Fast and Accurate Computation of Equi-Depth Histograms over Data Streams

Hamid Mousavi
Computer Science Department, UCLA
Los Angeles, USA
hmousavi@cs.ucla.edu

Carlo Zaniolo
Computer Science Department, UCLA
Los Angeles, USA
zaniolo@cs.ucla.edu

ABSTRACT

Equi-depth histograms represent a fundamental synopsis widely used in both database and data stream applications, as they provide the cornerstone of many techniques such as query optimization, approximate query answering, distribution fitting, and parallel database partitioning. Equi-depth histograms try to partition a sequence of data in a way that every part has the same number of data items. In this paper, we present a new algorithm to estimate equi-depth histograms for high speed data streams over sliding windows. While many previous methods were based on quantile computations, we propose a new method called BAr Splitting Histogram (BASH) that provides an expected ϵ-approximate solution to compute the equi-depth histogram. Extensive experiments show that BASH is at least four times faster than one of the best existing approaches, while achieving similar or better accuracy and in some cases using less memory. The experimental results also indicate that BASH is more stable on data affected by frequent concept shifts.

Keywords

Data Streams, Equi-depth Histograms, Quantiles, and Sliding Windows.

1. INTRODUCTION

Data distributions are frequently needed in database and data streaming systems, however they are very large to be stored accurately. Thus, we need approximate constructs such as *histograms* which provide statistically accurate and space-efficient synopses for the distribution of very large data sets; therefore, they proved invaluable in a wide spectrum of database applications, such as: query optimization, approximate query answering, distribution fitting, parallel database partitioning, and data mining [12]. Histograms are even more important for data streaming applications, where synopses become critical to provide real-time or quasi real-time response on continuous massive streams of bursty data and to minimize the memory required to represent these massive streams. However, finding fast and light algorithms to compute and continuously update histograms represents a difficult research problem,particularly if we seek the ideal solution to compute accurate histograms by performing only one pass over the incoming data. Data streaming applications tend to focus only on the most recent data. These data can be modeled by sliding windows [4], which are often partitioned into panes, or *slides* whereby new additional results from the standing query are returned at the completion of each slide [4][14].

An important type of histograms is the *equi-depth histogram*, which is the main focus of this paper. Given a data set of single-value tuples, the B-bucket equi-depth histogram algorithm seeks to find a sorted sequence of $B-1$ boundaries over the sorted list of the tuples, such that the number of tuples between each two consecutive boundaries is approximately N/B, where N is the data set size. For the sake of time and space issues, this goal can only be achieved within a certain approximation, and because of computational constraints and requirements of continuous queries ϵ-approximations with much coarser ϵs are expected in the data stream environment. In particular, we require that only one pass be made over the incoming data and the results must be reported each time the current window slides.

The equi-depth histogram problem is obviously akin to that of quantiles [9], which seeks to identify the item that occupies a given position in a sorted list of N items: Thus given a ϕ, $0 \leq \phi \leq 1$, that describes the scaled rank of an item in the list, the quantile algorithm must return the $\lceil \phi N \rceil$'s item in the list. For example, a 0.5-quantile is simply the median. Therefore, to compute the $B-1$ boundaries of any equi-depth histograms, we could employ a quantile structure and report ϕ-quantiles for $\phi = \frac{1}{B}, \frac{2}{B}, ..., \frac{B-1}{B}$. However, this solution suffers from the following problems: (i) quantile computation algorithms over sliding windows are too slow, since they must derive more information than is needed to built the histogram [20], and (ii) quantiles algorithms used to construct histograms focus primarily on minimizing the rank error, while in equi-depth histograms we focus on the size of individual buckets and seek to minimize the bucket size error. While there has been significant previous research on ϵ-approximate quantiles over sliding windows of the data streams [15][3], their focus has mostly been on space minimization, and their resulting speed is not sufficient for streaming applications, particularly when large window sizes are needed [21]. Noting this problem, Zhang *et al.* propose a faster approach to compute the quantile over the entire history of the data stream [21], but they did not provide an algorithm for the sliding window case. Although equi-depth histograms could be implemented by executing several copies of current quantile algorithms, this approach is reasonable only when N is small. To the best of the authors' knowledge, no previous algorithm has been proposed for computing equi-depth histograms over sliding windows of data streams without using quantiles.

In this paper, we study the problem of constructing equi-depth histograms for sliding windows on data streams and propose a new algorithm that achieves average ϵ-approximation. The new algorithm, called BAr Splitting Histogram (BASH), has the following properties: BASH (i) is much faster than current techniques, (ii) does not require prior knowledge of minimum and maximum values, (iii) works for both physical and logical windows, and (iv) leaves a smaller memory footprint for small window sizes. As we shall discuss in more detail later, the main idea of BASH builds on the *Exponential Histograms* technique that was used in [6] to estimate the number of 1's in a 0-1 stream over a sliding window. More specifically, our paper makes the following contributions:

- We introduce a new expected ϵ-approximate approach to compute equi-depth histograms over sliding windows.
- We show that the expected memory usage of our approach is bounded by $O(B\frac{1}{\epsilon^2}log(\epsilon^2 W/B))$, where B is the number of buckets in the histograms and W is the average size of the window.
- We present extensive experimental comparisons with existing approaches: the results show that BASH improves speed by more than 4 times while using less memory and providing more accurate histograms.

The rest of this paper is organized as follows. Section 2 summarizes the related works. In section 3, we provide the preliminary background and definitions. We describe the BASH algorithm and its time/space complexity as well as its approximation ratio in sections 4 and 5 respectively. Section 6 presents a series of experiments results, which provide the basis for the discussion in section 7 elucidating key properties of BASH. We finally conclude the paper in section 8.

2. RELATED WORK

Perhaps, the simplest type of histograms are the traditional *equi-width* histograms, in which the input value range is subdivided into intervals (buckets) having the same width, and then the count of items in each bucket is reported. Knowing the minimum and maximum values of the data, the equi-width histograms are the easiest to implement both in databases and in data streams. However, for many practical applications, such as fitting a distribution function or optimizing queries, equi-width histograms may not provide useful enough information [8]. A better choice for these applications is an *Equi-depth* histogram [8][19] (also known as equi-height or equi-probable) in which the goal is to specify boundaries between buckets such that the number of tuples in each bucket is the same. **This type of histograms is more effective than equi-width histograms particularly for the data sets with skewed distributions. [12]**

Other types of histograms proposed in the literature include the following: (i) *V-Optimal* Histograms [10][13] that estimate the ordered input as a step-function (or pairwise linear function) with a specific number of steps, (ii) *MaxDiff* histograms [20] which aim to find the $B-1$ largest gaps (boundaries) in the sorted list of input, and (iii) *Compressed* histograms [20] which place the highest frequency values in singleton buckets and use equi-width histogram for the rest of input data. This third type can be used to construct biased histograms [5]. Although these type of histograms can be more accurate than the other histograms, they are more expensive to construct and update incrementally [11]. For example, current V-optimal algorithms perform multiple passes on the database, and are not suitable for most data stream applications [13].

Gibbons *et al.* presented a sampling-based technique for maintaining the approximate equi-depth histograms on relational databases [7]. This work is of our interest mainly because (i) it has considered a stream of updates over database, which may make the approach applicable for the data stream environment as well, and (ii) it has employed a split/merge technique to keep the histograms up-to-date, which was inspiring for us. However in their work, the stream of updates is taken from a Zipfian distribution which means some records may be updated several times, while many of the other records remain unchanged; this is not the case in a data streaming environment where records enter windows once and leave in the same order in which they arrived.

Most past work, in the area of data streams, has focused on the related problem of quantiles. It has been proven [18] that single-pass algorithms for computing exact quantiles must store all the data; thus, the goal of previous researches were to find approximate quantile algorithms having low space complexity (i.e., low memory requirements). For this reason, Manku *et al.* introduced an ϵ-approximate algorithm to answer any quantile query over the entire history of the data stream using $O(\frac{1}{\epsilon}log^2(\epsilon N))$ space [16]. Greenwald and Khanna, in [9], improved the memory usage of the previously mentioned approach to $O(\frac{1}{\epsilon}log(\epsilon N))$, and also removed the restriction of knowing the size of the stream (N) in advance. Their work has been influential, and we refer to it as the GK algorithm. For instance, GK was used in [15] and [3] to answer quantile queries over sliding windows. The algorithm proposed in [15] has space complexity $O(\frac{1}{\epsilon^2}log^2(\epsilon W))$, where W is the window size and it is unknown a priori. In [3], Arasu and Manku proposed an algorithm which needs $O(\frac{1}{\epsilon}polylog(\frac{1}{\epsilon}, W))$ space. We will refer to this algorithm as AM.

AM runs several copies of GK over the incoming data stream at different levels. At these levels, the data stream is divided into blocks of size $\epsilon W/4$, $\epsilon W/2$, ϵW, etc. Once GK computes the results for a block in a level, AM stores these results until that block expires. In this way, they can easily manage the expiration in the sliding window. To report the final quantile at each time, AM combines the sketch of the unexpired largest blocks covering the entire window. Similar to AM, the algorithm proposed in [15] also run several copies of the GK algorithm. Therefore, both of them suffer from higher time complexity, as was shown in [21]. In fact, Zhang *et al.* also proposed a faster approach for computing ϵ-approximate quantile using $O(\frac{1}{\epsilon}log^2\epsilon N)$ space. However, their approach works only for histograms that are computed on the complete history of the data stream, rather than on a sliding window.

3. PRELIMINARIES

As already mentioned, BASH is based on Exponential Histograms (EH) [5], so we will next give a brief description of this sketch technique. We then provide the formal definitions of the equi-depth histograms and error measurements.

3.1 Exponential Histogram Sketch

In [6], Datar *et al.* proposed the EH sketch algorithm for approximating the number of 1's in sliding windows of a 0-1 stream and showed that for a δ-approximation of the number of 1's in the current window, the algorithm needs $O(\frac{1}{\delta}logW)$ space, where W is the window size. The EH sketch can be modeled as an ordered list of boxes. Every box in an EH sketch basically carries on two types of information; a time interval and the number of observed 1's in that interval. We refer to the latter as the size of the box. The intervals for different boxes do not overlap and every 1 in the current window should be counted in exactly one of the boxes. Boxes are sorted based on the start time of their intervals. Here are the brief descriptions for the main operations on this sketch:

Inserting a new 1: When at time t_i a new 1 arrives, EH creates a new box with size one, sets its interval to $[t_i, t_i]$, and adds the box to the head of the list. Then the algorithm checks if the number of boxes with size one exceeds $k/2 + 2$ (where $k = \frac{1}{\delta}$), and, if so, merges the oldest two such boxes. The merge operation adds up the size of the boxes and merges their intervals. Likewise for every $i > 0$: whenever the number of boxes with size 2^i exceeds $k/2+1$, the oldest two such boxes are merged. Figure 1 illustrates how this operation works.

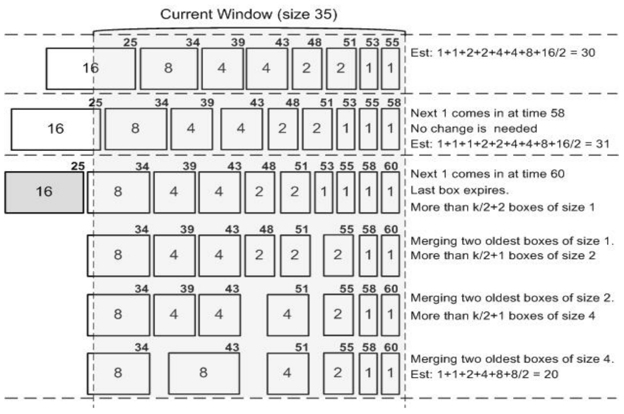


Figure 1: Incrementing an EH sketch twice at time 58 and 60. That is we have seen 1, 0, and 1 respectively at time 58, 59, and 60. (k=2 and W=35)

Expiration: The algorithm expires the last box when its interval no longer overlaps with the current window. This means that at any time, we only have one box that may contain information about some of the already expired tuples. The third row in Figure 1 shows an expiration scenario.

Count Estimation: To estimate the number of 1's, EH sums up the size of all the boxes except the oldest one, and adds half the size of the oldest box to the sum. We refer to this estimation as the count or size of the EH sketch.

It is easy to show that using only $O(\frac{1}{\delta} logW)$ space, the aforementioned estimation always gives us a δ-approximate number of 1's. It is quite fast too, since the amortized number of merges for each new 1 is only $O(1)$ (On average, less than one merge is needed for each increment in the EH sketch). It's also worth noting that instead of the counting number of 1's in a 0-1 stream, one can simply count the number of values falling within a given interval for a general stream. Thus, to construct an equi-width histogram, a copy of this sketch can be used to estimate the number of items in each interval, when the boundaries are fixed.

3.2 Definitions

Throughout this paper, we will frequently use three important terms: *boxes*, *bars*, and *buckets*. *Buckets* are the final intervals that we intend to report for the histogram. On the other hand, each bucket may contain one or more *bars*, and for each bar, the associated EH sketch has a list of *boxes* which has been described in previous section. With this clarification, we can formally define an equi-depth histogram as follows:

DEFINITION **1.** *A B-bucket equi-depth histogram of a given data set D with size N is a sequence of $B - 1$ boundaries B_1, B_2, ... $B_{B-1} \in D$, where B_i is the value of the item with rank $\lfloor \frac{i}{B} N \rfloor$ in the ordered list of items in D.*

Note that each data item (tuple)is simply considered as a floating point number. As in can be seen, the number of items between each two consecutive boundaries is the same (N/B). To ease this definition, B_i can be also defined as any values between the value of the items with rank $\lfloor \frac{i}{B} N \rfloor$ and $\lfloor \frac{i}{B} N + 1 \rfloor$. A similar definition can be used for the sliding window case:

DEFINITION **2.** *A B-bucket equi-depth histogram of a given data stream D at each window with size W is a sequence of $B - 1$ boundaries B_1, B_2, ... $B_{B-1} \in D_W$, where D_W is the set of data in the current window and B_i is the value of the item with rank $\lfloor \frac{i}{B} W \rfloor$ in the ordered list of items in D_W.*

Note that the above definitions is valid for both physical windows in which W is a fixed value and logical windows in which W may vary through the time. However, as already discussed, computing the exact equi-depth histograms is neither time efficient nor memory friendly particularly for the data streaming case [18]. Thus, we need algorithms to approximate the histogram with respect to some kind of error measurements. The most natural way to define the error is based on the difference between the ideal bucket size (i.e. (W/B) and the size of constructed buckets by any algorithms. The other way of computing this error, which is based on the error computation in quantiles, is to compute the expected error for the ranks of reported boundaries. A third approach can also be considered, which uses the differences between the ideal position (actual value) of the bucket-boundaries and the exact boundaries. Based on these three error types, the following definitions can be considered for the expected ϵ-approximate equi-depth histogram. These three error types are respectively called size error, rank error, and boundary error.

DEFINITION 3. *An equi-depth histogram summary of a window of size W is called a size-based expected ϵ-approximate summary when the expected error on the reported number of items in each bucket s_i is bounded by $\epsilon W/B$.*

DEFINITION 4. *An equi-depth histogram summary of a window of size W is called a rank-based expected ϵ-approximate if the expected error of the rank of all the reported boundary r_i is bounded by ϵW.*

DEFINITION 5. *An equi-depth histogram summary of a window of size W is called a boundary-based expected ϵ-approximate if the expected error of all the reported boundary b_i is bounded by ϵS, where S is the difference between the minimum and maximum values in the current window.*

The boundary error behaves very similar to the rank error. However, boundary error is much faster to be computed, since it deals with the value of the items instead of their ranks. Throughout this paper, we consider the first definition, because it is more appropriate for many applications that only care about the size of each individual bucket (not the position of the boundaries).

4. BAR SPLITTING HISTOGRAM (BASH)

In this section, we describe our BAr Splitting Histogram (BASH) algorithm which computes a size-based expected ϵ-approximate equi-depth histogram. For the rest of the paper, W denotes the number of items in the current window. In other words, W is the size of the most current window, whether it is physical or logical. We should also mention that, throughout this paper, we never assume that W is a fixed value as in physical windows. This implies that our approach works for both types of sliding windows. As already mentioned, B is the number of buckets in the histogram under construction. The pseudo-code for BASH is given in Algorithm 1, which is comprised of the following phases:

1. In the first phase, BASH initializes the structure, which may differ depending on whether the minimum and maximum values of the data stream is known or not. However, the general idea is to split the interval into at most $S_m = B \times p$ partitions (Bars), and assign an EH structure to each of the intervals ($p > 1$ is an expansion factor helping to provide accuracy guarantees for the algorithm, which will be discussed further in Section 5).

2. The next phase seeks to keep the size of each bar moderate. To achieve this, BASH splits bars with sizes greater than a threshold called $maxSize$. Note that $maxSize$ is proportional to current window size W, so it is not a constant value. Next subsection discusses this threshold with more details. Since the number of bars should not exceed S_m, BASH may need to merge some adjacent bars in order to make more room for the split operation.

3. Using the boundaries of the bars computed in the previous two phases, the third phase estimates the boundaries for the final buckets. This phase could be run each time the window slides or whenever the user needs to see the current results. To implement this phase, one could use a dynamic programming approach to find the optimum solution, but unfortunately this is not affordable under the time constraint in streaming environments, so we have to employ a simpler algorithm.

Algorithm 1 BASH()

```
$initialize()$;
while (true) {
  $next$ = next item in the data stream;
  find appropriate bar $bar$ for $next$;
  insert $next$ into $bar.EH$;
  $maxSize = \lceil maxCoef \times W/S_m \rceil$;
  if ($bar.count > maxSize$)
    $splitBar(bar)$;
  remove expired boxes from all EHs;
  if (window slides)
    output $computeBoundaries()$;
}
```

We will next describe these three phases in greater detail.

4.1 Bars Initialization

In many cases, the minimum and maximum possible values for the tuples in the stream are known or can be estimated. To initialize the structure for such cases, the $initialize()$ function in Algorithm 1 partitions the interval between the minimum and the maximum into $S_m = B \times p$ ($p > 1$) equal segments, that we call bars; an empty EH sketch is created for each segment. As it will be shown later, the value of our expansion factor does not need to be much larger than one. In practice, our experiments show that it is sufficient to set p value less than 10.

For those cases where there is no prior knowledge of the minimum and maximum values, $initialize()$ starts with a single bar with an empty EH. Once the first input, say $next$, comes in, the method sets the interval for this bar to $[next, next]$, and increments its EH by one. The algorithm keeps splitting the bars and expanding the interval of the first and last bars as more data are received from the stream. In this way, we can also preserve the minimum and maximum values of the tuples at each point in time.

The other critical step in Algorithm 1 is the split operation $splitBar()$. This operation is triggered when the size of a bar is larger than $maxSize = \lceil maxCoef \times W/S_m \rceil$, where $maxCoef$ is a constant factor. Obviously, $maxCoef$ should be greater than 1 since the ideal size of each bar is W/S_m and we need the $maxSize$ to be larger than W/S_m. On the other hand, after splitting a bar into two bars, the size of each bar should be less than or equal to the ideal size of bars. This implies that $maxCoef \leq 2$. This value is empirically determined to be approximately 1.7. Smaller values usually cause more splits and affects the performance, while larger values affect the accuracy since larger bars are more likely to be generated. Observe that as the window size changes, $maxSize$ also changes dynamically. This in turn implies that BASH starts splitting bars very early, when the window is empty and first starts to grow, because at that point $maxSize$ is also quite small. Starting the process of splitting the bars early is crucial to assure that accurate results are obtained from early on in the execution of the algorithm. This dynamicity also helps BASH to easily adapt to the changes in window size W for logical (i.e., time-based) windows.

Example 1: Assume that we want to compute a 3-Bucket histogram ($p = 2$, $k = 2$, and $W = 100$) and the data stream starts with 10, 123, 15, 98, ... Thus, the initialization phase starts with a single bar $B_1 = (1, [10, 10])$, which means the bar size and its interval are respectively 1 and $[10, 10]$. Once second data item, 123, enters, BASH inserts it into B_1, so $B_1 = (2, [10, 123])$. However, at this point B_1's size is larger than $maxSize$ ($\lceil 1.7 \times 2/6 \rceil$) and it should be split, so we will have two bars $B_1 = (1, [10, 66])$ and $B_2 = (1, [66, 123])$. Next data item, 15, goes to B_1, so $B_1 = (2, [10, 66])$. Again, B_1's size is larger than $maxSize$ ($\lceil 1.7 \times 3/6 \rceil$), so BASH splits it and now we have three bars: $B_1 = (1, [10, 38])$, $B_2 = (1, [38, 66])$ and $B_3 = (1, [66, 123])$. By repeating this approach for each incoming input, we obtain a sequence of non-overlapping bars covering the entire interval between the current minimum and maximum values in the window.

4.2 Merging and Splitting Operation

As already mentioned, for each new incoming tuple, the algorithm finds the corresponding bar and increments the EH sketch associated with that bar as, explained in section 3. However, the most important part of the algorithm is to keep the number of tuples in each bar below $maxSize$, allowing for a more accurate estimate of the final buckets. In order to do so, every time the size of a bar gets larger than $maxSize$, BASH splits it into two smaller bars. However, we might have to merge two other bars in order to keep the total number of bars less than S_m. This is necessary to control the memory usage of the algorithm. Due to the structure of the algorithms, it is easier to start with the merging method.

4.2.1 Merging

In order to merge the EH sketches of the pair of selected bars, the EH with the smaller number of boxes is set to *blocked*. BASH stops incrementing blocked bars, but continues removing expired boxes from them. In addition, the interval of the other bar in the pair is merged with the interval of the blocked bar, and all the new tuples for this merged interval are inserted into this bar. Algorithm 2 describes this in greater detail. The details on how the bars are chosen for merging will be discussed later in this section.

When running Algorithm 2, every bar may have one or more blocked bars attached to it. BASH keeps checking that the size of the actual bar is always bigger than those of its blocked bars, and whenever this is no longer the case, BASH switches the actual bar with the blocked bar of larger size. Although this case is rare, it can occur when the boxes of the actual bar expire and the bar length decreases, while the blocked EH has not been changed.

Also, observe that every bar might have more than one blocked bar associated with it. Consider the case where we have to merge two adjacent bars which already have an associated blocked bar. To merge these two, we follow the same general approach: the longest EH is picked and all other bars (blocked or not) are considered as the blocked bars of the selected (actual) bar. In the next section, we will show that the expected number of such blocked bars is S_m.

Algorithm 2 mergeBars()

```
if (! findBarsToMerge($X_l$, $X_r$)) return false;
Initialize new bar $X$;
if ($EH_{X_l}$.BoxNo $\geq$ $EH_{X_r}$.BoxNo){
  $EH_X$ = $EH_{X_r}$;
  Add $EH_{X_l}$ to the blocked bars list of $EH_X$;
} else {
  $EH_X$ = $EH_{X_l}$;
  Add $EH_{X_r}$ to the blocked bars list of $EH_X$;
}
Add the blocked bars of both $EH_{X_l}$
  and $EH_{X_r}$ to the blocked bars list of $EH_X$;
$X$.start = $X_l$.start;
$X$.end = $X_r$.end;
Remove $X_r$ and $X_l$ from the bars list;
Add bars $X$ and to the bars list;
return true;
```

4.2.2 Splitting

When the size of a bar, say X, exceeds the maximum threshold $maxSize$ as shown in Algorithm 1, we split it into two smaller bars according to Algorithm 3. Before the split operation takes place, we make sure that the number of bars is less than S_m. If this is not the case, as already explained, we would try to merge two other small adjacent bars. After X is determined to be appropriate for splitting, we divide the interval being covered by X into a pair of intervals and assign a new EH sketch to each of them. Let us call these new bars X_l and X_r. X must be split in such a way that the size (count) of X_l and X_r remain equal. To this end, the algorithm first tries to distribute the blocked bars of X, denoted as BB_X, into the blocked bars of each of the new bars. The splitting algorithms tries to do this distribution as evenly as possible; however, the size (count) of bars X_l and X_r may not be equal after running this step. Thus, BASH splits the actual bar of X (EH_X) accordingly to compensate for this difference. It is easy to show that it is always possible to compensate for this difference using an appropriate split on EH_X, since the size of the actual bar is guaranteed to be greater than the size of each its blocked bars.

To split the original EH sketch, EH_X, into a pair of sketches, BASH first computes the split ratio denoted as $Ratio$ in Algorithm 3. $Ratio$ simply indicates that to compensate for the difference between the size of the two new bars, the aggregate size of boxes going from EH_X to EH_{X_r} after splitting should be $Ratio$ times the size of EH_X. In order to have such a split ratio, BASH puts half of EH_X boxes with size one into EH_{X_l}, and the other half into EH_{X_r}. Then, it replaces each of the remaining boxes in EH_X with two boxes that are half the size of the original box. Finally, based on the current ratio of the sizes, BASH decides whether to put each copy to EH_{X_r} or EH_{X_l}. In other words, if the current ratio of the size of EH_{X_r} to the aggregate size of EH_{X_r} and EH_{X_l} is smaller than $Ratio$ the copy will go to EH_{X_r} otherwise it will go to EH_{X_l}.

Example 2: Let us get back to our running example. Assume we have the following six bars at some point: B_1=$(21, [1, 43])$, B_2=$(14, [43, 59])$, B_3 = $(10, [59, 113])$, B_4=$(9, [113, 120])$, B_5 =$(29, [120, 216])$, and B_6 = $(15, [216, 233])$, and the next input is 178. BASH inserts this input into B_5, and now B_5's size is 30 which is larger than $maxSize$=$\lceil 1.7 \times 100/6 \rceil$=29. Thus, B_5 should be split. However, since we already have six ($p \times B$) bars, we need to find two candidates for merging before we can split B_5. As the bars' size offers B_3 and B_4 are the best candidates, since after merging them the aggregate size is still smaller than $maxSize$. Therefore, after this phase is done the following bars would be in the system (note that for simplicity, we assumed that nothing expires in this phase): B_1=$(21, [1, 43])$, B_2=$(14, [43, 59])$, B_3=$(19, [59, 120])$, B_4=$(15, [120, 168])$, B_5= $(15, [168, 216])$, and B_6=$(15, [216, 233])$.

Algorithm 3 splitBars(X)

```
if ($curBarNo$ == $S_m$ && !$mergeBars$()) return;
Initialize new bars $X_l$ and $X_r$;
$l = 0$;
//distributing blocked bars of $X$ between the new bars
$BBSize$ = Aggregate Size of the blocked bars;
for each (blocked bar $bar$ in $BB_X$) {
  if ($l$ + $bar$.size < $BBSize$/2) {
    $l$ += $bar$.size;
    Add $bar$ to $BB_{X_l}$;
  } else
    Add $bar$ to $BB_{X_r}$;
}
$EH_{X_l}$.start = $EH_X$.start;
$EH_{X_l}$.end = ($EH_X$.start + $EH_X$.end)/2;
$EH_{X_r}$.start = ($EH_X$.start + $EH_X$.end)/2;
$EH_{X_r}$.end = $EH_X$.end;
$Ratio$ = (($X$.size/2)-$l$)/($X$.size - $BBSize$);
foreach (box $box$ in $EH_X$) {
  if ($box$.size == 1)
    Alternatively add a copy of $box$ to $EH_{X_l}$ or $EH_{X_r}$;
  else {
    $box$.size = $box$.size/2;
    for (2 times)
      if ($EH_{X_r}$.size /($EH_{X_r}$.size+$EH_{X_l}$.size) < $Ratio$)
        Add a copy of $box$ to $EH_{X_r}$;
      else
        Add a copy of $box$ to $EH_{X_l}$;
  }
}
Remove $X$ from the bars list;
Add bars $X_l$ and $X_r$ to the bars list;
```

4.2.3 Which Bars to Merge

To select two bars for merging, we first look for any two empty adjacent bars. If there is no such pair, we look for an empty bar and merge it with the smaller of its two neighbors. When there is no empty bar, then we find the two adjacent bars that have the minimum aggregate size. If this aggregate size is less than $maxSize$ (which is usually the case), we select them for merging. Otherwise, we do not perform any merge operations until some boxes expire, since we do not want to create bars with size greater than $maxSize$. Although bars may be initially empty or become empty due to the expiration of their boxes, the case in which the bars are not is obviously the most common one.

4.2.4 An Alternative Merging Approach

Although the aforementioned approach guarantees the approximation ratio, it is using blocked bars which may increase the memory requirement and as a result influence performance. As an alternative approach, one can use the following method, wherein the idea is to merge the boxes of the two bars, and make a new EH. The former technique is called BASH-BL, since it needs to deal with BLocked bars, and this alternative approach is referred to as

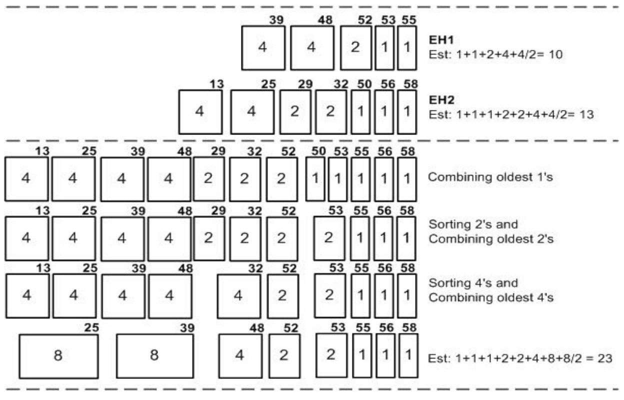


Figure 2: Merging two EH sketches EH1 and EH2. (k=2)

BASH-AL thorough the rest of the paper.

To merge two EH sketches, we start by selecting boxes with size one from both EH sketches, and mingle them into a list sorted by the start time of their intervals. If the number of such boxes is more than $k/2+2$ ($k = 1/\delta$), we keep combining the oldest boxes in this list until the number of boxes with size one is smaller than $k/2+2$. The same steps would be followed for boxes with size two. We compile all boxes of size two, sort them based on their start times, and if the number of these boxes is more than $k/2 + 1$, we combine the oldest ones until the number of such boxes is smaller than or equal to $k/2 + 1$. Note that while merging boxes with size two, we may be using some boxes from the first EH, some from the second EH, and some boxes generated from combining boxes in the previous iterations. This approach is then repeated for boxes with larger size (4, 8, etc.). Figure 2 illustrates an example of merging two EH sketches where $k = 2$ using this alternative method.

Although, we do not have any blocked bars, the same split approach, mentioned earlier, can be used for this alternative merging technique. This time, the splitting operation in Algorithm 3 splits the only existing EH into two EHs of equal size.

4.3 Computing Final Buckets

So far, we showed how the intervals can be partitioned into S_m parts in a way that each part contains no more than $maxSize$ items. The last phase is to report the final buckets or boundaries of the buckets. As already noted, a dynamic programming approach cannot be used to see which bars should go into each bucket. This is due to the quadratic delay of the dynamic approach that makes it completely inapplicable. Instead, we use a linear approach as shown in Algorithm 4.

First, this algorithm computes the total number of estimated tuples in the current window by adding up the estimated number of tuples in each bar. Let us call it $total_{Est}$, which is not necessarily equal to W[1]. Therefore, the expected number of tuples in each bucket should be $idealBuckSize = total_{Est}/B$. Thus, we start from the first bar and keep summing the estimated number of items in each bar until the sum exceeds $idealBuckSize$. At this point, we report the appropriate boundary for the current bucket as shown in Algorithm 4 and continue with the next bucket.

5. THEORETICAL RESULTS

[1]Note that the expected value of $total_{Est}$ is W, but at each point in time these two may have different values.

Algorithm 4 computeBoundaries()

```
$b = -1$; //An index over bars list
$count = 0$;
$idealBuckSize = total_{Est}/B$;
for (int $i = 0$; $i < B$; $i$++) {
  while ($count \leq idealBuckSize$) {
    $b$ ++;
    $count$ += $bars[b].count$;
  }
  $surplus = count - idealBuckSize$;
  $boundaries[i] = bars[b].start +$
    $bars[b].length * (1 - surplus/bars[b].count)$
  $count = surplus$;
}
return $boundaries$;
```

In this section, we provide the theoretical proof of the approximation ratio, time complexity, and space usage of the BASH algorithm. All the proofs are provided for the version of BASH which uses blocking to merge the bars (BASH-BL). Similar proofs can be provided for the other version as well.

5.1 Approximation Analysis

First, we start by proving that the splitting operation does not change the expected error of the estimations for the number of tuples in each new bar. Assume bar X is going to be split. Let the actual number of items in X be N_X, while the sketch has estimated this number to be n_X. Because of the EH sketch approximation ratio, we know:

$$(1-\delta)N_X \leq n_X \leq (1+\delta)N_X \tag{1}$$

After splitting X into two smaller bars, say X_l and X_r, our estimation for each of these new bars is $n_X/2$, because of the way the split operation works. Let the actual number of items in X_l be k_l. The expected error for this new bar (X_l) is computed as the average error over all possible values for k_l. Let $P\{k_l = k\}$ be the probability that the size of bar X_l is k ($0 \leq k \leq N_X$). It is then easy to see that:

$$P\{k_l = k\} = \binom{N_X}{k}(1/2)^{N_X} \tag{2}$$

Without loss of generality, assume $n_x \leq N_X$ and N_X is an even number. Thus:

$$\begin{aligned}
E\{err(X_l)\} &= \sum_{k=0}^{N_X} |\frac{n_X}{2} - k| P\{k_l = k\} \\
&= \sum_{k=0}^{n_X/2} (\frac{n_X}{2} - k) P\{k_l = k\} + \sum_{k=n_X/2+1}^{N_X} (k - \frac{n_X}{2}) P\{k_l = k\} \\
&= 2\sum_{k=0}^{n_X/2} (\frac{n_X}{2} - k) P\{k_l = k\} + \sum_{k=0}^{N_X} (k - \frac{n_X}{2}) P\{k_l = k\} \\
&= 2\sum_{k=0}^{N_X/2} (\frac{n_X}{2} - k) P\{k_l = k\} + \frac{N_X}{2} - \frac{n_X}{2} \qquad (3)
\end{aligned}$$

Now, we show that the first part of the last equation is negligible with respect to $N_X/2 - n_X/2$.

$$\sum_{k=0}^{n_X/2}(\frac{n_X}{2}-k)P\{k_l=k\} \leq \sum_{k=0}^{n_X/2}(\frac{N_X}{2}-k)P\{k_l=k\}$$
$$\leq \sum_{k=0}^{N_X/2}(\frac{N_X}{2}-k)P\{k_l=k\}$$

$$= \frac{N_X}{2}\sum_{k=0}^{N_X/2}P\{k_l=k\} - \sum_{k=0}^{N_X/2}kP\{k_l=k\}$$
$$= \frac{N_X}{2}\frac{1}{2}(1+P\{k_l=\frac{N_X}{2}\}) - \sum_{k=1}^{N_X/2}\frac{N_X}{2}P'\{k_l=k-1\}$$
$$= \frac{N_X}{4}(1+P\{k_l=\frac{N_X}{2}\}) - \frac{N_X}{2}\sum_{k=0}^{N_X/2-1}P'\{k_l=k\}$$
$$= \frac{N_X}{4}(1+P\{k_l=\frac{N_X}{2}\}) - \frac{N_X}{2}\frac{1}{2}$$
$$= \frac{N_X}{4}\binom{N_X}{N_X/2}(\frac{1}{2})^{N_X} \sim \frac{1}{4}\sqrt{\frac{N_X}{\pi}} \quad (4)$$

The last part is because of the direct application of Sterling's formula and $P'\{k_l = k\}$ is the probability of having k items in the left bar when you split a bar with size $N_X - 1$. Combining inequality (4) with equation (3), and using inequality (1), we can conclude that:

$$E\{err(X_l)\} = \frac{N_X}{2} - \frac{n_X}{2} + \frac{2}{4}\sqrt{\frac{N_X}{\pi}}$$
$$\leq \delta\frac{N_X}{2} + \frac{1}{2}\sqrt{\frac{N_X}{\pi}}$$
$$\leq (\delta + \frac{1}{\sqrt{\pi N_X}})\frac{N_X}{2} \quad (5)$$

Note that the second part of this error shows that no matter how accurate the estimation of the size of bar X is, when we split it, we will add $1/\sqrt{\pi N_X}$ error rate on average. Fortunately in practice, this increase in error is negligible since N_X is very large. Moreover, later we show that the number of split operations is bounded such that at the next split on the same bar, this extra error will be vanished due to the expiration of old and inaccurate boxes. Thus in practice, the expected δ-approximation still holds for this new bar. This also holds for $E\{err(X_r)\}$ symmetrically. Thus, we can state that the expected error does not change by splitting a bar. Moreover, the merge operation does not change this error, since the EH sketch of the merged bars does not change at all. These two facts lead us to the following lemma:

LEMMA 1. *At any moment in time, our sketch contains an expected δ-approximation of the number of tuples in each bar. Also, these estimations are not greater than $maxSize$.*

THEOREM 1. *For any given $0 < \epsilon < 1$, algorithm computeBoundaries() provides a size-based expected ϵ-approximate equi-depth histogram for sliding windows on data streams.*

PROOF. We need to show that the estimated number of tuples in each bucket is bounded by $\epsilon W/B$ on average. Remember that, on average every bucket consists of $S_m/B = p$ bars. According to Lemma 1, every bar also has an expected δ-approximate number of tuples in it. On the other hand, the first and last bars should be treated differently, because we may only consider a fraction of them in the bucket. Thus, in total, we would have δ error for the size estimation of the bars and $2 \times 1/p$ for the bars at the two ends of the bucket. The latter is because the entire counted number from those two bars may be on the wrong side of the boundaries we have selected. This proves that the expected total error is bounded by $\delta + 2/p$. Calling this bound ϵ, one can say that by setting $\delta = \epsilon/2$ ($k = 2/\epsilon$) and $p = 4/\epsilon$, we obtain a size-based expected ϵ-approximate equi-depth histogram. □

The proof, we provided here shows that the expected approximation error is bounded, which is enough for our purpose even though better bounds may hold in theory.

5.2 Space Complexity Analysis

To analyze the space usage of our approach, we first prove the following lemma showing that on average, the total number of blocked bars is limited.

LEMMA 2. *At any moment in time, the BASH algorithm on average generates $O(S_m)$ number of blocked bars.*

PROOF. The algorithm only performs a merge operation when it has to split a bar and there is no room left for a new bar. Thus, the number of merge operations is less than the number of split operations. Moreover, for each split operation on a bar, $maxSize/2$ tuples must have been inserted into the bar since the last split on the bar. This means that the average number of splits and consequently the average number of merges is $2 \times W/maxSize$. By selecting $maxSize$ to be $O(W/S_m)$, e.g., $maxCoef \times W/S_m$ which is less than twice the ideal size for each bar, we can conclude that the average number of split operation and as a result the average number of merge operations in each window is $O(S_m)$.

To show that the average number of blocked bars in the current window is $O(S_m)$, first consider that there is no switch between any actual bar and one of its larger blocked bars. For this case, every blocked bar stays in the system, until the whole window expires. This is because when a bar is blocked, it does not get incremented anymore and after the window slides for W tuples, all its boxes will be expired. This basically means the average number of blocked bars is the same as the average number of splits. Now, consider the case that we need to switch an actual bar with one of its blocked bars which has a larger size. We show that on average the number of such switches are constant, thus the average number of blocked bars is still $O(S_m)$. Let q be the probability of switching an actual bar with one of its blocked bars at any time between two consecutive split operations over that actual bar. Also, observe that the size of blocked bars are always decreasing, and on average, the rate of expiring from both types of bars are the same. Thus, blocked bars sizes are decreasing at a faster rate than those of actual bars, since we may insert new items into the actual bars. This basically means $q < 1/2$. Keeping this in mind, we also know that in the mentioned period, the probability of having one switch is q, the probability of having two switches is q^2, etc. Thus, the expected number of switches is $\Sigma iq^i \leq q/(1-q)^2$, which in turn is smaller than 2 for $q < 1/2$. This completes our proof. □

Based on the above lemma, we can now compute the average space used by BASH. On the one hand, each EH sketch for counting X items in an sliding window with δ-approximate ($\delta = \epsilon/2$) accuracy needs $O(\frac{1}{\delta}log(\delta X))$ or equivalently $O(\frac{1}{\epsilon}log(\epsilon X))$ of space. On the other hand, the total number of bars in the system is $O(S_m)$ for the worst case (Lemma 2). Thus the total memory used is $O(S_m \frac{1}{\epsilon} log(\epsilon \times maxSize))$ or $O(B \frac{1}{\epsilon^2} log(\epsilon^2 W/B))$. This lead us to the following theorem:

THEOREM 2. *The BASH algorithm computes the set of $B-1$ boundaries of the expected ϵ-approximate equi-depth Histogram on a data stream using $O(B \frac{1}{\epsilon^2} log(\epsilon^2 W/B))$ space, where W is the sliding window size.*

Observe that this bound is computed for the worst case scenario. As previously mentioned, we do not need to set k $(1/\delta)$) and p to very high values in practice. Consequently, BASH practically needs even less memory space than the existing approaches.

5.3 Time Complexity Analysis

We will compute the expected computational cost of BASH per tuple assuming that $k = 2$: for larger k's the algorithm is faster at the cost of using more memory space. When the next data value, $next$, comes in, the cost of the various steps in Algorithm 1 can be estimated as follows:

- Finding the appropriate bar, $curBar$, for $next$ to be inserted into takes $log(S_m)$ using a simple binary search.
- Incrementing the EH sketch of $curBar$ needs one box-combining operation on the average. To see why this is $O(1)$, observe that at most half of the increments cause two size-one boxes to get merged. From this half, half of them result in combining two boxes of size 2, and so on and so forth. Therefore, in total, for 2^i increments we need 2^{i-1} merges on boxes of size 1, 2^{i-2} of merges on box of size 2, ..., and 1 merge for boxes of size 2^{i-1}. Since this adds up to $2^i - 1$, each increment requires an average $(2^i - 1)/2^i \approx 1$ combining operations.
- The probability of needing a split operation is $1/S_m$, and to split a bar we only need to go through the box list of EH once; this takes $O(log(maxSize))$. Therefore, the average cost of a split operation is $O(log(maxSize))$ /S_m, which is practically a constant.
- The merge operation itself needs a constant amount of time, but to find the best bars to merge, we need to go through the entire list of bars of length S_m. Multiplying this by the probability of needing to merge $(1/S_m)$, we conclude that the average cost for this operation is also constant.
- Computing the final boundaries requires visiting all bars $(O(S_m))$. Fortunately, we do not need to report the results for each new incoming data value. Instead, we can report the results each time the window slides (S): this reduces the complexity to $O(S_m/S)$.
- The naive implementation of the expiration phase checks the last box of each EH $(O(S_m))$. This time can be reduced to $O(log(S_m))$ using a simple heap tree structure. To keep the implementation simple, we perform the expiration phase a constant number of times in each slide: this can lower the time complexity to $O(S_m/S)$.

Considering these computations, we can conclude that the total time complexity per data item would be $O(log(S_m) + S_m/S)$ = $O(log(B \times p) + (B \times p)/S)$, where S is the slide size. In practice, this time complexity would be very close to a constant time, since S is usually much larger than S_m, and $log(S_m)$ is usually very small.

Table 1: Data sets used for the experimental results

Name	Distribution	Size	Shifts	Parameters
DS1	Uniform	$1m$	0	$min0, max10k$
DS2	Normal	$1m$	0	$\mu5k, \sigma2k$
DS3	Normal	$1m$	0	$\mu5k, \sigma50$
DS4	Normal	$1m$	0	$\mu5k, \sigma20$
DS5	Normal	$1m$	0	$\mu5k, \sigma500$
DS6	Zipfian	$1m$	0	$\alpha1.1$
DS7	Zipfian	$1m$	0	$\alpha1.5$
DS8	Exponential	$1m$	0	$\lambda10^{-2}$
DS9	Exponential	$1m$	0	$\lambda10^{-3}$
DS10	Exponential	$1m$	0	$\lambda10^{-4}$
DS11	Normal	$1m$	100	$\mu10k$-$1k$, $\sigma50$
DS12	Normal	$1m$	100	$\mu15k$-$5k$, $\sigma200$
DS13	Normal	$1m$	$1k$	$\mu15k$-$5k$, $\sigma200$
DS14	Normal	$10m$	$1k$	$\mu15k$-$5k$, $\sigma500$
DS15	Normal	$10m$	$1k$	$\mu15k$-$5k$, $\sigma50$
DS16	Mix	$10m$	100	$\mu5k$, $\sigma200$, $\lambda10^{-3}$
DS17	Mix	$10m$	$1k$	$\mu5k$, $\sigma200$, $\lambda10^{-3}$
DS18	Poker-Hands	$10m$	0	-
DS19	S&P500	$165k$	-	-
DS20	Expanded S&P500	$1.03m$	-	-

6. EXPERIMENTAL RESULTS

In order to evaluate the performance of the proposed algorithms, we have implemented both versions of the BASH algorithm using the C++ programming language. The version of BASH which blocks the bars in the merge operation is referred to as BASH-BL, and the alternative approach which mixes the boxes of the EHs into a single EH at the merge time is called BASH-AL. In addition to BASH, we have also implemented the AM [3] algorithm in the exact same environment. To be able to compare the resulting histograms from each of the mentioned algorithms, we used a simple approach which computes the exact boundaries of the equi-depth histograms. It is worthy to note that for some cases this approach needs several hours to compute the exact histograms.

For all the experiments shown in this section, we have fixed the slide size at 100 tuples. To Make the error diagrams smoother, the average error for every 20000 tuples is shown at each point. No prior knowledge of the minimum and maximum values of the incoming data is considered. $maxSize$ and B are also set to $1.7 \times W/S_m$ and 20 respectively. Similar results hold for different values of B, which are eliminated due to space limitations.[2]

Most of the comparisons included in this part are performed with $\epsilon = 0.01$ for AM algorithms. The experimental results show that to obtain this error rate for window sizes larger than 100k, it is almost always sufficient to set δ to 0.1 and p to 7. Moreover, if one wants to compete with AM with an error rate of 0.025, setting δ to $1/8$ and p to 4 would always provide more accurate results in BASH than AM. All the experiments are run on a 64bit, 2.27 GHz machine running CentOS with 4GB of main memory (RAM) and 8MB of cache.

6.1 Data sets

The data sets used for evaluating the results can be categorized into two main parts: synthesized data sets and real-world data sets. Table 1 summarizes some information about these data sets.

Synthesized data sets: We have used a number of different synthesized data sets in our experiments:

- Uniform, Normal, Zipfian, and Exponential distributions with different settings for their parameters and without any con-

[2]For more results, see our more extensive report in [17].

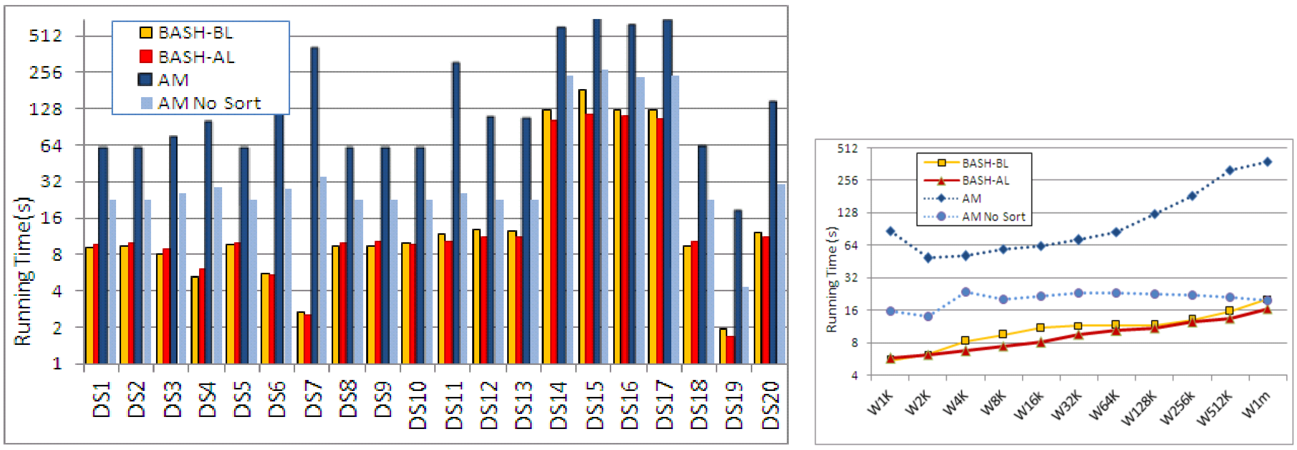


Figure 3: a) Execution time for all data sets ($W = 100k$), and b) The effect of changing window size from 1k to 1m on the execution time on DS13. ($k = 10$, $p = 7$, and $\epsilon = 0.01$.)

cept shift (DS1 through DS10).

- Normal distributions with many shifts on the value of the mean parameter. (DS11 to DS15)
- Random shifts on random distributions (DS16 and DS17): Just as in the previous data sets, the distribution of data in these data sets encounters several shifts. However, this time at each shift, we randomly select between normal and exponential distributions. If the choice is the normal distribution, with probability 0.5, we increase the mean by 50, otherwise we decrease it by 50. In the same way, we change the standard deviation by ± 5. On the other hand, if the exponential distribution is selected at the current shift, the rate parameter (λ) is randomly updated by $\pm 10\%$. The initial values of the parameters are shown in Table 1.

Real-world data set: One of the most important applications of the histograms is for those cases in which the distribution of the data is unknown or can not be simply modeled. To evaluate the performance of BASH algorithms over this kind of data, we have considered the following real-world data sets:

- *Poker-hands (DS18)[2]*: Each row in this data set contains 10 values ranging from 1 to 12. For this data set, we have computed the histogram over the multiplication of all these values in each row. Note that this data set contains no concept shift, since the probability of each outcome at each point is computable.
- *S&P500 (DS19)[1]*: This data set contains minute-by-minute prices of the S&P500 index starting from January 29, 2010 and ending on August 13, 2010. Each record contains the highest, lowest and last prices of the index in the corresponding minute.
- *Expanded S&P500 (DS20)*: For each row of the previous data set, we generated 17 more values with a normal distribution such that 95% of them are between the lowest and highest prices of the corresponding row. Combined with the existing three prices in each row, this makes a data set with more than 1.03 million records.

6.2 Timing Results

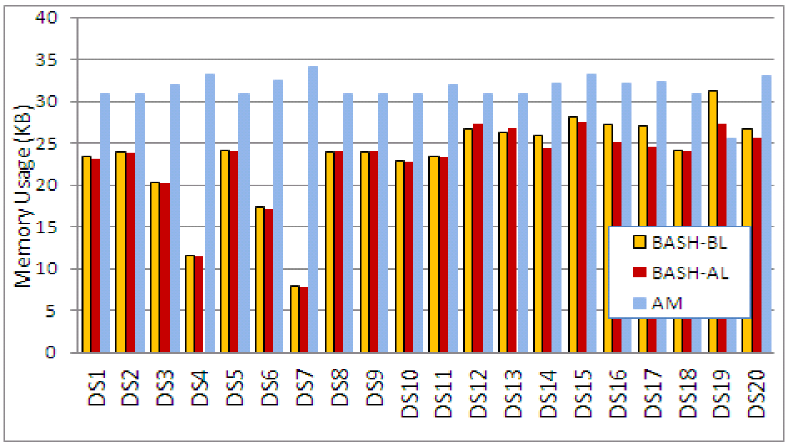
For each of the data sets introduced in Table 1, the running time of the three algorithms with $\epsilon = 0.01$ and $W = 100k$ is shown in part a of Figure 3. Similar results are obtained for different window sizes and ϵ which is not shown in this paper due to space limitations. For further results, readers are referred to our extensive report [17].

Based on the timing results in part a of Figure 3, we can state that both BASH-BL and BASH-AL are at least four times faster than AM in almost all the data sets, while also providing more accurate histograms which are discussed later in this section. This improvement on execution time makes BASH much more applicable than AM. particularly for larger windows (or equivalently faster data streams.) As an example, consider DS16 in which AM and BASH-AL (BASH-BL) respectively spend 607.56 and 111.79 (124.61) seconds to compute the results. Another interpretation of these timings would be that AM and BASH-AL (BASH-BL) can respectively support 16549 and 89453 (80250) data items per second with a window size of $100k$. Therefore, AM may not be practical for many data streaming systems, while BASH is performing properly. In the next few paragraphs, we show that the problem of high delay of the AM approach gets even worse with larger window sizes.

As already mentioned, AM runs several copies of the GK algorithm and employs a sorting algorithm over the output of these copies to generate the final quantiles. Thus, the timing performance of AM is hugely dependent on its sorting algorithm. In our implementation of AM, we have used quick sort, since it is one of the fastest sorting algorithms in practice. However, we have also included the running time of AM without considering the time it spends on sorting (bars titled as "AM No Sort" in part a of Figure 3). Interestingly, even after eliminating the sorting time from AM, both BASH-BL and BASH-AL are still faster. Another important factor affecting the performance of the mentioned algorithms is the window size W. Part b of Figure 3 sketches how the window size affects the response time. These results are taken over data set DS13. The delay time for all the three approaches increases with respect to the window size; however, the growth rate of AM is greater than the other two. It is worthy to mention that putting the sorting time aside, the rest of the AM algorithm still needs more time than BASH algorithms for all window sizes.

6.3 Space Usage Results

The diagram in part a of Figure 4 compares the space usage of the three algorithms for a window size of $W = 100k$. In this diagram, for each of the data sets, we have provided the size of the main structures in the algorithms. For all the cases (except DS19), both BASH-BL and BASH-AL need at least 20% less space than



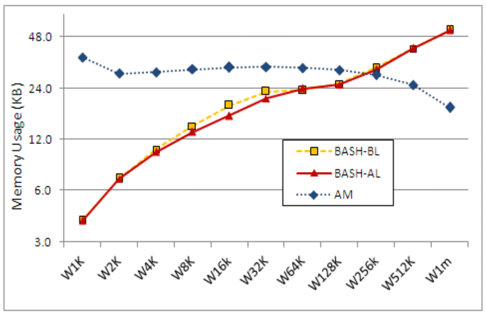


Figure 4: Space usage of the algorithms for a) all the data sets ($W = 100k$), and for b) different window sizes on DS13. ($k = 10$, $p = 7$, and $\epsilon = 0.01$.)

AM. However, BASH-BL uses slightly more space than BASH-AL. The reason that BASH algorithms need more memory than AM for DS19 is the small size of this data set (165K) with respect to the window size (100K). This is due to the fact that the BASH algorithms are in their initialization phase, during which reaching convergence might use more memory than other algorithms. It is also worth mentioning that the smaller the variance of data values, the less the memory usage for both BASH algorithms.

In part b of Figure 4, the diagram illustrates the space usage of the three algorithms for different window sizes. Both BASH algorithms use almost the same amount of memory in each window size, and that amount increases logarithmically with respect to the window size. On the other hand, memory usage of AM is almost constant for different window sizes, and it even decreases for very large window sizes. Therefore, for the window sizes larger than 256K, AM starts performing better than BASH algorithms in terms of space. This suggests that there are situations where AM performs better than BASHs with respect to memory usage. However, we also need to keep in mind the following two important points: First, as shown in the two diagrams of Figure 5, the histograms generated by either of the BASH algorithms over windows with size larger than 256K are at least three times more accurate than the ones computed by AM. Second, for all the cases that the memory usage of BASH algorithms is worse than AM, the running time of AM is at least 8 times worse than those of BASHs. Nevertheless, for larger window sizes, we can use smaller values for parameters k and p, and reduce the memory usage while still providing faster and more accurate histograms than AM. The current setting of k and p values is best suited for window sizes between 32K to 256K in terms of both accuracy and space requirements.

6.4 Error Results

To evaluate the accuracy of each approach, we have used three type of errors which have been already introduced in Section 3.2: The boundary error, bucket size error (or simply size error), and rank error. For each type of errors the absolute difference of the reported value by each algorithm with the ideal value is used to avoid domination of the final error values by the large differences. The average error for each of these three error types over time are shown in Figure 5. Part a, b, and c of the figure respectively compare the boundary, size, and rank errors of the three algorithms. As you can see, both BASH-BL and BASH-AL algorithms outperforms AM in terms of accuracy for most of the data sets. It is also worthy to mention that although not always true, BASH-BL is slightly more accurate than BASH-AL.

The aforementioned error results are for the case in which window size is $100k$. However, this error results differ for different window sizes. To see this, we compute the boundary, size and rank errors of the histograms generated by each of the algorithms over data set DS13 (Figure 6). All three algorithms perform almost equally in terms of errors up to the window size of 16k. However, from window size of 32k, the error rates of BASH algorithms start to decrease, while for the AM algorithm, this rate does not change, or it even increases. As a result, in larger window sizes, both BASH-BL and BASH-AL hugely outperform AM algorithms with respect to the three error types.

To have a better understanding of the evolution of accuracy for the histograms over time, we also compare the boundary error results for AM, BASH-BL, and BASH-AL for some of the data sets. We should mention that the errors of different diagrams are not comparable with each others since they are in different scales:

Data Sets with No Concept Shifts: Figure 7 compares the boundary errors for the three approaches (AM, BASH-BL, and BASH-AL) for uniform, normal, and exponential distribution. The X axis shows the number of tuples that have come into the system thus far, while the Y axis indicates the average boundary errors. Both BASH algorithms perform nearly the same, and at all points in time the generated histograms by the BASH algorithm are more accurate than the ones generated by the AM method.

Data Sets with Concept Shifts: Figure 8 illustrates the boundary errors of AM and two versions of the BASH algorithm on three of the data sets with concept shifts. The results mainly indicate that, not only are BASH-BL and BASH-AL on average more accurate than AM, but they are also more steady. This is more apparent in part a of Figure 8 that sketches the boundary error results for data set DS14 in which we have a concept shift at every 10,000 data items. To see this fluctuation in the errors computed for the AM method, all three diagrams in Figure 8 show the error of the generated histograms from tuple 4000k to tuple 6000k. In all these three data sets, the boundary error for the AM method fluctuates at each concept shift while BASH is more stable.

Real-World Data Sets: The boundary errors for the three real-world data sets are also computed for BASH-BL, BASH-AL, and AM algorithms (Figure 9). Similar to most of the cases we have seen thus far, BASH algorithms are far more accurate than AM for all the three data sets.

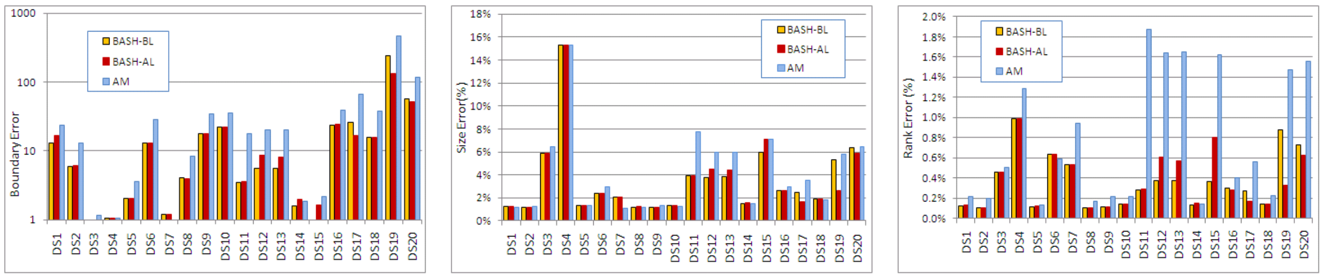


Figure 5: a) Boundary error, b) size error, and c) rank error for all data sets. ($k = 10$, $p = 7$, $W = 100k$, **and** $\epsilon = 0.01$.)

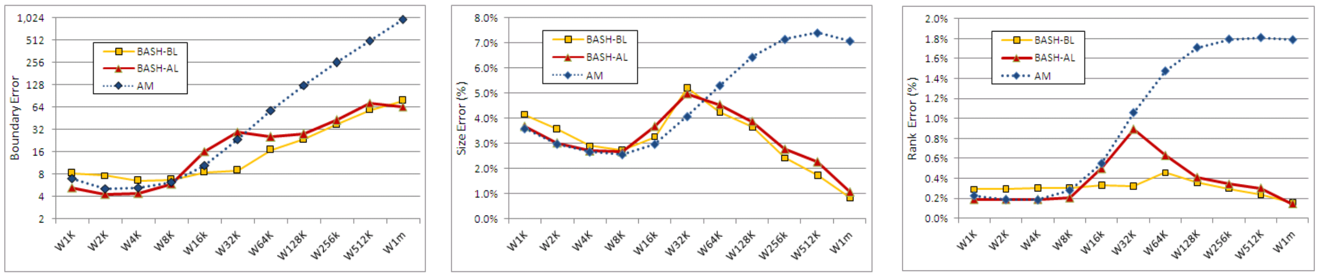


Figure 6: The effect of changing window size from 1k to 1m on the a) boundary error, b) size error, and c) rank error for DS13. ($k = 10$, $p = 7$, **and** $\epsilon = 0.01$.)

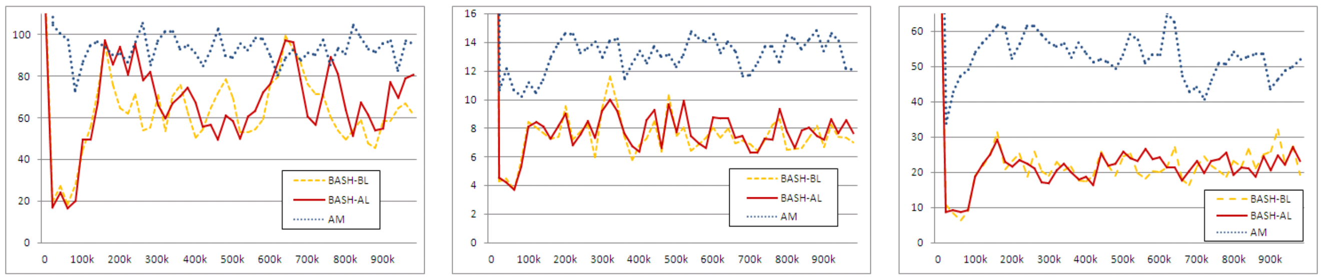


Figure 7: The boundary errors for data sets a) DS1, b) DS5, and c) DS8 from left to right. ($k = 10$, $p = 7$, $W = 100k$, **and** $\epsilon = 0.01$.)

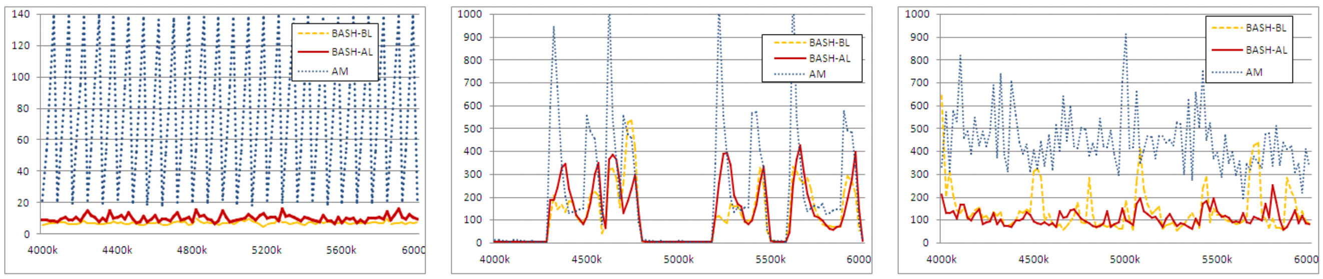


Figure 8: The boundary errors for data sets a) DS14, b) DS16, and c) DS17 from left to right. ($k = 10$, $p = 7$, $W = 1m$, **and** $\epsilon = 0.01$.)

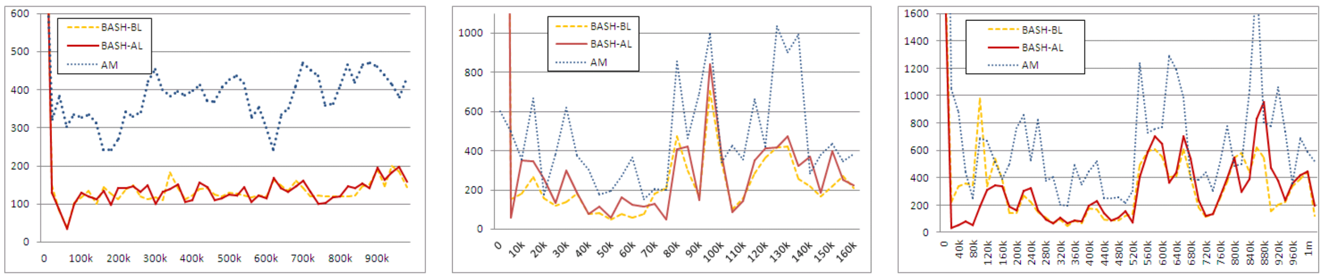


Figure 9: The boundary errors for data sets a) DS18, b) DS19, and c) DS20 from left to right. W **is set to** $10k$ **for DS19 and** $100k$ **for others.** ($k = 10$, $p = 7$, **and** $\epsilon = 0.01$.)

7. DISCUSSION

The experimental results for the performance of the algorithms on different window sizes, shown in Figures 3.b, Figure 4.b, and Figure 6 reveal some interesting features of the BASH algorithms. For all the cases, BASH-AL is slightly faster and more efficient (in terms of memory use) than BASH-BL. However, for window sizes larger than 32k BASH-BL typically produces more accurate histograms. The reason behind this observation is that BASH-BL does not lose any accuracy while merging, due to the use of blocked bars. On the other hand, the size of each bar (EH sketch) increases with the increase in window size and, as a result, larger boxes may appear in each bar for larger windows. In this case, merging technique in BASH-AL loses more accuracy due to miss-ordering larger boxes. Thus, BASH-BL performs more accurately than BASH-AL in larger windows; however, it needs more time and memory to maintain the blocked bars.

On the other hand, the interpretation of the comparison results for space usage between BASH methods and AM may not be so clear cut. The memory usage of AM is almost fixed (or even decreases) for different window sizes, while it logarithmically increases in BASH algorithms. The better scaling of space usage of AM for larger windows is mainly due to the difference in the main objectives of the two algorithms. Observe that for larger windows, the allowed error bounds increase for both algorithms since the errors are proportional to the window size. While AM uses this relaxation on the error constraint to minimize memory usage, BASH seeks to maximize accuracy and minimize the running time as well as keeping memory usage in a reasonable range. This claim is apparent in Figures 3.b, Figure 4.b, and Figure 6, wherein for those cases in which memory consumption of AM is more efficient than the BASH methods, the accuracy of the histograms generated by AM as well as its running time are significantly worse than those of both BASH algorithms.

According to the timing results in Figure 3, both BASH-BL and BASH-AL are at least four times faster than AM, while producing histograms that have almost the same or even better accuracy. A more important concern is the scalability of the algorithms with respect to running times. As part b of Figure 3 indicates, the execution time for both BASH-BL and BASH-AL linearly increases with the window size, while this increase for AM is much faster. This is mainly because BASH does not employ a sorting technique as opposed to AM. Moreover, the results in part a of Figure 3 show that the running times of BASH algorithms decreases for data sets with smaller standard deviations (such as DS4, DS7, and DS11), while AM needs more time and memory for these types of distributions.

8. CONCLUSION

Synoptic structures and techniques are widely recognized as being critical for supporting fast computations on massive data streams with light demands on computational resources [3], [4], [6], [7], [8], [9], [10], [13], [16], [21]. Nevertheless, while histograms are widely used in DBMS, DSMS have yet to see practical and efficient algorithms for building accurate histograms for windows over data streams. In this paper, we presented a new method called BAr Splitting Histograms (BASH) to estimate an expected ϵ-approximate equi-depth histogram for sliding windows over a data stream. We have performed extensive experimental evaluations to compare the speed, accuracy, and memory required by our algorithm vis a vis the state-of-the art AM method [3]. The comparison results obtained show that BASH delivers (i) significant improvements in the speed of computation (between 4 and 30 times faster), (ii) better overall accuracy, and (iii) comparable memory requirements. Moreover, when changes occur in the distribution of data, e.g., when concepts shift or drift, BASH responds and recovers its accuracy faster than the AM algorithm.

Several new research opportunities have also emerged in the course of this work, and will be pursued in the future by (i) integrating BASH with other synoptic techniques, such as sampling, (ii) constructing accurate equi-width histograms without requiring prior knowledge of the extrema, and (iii) extending our algorithm to compute biased histograms using the idea of biased quantiles [5]

9. ACKNOWLEDGMENTS

The authors would like to thank Armita Azari and Deirdre Kerr for their valuable help in gathering the data sets for the experimental results and also proof reading the paper.

10. REFERENCES

[1] Business databases at ucla anderson, school of management, http://www.anderson.ucla.edu/x14506.xml, December 2010.

[2] Uci machine learning repository, http://archive.ics.uci.edu/ml/datasets.html, December 2010.

[3] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *PODS*, pages 286–296, 2004.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[5] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In *PODS*, pages 263–272, 2006.

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

[7] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, pages 466–475, 1997.

[8] M. Greenwald. Practical algorithms for self scaling histograms or better than average data collection. *Perform. Eval.*, 27/28(4):19–40, 1996.

[9] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD Conference*, pages 58–66, 2001.

[10] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *STOC*, pages 471–475, 2001.

[11] F. Halim, P. Karras, and R. H. C. Yap. Fast and effective histogram construction. In *CIKM*, pages 1167–1176, 2009.

[12] Y. E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30, 2003.

[13] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 1998.

[14] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[15] X. Lin, H. Lu, J. Xu, and J. X. Yu. Continuously maintaining quantile summaries of the most recent n elements over a data stream. In *ICDE*, pages 362–374, 2004.

[16] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *SIGMOD Conference*, pages 426–435, 1998.

[17] H. Mousavi and C. Zaniolo. Fast and space-efficient

computation of equi-depth histograms for data streams. Technical report, UCLA, Los Angeles, USA, 2010.

[18] J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980.

[19] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *SIGMOD Conference*, pages 28–36, 1988.

[20] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD Conference*, pages 294–305, 1996.

[21] Q. Zhang and W. Wang. A fast algorithm for approximate quantiles in high speed data streams. In *SSDBM*, page 29, 2007.