SIAM J. COMPUT. © 2002 Society for Industrial and Applied Mathematics
Vol. 31, No. 6, pp. 1794–1813

MAINTAINING STREAM STATISTICS OVER SLIDING WINDOWS[*]

MAYUR DATAR[†], ARISTIDES GIONIS[†], PIOTR INDYK[‡], AND 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 and provide matching upper and lower bounds for this more general problem as well. We also show how to efficiently compute the L_p norms ($p \in [1,2]$) of vectors in the sliding window model using our techniques. Using our algorithm, 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.

Key words. statistics, data streams, sliding windows, approximation algorithms

AMS subject classifications. 68P05, 68Q25, 68R01, 68W25

PII. S0097539701398363

1. Introduction.

Traditional database management systems (DBMSs) expect all data to be managed within some form of persistent *data sets*. 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, it is still important to retain some ability to execute queries that reference past data. 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. In order to support queries of this sort using a bounded amount of storage (either in memory or in a traditional DBMS), it is necessary to devise techniques for storing summary or synoptic information about previously seen portions of data streams. Generally there is a tradeoff between the

[*]Received by the editors November 18, 2001; accepted for publication (in revised form) June 8, 2002; published electronically October 18, 2002. A preliminary version of this paper appeared in *Proceedings of the Thirteenth Annual ACM–SIAM Symposium on Discrete Algorithms*, San Francisco, CA, 2002.

http://www.siam.org/journals/sicomp/31-6/39836.html

[†]Department of Computer Science, Stanford University, Stanford, CA 94305 (datar@cs.stanford.edu, gionis@cs.stanford.edu, rajeev@cs.stanford.edu). The research of the first author was supported in part by NSF grant IIS-0118173 and a Microsoft Graduate Fellowship. The research of the second author was supported in part by NSF grant IIS-0118173 and a Microsoft Graduate Fellowship. The research of the fourth author was supported in part by NSF grant IIS-0118173, an Okawa Foundation research grant, and Veritas.

[‡]MIT Laboratory for Computer Science, 545 Technology Square, NE43-373, Cambridge, MA 02139 (indyk@theory.lcs.mit.edu).

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 and provide matching upper and lower bounds for this more general problem as well.

We also show how to efficiently compute the L_p norms ($p \in [1,2]$) of vectors in the sliding window model using our techniques. Using our algorithm, 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, maintaining hash tables, and maintaining statistics and aggregates such as sum and average. Our techniques are simple and easy to implement. We expect that it will be an attractive choice of implementation for streaming applications.

1.1. Motivation, model, and related work. Several applications naturally generate data streams as opposed to data sets. 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. For example, Cortes et al. [2] report working with AT&T long distance call records, consisting of 300 million records per day for 100 million customers. A second application is *network traffic engineering*, in which information about current network performance—latency, bandwidth, etc.—is generated online and is used to monitor and adjust network performance dynamically [7, 16]. In this application, it is generally both impractical and unnecessary to process anything but the most recent data.

There are other traditional and emerging applications in which data streams play an important and natural role, e.g., web tracking and personalization (where the data streams are web log entries or click-streams), medical monitoring (vital signs, treatments, and other measurements), sensor databases, and financial monitoring, to name but a few. There are also applications in which traditional (nonstreaming) 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 [12, 11]. 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 along these lines occurs when 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 results, only one pass on the data is possible [3].

In most of these 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 every instant; each data element expires after exactly N time steps; and the portion of data that is relevant to gathering statistics or answering queries is the set of the last N elements to arrive. The sliding window refers to the window of active data elements at a given time instant.

Previous work [1, 5, 13] 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 [14, 10]. While Jagadish et al. [14] address the off-line version of computing optimal histograms, Guha and Koudas [10] provide a technique for maintaining near optimal time-based histograms in an on-line fashion 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 of the data elements seen so far are relevant. Recent work by Gilbert et al. [8] considers, among other things, the problem of maintaining *aged aggregates* over data streams. For a data stream $\ldots, a^{(-2)}, a^{(-1)}, a^{(0)}$, where $a^{(0)}$ is the most recently seen data element, the λ-*aging* aggregate is defined as $\lambda a^{(0)} + \lambda(1-\lambda)a^{(-1)} + \lambda(1-\lambda)^2 a^{(-2)} + \cdots$. Aged aggregate queries tend to get asked in the context of telecommunications data. While aging is one technique to discount for the staleness of certain data elements, we believe that the sliding window model is also important since, for most applications, one is not interested in gathering statistics over outdated data. For instance, in network management, depending upon the specific application, we may not want data that is a month old or a year old to affect our decisions. 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 has been no previous work that addresses these problems for the sliding window model.

1.2. Summary of results.

We focus completely 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 1 (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 of memory. (Note that we measure space complexity in terms of the number of bits rather than the number of memory words.) 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.

It is interesting to observe why naïve schemes do not suffice for producing approximate answers with low memory requirement. For instance, consider the scheme in which we maintain a simple counter which is incremented upon the arrival of a data element, which is 1. The problem is that an old data element expires at every time instant, but we have no way of knowing whether that was a 0 or 1 and whether we should decrement the counter. It is also natural to consider random sampling. Just maintaining a sample of the window elements will fail in the case where the 1's are relatively sparse.

Another approach is to maintain histograms. While this is the approach that we follow, we argue that the known histogram techniques will not work. A histogram technique is characterized by the policy used to maintain the bucket boundaries. We would like to build time-based histograms in which every bucket represents a contiguous time interval and maintains the number of 1's that arrived in that interval. As with all histogram techniques, when a query is presented, we may have to interpolate in some bucket to estimate the answer because a proper subset of the buckets' elements may have expired. Let us consider some schemes of bucketizing and see why they will not work. The first scheme that we consider is that of dividing into k equiwidth buckets. The problem is that the distribution of 1's in the buckets may be nonuniform. We will incur large error when the interpolation takes place in buckets with a majority of the 1's. This suggests another scheme, in which we use buckets of nonuniform width, so as to ensure that each bucket has a near-uniform number of 1's. The problem is that the total number of 1's in the sliding window could change dramatically with time, and the current buckets may turn out to have more or less than their fair share of 1's as the window slides forward. Our solution is a form of a histogram which avoids these problems by using a set of well-structured and nonuniform bucket sizes.

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 a $1+\epsilon$ 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 $O(\log N)$ 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 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; this is referred to as the Sum problem. We provide matching upper and lower bounds on the memory usage for this general problem as well.

In section 5, we show how our schemes extend to a model which is more suited for real-life applications and also explore some ideas for reducing the memory requirements.

In section 6, we show that we can use our techniques along with the sketching techniques of [13] to efficiently maintain the L_p ($p \in [1,2]$) norms of vectors in the sliding window model.

Finally, section 7 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. This serves to illustrate the usefulness of focusing on the BasicCounting problem. We also discuss upper and lower bounds for other problems such as maintaining min/max, distinct values estimation, and maintaining sum in the presence of positive and negative values.

2. Algorithm for BasicCounting.

Our approach toward 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 the counter value of the current arrival. As mentioned earlier, we concentrate on the 1's in the data stream. When we refer to the kth 1, we mean the kth most recent 1 encountered in the data stream.

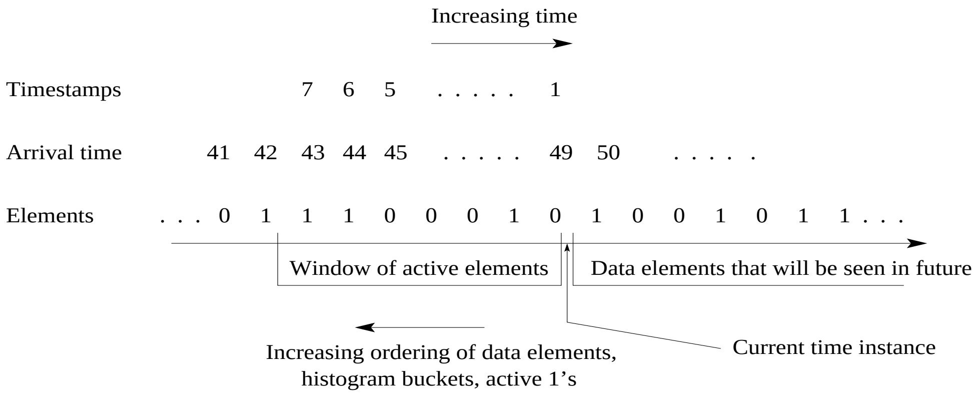

Fig. 1. *An illustration for the notation and conventions followed.*

For an illustration of this notation, consider the situation presented in Figure 1. The current time instant is 49, and the most recent arrival is a zero. The element with arrival time 48 is the most recent 1 and has timestamp 2 since it is the second most recent arrival in the current window. The element with arrival time 44 is the second most recent 1 and has timestamp 6.

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*). For example, in our figure, a bucket with timestamp 2 and size 2 represents a bucket that contains the two most recent 1's with timestamps 2 and 6. Note that the timestamp of a bucket increases as new elements arrive. 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 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, and so we estimate it to be $C/2$. We obtain the following fact.

FACT 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 do the same thing as before except that the last bucket is the bucket with the largest timestamp less than S.

2.1. The approximation scheme. We now define the EHs 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 1, the absolute error in the estimate is $C/2$, where C is the size of the last bucket. Let the buckets be numbered from right to left with the most recent bucket being numbered 1. If C_i is the size of the ith 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 the total count. Note that m is the index of the last bucket. Thus the relative estimation error is 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 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 $\sum_{i=1}^{m} C_i \geq N'$. In order to satisfy this and Invariant 1 with as few buckets as possible, we maintain buckets with exponentially increasing sizes so as to satisfy the following second invariant.

INVARIANT 2. *At all times, the bucket sizes are nondecreasing, 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 $m' \leq m$ and $m' \leq \log \frac{2N}{k} + 1$. For every bucket size other than the size of the last bucket, there are at most $\frac{k}{2} + 1$ and at least $\frac{k}{2}$ buckets of that size.*

Let $C_j = 2^r$ be the size of the jth bucket. If Invariant 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 is satisfied, then Invariant 1 is automatically satisfied. If we maintain Invariant 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 the 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, we are guaranteed the desired relative error and memory bounds.

The query time for the 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; otherwise, 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 $\frac{k}{2}+1$, leading to a cascade of such mergers.) Update the counter Last if the last bucket is the result of a new merger.

Example 1. We illustrate the execution of the algorithm for 10 steps, where, at each step, the new data element is 1. The numbers indicate the bucket sizes from left to right, and we assume that $\frac{k}{2}=1$.

32, 32, 16, 8, 8, 4, 2, 1
32, 32, 16, 8, 8, 4, 4, 2, 1, 1 (new 1 arrived)
32, 32, 16, 8, 8, 4, 4, 2, 1, 1, 1 (new 1 arrived)
32, 32, 16, 8, 8, 4, 4, 2, 2, 1 (merged the older 1's)
32, 32, 16, 8, 8, 4, 4, 2, 2, 1, 1 (new 1 arrived)
32, 32, 16, 8, 8, 4, 4, 2, 2, 1, 1, 1 (new 1 arrived)
32, 32, 16, 8, 8, 4, 4, 2, 2, 2, 1 (merged the older 1's)
32, 32, 16, 8, 8, 4, 4, 4, 2, 1 (merged the older 2's)
32, 32, 16, 8, 8, 8, 4, 2, 1 (merged the older 4's)
32, 32, 16, 16, 8, 4, 2, 1 (merged the older 8's)

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 more recent of the two buckets, i.e., the timestamp of the bucket that is to the right. 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. It is easy to see that the above algorithm maintains Invariant 2. We obtain the following theorem.

Theorem 1. *The EH algorithm maintains a data structure which can give an estimate for the* BasicCounting *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 2. *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 $k \le 4\sqrt{N}$ 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 1. *For $k/4 \le B \le N$, 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 ith block of size 2^iB, and subdivide it into B contiguous subblocks of size 2^i. For each block, we choose $\frac{k}{4}$ subblocks 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.

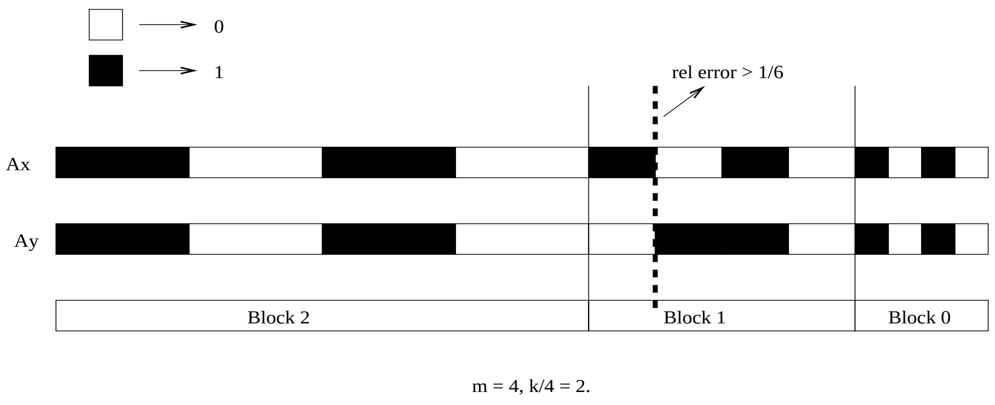

FIG. 2. *A pair of arrangements that should be differentiated by any deterministic algorithm with relative error less than* 1/8.

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 ith number in the sequence specifies the choice of the $k/4$ subblocks from the ith 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 $k/4$ subblocks in the dth block. Number the subblocks within block d from right to left, and let h be the highest numbered subblock that is chosen for one of the arrangements (say, A_x) but not for the other (A_y). Consider the time instant when this subblock h expires. At that instant, the number of active subblocks in block d for arrangement A_x is c, where $c+1 \leq k/4$, while the number of active subblocks 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 2, 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.

As a reminder, a *Las Vegas algorithm* is a randomized algorithm that always produces the correct answer, although the running time of the algorithm may vary with the different random choices that the algorithm makes. On the other hand, a *Monte Carlo algorithm* is a randomized algorithm that sometimes produces an incorrect solution. We obtain the following lower bounds for these two classes of algorithms.

THEOREM 3. *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.*

Proof. Define an algorithm A to be ϵ-correct for an input instance I if the value returned by A on input I has relative error less than ϵ. The Yao minimax principle [15] implies that the expected space complexity of the optimal ϵ-correct deterministic algorithm for an arbitrarily chosen input distribution $\mathbf{p}$ is a lower bound on the expected space complexity of the optimal ϵ-correct Las Vegas randomized algorithm. Consider the uniform distribution over the input arrangements in Lemma 1. Then any deterministic algorithm that is ϵ-correct for all of these instances must differentiate between any two distinct arrangements. As a result, the expected space complexity of an optimal deterministic algorithm on this distribution is at least equal to the optimal coding length for the probability distribution. Since the coding length is at least equal to the entropy of the distribution, we get the same lower bound (logarithm of the number of instances) as in the case of a deterministic algorithm. This proves the generalization of Theorem 2 to Las Vegas randomized algorithms. □

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

Proof. We use the analogous version of Yao's minimax principle for Monte Carlo randomized algorithms [15] to establish the lower bound for Monte Carlo algorithms. Consider a deterministic algorithm that is ϵ-correct with probability at least $1 - \delta$ for some $\delta < \frac{1}{2}$. As before, the input distribution $\mathbf{p}$ that we consider is the uniform distribution over all of the arrangements defined in Lemma 1. Since the deterministic algorithm is ϵ-correct with probability at least $1-\delta$, it is ϵ-correct for at least a $1-\delta$ fraction of the inputs. Thus, by arguments similar to those in the previous theorem, we get the same lower bound except for an additive loss of $\log(1-\delta)$ and a multiplicative loss of $\frac{1}{4}$. Asymptotically, the lower bound does not change. □

4. Beyond 0's and 1's.

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

PROBLEM 2 (SUM). *Given a stream of data elements that are positive integers in the range $[0 \dots R]$, maintain at every time instant the sum of the last N elements.*

One obvious way to solve the above problem would be to separately maintain a sliding window sum for each of the $\log R$ bit positions using an EH from section 2. As before, let $k = \lceil \frac{1}{\epsilon} \rceil$. The memory requirement for this approach would be $O(k \log^2 N \log R)$ bits. Next, we present a technique that has a smaller space requirement.

We assume that $\log R = o(N)$. This is a realistic assumption which simplifies our calculations. We generalize the 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 required only to be nondecreasing. While earlier there could be at most N active 1's, now there could be as many as NR. The results 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

$$\left(\frac{k}{2}+1\right)\left(\log\left(\frac{2NR}{k}+1\right)+1\right)(\log N + \log(\log N + \log R))$$
$$= O\left(\frac{1}{\epsilon}(\log N + \log R)(\log N)\right).$$

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 $j \leq \log(\frac{2NR}{k}+1)$, let $k_0, k_1, \dots, k_j$ 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, we have $l \leq k_i \leq l+1$ for $i < j$ and $1 \leq k_j \leq l+1$, where $l = \frac{k}{2} = \frac{\lceil \frac{1}{\epsilon} \rceil}{2} \geq 1$. Given $l \geq 1$ and S, a sequence $k_0, k_1, \dots, k_j$ 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 2. *The l-canonical representation of any positive number S is unique.*

Proof. We give a proof by contradiction. Assume that $\mathbf{k} = (k_0, k_1, \dots, k_j)$ and $\mathbf{k}' = (k'_0, k'_1, \dots, k'_{j'})$ are two distinct l-canonical representations of S. Without loss of generality, assume that $j \leq j'$. Let d be the smallest index where the sequences differ. We have $d \leq j$ since it cannot happen that they agree on all of 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.

Case 1 ($d < j$). Since $l \leq k_d, k'_d \leq l+1$, 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 integer $c \geq 0$, which is a contradiction since $|\sum_{i=0}^{j} k_i 2^i - \sum_{i=0}^{j'} k'_i 2^i| = 0$.

Case 2 ($d = j$). 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 $k_j \leq k'_j$ since otherwise $\mathbf{k}'$ will have a strictly greater sum. Thus $k_j > k'_j$, and $k_j \leq l+1$. Since k'_j is not the last index, we have $k'_j \geq l$. Therefore, $|k'_j - k_j| \leq 1$, which implies $|\sum_{i=0}^{j} k_i 2^i - \sum_{i=0}^{j} k'_i 2^i| \leq 2^j$. However, $\sum_{i \geq j+1} 2^i \geq k'_i 2^{j+1}$, which gives a contradiction. $\square$

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 of 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 a 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 the 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 the 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 the 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 5. *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 need only 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 ith 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. Timestamps.

In our model (given in section 2), we have assumed that data items arrive at regular time intervals and arrival time increases by one with every new data item that we have seen. However, in most real-life applications, this is not the case, and arrival rates of data items may be bursty. Moreover, we would like to define the sliding window based on real time. In other words, we may want to compute statistics based on the data items that arrived over the last hour, day, etc. It is easy to see that our algorithm can be easily adapted to do this by defining the arrival time based on *real time*; i.e., the arrival time increases by one with every clock-tick.[1] We define N (size of the sliding window) as the number of clock-ticks in the interval over which we want our sliding window to work. For example, N is 3600 if we want statistics based on the last hour, and clock-ticks occur every second. Note that the algorithm does no work except when it sees a data item, and hence it need not do anything during the clock-ticks for which no data items arrive. The invariants are automatically maintained during this period, and the algorithm never uses any extra space during this time. Hence it need not bother to delete the expired buckets until a new data item arrives since its memory requirement does not change. The memory requirement of the algorithm is $O(\frac{1}{\epsilon}(\log N)(\log N + \log R))$, where the second term $(\log N + \log R)$ is the logarithm of the maximum sum (NR) that can occur over N clock-ticks. Thus, if we are guaranteed that much less than N data items arrive over any sliding window, then the memory requirement would be less. This may happen for bursty arrival rates. In other words, our algorithm adapts its memory requirements with the amount of data that we observe.

5.1. Approximate timestamps.

The EHs developed in section 2 and section 4 have a memory requirement of $O(\log N)$ for every bucket of the histogram. The timestamp that we maintain with each bucket requires $\log N$ bits and dominates the memory requirement of every bucket. We now explore the idea of maintaining a coarser timestamp with every bucket which requires only $\log \log N$ bits of memory and reduces the memory requirement for the EH from $O(\frac{1}{\epsilon}(\log^2 N))$ to $O(\frac{1}{\epsilon}(\log N)(\log \log N))$. In the case of generalized EH, the memory requirement drops to $O(\frac{1}{\epsilon}(\log N + \log R)(\log(\log N + \log R)))$. The effect of maintaining a coarser

[1]Equal length intervals, into which we partition time, that are assumed small enough so that no two data items are observed within a single interval.

timestamp is that, instead of answering the query over the last N data elements (sliding window size), we may answer the query over the last N/c elements, where $1 \leq c \leq 2$. In other words, we are approximating the window size to within a factor of 2. Note that this does not contradict the lower bound presented before. We no longer guarantee that the answer that we provide has relative error at most ϵ as compared to the correct answer over the last N elements. Instead, we guarantee that the answer will have error at most ϵ as compared to the correct answer over the last N/c data elements, where $1 \leq c \leq 2$. The factor 2 can be further improved to $1+\tau$ for $0 < \tau \leq 1$, and the memory requirement for the timestamp is $\log\log N + \log(\frac{1}{\tau})$. The generalization is obvious, and we explain the idea below for a factor 2 approximation.

We will explain the idea in terms of timestamps. However, as mentioned in section 2, we do not explicitly maintain the timestamp for every bucket and instead maintain the arrival time of the most recent (rightmost) element and calculate the timestamp using the arrival time of the current element. The idea translates to maintaining coarser arrival times. In sections 2 and 4, we maintained the exact timestamp with every bucket. Instead, here we maintain the timestamp to the closest power of 2. Thus, if the timestamp is t, where $2^{l-1} < t \leq 2^l$ $(1 \leq l \leq \lceil \log N \rceil)$, we maintain the timestamp as 2^l. In other words, we approximate the timestamp to the closest power of 2 greater than the timestamp. Since the timestamps now take $\log N$ distinct values, they can be stored using $\log\log N$ bits. The effect of this approximation is as follows: At any time instance, a bucket is active iff its timestamp is less than N. Any bucket whose exact timestamp is less than $N/2$ will still be considered active since the timestamp will be approximated to a value less than $N/2$. On the other hand, buckets whose timestamps are greater than $N/2$ may be wrongly considered inactive and hence deleted as their timestamp will be approximated to a value no less than N. Thus, in the worst case, we are answering the query over the last $N/2$ elements instead of N. If, instead, we approximate the timestamp to the closest power of 2 *less* than the exact timestamp, we get that we will be answering the query over the last cN elements, where $1 \leq c \leq 2$.

6. Computing L_p norms for vectors.

We now extend the EH technique and combine it with the sketching technique from Indyk [13] 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. $f(B_1 + B_2) \geq f(B_1) + f(B_2)$, where $B_1 + B_2$ denotes 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. For now, we assume that the sketches provide the exact value of the function f. We will shortly relax this requirement and show that our technique works even if the sketches provide only an approximation to the actual function value.

INVARIANT 3. $f(B_{n+1}) \leq \frac{C_f}{k} \sum_{i=1}^{n} f(B_i)$.

INVARIANT 4. $f(B_{n+2}) + f(B_{n+1}) > \frac{1}{k} \sum_{i=1}^{n} f(B_i)$.

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 which are active. Let B_x be the part (suffix) of the leftmost bucket that is active and was ignored (i.e., did not contribute to the estimate). Let B_y be the concatenation of all of 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 $\boldsymbol{E_r}$ is $\frac{f(B_x+B_y)-f(B_y)}{f(B_x+B_y)} > 0$ by P1 and P3. Also, we have

$$\begin{aligned}
\boldsymbol{E_r} &\leq \frac{f(B_x + B_y) - f(B_y)}{f(B_y)} && \text{(P1, P3)} \\
&\leq \frac{C_f(f(B_x) + f(B_y)) - f(B_y)}{f(B_y)} && \text{(P4)} \\
&= \frac{C_f f(B_x)}{f(B_y)} + C_f - 1 \\
&\leq \frac{C_f f(B_{n+1})}{\sum_{i=1}^{n} f(B_i)} + C_f - 1 && \text{(P1, P3)} \\
&\leq \frac{C_f^2}{k} + C_f - 1 && \text{(Invariant 3).}
\end{aligned}$$

Observation 2. Invariant 4 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 timestamp and the sketches for all of 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 $0 \leq \boldsymbol{E_r} \leq \frac{C_f^2}{k} + C_f - 1$ using $O(k \log N(\log N + g_f(N)))$ memory bits. We can maintain the invariants along with the timestamp and the sketches as new data elements are added. The algorithm to do this is very similar to that for the 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 4 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 Invariants 3 and 4. Adding a new bucket does not violate Invariant 3, as we increase only the size of the suffix. Whenever Invariant 4 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 $f(B'_{n+1}) \leq C_f(f(B_{n+2})+f(B_{n+1})) \leq \frac{C_f}{k} \sum_{i=1}^{n} f(B_i)$, and hence Invariant 3 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 (i.e., $O(k \log N)$). 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 6. *A function f with properties* P1–P5 *can be estimated over sliding windows with relative error* $0 \leq \boldsymbol{E_r} \leq \frac{C_f^2}{k} + C_f - 1$ *using* $O(k \log N(\log N + g_f(N)))$ *bits of memory.*

So far we have assumed that the sketches compute the function value f exactly. Instead, in most sketching techniques, the sketches provide a $1+\hat{\epsilon}$ approximation $s(B)$ of the actual function value $f(B)$, i.e., $(1-\hat{\epsilon})f(B) \leq s(B) \leq (1+\hat{\epsilon})f(B)$. In that case, we maintain buckets with Invariants 3 and 4, replacing f with s. Invariant 4, property P2, and the fact that $s(B) \leq (1+\hat{\epsilon})f(B)$ guarantee that the number of buckets will be $O(k \log N)$ as before. We will now analyze the effect of approximation due to sketches on the error.

Maintaining the invariant $s(B_{n+1}) \leq \frac{C_f}{k} \sum_{i=1}^{n} s(B_i)$ guarantees that $f(B_{n+1}) \leq (1+\hat{\epsilon})^2 \frac{C_f}{k} \sum_{i=1}^{n} f(B_i)$. We will use the same technique (please refer to Observation 1) to estimate the function f using sketch estimates s instead of f. Thus, instead of providing the exact answer $f(B_x + B_y)$, we provide the estimate as $s(B_y)$. The relative error $\boldsymbol{E_r}$ is $\frac{f(B_x+B_y)-s(B_y)}{f(B_x+B_y)}$. We have

$$\begin{aligned}
\boldsymbol{E_r} &= \frac{f(B_x+B_y)-s(B_y)}{f(B_x+B_y)} \\
&\leq \frac{f(B_x+B_y)-f(B_y)+f(B_y)-s(B_y)}{f(B_y)} && \text{(P1, P3)} \\
&\leq \frac{f(B_x+B_y)-f(B_y)}{f(B_y)} + \frac{f(B_y)-s(B_y)}{f(B_y)} \\
&\leq \frac{C_f f(B_{n+1})}{\sum_{i=1}^{n} f(B_i)} + C_f - 1 + \hat{\epsilon} && \text{(proved earlier)} \\
&\leq (1+\hat{\epsilon})^2 \frac{C_f^2}{k} + C_f - 1 + \hat{\epsilon}.
\end{aligned}$$

This gives the following theorem.

THEOREM 7. *A function f with properties* P1–P5 *can be estimated over sliding windows with relative error* $0 \leq \boldsymbol{E_r} \leq (1+\hat{\epsilon})^2 \frac{C_f^2}{k} + C_f - 1 + \hat{\epsilon}$ *using* $O(k \log N(\log N + g_f(N)))$ *bits of memory, where $\hat{\epsilon}$ is the bound on relative error of the sketches.*

If $\hat{\epsilon}$ can be made arbitrarily close to 0, keeping the space requirement for the sketches (i.e., $g_f(|B|)$) small, we can get the same error as in the previous theorem by increasing k by a small constant factor.

6.1. $\boldsymbol{L_p}$ 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 [13] in which the jth data element is a pair (i_j, a_j), where $i_j \in [d] = \{0 \ldots d-1\}$ and $a_j \in \{0 \ldots M\}$ represents an increment to the i_jth dimension of an underlying vector. Every window B represents a vector, and its $L_p(B)$ norm is given by $L_p(B) = (\sum_{i \in [d]} |s_i|^p)^{1/p}$, where $s_i = \sum_{i_j = i, j \in B} a_j$ is the sum of unexpired increments to the ith dimension.

Note that the case in which $p = 1$ is the same as the SUM problem. If the dimension d of the underlying vector is small, one obvious way to maintain the L_p norm is to maintain the approximate sum for each dimension using the techniques in section 4. It would require $O(\frac{1}{\epsilon}(\log N + \log M)(\log N)d)$ bits of memory and give a relative error of ϵ. However, for high dimensional vectors, we propose the use of sketches. We denote $(L_p(B))^p$ by f_p and estimate f_p for $p \in [1,2]$. The function f_p clearly admits properties P1–P4, assuming $M \leq N^{O(1)}$. 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 [13].) The sketches for computing the function are not exact; instead, they provide an approximation with relative error less than $\hat{\epsilon}$ with probability $1-\delta$.

From Theorem 7, we have that, under the restricted model, we can compute f_p (i.e., $(L_p(B))^p$) with relative error at most $(1+\hat{\epsilon})^2\frac{4}{k}+1+\hat{\epsilon}$ using $O(k \log N(\log N + \log M \log(1/\delta)/\hat{\epsilon}^2))$ bits of memory. As mentioned before, an additional $O(\log M \log(d/\delta)$ $\log(1/\delta)/\hat{\epsilon}^2)$ bits of memory, which are common to all the sketches, are required. The estimate is approximately correct with high probability. Note that computing $(L_p(B))^p$ with a small relative error translates to computing $L_p(B)$ with a small relative error.

6.2. 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 asymptotically. The L_p norm for $p = 0$ is defined as the distinct value problem, and we deal with this problem in section 7.

7. 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, we will use an EH to play the role of a counter. A counter would have required $\Omega(\log N)$ bits of memory, while an EH requires $O(\frac{1}{\epsilon}\log^2 N)$ bits of memory and maintains the count with $1+\epsilon$ error.

7.1. Hash tables. This is the simplest case. Every data element gets hashed to a bucket, and the goal is to maintain the count of elements in each hash 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.

7.2. Sums and averages. In section 4, we showed how to maintain the sum of positive integer data elements using the generalized version of the EHs. This requires $O(\frac{1}{\epsilon}\log N(\log R + \log N))$ bits of memory. Since maintaining the sum would require $\log N + \log R$ bits, the multiplicative overhead is $O(\frac{1}{\epsilon}\log N)$. Maintaining averages is similar. The average of the most recent N data elements is just the sum divided by N. If the items are inserted irregularly in real time and we want the average value to represent the sum divided by the number of insertions in the last N clock-ticks, an easy solution would be to use a second instance of the EH that maintains the count (number of insertions) approximately. Since both the sum and count have small relative error, so will their quotient.

7.3. Histograms. Given the bucket boundaries in a histogram, we can maintain the sum, average, and other statistics corresponding to each bucket using generalized EH. Finding the optimal bucket boundaries to optimize the memory requirement is an orthogonal problem. Also, equiwidth histograms are a natural choice of histograms for which the bucket boundaries are fixed. Note that, unlike the histograms discussed in [10], these are not time-based histograms but instead could be based on any attribute of the data.

7.4. Min and max. We prove a lower bound for the memory requirement of an algorithm that maintains min or max over a sliding window. While we argue the lower bound for the case of min, the argument for max is similar. The lower bound is based on a counting argument like the one used to prove the lower bound for BasicCounting in Lemma 1. 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 $\binom{N+R-1}{N}$. Consider an algorithm that has seen one of these arrangements. We claim that any deterministic algorithm that gives the correct answer at every time instance henceforth must differentiate between any two such arrangements. To this end, we assume that we will present the algorithm with a sequence consisting of the highest number in the set, similarly to how we present the algorithm with a sequence of 0's in Lemma 1. Since the numbers presented to the algorithm were nondecreasing, at any time instance, the correct answer is the value of the oldest or least recent element which will expire in the next step. As a result, for every pair of arrangements, there will be a time when their oldest elements differ, and hence they have different correct answers. This proves our claim and establishes a lower bound on the number of memory bits required, which is $\log\binom{N+R-1}{N} \geq N\log(R/N)$. This lower bound is also valid for any randomized algorithm by arguments similar to the one in section 3. If $R = \mathit{poly}(N)$, 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 do the following: Keep the subsequence of data elements in which the leftmost item is the current minimum and the right neighbor or any element (in the subsequence) is the minimum of the elements to the right of the element in the stream. Such a subsequence can be maintained as a list of pairs (value, timestamp), where the list satisfies the property that both the value and the timestamp are strictly increasing. This scheme has a worst-case space requirement of $O(N\log R)$ bits. If the data elements arrive in a random order, then the list that we would maintain is analogous to the right spine of a "treap" where the timestamps are fully ordered and the values of the data elements are heap-ordered. In that case, the expected length of the list is $O(\log N)$, and the space complexity is given by $O(\log N \log R)$.

7.5. Distinct values. It 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*[2] 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 the BasicCounting problem applies to the distinct value problem. Given an instance of the 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 1's. This reduction shows that the lower bounds for the BasicCounting problem apply to the distinct value problem.

Consider the problem of estimating the number of distinct values over sliding

[2]The technique assumes perfect hash functions. However, it suffices to use hash functions which do not have complete independence.

windows in the presence of deletions. We prove a space lower bound of $\Omega(N)$ bits, where N is the window size. Every data element consists of a value and a bit that indicates if the value is being "inserted" or "deleted." Consider the last N elements that form the current window. These define a collection of values and multiplicities. The multiplicity of a value is the number of times it is inserted in the current window minus the number of times it is deleted from the current window. It is possible for a value to have negative multiplicity since it may have been deleted more times than it was inserted. The number of distinct values is the number of values that have multiplicity greater than zero. Our lower bound holds even if we define the number of distinct values as those with nonzero multiplicity. Given this model we claim the following.

Claim 1. *An algorithm that estimates the number of distinct values within a factor 2 of the correct answer, over sliding windows and in the presence of deletions, requires $\Omega(N)$ bits of space, where N is the size of the sliding window.*

Proof. Consider an algorithm A that estimates the number of distinct values to within a factor of 2, over sliding windows and in the presence of deletions. Given any arbitrary bit vector $\boldsymbol{X} = \{x_1, x_2, \ldots, x_N\}$, we present the algorithm with the following input. Every bit x_i is mapped to a value y_i as follows: If x_i is 1, then y_i is set to i. Otherwise, y_i is set to 0. These values are input to the algorithm in the order y_1 to y_N along with the additional bit that represents that these values are being inserted. After the N values have been inserted, let S be the state of the algorithm. We claim that we can recover the last $N/2$ bits of the vector $\boldsymbol{X}$ (i.e., $\{x_{N/2+1}, x_{N/2+2}, \ldots, x_N\}$) using the state S of the algorithm. This proves that the information content of the state S is at least $N/2$ bits, and hence it would require $\Omega(N)$ bits of space. Given the state S, we recover the last bit (x_N) as follows: We insert $N-1$ elements with value 0. The current window now contains $N-1$ inserts of value 0 and an insert of value y_N, which, depending upon the value of x_N, is either 0 or N. The correct answer (i.e., number of distinct values) in that case is either 1 or 2, depending upon whether x_N is 0 or 1. Since the algorithm A estimates to within a factor of 2, it will distinguish between the two cases, and we can infer the last bit x_N. If the algorithm estimates the number of distinct values to be less than 2, then $x_N = 0$; otherwise, $x_N = 1$. Having inferred the last bit x_N, we can infer the previous bit x_{N-1} as follows: We "rewind" to the state S (state after inserting y_1 to y_N). In other words, we run the algorithm from state S again, by storing the state S. We input an element with value equal to y_N and bit set to delete, followed by $N-3$ elements with value equal to 0 and bit set to insert. In other words, we are deleting the value y_N that was inserted last. The current window now consists of $N-3$ inserts of value 0 and an insert of value y_{N-1}, which, depending on x_{N-1}, is either 0 or $N-1$. Note that the current window also contains a pair of elements that insert and delete the value y_N. By similar arguments as before, we can now infer the bit x_{N-1} since the algorithm A gives a factor 2 approximation. We can proceed this way to infer the last $N/2$ bits in the order $x_N, x_{N-1}, \ldots, x_{N/2+1}$. To infer the x_{N-i}th bit (having inferred $x_{N-i+1}, \ldots, x_N$), start with the state S again. Input elements with values $y_{N-i+1}, y_{N-i+2}, \ldots, y_N$ and bit set to delete, followed by $N-2i-1$ inserts of value 0. The current window will then contain $N-2i-1$ inserts of value 0, an insert of value y_{N-i}, and i pairs of inserts and deletions of values $y_{N-i+1}, y_{N-i+2}, \ldots, y_N$. Again, we can infer the bit x_{N-i}. We can do this for $i < N/2$.

The argument above proves that the state S essentially encodes the last $N/2$ bits and probably more, proving the lower bound for the space requirement of S. ∎

7.6. General sum. Consider the problem of maintaining the sum of the last N integers when the integers could be positive or negative. We prove that, even if we restrict the set of integers to $\{1, 0, -1\}$, to approximate the sum within a constant factor requires $\Omega(N)$ bits of memory. Moreover, it is easy to maintain the sum by storing the last N integers, which requires $O(N)$ bits of memory. We assume that the storage required for every integer is a constant independent of the window size N. This proves that the complexity of the problem in the general case (i.e., allowing positive and negative integers) is $\Theta(N)$. We now argue the lower bound of $\Omega(N)$. Consider an algorithm A that provides a constant factor approximation to the problem of maintaining the general sum. Given a bit vector of size $N/2$, we present the algorithm A with the pair $(-1, 1)$ for every 1 in the bit vector and the pair $(1, -1)$ for every 0. Consider the state (i.e., time instant) after we have presented all of the $N/2$ pairs to the algorithm. We claim that we can completely recover the original bit vector by presenting a sequence of 0's henceforth and querying the algorithm on every odd time instant. If the current time instant is T (after having presented the $N/2$ pairs), then it is easy to see that the correct answer at time instant $T + 2i - 1$ $(1 \leq i \leq N/2)$ is 1 iff the ith bit was 1 and -1 iff the ith bit was 0. Since the algorithm A gives a constant factor approximation, its estimate would be positive if the correct answer is 1 and negative if the correct answer was -1. Since the state of the algorithm after feeding the $N/2$ pairs enables us to recover the bit vector exactly for any arbitrary bit vector, it must be using at least $N/2$ bits of memory. This proves the lower bound. We can state the following theorem.

THEOREM 8. *The space complexity of any algorithm that gives a constant factor approximation, at every instant, to the problem of maintaining the sum of last N integers, which appear as a stream of data elements and could be positive or negative, is equal to $\Theta(N)$.*

8. Conclusion.

In conclusion, this paper takes the first step toward computing over data streams in the sliding window model. We consider the problem of maintaining statistics over sliding windows and provide upper and lower space bounds for various problems. It remains to consider problems like maintaining other statistics (for instance, variance), clustering (maintaining k-medians), etc. in the sliding window model.

Acknowledgments. The authors would like to thank Yossi Matias for suggesting the idea of maintaining coarser timestamps presented in section 5. They would also like to thank the anonymous referees for their suggestions in improving the presentation.

REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy, *The space complexity of approximating the frequency moments*, in Proceedings of the 28th Annual ACM Symposium on Theory of Computing, ACM, New York, 1996, pp. 20–29.

[2] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers, *Hancock: A language for extracting signatures from data streams*, in Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, New York, 2000, pp. 9–17.

[3] S. Chaudhuri, R. Motwani, and V. R. Narasayya, *On random sampling over joins*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM, New York, 1999, pp. 263–274.

[4] M. Fang, H. Garcia-Molina, R. Motwani, N. Shivakumar, and J.D. Ullman, *Computing iceberg queries efficiently*, in Proceedings of the 24th International Conference on Very Large Data Bases, New York, NY, 1998, pp. 299–310.

[5] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan, *An approximate L1-difference algorithm for massive data streams*, in Proceedings of the 40th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1999, pp. 501–511.

[6] P. Flajolet and G. Martin, *Probabilistic counting*, in Proceedings of the 24th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1983, pp. 76–82.

[7] C. Fraleigh, S. Moon, C. Diot, B. Lyles, and F. Tobagi, *Architecture of a Passive Monitoring System for Backbone IP Networks*, Technical report TR00-ATL-101-801, Sprint Advanced Technology Laboratories, Burlingame, CA, 2000.

[8] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, *Surfing wavelets on streams: One-pass summaries for approximate aggregate queries*, in Proceedings of the 27th International Conference on Very Large Data Bases, Rome, Italy, 2001, pp. 79–88.

[9] S. Guha and N. Koudas, *Approximating a data stream for querying and estimation: Algorithms and performance evaluation*, in Proceedings of the Eighteenth International Conference on Data Engineering, San Jose, CA, 2002, pp. 567–576.

[10] S. Guha and N. Koudas, *Data-streams and histograms*, in Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing, ACM, New York, 2001, pp. 471–475.

[11] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan, *Clustering data streams*, in Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 2000, pp. 359–366.

[12] M. R. Henzinger, P. Raghavan, and S. Rajagopalan, *Computing on Data Streams*, Technical report TR 1998-011, Compaq Systems Research Center, Palo Alto, CA, 1998.

[13] P. Indyk, *Stable distributions, pseudorandom generators, embeddings and data stream computation*, in Proceedings of the 41st IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 2000, pp. 189–197.

[14] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, and T. Suel, *Optimal histograms with quality guarantees*, in Proceedings of the 24th International Conference on Very Large Data Bases, New York, NY, 1998, pp. 275–286.

[15] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, Cambridge, UK, 1995.

[16] Cisco Systems, *Netflow Services and Applications*, White paper, Cisco Systems, San Jose, CA, 2000; also available online from http://www.cisco.com/warp/public/cc/pd/iosw/ioft/neflct/tech/napps_wp.htm.