Approximating a Data Stream for Querying and Estimation: Algorithms and Performance Evaluation

Sudipto Guha
AT&T Labs-Research
sudipto@research.att.com

Nick Koudas*
AT&T Labs-Research
koudas@research.att.com

Abstract

Obtaining fast and good quality approximations to data distributions is a problem of central interest to database management. A variety of popular database applications including, approximate querying, similarity searching and data mining in most application domains, rely on such good quality approximations. Histogram based approximation is a very popular method in database theory and practice to succinctly represent a data distribution in a space efficient manner.

In this paper, we place the problem of histogram construction into perspective and we generalize it by raising the requirement of a finite data set and/or known data set size. We consider the case of an infinite data set on which data arrive continuously forming an infinite data stream. In this context, we present the first single pass algorithms capable of constructing histograms of provable good quality. We present algorithms for the fixed window variant of the basic histogram construction problem, supporting incremental maintenance of the histograms. The proposed algorithms trade accuracy for speed and allow for a graceful tradeoff between the two, based on application requirements.

In the case of approximate queries on infinite data streams, we present a detailed experimental evaluation comparing our algorithms with other applicable techniques using real data sets, demonstrating the superiority of our proposal.

1 Introduction

Obtaining fast and good quality approximations to data distributions is a problem of central interest to database management. A variety of database applications including, approximate querying, similarity searching and data mining in most application domains, rely on such good quality approximations. A very popular way in database theory and practice to approximate a data distribution is by means of a histogram.

Histograms approximate a data distribution using a fixed amount of space, and under certain assumptions strive to minimize the overall error incurred by the approximation. The problem of histogram construction is of profound importance and has attracted a lot of research attention. A fundamental requirement for any good histogram construction algorithm is to approximate the underlying data distribution in a provably good way and be efficient in terms of running time. The typical assumption for constructing histograms, is that the data set to be approximated is finite and of known size, or that the size can be easily derived by performing a single pass over the finite data set. The problem of histogram construction on datasets of known size has received a lot of research attention and the optimal as well as a lot of heuristic solutions exist. We refer to this problem as the *Classic Histogram* construction problem. A number of applications benefit from good solutions to the classic histogram problem, including query optimization [PI97], approximate queries on data warehouses [AGPR99] and time series similarity queries [KCMP01, YF00]. In this paper, we place the problem of histogram construction into perspective and we generalize it by raising the requirement of a finite data set and/or known data set size. We consider the case of an infinite data set on which data arrive continuously forming an infinite data stream.

Data streams prevail in a variety of applications, including networking, financial, performance management and tuning. Network elements, like routers and hubs produce vast amounts of stream data. For example, a router, for performance monitoring, records the traffic that flows through at specific time intervals (in terms of number of bytes, average utilization etc), producing data streams. Such a stream can easily accumulate to multiple gigabytes rapidly. Network elements produce many different types of information on a stream, like fault sequences recording various types of network faults or flow sequences specifying the different types of network flows initiated, etc. Stock sequences abound in the financial world and is a typical form of time series encountered. Web servers or other service based applications produce many different types of data streams (for example, number of connections to a web server at some time granularity, a click stream sequence in terms of number of bytes retrieved, etc.) which are imperative to monitor and analyze. In all the applications above, it is crucial to provide querying abilities, capable of adapting to the online and incremental nature of the data source. For example network operators commonly pose queries, requesting the aggregate number of bytes over network interfaces for time windows of interest. The dynamic nature of data streams raises challenging questions for query processing.

We formulate and propose solutions for problems related to maintenance of histograms of provable good quality over infinite data streams for query processing and estimation purposes. Analysis and query processing on data streams posses challenging questions both at a theoretical and a practical level. First, the algorithms should be able to operate on fixed amount of memory, sufficient to buffer a portion of the data stream at a time, since the entire stream is not available. Second, algorithms should be fast and able to keep up with the online nature of the stream. Third, for better efficiency and speed, algorithms should be incremental, in the sense that as new data are produced (and possibly old data discarded) intermediate results should be reused. The nature of data stream algorithms can either be **agglomerative** or **fixed window**. Consider the case of a data source that produces a new data element at each time unit. An agglomerative algorithm would maintain enough information to perform analysis or answer queries, starting from the beginning of the stream up to the current time. A fixed window algorithm would maintain information and perform analysis over specific temporal windows of interest, say over the latest T seconds of data produced. Figure 1 gives an example of the two kinds of data stream algorithms. In both cases, the algorithms are operating using a fixed amount of memory, M. In Figure 1(a) the operation of an agglomerative algorithm is presented. The algorithm operates on the entire stream, starting from time 0 up to and including the current time T_0. In contrast, a fixed window algorithm (Figure 1(b)) operates on a window of length T at any time.

In this paper, we present a formal study of histogramming algorithms for data streams, by introducing algorithms that could be used in a variety of popular database applications involving estimations. We present the first efficient data stream histogramming algorithm for the fixed window data stream model. Our algorithms have *provably good properties* and allow a *graceful tradeoff* between construction time and accuracy of estimation. These algorithms complement our agglomerative data stream histogram algorithms presented in [GKS01] and together solve the one pass histogram construction problem in its entirety.

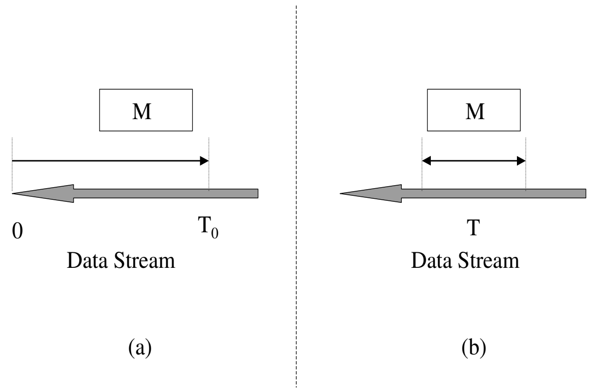


Figure 1. Examples of data stream algorithms: (a) An Agglomerative Algorithm (b) a Fixed window algorithm

In section 2 we review related work. Section 3 presents definitions necessary for the remainder of the paper. In section 4 we present our algorithms to approximate a data stream using histograms and we show the properties of the resulting approximation analytically. In section 5 we present the results of a detailed experimental evaluation of the application of our algorithms to various problems of interest. Section 6 concludes the paper and points to interesting problems for further study.

2 Related Work

Data stream algorithms have attracted research attention both in the theoretical as well as in the database community. The first results in data streams were the results of Munro and Paterson [MP80] where they studied the space requirements of selection and sorting as a function of the number of passes over the data, however stream algorithms are one pass algorithms. The model was formalized by Henzinger et. al., [HRR97] who gave several algorithms and complexity results related to graph theoretic problems. Guha et. al., [GMMO00] presented agglomerative algorithms to cluster data on a stream. Guha et. al., [GKS01] presented agglomerative algorithms for the problem of histogram construction on a data stream. Manku et. al., [SRL98, SRL99] considered the problems of computing medians, quantiles and order statistics with a single pass over a data set. Greenwald and Khanna, [GK01] presented an improvement on the algorithms by Manku et. al., requiring less memory. Domingos and Hulten considered the problem of decision tree construction with a single pass over the data set [DH00] and Hidber the problem of online association rule mining [Hid99]. Other results are available elsewhere [FM83, FKSV99, Ind00].

In the database community, the problem of efficient histogram construction has received much research attention, due to the central role histograms play in query optimization and approximate query answering [IP95, AGPR99]. There exists an optimal solution to the classical histogram construction problem [JKM+98], with quadratic (on the number of data points) complexity. Alternate histogramming algorithms have been proposed based on the use of transforms (e.g discrete Cosine, Wavelet etc) [LKC99, MVW, MVW00].

3 Histogramming Problem Definition

We consider data streams consisting of an ordered sequence of data points that can be read only once. Formally, a data stream is an infinite sequence of points $\ldots x_i, \ldots$ read in increasing order of the indices i. Each value x_i is an integer drawn from some bounded range. Data stream algorithms are incremental bounded memory, one pass algorithms. On seeing a new point x_i, two situations of interest arise, either we are interested in a finite window of n points, $x_{i-n}, \ldots, x_i$, or we are interested in all points seen. We will denote the former as a *fixed window* data stream model and the latter as an *agglomerative* data stream model. In the agglomerative model, the sequence is denoted by $x_1, \ldots, x_N$, where N is the total number of points seen. The central aspect of data stream computation is modeling computation in small space relevant to the parameter of interest n or N [1]. In this paper we are primarily interested in efficient incremental algorithms for data streams. We concentrate on *fixed window* data stream algorithms, however inside a window the behavior of the algorithm is agglomerative, necessitated by the incremental nature.

We assume availability of fixed buffer space (memory available) M capable of holding n points of the stream (a data stream subsequence of length n) at any time. Let $M[0], \ldots M[n-1]$ the locations of data points in the buffer. Buffer M operates in a cyclic fashion. Thus, when point $i \geq n$ arrives, the temporally oldest point ($M[0]$) is evicted from the buffer, and point i is placed at $M[n-1]$. Consequently, buffer M acts as a sliding window of length n over the data stream. Successive data stream subsequences in the buffer, have $n-1$ data points in common [2].

As a first step, we are interested in designing algorithms that approximate and dynamically maintain the approximation of the data distribution of the data stream points in the buffer M. The approximation obtained should be suitable for obtaining answers to common queries about the values of points in the buffer, such as point and range queries. For representation of the approximation we choose histograms, which is a popular estimation technique. The resulting histograms should have good estimation quality, they should allow fast incremental maintenance, as points are added or deleted from the buffer, which should preserve the estimation quality. An agglomerative data stream histogram algorithm will maintain histograms reflecting the data distribution of the stream starting from time 0. A fixed window data stream histogram algorithm will maintain histograms reflecting at any time the data distribution of the sequence of the n buffer points.

Let X be a finite data point sequence, $v_i, 1 \leq i \leq |X|$. The general problem of histogram construction is as follows: given some space constraint B, create and store a compact representation of the data sequence using at most B storage, denoted H_B, such that H_B is optimal under some notion of error $E_X(H_B)$ defined between the data sequence and H_B. We first need to decide on a representation for H_B. The typical representation adopted results from multiple instances of the following technique: collapse the values in a sequence of consecutive points (say $s_i \ldots e_i$) into a single value h_i (typically their average, $h_i = \sum_{j=s_i}^{e_i} v_j$), thus forming a bucket b_i, that is, $b_i = (s_i, e_i, h_i)$. Since h_i is only an approximation for the values in bucket b_i, whenever a bucket is formed an error is induced due to the approximation. We need to choose a way to quantify this error. As is common, we quantify this error using the *Sum Squared Error* (SSE) metric, defined as:

$$F(b_i) = \sum_{j=s_i}^{e_i} (v_j - h_i)^2 \qquad (1)$$

Assuming that the space constraint is expressed in buckets, the resulting approximation H_B will consist of B buckets and the total error in the approximation is $E_X(H_B) = \sum_{i=1}^{B} F(b_i)$.[3] The optimal histogram construction problem is then defined as follows:

Problem 1 (Optimal Histogram Construction) *Given a sequence X of length $|X|$, a number of buckets B, and an error function $E_X()$, find H_B to minimize $E_X(H_B)$.*

The resulting optimal histogram is the well known `vopt` histogram [IP95]. There exists an efficient algorithm to determine the `vopt` histogram [JKM+98], in time $O(n^2 B)$ and requires $O(n)$ space. The algorithm requires the $O(n)$ prefix sums to be computed and stored and it is quadratic in the parameter of interest (B is typically assumed to

[1] Several authors have also introduced models allowing small, but fixed number of passes over the dataset, we will assume a single pass in this paper.

[2] Without loss of generality we assume that a new point arrives at each time step, other possibilities exist (e.g., batched arrivals) and indeed our framework can incorporate those as well.

[3] Other error functions are possible such as $\max_i F(b_i)$ etc. All our results will hold for any point-wise additive error function $E_X(H_B)$, but we will focus on the specific error function above which is the most common one in the literature.

be a small constant). A naive application of the optimal histogram construction algorithm to each subsequence of length n in the stream will result in an incremental algorithm that requires $O(n^2)$ time per new data item and $O(n^2)$ space. A fairly simple trick reduces the required space to $O(n)$, but the time required per update is excessive.

However, the histogram is used to *approximate* the data, and an obvious question would be if we are given a precision ϵ (say 0.01%) close to 0, can we compute a histogram that is not necessarily optimal, but close to the optimal in the precision parameter, faster? This leads to the following problem:

Problem 2 (ϵ-approximate Histograms)
Given a sequence X of length $|X|$, a number of buckets B, an error function $E_X()$, and a precision $\epsilon > 0$ find H_B with $E_X(H_B)$ less than $(1+\epsilon)\min_H E_X(H)$ where the minimization is taken over all histograms with B buckets.

We will present an algorithm that will take $O((nB^2/\epsilon)\log n)$ time to construct an ϵ-approximate histogram. However even such will be adequate, since it will require $O(n)$ time incrementally. Our target would be an algorithm which grows very slowly on n, which gives rise to the following problem:

Problem 3 (Histogramming a Data Stream) *Given a data stream S, a number of buckets B, a fixed memory space to store a portion of the stream, capable of storing n data stream points, an error function $E_S()$, and a precision ϵ, incrementally maintain the best H_B, to minimize $E_S(H_B)$ within $1+\epsilon$ factor.*

We will present an algorithm for the fixed window variant of the data stream histogram construction problem. The algorithm will approximate the result of the optimal algorithm on the n points in the buffer, requiring $O((B^3/\epsilon^2)\log^3 n)$ incrementally, trading accuracy with construction speed and vice versa.

4 Histogramming a Data Stream

We will present our solution to the data stream histogramming problem in the following steps:

- We will first present a description of the optimal histogram algorithm of Jagadish et. al., [JKM+98].
- We will show directions for improvement of the optimal histogram algorithm, which will be used in developing our fixed window algorithm.
- For completeness, we will present the agglomerative data stream construction algorithm [GKS01] and subsequently show why it does not give a reasonable fixed window algorithm.
- Finally, we will present the fixed window histogram construction algorithm.

4.1 Optimal Histogram Construction

Consider a sequence of data points indexed by $1\ldots n$. The data points within each bucket are represented by their mean. We refer to the optimal histogram consisting of B buckets as the optimal B-histogram. A basic observation is that if the last bucket contains the data points indexed by $[i+1,\ldots,n]$ in the optimal B-histogram, then the rest of the buckets must form an optimal $B-1$-histogram for $[1,\ldots,i]$. It is easy to observe that otherwise the cost of the solution can decrease by taking the optimal $B-1$-histogram and the last bucket defined on points $[i+1,\ldots,n]$. Since the last point, n, must belong to some bucket, we search over the $n-1$ bucket boundaries. To put things together, we must solve the problem for $2 \le k \le B-1$ for all points, since the bucket boundary for the last bucket can be anywhere between 2 and n[4]. Given that a bucket is defined by $[i,\ldots,j]$, the error $\text{SQERROR}[i,j]$ in it, is given by

$$\text{SQERROR}[i,j] = \sum_{\ell=i}^{j} v_\ell^2 - \frac{1}{j-i+1}\left(\sum_{\ell=i}^{j} v_\ell\right)^2 \qquad (2)$$

After we decide that the last bucket is $[i+1,\ldots,n]$, we need to compute its error efficiently. By a naive approach this can be easily done in $O(n)$ time. However, if we maintain two arrays SUM and SQSUM, we can compute the error of a bucket in $O(1)$ time.

$$\text{SUM}[1,i] = \sum_{\ell=1}^{i} v_\ell \qquad \text{SQSUM}[1,i] = \sum_{\ell=1}^{i} v_\ell^2 \qquad (3)$$

The partial sums in equation 2 can be replaced by $\text{SUM}[1,j]-\text{SUM}[1,i-1]$ and $\text{SQSUM}[1,j]-\text{SQSUM}[1,i-1]$. Denote by $\text{HERROR}[i,k]$ the error of representing $[1,\ldots,i]$ by a histogram with k buckets. The algorithm is presented in Figure 2. We can establish that the complexity of the above algorithm is $O(n^2B)$.

4.2 Directions for Improving the OptimalHistogram Algorithm

The basic structure of dynamic programming requires the top two loops (lines 1 and 2). Thus, we focus our attention to the last loop (lines 3,4) and try to improve the running time. We observe the following,

1. $\text{SQERROR}[i+1,j]$ is a positive non-increasing function if j is fixed and i increases. This can be shown

[4] While solving the case $k=B$, the last bucket can start at j, $B \le j \le n$.

```
Algorithm OptimalHistogram()
Compute SUM[1, i] and SQSUM[1, i] for all 1 ≤ i ≤ n
Initialize HERROR[j, 1] = SQSUM[j, n], 1 ≤ j ≤ n
1.  For j=1 to n do
2.   For k=2 to B do
3.    For i=1 to j-1 do
4.     HERROR[j, k] =
       min(HERROR[j, k], HERROR[i, k − 1] + SQERROR[i + 1, j])
```

Figure 2. Algorithm OptimalHistogram

mathematically, the intuitive reasoning is that: since i increases, we have a smaller set of numbers to summarize in a bucket.

2. $\text{HERROR}[i, k-1]$ is a positive non-decreasing function as i increases. Once again this can be shown rigorously, but the idea is that as i increases, we are trying to represent larger set of numbers by the same number of buckets (in this case $k-1$).

This could potentially be useful since we are searching for the minimum of the sum of two functions, both positive, one of which is non-increasing and the other non-decreasing. It would be a natural idea to use the monotonicity and reduce the search, say to logarithmic terms. However it can be easily shown that any sequence of numbers can be represented by a sum of a non-increasing and a non-decreasing function. Consider a sequence $x_1, \ldots x_n$ of positive values. Consider the two functions

$$F(i) = \sum_{j=1}^{n} x_j - \sum_{j=1}^{i-1} x_j \quad \text{and} \quad G(i) = \sum_{j=1}^{i} x_j$$

Observe that $x_i = F(i) + G(i)$ and that both functions are positive and satisfy our observation that $F(i)$ is non-increasing and $G(i)$ is non-decreasing. But the value of the i'th element in the sum is $\sum_{j=1}^{n} x_j + x_i$ which still leaves us with finding the minimum of x_i over all i. For example consider the initial sequence $3, 7, 5, 8, 2, 6, 4$; F is $35, 32, 25, 20, 12, 10, 4$ and G is $3, 10, 15, 23, 25, 31, 35$. The sum evaluates to the sequence $38, 42, 40, 43, 37, 41, 39$ which is the original sequence shifted by 35. Thus, the complexity of finding the exact minimum is the same as finding the exact minimum of the original sequence. This implies that the search step (lines 3,4 as described in the OptimalHistogram algorithm above) cannot be reduced if we are searching for an exact minimum, because we reduced the search for the minimum in a sequence to the search for the minimum in the sum of two functions with the desired properties. However, this reduction does not preserve approximation since 38 is closer to 37 that 3 is to 2 in terms of ratio. The shift by 35 does not preserve the approximation ratio. Since we are interested in finding the minimum upto some ratio close to one as opposed to exact, we can do much better than the simple search over n values, as we describe next.

4.2.1 Improving the search

In our search problem, we are trying to find the minimum of n values, which arise from the sum of two functions. In an interesting twist, we can attempt to represent the functions as histograms. More clearly, suppose we approximate the functions with a set of ranges (much fewer than n) and still have a good bound on the values of the function in each range. Then, the search can be performed at one point in each of the ranges, which will reduce the search from n to the size of the range set. This is precisely how histograms are used in applications.

Suppose for $a \leq b$, we have $(1+\delta)\text{HERROR}[a, k-1] \leq \text{HERROR}[b, k-1]$, for some $\delta \geq 0$. For any i, $(a \leq i \leq b)$, using the monotonicity properties we get:

$$\begin{aligned} &\text{HERROR}[i, k-1] + \text{SQERROR}[i+1, j] \geq \\ &\text{HERROR}[a, k-1] + \text{SQERROR}[b+1, j] \geq \\ &\frac{1}{1+\delta}(\text{HERROR}[b, k-1] + \text{SQERROR}[b+1, j]) \end{aligned}$$

Thus, we can reduce the search, by evaluating the sum of the two functions at the endpoint b instead of the entire range $[a..b]$. Given a non-decreasing function we will construct intervals which will satisfy the assumptions as the a, b pair. These intervals are the range sets, and approximate the function by a histogram (piecewise constant function) such that the difference of values in each bucket is small, but relative to the magnitude of the values in the bucket.

4.3 An Agglomerative Histogram Algorithm

The agglomerative histogram algorithm presented in [GKS01] does not give the incremental (fixed window) algorithm we seek, but we present the algorithm for the sake of comparison, and to point out where such an algorithm fails to be incremental. The entire algorithm is presented in Figure 3. To reduce the search the algorithm evolves by covering the index set $[1, \ldots, j-1]$ by intervals $(a_1, b_1), \ldots, (a_\ell, b_\ell), \ldots$ with the following property (Let $\delta = \epsilon/(2B)$, for some $\epsilon \geq 0$):

$$\begin{aligned} a_\ell = b_{\ell-1} + 1 \ \text{ and } \ &\text{HERROR}[b_\ell, k-1] \\ &\leq (1+\delta)\text{HERROR}[a_\ell, k-1] \ \text{ and } \ b_\ell \\ &\leq j-1 \text{ is maximal} \end{aligned}$$

In [GKS01] it is shown that the maximum number of intervals required can be bounded and that the space required to store all the intervals is $O(\frac{B}{\delta}\log n)$. Observe that the algorithm will never be interested in $\text{SUM}[1, i]$ or $\text{SQSUM}[1, i]$ unless i is one of the endpoints of the intervals. Thus

Algorithm AgglomerativeHistogram()
Set up $B-1$ *queues to store the intervals*
Assume we have access to last element in queue
1. For j:=1 to n do $\{$
2. Compute $\text{H}\textsc{error}[j,1] = \text{S}\textsc{qerror}[1,j]$.
3. For k:=2 to B do
4. For i:= endpoint b_ℓ *of queue* $k-1$
5. $\text{H}\textsc{error}[j,k] =$
$\min\left(\text{H}\textsc{error}[j,k], \text{H}\textsc{error}[i,k-1] + \text{S}\textsc{qerror}[i+1,j]\right)$
6. $\}$
7. If $(1 \le k \le B-1$ *and* $\text{H}\textsc{error}[j,k] > (1+\delta)\text{H}\textsc{error}[a_l,k])$
*8. /** a_l *is the start of the last interval in* k*'th queue */*
9. then start a new interval $a_{l+1} = b_{l+1} = j$ *for the* k*'th* $\{$
10. queue and store the values $\text{S}\textsc{um}[j]$ *and* $\text{S}\textsc{qsum}[j]$.
11. $\}\}$

Figure 3. Algorithm AgglomerativeHistogram

one can maintain a *running* prefix sum, and prefix sum-of-square, and store them only when we create an interval. The algorithm maintains $B-1$ queues implemented through arrays of bounded size, since there is a bound on the size of the queues. Keeping track of the number of elements in the queue, one can also access the last element and update it easily. Each element of the i'th queue will store the index number x, the $\text{S}\textsc{um}[1,x]$, $\text{S}\textsc{qsum}[1,x]$ and $\text{H}\textsc{error}[x,i]$ values.

Putting everything together, as shown in [GKS01], one can compute an ϵ-approximate B-histogram in time $O((nB^2/\epsilon)\log n)$ in space $O((B^2/\epsilon)\log n)$ in a data stream.

4.4 Considering Incremental Behavior

The above procedure is not very useful in constructing a fixed window histogramming algorithm, since the computation of a histogram on $[1,\ldots,n]$ does not allow us any information on $[2,\ldots,n+1]$. If we have a good approximation by intervals of a function, it does not necessarily approximate the same function, if the function is shifted by a constant amount. Consider Figure 4. Figure 4(a) shows a function approximated by piece wise constant functions over four intervals. Figure 4(b) shows the same function as in Figure 4(a) but the function is shifted downwards by a constant amount. Figure 4(b) uses the same intervals as in Figure 4(a) and we observe that the function values within the same interval are not so close to their approximation anymore. The correct covering is given in Figure 4(c). It can be easily argued that in case of a downward shift, the number of buckets would go up, to maintain the same guarantees as before. Thus, an agglomerative algorithm, will require $O(n \log n)$ time per data point to adjust the histogram, since the best one can do is to compute all the relevant information from scratch. This is a problem for any agglomerative algorithm that tries to maintain a data structure which approximates $\text{H}\textsc{error}$ starting from the first point seen. In some sense, in the agglomerative algorithm the next boundary of an interval depends on the initial points much more than on the latter points.

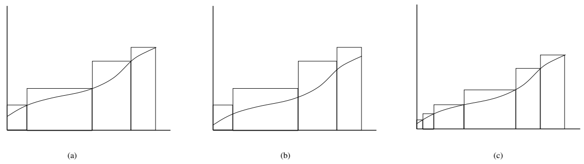


Figure 4. Shifting a function downwards

We require an algorithm that builds these list of intervals approximating the function on the fly. In addition, an element of recursion is required, since to compute histogram approximations we would need approximate histograms with fewer buckets. This is what we are going to present next. In this new algorithm, on seeing a new data point, we will *create* the space of ranges to be searched, and then locate the best value within that space. For all these, the time will be bounded by a polynomial in B and the space by a polynomial in $\frac{1}{\epsilon}$. Thus, in comparison, we will have a gain of $\frac{n\epsilon}{B}$ over the straightforward use of the agglomerative algorithm.

4.5 Fixed Window Data Stream Histograms

In this section we present an algorithm for the fixed window histogram construction problem. Our computation will resemble a *need based* strategy. To compute the optimal B-histogram we need the values of the $B-1$-histogram and so on. On seeing a new point we would have to create sets of intervals for each of the k-histograms. But first, we need to formalize the fixed window algorithm. Once again, let $\delta = \epsilon/(2B)$.

We assume a circular buffer that reduces the storage to $O(n)$ which is storage for the window anyways. Using this buffer we will create the lists of intervals on seeing the next input point. These will be computed on the fly; we will take $o(n)$ time to compute the intervals, since not all points in the buffer will be inspected. We store the sum of the values $(\sum_{j=\ell}^{i} x_i)$ and the sum of squares of the values $(\sum_{j=\ell}^{i} x_i^2)$ from some point in the past, ℓ. From time to time (after n iterations) we will update this point to be the first point in our window. This will require $O(n)$ time, but amortized over n iterations, can be ignored. To maintain these sums we use two arrays $\text{S}\textsc{um}'$ and $\text{S}\textsc{qsum}'$, indexed on $[0..n]$, where $\text{S}\textsc{um}'[i] = \text{S}\textsc{um}[1,i]$ and likewise for $\text{S}\textsc{qsum}'$. On seeing the $n+1$'st point of value v we will update the 0'th entry to be $\text{SUM}'[n] + v$ and likewise for SQSUM'. It is easy to verify that the partial sum of elements can be easily computed by subtracting $\text{SUM}'[1]$ from every other SUM' entry (similarly for SQSUM'). Whenever the algorithm requires the value of $\text{SQERROR}[i, j]$ we will compute it by $\text{SQSUM}'[j] - \text{SQSUM}'[i] - \frac{1}{j-i} * (\text{SUM}'[j] - \text{SUM}'[i])^2$ following equation 2.

We present the algorithm in Figure 5. The main issue is the procedure to build the list of required intervals each time. This is accomplished by function $CreateList[a, b, k]$. It accepts as input the start, a and end, b of the range for which the list of intervals is build and the number of required intervals k. It is defined recursively as shown in figure 5. The main feature of this function is to create a list of intervals approximating $\text{HERROR}[j, k]$ for each j. This is done for each $k \leq B - 1$. If the list for some k is available, we can compute $\text{HERROR}[j, k + 1]$ for all j. We can use these values to create the $k + 1$'th list of intervals, but the challenge is to create the list without evaluating *all* the values in the buffer. This we achieve by a binary search like procedure.

The algorithm is demonstrated by the following example:

Example 1 *Consider the case of a stream consisting of the values* $100, 0, 0, 0, 1, 1, 1, 1$ *with* $\delta = 1$*, and* $B = 2$*. When we run* $CreateList[1, 8, 1]$*, we will compute the intervals* $(1, 1), (2, 8)$ *as before. Consider the case of* 100 *being dropped and* 1 *being inserted at the end. We will now call* $CreateList[1, 8, 1]$ [5]*. The value of* $\text{HERROR}[1, 1] = 0$*. Thus when we invoke* $CreateList[1, 8, 1]$*, since* $a = 1$*, we will search for* c *such that* $\text{HERROR}[c, 1] = 0$ *and* c *is maximal. This would be satisfied for* $c = 3$*. Thus* 3 *would be inserted at the end of the queue, the queue will be* $(1, 3)$ *and we will invoke* $CreateList[4, 8, 1]$*. The value of* $\text{HERROR}[4, 1]$ *is* 0.75*, and* c *would evaluate to* 6 *in this case since* $\text{HERROR}[6, 1] = 1.5$*. We add* $(4, 6)$ *to the queue, making it* $(1, 3), (4, 6)$ *and invoke* $CreateList[7, 8, 1]$*. The last interval will be* $(7, 8)$*. The endpoints of the intervals would be* $3, 6, 8$*. Thus while computing* $\text{HERROR}[8, 2]$*, which is the solution required, we will minimize over the partition being at* 3 *or* 6 *and compute the right solution to be* $(1, 3), (4, 8)$*.*

The important point is that the binary search has now detected the transition at position 3 *from* 0 *to* 1*, thus we will find the optimal solution. Also we computed the correct interval partitioning.*

It is straightforward to observe that we would be creating valid intervals, and the endpoint b_ℓ's will have the maximal property. Since at the start of the new interval the function HERROR increases by a factor of $(1 + \delta)$, the maximum number of intervals will be $1 + \log_{(1+\delta)}(\text{HERROR}[n, B])$. This follows from the fact that the index values for which the HERROR is zero will form one interval, and then the HERROR is at least $\frac{1}{2}$ (the worst example is 1 position differing by 1, giving an HERROR of $1 - \frac{1}{n}$; we assume $n > 1$) and the start of a new interval has HERROR value greater than $(1+\delta)$ times the HERROR value at the start of the previous interval[6]. The term, $\log \text{HERROR}[n, B]$ can be bounded by $\log n + 2 \log R$ where R is the largest number, which is $O(\log n)$ since the values of the data points are assumed to be bounded. Thus the maximum number of intervals inserted in the queue is $O(\delta^{-1} \log n)$.[7]

To analyze the running time observe that there are at most $\frac{1}{\delta} \log n$ binary searches, at most one search per insertion. Each binary search will involve at most $\log n$ computations of $\text{HERROR}[c, k]$. Each such computation of $\text{HERROR}[c, k]$ will involve a minimization over $\frac{1}{\delta} \log n$ endpoints whose values are already stored in the $k-1$'th queue. Each binary search will require $O(\frac{1}{\delta} \log^2 n)$ time. Thus the total running time of $CreateList[1, n, k]$ is $O((\frac{1}{\delta})^2 \log^3 n)$. This for a single value of k; the total time taken by the incremental algorithm is $O(B(\frac{1}{\delta})^2 \log^3 n)$ which is $O(B^3 \epsilon^{-2} \log^3 n)$.

Theorem 1 *In time* $O((B^3/\epsilon^2) \log^3 n)$ *operations per data point we can output an* ϵ*-approximate* B*-histogram of the previous* n *points.*

However the binary search involves a cost, and if we were to use this algorithm to compute the approximate histogram for an agglomerative problem (which require one solution after seeing all points, as opposed to 1 solution on seeing every new point), the running time increases by a factor of $B/\epsilon \log^2 n$. This is an interesting tradeoff between the incremental nature and speed of the algorithm.

5 Experimental Evaluation

In this section we present the results of a detailed experimental evaluation of the application of our algorithms to the problem of performing Approximate Queries on Data Streams. We evaluate the applications of the proposed fixed window algorithm, in the case of approximate range aggregation queries on a stream, using real data sets, comparing them with other applicable methods. We demonstrate experimentally as well, that our algorithms offer histograms of superior accuracy that can be constructed efficiently. We also conducted an extensive evaluation of the applications of the proposed techniques on similarity queries and approximate queries on data warehouses. We omit detailed results on these experiments due to space limitations; we only include a short summary. In all cases we use real time series data sets extracted from operational data warehouses maintained at AT&T Labs.

[5] This time the data is $0, 0, 0, 1, 1, 1, 1, 1$

[6] Follows from the maximal property in Equation 4.

[7] The hidden constant is about 3.

Algorithm FixedWindowHistogram()
Compute $\textsc{Sum}'$ *and* $\textsc{SqSum}'$
Assume 1 *to be the first point in the circular buffer*
For k=1 to B-1 {
  Initialize k'th queue to empty
  CreateList[1,n,k]
}
For i:= end point b_ℓ *of* $B-1$*'th queue* {
  $\textsc{HError}[n,B] = \min(\textsc{HError}[n,B],$
  $\textsc{HError}[i,B-1] + \textsc{SqError}[i+1,n])$
}

CreateList[a,b,k]:=
  If $(a > b)$ *return*
  Otherwise If $(a == b)$ *insert* a *at the end of* k*'th queue*
  Otherwise {
    Compute $t = \textsc{HError}[a,k]$
    / If* $k == 1$ *then these are simply*
      $\textsc{SqError}[1,a]$ *and* $\textsc{SqError}[1,c]$ **/*
    / for* $k > 1$ *minimize over endpoints* i
      of $k-1$*'th queue */*
    Perform a binary search to find c *such*
    that $\textsc{HError}[c,k] \leq (1+\delta)t$
    and either $\textsc{HError}[c+1,k] > (1+\delta)t$ *or* $c == b$
    Insert c *at the end of* k*'th queue*
    CreateList[c+1,b,k]
  }

Figure 5. Algorithm FixedWindowHistogram

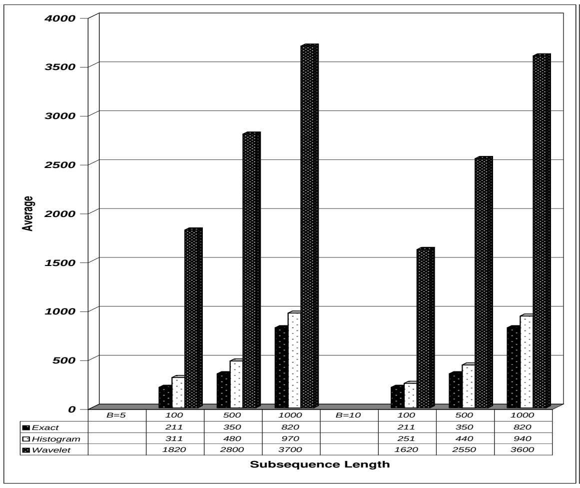
5.1 Approximate Queries on a Data Stream

In this section we evaluate the construction performance and accuracy of the fixed window algorithm in the case of approximate queries on a data stream. For our experiments we use real data sets extracted from AT&T data warehouses, representing utilization information of one of the services provided by the company. In all cases, the construction time reported corresponds to the elapsed time devoted to constructing and incrementally maintaining the data structures required from the beginning of the data stream till its end. We focus on evaluating the accuracy of range sum queries, since they are more relevant in the data stream context (similar results are obtained for range queries requesting average or point queries). Accuracy is measured by reporting the average result obtained by performing random queries; the starting points as well as the span of the queries (size of the requested aggregation range) is chosen uniformly and independently.

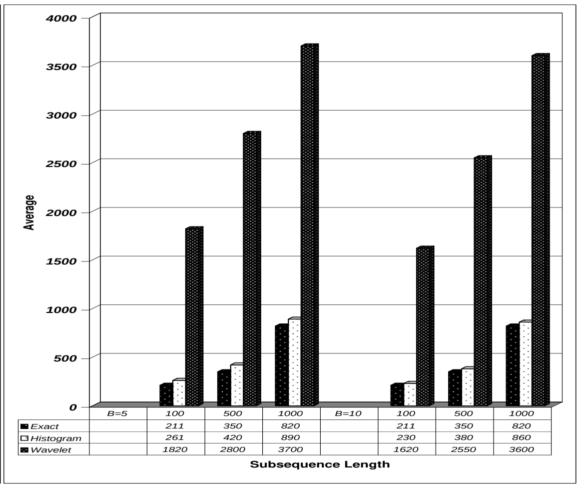
Figure 6 presents the results of an experiment using Fixed window histograms. We vary the length of the subsequence we allow in the window, the number of buckets as well as the value of ϵ. The underlying data stream consists of 1M points of real AT&T time series data and we approximate the stream using fixed window histograms, measuring both the accuracy on random queries as well as the overall elapsed time. In this case, the elapsed time corresponds to the time required to incrementally maintain a fixed window histogram as the entire data set consisting on 1M points is observed. Figures 6(a)(b) present the accuracy results for fixed window histograms. Wavelet histograms are computed again from scratch every time a new point enters and the temporally oldest point leaves the buffer. Accuracy of estimation using fixed window histograms improves with B and ϵ. The benefits in accuracy when compared with Wavelet based histograms are evident. In terms of construction time (Figures 6(c),(d)), Fixed window histograms, require more time to compute as B increases or ϵ decreases. However, the penalty is small as is evident from the figure, and the graphs are consistent with our analytical expectations. We omit the construction time performance for Wavelet since it was much worse than that of the proposed fixed window histograms (up to an order of magnitude in our experiments).

5.2 Additional Experiments

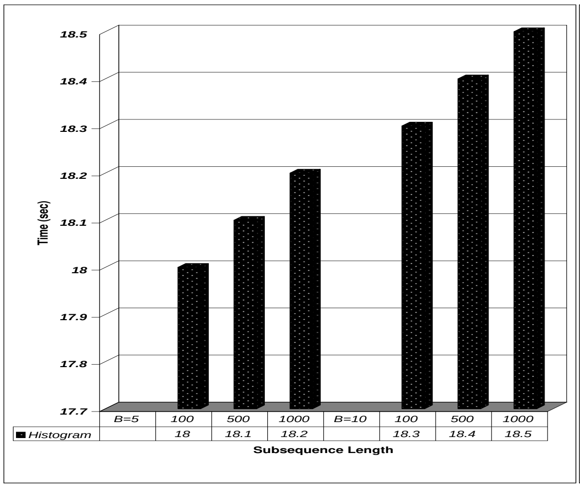
We prototyped algorithm $AgglomerativeHistogram$ and evaluated its accuracy and performance for agglomerative stream histogram construction, compared with a wavelet approach. The resulting histograms are superior both in accuracy as well as construction time. We also compared algorithm $AgglomerativeHistogram$ with the optimal histogram construction algorithm of Jagadish et. al., [JKM+98]. In these experiments we construct histograms in one database pass with algorithm $AgglomerativeHistogram$ and utilize them for answering queries approximately in a data warehouse. The resulting histograms are comparable in accuracy with those resulting from the optimal histogram construction algorithm (for various values of ϵ) and the savings in construction time are profound; these savings increase as the size of the underlying data set increases. Finally, we approximated collections of time



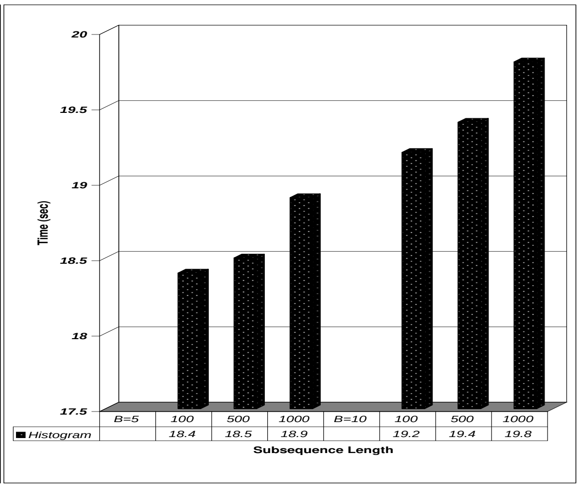
(a) $\epsilon = 0.1$



(b) $\epsilon = 0.01$



(c) $\epsilon = 0.1$



(d) $\epsilon = 0.01$

Figure 6. (a),(b) Accuracy and Estimation and (c),(d) Construction time for Fixed window Histograms, for various values of ϵ

series, using algorithms $AgglomerativeHistogram$ and $FixedWindowHistogram$ and utilized the techniques of Keogh et. al., [KCMP01] in the problem of querying collections of time series based on similarity. We experimented with both whole time series and subsequence time series matching versions of the problem. Our results, indicate that the histogram approximations resulting from our algorithms are far superior than those resulting from the APCA algorithm of Keogh et. al., [KCMP01]. The superior quality of our histograms is reflected in these problems by reducing the number of false positives during time series similarity indexing, while remaining competitive in terms of the time required to approximate the time series.

6 Conclusions

Histogram construction is a problem of central interest to many database applications. A variety of applications view data as an *infinite data sequence* rather as a finite and stored data set. In this case querying must be performed in an *online fashion*. To this end, we have proposed algorithms to incrementally approximate a data stream for querying purposes. We have shown both analytically and experimentally that our approach offers very large performance advantages and at the same time is able to attain excellent accuracy comparable with that of the optimal approach and superior to that of other applicable approaches.

Through a detailed experimental evaluation, we have demonstrated advantages of our algorithms, not only for novel applications like querying data streams, but also for traditional problems like approximate querying in warehouses and similarity searching in time series databases.

This work raises several issues for further discussion and exploration. The algorithms presented herein are fairly generic and we believe that many other problems of interest in database theory and practice can benefit from them. In particular several data mining applications can make use of the superior quality histograms our algorithms offer. The incremental nature of our algorithms, makes them applicable to mining problems in data streams. We are currently investigating these opportunities.

7 Acknowledgments

We wish to thank the anonymous referees for their comments.

References

[AGPR99] S. Acharya, P. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua Approximate Query Answering System. *Proceedings of ACM SIGMOD, Philladephia PA*, pages 574–578, June 1999.

[DH00] P. Domingos and G. Hulten. Mining High-Speed Datastreams. *Proceedings of SIGKDD*, August 2000.

[FKSV99] J. Feigenbaum, S. Kannan, M. Strauss, and M. Vishwanathan. An Approximate L^1-difference Algorithm For Massive Datasets. *Foundations of Computer Science (FOCS)*, pages 501–511, June 1999.

[FM83] F. Flajolet and G. Nigel Martin. Probabilistic Counting. *Proceedings of FOCS*, pages 76–82, June 1983.

[GK01] M. Greenwald and S. Khanna. Space-Efficient Online Computation of Quantile Summaries. *Proceedings of ACM SIGMOD, Santa Barbara*, May 2001.

[GKS01] S. Guha, N. Koudas, and K. Shim. Data Streams and Histograms. *Symposium on the Theory of Computing (STOC)*, July 2001.

[GMMO00] S. Guha, N. Mishra, R. Motwani, and L. O'callahan. Clustering Data Streams. *Foundations of Computer Science (FOCS)*, September 2000.

[Hid99] C. Hidber. Online Association Rule Mining. *Proceedings of ACM SIGMOD*, pages 145–156, June 1999.

[HRR97] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on Data Streams. *Digital Equipment Comrporation, TR-1998-011*, August 1997.

[Ind00] P. Indyk. Stable Distributions, Pseudorandom Generators, Embeddings and Data Stream Computation". *Foundations of Computer Science (FOCS)*, September 2000.

[IP95] Y. Ioannidis and Viswanath Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. *Proceedings of ACM SIGMOD, San Hose, CA*, pages 233–244, June 1995.

[JKM+98] H. V Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal Histograms with Quality Guarantees. *Proceedings of VLDB*, pages 275–286, August 1998.

[KCMP01] E. Keogh, K. Chakrabati, S. Mehrotra, and M. Pazzani. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. *Proceedings of ACM SIGMOD, Santa Barbara*, March 2001.

[LKC99] J. Lee, D. Kim, and C. Chung. Multidimensional Selectivity Estimation Using Compressed Histogram Information. *Proceedings of ACM SIGMOD*, pages 205–214, June 1999.

[MP80] J. Munro and M. Paterson. Selection and Sorting with Limited Storage. *Theoretical Computer Science*, pages 315–323, 1980.

[MVW] Y. Mattias, J. Scott Vitter, and M. Wang. Wavelet-Based Histograms for Selectivity Estimation. *Proc. of the 1998 ACM SIGMOD Intern. Conf. on Management of Data, Jun e 1998*.

[MVW00] Y. Mattias, J. S. Vitter, and M. Wang. Dynamic Maintenance of Wavelet-Based Histograms. *Proceedings of the International Conference on Very Large Databases, (VLDB), Cairo, Egypt*, pages 101–111, September 2000.

[PI97] V. Poosala and Y. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. *Proceedings of VLDB, Athens Greece*, pages 486–495, August 1997.

[SRL98] G. Singh, S. Rajagopalan, and B. Lindsay. Approximate Medians and Other Quantiles In One Pass and With Limited Memory. *Proceedings of ACM SIGMOD*, pages 426–435, June 1998.

[SRL99] G. Singh, S. Rajagopalan, and B. Lindsay. Random Sampling Techniques For Space Efficient Computation Of Large Dataset s. *Proceedings of SIGMOD, Philladelphia PA*, pages 251–262, June 1999.

[YF00] B. K. Yi and C. Faloutsos. Fast Time Sequence Indexing for Arbitrary Lp norms. *Proceedings of VLDB, Cairo Egypt*, 2000.