Space Efficient Streaming Algorithms for the Maximum Error Histogram *

Chiranjeeb Buragohain†
Amazon.com
Seattle, WA 98106
chiran@amazon.com

Nisheeth Shrivastava†
Bell Labs Research, India
nisheeths@lucent.com

Subhash Suri
Dept. of Computer Science
University of California
Santa Barbara, CA 93106, USA
suri@cs.ucsb.edu

Abstract

*We propose new algorithms for constructing maximum error (L_∞) histograms in the data stream model. Our first algorithm (*MIN-MERGE*) achieves the following performance guarantee: using $O(B)$ memory, it constructs a $2B$-bucket histogram whose approximation error is at most the error of the* optimal *B-bucket histogram. Our second algorithm (*MIN-INCREMENT*) achieves a $(1+\varepsilon)$-approximation of a B-bucket histogram using $O(\varepsilon^{-1}B\log U)$ space, where U is the size of the domain for data values. The memory requirements of these algorithms are a significant improvement over the previous best schemes for constructing near-optimal histograms in the data stream model, making them ideal for data summary applications where memory is at a premium, such as* wireless sensor networks*. Our* MIN-INCREMENT *algorithm also extends to the* sliding window *model without any asymptotic increase in space. Finally, using synthetic and real-world data, we show that our algorithms are indeed as space-efficient in practice as their theoretical analysis predicts—compared to previous best algorithms, they require two or more orders of magnitude less memory for the same approximation error.*

1 Introduction

We live in an age of data glut: from telecommunications to financial markets, transactional databases, satellite imaging, surveillance, and environmental monitoring, enormous quantities of data are routinely generated and archived. Mining these large data sets to discover significant trends, anomalies, or answer queries, poses a major challenge for database systems. In many settings, a quickly-computed approximate answer is preferable to a precise answer that may take hours or days to determine. This is especially true for *real-time monitoring* applications where a major trend change or anomaly requires quick response. The following two application scenarios provide a motivating context for our work on space-efficient histogram algorithms.

- Small devices with integrated sensing, computing, and wireless communication capabilities are now widely available, and efforts are underway to deploy networks of these sensor nodes to achieve pervasive computing and monitoring [23]. Due to low cost and small form factor, these devices are highly constrained in their resources: battery power, memory, and communication bandwidth. In particular, the on-board memory for the sensor nodes is limited to a few KBytes [15] and so highly space-efficient algorithms are needed for histogram computation in sensor networks.
- A key query in many data stream management systems [2, 14] such as StatStream [25] is the similarity between two time series, which is typically defined as the L_p distance between the series. The similarity queries, in turn, are the building blocks for more complex queries such as clustering. When dealing with many large time series, a common practice is to work with compressed approximate representations of the series. Piecewise linear summaries or histograms are the preferred tool because of their versatility and computational efficiency [19, 24]. Concurrently computing the histograms for thousands of data streams requires that the histogram algorithm be highly frugal in its space usage.

In database systems, histograms have been a subject of tremendous research in the last few decades [16, 21, 22]. In the last decade, they have also been investigated intensely under the (single pass) *data stream* model. For most error metrics, the best known data stream algorithms currently known are due to Guha and his colleagues [7, 9, 11, 12, 13]. While the cumulative error (the L_2 norm) has been the dominant error metric in histograms, the maximum error (the

*This work was supported in part by the National Science Foundation Grants CCF 0514738 and CNS 0626954.

†This work was done while the authors were in UCSB.

1-4244-0803-2/07/$20.00 ©2007 IEEE.

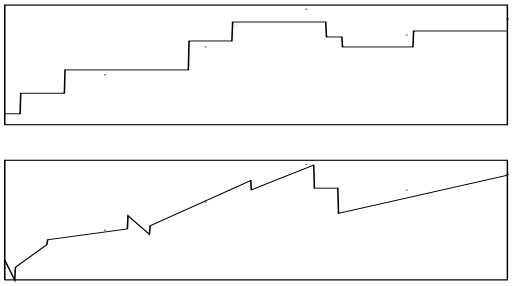


Figure 1. A (piecewise constant) histogram (top) and a piecewise linear histogram (bottom) approximations using 8 buckets.

L_∞ norm) is also used frequently. In real-time monitoring applications, where detection of sudden spikes or anomalies is significant, minimizing the maximum error is a natural objective function, and the focus of our paper.

Computing histograms for the maximum error is both conceptually and algorithmically simpler than the L_2 norm, and this point has been noted before by others, including Guha [7] and Guha, Kim, and Shim [9], who propose simpler and faster streaming algorithms for approximating the histograms under the L_∞ norm. Curiously, however, the *space complexity* (working storage) for the max error histograms has not benefited from these improvements. Specifically, the best space bound for computing a B-bucket $(1+\varepsilon)$ approximation of the histogram under the data stream model is $O(\varepsilon^{-1}B^2 \log n)$ both for the L_2 and the L_∞ error metric [7]. The main result of our paper is to show that the space complexity of the histogram algorithm (in the data stream model) can be improved by a *factor of* B. Since the working storage is severely limited in lightweight computational models such as sensor networks, this is an important, and often a critical, issue in those settings; even for small histograms, say, $B = 10$ or 20, the space savings from our algorithms are substantial.

1.1 Our Contributions

We consider the histogram approximation problem with two different bucket representations—the values inside each bucket are approximated by either a single *value* or a *line* with arbitrary slope. The first approximation is the traditional (serial) histogram; we call the second type of approximation *piecewise-linear* (PWL) histogram [20, 16]. Figure 1 shows examples of the two histograms. The PWL-histogram is computationally a little more expensive, but it can offer better approximation quality, especially for time-series data, which may exhibit rising or falling linear trends.

Following Jagadish et al. [17] we characterize our approximation algorithms by two parameters (α, β) as follows. Suppose the optimal histogram uses B^* buckets and approximates the input data within error E^*. Then, an algorithm is called an (α, β) approximation algorithm if it can construct an approximation to the original data stream with error at most αE^* using at most βB^* buckets. By this definition, the algorithm of Guha et al. [12] gives a $(1+\varepsilon, 1)$ approximation. The main results of our paper can be summarized as follows.

1. We propose an $(1+\varepsilon, 1)$ approximation algorithm, called MIN-INCREMENT, which requires $O(\varepsilon^{-1}B \log U)$ space over a data stream, where U is the size of the domain of input values. This compares with the $O(\varepsilon^{-1}B^2 \log n)$ bound in [12]. Typically, n and U are polynomially related, and so we treat $\log n$ and $\log U$ as comparable; in fact, in stream setting, n is ofter much larger than U.
2. We also propose a second algorithm, called MIN-MERGE, which achieves $(1, 2)$ approximation using just $O(B)$ memory. This algorithm is extremely simple, involves a cute analysis, is worst-case optimal in space usage, and performs extremely well in simulation.
3. We show that both MIN-MERGE and MIN-INCREMENT extend to piecewise linear histograms. In particular, the MIN-MERGE algorithm provides $(1+\varepsilon, 2)$ approximation, and uses $O(\varepsilon^{-1/2}B \log(1/\varepsilon))$ space. The MIN-INCREMENT remains a $(1+\varepsilon, 1)$ approximation algorithm, and uses $O(\varepsilon^{-1}B \log U + \varepsilon^{-3/2} \log(1/\varepsilon) \log U)$ space.
4. The MIN-INCREMENT algorithm also extends to the sliding window model, and it does so without any asymptotic increase in memory. The new contribution here is that the space requirement does not grow with the window size; the previous best scheme [10] needs $\Theta(w)$ memory for w size sliding window.
5. Using synthetic and real-world data, we show that our algorithms are indeed as space-efficient in practice as their theoretical analysis predicts.

1.2 Related Work

There is a long history of approximating curves, time series and data streams by various functions—an interested reader may refer to the recent surveys [1, 20, 16]. Wavelets are a popular tool for approximating time series, and a streaming algorithm for constructing a B-coefficient wavelet approximation, using $O(B + \log n)$ memory, is given by Gilbert et al. [6]. Wavelets give acceptable results for the L_2 error, but can perform quite poorly under the L_∞ norm. In the streaming context, there has been attempts

to remedy this defect by heuristics [18] and more rigorous methods [8].

Most relevant to our work are schemes for constructing histograms with guaranteed error bounds. In [17], Jagadish et al. describe a dynamic-programming based algorithm for computing an optimal histogram for the L_2 error. The dynamic programming framework is quite versatile and, in fact, it can be used to compute optimal histograms for most metrics. The algorithm of [17], however, does not work in the data stream model because it makes multiple passes over the data.

The first streaming histogram approximation algorithm was given by Guha, Koudas and Shim [11] based upon approximate dynamic programming. This work showed that for a very large variety of metrics, $(1+\varepsilon)$-approximations of the optimal histograms can be computed using memory $O(\varepsilon^{-1}B^2 \log n)$. Guha, Shim and Woo [12, 7] showed that in the case of the L_∞ metric, $(1+\varepsilon)$-approximations of the optimal histograms can be computed using memory $O(\varepsilon^{-1}B^2 \log n)$, and *linear* time leveraging the simplicity of the L_∞ metric. This is the best bound currently known for the problem of interest to this paper. Guha [13] has also made a comprehensive study of space-efficiency of optimal and approximate off-line histogram algorithms. In fact, Guha [13, 9] has shown that optimal offline histograms for both L_∞ and L_2 histograms can be computed using linear space.

In [10], Guha et al. propose an algorithm for computing an approximate histogram under the sliding window model, but their focus is on improving the time complexity; the space required by their algorithm is linear in the window size. In [5], Gilbert et al. propose streaming algorithms for approximate histograms, but their focus is on maintaining these histograms under updates to the stream.

2 Streaming Algorithms for L_∞ Histograms

We assume that the input stream is a sequence of values $x_1, x_2, \ldots, x_n$ from an (integer) universe $[1, U]$. Without loss of generality, we assume that the value x_i arrives at time i. We wish to maintain a guaranteed quality histogram approximation of the past stream; in the full stream model, the past includes all the values seen so far, while in the sliding window model, only the last w (window size) values will be relevant. We consider two different variants of the histogram: the traditional serial histogram, where each bucket will be represented by a single value, and a piecewise linear (PWL) histogram, where the data in each bucket is approximated by a linear function (line). In either case, the histogram's reconstruction (approximation) of data value x_i is denoted $\hat{x}_i$. The *quality* of a histogram is measured by its error in approximating the input sequence $x_1, x_2, \ldots, x_n$. The error metric of interest for us is the L_∞ norm, defined as follows:

$$E_\infty = \max_i |x_i - \hat{x}_i| \quad (1)$$

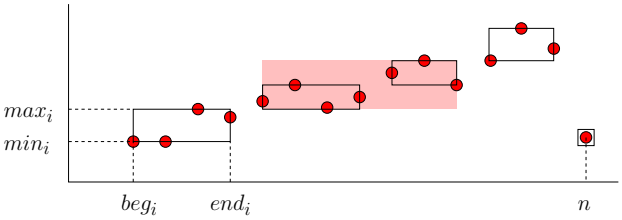


Figure 2. Example operation of the MIN-MERGE algorithm with 4 buckets. The new item at index n is given its own bucket and then buckets s_2 and s_3 (shaded), which will cause least error, are merged.

We first describe the MIN-MERGE algorithm. This algorithm constructs a $2B$-bucket histogram whose L_∞ approximation error is no worse than the error of an optimal B-bucket histogram. While the theoretical approximation quality of this algorithm is not as nice as the $(1+\varepsilon)$ approximation, it has several attractive properties: it is extremely simple yet comes with a worst-case guarantee; the analysis builds on a simple yet elegant *min-merge* property; and its space efficiency is worst-case optimal. One natural way to think about this algorithm is that if we want to achieve optimal maximum error for a data stream using B buckets, then a simple doubling of the bucket allocation allows us to achieve that with a worst-case guarantee, while using only $O(B)$ working space. An offline algorithm with similar approximation guarantee was provided by Jagadish et al. [17] for the L_2 error metric, which gave a $(3, 3)$ approximation. We do not know of any previous streaming histogram algorithm of this type, but we believe that it can be a very useful algorithm in practice due to its low memory usage.

2.1 The MIN-MERGE Algorithm

The MIN-MERGE algorithm dynamically maintains a set of buckets that form a serial histogram of the stream seen so far. Whenever the number of buckets exceeds its budget of $2B$, it *merges* the two *adjacent* buckets that cause the least increase in the error. We prove that this simple algorithm achieves the $(1, 2)$ approximation guarantee.

In MIN-MERGE approximation we maintain a data structure S, which is a set of k buckets $\{s_1, s_2, ..., s_k\}$, where each bucket is a tuple $s_i = \{beg_i, end_i, max_i, min_i\}$. The values beg_i and end_i mark the begin and end index of contiguous data-stream values which are approximated by s_i, and min_i and max_i, respectively, are the minimum and the maximum values in

the bucket s_i (see Fig. 2 for an example). We denote the error in s_i by $err(s_i)$. To minimize the L_∞ error in each bucket, we represent s_i by a single value $(max_i + min_i)/2$, which gives $err(s_i) = (max_i - min_i)/2$. The error of the entire histogram $err(S)$, is the largest error in any of the buckets, namely, $err(S) = \max_i\{err(s_i)\}$.

Algorithm 1 MIN-MERGE(S, v)

1: $S \leftarrow S \cup \{n, n, v, v\}$;
2: **if** $|S| > 2B$ **then**
3: $\quad i \leftarrow$ FINDMIN(S);
4: $\quad$ delete s_i and s_{i+1};
5: $\quad S \leftarrow S \cup$ MERGE(s_i, s_{i+1});
6: **end if**
7: $n \leftarrow n + 1$;

A pseudo-code for the algorithm is shown in Algorithm 1. The basic idea is as follows. We maintain a set of $2B$ buckets. Suppose the next element to arrive is indexed n, and has value v. We create a new bucket $\{n, n, v, v\}$ and insert it into the summary. The number of buckets now exceeds $2B$, so we determine the two *adjacent* buckets whose merger gives the smallest increase in the total error, and merge those two buckets. See Fig. 2 for an example. The function FINDMIN finds the index (i) of the bucket which after merging with its successor bucket gives the minimum increase in total error. The function MERGE takes two adjacent buckets, s_i and s_{i+1}, and merges them into a bucket with value index in range $[beg_i, end_{i+1}]$, and set the max and min values of the new bucket to $\max(max_i, max_{i+1})$ and $\min(min_i, min_{i+1})$, respectively. We now analyze the running time and the space complexity of this algorithm, and also prove that it achieves a $(1, 2)$ approximation.

2.1.1 Space and Time complexity

The MIN-MERGE algorithm maintains at most $2B$ buckets, each represented in constant memory (4 integers), and so its worst-case memory requirement is $O(B)$. The per-item processing time is dominated by the FINDMIN procedure—the remaining operations take constant time. In order to speed up FINDMIN, we pre-compute the errors, call them key_i, caused by merging buckets s_i and s_{i+1}, for each i, and build a min-heap over these $2B - 1$ key values. Thus, FINDMIN can be executed in constant time, because it is available at the root of the heap. After every merge, we need to delete (up to) three keys from the heap and insert two updated keys (merge errors between the new bucket and its neighbors), which takes $O(\log B)$ total time. Therefore, the per-item update time of MIN-MERGE is $O(\log B)$. The size of heap is $O(B)$, and so the total space complexity of the algorithm is $O(B)$.

2.1.2 The Approximation Quality

The key to analyzing MIN-MERGE is the following simple, yet important, property, which we call the *min-merge* property: *a set of buckets S is said to have the min-merge property if any bucket formed by merging two adjacent buckets has error at least $err(S)$.* We now prove that the $err(S)$ of any set S with $2B$ buckets following the min-merge property is at most the optimal error with B buckets.

Lemma 1. *Let S be a set of $2B$ buckets that satisfy the min-merge property. Then, $err(S) \leq err(S_{\rm OPT})$, where $S_{\rm OPT}$ is the optimal B bucket approximation.*

Proof. Any partition $S_{\rm OPT}$ of the stream into B buckets will split at most $B - 1$ buckets of the partition S: that is, at least $B + 1$ buckets of the partition S will remain unsplit. By the pigeonhole principle, there must be some bucket s_i of $S_{\rm OPT}$ containing at least two unsplit buckets of S. By definition, $err(S_{\rm OPT}) \geq err(s_i)$, which in turn is at least as large as the error of the union of those two unsplit buckets. However, by the min-merge property, the error of the union of those two unsplit buckets is at least $err(S)$. This completes the proof that $err(S) \leq err(S_{\rm OPT})$. □

Finally, it only remains to show that the buckets computed by the MIN-MERGE algorithm satisfy the min-merge property. By induction, let us assume that the property holds before inserting the input value v. If the error of merging the last bucket s_{2B} with the new item's bucket b_v is less than $err(S)$, then FindMin will choose the pair (s_{2B}, b_v) to merge. In this case, the $err(S)$ does not increase and so the min-merge remains satisfied. On the other hand, if $err(S)$ is smaller than the error of merging s_{2B} with b_v, then FindMin determines the adjacent pair that causes the least increase in the total error. That pair could be (s_{2B}, b_v) or some other adjacent pair, but in either case, after that merge, the number of buckets drops to $2B$ and by induction hypothesis the remaining set of buckets satisfies the min-merge property. This gives us the following result on the quality of approximation of MIN-MERGE algorithm.

Theorem 1. *In the data stream model, the* MIN-MERGE *algorithm produces a $(1, 2)$-approximation for L_∞ error, using $O(B)$ memory and $O(\log B)$ update time per-item.*

2.2 The MIN-INCREMENT Algorithm

We now describe our second approximation scheme, called MIN-INCREMENT, which is an $(1 + \varepsilon, 1)$ algorithm. The problem at hand is to minimize the error in approximation using B buckets. For a moment, let us consider the following dual problem: *suppose we know the optimal error $e_{\rm OPT}$ and we want to minimize the number of buckets required to achieve this error.* (The two problems are

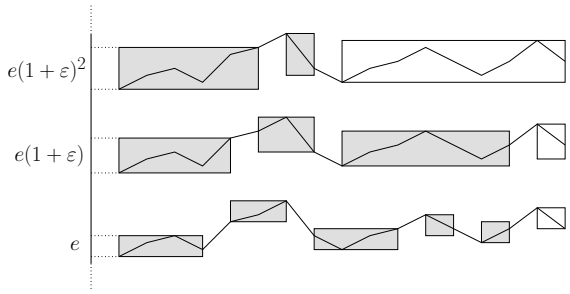


Figure 3. Example operation of the MIN-INCREMENT algorithm with exponentially increasing error summaries. The open buckets are shown unshaded.

duals in the following sense: one minimizes error given a fixed number of bucket, while the other minimizes number of buckets given a fixed error bound.) The simple but key insight is that this dual problem can be solved *optimally* by a simple greedy algorithm, which we will describe shortly.

The algorithm MIN-INCREMENT works by solving the afore-mentioned dual problem for several values of error. In particular, we divide the possible range of error $[1, U]$ into a set of exponentially increasing intervals defined by $\{e_i = (1+\varepsilon)^i : i = 0, 1, \ldots, \varepsilon^{-1} \log U\}$ for $0 < \varepsilon < 1$, and compute the minimum bucket approximation for each error e_i. Since the values e_i are separated by factors of $(1+\varepsilon)$, there always exists an e_j for which the following holds:

$$e_{j-1} \leq e_{\text{OPT}} \leq e_j \leq (1+\varepsilon) e_{\text{OPT}}. \quad (2)$$

The minimum bucket approximation for e_j will serve as our desired approximation

We now come back to the dual problem and describe the algorithm GREEDY-INSERT, which finds the minimum bucket approximation for target error e_i in a streaming fashion. The algorithm maintains a summary A_i consisting of a list of buckets, where the last bucket is *open*, i.e. new points can be added to that bucket; rest of the buckets are *closed* (see Fig. 3). Initially, A_i consists of exactly one empty open bucket with zero error. As new points arrive, we add them to the last bucket in the list, *as long as* the error of the bucket does not exceed e_i. When the error exceeds e_i, we close the last bucket and start a new open bucket, at the end of the list.

Lemma 2. *Given an error e, the* GREEDY-INSERT *algorithm finds a minimum bucket approximation whose error is at most e.*

Proof. Suppose GREEDY-INSERT computes the set of buckets A, for the given error e, and A has k buckets. Let $\{a_i : i = 0, \ldots, k\}$ be the markers defining the buckets, such that bucket i begins at stream index $a_{i-1} + 1$ and ends at a_i (since the buckets cover the entire stream, $a_0 = 0$ and $a_k = n$). Suppose an algorithm H produces a m-bucket approximation with the same error, where $m < k$, and let $\{h_i : i = 0, \ldots, m\}$ be the markers for H. There must be at least one index $i \in [0, m]$ for which $a_i < h_i$ because otherwise the m markers in H cannot cover the entire stream. We prove by induction that $a_i \geq h_i$, for all indices i, which immediately implies that H must have at least k buckets.

The claim is true for base case, since $a_0 = h_0 \ (= 0)$. Now assume that $a_{i-1} \geq h_{i-1}$ $(i > 1)$, and we want to show that $a_i \geq h_i$. For the sake of contradiction, assume that $a_i < h_i$. Then the i-th bucket in H contains stream values $[a_{i-1} + 1, a_i + 1]$. But since the error is monotonic, the error in this bucket must be greater than e, otherwise the greedy procedure would have added the value at $(a_i + 1)$ to the ith bucket in A. Since every bucket in H also has error at most e, it must be that $a_i \geq h_i$, for all i. □

In the MIN-INCREMENT algorithm (Algorithm 2), we keep a set of minimum bucket approximations A_i, with target errors $\{e_i = (1+\varepsilon)^i : i \in [0, \varepsilon^{-1} \log U]\}$. We insert each stream value in all the summaries A_i, using the GREEDY-INSERT algorithm (see Fig. 3). If a summary A_i grows more than B buckets, we delete it, because by Lemma 2, the error of optimum B-bucket histogram must be larger than e_i.

Algorithm 2 MIN-INCREMENT(A, v)

```
1: for all i ∈ 1, 2, ...|A| do
2:    A_i.GREEDY-INSERT(v);
3:    if |A_i| > B then
4:       delete A_i and e_i;
5:    end if
6: end for
7: n ← n + 1;
```

2.2.1 The Approximation Quality

To query for the min-error approximation for B buckets, we pick the summary for the smallest target error e_i that is still present. By construction this summary has at most B buckets. By ineq. 2 and lemma 2, the error in the approximation is at most $(1+\varepsilon)$ times the optimal, and therefore we have a $(1+\varepsilon, 1)$-approximation of the optimal histogram.

2.2.2 The Space and Time complexity

The MIN-INCREMENT algorithm stores summaries for at most $\varepsilon^{-1} \log U$ target errors, each with at most B buckets. Thus, the space requirement of MIN-INCREMENT is $O(\varepsilon^{-1} B \log U)$. For each input item, we need to update all

the summaries, at constant time apiece, and so the per-item update time is $O(\varepsilon^{-1} \log U)$.

The processing time can be improved further, by using batch processing. We keep an input buffer b of size M, in which we read the next M values of the stream. Now for each target error, we first check if the entire buffer fits into the last open bucket without exceeding the target error (using the max and min values of the buffer); if it does (Case 1), we simply put the entire buffer into the last bucket, otherwise (Case 2) we need to the scan the buffer and execute the GREEDY-INSERT algorithm as before. The first case requires a total processing of $O(M + \varepsilon^{-1} \log U)$ for the entire buffer. But notice if Case 2 occurs, we will add at least one more bucket to that approximation, and since we keep only B buckets per approximation, this can only happen $\varepsilon^{-1} B \log U$ times over the entire stream. So the cost for processing entire stream is $O(\frac{n}{M} \cdot (M + \varepsilon^{-1} \log U) + M\varepsilon^{-1} B \log U)$. We set $M = \varepsilon^{-1} \log U$, and since $n \gg \varepsilon^{-2} B \log^2 U$, we get the following theorem.

Theorem 2. *In the data stream model, the* MIN-INCREMENT *algorithm produces a* $(1+\varepsilon, 1)$*-approximation for* L_∞ *error, using* $O(\varepsilon^{-1} B \log U)$ *memory and* $O(1)$ *update time per-item.*

3 Piecewise Linear Histograms

We next discuss how to extend our two algorithms to piecewise linear (PWL) histograms over a data stream. A PWL-histogram uses a linear function (a line of arbitrary slope) to approximate the data in each bucket. The line is chosen to minimize the largest L_∞ distance of any point in the bucket. See Figure 1 for an example. (The traditional, single-value histogram can be thought of as a PWL-histogram that is limited to using *horizontal lines* only.)

Computing an L_∞ PWL-histogram is considerably more difficult than computing the ordinary histogram. In an ordinary histogram, when we add a new point to a bucket, we need to merely update the min and max values of the bucket, but these values are not sufficient for the PWL-histogram. In a PWL-histogram, we need to maintain the convex hull of all the points in the bucket.

3.1 PWL-histogram and Convex Hulls

In the PWL-histogram, we divide the input into k buckets, $\{s_1, s_2, ..., s_k\}$, where each bucket is represented by a tuple $s_i = \{beg_i, end_i, \ell_i, r_i\}$. (The values ℓ_i and r_i are not necessarily the input values at index beg_i and end_i, but are chosen to minimize the error of the bucket.) The data in bucket s_i is approximated using the line segment from point (beg_i, ℓ_i) to point (end_i, r_i), which has the slope

$$slope(s_i) = \frac{r_i - \ell_i}{end_i - beg_i}$$

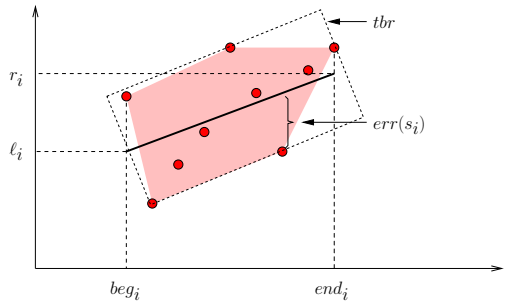


Figure 4. Representation of a PWL-histogram bucket: convex hull is shown shaded and the optimal line segment as a thick line.

The estimated value of item at index t (in bucket s_i) is $\widehat{v_t} = \ell_i + (t - beg_i) \cdot slope(s_i)$, and so the L_∞ error of this approximation is $|v_t - \widehat{v_t}|$. The approximation error of a bucket is the maximum error among all the items in the bucket.

To compute the optimum (least error) line segment for s_i, we need to maintain the convex hull of the points in s_i, which we denote by h_i. From the convex hull, we first compute the *thinnest bounding rectangle* (or *tbr*)[1]; the line segment bisecting the width of the *tbr* is the desired line segment. In Figure 4, the *tbr* is shown by dotted lines, and the optimum line segment (shown in bold) bisects the *tbr*. To control the number of points on the convex hull, we use the approximate convex hull algorithm proposed by Chan [3]. The approximate convex hull $\widehat{h}_i$ corresponding to the exact convex hull h_i, for any user specified threshold $0 < \varepsilon < 1$, has size $O(\varepsilon^{-1/2} \log(1/\varepsilon))$ and satisfies the following property:

$$(1 - \varepsilon) \cdot width(h_i) \leq width(\widehat{h}_i) \leq width(h_i). \quad (3)$$

The *width* of a point set is defined as the minimum distance between two parallel lines such that all the points lie between them. The approximate convex hull can be computed in a streaming fashion with one pass over all the points. In the interest of brevity, we omit the discussion of Chan's algorithm and refer the reader to the original paper [3].

3.2 MIN-MERGE and MIN-INCREMENT for PWL-Histogram

We first describe our extension of the MIN-MERGE algorithm for PWL-histogram approximation, which yields a

[1] The thinnest bounding rectangle is a rectangle of minimum width (side length) which encompasses the entire point-set. It can be efficiently computed by a linear scan of points in the convex hull [4].

$(1+\varepsilon, 2)$-approximation of the input stream. We maintain $2B$ buckets, $\{s_1, s_2, ..., s_{2B}\}$, where each bucket s_i contains the beg_i and end_i indexes, along with the approximate convex hull $\widehat{h}_i$ of points in s_i.

The main body of this algorithm is the same as outlined in Algorithm 1—only the details of how the buckets are merged in MERGE are somewhat different. Specifically, we take two adjacent buckets, s_i and s_{i+1}, and merge them by setting the convex hull of final bucket as the convex hull of union of $\widehat{h}_i$ and $\widehat{h}_{i+1}$. Two disjoint convex hulls can be merged in linear time [4]. This gives us the following theorem.

Theorem 3. *The* MIN-MERGE *algorithm produces a PWL-histogram with* $(1+\varepsilon, 2)$ *approximation under the* L_∞ *error, using* $O(\varepsilon^{-1/2} B \log(1/\varepsilon))$ *memory and* $O(\log B + \varepsilon^{-1/2}\log(1/\varepsilon))$ *per-item update time.*

In the PWL-histogram MIN-INCREMENT, the main algorithm remains the same as section 2.2, except that when the new point arrives, instead of updating the max and min values in the last bucket, we update the corresponding approximate convex hull. Due to lack of space, we omit details, and simply state our result.

Theorem 4. *The* MIN-INCREMENT *algorithm produces a PWL-histogram with* $(1+\varepsilon, 1)$ *approximation guarantee for* L_∞ *error, using* $O(\varepsilon^{-1} B \log U + \varepsilon^{-3/2}\log(1/\varepsilon)\log U)$ *memory and* $O(\varepsilon^{-1/2})$ *update time per-item.*

4 Extensions

In this section, we describe some extensions of our core algorithms (MIN-MERGE and MIN-INCREMENT).

4.1 Sliding Window Model

In the sliding window model, the goal is to maintain an approximate histogram for the most recent w items of the stream, $\{v_{n-w+1}, v_{n-w+1}, ..., v_n\}$, where w is the size of the window. The available memory is assumed to be much smaller than w, the window size, which means it is not possible to simply store the entire (or large portions) of the window. We first give a negative result, showing that it is not possible to compute the optimum B-bucket histogram using less than $\Omega(w)$ memory.

Lemma 3. *Any sliding window algorithm that computes an approximation with error less than or equal to the* B*-bucket optimal histogram requires* $\Omega(w)$ *memory, where* w *is the size of the window.*

We omit the proof due to space constraints. The main implication of this lower bound is that it is not possible to achieve the approximation quality of MIN-MERGE algorithm in the sliding window model: since the MIN-MERGE with $2B$-buckets will generate the optimum error for B-bucket histogram. In fact, no space-efficient $(1, \beta)$ histogram approximation for sliding window is possible.

We, therefore, resort to bounded memory (α, β)-approximations for $\alpha > 1$. Recently, Guha et al. [10] gave a $(1+\varepsilon, 1)$-approximation for sliding window, but the memory usage for their algorithm is $O(w)$. Thus, to the best of our knowledge, no sublinear space algorithm is currently known for constructing guaranteed quality histograms in the sliding window model. Somewhat surprisingly, it turns out that the MIN-INCREMENT algorithm generalizes to the sliding window model, and is able to do so *without any increase in memory usage*. In particular, the sliding window version of MIN-INCREMENT is a $(1+\varepsilon, 1+1/B)$-approximation using $O(\varepsilon^{-1} B \log U)$ memory.

In the sliding window version of the MIN-INCREMENT algorithm, we again maintain approximations for target errors $\{e_i : i = 0, 1, ..., \varepsilon^{-1}\log U\}$, and apply the GREEDY-INSERT algorithm to keep track of the approximation for each target error. Additionally, for each approximation, we delete a bucket s_j if all points in it are outside the window (i.e. $end_j < n - w$). It is not hard to prove that GREEDY-INSERT along with such deletion policy will give the following approximation result.

Lemma 4. *In the sliding window model, given an error* e*,* GREEDY-INSERT *produces a histogram with at most one more bucket than the optimum for the given error* e*.*

Notice that now we can no longer delete an approximation A_i if its size becomes larger than B, because as the window changes, the same target error can later require less than B buckets. To keep the memory in check, we apply the following policy: if $|A_i| > B+1$, we delete the oldest bucket in A_i (even if it is inside the window). The reason is that in this case, according to Lemma 4, the optimum B bucket approximation for current window must have error more than e_i. Hence we can safely trim the size of that approximation, and only keep the buckets for future windows. This leads to the following result.

Theorem 5. *In the sliding window model, the* MIN-INCREMENT *algorithm produces a* $(1+\varepsilon, 1+1/B)$ *approximate histogram for* L_∞ *error, using* $O(\varepsilon^{-1} B \log U)$ *memory and* $O(\varepsilon^{-1} B \log U)$ *update time per-item.*

4.2 Offline Optimal Histograms

While our primary focus is the histogram approximations in data streams, the ideas of this paper in fact also lead to a space-efficient construction of the optimal L_∞ histograms in the *offline* (non-streaming) model. Recently, in

an unpublished work, Guha et. al [9], have proposed an offline algorithm for the optimal L_∞ histogram which runs in $O(n + B^2 \log^3 n)$ and $O(n)$ space. The ideas of GREEDY-INSERT can also be used to construct an offline algorithm for the same problem in near linear time and space. Due to lack of space, we omit the details and simply state the following result.

Theorem 6. *There exists an algorithm for computing the optimal L_∞ histogram using memory $O(n)$ and time $O\left(n + n\frac{\log U}{\log(n/B)}\right)$.*

5 Experimental Results

In order to experimentally evaluate the performance of our algorithms, we tested them on a variety of real-world and synthetic data. In particular, we used the following data sets.

- *Dow-Jones*: Dow-Jones Industrial Average (DJIA) daily closing values from 1900 to 1993. [2] The size of this dataset is 25771.
- *Merced*: Hourly flow rate (in cubic-feet per second) of the Merced river at the Happy Isles bridge at Yosemite national park starting from 1997 to present. [3] The size of this data set is 65536.
- *Brownian*: This is a synthetic data set with 1 million points that simulate a one dimensional Brownian motion or random walk.

All the values are integers in the range $[0, 2^{15}-1]$, and so $U = 2^{15}$. For most of our experiments, the approximation parameter is set to $\varepsilon = 0.2$.

As a point of comparison, we used the REHIST scheme of Guha, Shim and Woo [12], which has the best space complexity among all previously known data stream algorithms for the histogram approximation. Their algorithm is described under the relative error metric, but it works for the maximum error as well, with the same bounds. In [12], several different implementations of REHIST are offered, with various time-space tradeoffs. Because our primary focus in space, we chose the version of REHIST that is optimized for memory. This version of the REHIST algorithm finds an $(1+\varepsilon, 1)$ approximation to the optimal B bucket histogram for a stream of size n, using total memory $O(\varepsilon^{-1}B^2 \log n)$ in time $O(nB(\log\log n + \log(B/\varepsilon)))$. We also implemented the optimal (offline) algorithm for benchmarking the error. (This algorithm is used only for the experiment reported in Figure 7.) Our experiments will refer to the following schemes:

1. OPTIMAL: The optimal L_∞ dynamic programming algorithm [17, 12]: it sets the benchmark error against which we compare all other approximation algorithms.
2. REHIST: The L_∞ histogram approximation scheme of [12].
3. MIN-MERGE: We implemented both ordinary and PWL versions of the MIN-MERGE algorithm for the L_∞ error.[4]
4. MIN-INCREMENT: We implemented both ordinary and PWL versions of the MIN-INCREMENT algorithm for L_∞ error.[5]

All implementations were done in C++ with STL. We used gcc 4.0 with all optimization options turned on. The experiments were run on an AMD Athlon 1.8 GHz PC with 350 MB RAM.

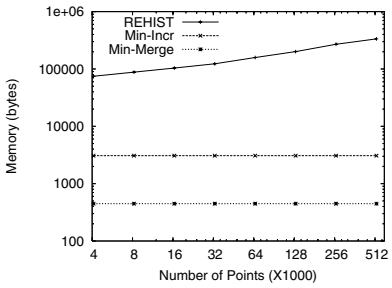


Figure 6. The memory usage as a function of n, the stream size, on the *Brownian* data set.

5.1 Memory Performance

In this experiment, we compare the space usage of our algorithms for the L_∞ error. We used 16384 points from the *Dow-Jones*, *Merced* and *Brownian* datasets to run the MIN-MERGE, MIN-INCREMENT, and REHIST algorithms for different values of B between 16 and 128, with the approximation error parameter fixed at $\varepsilon = 0.2$. The memory usage of the algorithms was measured in bytes.

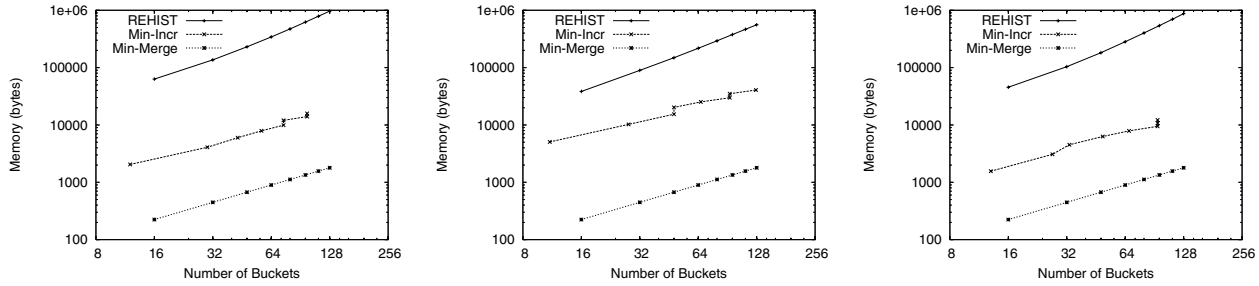
The results are shown in Fig. 5 on a log-log scale. The observed space usage (reported in bytes) closely tracks the theoretical prediction, and shows the factor B improvement over the REHIST scheme. For $B = 128$ buckets, the total space required is less than 2 KB for MIN-MERGE and about 10 KB for MIN-INCREMENT. The reader will notice that the plot for MIN-INCREMENT shows occasional

[2] Source: StatLib datasets archive. `http://lib.stat.cmu.edu/datasets/`

[3] Source: California Department of Water Resources. `http://cdec.water.ca.gov/cgi-progs/queryID?s=6946`

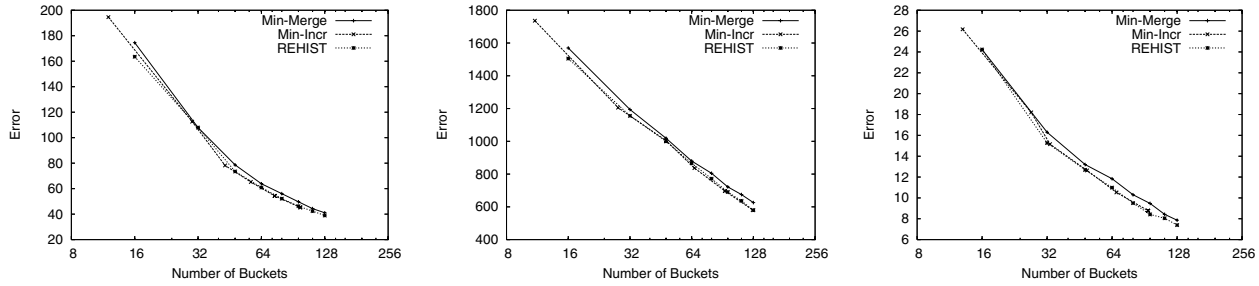
[4] We did not implement the heap data structure, so the running time reported in our experiments are with the slower $O(B)$ update time per item instead of $O(\log B)$.

[5] In this case also, the reported running times are using the slower $O(\varepsilon^{-1}\log U)$ update per item rather than $O(1)$.



(a) *Dow-Jones* dataset (b) *Merced* dataset (c) *Brownian* dataset

Figure 5. The memory usage as a function of the histogram size B (number of buckets).



(a) *Dow-Jones* dataset (b) *Merced* dataset (c) *Brownian* dataset

Figure 7. Approximation error as a function of the histogram size B (number of buckets).

jumps—this is because MIN-INCREMENT is driven by the approximation error, and does not always use all B buckets assigned to it.

Our second experiment (see Fig. 6) measures the memory usage as a function of the input stream size n. In order to scale to large n, we used the *Brownian* dataset, varying n from 4000 to 512000, while the number of buckets was fixed at $B = 32$. As expected, the space required is essentially independent of n.

5.2 Error Performance

Our third experiment (see Fig. 7) measures the number of buckets (size of the approximate histogram) required to achieve a fixed error. The actual measured error in all cases is significantly better than the $(1 + \varepsilon) = 1.2$ factor guarantee. In fact, the performance of REHIST and MIN-INCREMENT is almost the same as the optimal dynamic program, while the error produced by MIN-MERGE is marginally worse, as expected, reflecting its $(1, 2)$ approximation. In this example, to achieve the error of 9.5, OPTIMAL requires 80 buckets, while MIN-MERGE requires 96 buckets, i.e. 20% more. But for larger values of errors, MIN-MERGE requires about the same number of buckets as OPTIMAL. Conversely, for a fixed number of buckets (say 80) the error achieved by MIN-MERGE is 10.3 (about 13% worse than the optimal error 9.5).

5.3 Running Time

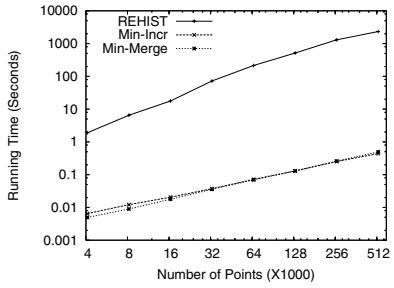


Figure 8. The running time for the *Brownian* data set.

We measured the running times of REHIST, MIN-INCREMENT and MIN-MERGE as a function of stream size n and the number of buckets B. Fig. 8 shows the plots of running times as a function of n, while the number of

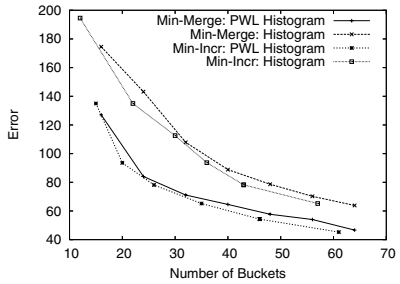


Figure 9. Improvement in approximation quality achieved by PWL-histogram.

buckets is fixed at $B = 32$. Both MIN-MERGE and MIN-INCREMENT show excellent performance, and are able to process more than half a million points in a fraction of a second.

5.4 Histograms vs. Piecewise Linear Histograms

This experiment (see Fig. 9) measures the improvement in the approximation quality due to the use of piecewise linear histograms, using the 16384 points *Dow-Jones* data set. The piecewise linear functions lead to about 30%–40% better approximation that the traditional (piecewise constant) histograms, for the same number of buckets.

6 Concluding Remarks

Histograms are an important tool for building compact summaries of large data streams. In this paper, we presented two space-efficient algorithms for computing guaranteed quality L_∞ histogram approximations in the data stream model. The space complexity of our algorithms improves the previous results by a factor of B (the number of buckets in the histogram), making them especially attractive in lightweight monitoring applications, such as sensor networks. Previous research has noted that the conceptual simplicity of the maximum errors leads to simpler and faster schemes for computing histogram. Our results extend that advantage to the space (memory) consumption as well.

Acknowledgments

We are grateful to Sudipto Guha for his helpful comments and for providing us with preprints of his unpublished work.

References

[1] P. K. Agarwal, S. Har-Peled, N. H. Mustafa, and Y. Wang. Near-linear time approximation algorithms for curve simplification. *Algorithmica*, 42:203–219, 2005.

[2] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, Sep. 2001.

[3] T. Chan. Faster core-set constructions and data stream algorithms in fixed dimensions. In *Proc. ACM SOCG*, 2004.

[4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry : Algorithms and Applications*. Springer-Verlag, 2000.

[5] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proc. of ACM STOC*, 2002.

[6] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. One-pass wavelet decompositions of data streams. *IEEE Trans. on Knowledge and Data Engineering*, 15(3):541–554, 2003.

[7] S. Guha. Histograms, wavelets, streams and approximation. *unpublished*, 2006.

[8] S. Guha and B. Harb. Wavelet synopsis for data streams: Minimizing non-euclidean error. In *Proc. of ACM SIGKDD*, 2005.

[9] S. Guha, C. Kim, and K. Shim. A short note on optimum algorithms for maximum error histograms. *unpublished*, 2006.

[10] S. Guha and N. Koudas. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In *Proc. of ICDE*, 2002.

[11] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. of ACM STOC*, 2001.

[12] S. Guha, K. Shim, and J. Woo. Rehist: Relative error histogram construction algorithms. In *Proc. of VLDB*, 2004.

[13] Sudipto Guha. Space efficiency in synopsis construction algorithms. In *Proc of VLDB*, 2005.

[14] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engg. Bulletin*, 2000.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System archtecture directions for networked sensors. *ACM SIGPLAN Notices*, 35:93–104, 2000.

[16] Y. E. Ioannidis. The history of histograms (abridged). In *Proc. of VLDB*, 2003.

[17] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *Proc. of VLDB*, 1998.

[18] P. Karras and N. Mamoulis. One-pass wavelet synopses for maximum-error metrics. In *Proc. of VLDB*, 2005.

[19] E. J. Keogh, K. Chakrabarti, S. Mehrotra, and M. J. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In *Proc. of SIGMOD*, 2001.

[20] E. J. Keogh, S. Chu, D. Hart, and M. J. Pazzani. An online algorithm for segmenting time series. In *Proc. of ICDM*, 2001.

[21] R. Kooi. *The Optimization of Queries in Relational Databases.* PhD thesis, Case Western Reserve Univ., 1980.

[22] M. Muralikrishna and D. J. DeWitt. Equi-depth multidimensional histograms. In *Proc. of SIGMOD*, 1988.

[23] R. Szewczyk, A. Mainwaring, J. Polastre, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proc. of SenSys*, 2004.

[24] B.-K. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary lp norms. In *Proc. of VLDB*, 2000.

[25] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proc. of VLDB*, 2002.