively low and affordable given the memory usage benefits. The impact on storing the majority of the data in a distributed data store is around 1 microsecond with the greater window sizes and less than 500 nanoseconds compared to *DABA*. This is the result of keeping data store communications asynchronous when possible. The same difference can also be appreciated in the *Log MTA* implementations, which has a the expected $O(\log n)$ behaviour.

This experiment proves that the theoretical complexity for constant-sized window is also shown in practice. Furthermore, the average AMTA latency for constant-sized windows goes in line with the state of the art, and the data-computation decoupling performed in *KVS AMTA* and *LMTA* have marginal a effect for constant-sized windows.

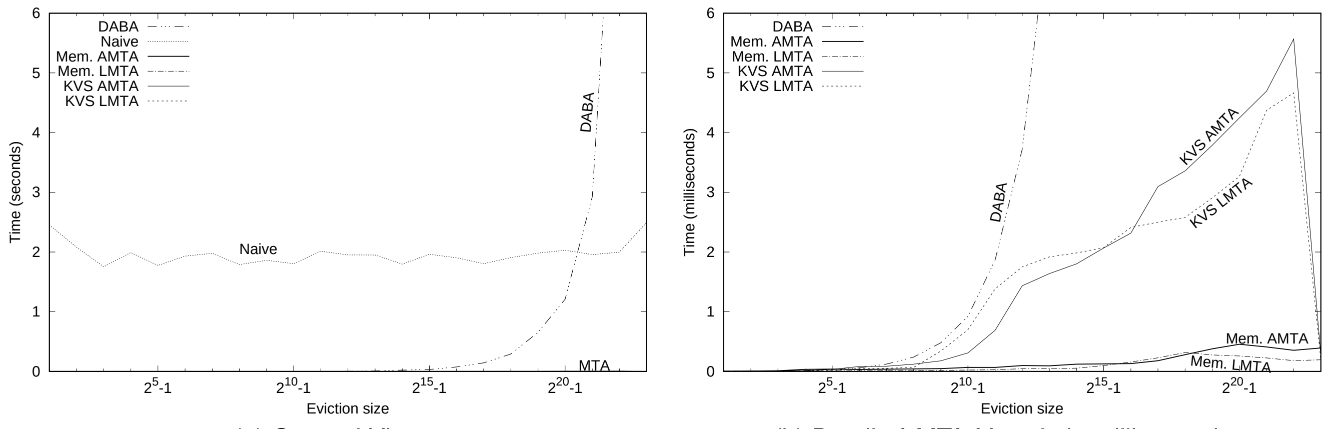
5.5 Experiment 2: Bulk eviction latency

This experiment evaluates the variable-sized windows scenario. In these cases, several updates need to be evicted from the window triggered by a single new update insertion. Using the monoid in Listing 2 and the WSP in Listing 3, we measured the average latency of the *enforceWSP* operation for each algorithm. The windows are initialized with the same initial size: 2^{23} updates. A series of iterations evict from 1 to $2^{23} - 1$ updates per insertion, averaging its latencies for each removal size. The results can be seen in Figure 7, divided in two different y-axis crops to visualize distinct groups of results, one in seconds and the other in milliseconds. The x-axis have a logarithmic scale.

Figure 7a is the global view and emphasizes *DABA* and *Naive* windows. *Naive* aggregation bulk eviction latency is around 2 seconds constantly. All updates in the *Naive* window are aggregated when generating a result. On the one hand, it needs to aggregate all the updates checked by WSP after an insertion. On the other hand, it also needs to aggregate the remaining elements to produce a result for the operation. Therefore, the number of aggregated elements remains constant. Furthermore, *DABA* has a clear linear latency growth behaving worst than *Naive* when removing sub-windows with size $2^{20} - 1$ or greater, and becoming an unfitted operation for real-time stream processing.

Figure 7b reduces the y-axis scale by three orders of magnitude, and *Naive* window is now out of the scope of the chart. It focuses on comparing the four *MTA* solutions and *DABA*. Bulk eviction latencies are very similar between the *KVS MTA* implementations, growing logarithmically. *KVS LMTA* is a almost a millisecond faster for most periods as its WSP enforcement process has the same complexity but fewer stages, i.e. trying multiple single update evictions. In this scenario, they suffer from the greatest impact of having the majority of the data in a distributed data store. The consistent latency growth from *KVS MTA* compared to the *Mem. MTA* counterparts is due to the data store query time, triggered by cache misses. However, the latencies decrease significantly in the last iteration, because the data can be found locally in the structure buffer. Maximum latency for both algorithms is around 5 milliseconds, 400 times less than *Naive* window running completely in local memory. The effect of having most of the data structure in a *KVS* is noticeable by comparing them with *Mem. AMTA* and *LMTA*. The *Mem. MTA* algorithms have the best time performance: *Mem. AMTA* has 455 microseconds worst latency and *Mem. LMTA* 258 microseconds. Note that *Mem. LMTA* also performs better for the greater part of the iterations than *Mem. AMTA*, like in the *KVS* scenario.

AMTA Framework shows a significant improvement compared to the state of the art for bulk window evictions. For big window bulk evictions, even *KVS MTA* behaves faster than the memory allocated *DABA*, while the *Mem. MTA* solutions is faster throughout the execution. The latency/memory tradeoff offered by *KVS MTA* is demonstrated later on.



(a) General View

(b) Detail of *MTA*: Y scale in milliseconds

Fig. 7. Window bulk eviction average latency, using different y-axis scales to show different details

	Naive	KVS LMTA	Local LMTA	DABA	KVS AMTA	Local AMTA
Sum	$3.69 \cdot 10^8$	14 320	10 341	4 288	6 253	4 163
Mean	$3.26 \cdot 10^8$	14 027	10 378	4 389	6 111	4 033
G. Mean	$2.83 \cdot 10^8$	15 267	11 166	4 795	6 198	4 183
Std. Dev.	$3.5 \cdot 10^8$	15 439	12 934	4 880	6 131	3 864
Max	2 554	8 501	6 188	3 500	2 886	1 763
LIS	19 294	22 306	19 794	10 027	7 350	6 476

TABLE 2
Window latencies in nanoseconds with different monoids and WSPs

5.6 Experiment 3: Stream analytics latency

The previous experiments show how the different algorithms behave in terms of latency. In this experiment we evaluate how different real window aggregations behave with each algorithm. The analysed stream consists on 62 208 000 updates monitoring computer memory usage, one reading per second for two years.

This stream has been subjected to different operations performed by the window monoid: *sum*, *mean*, *geometric mean*, *standard deviation*, *maximum* and *longest increasing subsequence* (LIS). The particular case of *LIS* is the most complex one, since it measures multiple dimensions: initial timestamp, final timestamp, interval covered by the subsequence, and the number of updates in the subsequence.

In terms of WSP, there is a general rule for all the operations: the window contains at most 2^{20} elements. This policy alone makes the window static-sized. However, *max* and *LIS* extend the size limit policy: *max* operation evicts the older subwindow not containing the maximum value in the window, and *LIS* operation evicts the older subwindow not containing any portion of the LIS. Updates older than a max value or a LIS are never going to contain a future new result, it will only be found within newer updates. Therefore, these updates are not necessary to perform the aggregation and the memory they are using can be cleared. By doing that, an efficient bulk eviction mechanism can reduce the total time of evictions performed during the whole data stream analysis.

Table 2 shows the mean latency in nanoseconds for each operation and sliding window algorithm. All operations run faster in Local AMTA than in the other algorithms. In DABA they behave slower but similar to Local AMTA, except for *max* and *LIS*, where the difference is more noteworthy. Both operations clearly benefit from reducing the number of single evictions in both KVS and Local AMTA, getting better performance than executed in DABA. These operations also perform well in the Naive algorithm, being Naive the second best algorithm to run *max*. The cost of an insertion in the Naive algorithm without evicting any update is as cheap as performing a single monoid execution, while the evictions cost is very expensive but constant for evicting any number of updates (Figure 7a).

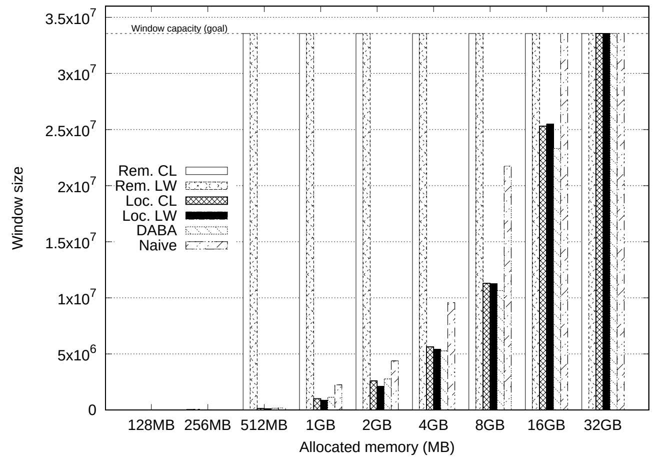


Fig. 8. Average window size reached per allocated memory amount, for a 2^{25} updates capacity.

In this experiment we proved that the performance and time-complexity exhibited in the previous experiments has a relevant impact in different stream analytics on real data. Furthermore, the experiment tests multiple distinctive monoids and WSPs, analysing their impact rather than testing only the algorithms with a minimal aggregation. It shows consistency with the theoretical complexity of each algorithm and their tested performances.

5.7 Experiment 4: Memory requirements

This experiment evaluates the local memory requirements in order to run each sliding window algorithm in *rapids*. As previously introduced, *rapids* is a stream processing platform written in Java. For this experiment, we assigned different memory heap sizes for the Java Virtual Machine (JVM), up to 32GB; and for each size, the sliding window algorithms were executed individually, with capacity for 2^{25} updates.

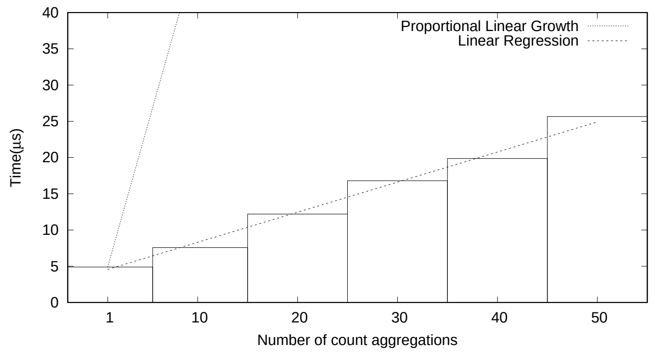


Fig. 9. Average latency for multiple aggregators

There are three possible outcomes for each execution: In the first one, the window is filled up and older updates start to be removed, showing a normal behavior. In this case the window size reached is its capacity and the goal is met. In the second one, *rapids* runs out of memory as the window requires more memory than the heap provides, and the last window size measured is the reached window size. In the third one, the computation becomes very slow because of the lack of memory and the impact of the JVM Garbage Collector (GC). Given a timeout for update computations set to 5 seconds, when exceeded, the last window size measured is the reached window size. This experiment was done using the same GC (Java 8 G1GC) as in the rest of experiments.

Figure 8 shows the average window sizes for each tested heap size, for capacities of 2^{25} updates. Not appreciated in the chart but relevant, is that *KVS AMTA* was able to insert $1\,281$ and $59\,276$ updates with heap sizes of 128MB and 256MB respectively, while *KVS LMTA* was able to insert $1\,459$ and $66\,171$ updates. *Mem. AMTA*, *Mem. LMTA* and *DABA* were able to insert updates from 256MB heap size and greater, starting with $1\,015$, 877 and $1\,120$ updates each. *Naive* inserted elements from 512MB heap size and greater, starting with $171\,923$ updates.

The reasons why the *KVS* algorithms *AMTA* and *LMTA* start inserting messages with less memory is their reduced need of allocated memory for the empty data structure, being $O(\log n)$ compared to $O(n)$ in the other algorithms. Also notice that *KVS AMTA* and *LMTA* reached the window capacity with 512MB of heap memory behaving normally. This size is smaller by far compared to the heap sizes of the other algorithms. Except for *Naive* that reached the window capacity with 16GB of memory heap, the rest did not reached such capacity until memory heaps of 32GB. This proves the memory-wise benefits of using the AMTA Framework by decoupling most of the data from the local memory aggregation. It also shows that *KVS LMTA* has a slightly better performance in terms of memory usage than *KVS AMTA*, in addition to the capacity to perform fast bulk evictions.

5.8 Experiment 5: Multi-dimensional aggregation

Finally, we evaluate the impact of operating over multi-dimensional data, by analyzing how adding dimensions to data, and making the aggregations on each dimension share resources like the sliding window data structure and the WSP), affects the average computation latency.

Streams can contain synchronous dimensions of data, and the window can aggregate each one individually in the user defined monoid. E.g. dimensions like wind speed, humidity, and temperature, coming from the same stream, might need to be independently averaged with the same WSP. Here we ran a constant-size *KVS AMTA* window with the WSP from Listing 3 and with $maxSize = 2^{15}$, then measured the latency of update insertions for a different number of stream dimensions. The dimensions in the stream are *maxSize* and a k number of *count* dimensions (from $count_1$ to $count_k$). We chose dimensions with simple aggregations in order to quantify the overhead around them. Figure 9 shows: 1) the average latency for k from 1 to 50 as a barplot, 2) the linear regression on the collected results, highlighting the latency growth, and 3) how the latency would sum if each dimension was sequentially aggregated in different windows, repeating operations like data structure management or WSP with their corresponding latencies. E.g., the latency from $k = 1$ being $4\,895$ nanoseconds, with $k = 2$ it would be $4\,895 \times 2 = 9\,790$.

We can see that the linear regression grows slower than the proportional latency, as the monoid computation is a small fraction of the average latency for $k = 1$. The latency of a single *count* aggregation is quantified in 411 nanoseconds and $4,158$ nanoseconds are spent differently and shared between data dimensions. The latency grows linearly with the number of dimensions, although the monoid's impact would be higher depending on the operators used.

6 Conclusions

In this paper we have introduced the Monoid Tree Aggregator Window Framework, a new framework for general sliding window aggregation that advances the state of the art in several aspects: 1) it exhibits an amortized constant $O(1)$ time-complexity between updates, and for the worst-case scenario it exhibits logarithmic cost $O(\log n)$ ahead of the linear cost $O(n)$ of the current existing solutions; 2) it includes a general aggregation mechanism that uses binary associative operations, and a general mechanism to enforce the Window Slide Policy (WSP) with amortized cost $O(1)$, both programmable by framework users; 3) it provides a mechanism to automatically enforce the Window Slide Policy, which enforces efficient bulk data evictions with cost $O(\log n)$ which, to our knowledge, is not supported by any other existing framework; 4) it provides support for multi-dimensional data aggregation, that can be also leveraged to implement the Window Slide Policies; and 5) it was designed to support a scalable implementation backed by a distributed key/value store instead of leveraging local memory only.

The framework has been presented through a detailed description of the main algorithms involved in the manipulation of the critical data structures of the sliding window. The framework has been implemented in two flavours: a local version in which all data is stored in memory and a remote-store version that leverages a distributed Key-Value Store to keep most of the data. In both cases, the algorithms have been implemented on top of Apache STORM, which has been used as the streaming platform, providing a multi-tenant environment to build several sliding window aggregations in parallel. A comprehensive evaluation has been conducted to proof the efficiency of the implementation, and results show that the framework can manage large windows (up to tens of millions of elements) efficiently, with a cost in the order of a few microseconds to insert elements and slide the window. The experiments on bulk data eviction show that the cost of removing large amounts of elements from the window is extremely low, which is a critical requirement for implementing efficient and reactive Window Slide Policies that drive the criteria to include or exclude elements in the sliding window.

7 Acknowledgments

This project is partially supported by the European Research Council (ERC), Spain under the European Unions Horizon 2020 research and innovation programme (grant agreement No 639595). It is also partially supported by the Ministry of Economy of Spain under contract TIN2015-65316-P and Generalitat de Catalunya, Spain under contract 2014SGR1051, by the ICREA Academia program, and by the BSC-CNS Severo Ochoa program (SEV-2015-0493).

References

[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[3] "Apache Samza," http://samza.apache.org, 2017, accessed: May 2017.

[4] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 239–250. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742788

[5] "Amazon Kinesis," https://aws.amazon.com/kinesis/, 2017, accessed: May 2017.

[6] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, Aug. 2013. [Online]. Available: http://dx.doi.org/10.14778/2536222.2536229

[7] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé *et al.*, "Ibm streams processing language: Analyzing big data in motion," *IBM Journal of Research and Development*, vol. 57, no. 3/4, pp. 7–1, 2013.

[8] S. Klein, "Azure stream analytics," in *IoT Solutions in Microsoft's Azure IoT Suite*. Springer, 2017, pp. 71–84.

[9] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[10] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 702–713, 2015.

[11] K. Tangwongsan, M. Hirzel, and S. Schneider, "Constant-time sliding window aggregation," *IBM, IBM Research Report RC25574 (WAT1511-030)*, 2015.

[12] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin, "Summingbird: A framework for integrating batch and online mapreduce computations," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1441–1451, 2014.

[13] B. Moon, I. F. V. López, and V. Immanuel, "Scalable algorithms for large temporal aggregation," in *Data Engineering, 2000. Proceedings. 16th International Conference on*. IEEE, 2000, pp. 145–154.

[14] J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates," in *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE, 2001, pp. 51–60.

[15] A. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 336–347.

[16] P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues, "Slider: Incremental sliding window analytics," in *Proceedings of the 15th International Middleware Conference*. ACM, 2014, pp. 61–72.

[17] A. Bifet and R. Gavalda, "Learning from time-changing data with adaptive windowing," in *Proceedings of the 2007 SIAM international conference on data mining*. SIAM, 2007, pp. 443–448.

[18] ——, "Kalman filters and adaptive windows for learning in data streams," in *International Conference on Discovery Science*. Springer, 2006, pp. 29–40.

[19] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *ACM SIGMOD Record*, vol. 34, no. 1, pp. 39–44, 2005.

[20] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM journal on computing*, vol. 31, no. 6, pp. 1794–1813, 2002.

[21] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues, "Incapprox: A data analytics system for incremental approximate computing," in *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2016, pp. 1133–1144.

[22] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[23] S. Krishnamurthy, C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 623–634.

[24] K. Tangwongsan, M. Hirzel, and S. Schneider, "Low-latency sliding-window aggregation in worst-case constant time," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM, 2017, pp. 66–77.

[25] "Couchbase," https://www.couchbase.com, 2017, accessed: May 2017.

[26] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

Álvaro Villalba received the MS degree at BarcelonaTech-UPC in 2011. He is research engineer and a Ph.D. student, working in processing of data streams as a service and reactive algorithms for the Internet of Things, Fog Computing and real-time Big Data analytics at the Barcelona Supercomputing Center (BSC) within the "Data-Centric Computing" research line.

Josep Lluís Berral received his degree in Informatics (2007), M.Sc in Computer Architecture (2008), and Ph.D. at BarcelonaTech-UPC, speciality on Computer Science (2013). He is a data scientist, working in applications of data mining and machine learning on data-center and cloud environments at the Barcelona Supercomputing Center (BSC) within the "Data-Centric Computing" research line. He has worked at the High Performance Computing group at the Computer Architecture Department-UPC, also at the Relational Algorithms, Complexity and Learning group at the Computer Science Department-UPC. He is an IEEE member.

David Carrera received the MS degree at BarcelonaTech-UPC in 2002 and his PhD from the same university in 2008. He is an associate professor at the Computer Architecture Department of the UPC. He is also an associate researcher in the Barcelona Supercomputing Center (BSC) within the Data-Centric Computing research line. His research interests are focused on the performance management of data center workloads. He has been involved in several EU and industrial research projects. In 2015 he was awarded an ERC Starting Grant for the project HiEST. He received an IBM Faculty Award in 2010. He is an IEEE member.