Constant-Time Approximate Sliding Window Framework with Error Control

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

***Abstract*—Stream Processing is a crucial element for the Edge Computing paradigm, in which large amount of devices generate data at the edge of the network. This data needs to be aggregated and processed on-the-move across different layers before reaching the Cloud. Therefore, defining Stream Processing services that adapt to different levels of resource availability is of paramount importance. In this context, Stream Processing frameworks need to combine efficient algorithms with low computational complexity to manage sliding windows, with the ability to adjust resource demands for different deployment scenarios, from very low capacity edge devices to virtually unlimited Cloud platforms. The Approximate Computing paradigm provides improved performance and adaptive resource demands in data analytics, at the price of introducing some level of inaccuracy that can be calculated.**

In this paper we present the Approximate and Amortized Monoid Tree Aggregator (A^{2}MTA). It is, to our knowledge, the first general purpose sliding window programable framework that combines constant-time aggregations with error bounded approximate computing techniques. It is very suitable for adverse stream processing environments, such as resource scarce multi-tenant edge computing. The framework can compute aggregations over multiple data dimensions, setting error bounds on any of them, 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*—Big Data, Analytics, Stream Processing, Real-time, Aggregation, Sliding Window**

I. Introduction

Data stream aggregations are a critical requirement for many data mining and monitoring scenarios. Such scenarios, like telemetry data analysis in large data centers, or advance analytics for the Internet of Things, often require continuous low-latency aggregation of vast amounts of data and immediacy of the aggregation results in order to produce fast on-site actuations. Processing data close to the source also becomes an important factor when data flow is expensive due to high volume of data and poor connectivity. The environments in which the data analytics need to be computed are not always favorable. Low power consuming hardware, limited resources and unreliable internet access are usual conditions for Smart Cities and Fog Computing [9].

Data streams are unbounded sequences of ordered atomic updates from the same information feature. For instance, the stream produced by a GPS sensor contains a time-ordered sequence of independent updates containing latitude and longitude of the specific device where the sensor is attached to. Each update in the sequence replaces the previous one. Data streams will contain virtually infinite updates, therefore such sequences of updates can not be traversed upstream as they do not have finite size and lack boundaries. Sliding window aggregations set an upper boundary, i.e. 1000 updates, starting from the newest update in the stream. Therefore, the boundary defined by the window slides with the update stream generation. The window operations to add and remove updates work as a FIFO data-structure, such as a queue. Generally, sliding windows aggregate the window contents every time the window changes (a new stream update is generated), producing a new data stream of aggregated data.

Due to the unbound nature of streams, sliding windows are a convenient approach to processing such aggregations. However, the size of the contents in a window can still be considerably big and this can have a major impact in terms of performance. Therefore, sliding windows ideally also need to: a) have low latency and low time complexity, b) work with low memory resources and unreliable connectivity. Precedent work introduced Amortized Monoid Tree Aggregator (AMTA) [32] as an amortized constant-time sliding window framework with its contents distributed in a *Key-Value Store* (KVS) instead of residing in local memory. AMTA takes advance of incremental aggregation algorithms optimized for distributed fault-tolerance data replication, in order to free the local memory from the window data-structure. The aggregation functions can be provided by the user with a *MapReduce*-like programing model, in which the *reduce* function is an associative operation (*monoid*). However, when the connectivity to the KVS is unreliable, the window aggregation will either fail or its data-structure will indefinitely grow in local memory. On the other hand, when the connectivity is reliable, the data-structure might still be consuming a substantial amount of shared resources from the KVS cluster that could be used for multiple additional aggregations.

In this paper we introduce the Approximate and Amortized Monoid Tree Aggregator (A^{2}MTA) general window aggregation framework. A^{2}MTA is an approximate aggregation framework that benefits from the work in AMTA as to: 1) Amortized constant $O(1)$ time-complexity between updates, while logarithmic $O(\log(n))$ in the worst-case scenario. 2) Distributed and replicated data-structure in a KVS, freeing local resources and facilitating a fault-tolerance system. 3) Only $O(\log(n))$ of

 DOI 10.1109/ISORC.2019.00031

the data in the data-structure sits in local memory. 4) User-programmable window aggregation mechanism and window slide policy. 5) Bulk update evictions triggered by the window slide policy are considered atomic operations and have a worst-case $O(\log(n))$ cost.

On top of AMTA, A^{2}MTA provides a set of mechanisms that reduce considerably the size of the data-structure and the computation time, in exchange of a degree of error in the aggregation results. In other words, provides an approximate computing framework for scalable sliding window aggregations. In this scenario, the granularity in a sliding window contents is divided into multiple aggregated updates, or update buckets, instead of individual stream updates. For instance, in a summation window, we keep only update buckets containing the summation of k updates instead of k separated updates. When evicting stream updates from the window, the minimum unit to be evicted are whole buckets, even if only a portion of the bucket needs to be evicted. More specifically and within a defined confidence level, A^{2}MTA defines buckets by:

- Aggregation error control. The aggregation in a bucket is used to estimate its impact in the window aggregation result and contain it. In cases in which enforcing a level of error is necessary to make buckets grow, that error is bounded by the user.
- Size of predicted bulk evictions. Frequent and highly probable evictions of at least k updates will entail buckets with k aggregated updates.
- Maximum number of buckets, as an ultimate memory resources restriction (constant $O(1)$ size). Updates are spread out among buckets to distribute the weight of the window in order to comply with the restriction.
- Network availability. The number of buckets in local memory will be limited. Therefore, when it is not possible to send them to a KVS, the buckets need to aggregate more updates. This can done by dynamically reducing the maximum number of buckets.

The rest of the paper is structured as follows: Section II defines the Approximate AMTA Framework; Section III provides the results of an experimental evaluation of the Approximate AMTA Framework; Section IV discusses the state of the art in the fields of both approximate computing and efficient sliding windows; Finally, Section V summarizes the conclusions extracted from this work.

II. Approximate AMTA

Approximate computing is a widely used paradigm in data analytics algorithms that can drastically reduce the needed resources in order to obtain a result. It relies on the degree of tolerance a system may have to some loss of quality or optimality in the computation result.

In this section we introduce Approximate AMTA (A^{2}MTA), a sliding window framework that assumes the AMTA [32] data-structure, based on binary trees. The leaves' level of a tree contains the values inserted to the window, while the rest of the tree levels contain partial incremental aggregations. Depth-wise, the closer a node is to the root, the more updates it aggregates. Breadth-wise, the closer a tree node is to the leftmost branch, the older the aggregated updates are. The aggregation functions are *monoids*: binary associative functions with a neutral element and function inputs and output from the same set; i.e. $+$ monoid is a binary and associative function, its neutral element is 0 and integer inputs result in integer outputs. This data-structure has been demonstrated to keep its amortized constant-time, efficient bulk evictions and enable horizontal scalability with a distributed data store.

In A^{2}MTA we propose to only keep some partial aggregations from consecutive updates, called buckets, building the window as a histogram of updates. The required memory can be drastically reduced, and we will prove that the performance is also improved. However, the aggregation result might not be accurate due to having effectively too many or too few updates in the window, due to the coarse granularity given by the buckets. Consider a scenario in which we have a static size sliding window containing buckets with two aggregated updates each. The window requires to evict one update for every insertion. However, for every insertion either two updates (a bucket) or none will be removed. Therefore, every two insertions, the window will either be one update more or one update less than in an accurate aggregation. In other words, any bucket eviction policy may turn out to result in: a *false positive* bucket, by keeping a bucket aggregating updates that need to be evicted; a *false negative* bucket, by removing a bucket aggregating updates that should be kept in the window. Note that removing a false positive bucket would result in a false negative bucket, and vice versa.

In order to mitigate the effects of false positive/negative bucket error, the proposed methods in this section decide whether a new update must start its own bucket in the window, or it must be aggregated to the newer existing bucket. This is done by either: controlling the result error, keeping a reduced number of inaccurate results, or prioritizing a maximum number of elements in the data-structure.

Different kinds of aggregations need for specific approaches to adjust the error. Hirzel et al. [22] classify the types of window aggregations into five groups: *Sum-like*, *max-like*, *collect-like*, *median-like* and *sketch-like*. *Sum-like* aggregations compute values with invertible functions and include aggregations such as *sum*, *count* and *average*. This kind of aggregation have a single neutral element (i.e. 0 for a *sum*), and therefore the results tend to vary. *Max-like* aggregations generally make a selection of a non-ranked input update, leaving the rest of the updates without any effect on the result. They are not invertible and include algorithms like *max*, *min*, *argMax*, *argMin* and *maxCount*. Neutral elements in a *max* aggregation, for example, would be all values below the current result, therefore there is a high probability that a new update insertion does not affect the result.

Collect-like and *median-like* aggregation algorithms have collections of values as the *monoid* set instead of a single one, and therefore the result error can not be quantified with a single numerical value. *Sketch-like* algorithms are approximate computing algorithms by themselves, such as HyperLogLog

or Bloom filter, and therefore will not be considered in this section either.

On the one hand, we propose two approximate computing methods with result error control, one specific for *sum-like* aggregation algorithms and another for *max-like* ones: *Sum-like histogram* and *Max-like histogram*. On the other hand, we propose two more methods that can be combined with the previous ones: *Hop histogram*, which focus aggregating frequent bulk evictions into buckets, and *Maximum size enforcement*, which forces the window data-structure to have a deterministic maximum number of buckets while keeping a uniform bucket size.

A. Sum-like histogram

Since *sum-like* aggregations only have a single neutral element, its result change whenever a non-neutral value (all except for one) is inserted or evicted. That makes this kind of aggregation improbable to keep without any error while keeping the values in aggregated buckets. The goal of value error control for *sum-like* aggregations is to make buckets grow while keeping the aggregation error under the error tolerance defined by the user.

The bucket error can be calculated as the maximum between its *false positive* and *false negative* bucket errors. The *false positive* bucket error is the maximum absolute aggregation of the oldest updates aggregated in a bucket, not including the single newest one. On the contrary, the *false negative* bucket error is the maximum absolute aggregation of the newer updates aggregated in a bucket, not including the single oldest one. For example, if a bucket contains an aggregation of the sequence of updates $[1, 1, -1, -1]$, the false positive error is $1 + 1 = 2$, while the false negative error is $-1 - 1 = -2$. In this example, both errors have the maximum absolute error value: 2. In case that we want to control the error of multiple dimensions of the aggregation (i.e. *sum* and *count* in an average aggregation), this process can be applied to each dimension.

The error can be constrained by the user either relatively to the result or as an absolute error. When the aggregation requires to calculate the error relative to the result, we need a window result prediction interval. The extremes of the prediction interval will be used to estimate the maximum error the bucket can generate when the bucket becomes a potential false positive or negative bucket. We estimate the aggregation result with a prediction interval using a sample of the previous results and assuming the *central limit theorem* as follows:

$$\left(\bar{x} - t^* s\sqrt{1 + \frac{1}{n}}, \bar{x} + t^* s\sqrt{1 + \frac{1}{n}}\right)$$

Where $\bar{x}$ is the sample mean, s is the sample's standard deviation, and t^* is the two tailed percentage point of Student's t distribution given a specific confidence level with $n - 1$ degrees of freedom. We defined the sample as, at least, all the results generated by each update currently aggregated in the window, with a minimum of 30 elements.

The use of this method can be generalized to any kind of aggregation in terms of controlling the error on the number of elements in the window, instead of controlling the actual result of the window. This way, even non-numerical aggregations can benefit from A^{2}MTA, getting an approximate result.

B. Max-like histogram

In max-like aggregations (or extreme value aggregations) only a subset of the computed values have any influence on the result. The rest of the elements are irrelevant and the aggregation would provide the same result if they were ignored. The goal of this method is not to discard the irrelevant updates, but to aggregate all the consecutive ones in the same bucket.

For instance, in the window [3, 1] with monoid max, '1' might be the result of the window when '3' gets evicted. However, in the window [3, 1, 2], the update '1' will never affect the result in this window. When the update '3' gets evicted from the window, the result will be at least '2'. If we knew that the window results would be between '3' and '2', '1' could have been aggregated in the same bucket as '3', and there would not have been any difference in the result. In case that there were not other dimensions aggregated in the window, the update could even be discarded.

A^{2}MTA estimates a range of value candidates to become the result in the window. If an inserted update is found within this range, it will generate a new bucket. The update will be aggregated to the last bucket otherwise. There is no error constraint specified by the user to be considered, but the aim is to mainly produce accurate results. The range result candidates is estimated using extreme value theory [12].

The Fisher-Tippett [16] theorem states that the cumulative distribution function from a sample of size n with independent and identically distributed random variables converge to the Generalized Extreme Value (GEV) distribution, as $n \to \infty$. There are three parameters for the GEV distribution: μ for the location, σ for the scale, and ξ for the shape. By this theorem, it is possible to estimate a fitting GEV distribution given a sample of extreme values. A^{2}MTA uses Block Maxima (BM) [19] and the GEV probability-weighted moments (PWM) estimation method [15], [23]. The main reason to use PWM rather than Maximum Likelihood Estimation (MLE) method is because it performs better with small samples, and our goal is to keep the minimum amount of data possible. From the estimated GEV we will be able to extract the estimated boundaries for the extreme values.

Following the BM method, the monoid is applied to the inserted updates in blocks of an specified size. The result of each aggregated block will be added to the sample of extreme values. The sample size has been set to 30, and the block size is defined by the user. With this sample a fitting GEV distribution is computed using PWM. From there, the upper and lower bounds can be extracted. If $\xi \geq 0$, then the upper bound that we will consider will be a GEV quantile (e.g. 0.99). Otherwise, if $\xi \leq 0$ then the lower bound considered will be the remaining quantile (e.g. 0.01).

Deciding an optimal block size is out of the scope of this work, and therefore is left as a user decision, although in many

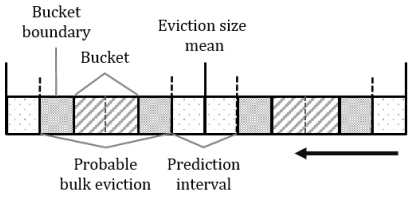


Fig. 1: Bulk eviction buckets. Predicted eviction: 6 ± 1

situations the block periods can appear naturally [27], [30], [31]. Small block sizes would compute wide GEV boundaries and, therefore, the rate of bucket aggregations would be very low. Also, having small blocks causes a higher ratio of more costly GEV fitting computation. On the other hand, big block sizes would cause a biased GEV fitting computation that would translate to multiple inaccurate results. In Section III we compare how different block sizes behave.

This method is compatible with a window eviction policy performing bulk evictions of all the updates that precede the update that brings the window result. As it was evaluated in AMTA's article [32], this kind of eviction policy significantly improves time performance as it reduces average window size. Furthermore, the use of buckets improves these figures and reduce the number of inserted elements in the data-structure and in the KVS.

C. Hop histogram

Consider sliding windows programmed to evict multiple updates each time. Usually referred as *hopping window*, they remove constant amounts of updates from the window. For example a hopping window could remove 500 updates when the size of the window is 1500. In such a situation, if the updates waiting to be evicted were aggregated in a single bucket, then the size of the window would be reduced and there would not be an effective error in the result. In addition, if we managed to aggregate all the future evictions in the window into buckets, the window size reduction would be vast and produce no error at all. From the previous example, if we aggregated every 500 in a single bucket, we would have only three buckets in memory instead of 1500 updates.

However, AMTA eviction mechanism gives freedom to program dynamic sized windows with evictions sizes changing over time. That can create scenarios in which there is some variability between bulk evictions. We propose a method that aggregates the inserted updates into buckets within a predicted eviction size. We estimate the size of the evictions with a prediction interval using a sample of the previous eviction sizes and assuming the *central limit theorem*, same as in Section II-A. We defined the sample as, at least, all the evictions that fit in the current window, with a minimum of 30 elements.

The prediction interval is used to predict sequences in the window with high probability to include future eviction boundaries. Figure 1 shows a window where every square represents an update in a bucket, with eviction size prediction interval of 6 ± 1 so far. As it can be seen, the window updates

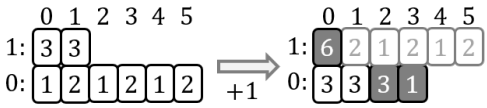


Fig. 2: A^{2}MTA data-structure constrained with 6 leaves

are divided into sequences containing as many elements as the mean eviction size. The number of elements in the probable eviction boundaries (prediction interval) cover the updates around the mean eviction size. These updates will not be aggregated in a bulk eviction bucket, avoiding introducing error to the window according to the prediction. However they will be aggregated using criteria from other methods. That leaves a block of four updates that are predicted to be evicted together in the same bulk eviction. As we saw previously, the size error in a window is generated by the false positive and false negative bucket errors. Considering this, the complete aggregation of the probable bulk eviction generates an error if it becomes the last evicted bucket or oldest bucket in the window, even if the estimation was correct. In this situation we can not know if the eviction was performed as predicted or the prediction was actually incorrect. In order to avoid the error after a successfully predicted eviction, the bucket needs to contain all the updates from the probable bulk eviction block but the ones in its boundaries, one from each side.

In case that the prediction actually failed, it should be reflected in the statistics from the eviction size sample. Otherwise the prediction interval would be biased. Therefore, the worst-case eviction size is added to the statistics sample.

D. Constrained footprint enforcement

A^{2}MTA can be executed in environments with resource constraints that need to be taken into account, either because it is running on a low-resources environment or because it is a shared multi-tenant environment. Particularly, A^{2}MTA needs a deterministic limit of its data-structure size and the network traffic a distributed data store. The A^{2}MTA maximum size enforcement method uses two mechanisms in order to limit the data-structure size. The first mechanism is level eviction. A^{2}MTA data-structure is based on trees, in which the leaves level contain the buckets, and the rest of the tree levels have aggregations from the nodes on the previous level. Each level is a queue with level nodes and the levels are found in a circular queue. When the number of elements in the leaves level is going to be exceeded, the level is marked as empty and moved from the front to the back of the circular queue. Therefore, the new leaves level contains the previous leaves aggregated into bigger buckets.

Figure 2 shows the level eviction mechanism applied to an example A^{2}MTA data-structure. From the constant-time AMTA data-structure, the nodes in level 1 are the parents that aggregate the initial four elements from level 0. The size constraint in this window is of 6 leaves (level 0). After inserting the update '1', the window would have 7 leaves, exceeding the limit set. Before inserting the update, the level 1 is shifted to the position 0 and the previous leaves level is

now in the position 1 and marked as empty. Then, the last pair in the previous leaves level is aggregated ('3') and inserted to level 0, and its aggregation propagated upwards generating a new root '6'. Now the window structure is consistent and contains the same aggregation as before starting the insertion, but with an evenly distributed growth of the buckets sizes. The new update is now inserted to the data-structure without exceeding the size limit.

The second mechanism for constraining the window size consist of aggregating updates into buckets with a calculated size continuously after the first level eviction. This mechanism has three main goals: keep uniformity in terms of bucket size and therefore, have a uniform update count error; reduce the number of element insertion operation in the data-structure, which can be more costly than inserting the update to the bucket; reduce the level evictions, and use them as a last resort. The max size of the currently building buckets is calculated as: $\lceil \frac{count}{constraint} \rceil$, where $count$ is the number of aggregated *updates* in the window and $constraint$ is the max number of *buckets* in the sliding windows. While the number of updates in the window changes, the bucket size increases or decreases with it. This mechanisms reduces the number of elements inserted to the data-structure, by aggregating new updates in existing elements. By reducing insertions, the number of nodes transferred to the AMTA distributed data store is also reduced. Therefore, when the bandwidth to the distributed data store and local memory are too low, the maximum window size can be reduced in order to reduce the data traffic in exchange for aggregation granularity.

Another advantage of having a deterministic data-structure size regardless of the number of updates aggregated is that the time and size complexities are effectively constant in the worst case.

III. Evaluation

The evaluation of Approximate AMTA analyzes how it behaves in terms of result accuracy, time performance and footprint. The the data-structure footprint affects both the memory usage and the network traffic. Reducing the data-structure footprint implies a slower growing window and less elements sent to the distributed data store or KVS.

This section is divided into two main parts. The first one analyzes the effect of different values in the user-configurable parameters of each A^{2}MTA method applied to its respective use case. Furthermore, the maximum size enforcement will be tested on the three scenarios: sum-like aggregation, max-like aggregation and hopping window. The second part is focused on the time performance impact of the three scenario-specific methods, compared to the state-of-the-art sliding window framework with best performance to our knowledge.

The data set used as a stream in the following experiments contains two years of a server's RAM memory usage monitoring in KB, where the available memory is 64GB. There is one memory usage update per second, which adds up to 62 208 000 updates. The operation performed in the experiments and its eviction policy will differ between the experiments due to the nature of the different scenarios considered, but they all share a maximum of 2 592 000 aggregated updates, which corresponds to a month worth of updates.

	Footprint
Max error	Sum-like histogram
$10^{-4}\%$	44.02%
$10^{-3}\%$	6.591%
$10^{-2}\%$	$8.335 \cdot 10^{-1}\%$
$10^{-1}\%$	$9.9 \cdot 10^{-2}\%$
1%	$1.022 \cdot 10^{-2}\%$
10%	$9.854 \cdot 10^{-4}\%$

	Footprint
Block size	Max-like histogram
10	91.33%
10^2	91.1%
10^3	95.49%
10^4	60.97%
10^5	4.394%
10^6	19.88%

	Footprint
Parameter	Hop histogram
$\times$	$5.229 \cdot 10^{-1}\%$

TABLE I: Scenario-specific bucket aggregation method's footprint relative to AMTA's

	Footprint		
Max size	Sum-like aggregation	Max-like aggregation	Hopping window
10^6	33.33%	42.98%	38.56%
10^5	3.846%	11.75%	4.583%
10^4	$3.845 \cdot 10^{-1}\%$	2.189%	$4.679 \cdot 10^{-1}\%$
10^3	$3.858 \cdot 10^{-2}\%$	$3.368 \cdot 10^{-1}\%$	$4.69 \cdot 10^{-2}\%$
100	$3.864 \cdot 10^{-3}\%$	$3.676 \cdot 10^{-2}\%$	$4.7 \cdot 10^{-3}\%$
10	$4.018 \cdot 10^{-4}\%$	$3.298 \cdot 10^{-3}\%$	$4.9 \cdot 10^{-4}\%$

TABLE II: Constrained A^{2}MTA footprint relative to AMTA's

A. Implementation

All methods are implemented in Java 1.8 and executed in the A^{2}MTA operator 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.

A^{2}MTA will buffer up to 512 buckets from the data-structure before storing them to a distributed data store. Moreover, the data store used in the experiments is Couchbase [1]. Couchbase is a KVS based on memcached [17], with a distributed LRU cache in RAM. It prioritizes access in memory over disk for low-latency.

B. 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. The logic was executed in a single node, but Couchbase ran as a cluster in three extra nodes. Therefore, the contents of the data-structure were distributed between 4 nodes.

C. Experiment 1: Parameters

On this first experiment we tested different user-configurable parameter values in each bucket aggregation method, for the different considered aggregation scenarios. Three scenarios were considered: sum-like aggregation, max-like aggregation and hopping window.

Sum-like aggregation computes the average of the monitored memory values in a static-size window of $2\,592\,000$ updates. This scenario is designed to evaluate the sum-like histogram method's parameters. The sum-like histogram method is configured with a 95% confidence, and it controls two dimensions from the update: the sum value and the count of elements.

Max-like aggregation extracts the maximum value from a window of $2\,592\,000$ update. A relevant consideration about this scenario is that the eviction policy keeps the current maximum update as the oldest one in the window. This is applied in all cases, including the accurate aggregation, as it improves the computation time and the memory footprint [32]. Also, only when the max update changes, a new result is produced. This scenario is used to evaluate the max-like histogram method's parameters. The max-like histogram method is configured with a 95% confidence.

Hopping window will reach $2\,592\,000$ and then perform a bulk eviction with a random number updates with mean $864\,000$ and standard deviation 100. The aggregation performed is also an average of the monitored memory values. The hop histogram method is also configured with a 95% confidence.

For each scenario we also evaluated the behavior of a constrained footprint window. The evaluated parameter values were incremented exponentially in order to get a clear sense of its impact.

The impact of the parameters is evaluated in terms of the effective error produced on the aggregation when compared to the accurate aggregation, and the generated footprint using the accurate aggregation footprint as baseline. The error will be shown as cumulative distribution. The footprint is calculated as the number of new elements generated in the MTA data-structure. This number affects the memory usage, but also network traffic to the KVS; 1% smaller footprint means using 1% less of memory, but also 1% less of messages exchanged with the KVS.

1) Sum-like aggregation: In Figure 3a we can see the error cumulative distribution of the sum-like histogram, with a *max error* parameter from $10^{-4}\%$ to 10%. The x-axis has a log-scale for readability. The most noticeable outcome from this figure is that, indeed, the max error defined by the user has not been exceed. Particularly, $10^{-4}\%$ has a 45% of accurate results. However, in Table I, where it shows the footprint of each parameter, we can see that the same *max error* has also big footprint (half of A^2MTA's) compared to the rest values. As it can be seen, footprint of the window grows linearly as the specified max error decreases.

For the constrained footprint sum-like window, in Figure 3b and Table II, we chose *max size* values that generate the same *count* aggregation error as in the sum-like histogram. Since the window have a static size in both cases, the accurate *count* value is always $2\,592\,000$. In order to generate a maximum error of $10^{-4}\%$, we need to limit its size to 10^6 buckets. We can see that the constrained footprint shows a similar trend to the sum-like histogram, but with smaller footprints. However, as there is no error control (needed for the *sum* aggregation), in Figure 3b can be seen that the error is greater in the constrained footprint window in all cases. Also fewer results have accurate results, with parameters from 10^3 to 10 notably having none.

2) Max-like aggregation: Figure 4a shows the max-like histogram's error with different block sizes. As we expected, the a block size too big (10^6) makes a biased estimation of the GEV distribution and ends up generating results with elevated error, and low number of accurate results. The rest of the block sizes have more than 98% of accurate results, and maximum errors from none to 12%. In the figure, block sizes 10^2 and 10^3 can not be found because all their results are accurate. However, in Table I we can see that small block sizes generate very little footprint reduction. In contrast to sum-like histogram which choosing the parameter is a matter of priorities, in this case we have a clear most convenient parameter: 10^5, which covers a sample of $3\,000\,000$ elements. It is the best managing the trend changes on the data values. It has a 4.583% of footprint, 98.55% of accurate results and a maximum error of 12.51%.

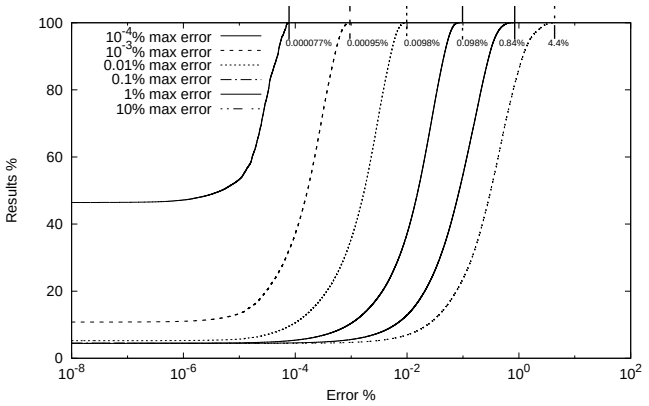
On the other hand, in Figure 4b we can see the behavior of a constrained max-like window, with different max number of buckets. We can see that even though the max errors are similar to the max-like histogram and the number of accurate results are acceptable (between 87.11% and 91%), with similar footprints as the optimal parameter in the max-like histogram experiment we get a lower number of accurate results. With a footprint of 11.75% (Table II) we get 88.92% of accurate results, while with 2.189% footprint the number of accurate results is 88.73%. Therefore, the estimation of the GEV distribution has to predict extreme values has a clear effect on the error control and the footprint. It is worth noticing that the footprint is one order of magnitude higher than *Sum-like aggregation*, and that is due to the bulk evictions done when the maximum value is between the newer updates in the window.

3) Hopping window aggregation: Hopping window histograms do not have any configurable parameters, therefore in Table I we can only see a single value from the performed experiment. The footprint is as small as the 0.5229% of the accurate window. Furthermore, in the experiment all results were 100% accurate. This clearly demonstrate that non-deterministic hopping windows can be greatly improved by using hopping window histograms. This method requires the window to have a clear hopping pattern in its bulk evictions in order to reduce the footprint. However, when this scenario happens, the footprint reduction is generally at no cost.

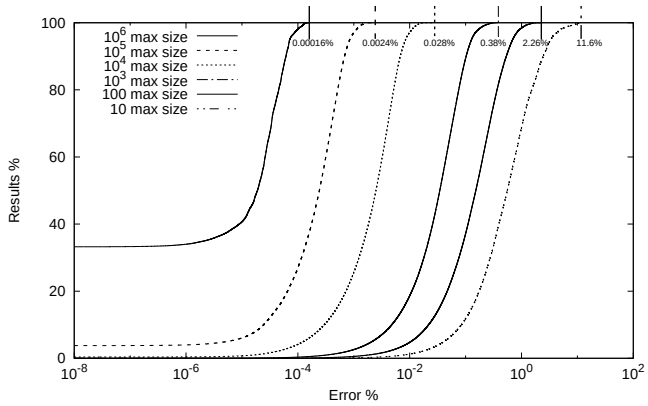
However, the constrained windows show a poor behavior in terms of error in Figure 5, with no accurate results in any of the tried parameter values. Table II shows that the footprint is higher than *Sum-like aggregation* due to the sudden size changes, but not as high as *Max-like aggregation*.

D. Experiment 2: Time performance

In this experiment we will focus on a single parameter value from the scenario-specific methods from Section III-C in order

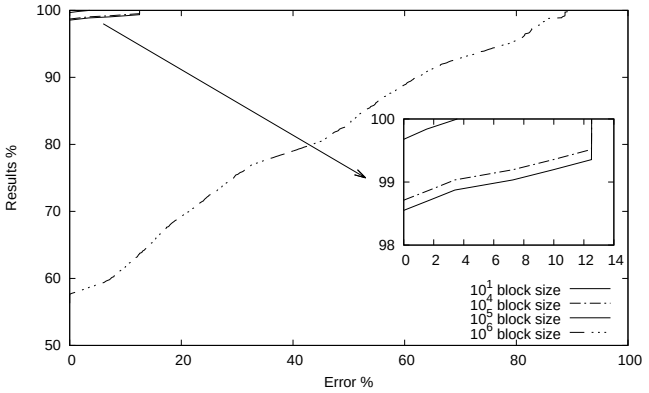


(a) Sum-like histogram

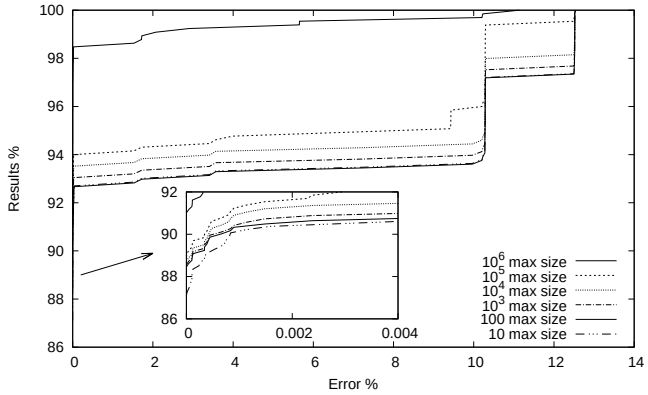


(b) Constrained window

Fig. 3: Effective error in a sum-like aggregation



(a) Max-like histogram



(b) Constrained window

Fig. 4: Effective error in a max-like aggregation

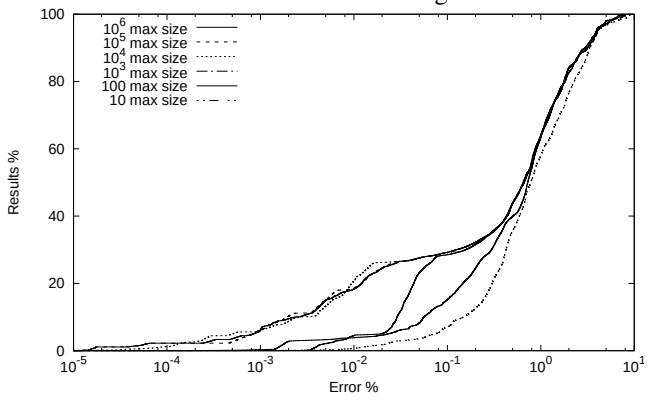


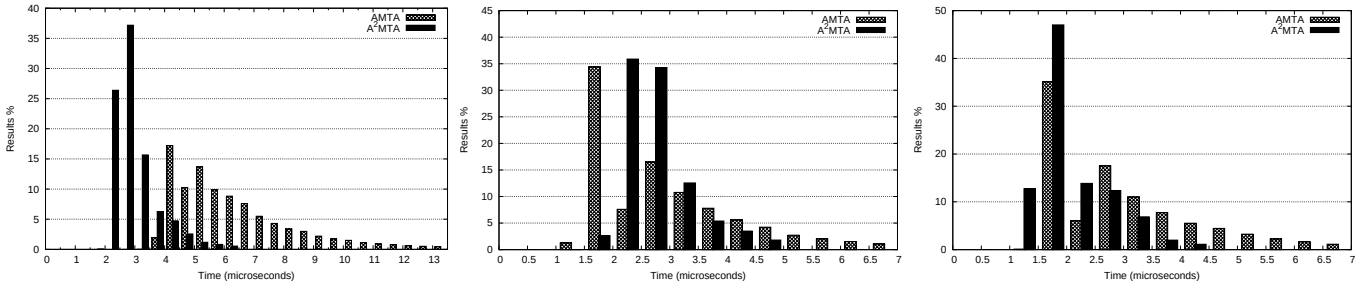
Fig. 5: Effective error in a constrained hopping window

to compare their time performance with the same aggregation executed in AMTA. The goal is to determine if the additional computation required from the different bucket aggregation methods make the approximate computation more costly in terms of time performance, or if it generally saves computation time from the AMTA baseline.

From the three scenarios considered, the one that is more time consuming in AMTA is the sum-like aggregation. The reason is that for every insertion it also needs to perform a $O(1)$ eviction, while the two other scenarios perform $O(logn)$ bulk evictions after fewer insertions. For that reason, it can be seen in Figure 6a that there is a clear improvement with A^{2}MTA's sum-like histograms. The maximum error in this execution was $0.1\$$. The time interval that the aggregations take is narrower and lower than in AMTA. A^{2}MTA concentrates all the result times between 1.5μs and 6.5μs with almost 80% of the results between 1.5μs and 3.5μs, while AMTA is spreaded from 3.5μs to 13μs. There are two main reasons for these results. On the one hand, buckets reduce the number of evictions. If an static size window with $1\,000$ updates is divided by 10 buckets, then there will be 1 eviction for every 100 insertions. On the other hand, when an update is aggregated into a bucket, the insertion cost is always constant.

In Figure 6b, for the max-like histogram with 10^5 block size, the A^{2}MTA still has a narrower interval of execution times in relation to AMTA, with 70% of the results having times between 2μs and 3μs. Whereas, the same amount of results can be found between 1.5μs and 3.5μs in AMTA. However, the improvement is not as noticeable as in the sum-like aggregation: the peak time in AMTA is already low, because it is getting benefit from bulk evictions that reduce the number of overall evictions.

Finally, Figure 6c shows the hopping window scenario. In this case, as the eviction size is estimated and used as the bucket size, then the bulk eviction cost is close to a single eviction (lower). Therefore, the time for A^{2}MTA is globally



(a) Sum-like histogram: 0.1% error (b) Max-like histogram: 10^5 block size (c) Hop histogram

Fig. 6: A^{2}MTA and AMTA time performance

better both in terms of peak minimum time and interval size, compared to AMTA and to the other A^{2}MTA methods.

IV. Related Work

Extensive work has been done in the last years on efficient sliding window aggregations and frameworks. Aside from Amortized MTA [32], which is the initial sliding window framework to which we applied the approximate computing paradigm, the literature propose FIFO data-structures and incremental operations that keep a logarithmic or constant complexity in stream processing aggregation algorithms.

Tangwongsan et al. proposed two sliding window aggregation frameworks called *Reactive Aggregator* (RA) [29] and *Sliding-Window Aggregation* (SWAG) [28]. Being SWAG an important improvement from RA, both approaches follow Boykin et al. [10] method of using associative operations as programmatic aggregators interface. RA has $O(\log n)$ time complexity 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. 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 [5], [6], [26], [33] 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 runs in a constant $O(1)$ time for each one of its atomic operations, bulk eviction not included among them. The algorithm is based on a data-structure with two stacks, one in charge of managing the insertions and the other the single evictions. The lack of an efficient bulk eviction operation for these frameworks' FIFO data-structures, make them unsuitable to distribute the window contents using, for example, a KVS.

Approximate computing for data analytics has been a wide area of study for decades, mainly for aggregations in relational databases, with techniques such as sampling [2], [18], [20], [24], histograms [7], [8], [14], [25], stream sketches [3], [11], [13] or online aggregation [21]. Goiri et al. [20] proposed an approximate computing set of mechanisms for batch processing for Hadoop, called ApproxHadoop. Like A^{2}MTA, ApproxHadoop distinguishes between sum-like and extreme value aggregations. However, the approach is to perform multi-stage sampling, instead of histograms. Regarding sliding window approximate aggregation, Datar et al. [14] propose an exponential histogram count aggregation, with a $O(\frac{1}{\epsilon}\log N)$ overhead and a $1+\epsilon$ loss in accuracy concerning number of elements. This method can be easily applied to other aggregations, but the error is always measured in terms of number of aggregated elements. Bifet et al. contributed *ADWIN* [8] and *K-ADWIN* [7] frameworks, based on Datar's exponential histograms. The two sliding window frameworks aggregate data that has a similar tendency. When two sub-windows have very different average values, the oldest one is removed. Having a defined eviction policy based on the difference between buckets containing similar values, the error is kept very low while the time complexity is constant and the window overhead is given by exponential histograms. However, while the aggregation is user-programmable, by design it does not support any kind of programmable eviction policy. Arasu & Manku [4] describe a variety of algorithms to calculate approximate count and quantile sliding windows. Krishnan et al. present IncApprox [24], a general purpose incremental approximate computing framework with error boundaries. Having a logarithmic time complexity, instead of building a histogram, it benefits form an online stratified sampling algorithm guided by an error prediction.

V. Conclusions

In this paper we have introduced the Approximate AMTA (A^{2}MTA) framework, a novel general sliding window aggregation framework that combines a constant-time FIFO data-structure with the resource reduction benefits from the approximate computing paradigm. While a completely user-programable sliding window is bound to have a non-deterministic resource consumption, the leverage of approximate computing techniques delimits it and contributes with better performance. Furthermore, the accuracy of the results can be configured with some confidence levels.

We described A^{2}MTA as a framework with a set of the different approximate computing methods for the different kinds of sliding windows. On the one hand we defined *Sum-like histograms* and *Max-like histograms*, which are applied to different types of aggregations and therefore can not be combined. The first will keep the aggregation error bellow the boundaries set by the user, while the second aims to produce

accurate results with a confidence level. On the other hand, *Hop histograms* focuses on the detection of a usual type of eviction policy that corresponds to hopping windows and also aims for accurate results, as long as the *Constrained footprint enforcement* prioritizes the window memory footprint over the error. These last two methods can be combined with all the rest.

A thorough evaluation has been performed to give evidence of the impact of the approximate aggregation techniques. In addition, we evaluated the controlled degradation of the aggregation results, confirming that it behaves accordingly to the parameters given by the user. The result show that even having as a baseline the most efficient sliding window aggregation framework to our knowledge, the computation time has been improved in all the tested cases. Furthermore, the impact on the window footprint, which affects both memory and bandwidth resources, makes A^{2}MTA very engaging for adverse environments.

VI. Acknowledgments

This project is partially supported by the European Research Council (ERC), Spain under the European Union's 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] Couchbase. https://www.couchbase.com (Oct 2018).

[2] M. Al-Kateb and B. S. Lee. Stratified reservoir sampling over heterogeneous data streams. In *International Conference on Scientific and Statistical Database Management*, pages 621–639. Springer, 2010.

[3] H. C. Andrade, B. Gedik, and D. S. Turaga. *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press, 2014.

[4] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 286–296. ACM, 2004.

[5] 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*, pages 336–347. VLDB Endowment, 2004.

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

[7] A. Bifet and R. Gavalda. Kalman filters and adaptive windows for learning in data streams. In *International Conference on Discovery Science*, pages 29–40. Springer, 2006.

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

[9] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[10] 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*, 7(13):1441–1451, 2014.

[11] K. L. Clarkson and D. P. Woodruff. Numerical linear algebra in the streaming model. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 205–214. ACM, 2009.

[12] S. Coles, J. Bawa, L. Trenner, and P. Dorazio. *An introduction to statistical modeling of extreme values*, volume 208. Springer, 2001.

[13] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2011.

[14] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.

[15] J. Diebolt, A. Guillou, P. Naveau, and P. Ribereau. Improving probability-weighted moment methods for the generalized extreme value distribution. *REVSTAT-Statistical Journal*, 6(1):33–50, 2008.

[16] R. A. Fisher and L. H. C. Tippett. Limiting forms of the frequency distribution of the largest or smallest member of a sample. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 24, pages 180–190. Cambridge University Press, 1928.

[17] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[18] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, pages 343–352, 2001.

[19] M. Gilli et al. An application of extreme value theory for measuring financial risk. *Computational Economics*, 27(2-3):207–228, 2006.

[20] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 383–397. ACM, 2015.

[21] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Acm Sigmod Record*, volume 26, pages 171–182. ACM, 1997.

[22] M. Hirzel, S. Schneider, and K. Tangwongsan. Sliding-window aggregation algorithms: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 11–14. ACM, 2017.

[23] J. R. Hosking, J. R. Wallis, and E. F. Wood. Estimation of the generalized extreme-value distribution by the method of probability-weighted moments. *Technometrics*, 27(3):251–261, 1985.

[24] 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*, pages 1133–1144. International World Wide Web Conferences Steering Committee, 2016.

[25] 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*, 34(1):39–44, 2005.

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

[27] P. Naveau, A. Guillou, D. Cooley, and J. Diebolt. Modelling pairwise dependence of maxima in space. *Biometrika*, 96(1):1–17, 2009.

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

[29] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment*, 8(7):702–713, 2015.

[30] H. Van den Brink, G. Können, J. Opsteegh, G. Van Oldenborgh, and G. Burgers. Estimating return periods of extreme events from ecmwf seasonal forecast ensembles. *International Journal of Climatology: A Journal of the Royal Meteorological Society*, 25(10):1345–1354, 2005.

[31] A. Van der Valk, B. Middleton, R. Williams, D. Mason, and C. Davis. The biomass of an indian monsoonal wetland before and after being overgrown with paspalum distichum l. *Vegetatio*, 109(1):81–90, 1993.

[32] Á. Villalba, J. L. Berral, and D. Carrera. Constant-time sliding window framework with reduced memory footprint and efficient bulk evictions. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

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