A Deterministic Algorithm for Summarizing Asynchronous Streams over a Sliding Window

Costas Busch[1] and Srikanta Tirthapura[2]

[1] Department of Computer Science
Rensselaer Polytechnic Institute, Troy, NY 12180, USA
buschc@cs.rpi.edu
[2] Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50010, USA
snt@iastate.edu

Abstract. We consider the problem of maintaining aggregates over recent elements of a massive data stream. Motivated by applications involving network data, we consider *asynchronous* data streams, where the observed order of data may be different from the order in which the data was generated. The set of recent elements is modeled as a *sliding timestamp window* of the stream, whose elements are changing continuously with time. We present the first *deterministic* algorithms for maintaining a small space summary of elements in a sliding timestamp window of an asynchronous data stream. The summary can return approximate answers for the following fundamental aggregates: *basic count*, the number of elements within the sliding window, and *sum*, the sum of all element values within the sliding window. For basic counting, the space taken by our summary is $O(\log W \cdot \log B \cdot (\log W + \log B)/\epsilon)$ bits, where B is an upper bound on the value of the basic count, W is an upper bound on the width of the timestamp window, and ϵ is the desired relative error. Our algorithms are based on a novel data structure called *splittable histogram*. Prior to this work, randomized algorithms were known for this problem, which provide weaker guarantees than those provided by our deterministic algorithms.

1 Introduction

Many massive data sets naturally occur as *streams*; elements of a stream are visible to the processor in a sequence, one after another, and random access is impossible. Often, streams are too large to be stored in memory, and have to be processed in a single pass using extremely limited workspace, typically much smaller than the size of the data. Examples include IP packet streams observed by internet routers, a stream of stock quotes observed by an electronic stock exchange, and a sequence of sensor observations observed by an aggregator. In spite of the volume of the data and a highly constrained model of computation, in all the above applications it is important to maintain reasonably accurate estimates of aggregates and statistics on the data.

In many applications, only the most recent elements of a stream are important. For example, in a stream of temperature readings obtained from a sensor network, it may be necessary to maintain the moving average of the temperature over the last 1 hour. In network monitoring, it is useful to track aggregates such as the volume of traffic originating from a particular subnetwork over a recent window of time. Motivated by such applications, there has been extensive work [1, 6, 7, 2, 5, 9] on designing algorithms to compute aggregates over a *sliding window* of the most recent elements of a data stream.

Most previous work on computing aggregates over a stream has focused on a *synchronous* data stream where it is assumed that the order of arrival of elements in the data aggregator is the same as the time order of their generation. However, in many applications, especially those involving network data, this may not be the case. Data streams may be *asynchronous*, and the order of arrival of elements may not be the same as their order of generation. For example, nodes in a sensor network generate observations that are aggregated at the *sink* node. When data is being transmitted to the sink, different observations may experience different delays in reaching the sink due to the inherent asynchrony in the network. Thus, the received order of observations at the sink may be different from the time order in which data was generated. If each data item had a timestamp that was tagged at the time of generation, the sink may observe a data stream whose elements are not arriving in increasing order of timestamps. Asynchronous data streams are inevitable anytime two streams of observations, say A and B, fuse with each other and data processing has to be done on the stream formed by the interleaving of A and B. Even if individual streams A or B are not inherently asynchronous, i.e. elements within A or within B arrive in increasing order of timestamps, when the streams are fused, the stream could become asynchronous. For example, if the network delay in receiving stream B is greater than the delay in receiving elements in stream A, then the aggregator may consistently observe elements with earlier timestamps from B after elements with more recent timestamps from A.

We consider the problem of maintaining aggregates over recent elements of an asynchronous data stream. An asynchronous stream is modeled as a sequence of elements $R = d_1, d_2, \ldots, d_n$ observed by an aggregator node, where d_1 is the element that was received the earliest and d_n is the element that was received most recently. Each element is a tuple $d_i = (v_i, t_i)$ where v_i is the value of the observation, and t_i is a timestamp, tagged at the time the value was generated. Let c denote the current time at the aggregator. We are interested in all elements that have a timestamp within w of the current time, i.e. all elements in the set $R_w = \{d = (v, t) \in R \,|t \in [c - w, c]\}$. Since this window of allowed timestamps $[c - w, c]$ is constantly changing with the current time c, we call it a *sliding timestamp window*. When the context is clear, we sometimes use the term sliding timestamp window to refer to the set R_w.

Definition 1. *For $0 < \epsilon < 1$, an ϵ-approximation to a number X is a number Y such that $|X - Y| \leq \epsilon X$.*

Contributions. We present the first deterministic algorithms for summarizing asynchronous data streams over a sliding window. We first consider a fundamental aggregate called the *basic count*, which is simply the number of elements within the sliding window. We present a data structure that can summarize an asynchronous stream in a small space and is able to provide a provably accurate estimate of the basic count. More precisely, let W denote an upper bound on the window size and B denote an upper bound on the basic count. For any $\epsilon \in (0,1)$, we present a summary of the stream that uses space $O(\log W \cdot \log B \cdot (\log W + \log B)/\epsilon)$ bits. For any window size $w \leq W$ presented at the time of query, the summary can return an ϵ-approximation to the number of elements whose timestamps are in the range $[c-w, c]$ and arrive in the aggregator no later than c, where c denotes the current time. The time taken to process a new stream element is $O(\log W \cdot \log B)$ and the time taken to answer a query for basic count is $O(\log B + \frac{\log W}{\epsilon})$.

We next consider a generalization of basic counting, the *sum* problem. In a stream whose observations $\{v_i\}$ are positive integer values, the sum problem is to maintain the sum of all observations within the sliding window, $\sum_{\{(v,t)\in R \ | t\in[c-w,c]\}} v$. Our summary for the sum provides similar guarantees as for basic counting. For any $\epsilon \in (0,1)$ the summary for the sum uses space $O(\log W \cdot \log B \cdot (\log W + \log B)/\epsilon)$ bits, where W is an upper bound on the window size, and B is an upper bound on the value of the sum. For any window size $w \leq W$, the summary can return an ϵ-approximation to the sum of all element values within the sliding window $[c-w, c]$. The time taken to process a new stream element is $O(\log W \cdot \log B)$ and the time taken to answer a query for the sum is $O(\log B + \frac{\log W}{\epsilon})$.

It is easy to verify that even on a synchronous data stream, a stream summary that can return the exact value of the basic count within the sliding window must use $\Omega(W)$ space in the worst case. The reason is that using such a summary one can reconstruct the number of elements arriving at each instant within the sliding window. Hence, to achieve space efficiency it is necessary to introduce approximations. Datar *et. al.* [5] show lower bounds for the space complexity of approximate basic counting on a synchronous stream. They show that if a summary has to return an ϵ-approximation for the basic count on distinct timestamp elements, then it should use space at least $\Omega(\log^2 W/\epsilon)$. Since the synchronous stream is a special case of an asynchronous stream, the above lower bound of $\Omega(\log^2 W/\epsilon)$ applies to approximate basic counting over asynchronous streams too. To compare our results for basic counting with this lower bound, let us consider the case when the timestamps of the elements are unique. In such a case, $\log B = O(\log W)$, since the value of the basic count cannot exceed W, and thus the space required by our summary is $O(\log^3 W/\epsilon)$.

Techniques. Our algorithm for basic counting is based on a novel data structure that we call a *splittable histogram.* The data structure consists of a small number of histograms that summarize the elements within the sliding window at various granularities. Within each histogram, the elements in the sliding window are grouped into buckets, that are each responsible for a certain range of timestamps.

Arriving elements are placed in appropriate buckets within this histogram. When a bucket becomes “heavy”, i.e. gets too many elements, it is *split* in half to produce two buckets of smaller sizes, each responsible for a smaller range of timestamps. Buckets may be recursively split if the again become too heavy due to future insertions. A key technical ingredient is the analysis of the error resulting from this recursive splitting of buckets. In contrast, earlier uses of histograms in processing data streams over a sliding window, for example, Datar *et al.* [5] and Arasu and Manku [1] have all been based on *merging* smaller histogram buckets into larger ones, rather than splitting them as we do here.

Comparison to Prior Work. Prior to our work, deterministic algorithms were known for summarizing synchronous streams over a sliding window [5, 7], but only randomized algorithms were known for summarizing asynchronous streams. In a previous work, Tirthapura, Xu and Busch [12] presented randomized algorithms for summarizing asynchronous streams over a sliding window. Their summary yields an (ϵ, δ)-approximation for the *sum* problem and for basic counting, i.e. the answer returned is within a relative error ϵ of the actual answer with probability at least $1 - \delta$; this is a weaker guarantee that is provided by the deterministic algorithm. The space used by their algorithm for the sum is $O((\frac{1}{\epsilon^2})(\log \frac{1}{\delta})(\log W \log B))$, where W is a bound on the maximum window size, B is an upper bound on the value of the sum, ϵ is the relative error, and δ is the failure probability. When compared with our deterministic algorithm, which uses space $O(\log W \cdot \log B \cdot (\log W + \log B)/\epsilon)$, the randomized algorithm arguably takes more space, since $(\log W + \log B)$ is typically smaller than $(\frac{1}{\epsilon})(\log \frac{1}{\delta})$. Thus, the deterministic algorithm that we present here not only gives a stronger guarantee than the randomized one but also (arguably) uses lesser space. Nevertheless, the randomized algorithm in [12] has the advantage of being more flexible and it can be used for other aggregates, including the median and quantiles.

Related Work. With the exception of [12], earlier work on summarizing data streams over a sliding window have all considered the case of *synchronous* streams, where the stream elements appear in increasing order of timestamps. Datar *et al.* [5] were the first to consider basic counting over a sliding window under synchronous arrivals. They present a deterministic algorithm for summarizing synchronous streams which is based on a data structure called the *exponential histogram.* This summary can give an ϵ-approximate answer for basic counting, sum and other aggregates. For a sliding window size of maximum size W, and an ϵ relative error, the space taken by the exponential histogram for basic counting is $O(\frac{1}{\epsilon} \log^2 W)$, and the time taken to process each element is $O(\log W)$ worst case, and $O(1)$ amortized. Their summary for the sum of elements within the sliding window has space complexity $O(\frac{1}{\epsilon} \log W(\log W + \log m))$, and worst case time complexity of $O(\log W + \log m)$ where m is an upper bound on the value of an item. Gibbons and Tirthapura [7] gave an improved algorithm for basic counting that uses the same space as in [5], but whose time per element is $O(1)$ worst case. Since then, there has been much work on summarizing synchronous data streams to approximate various aggregates over a sliding window, including Arasu and Manku [1] on frequency counts and quantiles, Babcock *et al.* [2] on variance and k-medians, Feigenbaum *et al.* [6] on the diameter of a set of points. Much other recent work on data stream algorithms has been surveyed in [10].

2 Basic Counting

For basic counting, the values of the stream elements do not matter, so the stream is essentially a sequence of timestamps $R = t_1, t_2, \ldots, t_n$. The timestamps may not be distinct and do not necessarily arrive in an increasing order. Let c denote the current time. The goal is to maintain a sketch of R which will provide an answer for the following query: for a user-provided $w \leq W$, which is given at the time of the query, return the number of elements in the current timestamp window $[c - w, c]$.

2.1 Algorithm

We assume that timestamps are non-negative integers. The universe of possible timestamps is divided into intervals $I_0, I_1, \ldots$ of length W each; $I_0 = [0, W - 1], I_1 = [W, 2W - 1], \ldots, I_k = [kW, (k+1)W - 1], \ldots$. A separate data structure D_i is maintained for each interval I_i, and all timestamps belonging in I_i are inserted into D_i. If c is the current time, then any timestamp that is less than $c - W$ will never be useful for a query, whether current or future. Thus we only need to maintain data structures D_i for those I_i that intersect $[c - W, c]$. It is easy to verify that there exists $j \geq 0$ such that $[c - W, c] \subset I_j \cup I_{j+1}$. Thus, the only data structures that are needed at time c are D_j and D_{j+1}, and the algorithm only needs to maintain two such data structures at any time. When a query is asked for the basic count over a timestamp window of width $w \leq W$, there are two possibilities:

(1)The window $[c - w, c]$ is completely contained within I_j, i.e. $[c - w, c] \subseteq I_j$. In this case D_j is queried for the number of elements in the range $[c - w, c]$, and this estimate is returned by the algorithm.

(2)The window $[c - w, c]$ falls partially in I_j and in I_{j+1}. In such a case, the algorithm consults D_j for the number of elements in the range $[c - w, (j+1)W - 1]$ and consults D_{j+1} for the number of timestamps in the range $[(j + 1)W, c]$, and returns the sum of the two estimates. If each estimate is within an ϵ relative error of the correct value, their sum is also within an ϵ relative error of the total number of elements in the sliding timestamp window.

In the remainder of this section, we discuss the algorithms for maintaining and querying data structure D_0. Other D_is can be maintained similarly. For D_0 we assume that all timestamps are in the range $[0, W - 1]$. Without loss of generality, we assume W is a power of 2 (since W only needs to be an upper bound on the window size, it is always acceptable to increase it without affecting the correctness). Let B be an upper bound on the number of elements with timestamps in $I_0 = [0, W - 1]$. Let $M = \lceil \log B \rceil$, and $\alpha = \left\lceil (1 + \log W) \cdot \frac{2+\epsilon}{\epsilon} \right\rceil$ where ϵ is the desired relative error.

Intuition: Our algorithm is based on a novel data structure *splittable histogram*, which we introduce here. Data structure D_0 consists of $M+1$ histograms $S_0, S_1, \ldots, S_M$. Each histogram S_i consists of no more than α *buckets.* Each bucket in S_i is a tuple $b = \langle w(b), l(b), r(b) \rangle$ where: which is (1)$[l(b), r(b)] \subseteq [0, W-1]$ is the range of all timestamps the bucket is responsible for and (2)$w(b)$ is the *weight* of the bucket which is an estimate of the number of elements with timestamps in the range $[l(b), r(b)]$.

The timestamp ranges of different buckets within S_i are disjoint. For each $i = 0, \ldots, M$, we maintain the following invariant for S_i: *If S_i has two or more buckets, then the weight of every bucket in S_i is in the range $[2^i, 2^{i+1}-1]$, except for those buckets which are responsible for a single timestamp.* Intuitively, if $i_1 > i_2$, then histogram S_{i_1} contains "coarser' information about the distribution of elements than does S_{i_2}, since it uses buckets of a larger size. Modulo some significant details, this setup is similar to the one used in Datar *et. al.* [5] and Gibbons and Tirthapura [7] to process synchronous streams.

An arriving element with timestamp t is inserted into every S_i, $i = 0, \ldots, M$. Within S_i, the element is inserted into a bucket b which is responsible for the timestamp of the element ($t \in [l(b), r(b)]$), and the weight $w(b)$ of the bucket is incremented. Since stream elements are arriving asynchronously, the bucket into which the arriving element is inserted may not be the bucket responsible for the most recent timestamps. This is a fundamental departure from the way histograms were employed to process synchronous streams in previous work [5, 7]. The algorithms in [5, 7] rest on the fact that an arriving element is always inserted into the most recent bucket. Thus, when the size of the most recent bucket exceeds 2^i, the most recent bucket is "closed" and a new bucket is created to hold future elements.

In our case, since elements arriving in the future may fall into a bucket which is not the most recent bucket, we are unable to "close" a bucket. Thus, due to arrival of elements in an arbitrary order, the weight of a bucket may increase and may reach 2^{i+1}, causing it to become too heavy. A heavy bucket of the form $\langle 2^{i+1}, l, r \rangle$ is "split" into two lighter buckets $\langle 2^i, l, (l+r+1)/2 - 1 \rangle$ and $\langle 2^{i+1}, (l+r+1)/2, r \rangle$, each of which has half the weight of the original bucket, and is responsible for half the timestamp range of the original bucket.

Clearly, this splitting is inaccurate, since in the earlier grouping of all 2^{i+1} elements into a single bucket, the information about the timestamps of the individual elements has already been lost, and assigning half the elements of the bucket into half the timestamp range may be incorrect. The key intuition here is that *the error due to this split is controlled, and is no more than 2^i at each bucket resulting from the split.* Any future insertions of elements in the timestamp range $[l, r]$ are considered more carefully, since they are being inserted into buckets whose timestamp ranges are smaller. The buckets resulting from the split may further increase in weight due to future insertions, and may split recursively. The error due to splitting may accumulate, but only to a limited extent, as we prove. A bucket resulting from $\log W$ recursive splits is responsible for only a single timestamp, since the range of timestamps for a bucket decreases by a factor of 2 during every split, and the initial bucket is responsible for a timestamp range of length W. A bucket that is responsible for a single timestamp is treated as a special case, and is not split further, even if its weight increases beyond 2^{i+1}.

Due to the splits, the number of buckets within S_i may increase beyond α, in which case we only maintain the α buckets that are responsible for the most recent timestamps. Given a query for the basic count in window $[c-w, c]$, the different S_is are examined in increasing order of i. For smaller values of i, S_i may have already discarded some buckets that are responsible for timestamps in $[c-w, c]$. But, there will always be a level $\ell \leq M$ that will have all buckets intersecting the range $[c-w, c]$ (this is formally proved in Lemma 2). The algorithm selects the earliest such level to answer the basic counting query, and we show that the resulting relative error is within ϵ.

The algorithm for basic counting is given below. Algorithm 1 describes the initialization of the data structure, Algorithm 2 describes the steps taken to process a new element with a timestamp t, and Algorithm 3 describes the procedure for answering a query for basic count.

Algorithm 1: Basic Counting: Initialization

```
$\alpha \leftarrow \left\lceil (1 + \log W) \cdot \frac{2+\epsilon}{\epsilon} \right\rceil$, where $\epsilon$ is the desired relative error;
$S_0 \leftarrow \phi$; $T_0 \leftarrow -1$;
for $i = 1, \ldots, M$ do
    $S_i$ is a set with a single element $\langle 0, 0, W-1 \rangle$;
    $T_i \leftarrow -1$;
end
```

2.2 Proof of Correctness

Let c denote the current time. We consider the contents of sets S_i and the values of T_i at time c. For any time t, $0 \leq t \leq c$, let s_t denote the number of elements with timestamps in the range $[t, W-1]$ which arrive until time c. For level i, $0 \leq i \leq M$, e_t^i is defined as follows.

Definition 2.

$$e_t^i = \sum_{\{b \in S_i | l(b) \geq t\}} w(b)$$

Lemma 1. *For any level $i \in [0, M]$, for any t such that $T_i < t \leq c$, $|s_t - e_t^i| \leq 2^i \cdot (1 + \log W)$*

Proof. For level $i = 0$ we have $s_t = e_t^0$, since each element x with timestamp t', where $t \leq t' \leq W-1$, is counted in the bucket $b = \langle w(b), t', t' \rangle$ which is a member of S_0 at time c. Thus, $|s_t - e_t^0| = 0$.

Consider now some level $i > 0$. We can construct a binary tree A whose nodes are all the buckets that appeared in S_i up to current time c. Let $b_0 = \langle 0, 0, W-1 \rangle$

Algorithm 2: Basic Counting: When an element with timestamp t arrives

```
// level 0
if there is bucket $\langle w(b), t, t\rangle \in S_0$ then
    Increment $w(b)$;
else
    Insert bucket $\langle 1, t, t\rangle$ into $S_0$;
end
// level $i$, $i > 0$
for $i = 1, \ldots, M$ do
    if there is bucket $b = \langle w(b), l(b), r(b)\rangle \in S_i$ with $t \in [l(b), r(b)]$ then
        Increment $w(b)$;
        if $w(b) = 2^{i+1}$ and $l(b) \neq r(b)$ then
            // bucket too heavy, split
            // note that a bucket is not split
            // if it is responsible for only a single time stamp
            New bucket $b_1 = \langle 2^i, l(b), \frac{l(b)+r(b)+1}{2} - 1\rangle$;
            New bucket $b_2 = \langle 2^i, \frac{l(b)+r(b)+1}{2}, r(b)\rangle$;
            Delete $b$ from $S_i$;
            Insert $b_1$ and $b_2$ into $S_i$;
        end
    end
end
// handle overflow
for $i = 0, \ldots, M$ do
    if $|S_i| > \alpha$ then
        // overflow
        Discard bucket $b^* \in S_i$ such that $r(b^*) = \min_{b \in S_i} r(b)$;
        $T_i \leftarrow r(b^*)$;
    end
end
```

Algorithm 3: Basic Counting: Query(w)

```
Input: $w$, the width of the query window, where $w \leq W$
Output: An estimate of the number of elements with timestamps in $[c - w, c]$
        where $c$ is the current time
Let $\ell \in [0, \ldots, M]$ be the smallest integer such that $T_\ell < c - w$;
return $\sum_{\{b \in S_\ell | l(b) \geq c-w\}} w(b)$;
```

be the initial bucket which is inserted into S_i during initialization (Algorithm 1). The root of A is b_0. For any bucket $b \in A$, if b is split into two buckets b_l and b_r, then b_l and b_r will appear as the respective left and right children of b in A. Note that in A a node is either a leaf or has exactly two children. Tree A has depth at most $\log W$ (the root is at depth 0), since every time that a bucket splits the time period divides in half, and the smallest time period is a discrete time step. For any node $b \in A$ let $A(b)$ denote the subtree with root b; we will also refer to this as the subtree of b.

Consider now the tree A at time c. The buckets in S_i appear as the $|S_i|$ rightmost leaves of A. Let S'_i denote the set of buckets in S_i with $l(b) \geq t$. clearly, $e^i_t = \sum_{b \in S'_i} w(b)$. The buckets in S'_i are the $|S'_i|$ rightmost leaves of A. Suppose that $S'_i \neq \emptyset$ (the case $S'_i = \emptyset$ is discussed below). Let b' be the leftmost leaf in A among the buckets in S'_i. Let p denote the path in A from the root to b'. For the number of nodes $|p|$ of p it holds $|p| \leq 1 + \log W$. Let H_1 (H_2) be the set that consists of the right (left) children of the nodes in p, such that these children are not members of the path p. Note that $b' \notin H_1 \cup H_2$. The union of b' and the leaves in the subtrees of H_1 ($\cup_{b \in H_2} A(b)$) constitute the nodes in S'_i. Further, each bucket $b \notin S'_i$ is in a leaf in a subtree of H_2.

Consider some element x with timestamp t'. Initially, when x arrives it is *initially assigned* to the bucket b which t' belongs to. If b splits to two (children) buckets b_1 and b_2, then we can assume that x is *assigned* arbitrarily to one of the two new buckets arbitrarily. Even through x's timestamp may belong to b_1, x may be assigned to b_2, and vice-versa. If again the new bucket splits, x is assigned to one of its children, and so on. Note that x is always assigned to a leaf of A.

At time c, we can write

$$e^i_t = s_t + |X_1| - |X_2 \cup X_3|, \tag{1}$$

such that: X_1 is the set of elements with timestamps in $[0, t-1]$ which are assigned to buckets in S'_i; X_2 is the set of elements with timestamps in $[l(b'), W-1]$ which are assigned to buckets outside of S'_i; and, for $t < l(b')$, X_3 is the set of elements with timestamps in $[t, l(b')-1]$ which are assigned to buckets outside of S'_i, while for $t = l(b')$, $X_3 = \emptyset$. Note that the sets X_1, X_2, X_3 are disjoint.

First, we bound $|X_1|$. Consider some element $x \in X_1$ with timestamp in $[0, t-1]$ which at time c appears assigned to a leaf bucket $b_l \in S'_i$. Since $b_l \in S'_i$, t cannot be a member of the time range of b_l, that is, $t \notin [l(b_l), r(b_l)]$. Thus, x could not have been initially assigned to b_l. Suppose that $b_l \neq b'$. Then, there is a node $\widehat{b} \in H_1$ such that b_l is the leaf of the subtree $A(\widehat{b})$. None of the nodes in $A(\widehat{b})$ contain t in their time range, since all the leaves of $A(\widehat{b})$ are members of S'_i. Therefore, x could not have been initially assigned to $A(\widehat{b})$. Thus, x is initially assigned to a node $b_p \in p' = p - \{b'\}$, since x could not have been assigned to any node in the subtrees of H_2 which would certainly bring x outside of S'_i. Similarly, if $b_l \neq b'$, x is initially assigned to a node $b_p \in p'$. Since at most 2^{i+1} elements are initially assigned to the root, and at most 2^i elements are initially assigned to each of the subsequent nodes of p', we get:

$$|X_1| \leq 2^i \cdot (|p'| - 1) + 2^{i+1} = 2^i \cdot |p| \leq 2^i \cdot (1 + \log W). \tag{2}$$

With a similar analysis (the details are omitted due to space constraints) in can be shown that:

$$|X_2 \cup X_3| \leq 2^i \cdot (1 + \log W). \tag{3}$$

Combining Equations 1, 2, and 3 we can bound $s_t - e_t^i$:

$$-2^i \cdot (1 + \log W) \leq -|X_1| \leq \; s_t - e_t^i \; \leq |X_2 \cup X_3| \leq 2^i \cdot (1 + \log W).$$

Therefore, $|s_t - e_t^i| \leq 2^i \cdot (1 + \log W)$. In case $S_i' = \emptyset$, $e_t^i = s_t - |X_3| = 0$, and the same bound follows immediately. □

Lemma 2. *When asked for an estimate of the number of timestamps in $[c-w, c]$*
(1)There exists a level $i \in [0, M]$ such that $T_i < c - w$, and
(2)Algorithm 3 returns e_{c-w}^ℓ where $\ell \in [0, M]$ is the smallest level such that $T_\ell < c - w$.

The proof of Lemma 2 is omitted due to space constraints, and can be found in the full version [3]. Let ℓ denote the level used by Algorithm 3 to answer a query for the number of timestamps in $[c - w, c]$. From Lemma 2 we know ℓ always exists.

Lemma 3. *If $\ell > 0$, then $s_{c-w} \geq \frac{(1+\log W)\cdot 2^\ell}{\epsilon}$.*

Proof. If $\ell > 0$, it must be true that $T_{\ell-1} \geq c - w$, since otherwise level $\ell - 1$ would have been chosen. Let $t = T_{\ell-1} + 1$. Then, $t > c - w$, and thus $s_{c-w} \geq s_t$. From Lemma 1, we know $s_t \geq e_t^{\ell-1} - (1 + \log W) \cdot 2^{\ell-1}$. Thus we have:

$$s_{c-w} \geq e_t^{\ell-1} - (1 + \log W) \cdot 2^{\ell-1} \tag{4}$$

We know that for each bucket $b \in S_{\ell-1}$, $l(b) \geq t$. Further we know that each bucket in $S_{\ell-1}$ has a weight of at least $2^{\ell-1}$ (only the initial bucket in $S_{\ell-1}$ may have a smaller weight, but this bucket must have split, since otherwise $T_{\ell-1}$ would still be -1). Since there are α buckets in $S_{\ell-1}$, we have:

$$e_t^{\ell-1} \geq \alpha 2^{\ell-1} \geq (1 + \log W) \cdot \frac{2+\epsilon}{\epsilon} \cdot 2^{\ell-1} \tag{5}$$

The lemma follows from Equations 4 and 5. □

Theorem 1. *The answer returned by Algorithm 3 is within an ϵ relative error of s_{c-w}.*

Proof. Let X denote the value returned by Algorithm 3. If $\ell = 0$, it can be verified that Algorithm 3 returns exactly s_{c-w} (proof omitted due to space constraints). If $\ell > 0$, from Lemmas 1 and 2, we have $|X - s_{c-w}| \leq (1 + \log W) \cdot 2^\ell$. Using Lemma 3, we get $|X - s_{c-w}| \leq \epsilon \cdot s_{c-w}$ as needed. □

Theorem 2. *The worst case space required by the data structure for basic counting is $O((\log W \cdot \log B) \cdot (\log W + \log B)/\epsilon)$ where B is an upper bound on the value of the basic count, W is an upper bound on the window size w, and ϵ is the desired upper bound on the relative error. The worst case time taken by Algorithm 2 to process a new element is $O(\log W \cdot \log B)$, and the worst case time taken by Algorithm 3 to answer a query for basic counting is $O(\log B + \frac{\log W}{\epsilon})$.*

The proof is omitted due to space constraints, and can be found in the full version [3].

3 Sum of Positive Integers

We now consider the maintenance of a sketch for the *sum*, which is a generalization of basic counting. The stream is a sequence of tuples $R = d_1 = (v_1, t_1), d_2 = (v_2, t_2), \ldots, d_n = (v_n, t_n)$ where the v_is are positive integers, corresponding to the observations, and t_is are the timestamps of the observations. Let c denote the current time. The goal is to maintain a sketch of R which will provide an answer for the following query. For a user provided $w \leq W$ that is given at the time of the query, return the sum of the values of stream elements that are within the current timestamp window $[c-w, c]$. Clearly, basic counting is a special case where all v_is are equal to 1.

An arriving element (v, t), is treated as v different elements each of value 1 and timestamp t, and these v elements are inserted into the data structure for basic counting. Finally, when asked for an estimate for the sum, the algorithm for handling a query in basic counting (Algorithm 3) is used. The correctness of this algorithm for the sum follows from the correctness of the basic counting algorithm (Theorem 1). The space complexity of this algorithm is the same as the space complexity of basic counting, the only difference being that the number of levels in the algorithm for the sum is $M = \lceil \log B \rceil$, where B is an upper bound on the value of the sum within the sliding window (in the case of basic counting, B was an upper bound on the number of elements within the window).

If naively executed, the time complexity of the above procedure for processing an element (v, t) could be large, since v could be large. The time complexity of processing an element can be reduced by directly computing the final state of the basic counting data structure after inserting all the v elements. The intuition behind the faster processing is as follows. The element (v, t) is inserted into each of the $M + 1$ levels. In each level $i, i = 0, \ldots, M$, the v elements are inserted into S_i in *batches* of unit elements $(1, t)$ taken from (v, t). A batch contains enough elements to cause the current bucket containing timestamp t to split. The next batch contains enough elements from v to cause the new bucket containing timestamp t to split, too, and so on. The process repeats until a bucket containing timestamp t cannot split further. This occurs when at most $O(\max(v/2^i, \log W))$ batches are processed (and a similar number of respective new buckets is created), since at most $O(2^i)$ elements from v are processed at each iteration in a batch, and a bucket can be recursively split at most $\log W$

times until it is responsible for only one timestamp, at which point no further splitting can occur (and any remaining elements are directly inserted into this bucket). The complete algorithm for processing (v, t) and its analysis can be found in the full version of the paper [3], where it is proved that upon receiving element (v, t), the algorithm for the sum simulates the behavior of Algorithm 2 upon receiving v elements each with a timestamp of t.

Theorem 3. *The worst case space required by the data structure for the sum is $O((\log W \cdot \log B)(\log W + \log B)/\epsilon)$ bits where B is an upper bound on the value of the sum, W is an upper bound on the window size w, and ϵ is the desired upper bound on the relative error. The worst case time taken by the algorithm for the sum to process a new element is $O(\log W \cdot \log B)$, and the time taken to answer a query for the sum is $O(\log B + (\log W)/\epsilon)$.*

References

1. A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 286–296, 2004.
2. B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *Proc. 22nd ACM Symp. on Principles of Database Systems (PODS)*, pages 234–243, June 2003.
3. C. Busch and S. Tirthapura. A deterministic algorithm for summarizing asynchronous streams over a sliding window. Technical report, Iowa State University, 2006. Available at `http://archives.ece.iastate.edu/view/year/2006.html`.
4. G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In *Proc. ACM Symposium on Principles of Database Systems*, pages 263–272, 2006.
5. M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.
6. J. Feigenbaum, S. Kannan, and J. Zhang. Computing diameter in the streaming and sliding-window models. *Algorithmica*, 41:25–41, 2005.
7. P. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. *Theory of Computing Systems*, 37:457–478, 2004.
8. S. Guha, D. Gunopulos, and N. Koudas. Correlating synchronous and asynchronous data streams. In *Proc.9th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 529–534, 2003.
9. A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 767–778, 2005.
10. S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Foundations and Trends in Theoretical Computer Science. Now Publishers, August 2005.
11. U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. 23rd ACM Symposium on Principles of Database Systems (PODS)*, pages 263–274, 2004.
12. S. Tirthapura, B. Xu, and C. Busch. Sketching asynchronous streams over a sliding window. In *Proc. 25th annual ACM symposium on Principles of distributed computing (PODC)*, pages 82–91, 2006.