Maintaining Time-Decaying Stream Aggregates*

Edith Cohen † Martin J. Strauss ‡

January 25, 2005

Abstract

We formalize the problem of maintaining *time-decaying* aggregates and statistics of a data stream: the relative contribution of each data item to the aggregate is scaled down by a factor that depends on, and is non-increasing with, elapsed time. Time-decaying aggregates are used in applications where the significance of data items decreases over time. We develop storage-efficient algorithms, and establish upper and lower bounds. Surprisingly, even though maintaining decaying aggregates have become a widely-used tool, our work seems to be the first both to explore it formally and to provide storage-efficient algorithms for important families of decay functions, including polynomial decay.

1 Introduction

The advent of high speed communication networks and massive transient datasets has made the streaming data model and stream management systems the focus of much recent work (see [1] for an excellent survey). In the data stream model, data arrives sequentially and is processed by an algorithm whose workspace is insufficient to store all the data, so the algorithm must process and then discard each data item. The algorithm maintains a *summary* whose storage requirement is considerably smaller than that of the stream. Using this compact summary, the algorithm is able to answer queries about the data stream.

In many applications, however, older data items are less significant than more recent ones, and thus should be discounted accordingly. This is because the characteristics or "state" of the data generator may change over time, and, for purposes such as prediction of future behavior or resource allocation, the most recent behavior should be given larger weight. The rate of decay is captured by a *decay function*, which is a non-increasing function—the weight of each item is the decay function applied to the elapsed time since the item was observed.

One of the most fundamental queries on a stream of values is a sum, or, equivalently, an average. Consider a stream of nonnegative integers. It is not hard to see that exactly maintaining the sum (or average) requires storage of $\Theta(\log(n))$ bits (where n is the value of the sum). Work of Morris [16] and subsequent folklore work shows how to maintain an approximate value of the sum using only $O(\log\log(n))$ storage bits. A *time-decaying sum* or (similarly, *average*) of the stream is a weighted sum according to a decay function.

*A preliminary version of this paper appeared in [7]

†AT&T Research Labs, 180 Park Ave. Florham Park, NJ, USA. Email: edith@research.att.com.

‡Depts. of Mathematics and EECS, Univ. of Michigan, Ann Arbor, MI, USA. Email: martinjs@umich.edu. Part of this work was done while the second author was at AT&T.

1.1 Applications

There has been an abundance of applications work with time-decaying sum or average aggregates. We list some examples we are aware of. The listed applications use time-decaying averages to estimate network delay, congestion, or connection activity, but many other applications are conceivable.

- **The Random Early Detection (RED) protocol.** RED is a popular protocol deployed in Internet routers for congestion avoidance and control. RED uses the weighted average of previous queue lengths to estimate the impending congestion at the router; the estimate is then used to determine what fraction of packets to discard [14, 11].

- **Holding-time policies for ATM virtual circuits.** Circuit-switched data connections have an associated cost of being kept open; data transfers are bursty and intermittent and, when a data burst arrives, it incurs smaller delay when the circuit is open. Thus, when there are multiple connections, it is important to assess the anticipated idle times (time to the next data burst) of each circuit and to close first those circuits that have longer anticipated idle times. This can be done using a time-decaying weighted average of previous idle times [15]. A similar model applies to maintaining open TCP connections at a busy Web server [6].

- **Internet gateway selection products.** Multiple Internet paths are available to each destination host and the product needs to assess the reliability of each path in order to facilitate a better selection of a path. A time-decaying average of previous measurements can be used as a measure of path quality [2].

- **Maintaining statistics about usage patterns of AT&T telecom customers.** In this application, a summary is maintained per field on each of around 100 million customers; thus, optimal balancing of information value and available storage is very important [8].

1.2 Decay functions

Much of the applications of time-decaying aggregates predate *formal* analysis of stream algorithms, and the most commonly used decay function is *Exponential decay*, the only natural alternative for which simple and efficient algorithms were known. The exponentially-decaying weighted average can be maintained easily in a single counter, C, by the formula $C \leftarrow (1-w)\,x + wC$, where $0 < w < 1$ determines the rate of decay and x is the new data value. Under this formula, the contribution to the weighted average of a data item value observed T time units ago is w^T times its original value. The required storage for approximately-maintaining this value is $\Theta(\log N)$, where N is elapsed time.[1]

Another natural model for discounting older data is *Sliding Windows*. For a parameter N, we want to maintain the count of 1's (or sum of values) over the last N time units. It is not hard to see that maintaining an exact count requires $\Omega(N)$ storage bits, as the algorithm must track the most recent N data items exactly. Sliding Window decay has recently been addressed in [9], where the authors consider the problem of maintaining an *approximate* count. They give an algorithm (the "exponential histogram," EH) and a matching lower bound for maintaining this count using $\Theta(\log^2(N))$ storage bits. Further work on maintaining sliding window decay for distributed streams was performed by Gibbons and Tirthapura [12].

Note that there is an exponential gap between the storage required for approximately maintaining a non-decaying sum ($\Theta(\log\log N)$) and the storage required for Exponential or Sliding-Window decaying

[1]This bound holds for 0/1 streams, but extends in a simple way to streams where items have values that are polynomial (in elapsed time). This assumption also holds for bounds that are stated in the sequel.

sums. In turn, Sliding Window decay ($\Theta(\log^2 N)$ storage bits) is quadratically worse than Exponential decay ($\Theta(\log N)$ bits).[2] These differences in required storage can be very significant for applications.

We have, so far, discussed two particular families of decay functions: sliding windows and Exponential decay. The storage requirements of both families are well understood and time-decaying aggregates according to these functions can be approximately maintained with polylogarithmic storage. We next identify some desired properties of decay functions. We then argue, using examples, that these two families are "insufficient:" First, they do not possess some intuitive properties, and, second, they do not provide a rich-enough class of decay rates, as the desired rate of decay for a particular application depends on the time scales of correlations between values. Both sliding windows and exponential decay discount older data very severely. Intuitively, we expect the significance of an event indeed to decay with elapsed time, but also to be larger for a more severe event. For different applications, we would like to be able to tune the balance of severity with time-decay.

As an illustrative example, we consider a time-decaying aggregate of past performance as a measure of availability of network links. We consider two links (see Figure 1), L1 and L2. Suppose that link L1 fails for a 5-hour duration and, 24 hours later, alternative link L2 experiences a 30-minute long failure. The two links were otherwise reliable and no more failures are observed later on. We would like to compute online simple numerical ratings that capture the relative reliability of the two links. The link L1 experienced a more severe failure event than L2, but less recently than L2. One would imagine that the initial rating of the two links should depend on the application and general properties of the failure patterns and the desired balance between the severity of an event and the rate of time-decay. Thus, one may initially want to deem one of the links as more reliable or deem both of them to be similarly reliable. We would like to have a rich enough range of decay rates to support these possibilities. Regardless of the initial rating, as time progresses, and the time difference between the failure events become small relative to elapsed time, we expect that L2, which experienced a less-severe failure, to emerge eventually as more reliable than L1.

Consider first using a sliding window decay function for our two links. A small window size would completely discount the failure event experienced by L1, thus viewing it as a more reliable link, and later on as an equally-reliable link. A larger window size will provide a view that changes from one that deems L2 to be much more reliable than L1 to one that deems L1 to be much more reliable. This contradicts our expectation that the relative reliability rating of L2 increases as time progresses.

We next consider using an Exponential decay function. Exponential decay has the property that the relative contribution of both failure events remains fixed through time, and, thus, its "view" on the relative reliability of links remains fixed as time progresses. So depending on the decay parameter, either L1 or L2 will be consistently viewed as the more reliable link. Again, this view contradicts our expectation.

In particular, no member in these two families can provide a view where first L1 is rated as more reliable and then L2 emerges as more reliable. We can now ask which decay functions fulfill these intuitive expectations? In particular, we need decay functions that allow the weights of two items to become closer as time progresses. We noted above that this property does not hold for Exponential decay or sliding windows. Smooth subexponential decays do have this property, and one particularly appealing family of decay functions is polynomial decay, where the weight given to a data item is inverse-polynomial in elapsed time. These functions can also tremendously expand our ability to tune the rate of decay. As far as we know, however, until now, there was no known method to maintain aggregates with these decay functions with polylogarithmic storage.

[2] In the sequel we define N to be the minimum of elapsed time and the minimum value after which the decay function nullifies. Exponential decay and window size are compared under this unifying metric.

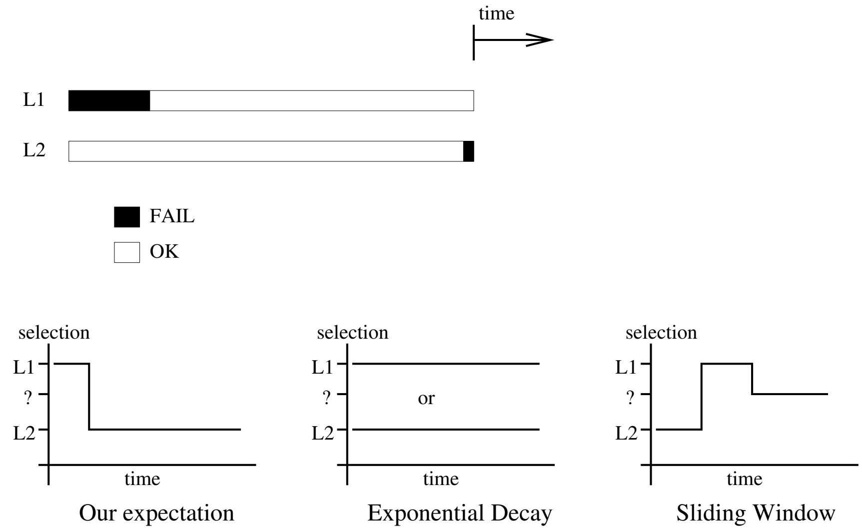

Figure 1: Relative reliability rating for the links L1 and L2. The figure shows the more reliable link as a function of time when using different decay functions.

1.3 Our contributions

We now summarize the main contributions of our work.

- We formulate the problem of maintaining time-decaying aggregates for general decay functions and identify desired properties of the family of available decay functions. We argue that these properties are not satisfied by decay functions with known storage-efficient algorithms (namely, Exponential and sliding-window decay) and identify new classes of decay functions that fulfill these properties.

- We show that time-decaying sums (and averages) under *any* decay function can be approximately tracked using $O(\log^2(N))$ storage bits; this is the first polylogarithmic-storage approximation algorithm known for general decay functions. Our algorithm builds on the Exponential Histogram technique [9]. Interestingly, this result shows that, in a sense, sliding windows, for which a matching lower bound of $\Omega(\log^2 N)$ is known [9], are the "hardest" decay functions for maintaining time-decaying sums.

- We give a different algorithm for maintaining time-decaying sums for decay functions with certain properties. The algorithm maintains polynomial decay (approximately) with storage of $O(\log(N)\log\log(N))$ bits. This result shows that polynomially decaying sums can be tracked nearly as efficiently as Exponentially decaying sums.

- We give a nearly-matching lower bound of $\Omega(\log(N))$ storage bits for maintaining polynomially decaying sums.

- We define time-decaying aggregation and explore algorithms for other interesting aggregates including random selection, L_p norms, and variance. Algorithms for some of these aggregates were developed in the more restricted sliding-window time-decay model [9, 1] and in the more general spatial-decay model [5, 4] (that was introduced in subsequent work). We discuss adaptations of these algorithms for the time-decay model and general decay functions.

The paper is structured as follows. We start in Section 2 with preliminaries and then, in Section 3, discuss some specific decay functions. In Section 4 we show how to use cascaded computation with an Exponential

Histogram to approximate general decay functions. In Section 5 we develop a different algorithm, weight-based merging histogram (WBMH) and show that it performs better than cascaded Exponential Histograms on some families of decay functions that include polynomial decay. In Section 6 we give the lower bound on the storage requirement of any algorithm that approximates polynomially decaying aggregates. Beyond sums and averages, in Section 7 we discuss other basic time-decaying aggregates including L_p norms, variance, and random selection.

2 Preliminaries

Consider a stream of data items $\{(t_i, f_i)\}$, where t_i is the arrival time and f_i is the value of the ith item.

A *decay function* is a non-increasing $g(x) \geq 0$ defined for $x \geq 0$. We use T to denote current time. At time T, the *weight* of item i is $g(T - t_i)$ and the *decaying value* of the item is $f_i g(T - t_i)$. For simplicity of presentation, we assume that time is discretized and obtains integral values. We will use the notation $f(t) = \sum_{i|t_i=t} f_i$ as the sum of values of items obtained at time t. Below we define the two main decaying aggregates we consider: the time-decaying sum and the time-decaying average of the stream $\{(t_i, f_i)\}$ under the decay function $g()$.

2.1 Time-Decaying Sum

Problem 2.1 Decaying Sum Problem *(DSP). The input to the problem is a stream of data items and the goal is to estimate, at any current time T, the* time-decaying sum

$$S_g(T) = \sum_{i|t_i<T} f_i g(T - t_i) \ .$$

We assume that the values f_i are in $\{0, 1\}$, as the results can be easily generalized to polynomial values (as in [9]). We refer to the DSP problem with binary f_i values as the *Time-Decaying Count Problem* (DCP).

As with traditional streaming applications, our goal is to obtain an estimate of $S_g(T)$ at any time T while using a small amount of memory. We are interested in $(1 \pm \epsilon)$-approximate estimates, where $\epsilon > 0$ is fixed. In other words, we are interested in obtaining $S'_g(T)$ such that $\frac{S'_g(T) - S_g(T)}{S_g(T)} \leq \epsilon$.

2.2 Time-Decaying Average

The decaying average of a stream is a time-decaying weighted average of observed values.

Problem 2.2 Decaying Average Problem *(DAP). The input is a stream of data items, and the goal is to estimate at any time T, the* time-decaying average *of the stream, defined as*

$$A_g(T) = \frac{\sum_{i|t_i<T} f_i g(T - t_i)}{\sum_{i|t_i<T} g(T - t_i)} \ .$$

Each item in the decaying average is weighted according to the decay function. The decaying average is especially useful if $\sum_t g(t) = \infty$, in which case the decaying count may also go to infinity.

Observe that the numerator of the decaying average is a decaying sum of the stream $\{(t_i, f_i)\}$ and the denominator of the decaying average is a normalization factor by the sum of the weights and essentially is a decaying count of the stream $\{(t_i, 1)\}$. It follows that an approximate weighted average can be obtained from the corresponding approximate decaying sum and decaying count. Therefore in the sequel, we focus on algorithms for the more basic decaying count problem.

2.3 Histograms and Metrics

The main performance metric we use is the amount of storage used by the algorithm as a function of N, where N is the defined to be the minimum of elapsed time since the arrival of the first data value and $N(g()) = \arg\max_x g(x) > 0$ (the maximum value for which the decay function is positive).

The algorithms we discuss aggregate the items into a *histogram*. Each bucket of the histogram aggregates all items that were observed in some time interval. Thus, each bucket of the histogram has a start-time and an end-time, and we refer to the difference between the start and end times as the *time-width* of the bucket and to the sum of values of all items in the bucket as the *count-width* of the bucket. Buckets are merged and discarded in a particular way which depends on the algorithm. When two consecutive buckets are merged, the new bucket inherits the start-time of the earlier bucket, the end-time of the later bucket, and count-width which is the sum of the count-widths of the two buckets.

Different algorithms may use time- or count-based criteria to place the boundaries of the buckets. The Exponential Histogram algorithm is such that the count boundaries of the buckets are independent of the stream, but the time boundaries vary. We will present another algorithm, weight-based merging histograms (WBMH), where the time boundaries are independent of the particular stream but the count boundaries vary.

Boundaries that are independent of the stream are useful for reducing the amount of storage needed for the histogram, since the boundary values do not need to be stored for each stream. This distinction between time and count, and which boundaries are stream-independent, turns out to be important. The total storage we require is determined by the number of buckets we use and the storage that is needed for each bucket. The WBMH algorithms we explore can maintain an approximate count instead of the exact count-width of each bucket. In contrast, approximate tracking of time-boundaries seems to be problematic, as a small deviation in a boundary can lead to a large deviation in the value of the decaying sum.

3 Decay functions

We describe some families of decay functions that are of particular interest.

3.1 Exponential decay EXPD_λ

For a parameter $\lambda > 0$, $g(x) = \exp(-\lambda x)$. With EXPD, the relative significance of each measurement decreases exponentially with elapsed time. EXPD is commonly used in practice. One of the appeals of EXPD is that it can be maintained easily using

$$S_{\text{EXPD}}(t) = f(t) + \exp(-\lambda) S_{\text{EXPD}}(t-1) \ . \tag{1}$$

We explore the storage requirements of EXPD.

Lemma 3.1

- *Exact tracking of* S_{EXPD} *requires* $\Omega(N)$ *storage bits.*
- *Approximate tracking of* S_{EXPD} *requires* $\Theta(\log N)$ *storage bits.*

Proof: We start with exact tracking. We show that a binary stream of length N (where an item with value either 0 or 1 can be observed every time unit) can have exponentially many distinct values of the decay function. Let $k = \lceil \lambda^{-1} \rceil$ be a constant. Consider now only binary streams with either a 0 or a 1 at times t with $t \bmod k = 0$ (and 0 value or no items if $t \bmod k \neq 0$). Consider now the value of the decaying count N time units after the start of the stream. There are at least $2^{(\lceil N/k \rceil)}$ such streams and each stream corresponds to a unique value of the decaying count. Thus, our algorithm must be able to differentiate all these streams, and thus use at least $\lceil N/k \rceil$ storage bits.

For the lower bound on approximate tracking, consider all streams that consist of a single item of value "1" during elapsed time of N time units. In order to obtain a factor 2 approximation, our algorithm must be able to differentiate the timestamp of the "1" item to within k time units. Thus, it requires at least $\log \lfloor N/2k \rfloor$ bits.

For the upper bound note that it is sufficient to track the time stamps of the C most recent 0/1-valued items,[3] where $C = \lceil -\lambda^{-1} \ln((1 - \exp(-\lambda))\epsilon) \rceil$ is a constant that depends on λ and ϵ. Alternatively, we can use the classic algorithm (1) over floating point arithmetic with a logarithmic number of bits. ∎

3.2 Sliding Window SLIWIN_W.

Sliding windows are parameterized by the window size W. The corresponding decay function has $g(x) = 1$ for $x \leq W$ and $g(x) = 0$ otherwise. Sliding windows were introduced in [9], which also presented an algorithm that obtains a $(1 + \epsilon)$ approximation for any constant ϵ using $O(\epsilon^{-1} \log^2 W)$ bits. With SLIWIN decay, all data values in a recent window of size W have equal importance and all older data values have 0 weight.

3.3 Polynomial decay POLYD_α

For a parameter α, put $g(x) = 1/x^\alpha$. In the introduction we argued that POLYD is often a better choice for a decay function than EXPD. As with EXPD, exactly maintaining a decaying count with POLYD requires linear storage, at least for certain polynomials. In the sequel we present efficient algorithms for approximately maintaining POLYD counts.

Lemma 3.2 *For decay function $g(x) = 1/x$, exact tracking of S_{POLYD} requires $\Omega(N)$ storage bits.*

Proof: We consider just decay by the power $g(t) = 1/t$; this argument extends also to many other polynomial decay functions.

The vector of values $S_{\text{POLYD}}(T)$, for $N < T \leq 2N$, is obtained from the vector of values $f(t)$, for $0 < t \leq N$, by multiplication by the matrix whose $(t, T - N)$ entry is $1/(T - t)$. Reversing the order of the rows, we get the Hilbert matrix, whose (j, k) entry is $1/(j + k)$. Since this matrix is known to be non-singular, it follows that all 2^N possible $(0/1)$-valued vectors $f(t)$ can be recovered from the exact vector $S_{\text{POLYD}}(T)$ of the decayed sum.

For completeness, to see that the Hilbert matrix is non-singular, observe that the decay $S_{\text{POLYD}}(T) = \sum_{0<t\leq N} \frac{f(t)}{T-t}$ is a rational function of T, and $S_{\text{POLYD}}(T) \prod_{0<t\leq N}(T - t)$ is (except at removable discontinuities), a polynomial of degree $N - 1$. So it is completely determined by the N values $S_{\text{POLYD}}(T)$ for $N < T \leq 2N$. ∎

3.4 Polyexponential Decay

Polyexp decay with parameters k and $\lambda > 0$ is decay by the function $g(x) = x^k e^{-\lambda x}/k!$. Building on this, let $p_k(x)$ be a polynomial of degree k. It was shown, in [7], how to track decay by $p_k(x)e^{-\lambda x}$ by a reduction to $k + 1$ (pipelined) instances of exponential decay. This gives a generalization of exponential

[3]For general (non binary) streams, we record the C "most recent" values, but instead of using actual time stamps we treat an item of values v received at time t as an item of value 1 received at time $t + \lambda^{-1} \ln v$ (note that the contribution to the decaying sum is the same).

decay to a much broader class of functions while preserving much of the simplicity of the natural algorithm for exponential decay.

For $k = 2$ ($k = 3$), this technique is known as Brown's double (triple) exponential smoothing [13], from around 1960, and is still used in a different context to model data by a line or quadratic.

We will not discuss polyexponential decay further in this article.

4 Domination-based aggregation

We quickly review Exponential Histograms and then show how cascaded computation can make them applicable to general decay functions.

4.1 Exponential Histograms

Exponential Histograms (EHs) were introduced by Datar et al [9] as a method to estimate SLIWIN decaying counts; Datar et al proved tight bounds showing that $\log^2(N)$ bits are sufficient and necessary for $(1+\epsilon)$-approximate estimates. Note that, for sliding windows, N is the minimum of elapsed time since the arrival of the first data value and the window size W.

The EH data structure maintains buckets over ranges of data points. All data seen in a time range $(w_{i-1}, w_i]$ are aggregated into the ith bucket, and the EH maintains the values w_i and the count

$$C_i = \sum_{w_{i-1} < t \leq w_i} f(t)$$

for each bucket. Buckets where $w_i < T - W$ are discarded. When a new data point arrives, it is placed in its own new bucket. Buckets are then merged in a certain way such that there is always $O(\log N)$ buckets. At any given point T, the estimate on the decaying count $S_{\text{SLIWIN}_N}(T)$ can be obtained from

$$S'_{\text{SLIWIN}_N}(T) = \sum_i C_i \; .$$

A characterization of the merging process of EH is that two consecutive buckets are merged if the combined count of the merged buckets is dominated by the total count of all more-recent buckets. We refer to this aggregation process as *domination-based*. (Note that the factor by which the preceding buckets need to dominate depends on the desired approximation factor.) Furthermore, because all the $f(t)$ are either 0 or 1, the sequence of bucket counts is a non-decreasing sequence of powers of 2, and, for some $k = \Theta(1/\epsilon)$ and some P, for each possible count $2^p < 2^P$, there are exactly k or $k+1$ buckets having count 2^p, there are at most $k+1$ buckets having count 2^P, and no buckets have counts greater than 2^P [9].

We will next argue that EHs can be used to obtain a decaying count for general decay functions. To do so we use the following immediate property of the EH data structure.

Lemma 4.1 *EH for window size N can be used to estimate* SLIWIN *DCs for all window sizes smaller than N ($S_{\text{SLIWIN}_i}(T)$ for all $i \leq N$).*

Proof: Let the points

$$T - N \leq w_0 < w_1 < w_2 \cdots < w_k < T$$

be the end times of buckets in the EH. Let j be the minimum such that $w_j \geq T - i$. Observe that the buckets $j, \ldots, k$ are the buckets of an EH for SLIWIN$_i$, since bucket formation and the domination criterion for merging are independent of the window size. The estimate for $S_{\text{SLIWIN}_{N-i}}(T)$ produced by the EH is thus

$$S'_{\text{SLIWIN}_i}(T) = \sum_{\ell=j}^{k} C_\ell \; . \tag{2}$$

∎

4.2 Cascaded Exponential Histograms (CEH)

We now consider general decay function.

Theorem 1 *Decaying sum under any decay function can be estimated using an Exponential Histogram with window size N.*

Proof: Using summation by parts, we can rewrite the decaying sum under g as

$$\begin{aligned} S_g(T) &= \sum_{T-N\le t<T} f(t)g(T-t) \\ &= g(N)\sum_{T-N\le t<T} f(t) + \\ &\quad \sum_{i=1}^{N-1}(g(N-i)-g(N+1-i))\sum_{T-N+i\le t<T} f(t) \\ &= g(N)S_{\text{SLIWIN}_N}(T) + \\ &\quad \sum_{i=1}^{N-1}(g(N-i)-g(N+1-i))S_{\text{SLIWIN}_{N-i}}(T) \end{aligned} \tag{3}$$

Equation (3) represents the decaying sum $S_g(T)$ as a positively weighted sum of SLIWIN decaying sums. Since a $(1+\epsilon)$-accurate estimate of each of the decaying sums $S_{\text{SLIWIN}_{N-i}}(T)$ can be obtained from the EH, we obtain a $(1+\epsilon)$-estimate for the decaying sum $S_g(T)$. ∎

We now discuss the computation involved in maintaining the approximate count $S'_g(T)$ given the EH. Straightforward use of (3) suggests an $O(N)$ time computation in each time step. It is not hard to see, however, that, using the partition of the EH into buckets, we can considerably reduce the computation: To do so, we first substitute the approximations supplied by Equation (2) for the decaying sums $S_{\text{SLIWIN}_{N-i}}(T)$ in Expression (3). We obtain that the estimated decaying sum is equal to

$$S'_g(T) = g(T-w_0)C_0 + \sum_{j\ge 1}(g(T-w_j)-g(T-w_{j-1}))C_j \; . \tag{4}$$

Direct re-computation of Equation (4) in each time T requires $O(k) = O(\log N)$ steps. It is not hard to show, however, (using similar arguments as in [9]) that the amortized update time can be brought down to a constant. The total memory (words) required for this estimate is $\log(N)$.

For example, suppose consecutive weights in g are $8, 5, 3, 2$. Then, at time $T = 4$, the decaying count is $8f(3) + 5f(2) + 3f(1) + 2f(0)$. This can be rewritten as $2[f(0) + f(1) + f(2) + f(3)] + (3-2)[f(1) + f(2) + f(3)] + (5-3)[f(2) + f(3)] + (8-5)[f(3)]$. Each expression in square brackets is a sliding window decaying count of f; all of these can be approximated with good relative error by a single instance of the EH method. Many of these approximations will be the same; we suppose that $f(1) + f(2) + f(3)$ and $f(2) + f(3)$ are each approximated by $f(2) + f(3)$. Then the decaying count is approximated by $2[f(0) + f(1) + f(2) + f(3)] + (5 - 2)[f(2) + f(3)] + (8 - 5)[f(3)]$.

It is shown in [9] that EHs are essentially optimal for SLIWIN, that is, that a logarithmic number of counters is indeed necessary. The result above thus shows that SLIWIN decay is in the sense the "hardest" decay function, as any other decay function can be estimated using the EH data structure.

In the sequel, we refer to cascaded use of an EH applied to general decay functions as CEH (Cascaded Exponential Histogram).

5 Weight-Based Aggregation

We develop another class of histograms, which we refer to as **Weight-Based Merging** histograms (WBMH). WBMH histograms also aggregate values into buckets but the bucket boundaries are determined by the decay function and the current time and are independent of the particular input stream. This feature is important for giving storage-efficient algorithms, since the boundaries do not need to be explicitly stored per stream.

Each WBMH bucket is such that if t_s and t_f are the start and end time of the bucket, and T is the time the bucket is formed, we have that $g(T + x - t_f) \leq (1 + \epsilon)g(T + x - t_s)$ for all $x \geq 0$. This means that all items aggregated in the bucket are assigned similar weight by the decay function at the time the bucket is formed and as time goes on.

WBMH are applicable to decay functions with the following property: for any fixed time window Δ, the ratio $g(x)/g(x + \Delta)$ is non-increasing with x. (Note that it suffices to require this for $\Delta = 1$.) Stated in words, this property implies that the ratio of the "weights" of two items remains fixed or becomes closer to 1 as time progresses. As argued in the introduction, this is a natural intuitive requirement. This property is possessed by many families of decay functions including Exponential decay (where $g(x)/g(x + \Delta)$ is constant) and Polynomial decay. So intuitively, WBMH are applicable to decay functions that drop off like exponentials or more slowly.

Let $D(g()) = g(1)/g(N)$ be the weight ratio between the initial weight of an item and the weight of the oldest item for which the decay function assigns positive weight. We now outline a deterministic process that determines bucket boundaries as time progresses. Let b_1 be the maximum such that $(1+\epsilon)g(b_1 - 1) \geq g(1)$. Similarly, for $i > 1$, let b_i be maximum such that $(1 + \epsilon)g(b_i - 1) \geq g(b_{i-1})$. The "current" bucket is sealed and a new bucket is started at times T such that $T \bmod b_1 = 0$. Whenever there is an i and two consecutive buckets $[a_s, a_t]$ and $[c_s = a_t + 1, c_t]$ such that $b_i \leq T - c_t$ and $T - a_s \leq b_{i+1} - 1$, the two buckets are merged into a single bucket $[a_s, c_t]$.

Observe that the process above is independent of the stream (the count in each bucket depends on the stream, but the boundaries of each bucket do not). We refer to a range $[b_i, b_{i+1} - 1]$ as a *region*. The total number of regions is $\lceil \log_{1+\epsilon} D(g) \rceil$, and there can be at most two buckets per region, on average (1/2 bucket shared with each neighboring region and 1 bucket contained in the region), so the total number of buckets is $O(\epsilon^{-1} \log D(g))$. If the decay function has the property that $g(x)/g(x + 1)$ is non-increasing with x, we can further argue that the ratio of weights of two items aggregated into the same bucket is at most $1 + \epsilon$.

For example, consider decay $g(x) = 1/x^2$, and suppose $(1 + \epsilon) = 5$. Then the regions are determined by $b_1 = 3$, $b_2 = 7$, $b_3 = 16$, and so on. The weights associated with "spots" in the region are thus

$$\left(1, \frac{1}{4}\right); \left(\frac{1}{9}, \frac{1}{16}, \frac{1}{25}, \frac{1}{36}\right); \left(\frac{1}{49}, \ldots, \frac{1}{225}\right); \ldots$$

(the weights within each pair of parentheses corresponds to a region). This is correct because the weights in each region span a range where the largest value is at most a factor of $5 = (1 + \epsilon)$ times the smallest value. Consider now the formation of buckets starting at time $T = 1$. Initially at $T = 1$ there is a single bucket formed with corresponding weights (1). At $T=2$ the new item is appended to the current bucket and it is sealed as $(1, \frac{1}{4})$. At $T=3$ we have, $(1); (\frac{1}{4}, \frac{1}{9})$, at $T=4$ we have $(1, \frac{1}{4}); (\frac{1}{9}, \frac{1}{16})$, at $T=6$ we have

$$\left(1, \frac{1}{4}\right); \left(\frac{1}{9}, \frac{1}{16}, \frac{1}{25}, \frac{1}{36}\right) ,$$

at $T=8$ we have

$$\left(1, \frac{1}{4}\right); \left(\frac{1}{9}, \frac{1}{16}\right); \left(\frac{1}{25}, \frac{1}{36}, \frac{1}{49}, \frac{1}{64}\right) ,$$

at $T=9$ we have

$$(1); \left(\frac{1}{4}, \frac{1}{9}\right); \left(\frac{1}{16}, \frac{1}{25}\right); \left(\frac{1}{36}, \frac{1}{49}, \frac{1}{64}, \frac{1}{81}\right) ,$$

and at $T=10$

$$\left(1, \frac{1}{4}\right); \left(\frac{1}{9}, \frac{1}{16}, \frac{1}{25}, \frac{1}{36}\right); \left(\frac{1}{49}, \frac{1}{64}, \frac{1}{81}, \frac{1}{100}\right) .$$

Note that the bucket of most recent items always alternates between time-width 1 and time-width 2.

The information stored per stream is the total count of items that arrived within the time boundaries of each bucket. Since each such number is at most N, each number requires $\log(N)$ bits to store exactly, for a total of $O(\log^2(N))$ bits over the $O(\log(N))$ buckets. But the counts need only be stored approximately. In floating point, the exponent requires $\log\log(N)$ bits. We will also store the most significant $\log(1/\beta) = \log(1/\epsilon) + \log\log(N)$ bits of the expansion, where $\beta = \epsilon/\log(N)$. Note that the count in each bucket is obtained by summing exact counts of 1 or 0 in a summation tree of depth $d \leq O(\log(N))$. Recursively assume that, at level i of the summation tree, the values are good to the factor $(1+\beta)^i$. We can add two such counts, getting a sum also good to the factor $(1+\beta)^i$. We next round the sum to the most significant $\log(\beta)$ bits, which is equivalent to multiplying by a number between 1 and $(1+\beta)$. Thus the rounded sum is good to the factor $(1+\beta)^{i+1}$, as desired. Thus, at the top of the tree, the rounded sum is good to $(1+\beta)^{\log(N)} \approx (1+\beta\log(N)) = (1+\epsilon)$, by definition of β.

The foregoing assumes that we know N in advance. More generally, at level i, rounding is equivalent to multiplying by a number between 1 and $1+\beta_i$, where the β_i's are not necessarily all equal. We want $\prod_i (1+\beta_i) \approx \left(1 + \sum_{i=1}^{\log(N)} \beta_i\right)$ to be at most $(1+\epsilon)$, or $\sum_{i=1}^{\log(N)} \beta_i < \epsilon$. For this, it suffices to put $\beta_i \approx \epsilon/i^2$, so that the number of storage bits at level i is $\log(1/\beta_i) = \log(1/\epsilon) + 2\log(i)$, for $i \leq \log(N)$. This way N does not need to be known in advance.

We have established the following:

Lemma 5.1 *Let g be a decay function such that $g(x)/g(x+1)$ is non-increasing with x. The WBMH algorithm uses $O(\log\log(N)\log D(g))$ storage bits and approximates the decaying count to within $1+\epsilon$.*

For which decay functions does the WBMH algorithms outperform CEHs ? It is not hard to see that for ExpD, $\log D(g) = O(N)$, and thus WBMH requires a linear number of buckets and CEH is more efficient. For PolyD, $\log D(g) = O(\log N)$, WBMH uses only a logarithmic number of buckets, and requires considerably fewer bits per bucket than CEH. In total, it uses $O(\log N \log\log N)$ storage bits versus $O(\log^2 N)$ bits used by CEH; thus, there is a quadratic gap. More generally, WBMH beats CEHs also for sub-polynomial decay, as the number of buckets of WBMH is sub-logarithmic in elapsed time; WBMH also beats CEH for slightly super-polynomial decays.

Lastly, we remark that more efficient tracking of polynomially-decaying counts can also be achieved through a CEH with "approximate" maintenance of each histogram time boundary (which would require only $O(\log\log(N))$ bits). For polynomial-decay, a constant factor error in the time boundary translates to a constant factor error in the contribution of the bucket. This relaxation is discussed in [9] (and attributed to Y. Matias).

6 Lower bound for polynomially-decaying counts

We now establish that a logarithmic number of bits are necessary for maintaining polynomially-decaying counts.

Theorem 2 *A logarithmic number of bits (in elapsed time) is necessary for approximately maintaining decay by* $g(x) = 1/x^{\alpha}$.

Proof: We first give the proof for decaying sums (of data that can take non-negative integer values other than 0's and 1's). Later, we extend the proof to 0/1-valued data.

Consider the decay function $g(x) = 1/x^{\alpha}$. Let k be a constant such that $\frac{2}{k} \cdot \frac{k+1}{k-1} < \frac{1}{4}$; *e.g.*, $k = 10$. We will consider a time interval of length N, from $-N/2$ to $N/2$. Let $r = \lfloor \frac{\alpha}{2 \log k} \log(N/2) \rfloor$.

Consider the following family of streams: for $i \leq r$, at time $t = -k^{2i/\alpha}$, we receive a burst of count $C_i \in \{k^i, 2k^i\}$. The streams differ by the size of the burst and thus there are 2^r different streams in our family. Note that the size of the bursts decreases with time (increases with i) and no more data arrives after time $t = -1$.

We now consider the time interval $[0, N/2]$. During this time no more data arrives, but we need to "remember" enough about the stream we had seen so that we can estimate the value of the decaying count, to within some constant accuracy $\epsilon < 1/4$, at all times $t \in [0, N/2]$.

Let $n_i = C_i/k^i \in \{1, 2\}$. A particular stream in the family is determined by the values n_i ($i = 1, \ldots, r$). We will show that we will need to know all the values n_i in order to estimate the decaying count correctly. Thus, at time 0, we will need a storage of at least r bits.

Consider the decaying count value $S_g(t)$ at time $t_i = k^{2i/\alpha}$. We have

$$S_g(t_i) = \sum_{j=1}^{r} g(k^{2i/\alpha} + k^{2j/\alpha}) k^j n_j \ .$$

We now compute upper bounds on the contribution of the prefix and the suffix of this sum. For the prefix of the sum up to term $i-1$ we obtain

$$\begin{aligned} \sum_{j=1}^{i-1} g(k^{2i/\alpha} + k^{2j/\alpha}) k^j n_j &\leq 2 \sum_{j=1}^{i-1} g(2k^{2i/\alpha}) k^j \\ &= \frac{2^{1-\alpha}}{k^{2i}} \sum_{j=1}^{i-1} k^j \\ &\leq \frac{2^{1-\alpha}}{k^{i+1}} \frac{k}{k-1} \ . \end{aligned} \tag{5}$$

For the suffix of the sum, starting at term $i+1$, we obtain

$$\begin{aligned} \sum_{j=i+1}^{r} g(k^{2i/\alpha} + k^{2j/\alpha}) k^j n_j &\leq 2 \sum_{j=i+1}^{r} g(2k^{2j/\alpha}) k^j \\ &\leq 2^{1-\alpha} \sum_{j=i+1}^{\infty} 1/k^j \\ &\leq \frac{2^{1-\alpha}}{k^{i+1}} \frac{1}{k-1} \ . \end{aligned} \tag{6}$$

By combining the bounds (5) and (6) we obtain that

$$\begin{aligned}\sum_{j\neq i} g(k^{2i/\alpha}+k^{2j/\alpha})k^j n_j &\leq 2^{1-\alpha}k^{-(i+1)}\frac{k+1}{k-1} \\ &\leq \frac{2}{k}\cdot\frac{k+1}{k-1}2^{-\alpha}k^{-i}.\end{aligned}$$

Observe now that the ith term is equal to

$$g(2k^{2i/\alpha})k^i n_i = 2^{-\alpha}k^{-i}n_i \ .$$

Since $n_i \geq 1$ and $\frac{2}{k}\cdot\frac{k+1}{k-1} < \frac{1}{4}$, we obtain that the contribution of the prefix and suffix is at most the fraction $1/4$ of the contribution of the ith term. It follows that (independently of the rest of the stream) we must know the value of n_i in order to estimate the decaying count with $\epsilon < 1/4$ at time t_i.

Now suppose we require that the data takes values 0 or 1 and arrives at equally-spaced time steps. Note that the above proof goes through, essentially unchanged, if $\alpha < 2$, since a single burst of count $C_i = \Theta(k^i)$ at time $-k^{2i/\alpha}$ can be spread out to C_i 1's. At time $k^{2i/\alpha}$, the burst will have ocurred $2k^{2i/\alpha}$ to $2k^{2i/\alpha}+k^i$ time units in the past, and so all 1's in the burst will have approximately the same weight. If $\alpha \geq 2$, however, then the proof needs modification, since there isn't time to read k^i 1's from time $-k^{2i/\alpha}$ to 0 (we actually need to absorb the 1's between consecutive bursts). Instead, we proceed as follows. Instead of using bursts of size k^i, use bursts of size $k^i/N^{1/\alpha}$, and only use some of the r slots—the ones for which the size of the burst is at least 1 and at most $\frac{2\epsilon}{\alpha}k^{2i/\alpha}$. First note that, if we divide all the bursts by the same number (still concentrating each burst at a single time instant), the relative contributions of the prefix, suffix, and i'th terms remains the same. Next note that, by spreading out a burst as several 1's into the past, all contributions decrease, compared with a burst ocurring at a single time instant. A decrease in contributions from prefixes and suffixes can only help. If the i'th burst has size at most $\frac{2\epsilon}{\alpha}k^{2i/\alpha}$, then, at time $k^{2i/\alpha}$, the weight of data in this burst varies from $g(2k^{2i/\alpha})$ to $g(2k^{2i/\alpha}+\frac{2\epsilon}{\alpha}k^{2i/\alpha})$; *i.e.,* the weight can decrease by the factor $1/(1+\epsilon/\alpha)^\alpha$, which, assuming $\alpha \geq 2$, is approximately $e^{-\epsilon} \approx (1-\epsilon)$, a tolerable amount. Next, we show that there are $\Omega(\log(N))$ such slots. The requirement $k^i/N^{1/\alpha} \geq 1$ is equivalent to $i \geq \log_k(N)/\alpha$. The requirement $k^i/N^{1/\alpha} \leq \frac{2\epsilon}{\alpha}k^{2i/\alpha}$ is equivalent to

$$i \leq \frac{\log_k(N)}{\alpha-2}+\frac{\alpha\log(2\epsilon/\alpha)}{\alpha-2},$$

i.e., $i \leq \frac{\log_k(N)}{\alpha-2} - \Theta(1)$, when $\alpha > 2$, and vacuous otherwise. Thus the number of allowable i's is $(1/(\alpha-2)-1/\alpha)\log_k(N)-\Theta(1) = \Theta(\log(N))$, as desired. Finally, for this to work, we also need that the i'th and $(i+1)$'st bursts are disjoint. That is, we need $\frac{2\epsilon}{\alpha}k^{2i/\alpha} < k^{2(i+1)/\alpha}-k^{2i/\alpha} = (k^{2/\alpha}-1)k^{2i/\alpha}$. Choose k large enough that $(k^{2/\alpha}-1) \geq 1$. Then, since $\alpha \geq 2$ and $\epsilon = 1/4$, $\frac{2\epsilon}{\alpha}k^{2i/\alpha} \leq k^{2i/\alpha} \leq (k^{2/\alpha}-1)k^{2i/\alpha}$, as desired. Thus, for $\epsilon = 1/4$, given arbitrary α, $\Omega(\log(N))$ bits of storage (depending on α) are required to track $1/x^\alpha$ decay of 0/1-valued data to within the factor $(1+\epsilon)$. ∎

7 Other aggregate functions

7.1 Time-decaying L_p norms

Each data item is an increment update to the value of a coordinate of a d-dimensional vector. The increment is specified by the amount $a_i \in \{0,\ldots,M\}$ and the coordinate $c_i \in [d] = \{0,\ldots,d-1\}$. The d-dimensional vector $\mathbf{H}_g(T)$ is defined by the coordinate values

$$\mathbf{H}_g(T)_j = \sum_{i|t_i<T\wedge c_i=j} g(T-t_i)a_i \ .$$

For a fixed $p \in [1, 2]$, we show how to maintain a summary of size $o(d)$, such that for any decay function $g()$, we can obtain (with high confidence) an approximate value of the L_p norm $||\mathbf{H}_g(T)||_p$.

The sliding-windows version of the problem was formulated in [9] and the proposed solution is based on a sketching technique of Indyk [10]. By using the cascaded computation (CEH) with the standard EH data structures, the solution of [9] extends for arbitrary decay functions.

The basic idea is to (randomly) generate (using Indyk's method) a fixed (over time) $L \times d$ matrix (the matrix entries need not be stored and can be generated from seeds on the fly. They depend on $p \in [1, 2]$.), the value L is determined by the desired confidence and accuracy. Each data point is then multiplied in turn by the entries of the c_ith column of the matrix to obtain a vector of size L. The summary that is maintained over time (just like [9]) are L EHs, where the jth EH keeps the cumulative sum of the j entry of the L-vector. The jth EH thus allows us to retrieve an approximate value of the decaying sum with respect to any decay function. These L values, using Indyk's method, allow us to obtain an approximate value of $||\mathbf{H}_g(T)||_p$. The storage required for this solution (see [9]) is (for fixed confidence and accuracy) $O(\log N(\log N + \log M) + \log M \log d)$.

7.2 Time-decaying Random selection

Consider a stream of items, where t_i is the arrival time of item i. We define $R_g(T)$ to be a random variable that assumes the value i with probability proportional to $g(T - t_i)$, that is,

$$\frac{g(T - t_i)}{\sum_{i|t_i<T} g(T - t_i)} \,.$$

The random selection problem in the more general setting of spatial decay is addressed in [5]. We describe the method in our context of time-decay. The first component is to perform random selection from sliding windows. In order to do that, we draw a uniform random number (*rank*) for each item which arrives and store an item if and only if it has the minimum rank of all items that succeeded it (this list is termed in [5, 3] an MV/D list). It is not hard to see that the expected size of this MV/D list is logarithmic in the total number of items and it allows us to obtain, for each window w, the least-ranked item in that window. Observe that the least-ranked item is in fact a uniform random selection from all items in the w-window. The next component in [5] (translated to our context) is a reduction of performing time-decaying random selection according to an arbitrary decay function to uniform random selection from w-windows (the reduction allows dependencies between the selection for different-size windows) and the decaying count problem.[4]

A *time-decaying approximate p-quantile* is an item that with high probability is a $[p \pm \epsilon]$-quantile of the distribution of values weighted by $g(T - t_i)$. As discussed in [5], using an existing folklore technique, an approximate median with constant confidence arbitrarily close to 1 can be obtained by performing constant number of independent random selections.

7.3 Time-decaying variance

The *time-decaying variance* is defined by

$$V_g^2(T) = \sum_{i|t_i<T} \Big(g(T - t_i)(f_i - A_g(T))^2 \Big) \,.$$

The problem, for the special case of sliding-window decay (where variance is computed over a sliding window), was studied in [1] and an algorithm based on a variant of the EH technique was proposed. The

[4]In fact, we need the estimates on the count to be *unbiased*. "Plain" EHs do not provide unbiased estimates, but a simple method to obtain unbiased estimates is through two MV/D lists [5, 3].

algorithm of [1] can retrieve the w-window variance for all $w \leq N$. It is not clear, however, if there is a variant of this algorithm which applies to time-decaying variance under arbitrary decay functions. A general reduction of the spatially-decaying moments problem to maintaining a polylogarithmic number of spatially-decaying counts was presented in [4]. In particular, this reduction means that we can obtain an approximate time-decaying variance (for any desired decay functions) by maintaining a polylogarithmic number of "plain" EHs (each EH is over a different predicate applied to the values f_i). The reduction in [4] also requires a time-decaying approximate median (which can be obtained efficiently as outlined earlier).

8 Summary

Decay function are important tools for summarizing streams of data values. The use of time-decaying summations predates the theory of stream algorithms, but was essentially limited to Exponential decays, for which there was a known simple and efficient algorithm that requires only $O(\log N)$ storage bits. In recent work, Datar et al [9] considered sliding-window time-decay and developed algorithms for approximately maintaining sum and average aggregates using $O(\log^2 N)$ storage.

We argue that Exponential and sliding window decay lack some important intuitive properties and do not provide sufficient flexibility in tuning the rate of decay. Thus, it is important to develop efficient algorithms for other families of decay functions. One very appealing such family is polynomial decay.

We then extend the technique of [9] and present algorithms for maintaining time-decaying sum (and average) aggregates under general decay functions using $O(\log^2 N)$ storage, in particular, showing that in a sense, sliding windows are the "hardest" decay functions in terms of required storage. We continue and develop a different algorithm which beats the $O(\log^2 N)$ storage bound for some families of decay functions. In particular, polynomially-decaying aggregates can be maintained using $O(\log N \log\log N)$ storage bits. We thus show that, interestingly, polynomial decay can be maintained almost as efficiently as Exponential decay and considerably more efficiently than sliding window decay. We also provide a lower bound of $\Omega(\log N)$ for polynomially-decaying aggregates.

Beyond time-decaying summations and averages, we formulate and discuss efficent algorithms for other natural aggregate functions, including variance, L_p norms, and random sampling.

Beyond the theoretical and conceptual contributions, our work introduces a powerful new tool for a range of applications involving large data streams.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems (PODS 2002)*. ACM, 2002.

[2] 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.

[3] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. System Sci.*, 55:441–453, 1997.

[4] E. Cohen and H. Kaplan. Efficient estimation algorithms for neighborhood variance and other moments. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 2004.

[5] E. Cohen and H. Kaplan. Spatially-decaying aggregation over a network: model and algorithms. In *SIGMOD*. ACM, 2004.

[6] E. Cohen, H. Kaplan, and J. D. Oldham. Managing TCP connections under persistent HTTP. *Computer Networks*, 31:1709–1723, 1999.

[7] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *Proc. of the 2003 ACM Symp. on Principles of Database Systems (PODS 2003)*. ACM, 2003.

[8] Corinna Cortes and Daryl Pregibon. Giga-mining. In *Proceedings of KDD*, New York, August 1998.

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

[10] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate L1-difference algorithm for massive data streams. In *Proc. 39th IEEE Annual Symposium on Foundations of Computer Science*, pages 501–511. IEEE, 1999.

[11] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transaction on Networking*, 1(4), 1993.

[12] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *Proc. of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 63–72. ACM, 2002.

[13] Warren Gilchrist. *Statistical Forecsting*. Wiley, 1976.

[14] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM'88 Conference*, August 1988.

[15] S. Keshav, C. Lund, S. Phillips, N. Reingold, and H. Saran. An empirical evaluation of virtual circuit holding time policies in IP-over-ATM networks. *IEEE J. on Selected Areas in Communication*, 13, 1995.

[16] R. Morris. Counting large numbers of events in small registers. *CACM*, 21:840–842, 1978.