Maintaining Stream Statistics over Sliding Windows (Extended Abstract)

Mayur Datar* Aristides Gionis† Piotr Indyk‡ Rajeev Motwani§

Abstract

We consider the problem of maintaining aggregates and statistics over data streams, with respect to the last N data elements seen so far. We refer to this model as the *sliding window* model. We consider the following basic problem: Given a stream of bits, maintain a count of the number of 1's in the last N elements seen from the stream. We show that using $O(\frac{1}{\epsilon}\log^2 N)$ bits of memory, we can estimate the number of 1's to within a factor of $1+\epsilon$. We also give a matching lower bound of $\Omega(\frac{1}{\epsilon}\log^2 N)$ memory bits for any deterministic or randomized algorithms. We extend our scheme to maintain the sum of the last N positive integers. We provide matching upper and lower bounds for this more general problem as well. We apply our techniques to obtain efficient algorithms for the L_p norms (for $p \in [1,2]$) of vectors under the sliding window model. Using the algorithm for the basic counting problem, one can adapt many other techniques to work for the sliding window model, with a multiplicative overhead of $O(\frac{1}{\epsilon}\log N)$ in memory and a $1+\epsilon$ factor loss in accuracy. These include maintaining approximate histograms, hash tables, and statistics or aggregates such as sum and averages.

1 Introduction

For many recent applications, the concept of a *data stream*, possibly infinite, is more appropriate than a data set. By nature, a stored data set is appropriate when significant portions of the data are queried again and again, and updates are small and/or relatively infrequent. In contrast, a data stream is appropriate when the data is changing constantly (often exclusively through insertions of new elements), and it is either unnecessary or impractical to operate on large portions of the data multiple times. One of the challenging aspects of processing over data streams is that while the length of a data stream may be unbounded, making it impractical or undesirable to store the entire contents of the stream, for many applications[1] it is still important to retain some ability to execute queries that reference past data. In order to support queries of this sort using a bounded amount of storage, it is necessary to devise techniques for storing summary or synopsis information about previously seen portions of data streams. Generally there is a tradeoff between the size of the summaries and the ability to provide precise answers to queries involving past data.

We consider the problem of maintaining statistics over streams with regard to the last N data elements seen so far. We refer to this model as the *sliding window* model. We identify a simple counting problem whose solution is a prerequisite for efficient maintenance of a variety of more complex statistical aggregates: Given a stream of bits, maintain a count of the number of 1's in the last N elements seen from the stream. We show that using $O(\frac{1}{\epsilon}\log^2 N)$ bits of memory, we can estimate the number of 1's to within a factor of $1+\epsilon$. We also give a matching lower bound of $\Omega(\frac{1}{\epsilon}\log^2 N)$ memory bits for any deterministic or randomized algorithm.

We extend our scheme to maintain the sum of the last N positive integers. We provide matching upper and lower bounds for this more general problem as well. We apply our techniques to obtain efficiently algorithms for the L_p norms (for $p \in [1,2]$) of vectors under the sliding window model. Using the algorithm for the basic counting problem, one can adapt many other techniques to work for the sliding window model, with a multiplicative overhead of $O(\frac{1}{\epsilon}\log N)$ in memory and a $1+\epsilon$ factor loss in accuracy. These include maintaining

*Department of Computer Science, Stanford University, Stanford, CA 94305. Email:datar@cs.stanford.edu. Supported in part by NSF Grant IIS-0118173 and Microsoft Graduate Fellowship.

†Department of Computer Science, Stanford University, Stanford, CA 94305. Email:gionis@cs.stanford.edu. Supported in part by NSF Grant IIS-0118173.

‡MIT Laboratory for Computer Science, 545 Technology Square, NE43-373, Cambridge, Massachusetts 02139. Email:indyk@theory.lcs.mit.edu

§Department of Computer Science, Stanford University, Stanford, CA 94305. Email:rajeev@cs.stanford.edu. Supported in part by NSF Grant IIS-0118173 and a grant from the Okawa Foundation.

[1] For example, in order to detect fraudulent credit card transactions, it is useful to be able to detect when the pattern of recent transactions for a particular account differs significantly from the earlier transactional history of that account.

approximate histograms, hash tables, and statistics or aggregates such as sum and averages.

1.1 Motivation, Model, and Related Work: Several applications naturally generate data streams. In telecommunications, for example, *call records* are generated continuously. Typically, most processing is done by examining a call record once, or operating on a "window" of recent call records (e.g., to update customer billing information), after which records are archived and not examined again. Cortes et al [2] report working with AT&T long distance call records, consisting of 300 million records per day for a 100 million customers. A second application is *network traffic engineering*, where information about current network performance (e.g., latency and bandwidth) is generated online and is used to monitor and adjust network performance dynamically [7, 14]. It is generally both impractical and unnecessary to process anything but the most recent data.

There are other traditional and emerging applications where data streams play an important and natural role, e.g., web tracking and personalization (the streams are web log entries or clickstreams), medical monitoring (vital signs, treatments, and other measurements), sensor databases, and financial monitoring, to name but a few. There are also applications where traditional (non-streaming) data is treated as a stream due to performance constraints. In data mining applications, for example, the volume of data stored on disk is so large that it is only possible to make one pass (or perhaps a very small number of passes) over the data [10, 9]. The objective is to perform the required computations using the stream generated by a single scan of the data, using only a bounded amount of memory and without recourse to indexes, hash tables, or other precomputed summaries of the data. Another example is that data streams are generated as intermediate results of pipelined operators during evaluation of a query plan in an SQL database—without materializing some or all of the temporary result, only one pass on the data is possible [3].

In most applications, the goal is to make decisions based on the statistics or models gathered over the "recently observed" data elements. For example, one might be interested in gathering statistics about packets processed by a set of routers over the last day. Moreover, we would like to maintain these statistics in a continuous fashion. This gives rise to the *sliding window model:* Data elements arrive at each instant and expire after exactly N time steps; and, the portion of data that is relevant to gathering statistics or answering queries is set of the last N elements to arrive. The sliding window refers to the window of active data elements at any time instant.

Previous work [1, 5, 11] on stream computations addresses the problems of approximating frequency moments and computing the L_p differences of streams. There has also been work on maintaining histograms [12, 8]. While Jagadish et al [12] address the off-line version of computing optimal histograms, Guha and Koudas [8] give a technique for maintaining near optimal time-based histograms over streaming data. The queries that are supported by histograms constructed in the latter work are range or point queries over the time attribute. In the earlier work, the underlying model is that all the data elements seen so far are relevant. We believe that the sliding window model is perhaps more important since for most applications one is not interested in gathering statistics over outdated data. Maintaining statistics like sum/average, histograms, hash tables, frequency moments, and L_p differences over sliding windows is critical to most applications. To our knowledge, there is no previous work addressing these problems for the sliding window model.

1.2 Summary of Results: We focus on the sliding window model for data streams. We formulate a basic counting problem whose solution can be used as a building block for solving most of the problems mentioned earlier.

Problem. [BASICCOUNTING] Given a stream of data elements, consisting of 0's and 1's, maintain at every time instant the count of the number of 1's in the last N elements.

It is easy to verify that an exact solution requires $\Theta(N)$ bits[2] of memory. For most applications it is prohibitive to use $\Omega(N)$ memory. For instance, consider the network management application where a large number of data packets pass through a router every second. However, in most applications it suffices to produce an approximate answer. Thus, our goal is to provide a good approximation using $o(N)$ memory.

In Section 2, we provide a solution for BASICCOUNTING which uses $O(\frac{1}{\epsilon}\log^2 N)$ bits of memory (equivalently $O(\frac{1}{\epsilon}\log N)$ buckets of size $O(\log N)$) and provides an estimate of the answer at every instant that is within $1+\epsilon$ factor of the actual answer. Moreover, our algorithm does not require an a priori knowledge of N and caters to the possibility that the window size can be changed dynamically. Our algorithm is guaranteed to work with $O(\log^2 N)$ memory as long as the window size is bounded by N. The algorithm takes $O(\log N)$

[2]Note that we measure space complexity in terms of number of bits, rather than number of memory words.

worst-case time to process each new data element's arrival, but only $O(1)$ amortized time per element. Count queries can be processed in $O(1)$ time. The algorithm itself is relatively simple and easy to implement.

Section 3 presents a matching lower bound. We show that any approximation algorithm (deterministic or randomized) for BASICCOUNTING with with relative error $1+\epsilon$ must use $\Omega(\frac{1}{\epsilon}\log^2 N)$ bits of memory. This proves that our algorithm is optimal in terms of memory usage.

In Section 4 we extend the technique to handle data elements with positive integer values, instead of just binary values, referred to as the SUM problem. We provide matching upper and lower bounds on the memory usage for this general problem as well. Then, in Section 5 we show we can use our techniques along with the sketching techniques of Indyk [11] to efficiently maintain the L_p norms (for $p \in [1,2]$) of vectors under the sliding window model.

Section 6 provides a brief discussion of the application of the BASICCOUNTING and SUM algorithms to adapting several other problems in the sliding window model, such as maintaining histograms, hash tables, and statistics or aggregates such as averages/sums. The reduction of these problems to BASICCOUNTING entails a multiplicative overhead of $O(\frac{1}{\epsilon}\log N)$ in memory and a $1+\epsilon$ factor loss in accuracy. We also discuss problems such as Min/Max and Distinct Values.

2 Algorithm for BASICCOUNTING

Our approach towards solving the BASICCOUNTING problem is to maintain a histogram that records the timestamp of selected 1's that are *active* in that they belong to the last N elements. We call this histogram the Exponential Histogram (EH) for reasons that will be clear later. Before getting into the details of our algorithms we need to introduce some notation.

We follow the conventions illustrated in Figure 1. In particular, we assume that new data elements are coming from the right and the elements at the left are ones already seen. Note that each data element has an *arrival time* which increments by one at each arrival, with the leftmost element considered to have arrived at time 1. But, in addition, we employ the notion of a *timestamp* which corresponds to the position of an *active* data element in the current window. We timestamp the active data elements from right to left, with the most recent element being at position 1. Clearly, the timestamps change with every new arrival and we do not wish to make explicit updates. A simple solution is to record the arrival times in a wraparound counter of $\log N$ bits and then the timestamp can be extracted by comparison with counter value of the current arrival. As mentioned earlier, we concentrate on the 1's in the data stream. When we refer to the k-th 1, we mean the k-th most recent 1 encountered in the data stream.

We will maintain histograms for the active 1's in the data stream. For every bucket in the histogram, we keep the timestamp of the most recent 1 (called *timestamp*), and the number of 1's (called *bucket size*). When the timestamp of a bucket expires (reaches $N+1$), we are no longer interested in data elements contained in it, so we drop that bucket and reclaim its memory. If a bucket is still active, we are guaranteed that it contains at least a single 1 that has not expired. Thus, at any instant there is at most one bucket (the last bucket) containing 1's which may have expired. At any time instant we may produce an estimate of the number of active 1's as follows. For all but the last bucket, we add the number of 1's that are there in them. For the last bucket, let C be the count of the number of 1's in that bucket. The actual number of active 1's in this bucket could be anywhere between 1 and C, so we estimate it to be $C/2$. We obtain the following:

FACT 2.1. *The absolute error in our estimate is at most $C/2$, where C is the size of the last bucket.*

Note that for this approach the window size does not have to be fixed a-priori at N. Given a window size S, we calculate the expiry time and do the same thing as before except that the last bucket is the bucket with the largest timestamp that is less than the expire time.

2.1 The Approximation Scheme: We now define the Exponential Histograms and present a technique to maintain them, so as to guarantee count estimates with relative error at most ϵ, for any $\epsilon > 0$. Define $k = \lceil \frac{1}{\epsilon} \rceil$, and assume that $\frac{k}{2}$ is an integer; if $\frac{k}{2}$ is not an integer we can replace $\frac{k}{2}$ by $\lceil \frac{k}{2} \rceil$ without affecting the basic results.

As per Fact 2.1, the absolute error in the estimate is $C/2$, where C is the size of the last bucket. If C_i is the size of the i-th bucket, we know that the true count is at least $1+\sum_{i=1}^{m-1} C_i$, since the last bucket contains at least one 1 and the remaining buckets contribute exactly their size to total count. Thus, the relative estimation error at most $C_m/2(1+\sum_{i=1}^{m-1} C_i)$. We will ensure that the relative error is at most $1/k$ by maintaining the following invariant:

INVARIANT 2.1. *At all times, the bucket sizes $C_1, \ldots, C_m$ are such that: For all $j \leq m$, we have $C_j/2(1+\sum_{i=1}^{j-1} C_i) \leq \frac{1}{k}$.*

Let $N' \leq N$ be the number of 1's that are active at any instant. Then the bucket sizes must satisfy

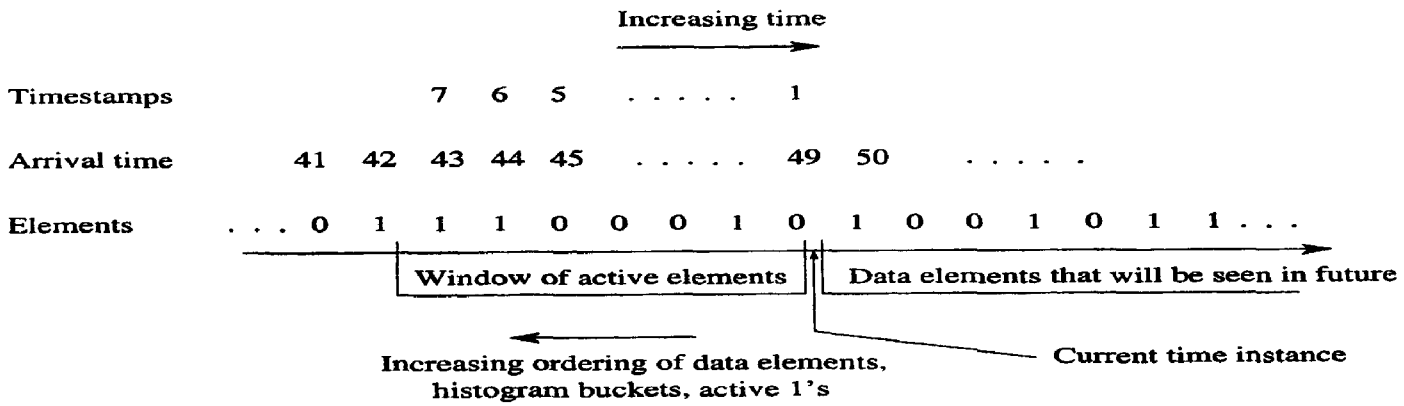

Figure 1: An illustration for the notation and conventions followed.

$\sum_{i=1}^{m} C_i \geq N'$. In order to satisfy this and Invariant 2.1 with as few buckets as possible, we maintain buckets with exponentially increasing sizes so as to satisfy the following second invariant.

INVARIANT 2.2. *At all times the bucket sizes are non-decreasing, i.e.,* $C_1 \leq C_2 \leq \cdots \leq C_{m-1} \leq C_m$. *Further, the bucket sizes are constrained to the following:* $\{1, 2, 4, \ldots, 2^{m'}\}$, *for some* $m' \leq m < \log \frac{2N}{k} + 1$. *Let* 2^h *be the size of the last bucket; then, for every bucket size other than the size of the last bucket, there are at most* $\frac{k}{2} + 1$ *and at least* $\frac{k}{2}$ *bucket of that size.*

Let $C_j = 2^r$ be the size of the j-th bucket. If Invariant 2.2 is satisfied, then we are guaranteed that there are at least $\frac{k}{2}$ buckets each of sizes $1, 2, 4, \ldots, 2^{r-1}$ which have indexes less than j. Consequently, $C_j \leq \frac{2}{k}(1 + \sum_{i=1}^{j-1} C_i)$. It follows that if Invariant 2.2 is satisfied then Invariant 2.1 is automatically satisfied. If we maintain Invariant 2.2, it is easy to see that to cover all the active 1's, we would require no more than $m \leq (\frac{k}{2} + 1)(\log(\frac{2N}{k} + 1) + 1)$ buckets. Associated with bucket is its size and a timestamp. The bucket size takes at most $\log N$ values and hence we can maintain them using $\log \log N$ bits. Since a timestamp requires $\log N$ bits, the total memory requirement of each bucket is $\log N + \log \log N$ bits. Therefore, the total memory requirement (in bits) for an EH is $O(\frac{1}{\epsilon} \log^2 N)$. It is implied that by maintaining Invariant 2.2, we are guaranteed the desired relative error and memory bounds.

The query time for EH is $O(1)$. We achieve this by maintaining two counters, one for the size of the last bucket (LAST) and one for the sum of the sizes of all buckets (TOTAL). The estimate itself is TOTAL minus half of LAST. Both counters can be updated in $O(1)$ time for every data element. The following is a detailed description of the update algorithm.

Algorithm Insert:

1. When a new data element arrives, calculate the new expiry time. If the timestamp of the last bucket indicates expiry, delete that bucket and update the counter LAST containing the size of the last bucket and the counter TOTAL containing the total size of the buckets.
2. If the new data element is 0 ignore it; else, create a new bucket with size 1 and the current timestamp, and increment the counter TOTAL.
3. Traverse the list of buckets in order of increasing sizes. If there are $\frac{k}{2} + 2$ buckets of the same size, merge the oldest two of these buckets into a single bucket of double the size. (A merger of buckets of size 2^r may cause the number of buckets of size 2^{r+1} to exceed $\frac{k}{2} + 2$, leading to a cascade of such mergers.) Update the counter LAST if the last bucket is the result of a new merger.

Example. We illustrate the algorithm for a few steps. Assume that $\frac{k}{2} = 2$ and that the current bucket sizes from left to right are 32, 16, 8, 8, 4, 4, 2, 1, 1. When a new 1 arrives, the older 1's are merged and the bucket sizes become 32, 16, 8, 8, 4, 4, 2, 2, 1. After two more 1's arrive, the merging cascades up to the buckets of size 8, and we get buckets of sizes 32, 16, 16, 8, 4, 2, 1.

Merging two buckets corresponds to creating a new bucket whose size is equal to the sum of the sizes of the two buckets and whose timestamp is the timestamp of the older bucket. A merger requires $O(1)$ time. Moreover, while cascading may require $\Theta(\log \frac{2N}{k})$ mergers upon the arrival of a single new element, standard arguments allow us to argue that the amortized cost of mergers is $O(1)$ per new data element. We obtain the following theorem:

THEOREM 2.1. *The EH algorithm maintains a data structure which can give an estimate for the* BASIC-COUNTING *problem with relative error at most* ϵ *using at*

most $(\frac{k}{2}+1)(\log(\frac{2N}{k}+1)+1)$ *buckets, where* $k=\lceil\frac{1}{\epsilon}\rceil$. *The memory requirement is* $\log N+\log\log N$ *bits per bucket. The arrival of each new element can be processed in* $O(1)$ *amortized time and* $O(\log N)$ *worst-case time. At each time instant, the data structure provides a count estimate in* $O(1)$ *time.*

If instead of maintaining a timestamp for every bucket, we maintain a timestamp for the most recent bucket and maintain the difference between the timestamps for the successive buckets then we can reduce the total memory requirement to $O(k\log^2\frac{N}{k})$.

3 Lower Bounds

We provide a lower bound which verifies that the EH Algorithm is optimal in its memory requirement. We start with a deterministic lower bound of $\Omega(k\log^2\frac{N}{k})$.

THEOREM 3.1. *Any deterministic algorithm that provides an estimate for the* BASICCOUNTING *problem at every time instant with relative error less than* $\frac{1}{k}$ *for some integer* $k\leq 4\sqrt{N}$ *requires at least* $\frac{k}{16}\log^2\frac{N}{k}$ *bits of memory.*

The proof argument will go as follows. We will show that there are a large number of arrangements of 0's and 1's such that any deterministic algorithm which provides estimates with small relative error has to differentiate between every pair of these arrangements. The number of memory bits required by such an algorithm must therefore exceed the logarithm of the number of arrangements. The above argument is formalized by the following lemma.

LEMMA 3.1. *For* $k/4\leq B\leq N$, *there exist* $L=\binom{B}{k/4}^{\lfloor\log\frac{N}{B}\rfloor}$ *arrangements of* 0*'s and* 1*'s of length* N *such that any deterministic algorithm for* BASICCOUNTING *with relative error less than* $\frac{1}{k}$ *must differentiate between any two of the arrangements.*

Proof. We partition a window of size N into blocks of size $B,2B,4B,\ldots,2^jB$ from right to left, for $(j=\lfloor\log\frac{N}{B}\rfloor-1)$. Consider the i-th block of size 2^iB and subdivide it into B contiguous sub-blocks of size 2^i. For each block, we choose $\frac{k}{4}$ sub-blocks and populate them with 1's, placing 0's in the remaining positions. In every block, there are $\binom{B}{k/4}$ possible ways to place the 1's, and therefore the total number of distinct arrangements is $L=\binom{B}{k/4}^{\lfloor\log N/B\rfloor}$.

We now argue that any deterministic algorithm for BASICCOUNTING with relative error less than $\frac{1}{k}$ must differentiate between any pair of these arrangements. In other words, if there exists a pair of arrangements A_x,A_y such that a deterministic algorithm does not differentiate between them, then after some time interval the two arrangements will have different answers to the BASICCOUNTING problem and the algorithm will give a relative error of at least $\frac{1}{k}$ for one of them. To this end, we will assume that the algorithm is presented with one of these L arrangements of length N, followed by a sequence of all 0's of length N.

Refer to Figure 2 for an illustration of a pair of arrangements that should be differentiated by any deterministic algorithm with relative error less than $\frac{1}{8}$.

Consider an algorithm that does not differentiate between two of the above arrangements A_x and A_y. We will use the numerical sequences $x_0,x_1,\ldots,x_j$ and $y_0,y_1,\ldots,y_j$, for $j=\lfloor\log\frac{N}{B}\rfloor-1$, to encode the two arrangements. The i-th number in the sequence specifies the choice of the $k/4$ sub-blocks from the i-th block which are populated with 1's. The two sequences must be distinct since the two arrangements being encoded are distinct. Let d be an index of a point where the two sequences differ, i.e., $x_d\neq y_d$. Then the two arrangements have a different choice of $k/4$ sub-blocks in the d-th block. Number the sub-blocks within block d from right to left, and let h be the highest numbered sub-block that is chosen for one of the arrangements (say A_x) but not for the other (A_y).Consider the time instant when this sub-block h expires. At that instant, the number of active sub-blocks in block d for arrangement A_x is c, where $c+1\leq k/4$, while the number of active sub-blocks in block d for A_y is $c+1$. Since the arrangements are followed by a sequence of 0's, at this time the correct answer for A_x is $c2^d+\frac{k}{4}(2^d-1)$, while for A_y the correct answer is $(c+1)2^d+\frac{k}{4}(2^d-1)$. Thus, the algorithm will give an absolute error of at least 2^{d-1} for one of the arrangements, which translates to a relative error of $\frac{1}{k}$ at that point in time. ■

To prove Theorem 3.1, observe that if we choose $B=\sqrt{Nk}$ then $\log L\geq\frac{k}{16}\log^2\frac{N}{k}$. We also extend the lower bound on the space complexity to randomized algorithms. The proofs for the following two theorems are omitted. They follow easily from Yao's Minimax Principle [13] and Lemma 3.1.

THEOREM 3.2. *Any randomized Las Vegas algorithm for* BASICCOUNTING *with relative error less than* $\frac{1}{k}$, *for some integer* $k\leq 4\sqrt{N}$, *requires at least* $\frac{k}{16}\log^2\frac{N}{k}$ *bits of memory.*

THEOREM 3.3. *Any randomized Monte Carlo algorithm for* BASICCOUNTING *problem with relative error less than* $\frac{1}{k}$, *for some integer* $k\leq 4\sqrt{N}$, *with prob-*

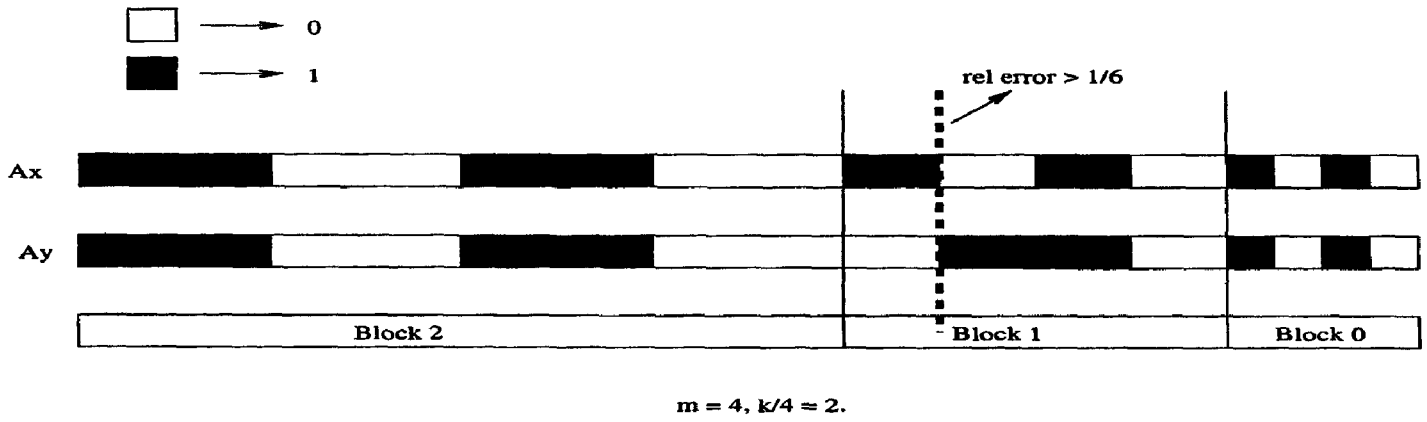

Figure 2: A pair of arrangements that should be differentiated by any deterministic algorithm with relative error less than 1/8.

ability at least $1-\delta$ *(for* $\delta < \frac{1}{2}$*) requires at least* $\frac{k}{64}\log^2\frac{N}{k} - \log(1-\delta)$ *bits of memory.*

4 Beyond 0's and 1's

Consider now the extension of BASICCOUNTING to the case where the elements are positive integers:

Problem. [SUM] Given a stream of data elements that are positive integers in the range $[0 \ldots R]$, maintain at every time instant the sum of the last N elements.

We assume that $\log R = o(N)$. This is a realistic assumption which simplifies our calculations. We generalize EH to this setting as follows. View the arrival of a data element of value v as the arrival of v data elements with value 1 all at the same time and employ the same insertion procedure as before. Note that the algorithm in Section 2 does not require distinct timestamps, they are only required to be nondecreasing. While earlier there could be at most N active 1's, now there could be as many as NR. As before, let $k = \lceil \frac{1}{\epsilon} \rceil$. The results in in Section 2 imply that the EH will require at most $(\frac{k}{2}+1)(\log(\frac{2NR}{k}+1)+1)$ buckets. Now, each bucket will require $\log N + \log(\log N + \log R)$ bits of memory to store the timestamp and the size of the bucket. Note that there are N distinct timestamps at any point (as before), but that the bucket sizes could take on $\log N + \log R$ distinct values. Thus, the number of memory bits required is $O(\frac{1}{\epsilon}(\log N + \log R)(\log N))$. The only catch appears to be that we need $\Omega(R)$ time per insertion. The rest of the section is devoted to devising a scheme that requires only $O(\frac{\log R}{\log N})$ amortized time and $O(\log N + \log R)$ worst case time per insertion. Note that if $R = O(poly(N))$ then the amortized insertion time becomes $O(1)$ and the worst case time becomes $O(\log N)$.

Let S be the total size of the buckets at some time instant. For $j = \log(\frac{2NR}{k}+1)$, let $k_0, k_1, \ldots, k_j$ be a sequence in which k_i denotes the number of buckets of size 2^i. Then, $S = \sum_{i=0}^{j} k_i 2^i$. By Invariant 2.2, we have $l \le k_i \le l+1$, for $i < j$, and $1 \le k_j \le l+1$, where $l = \frac{k}{2} \ge 1$. Given $l \ge 1$ and S, a sequence $k_0, k_1, \ldots, k_j$ satisfying the above conditions is called an l-canonical representation of S. The algorithm represents every valid sum in its l-canonical form. We claim that the l-canonical representation of any sum S is unique and can be computed in time $O(\log S)$.

LEMMA 4.1. *The l-canonical representation of any positive number S is unique.*

Proof. We give a proof by contradiction. Assume that $\mathbf{k} = (k_0, k_1, \ldots, k_j)$ and $\mathbf{k'} = (k'_0, k'_1, \ldots, k'_{j'})$ are two distinct l-canonical representations of S. Without loss of generality, assume that $j \le j'$. Let d be the smallest index where the sequences differ. We have $d \le j$ since it cannot happen that they agree on all the indices less than or equal to j and the second sequence has nonzero components for indices greater than j, given that they have the same sum.

Case1 $(d < j)$: Since $l \le k_d, k'_d \le l+1$, we have $|\sum_{i=0}^{d} k_i 2^i - \sum_{i=0}^{d} k'_i 2^i| = 2^d$. However, $|\sum_{i=d+1}^{j} k_i 2^i - \sum_{i=d+1}^{j'} k'_i 2^i| = c2^{d+1}$, for some $c \ge 0$, which is a contradiction since $|\sum_{i=0}^{j} k_i 2^i - \sum_{i=0}^{j'} k'_i 2^i| = 0$.

Case2 $(d = j)$: The sequence $\mathbf{k'}$ must have nonzero indices greater than j, otherwise the two representations cannot give the same sum. Moreover, it cannot happen that $k_j \le k'_j$, since otherwise $\mathbf{k'}$ will have a strictly greater sum. Thus, $k_j > k'_j$ and $k_j \le l+1$. Since k'_j is not the last index, we have $k'_j \ge l$. Therefore $|k'_j - k_j| \le$

1, which implies $|\sum_{i=0}^{j} k_i 2^i - \sum_{i=0}^{j} k_i' 2^i| \leq 2^j$. However, $\sum_{i \geq j+1}^{k_i'} 2^i \geq 2^{j+1}$ which gives a contradiction. ■

The following procedure computes the l-canonical representation of S in time $O(\log S)$.

Procedure l-Canonical: Given S find the largest j such that $2^j \leq \frac{S}{l}+1$ and let $S' = S-(2^j-1)l$. If $S' \geq 2^j$, find m such that $m2^j \leq S' < (m+1)2^j$ and set $k_j = m$; we are guaranteed that $m < l$. Let $\widehat{S} = S' - m2^j < 2^j$. Let $b_0, \ldots, b_{j-1}$ be the binary representation of $\widehat{S}$. Set $k_i = l + b_i$ for $i < j$.

Given S and l, the l-canonical representation of S tells us the exact positions of all the 1's where the buckets will start. Note that since multiple 1's "belong" to the same data element, we may have multiple buckets starting at a single data element, implying that multiple buckets could have the same timestamp. The following observation is critical to the incremental maintenance of the buckets. The algorithm in Section 2 guarantees that if a certain data element (which in that case was some active 1) is not "indexed" at a certain time interval then it will never be "indexed" in the future. By "indexed" we mean that it is the first element of some bucket and hence its timestamp is maintained as the timestamp of that bucket. As time progresses, buckets may get merged and some data elements may not be indexed any more. However, it never happens that an element that was not indexed at some time gets indexed later.

The preceding observation allows us to devise the following scheme to incrementally maintain the buckets with small amortized update time. Let us assume that we know the buckets at a certain time instant. We think of each data element as series of 1's. We buffer B new elements separately and maintain the sum for these elements; that is, the EH is not updated for B steps. During this period, any query can be answered using a combination of the EH and the buffer sum. When the buffer gets full, we first delete any expired buckets in the EH. After the expired buckets are deleted, let S_1 be the sum of the sizes of the active buckets. Let S_2 be sum of the elements in the buffer. We calculate the l-canonical ($l = \frac{k}{2}$) representation of $S_1 + S_2$ to determine the positions of the new buckets. This requires $O(\log(S_1 + S_2)) = O(\log N + \log R)$ time since $S_1 + S_2 = O(NR)$. We then create the new buckets using the timestamps and values of the elements in the buffer, and the timestamps and sizes of the old buckets. The total time required to process the B elements in buffer is $O(B + \log N + \log R)$, since $O(B)$ time suffices to maintain the buffer sum and the number of buckets in the new histogram is $O(\log N + \log R)$. Since the time required to construct the new histogram is $O(\log N + \log R + B)$, the amortized update time per element is $O(1 + \frac{\log N + \log R}{B})$. Choosing $B = \Theta(\log N)$ makes the amortized update time $O(\frac{\log R}{\log N})$ and worst case time $O(\log N + \log R)$. The buffer needs $O(\log N(\log N + \log R))$ memory bits, which is the same as the memory requirement of the EH. Note that if R is *poly*(N) then the amortized update time is $O(1)$ and worst case time is $O(\log N)$. We have obtained a memory upper bound of $O(\frac{1}{\epsilon}(\log N + \log R)(\log N))$ bits, as summarized in the following theorem.

THEOREM 4.1. *The generalized EH for the* SUM *problem maintains a data structure which provides estimates with relative error at most ϵ using at most $(\frac{k}{2}+1)(\log(\frac{2NR}{k}+1)+1)$ buckets, where $k = \lceil \frac{1}{\epsilon} \rceil$. The memory requirement is $\log N + \log(\log N + \log R)$ bits per bucket. The arrival of each new element can be processed in $O(\frac{\log R}{\log N})$ amortized time and $O(\log N + \log R)$ worst case time. At each time instant, the data structure provides a sum estimate in $O(1)$ time.*

We now prove a lower bound of $\Omega(\frac{1}{\epsilon}(\log N + \log R)(\log N))$ bits. If $\log N = \Omega(\log R)$ then the lower bound from Section 3 applies. Thus, we only need to consider the case when $R > N$. We will assume that $\log R \leq \frac{N}{k}$; in fact we assume $\log R = o(N)$. Consider the following arrangements. We break the window of size N into $\log R$ blocks, each of size $\lfloor \frac{N}{\log R} \rfloor$. Consider the i-th block, for $0 \leq i < \log R$. We choose $k/4$ of the $\lfloor \frac{N}{\log R} \rfloor$ positions and place an element with value 2^i there, setting all other elements to 0. By an argument similar to the one in Section 3, any deterministic algorithm with relative error less than $\frac{1}{k}$ must differentiate between any two of these arrangements. The total number of these arrangements is $\binom{N/\log R}{k/4}^{\log R} \geq (\frac{4N}{k \log R})^{\frac{k}{4} \log R}$. The number of memory bits required is at least $\frac{k}{4} \log R \log(\frac{4N}{k \log R}) = \Omega(\frac{1}{\epsilon}(\log N + \log R)(\log N))$. We assume that $R > N$ and that $\log R = O(N^\delta)$ for some $\delta < 1$. Note that the lower bounds also apply for randomized algorithms that provide an approximate answer.

5 Computing L_p norms for vectors

We now extend the EH technique and combine it with the sketching technique from Indyk [11] to compute the L_p norms of vectors in the sliding window model. Assume that the window is broken into smaller contiguous buckets. These are numbered right to left and are denoted by $B_1, B_2, \ldots, B_m$. Consider a function f, defined over the intervals, with the following properties:

P1: $f(B_i) \geq 0$.

P2: $f(B_i) \leq poly(|B_i|)$.

P3: $f(B_1 + B_2) \geq f(B_1) + f(B_2)$, with $B_1 + B_2$ the concatenation of adjacent buckets B_1 and B_2.

P4: $f(B_1 + B_2) \leq C_f(f(B_1) + f(B_2))$, where $C_f \geq 1$ is a constant.

P5: The function $f(B)$ admits a "sketch" which requires $g_f(|B|)$ space and is composable, i.e., the sketch for $f(B_1 + B_2)$ can be composed efficiently from the sketches for $f(B_1)$ and $f(B_2)$.

If the function f admits these properties then we can efficiently estimate it for sliding windows using the EH technique. We maintain buckets with the following two invariants; we also associate with every bucket a timestamp and the sketch.

INVARIANT 5.1. $f(B_{n+1}) \leq \frac{C_f}{k} \sum_{i=1}^{n} f(B_i)$.

INVARIANT 5.2. $f(B_{n+2}) + f(B_{n+1}) > \frac{1}{k} \sum_{i=1}^{n} f(B_i)$.

Observation 1: We estimate the function f for the current window by composing the sketches of all but the earliest (leftmost) bucket. The leftmost bucket may have certain expired data elements along with a suffix of data elements that are active. Let B_x be the part (suffix) of the leftmost bucket that is active and was ignored (did not contribute to the estimate). Let B_y be the concatenation of the all the other buckets whose sketch we compose using the sketches of the individual buckets. Then $B_x + B_y$ is the current window and the exact answer is $f(B_x + B_y)$. However we estimate the answer as $f(B_y)$; thus, we always underestimate. The relative error E_r is $\frac{f(B_x+B_y)-f(B_y)}{f(B_x+B_y)} > 0$, by **P1** and **P3**. Also we have

$$\begin{aligned} E_r &\leq \frac{f(B_x+B_y)-f(B_y)}{f(B_y)} && \textbf{(P1, P3)} \\ &\leq \frac{C_f(f(B_x)+f(B_y))-f(B_y)}{f(B_y)} && \textbf{(P4)} \\ &= \frac{C_f f(B_x)}{f(B_y)} + C_f - 1 \\ &\leq \frac{C_f f(B_{n+1})}{\sum_{i=1}^{n} f(B_i)} + C_f - 1 && \textbf{(P1, P3)} \\ &\leq \frac{C_f^2}{k} + C_f - 1 && \text{(Invariant 5.1)} \end{aligned}$$

Observation 2: Invariant 5.2 and property **P2** imply that the number of buckets will be $O(k \log N)$, where N is the size of the window. Thus, the memory required to maintain the time-stamp and sketches for all the buckets will be $O(k \log N(\log N + g_f(N)))$.

Hence, if we maintain the invariants along with the timestamp and the sketches, we can estimate the function f with relative error $0 \leq E_r \leq \frac{C_f^2}{k} + C_f - 1$ using $O(k \log N(\log N + g_f(N)))$ memory bits. We can maintain the invariants along with the timestamp and sketches as new data elements are added. The algorithm to do this is very similar to that for EH.

1. When a new data element arrives, calculate the new expiry time. If the timestamp of the last bucket indicates expiry, delete that bucket.
2. Create a new bucket with just the new data element.
3. Traverse the list of buckets from right to left. If Invariant 5.2 is violated for a pair of buckets (B_{n+1}, B_{n+2}), merge them into a new bucket B'_{n+1}. The sketch for this bucket is composed from the sketches for B_{n+1} and B_{n+2}. We may need to do more than one merge.

We argue that the algorithm maintains Invariant 5.1 and Invariant 5.2. Adding a new bucket does not violate Invariant 5.1, as we only increase the size of the suffix. Whenever Invariant 5.2 is violated, the two buckets involved satisfy $(f(B_{n+2}) + f(B_{n+1}) \leq \frac{1}{k} \sum_{i=1}^{n} f(B_i))$. When we merge them, property **P4** guarantees that $f(B'_{n+1}) \leq \frac{C_f}{k} \sum_{i=1}^{n} f(B_i)$ and hence Invariant 5.1 is valid for the new bucket B'_{n+1}. The algorithm may need to do a lot of merges, as many as the number of buckets $(O(\log N))$. However, the amortized time is $O(1)$. We omit details dealing with the fact that the function f for a window of size 1 may be greater than 1 although bounded by some constant R.

THEOREM 5.1. *A function f with properties* **P1–P5** *can be estimated over sliding windows with relative error $0 \leq E_r \leq \frac{C_f^2}{k} + C_f - 1$ using $O(k \log N(\log N + g_f(N)))$ bits of memory.*

5.1 L_p Norms: We now argue that L_p norms (for $p \in [1, 2]$) of vectors under a restricted model admit the properties **P1–P5** and hence can be efficiently computed for sliding windows. Consider the restricted model [11] in which each data element is a pair (i, a), where $i \in [d] = 0 \ldots d-1$ and $a \in 0 \ldots M$ represents an increment to the ith dimension of an underlying vector. Every window B represents a vector and its $L_p(B)$ norm is given by $L_p(B) = (\sum_{i \in [d]} |\sum_{(i,a) \in B} a|^p)^{1/p}$.

Note that the case $p = 1$ is the same as the SUM problem. We denote $(L_p)^p$ by f_p and estimate f_p, for $p \in [1, 2]$. The function f_p clearly admits properties **P1–P4**. For **P5**, $f_p(B)$ admits a sketching technique which requires $O(\log M \log(1/\delta)/\hat{\epsilon}^2)$ memory bits per sketch and is composable. The technique also requires $O(\log M \log(d/\delta) \log(1/\delta)/\hat{\epsilon}^2)$ random bits which are common to all sketches. (See Theorem 2 in [11].) The sketches for computing the function are not exact. They

provide an approximation with relative error less than $\hat{\epsilon}$ with probability $1-\delta$. However, by setting the accuracy parameter $\hat{\epsilon}$ correctly we can make sure that our algorithm also works in an probabilistic manner and has relative error at most $\frac{4}{k}+1+\epsilon$, where ϵ is a function of $\hat{\epsilon}$.

This proves that under the restricted model we can compute $(L_p)^p$ with relative error at most $\frac{4}{k}+1+\epsilon$ using $O(k \log N(\log N + \log M \log(1/\delta)/\hat{\epsilon}^2))$ bits of memory. The estimate is probabilistically approximately correct. Note that computing $(L_p)^p$ with small relative error translates to computing L_p with a small relative error.

5.2 Lower Bounds: The BASICCOUNTING and SUM problems are the special cases of computing L_p norms, where the underlying vector has a single dimension. Thus, the lower bounds for these problems apply to the problem of computing the L_p norm. Note that the upper bounds obtained in this section match the lower bounds. The L_p norm for $p = 0$ is defined as the distinct value problem and we deal with this problem in Section 6.

6 Applications

We briefly discuss how the EH algorithm for BASICCOUNTING can be used as a building block to adapt several techniques to the sliding window model with a multiplicative overhead of $O(\frac{1}{\epsilon}\log N)$ in memory and a $1+\epsilon$ factor loss in accuracy. The basic idea is that to adapt to the sliding window setting a scheme relying on exact counters for positive integers, use an EH to play the role of a counter. A counter would have required $\Omega(\log N)$ memory bits, while EH requires $O(\frac{1}{\epsilon}\log^2 N)$ memory bits and maintains the count within $1+\epsilon$ error.

6.1 Hash Tables: This is the simplest case. Every data element gets hashed to a bucket. Instead of maintaining a counter for each bucket, we use the EH to maintain approximate counts of the number of data elements hashed into the bucket from the last N data elements in the stream. This works if we are required to maintain only the count of elements in each hash bucket.

6.2 Sums and Averages: In Section 4, we showed to maintain the sum of positive integer data elements using the generalized version of the EH. This requires $O(\frac{1}{\epsilon}\log N(\log R + \log N))$ bits of memory. Since maintaining sum would require $\log N + \log R$ bits the multiplicative overhead is $O(\frac{1}{\epsilon}\log N)$. Maintaining averages is similar.

6.3 Histograms: Given the bucket boundaries in a histogram, we can maintain the sum, average, and other statistics corresponding to each bucket using the generalized EH. Finding the optimal bucket boundaries to optimize the memory requirement is an orthogonal problem. Also equi-width histograms are a natural choice of histograms for which the bucket boundaries are fixed. Note that unlike the histograms discussed in [8] these are not time-based histograms, but instead could be based on any attribute of the data.

6.4 Min and Max: We prove a lower bound for the memory requirement of an algorithm that maintains min or max over a sliding window. The lower bound is based on a counting argument like the one used to prove the lower bound for BASICCOUNTING. Let the data elements be drawn from a set of R distinct numbers. Consider all *nondecreasing* arrangements of N numbers. The number of such arrangements is $\binom{N+R}{N}$. Any deterministic algorithm that gives the correct answer at every time instant must differentiate between any two such arrangements. This is because the two arrangements will have a different minimum at the first place that they differ from left to right. The lower bound on the number of memory bits required is then $\log\binom{N+R}{N} \geq N\log(R/N)$. This lower bound is also valid for any randomized algorithms by arguments similar to the one in Section 3. If $R = poly(N)$ then the lower bound says that we have to store all of the last N elements. The easiest way to maintain the exact minimum over sliding windows is to maintain a list of pairs (value, timestamp) such that both the value and the timestamp are strictly increasing. This scheme has a worst-case space requirement of $O(N \log R)$ bits. However, if the data elements arrive in a random order, the expected space complexity will be $O(\log N \log R)$.

6.5 Distinct Values: dIt is easy to adapt the technique of Flajolet and Martin [6] to estimate the number of distinct elements in the last N data elements. Their *probabilistic counting technique* maintains a bitmap of size $O(\log R)$, where R is an upper bound on the number of distinct values in the data set. In the case of sliding windows, $R \leq N$ and a bitmap of size $O(\log N)$ suffices. We also maintain with each bit a timestamp of size $O(\log N)$. Whenever a bit is (re)set by a data element we update the timestamp to that of the data element. This enables us to keep track of the bits that were set by the last N elements. Consequently, we can estimate the number of distinct elements with an expected relative accuracy of $O(\frac{1}{\sqrt{m}})$ using $O(m\log^2 N)$ bits of memory. Note that the lower bound for BASICCOUNTING problem applies to the Distinct Value problem. Given an instance of BASICCOUNTING problem we can create an input where a 0 is mapped to 0 while every 1 is mapped to some distinct value (the *arrival time* of

the element for instance). Then the number of Distinct Values is one more than the number of ones. This reduction shows that the lower bounds for BASICCOUNTING problem apply to the Distinct Value problem.

References

[1] N. Alon, Y. Matias, M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. Twenty-Eighth Annual ACM Symposium on Theory of Computing*, 1996.

[2] C. Cortes, K. Fisher, D. Pregibon, A. Rogers. Hancock: a language for extracting signatures from data streams. In *Proc. 2000 ACM SIGKDD*, pp. 9–17, 2000.

[3] S. Chaudhuri, R. Motwani, V. R. Narasayya. On random sampling over joins. In *Proc. 1999 ACM SIGMOD*, pp. 263–274, 1999.

[4] M. Fang, H. Garcia-Molina, R. Motwani, N. Shivakumar, J.D. Ullman. Computing iceberg queries efficiently. In *Proc. 24th International Conference on Very Large Data Bases (VLDB)*, 1998.

[5] J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan. An Approximate L1-Difference Algorithm for Massive Data Streams. In *Proc. 40th Symposium on Foundations of Computer Science*, 1999.

[6] P. Flajolet, G. Martin. Probabilistic Counting. In *Proc. 24th Symposium on Foundations of Computer Science*, 1983.

[7] C. Fraleigh, S. Moon, C. Diot, B. Lyles, F. Tobagi. Architecture of a passive monitoring system for backbone IP networks. Technical Report TR00-ATL-101-801, Sprint Labs, 2000.

[8] S. Guha, N. Koudas, K. Shim. Data-Streams and Histograms. To appear in *Proc. Thirty-Third Annual ACM Symposium on Theory of Computing*, 2001.

[9] S. Guha, N. Mishra, R. Motwani, L. O'Callaghan. Clustering data streams. In *Proc. 2000 Annual IEEE Symp. on Foundations of Computer Science*, pages 359–366, 2000.

[10] M. R. Henzinger, P. Raghavan, S. Rajagopalan. Computing on data streams. Technical Report TR 1998-011, Compaq Systems Research Center, Palo Alto, California, May 1998.

[11] P. Indyk. Stable Distributions, Pseudorandom Generators, Embeddings and Data Stream Computation. In *Proc. 41st Symposium on Foundations of Computer Science*, 2000.

[12] Jagadish, Koudas, Muthukrishnan, Poosala, Sevcik, Suel. Optimal Histograms with Quality Guarantees. In *Proc. 24th International Conference on Very Large Data Bases (VLDB)*, 1998.

[13] R. Motwani, P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[14] *Netflow Services and Applications*. Whitepaper, Cisco Systems, 2000. Available at http://www.cisco.com/warp/public/cc/pd/iosw/ioft/neflct/tech/napps_wp.htm.