Improved algorithms for Polynomial-Time Decay and Time-Decay with additive error

Tsvi Kopelowitz and Ely Porat

Department of Computer Science
Bar-Ilan University, Ramat-Gan 52900, Israel
{kopelot,porately}@cs.biu.ac.il

Abstract. We consider the problem of maintaining polynomial and exponential decay aggregates of a data stream, where the weight of values seen from the stream diminishes as time elapses. These types of aggregation were discussed by Cohen and Strauss in [2], and can be used in many applications in which the relative value of streaming data decreases since the time the data was seen. Some recent work and space efficient algorithms were developed for time-decaying aggregations, and in particular polynomial and exponential decaying aggregations. All of the work done so far has maintained multiplicative approximations for the aggregates. In this paper we present the first $O(\log N)$ space algorithm for the polynomial decay under a multiplicative approximation, matching a lower bound. In addition, we explore and develop algorithms and lower bounds for approximations allowing an additive error in addition to the multiplicative error. We show that in some cases, allowing an additive error can decrease the amount of space required, while in other cases we cannot do any better than a solution without additive error.

Keywords: streaming algorithms, weighted aggregates, sketches.

1 Introduction

In many recent applications there exists a great need to manage systems with high speed communication and massive data sets. In such applications it is more appropriate to use the data stream model in which data arrives rapidly and needs to be processed by an algorithm that lacks the needed space in order to store all of the streaming data. The algorithm typically stores a synopsis or summary of the data using space that is much less than the amount of data arriving from the stream. Using the synopsis the algorithm can answer queries regarding the data. However, generally there is a tradeoff between the size of the synopsis and the precision of the answers.

Consider the problem of maintaining the sum of all data seen by a stream. This can be answered exactly using $\Theta(\log S)$ bits, where S is the value of the sum. Morris in [8] showed how to approximate the sum using $O(\log \log S)$ bits.

However, in many applications the weight of data diminishes with time, and therefore older data should weigh less towards the sum of a stream relative to

newer data. In some applications we might be interested in the sum over some recent time frame. We can use a *decay function*, dependent on elapsed time, in order to determine the weight of each item. A *decayed sum* is a weighted sum of the data in accordance to the decay function.

In some cases we are interested in a *decayed average* of a stream, which is simply the decayed sum divided by the number of element in the stream. However, by approximating the decayed sum of a stream, and by approximately counting the number of elements seen in the stream (using [8] for example), we can easily approximate the decayed average.

1.1 Applications

We list some applications in which the decayed aggregation or decayed averages are used. Many other applications that use the data stream model can conceivably use the decayed sum scheme for their purpose. We note that most of the examples were originally mentioned in [2].

- **Detecting Fraud Transactions by Credit Cards.** Credit card companies use recent behavior of costumers in order to detect whether a particular transaction deviates considerably from a pattern of transaction history of a given costumer [4]. One way of detecting such frauds would be to compute a decayed sum of the transactions by giving a higher weight to recent transactions, and thus detecting a variance in the pattern of transactions of a given user.
- **Patterns of AT&T Telecom Costumers.** As mentioned in [3] and [2], AT&T has an application in which they maintain statistics of approximately 100 million costumers. It is vital to balance the weight of information and available storage space in such an application (see [3] for more details).
- **The Random Early Detection (RED) Protocol.** Many routers on the internet use the RED protocol for avoiding congestions and controlling paths of data. In order to estimate the congestion, RED uses a weighted average of previous queue lengths [?],[5],[2].
- **Selecting Internet Gateways.** In the internet graph, there are many paths from a given node to each destination. When selecting a path on which one would like to send information, reliability of that path plays a major role. A decayed average of previous reliability measures of each path can be used as an estimation of the reliability of each path [2, 1].

1.2 Decay Functions

We recall the definitions given by Cohen and Strauss in [2] for the decay functions we are interested in. Consider a stream of data where $f(t) \geq 0$ is the item value of the stream obtained at time t . For sake of simplicity we assume our stream

only receives values at discrete times, and therefore, t is integral. We define a *decay function* $g(x) \geq 0$ defined for $x \geq 0$ to be a non-increasing function. At time T the weight of the item that arrived at time $t \leq T$ is $g(T - t)$ and the *decayed value* of that item is $f(t)g(T - t)$. We are interested in obtaining the *decayed sum* of $f(t)$ under the decay function $g(x)$ that is defined as follows:

Problem 1. Decayed Sum Problem (DSP). *Given a stream $f(t)$ and a decay function $g(T - t)$ estimate at any time T the decayed sum that is given by:*

$$V_g(T) = \sum_{t \leq T} f(t)g(T - t). \quad (1)$$

When $f(t)$ has binary values, we refer to the DSP as the *Decayed Count Problem* (DCP).

We will be interested in three types of decays. The first one is known as the *Exponential Decay* (ExpD) where for a given parameter $\lambda > 0$ we have $g(t) = 1/\lambda^t$. The other type of decay is the *Polynomial Decay* (PolyD) where for a given parameter $\alpha > 0$ we have $g(t) = 1/t^\alpha$.

Sliding windows are also a family of decay functions that have been discussed in [4] by Datar et al. For a window of size W we have the decay function $g(x)$ such that for $x \leq W$, $g(x) = 1$, and zero otherwise. All of the data in the recent time frame of size W have equal weight, while all of the data not in this time frame has no weight at all. We refer to the decayed sum of a sliding window of size W as V_{SLIWIN_W} .

1.3 Related and Previous Work

As mentioned by Cohen and Strauss in [2], one of the most intuitive and common decays is the exponential decay that can be easily maintained in $\Theta(\log N)$ space (throughout this paper we assume that N is the amount of time elapsed) by the formula $C \leftarrow (1 - w)x + wC$, where x is the value of the new data and $0 < w < 1$ determines the decay.

Datar et al in [4] showed an algorithm and a matching lower bound of $\Theta(\log^2 N)$ bits needed for approximating sliding windows up to some constant multiplicative factor. A more detailed explanation of their techniques is in Sect. 3. Gibbons and Tirthapura in [6] extended sliding windows for distributed streams.

Cohen and Strauss in [2] presented an $O(\log^2 N)$ space algorithm for approximately maintaining general decay functions up to some constant multiplicative factor. This showed that sliding window decay is the hardest decay function, in the sense that its space lower bound meets the general decay function space upper bound. They also presented an $O(\log N \log \log N)$ space algorithm for approximating polynomial decay up to some constant multiplicative factor, and showed a lower bound of $\Omega(\log N)$ bits of space.

In addition, Cohen and Strauss in [2] argue that the exponential decay and sliding window decay are insufficient for many applications in the sense that they

both strongly disregard old data. One expects the weight of data to diminish with time, however, for severe events we don't want the value to diminish entirely. We present an example from [2] that illustrates this.

We wish to measure the availability of network links using a time decay function. Consider two links, L1 and L2. Assume that L1 fails for the duration of five time units between time units 5 and 10, while L2 fails for the duration of half a time unit, starting from time unit 34. In addition assume no more failures occur in either of the links. As time progresses we expect L2 to be considered a more reliable link as opposed to L1.

If we use sliding window decay, we might miss out on the failure of L1 (if the window size is too small), and assume L1 is a more reliable link. If we use exponential decay we notice that the ratio between the weight of L1's failure and the weight of L2's failure remains fixed as time progresses. This means that whatever was considered more reliable at time unit 35 (that can be either one of the links) will be considered more reliable forever.

Ultimately, neither the sliding window decay nor the exponential decay can provide us with a view in which we can guarantee that L1 will be considered more reliable. However, polynomial decay has the property that as time progresses, items seen in the same time vicinity are given approximately the same weight. Using Polynomial decay we can obtain a view in which L1 will be considered more reliable as time progresses. Also, polynomial decay functions enhance our ability to tune the rate of decay. This gives a strong motivation for studying polynomial decay aggregates

1.4 Our Results

In this paper we present the following. We show the first $O(\log N)$ space algorithm for approximating polynomial decay up to some constant multiplicative factor. This matches the bound obtained by Cohen and Strauss in [2]. In addition, we provide optimal space algorithms for approximating sliding window decay, general decay, and polynomial decay, when the values of the stream are only bounded by some polynomial of N .

We also explore another model of approximation that allows an additive error in addition to the multiplicative one. We show that in this model, the amount of space necessary in the Exponential decay can be reduced due to the additional freedom we get from the additive error. Finally, we show that generally, in Polynomial decay, the amount of space required cannot be reduced (we prove a lower bound of $\Omega(\log n)$ space), unless the values of the stream are bounded by a constant, and the sum of the decay function converges.

The paper is structured as follows. In Sect. 2 we provide the definitions for the decay functions that we will use throughout the paper. In Sect. 3 we present some known data structures used for approximating decay sums. In Sect. 4 we present a new data structure that we use to achieve an $O(\log N)$ space algorithm for polynomial decay up to some constant multiplicative factor. In Sect. 5 we define the model that allows more slackness of an additive error, and we analyze the exponential and polynomial decays under the model presented including a

lower bound for some cases of the polynomial decay.

2 Data Structures for Decayed Sums

We now present the data structures we will use in this paper. We first provide an overview of Exponential Histograms that were originally introduced by Datar et al in [4], and then continue on with a review on Weight-Based Merging Histograms, that were introduced in [2].

2.1 The Exponential Histogram

The *Exponential Histogram* (EH) was originally introduced in [4] in order to approximate the decayed sum of sliding windows over binary streams. For a given constant k , called the *parameter* of the EH, we define the EH for a binary stream $f(t)$ as a collection of buckets that are formed as follows. When the first non-zero value arrives we open a bucket and insert that value into it. That bucket is immediately sealed. We continue creating buckets in the same manner until we receive the $(k + 1)$ 'th non-zero value. At this point we merge the earliest two buckets (i.e. the two buckets that were created first) into one bucket of size 2, and create a new bucket for the newly arrived non-zero value. We continue in this manner so that whenever we have $k + 1$ buckets of size 1 we merge the earliest two buckets of size 1 into one bucket of size 2. Recursively, whenever we have $k + 1$ buckets of size 2^i we merge the two least recent ones of size 2^i into one bucket of size 2^{i+1} . Clearly, this gives a total of $O(\log N)$ buckets.

For each bucket we maintain a *timestamp* which is the time elapsed since the latest value in the bucket has arrived. The timestamp requires $O(\log N)$ space per bucket as the amount of time that has elapsed is at most N . We also maintain the size (the number of ones) of each bucket. This can be done using only one bit per bucket as for each bucket we just need to note whether it is the same size as the previous one, or double the size. This gives a total of $O(\log^2 N)$ bits of space.

When approximating sliding window decay, we sum up the counts for all of the buckets with *timestamp* inside the window. In case there is a bucket with some values inside the window and some outside, we add only half the count for that bucket. The absolute error in the estimate for the sliding window decay is half the size of the last bucket. As proven by Datar et al in [4], the approximation for the sliding window decay using the EH is $(1 + 1/k)$, so letting $k = 1/\epsilon$ we can get a multiplicative approximation of $1 + \epsilon$ for any constant ϵ . Note that one EH for a SW of size W can be used to calculate any SW of size less than W .

It was shown in [2] how an EH can be used to approximate the value of a stream over any decay function, by maintaining the EH for the entire stream. This can be seen by the following that was proven in [2]:

$$V_g(T) = g(N)V_{SLIWIN_N} + \sum_{i=1}^{N-1} (g(N-i) - g(N+1-i))V_{SLIWIN_{N-i}}. \quad (2)$$

As mentioned in [2], being that the EH gives us a $(1 + \epsilon)$ approximation for every sliding window, we get a $(1 + \epsilon)$ approximation for $V_g(T)$.

We will later show how the EH can be extended to work for polynomial bounded streams (i.e. the value $f(t)$ is bounded by some polynomial of N), while still using $O(\log^2 N)$ space. This will immediately imply an approximation for decayed sums over any decay function and polynomial bounded streams using $O(\log^2 N)$ space.

2.2 Weight-Based Merging Histogram

The Weight-Based Merging Histogram (WBMH) was originally introduced by Cohen and Strauss in [2]. Like the EH, the WBMH also aggregates values into buckets. However, the boundaries of the buckets are only dependent on the decay function at hand, and not on a particular stream. This means that the timestamps do not need to be stored. They are set by the decay function that is assumed to be known.

The WBMH utilizes the fact that if a decay function has the property that the ratio $g(x)/g(x + \Delta)$ is non-increasing with x for any time frame Δ then the ratio of two items remains fixed, or approaches one as time advances. The meaning of such a property is that as time progresses, items in larger vicinities have the same decayed weight up to a multiplicative factor, so we can group those values together. Say we want a multiplicative approximation for the DCP of of factor $1 \pm \epsilon$. For convenience we let $b_0 = 1$. We let b_1 be the maximum value such that $(1 + \epsilon)g(b_1 - 1) \geq g(1)$. In the same manner, let b_i be the maximum value such that $(1 + \epsilon)g(b_i - 1) \geq g(b_{i-1})$. The values of b_i are dependent on the decay function, and therefore (as we previously mentioned) they do not need to be stored. Each time a new value arrives it is added to the "current" bucket. At any time T where $T \equiv 0 \pmod{b_1}$ we seal the "current" bucket, and open a new one. Whenever there exists an i and two buckets such that the two buckets are within a region $[b_i, b_{i+1} - 1]$, we merge the two buckets into one.

When we wish to solve the DCP we use the WBMH as follows. First, we note that due to the method in which we picked our boundaries we have that the decay of all of the items in the same bucket is within $1 \pm \epsilon$ of the decay of the elapsed time from when the latest or earliest value in the bucket has arrived. Thus, we can compute the decayed sum of all of the items in a bucket by multiplying the number of non-zero values in the bucket with decayed weight of the latest or first item of the bucket (we know the start and end time from the decay function as explained above). Now, in order to compute the approximate decayed value of the stream we simply sum up the approximate decayed value of each of the buckets.

One family of decay functions with the property that the ratio $g(x)/g(x + \Delta)$ is non-increasing with x for any time frame Δ is PolyD in which we have $s = O(\log N)$ different b_i 's for each function. For each bucket we hold an approximate counter of size $O(\log \log N)$ (as described in [8], and mentioned in the introduction) that counts the number of bits in that bucket. Cohen and Strauss

in [2] proved that the WBMH gives an approximation for DCP of PolyD using $O(\log N \log \log N)$ space.

Finally, we prove the following lemmas on the boundaries, which will later be useful for us.

Lemma 2. *If $g(t) = \frac{1}{t^\alpha}$ for some constant $\alpha > 0$, then for any integer $i \geq 0$ we have that $\frac{b_{i+1}}{b_i} = \Theta(1)$, and $\frac{b_{i+1}}{b_i} > 1$.*

Proof. We know that for any integer $i > 0$, $\frac{g(b_i)}{g(b_{i+1})} \approx 1 + \epsilon$, which implies that $(\frac{b_{i+1}}{b_i})^\alpha \approx 1 + \epsilon$, and thus $\frac{b_{i+1}}{b_i} \approx (1 + \epsilon)^{\frac{1}{\alpha}} = \Theta(1)$. The lemma follows. $\square$

Corollary 3. *If $g(t) = \frac{1}{t^\alpha}$ for some constant $\alpha > 0$, then there exist constants $c, c' > 1$ such that for any integer $i \geq 0$ we have that $c < \frac{b_{i+1}}{b_i} < c'$. Furthermore, the $i + 1$ 'th region, bounded by b_i and $b_{i+1} - 1$ has length at least $c^i(c - 1)$.*

The last corollary implies that the lengths of consecutive regions bounded by boundaries form an exponentially growing sequence. This can be viewed as another proof to the fact that the number of boundaries for polynomial decay functions is $O(\log N)$.

Lemma 4. *If $g(t) = \frac{1}{t^\alpha}$ for some constant $\alpha > 0$, then for any integers $i \geq 0$ and $j, k > 0$, we have that $\frac{g(b_i+j+k)}{g(b_i+k)} \leq \frac{g(b_i)}{g(b_i+j)}$.*

Proof. Let $G(k) = \frac{g(b_i+j+k)}{g(b_i+k)}$. Thus, the derivative of $G(k)$ is $\dot{G}(k) = \alpha(\frac{b_i+k}{b_i+j+k})^{\alpha-1} \frac{j}{(b_i+j+k)^2} > 0$. Therefore, as k grows, so does $G(k)$, and the lemma follows. $\square$

Corollary 5. *If $g(t) = \frac{1}{t^\alpha}$ for some constant $\alpha > 0$, then for any integers $i \geq 0$ and $k > 0$, we have that $\frac{g(b_{i+1}+k)}{g(b_i+k)} \leq \frac{g(b_{i+1})}{g(b_i)} \approx 1 + \epsilon$.*

3 Polynomial Decay

In this section we present a new algorithm for maintaining PolyD aggregation using $O(\log N)$ space. This matches the lower bound of $\Omega(\log N)$ space from [2].

3.1 DCP under PolyD

Cohen and Strauss in [2] used the EH in order to get an approximation for general decay functions using $O(\log^2 N)$ space. In addition, [2] used another type of histograms – the (WBMH) – in order to maintain PolyD in $O(\log N \log \log N)$ space. We combine both of those methodologies, and alter the EH in order to achieve an upper bound of $O(\log N)$ space. This new data structure will be referred to as the Altered Exponential Histogram (AEH).

We start off with Equation 2. What the equation implies is that maintaining exact counts for all sliding windows, allows maintaining exact decayed counts for general decays. However, for functions with the non-decreasing ratio property, instead of maintaining counts of sliding windows for each $t < T$, we can maintain counts of sliding windows only for the boundary times b_i (defined in the previous section), the idea being that we can aggregate values of approximately same decayed weight. This gives us the following approximation for $V_g(T)$:

$$\bar{V}_g(T) = g(N)V_{SLIW\text{IN}_N} + \sum_{i=2}^s (g(N - b_i) - g(N - b_{i-1}))V_{SLIW\text{IN}_{N-b_i}}. \quad (3)$$

Lemma 6. *For any decay function $g(\cdot)$ that has the non-increasing ratio property we have that $\bar{V}_g(T) = (1 \pm \epsilon)V_g(T)$.*

Proof. Let $g(\cdot)$ be a decay function with the *non-increasing ratio* property. All of the streaming values that fall in between time b_i and time b_{i+1} for some i have decayed weight that is within a $1 \pm \epsilon$ of each other (from the way in which we picked the boundaries b_i and b_{i+1}). Thus, the claim immediately follows. $\square$

The problem with the solution we have just presented is that maintaining the sliding windows even approximately with EHs requires $O(\log^2 N)$ space. We will now show how to maintain approximations for the counts of the sliding windows for all of the boundaries, and we will later discuss how approximations of those sliding windows can be used in order to approximate PolyD.

In order to approximate the counts of the sliding windows for all of the $O(\log N)$ boundaries while reducing space, we do the following. We use an EH called E_1 in order to provide us with approximated *timestamps* as will be shown soon. E_1 is an exponential histogram on the special binary stream $f_1(t)$ where $f_1(t) = 1$ for all t , and the parameter¹ with which we decide when to merge buckets in E_1 is some constant k that will be determined later. Clearly, after N time units, the size of E_1 is $O(\log N)$ bits of space (we do not need to maintain the *timestamp* of every bucket as in the original EH, as it is implicit from the structure of E_1).

Next, we use another EH, called E_2 . This EH is used to maintain the data that is received from the binary data stream merging buckets in the same manner as the EH from [4]. However, we do not maintain exact *timestamps* in E_2 , but rather, for each bucket B in E_2 we maintain the index of the bucket in E_1 that contains the value from f_1 corresponding to the most recent value in B . Clearly, each such pointer uses $O(\log \log N)$ bits of space. This would take $O(\log N \log \log N)$ bits of space. We can further decrease the space if for each bucket in E_2 we maintain the difference in indexes between every two consecutive buckets in E_2 , instead of maintaining the indexes themselves. We can then find the value of a pointer of a given bucket in E_2 by summing the differences from the first bucket till the bucket we are interested in. The total space needed for all of the offsets is $O(\log N)$ (because we have that the sum of the offsets in

¹ This is the value k we referred to when explaining how the EH works.

$O(\log N)$, and each offset o_j needs $O(\log o_j)$ bits, so the sum of bits needed to store all of the offsets is $\sum \log o_j \leq \sum o_j = O(\log N)$. Thus the space used is $O(\log N)$ bits.

When a new item arrives in the data stream, we begin by updating E_1 , merging buckets as needed. We then scan E_2 updating the offsets of indexes as required (this is fairly trivial, and thus omitted). Finally, we update E_2 to include the newly arrived data, again merging buckets in E_2 as needed.

3.2 Approximating the DCP

In order to answer a DCP query, we need to approximate the counts of all of the sliding windows for all of the boundaries, and then we can use Equation 3 to approximate the DCP. Say we had the exact *timestamp* for each bucket in E_2 . Thus, approximating the counts of the sliding windows for all of the boundaries would be the same as in the regular EH. However, we do not have exact *timestamps*. What we do have is the index of the bucket in E_1 that contains within it the bit that arrived at the time we are interested in. So, in order to approximate the *timestamp* of a given bucket in E_2 , we sum the sizes of all of the buckets in E_1 , from the most recent one up to the bucket whose index is the index pointed by the bucket in E_2 whose *timestamp* we wish to approximate, but we count only half of the last bucket. Note that this is exactly the method in which we approximate sliding window decayed sums. We can use this fact in order to guarantee the desired approximation, as follows.

In order to approximate the sliding window decayed sum for a boundary b_i , we sum all of the buckets in E_2 whose *timestamp* is approximated to be at most b_i , taking only half of the last bucket. We lose some accuracy being that the *timestamps* are not accurate. Nevertheless, we can achieve our desired approximation by setting the parameter k of E_1 to our needs. The following lemma shows how.

Lemma 7. *There exists a constant parameter k for E_1 such that the approximation τ' of the timestamp of the bucket in E_2 whose real timestamp is τ has the property that $g(\tau') \in (1 \pm \epsilon') \cdot g(\tau)$, for any constant $\epsilon' > 0$.*

Proof. We will choose $k = \frac{1}{1+c}$ where c is the constant c from Corollary 3 (the lower bound). From the properties of the EH we know that every boundary b_i will be within b_{i-1} and b_{i+1} . This completes the proof for *timestamps* that are exact boundaries. Combining this together with Corollary 4, we conclude our proof. $\square$

Thus, we now have the following:

Theorem 8. *The DCP under PolyD can be answered approximately within a constant multiplicative factor of $(1 \pm \epsilon)$ using $O(\log n)$ bits of space, for any constant $\epsilon > 0$.*

Proof. Using the AEH, it follows from the discussion above that we are using only $O(\log n)$ bits of space.

Regarding the approximation, the decay given to a value is first rounded to the decay of the boundary preceding it. From Equation 3 this gives us a $1 \pm \epsilon'$ approximation. However, each boundary b_i is approximated to be between b_{i-1} and b_{i+1} , and thus we suffer another approximation of factor $1 \pm \epsilon'$. Combining the two we get an approximation of factor $(1 \pm \epsilon')^2$, and by setting $\epsilon' = \frac{\epsilon}{3}$ we get the desired approximation. $\square$

4 Extending Binary Streams to Polynomial Streams

In this section we describe how the previous data structures can be extended to provide the decayed sum of streams for polynomial bounded streams. In other words, we assume that for some constant β the values $f(t)$ given by the stream are bounded by $O(N^\beta)$. We assume general streams are still restricted to integral values.

4.1 Sliding Windows and General Decay

We will now show how to extend the EH for polynomial bounded streams so that we will be able to approximately compute V_{SLIWIN_t} for all $t \leq T$. This is done as follows.

Recall that in the EH for a binary stream, we sometimes merge two buckets of size 2^i into one bucket of size 2^{i+1} . Furthermore, the sum of values seen in a bucket of size 2^i was exactly 2^i . However, when considering a polynomial bounded stream, we must differentiate between the size of a bucket, which is the number of items in that bucket, and the sum of values in that bucket. This implies that we require a counter for each bucket of $O(\log N)$ bits for the sum of the values in that bucket. We emphasize that $O(\log N)$ bits are sufficient, as the sum of all of the values in the stream is bounded by $O(N^{\beta+1})$. Being that there are $O(\log N)$ buckets, the extra $O(\log^2 N)$ bits of space used for the counters can be easily afforded in addition to the space used for the EH as was presented in Subsection 2.1. We call the value of the counter of a given bucket the *value* of that bucket.

We are also faced with another problem related to our approximation guarantee. The problem is that very large values can greatly effect the approximation guarantee in the case where they are in the last bucket considered for answering sliding windows sums (which are the basis for approximating general decay sums). Recall that in the EH for binary streams we would split the sum of values in the last bucket by half, while in the EH for polynomial bounded streams, this might skew our approximation in case of a very large value in the bucket that should not be considered for the sliding window. In order for us to solve this problem, we use the same rule merging buckets in the EH for polynomial bounded stream with the following additional constraint. We will only merge two buckets of size 2^i if the sum of the values of the two buckets is less than a fraction of $1 + \epsilon'$ of the sum of all items that have arrived after the two buckets for some constant ϵ' to be determined later. If the two buckets are not merged

due to this new constrained, we will transfer the older bucket to be of size 2^{i+1} . Note that this might cause the value of a bucket of size 2^{i+1} to be less than 2^{i+1} .

Any bucket that changed size without undergoing a merge is called a *non-merged* bucket. A bucket that is not non-merged is said to be a *merged* bucket.

The following lemma shows that the number of buckets is still $O(\log N)$ which implies that the space used is $O(\log^2 N)$ as in the binary stream case.

Lemma 9. *The number of buckets in the EH for polynomial bounded streams is $O(\log N)$.*

Proof. For every two buckets that we decided not to merge we will consider the bucket of larger value. We will count the number of such buckets, and note that the total number of non-merged buckets is at most two times the number of such buckets. Such buckets will be called *interesting* buckets.

Consider any two interesting buckets such that all of the buckets between them are not interesting. Being that we did not merge the more recent of the buckets it must be that the value in that bucket is at least $1 + \epsilon'$ of the value of the other bucket. Thus, if we have $\omega(\log N)$ interesting buckets, the value in the last bucket is in $(1 + \epsilon')^{\omega(\log N)}$, which is more than polynomial in N - a contradiction. $\square$

When we wish to approximate the sliding windows, we do so in the same manner as in the method used in the EHs for binary streams - summing up all of the counters of the appropriate buckets (chosen according to the timestamp), and taking only half of the last bucket if it is not fully contained in the sliding window. This approach can also be extended to any decay function like in the binary stream scenario. Finally, we have:

Theorem 10. *There exists an algorithm that can estimate any decayed sum within a constant multiplicative factor of $1 \pm \epsilon$, for any constant $\epsilon > 0$, using $O(\log^2 N)$ bits of space.*

Proof. Say we wish to compute the SW of length W at time T . If the streaming value $f(T - W)$ is in a bucket that was never merged (indicated by the appropriate bit), we will return an exact solution, as it is simply the sum of all buckets up to the bucket containing $f(T - W)$ (the last bucket contains only $f(T - W)$).

If the streaming value $f(T - W)$ is in a bucket that has been merged, then we have that the value of the counter of that bucket is less than a fraction of $1 + \epsilon'$ of the sum of all items that have arrived after the timestamp of the bucket. Being that in such a case we take only half of the value of the counter of that bucket, we get an approximation of $1 + (1 + \epsilon')/2$, and if we set $\epsilon' = 2\epsilon - 1$ we get our desired approximation.

Regarding the space used, it follows immediately from the above discussion. $\square$

4.2 DSP under PolyD

In such a case, we can use the same method for DCP in order to maintain the decayed aggregation in the following manner.

Assume that at time t we receive the value $m = f(t)$. We can treat the value m as m bits of value one, all arriving at the same time from the stream. Therefore, it is as if we inserted m bits to the AEH at the same time. We can use the binary AEH method from the previous section that will give the same approximation, and the space required is bounded by $O(\log N^{\beta+1}) = O(\log N)$.

However, when using this method it takes $O(m)$ time to insert the value m to our AEH. For many applications this is far too much time. For example, a router on the internet must be able to process data quickly before the next piece of data arrives, so lowering the amount of time it takes to insert a value m is crucial. One possible solution would be to insert the value of m in $O(\log m)$ time by computing the presentation of m in the appropriate base as defined by the parameter of the AEH, and immediately create the appropriate buckets (this will give us $O(\log m)$ buckets, which we can then insert one by one taking $O(\log m)$ time). There are more details, however we would like to do better than $O(\log m)$ as well, as we do not want the time to be dependant on the possible values arriving in the data stream, and rather, spend a constant amount of time no matter what is the value arrived. The rest of this subsection shows how we can solve this problem in order to achieve $O(1)$ time for insertion of a value.

For intuition sake, we will begin by explaining the idea behind our structure using one AEH, and one WBMH. The idea is first to insert the newly inserted values into the WBMH, and once every $O(\log N)$ insertions of newly arrived items into the WBMH, we transfer all of the data from the WBMH into the AEH. Thus, the WBMH will have $O(\log \log N)$ buckets as it covers a time line of length $O(\log N)$ (as the number of buckets in the WBMH depends only on elapsed time). Furthermore, we maintain approximate counters for each bucket, each of size $O(\log \log N)$ bits, which are sufficient as the total sum of a stream of length $\log N$ is bounded by $N^\beta \log N$. We note that because the WBMH uses approximate counters, we need to take that extra approximation into consideration when choosing the constants for operation. To summarize, we require a total of $O((\log \log N)^2)$ bits of space for the WBMH, which we can easily afford within our $O(\log N)$ upper bound.

Nevertheless, the solution just described still requires $O(\log N)$ worst-case time per insertion, as each time we move the information stored in the WBMH into the AEH, we pay $O(\log N)$ time. In order to solve this, in addition to our AEH, we add two WBMHs, referred to as W1 and W2, each for a time frame of size $\log N$. The first $\log N$ items that arrive are inserted into W1. Each of these inserts takes $O(1)$ time because all of the bits fall in the same bucket (recall that we treat the value m as m bits arriving at the same time). This is because the beginning and end of each bucket does not depend on the values arrived, but rather on the time the value arrived, so each item that arrives at some time goes into only one bucket. This means that we only need to update one counter every time a new item arrives. The merges can be done in a way where the number of merges is constant per insertion using standard de-amortization techniques - details are omitted. After $\log N$ items arrive, we insert the next $\log N$ items into W2, and concurrently, we insert the values from W1 to the AEH (we will soon

show how to do this efficiently). We continue switching between W1 and W2 as time progresses.

Now we need to show how inserting the values from a WBMH to an AEH can be done efficiently. For sake of convenience, we will assume we want to insert the values from W1. The number of bits in W1 is bounded by $N^{\beta+1}$, hence the amount of buckets to be added to the AEH is $O(\log N)$. We also need to merge the appropriate buckets that were already in the AEH and update their times. Using an $O(\log N)$ sized counter we can exactly count the sum of the counters in W1. From this number we can calculate how many buckets will be added to the AEH, and update it accordingly. In $O(\log N)$ time we can update the buckets already inside the AEH, and in $O(\log N)$ time we can insert the new buckets into the AEH. This gives a total of $O(\log N)$ time to insert the values in W1 into the AEH, which is amortized $O(1)$. In order to obtain a worst case time of $O(1)$ we can take a lazy approach, and spread our operations on the $\log N$ insertions made into W2.

The last thing we need to take care of is answering a query while inserting the values from W1 to AEH. Specifically, in the lazy approach we need to wait till we are done with the whole process of merging W1 with the AEH before answering a query, as it is not clear how it can be done otherwise. To avoid this problem we keep two copies of the AEH and two copies of each WBMH. When joining W1 with the AEH, we will join a copy of each, while answering queries from the duplicates. Of course, we also need to query W2 due to items already inserted into it - this is fairly trivial. This completes our data structure and provides the following:

Theorem 11. *There exists an algorithm that can estimate DCP within a constant multiplicative factor using $O(\log N)$ bits of space, and constant update time.*

Proof. The space follows from the above discussion. Regarding the approximation, we note that the approximation guarantees follows immediately from the properties of the AEH, the WBMH, and the approximated counters used in the WBMH (which can be easily be taken into account when computing the approximation of the WBMH). $\square$

5 Adding Additive Error - Another Model

Until now all of the approximation models used in maintaining values for the DCP and DSP problems looked into $1 \pm \epsilon$ approximations. We present a new view where we allow an additive error as well. In other words, we want to be able to approximately answer the DCP and DSP problems allowing values of $(1 \pm \epsilon_1)RV \pm \epsilon_2$, where RV stands for the real value of the answers to the problems.

The motivation for such an approximation comes from realizing that when dealing with massive data sets, we are often not interested in answers with small values. This is because small decayed sums generally occur when we have been

receiving zero or very small valued items for an extended period of time - a scenario that is uncommon in many applications. Specifically for binary streams there are many such applications. Interestingly, allowing this extra relaxation, we can answer the DCP and DSP problems using much less space under some decay functions, while in other decay functions the extra relaxation doesn't help at all.

We now explore the PolyD and ExpD decay functions, when allowing an additive error.

5.1 Exponential Decay for DSP

In order to maintain ExpD for the DSP problem we note that in ExpD, every time a new value arrives in the stream, the new decayed aggregate has the value of the previous aggregate divided by the base, plus the new arrived value. Using this observation, we can maintain the $O(\log \log N + \log \frac{1}{\epsilon_1})$ most significant digits of our decayed aggregate. We also note that any polynomial bounded item that arrives, decays to a value of $o(\epsilon_2)$ within $O(\log N)$ time units (because we divide that value by the base each time). So what we really need to maintain is only the estimate of the decayed sum for the last $O(\log N)$ items. Now, each of those $O(\log N)$ items, when represented in $O(\log \log N + \log \frac{1}{\epsilon_1})$ most significant bits contributes a multiplicative error of $2^{-O(\log \log N + \log \frac{1}{\epsilon_1})} = \Omega(\text{poly}(\frac{\epsilon_1}{\log N}))$. We can choose the poly in the notation to simply be $O(\frac{\epsilon_1}{\log N})$, and thus, the sum all of the errors of the $\log N$ items is at most $O(\log N \cdot \frac{\epsilon_1}{\log N}) = O(\epsilon_1)$ of the total sum of the items, as required.

Note that we do not maintain separate representations for each of the last $O(\log N)$ items that have arrived. Rather, we maintain one aggregate, and the analysis can be decomposed to the components of the aggregate that are the decayed values of the items. Furthermore, we also need to maintain the location of the most significant bit in order to know the order of magnitude of our succinct representation (we need to know how many digits are to the right of our $O(\log \log N + \log \frac{1}{\epsilon_1})$ bits). Thus, this requires a total of $O(\log \log N)$ bits.

Finally, we note that using randomization, we can further reduce the constants of the $O(\log \log N)$. However, the solution would be probabilistic.

5.2 Exponential Decay for DCP

For a binary stream it is enough to maintain the last $O(\log 1/\epsilon_2)$ bits seen by the stream, and using the AEH from we can do this in $O(\log \log 1/\epsilon_2)$ space. This of course is a great improvement in space, as it is independent of N .

5.3 Polynomial Decay

When considering the polynomial decay, we differentiate between the case in which $\alpha > 1$ and the stream is binary, to the case in which $1 \geq \alpha > 0$ or the stream can receive general polynomial sized values. When $\alpha > 1$ we show how

to further reduce the amount of space used, while in the $1 \geq \alpha > 0$ case we have a lower bound showing that we cannot do better in space used than in the previous model where we allowed only a multiplicative error. The intuitive reason for separating the two cases comes from the well known fact that $\sum_{j \leq T} g(t)$ with $1 \geq \alpha > 0$ diverges, while the sum with $\alpha > 1$ converges. As we will show, the lower bound follows this intuition.

Binary Polynomial Decay with $\alpha > 1$ In this case we only need to maintain the last $1/\epsilon_2$ seen by the stream, and again we can use the AEH to do this in $O(\log 1/\epsilon_2)$ bits of space. We note that it is easily possible to extend this to any stream with values bounded by some constant.

A Lower Bound for Polynomial Decay

Theorem 12. *A logarithmic number of bits is necessary in order to approximately maintain decay by $g(x) = x^{-\alpha}$, within a multiplicative factor and an additive error (in elapsed time), unless $\alpha > 1$ and the stream's values are bounded by a constant.*

Proof. The proof is by reduction to the communication complexity of the subset problem. In the subset problem, we have two players A and B with unlimited computational power, where each player holds a subset of some larger set S . Player A performs some computations on its set and sends a binary string s to player B . Once player B receives string s , it performs some computation and outputs whether the subsets that both players have are equal or not. If the set S is of size $\Omega(\log n)$, then by [9] s must be of size $\Omega(\log n)$. We will show that if there exists an algorithm ϕ that uses less than $\Omega(\log N)$ bits in order to approximately maintain decay by the given function, then we can use that algorithm in order to solve the subset problem using $o(\log n)$ bits of space, contradicting the lower bound from [9].

We begin by defining a stream $f_A(t)$. The definition of the stream is taken from the proof of the lower bound for polynomial decay with multiplicative approximation in [2]. Let k be some constant to be determined later. Let $T = -k^{\frac{2}{\alpha}}$, and let $r = \lfloor \frac{\alpha}{2 \log k} \log(N/2) \rfloor$ be the size of S . For simplicity of notations, we assume that the time line is between $-\frac{N}{2}$ and $\frac{N}{2}$. At time $t = -T^i$ for $1 \leq i \leq r$, we have that $f_A(t) = 2k^i$ if the i 'th element of S is part of the subset maintained by player A , and otherwise, $f_A(t) = k^i$. At any other time t we have that $f_A(t) = 0$. Player A runs algorithm ϕ on the stream from time $-N/2$ till time 0, and then player A sends the space stored by ϕ to player B . Let $n_i = \frac{f_A(-T^i)}{k^i} \in \{1, 2\}$ be an indicator of whether the i 'th element in S is in the subset of player A .

In [2] it was shown that if $k = 10$ we have that

$$\frac{g(2k^{\frac{2i}{\alpha}})k^i n_i}{\sum_{j \neq i} g(k^{\frac{2i}{\alpha}} + k^{\frac{2j}{\alpha}})k^j n_j} \geq \frac{1}{4}.$$

Thus, if we use a multiplicative approximation of $\epsilon < \frac{1}{4}$, then player B can use the gap in ratio between the decayed value of the i 'th element at time T^i and the sum of the other decayed values at that time in order to reproduce whether A has the i 'th element in its subset or not. Thus, by running ϕ , player B can correctly compute whether player A has the same subset that player B has. This was the proof in [2] that showed how a multiplicative approximation requires $\Omega(\log N)$ bits.

Now, when considering an additive approximation as well, we do the following. Consider the stream $f_{A,X}(t) = X \cdot f_A(t)$, for some X to be determined. Note that for this stream we have that

$$\frac{Xg(2k^{\frac{2i}{\alpha}})k^i n_i}{\sum_{j \neq i} Xg(k^{\frac{2i}{\alpha}} + k^{\frac{2j}{\alpha}})k^j n_j} = \frac{g(2k^{\frac{2i}{\alpha}})k^i n_i}{\sum_{j \neq i} g(k^{\frac{2i}{\alpha}} + k^{\frac{2j}{\alpha}})k^j n_j} \geq \frac{1}{4}.$$

Thus, the multiplicative lower bound still holds, under the assumption that the stream can have polynomially bounded data. Now, in order to get our lower bound for the additional additive approximation, we will show that

$$Xg(2k^{\frac{2i}{\alpha}})k^i n_i - \sum_{j \neq i} Xg(k^{\frac{2i}{\alpha}} + k^{\frac{2j}{\alpha}})k^j n_j \geq \frac{3}{4}.$$

This will imply that an additional additive approximation which is less than $\frac{3}{8}$ will still require at least $\Omega(\log N)$ bits.

In [2] it was shown that $g(2k^{\frac{2i}{\alpha}})k^i n_i = 2^{-\alpha} k^{-i} n_i$, and that $\sum_{j \neq i} Xg(k^{\frac{2i}{\alpha}} + k^{\frac{2j}{\alpha}})k^j n_j \leq \frac{1}{4} 2^{-\alpha} k^{-i}$. Thus:

$$Xg(2k^{\frac{2i}{\alpha}})k^i n_i - \sum_{j \neq i} Xg(k^{\frac{2i}{\alpha}} + k^{\frac{2j}{\alpha}})k^j n_j \geq X2^{-\alpha} k^{-i} (n_i - \frac{1}{4}) \geq X2^{-\alpha} k^{-i} \frac{3}{4}.$$

If we let $X = 2^\alpha k^{r^2}$, then we have that for any i , $X2^{-\alpha} k^{-i} \frac{3}{4} \geq 2^\alpha k^i 2^{-\alpha} k^{-i} \frac{3}{4} = \frac{3}{4}$, as we required.

Now for the binary case with small α , we want to spread the value that arrives at time $t = -T^i$ to be a consecutive number of bits that arrive all from time t onwards like in the proof of [2]. If such a spreading is possible then the details are exactly as in [2], and so details are omitted. What we need to show is that such a spread is possible for small α . What we need to do in the worst case is spread $X2k^i = 2^{1+\alpha} k^i$ into the gap $k^{\frac{2i}{\alpha}} - k^{\frac{2i-2}{\alpha}}$ time slots. As α decreases, so does the value $2^{1+\alpha} k^i$, while the gap itself increases, giving more room to spread the bits, with no collisions. However, for large α , we do not have enough room within our gap for all of the values, and that is where our proof falls for large α . $\square$

² Note that X is polynomial in N .

6 Conclusions

We discussed polynomial and exponential decay in data streams, extending recent work by [4] and [2]. We present the tightest space-efficient algorithm for PolyD using $O(\log N)$ space when allowing approximations up to a multiplicative factor. We do this by presenting a new data structure, the AEH, that is an extension of the EH from [4] and the WBMH from [2].

In addition, we presented and analyzed our decay functions when allowing an approximation with an additive error in addition to the multiplicative one. We find that in ExpD for general streams $O(\log \log N)$ space suffices, while in binary streams a constant (depending on the approximation) amount of space is all we need. For the PolyD we prove that unless the exponent is larger than one, and the stream values are bounded by some constant, we cannot hope to do better than in the multiplicative only approximation. We also show that one can use the AEH in order to achieve a constant-space algorithm for PolyD when the exponent is larger than one, and the stream values are bounded by some constant.

The need to analyze other decay functions such as the chordal decay, polyexponential decay, etc. ([2]) still exists. We strongly feel that the AEH can be used in some of the other decay functions, and in many other applications as well.

References

1. A. Bremler-Barr, E. Cohen, H. Kaplan, and Y. Mansour. Predicting and bypassing internet end-to-end service degradations. In *Proc. 2nd ACM-SIGCOMM Internet Measurement Workshop.*, ACM, 2002.
2. E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *J. Algorithms*, 1(59), 2006
3. E. Cortess and D. Pregibon. Giga-mining. In *Proc. Of KDD*, New York, August 1998.
4. M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. 13th ACM-SIAM Symp. On Discrete Algorithms.*, ACM-SIAM, 2002.
5. S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. In *IEEE/ACM Transactions on Networking*, 1(4), 1993.
6. P. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *Proc. Of the 14th Annual ACM Symp. On Parallel Algorithms and Architectures*, Pages 63-72. ACM, 2002.
7. Warren Gilchrist. Statistical Forecasting. Wiley, 1976.
8. R. Morris. Counting large numbers of events in small registers. In *CACM*, 81:840-842, 1978.
9. A. C. Yao. Some complexity questions related to distributed computing. In *Proc. of the 11th ACM STOC*, 209:213, 1979.