Fast Computation of Approximate Biased Histograms on Sliding Windows 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

Histograms provide effective synopses of large data sets, and are thus used in a wide variety of applications, including query optimization, approximate query answering, distribution fitting, parallel database partitioning, and data mining. Moreover, very fast approximate algorithms are needed to compute accurate histograms on fast-arriving data streams, whereby online queries can be supported within the given memory and computing resources. Many real-life applications require that the data distribution in certain regions must be modeled with greater accuracy, and Biased Histograms are designed to address this need. In this paper, we define biased histograms over data streams and sliding windows on data streams, and propose the Bar Splitting Biased Histogram (BSBH) algorithm to construct them efficiently and accurately. We prove that BSBH generates expected ϵ-approximate biased histograms for data streams with stationary distributions, and our experiments show that BSBH also achieves good approximation in the presence of concept shifts, even major ones. Additionally, BSBH employs a new biased sampling technique which outperforms uniform sampling in terms of accuracy, while using about the same amount of time and memory. Therefore, BSBH outperforms previously proposed algorithms for computing biased histograms over the whole data stream, and it is the first algorithm that supports windows.

Keywords

Data Streams, Biased Histograms, Quantiles, and Sliding Windows.

1. INTRODUCTION

Histograms provide statistically accurate and memory-efficient synopses for large data sets, and thus find many important uses in database and data stream applications. Query optimization, approximate query answering, distribution fitting, parallel database partitioning, and data mining are only a few examples of such applications. In these applications, the queries may focus on some particular regions of the data distribution—typically regions located at the extremes of data distributions. For instance, consider the daily precipitation across the US in Figure 1[1]. This data set has a very long tail where more than 99% of the values are less than 0.1 inch/day. As shown in the figure, an equi-depth histogram (dashed line) does not provide much information on the tail of the distribution since almost all the data items at this area (tail) are assigned to one bar. The same type of problem exists for equi-width histograms.

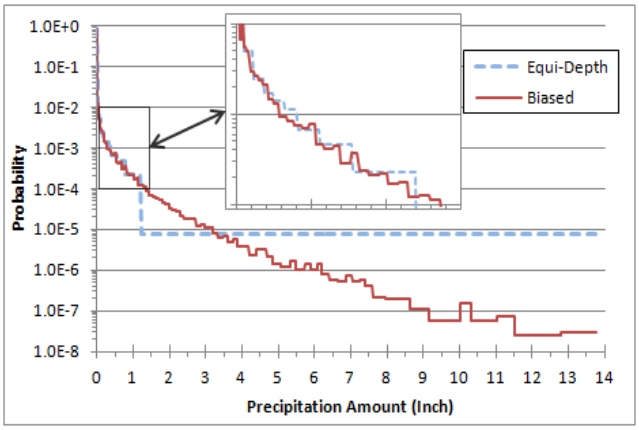

Figure 1: Equi-depth and biased histograms for annual precipitation in the US, with close-up magnification for the head of the distribution. (Each histogram has 100 bars.)

The biased-histogram synopsis proposed in this paper provides a much better alternative for such distributions. For instance, in Figure 1, we see that the biased histogram supports much more accurate estimations for the tail of the distribution while preserving a good estimation for the head of the distribution (shown in the box in the middle of Figure 1). This is achieved by letting the size of the histogram bars decrease exponentially toward the biased region. In particular, in Figure 1, the size of each bar is 90% of the size of the bar to its immediate left—thus, we will say that our histogram has a *bias factor* of 0.9 or 90%.

As another example, consider the distribution of the number of incoming (outgoing) URLs (links) in each page of the World Wide Web, which is known to be Zipfian. For this distribution, one may need reliable estimates of the average in-degree for the nodes that, say, rank in the interval 99.90% — 99.99% and for those in the interval 99.0% — 99.9%. Indeed this represents a crucial piece of information when we need to optimize the splitting/distributing of large data sets over different servers with roughly balanced load. Round Trip Times (RTTs) of the TCP packets provides another

[1]Data set is taken from http://www.ncdc.noaa.gov.

SSDBM '13, July 29 - 31 2013, Baltimore, MD, USA

good example. Since RTT delays can stretch over long periods in various situations, the distribution of RTTs is very skewed at its tail. Therefore, performance monitoring systems need to watch the RTT distribution with a biased interest over the tail of the distribution to detect suspicious behaviors.

The histogram problem is akin to that of quantiles [8], and there has also been some recent work on defining biased quantiles in data streams [2] [3] [21] [19]. Quantiles are used to extract the item that occupies a given position in a sorted list of items in a data set or in a window in a data stream. Generally, a head-biased (tail-biased) quantile consists of the sequence of ϕ, ϕ^2, ϕ^3 , ... $(1-\phi)$, $(1-\phi)^2$, $(1-\phi)^3$, ...) quantiles; for a given data set (or data stream) S with size N, the α-quantile ($0 \leq \alpha \leq 1$) is defined as the value of the item at the position $\lceil \alpha N \rceil$ in the sorted list of the items in S. Computing exact quantiles is a challenging problem even for small data sets [14], and thus incompatible with most data stream applications which require online response with limited memory and computational resources. As our experiments show, these problems carry over to biased quantiles. Moreover, although windows are crucial in most data stream applications, previous works on biased quantiles have always assumed that the quantiles represent the whole history of the data streams [2] [3] [21] [19].

Therefore in this paper, we first define approximate biased histograms on data streams with sliding windows, and then propose a new efficient algorithm called *Bar-Splitting Biased Histograms* (BSBH) to computes approximate biased histograms over sliding windows of fast data streams. To the authors' knowledge, this is the first work on designing biased histograms over sliding windows of data streams. BSBH employs a similar structure as in our previous work on designing equi-depth histograms in [13]. BSBH also utilizes a biased sampling technique to improve the performance in terms of both CPU and memory usage. More specifically, our paper makes the following contributions:

- We define the concept of *Biased Histograms* over sliding windows of data streams, and present a new algorithm called *Bar-Splitting Biased Histograms* (BSBH) to compute approximate biased histograms over fast data streams with sliding windows. (Section 3)
- To be able to tune the memory and CPU usage of BSBH, we propose a new biased sampling technique. Our experimental results indicate that biased sampling doubles the accuracy of the results with respect to uniform sampling while spending almost the same amount of memory and CPU. (Section 3.3)
- We prove that BSBH can guarantee expected ϵ-approximate biased histograms for data sets with no concept shift. We also provide theoretical bounds on both execution time and memory usage of our algorithm for this case in Section 4.

We use extensive experiments to evaluate the performance of BSBH and also compare it with CKMS which is one of the best existing algorithms for generating biased quantiles over the entire history of data streams [3]. Our results (Section 5) show that BSBH outperforms CKMS with respect to execution time, memory usage, and accuracy. We have also evaluate BSBH for data streams with different rates of concept shifts on their distribution, and the results show that BSBH provides acceptably accurate results very quickly (mostly in couple of slides) after observing the concept shifts.

2. BACKGROUND AND PRELIMINARIES

Since BSBH is based on the Exponential Histograms (EH) sketch [4], we first provide a brief description of this technique.

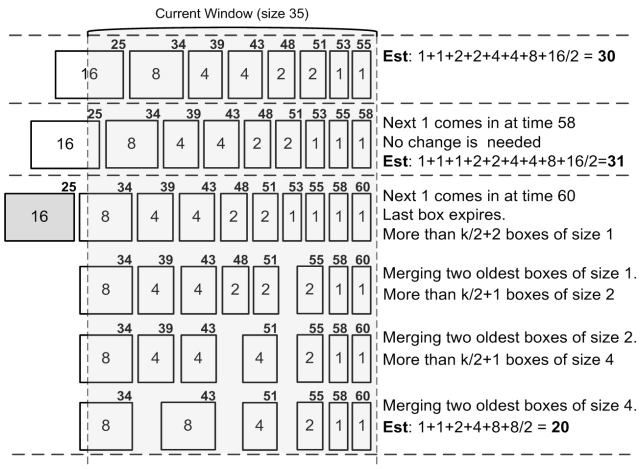

Figure 2: Incrementing an EH sketch twice at time 58 and 60. That is we have seen 1, 0, and 1 respectively at time 58, 59, and 60. (k=2 and W=35)

2.1 Exponential Histogram Sketch

In [4], Datar *et al.* proposed the Exponential Histograms (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 consists of an ordered list of buckets. In this paper, we refer to these buckets as boxes to avoid confusion since we will use the term *bucket* at another level in our approach. 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 2 illustrates how this operation works.

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 2 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. This approach is quite fast too, since the amortized number of merges for each new 1 is only $O(1)$. It is also worth noting that instead of the counting the 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. For instance, 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.

2.2 BAr Splitting Histogram (BASH)

In [13], an expected ϵ-approximate algorithm is proposed that generate equi-depth histograms for sliding windows over fast data streams. Equi-Depth Histograms [7][15] (also known as equi-height or equi-probable histograms) seek to specify boundaries between buckets such that the number of tuples in each bucket (buckets' size) is the same. Histograms of this type are more effective than equi-width histograms, particularly for data sets with skewed distributions [9].

The equi-depth histogram problem is obviously akin to that of quantiles [8], which seeks to identify the item that occupies a given position in a sorted list of N items: Thus given a ϕ, $0 \leq \phi \leq 1$, which 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 (e.g., 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 is not practical because quantile computation algorithms over sliding windows are too slow, since they must derive more information than what is needed to built a histogram [17].

In a nutshell, BASH is based on dividing the acceptable input range (U) into several chunks or $bars$ in such a way that each bar contains roughly equal number of items. The size of each of these bars is estimated using an EH sketch. To assure that the bars contain almost the same number of items as the stream passes by, a merge/split technique is employed to split big bars, and merge adjacent small bars. BASH provides a very fast and memory-efficient equi-depth histogram particularly for high-speed data streams. Note that BASH does not need any prior knowledge of U, N (the number of items), or minimum and maximum values of the data items since it dynamically adopts the boundaries of bars with the newly arrived data items. The next section discusses BSBH which exploits the main structure of BASH while introducing the concept of biased sampling to improve the CPU and memory usage of the algorithm.

3. BIASED HISTOGRAM COMPUTATION

This section first reviews the definitions of quantiles and histograms and then introduces the definition of biased histograms. The Bar-Splitting Biased Histogram (BSBH) algorithm is then thoroughly explained.

3.1 Definitions

Cormode *et. al.* defined *Biased quantiles* [2] as follows:

DEFINITION 1. *A Low-Biased Quantile with bias factor $\phi < 1$ for a given sequence of data items is the set of items with ranks $\lceil \phi^i N \rceil$ for i = 1, 2, ..., B=$log_{1/\phi}(N)$ in the ordered list of items, where N is the size of the data set.*

One can similarly define the *High-Biased Quantile* as well. The above definition is specifying B=$log_{1/\phi}(N)$ boundaries, which also can be seen as a B-bucket histogram. As stated in [14], one pass algorithms for computing the exact quantiles need to store the entire data set which is too expensive in most data streaming computations. Approximate biased quantile were thus defined in [3] and [19] as follows:

DEFINITION 2. *An approximate Low-Biased Quantile with bias factor $\phi < 1$ of a given sequence of data items, say S, is the set of items $\{v_i\} \in S$ for i = 1, 2, ..., $log_{1/\phi}(N)$, where:*

$$|rank(v_i) - \phi^i N| \leq max\{\epsilon.\phi^i N, \epsilon_{min} N\}$$

where $rank(x)$ is the position of data item x in the ordered list of items, and ϵ and ϵ_{min} are two approximation factors. The reason for needing two approximation factors is that the term $\epsilon.\phi^i N$ becomes very small for quantiles that are near the biased point (for larger i's). Thus if the term $\epsilon\phi^i N$ is used, unreasonably high accuracy would be required for quantiles near the biased point . To alleviate this problem, the alternate approximation factor (ϵ_{min}) is used instead. This essentially means that for the biased regions a flat error curve is used.

Although the above definition is useful in many applications, it is not suitable for designing biased histograms due to the following issues: First, the error is based on the rank of the reported quantiles, while in biased histograms we seek to minimize the error on the size of each bucket. Second, the definition imposes the restriction of having exactly $log_{1/\phi}(N)$ buckets or boundaries. However, many application may need to set the number of buckets based on their internal settings[2] which is usually set independent of the stream size. Third, since flat error rate (ϵ_{min}) is used for the areas near biased points, it might not provide an accurate result at those areas. This also makes the tuning of ϵ_{min} a challenging issue. To address these issues, we next define an approximate *Low-Biased Histogram* in which the goal is to keep the size of each bucket in the histogram close enough to the ideal size:

DEFINITION 3. *An ϵ-approximate B-bucket Low-Biased Histogram (with bias factor $\phi < 1$) of a given sequence of ordered data items, say S, is the set of B buckets $\{B_i : i = 0, 1, ..., B-1\}$ partitioning S with the following invariant:*

$$|size(B_i) - \alpha_\phi \phi^i W| \leq \epsilon \alpha_\phi \phi^i W$$

where: W is the window size, $\alpha_\phi = (1-\phi)/(1-\phi^B)$ is a constant factor to make the buckets' sizes add up to W, and $size(B_i)$ is the number of items in bucket B_i. In other words, the ideal goal of the above definition is to partition the ordered data set into B buckets of sizes $\alpha_\phi W$, $\alpha_\phi \phi W$, $\alpha_\phi \phi^2 W$, ..., and $\alpha_\phi \phi^{B-1} W$. Note that with this definition B is independent from the size of the stream and does not need to be exactly $log_{1/\phi}(W)$—a desirable property since W is often unknown a priori. Approximate *High-Biased Histogram* or *Targeted Quantiles* [3] can be similarly defined[3].

3.2 Bar-Splitting Biased Histogram (BSBH)

The BSBH algorithm computes approximate B-bucket biased histograms over sliding windows of data streams. We assume the current size of the window at time t is W_t without needing to make any assumption about the type of the window (physical or logical.) As will be discussed later, BSBH maintains an estimation of W_t called W_{est}. Unlike other approaches [2] [3], BSBH does not need the prior knowledge of U's size[4]. This subsection focuses on BSBH algorithm for generating low-biased histograms without using any sampling. Later in Subsection 3.3, we show how the idea of biased

[2]e.g. It could be set according to the number of available processing units.

[3]The notion of *end-biased histogram* used in [9] should not be confused with the above definition of biased histograms, since their idea is to put high-frequency items in singleton buckets.

[4]Universe U is the range of acceptable data values.

sampling can be added to BSBH in order to improve its time and memory performance.

BSBH Core Algorithm: The main goal in BSBH is to partition the interval between current minimum and maximum in the sliding window into $S_m = B \times p$ intervals bar_0, bar_1, ..., bar_{S_m-1}, where p is an extension factor for improving the accuracy and is determined analytically. The size of each bar is estimated using the EH structure discussed in Subsection 2.1. These bars will be later used to approximate the final $(B-1)$ buckets' boundaries.

The ideal goal of the BSBH structure is to keep the size of bar_i (i.e., $|bar_i|$) to ρ times the size of bar_{i+1} for $i = 0, 1, ..., S_m - 2$, where $\rho = \phi^{1/p}$. In other words, the ideal case is when:

$$|bar_i| = \rho |bar_{i+1}| = \alpha_\rho \rho^i W_t$$

for $i = 0, 1, ..., S_m - 2$, where α_ρ is a factor that makes the bars' sizes add up to W_t (i.e. $\alpha_\rho = (1-\rho)/(1-\rho^{S_m})$). As for W_t, BSBH uses the aggregate size of all the current bars which we will refer to as W_{est}. Due to properties of the EH structure, the expected value of W_{est} would be W_t if no sampling is used, however at each point in time these two may have different values. With such a setting, if the bars' sizes are ideal and no sampling is used, the aggregate size of bars bar_j for $ip \leq j < (i+1)p$ will be $\alpha_\phi \phi^{i+1} W$ for $i = 0, 1, ..., B-1$. These B buckets are actually what the definition of biased histograms (Definition 3) was seeking.

Algorithm 1 shows how the BSBH structure, mentioned above, is initialized and then maintained. We will next discuss how the structure is maintained once it is built, and discuss how it is initialized later in the paper.

Maintenance: For each incoming data item from the data stream (say $next$), BSBH finds the appropriate bar, say bar_i, for $next$ based on the current boundaries of the bars (Line 20). This is done using a simple binary search. Then, the EH structure of bar_i is incremented by one (Line 23). While doing this, BSBH updates the current minimum and maximum values of the entire history of the data stream. At this point, if the bar_i's size is greater than a dynamically computed threshold ($maxSize_i$), BSBH splits the bar into two smaller bars (Lines 24 to 26).

While splitting bars, BSBH may need to merge two adjacent bars in order to keep the total number of bars below S_m. It is important to stress on that BSBH only merges two bars when a bar should be split and there is no room for a new bar. As opposed to BASH in which $maxSize$ is the same for all the bars at any given point in time, in BSBH, $maxSize$ is determined based on the bar's index and its closeness to the biased point. Note that, the idea is still similar to BASH; the size of bar_i should not be greater than $maxCoef > 1$ of its ideal size. In other words:

$$maxSize_i = maxCoef.\alpha_\rho.\rho^i.W_{est}$$

Although larger $maxCoef$ results in less splits, in practice we use $maxCoef = 1.7$ to reach higher accuracy as well as to accelerate the processes of stabilizing the boundaries for the initialization phase and to better cope with concept shift. Notice that having different thresholds for bars essentially allows smaller bars near the point of interest.

After assuring that the bars' sizes are in the acceptable range, if any of the boxes of the existing EHs is expired, BSBH will remove it from the structure (Line 27). Finally, after updating the structure for an incoming data item, we can compute the current boundaries for the final buckets. This is normally done at every slide (Lines 28 and 29). Next, we explain the main modules of the BSBH algorithm in more detail.

Initialization: The initialization phase plays a very important role in the BSBH algorithm. This is mostly because at the beginning, the minimum and maximum values of the data set are unknown. Thus, instead of starting with S_m bars BSBH starts with one empty bar, and keeps adding new data items into that bar until its size reaches a threshold ($maxSize_0$). At this point, the bar is full and it will be split into two bars. BSBH repeats this until S_m bars are created. After this stage for each split operation, two consecutive bars should be merged. Note that $maxSize_i$ at all time is proportional to the current window size $W_t \simeq W_{est}$; this essentially means that we allow more splits during the initiation phase of the algorithm, since W_t is small at this phase. This makes the initiation phase much faster. For instance due to our threshold-based mechanism, the first split happens usually at the second or third insert.

Algorithm 1 BSBH()

```
1 : BSBH (φ, μ, B, p)
2 : {
3 :    μ' = 1; /* for gradually decreasing sample rate*/
4 :    effectSmp []; /* initialized with 1's */
5 :    ρ = φ^(1/p);
6 :    if (μ < ρ)
          μ = ρ;
7 :    initialize();
8 :    cnt = 0;
9 :    while (true)
10:    {
11:       next = next item in the data stream;
12:       cnt ++;
13:       //Updating the Sampling Rate
14:       if (cnt ≤ X)
15:          if (μ' > μ)
16:             μ' = 1 − ((1 − μ) × (cnt − X))/X;
17:       if (cnt ≤ 2X)
18:          Compute effective sampling rate for each bar and
             Update effectSmp array;
19:       //Biased Sampling Phase
20:       find appropriate bar bar_i for next;
21:       if (!Biased-Sample(next, i, μ'))
22:          continue; /* Skipping the item */
23:       insert next into bar_i.EH;
24:       maxSize_i = α_ρ.maxCoef.ρ^i.W_est;
25:       if (|bar_i| > maxSize_i)
26:          splitBar(bar_i, ρ);
27:       remove expired boxes from all EHs;
28:       if (window slides)
29:          output computeBoundaries(B);
30:    }
31: }
```

Merge-Bar Operation: Let two adjacent bars, say X_L and X_R, are selected to be merged into one bar called X. Later in this section, we discuss the details on how the bars are chosen for merging. As depicted in Figure 3, BSBH first integrates the intervals that X_L and X_R are covering. Next, for the EH sketches of bars X_L and X_R called EH_{X_L} and EH_{X_R} respectively, BSBH set the smaller one to *blocked*, and the bigger one to *active*. In other words, the

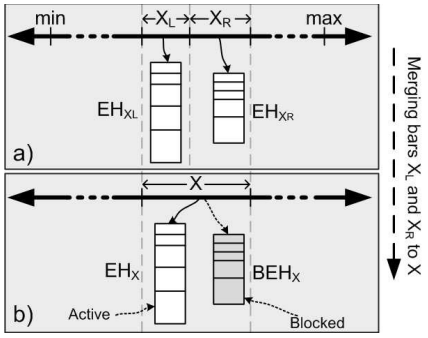

Figure 3: Merging two bars (X_L and X_R) into one bar (X). a) right before merge-bar operation and b) right after merge-bar operation.

new bar X now has two associated EH sketches but only one of them is active. This means that the incoming tuple for the X's interval will be inserted into the active EH (EH_{X_L} or EH_X in our example in Figure 3). As for the blocked EHs, BSBH stops incrementing them, but continues removing expired boxes from them. Thus, blocked EHs are prevented from growing, and as a result they will soon be disappeared from the structure and will no longer require memory. Also, observe that every bar may contain more than one blocked EH. For instance, consider the case where we have to merge two adjacent bars which already have an associated blocked EH (e.g. merging bar X in part b of Figure 3 with one of its neighbors). To merge these two, we follow the same general approach: the longest EH is set to the active one and all other EHs are considered as the blocked EHs of the new bar.

At all times, BSBH makes sure that the size of the active EH for each bar is bigger than those of the blocked EHs of the bar, and whenever this is no longer the case, BSBH switches the active EH with the blocked EH of larger size. Although this case is rare, it can occur when the boxes of the active EH expire more quickly than those of the blocked EHs. In Section 4, we will prove that the expected number of such blocked EHs is constant for large enough N's. Notice that since the internal structure of EHs is untouched in this merging technique, the technique does not affect the accuracy of the final results.

Split-Bar Operation: When the size of a bar, say X, exceeds its maximum threshold ($maxSize_i$, where i is the position of X in the list of current bars), we split it into two smaller bars using Algorithm 2. Before the split operation takes place, we make sure that the number of bars in the histogram is less than S_m. If this is not the case, as previously explained, we should first merge two other adjacent bars.

Let bar X need to be split to two bars called X_L and X_R. We divide the interval being covered by X into a pair of intervals (Lines 5 to 8). The goal is that $|X_R| = \rho \times |X_L|$ after the split operation takes place. To this end, the algorithm first distributes the blocked EHs of X, denoted as BEH_X, into the blocked EHs of X_L and X_R (Lines 9 to 16). The splitting algorithm tries to do this in a way that preserves the size constraint ($|X_R|/|X_L| = \rho$); however this is not always possible. That is after splitting blocked EHs of X to X_L and X_R the ratio $\rho' = |X_R|/|X_L|$ may be different than the ideal ratio (ρ). Thus, BSBH splits the active EH of X (EH_X) to compensate for this difference ($\rho - \rho'$) (Lines 17 to 29). It is easy to show that we can always achieve this goal by using appropriate split on EH_X, given that the size of the active EH is guaranteed to be greater than the size of all the blocked EHs.

To split the original EH sketch, EH_X, into a pair of sketches called EH_{X_R} and EH_{X_L}, BSBH first computes the split ratio denoted as λ (Line 17 in Algorithm 2). λ simply indicates that to compensate for the aforementioned difference ($\rho - \rho'$), the size of EH_{X_R} after splitting should be λ times the size of EH_X (e.i. $|EH_{X_R}| = \lambda|EH_X|$). In order to have such a split ratio, BSBH puts half of the EH_X boxes with size one into EH_{X_L}, and the other half into EH_{X_R} (Line 19 and 20). 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, BSBH decides whether to put each copy to EH_{X_R} or EH_{X_L} (Lines 23 to 27). In other words, if the current ratio of $|EH_{X_R}|/|EH_{X_R} + EH_{X_L}|$ is smaller than λ the copy will go to EH_{X_R}, otherwise it will go to EH_{X_L}. For instance if λ is 0.5, one half goes to EH_{X_R} and the other half goes to EH_{X_L}.

Algorithm 2 splitBars(X, ρ)

```
1 : if (curBarNo == S_m && !mergeBars())
      return;
2 : Initialize new bars X_L and X_R;
3 : l = 0;
4 : |BEHs| = Aggregate Size of the blocked EHs;
5 : EH_{X_L}.start = EH_X.start;
6 : EH_{X_L}.end = (EH_X.start + EH_X.end)/(1+ρ);
7 : EH_{X_R}.start = EH_{X_L}.end;
8 : EH_{X_R}.end = EH_X.end;
9 : for each (blocked EH bar in BEH_X) {
10:   if (l + |bar| < |BEHs|/(1+ρ)){
11:     l += |bar|;
12:     Add bar to BEH_{X_L};
13:   }
14:   else
15:     Add bar to BEH_{X_R};
16: }
17: λ = ((|X|/(1 + ρ))-l)/(|X| - |BEHs|);
18: foreach (box box in EH_X){
19:   if (|box| == 1)
20:     Alternatively add a copy of box to EH_{X_L} or EH_{X_R};
21:   else {
22:     |box| = |box|/2;
23:     for (2 times)
24:       if (|EH_{X_R}| /(|EH_{X_R}|+|EH_{X_L}|) < λ)
25:         Add a copy of box to EH_{X_R};
26:       else
27:         Add a copy of box to EH_{X_L};
28:   }
29: }
30: Remove X from the bars list;
31: Merge boxes of bars X_L and X_R if necessary.
32: Add bars X_L and X_R to the bars list;
```

Selecting Bars to Merge: We always merge adjacent bars that have the minimum aggregate size with respect to their positions and ideal size. Thus, we select a pair of adjacent bars that yields the least relative deviation from their ideal sizes. In other words, we pick bars bar_i and bar_{i+1} such that $|bar_i|/\rho^i + |bar_{i+1}|/\rho^{i+1}$. If this aggregate size of these two bars (e.i. $|bar_i| + |bar_{i+1}|$) is less than $maxSize_i$ (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 $maxSize_i$ by merging.

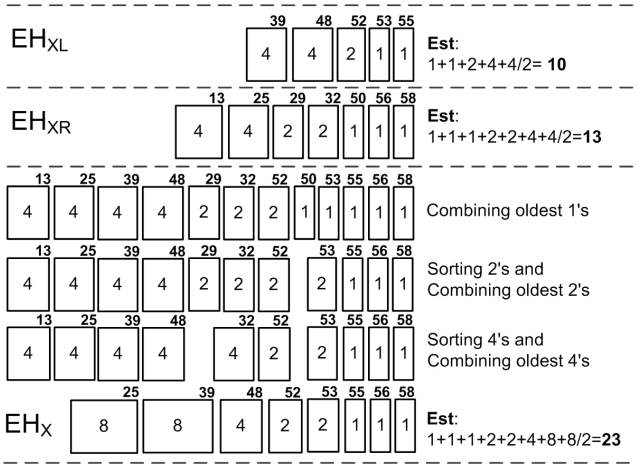

Figure 4: Merging two EH sketches EH_{XL} and EH_{XL} into EH_X. (k=2)

An Alternative Merging Approach: The merging approach introduced earlier uses blocked EHs which may increase the memory requirement and thus have a negative impact on memory performance. As an alternative approach, one can combine the boxes of the two to-be-merged bars into a single new EH sketch. This technique which is explored in the next paragraph needs slightly less memory, but makes the theoretical analysis a lot harder. The former technique is called BSBH-BL, since it uses BLocked EHs, and this alternative approach is referred to as BSBH-AL thorough the rest of the paper.

To merge two EH sketches in BSBH-AL, we start by selecting boxes with size one from both EH sketches, and put them into a list sorted by the start time of their intervals (Figure 4). 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 operation is then repeated for boxes with larger size (4, 8, etc.). Figure 4 illustrates an example of merging two EH sketches where $k = 2$ using this alternative method.

Note that by merging boxes from two different EH sketches, we may lose some information about timestamps of the items which affects the accuracy of the estimations as discussed in Section 5. Although, we do not have any blocked EHs for BASH-AL, the same split approach we mentioned earlier can be used for this alternative merging technique. This time, the splitting operation in Algorithm 2 splits the only existing EH into two EHs with appropriate size.

Reporting the Final Buckets: At each slide, the algorithm needs to generate a new set of boundaries/buckets. Algorithm 3 shows the pseudocode for this operation, in which the ideal goal is to find B buckets with sizes $\alpha_\phi \phi^i W_t$ for i=0, 1, ..., $B-1$. Similar to [13], Algorithm 3 passes over the list of bars once and estimates B_is boundaries assuming that the distribution of items inside each bar is uniform. Observe that the extension factor p plays an important role in this assumption. Indeed, larger values of p generate smaller intervals, and consequently make the distribution of the items in each bar closer to a uniform distribution. Since the value of W_t might not be known, we will instead use its estimate W_{est}.

Algorithm 3 computeBoundaries(B)

$b = -1$; //An index over bars list
$count = 0$;
$curBuckSize = \alpha_\phi \phi^{B-1} W_{est}$;
for(int $i = 0$; $i < B$; i++) {
  while ($count \leq curBuckSize$) {
    b ++;
    $count$ += $|bars[b]|/effectSmp[b]$;
  }
  $surplus = count - curBuckSize$;
  $boundaries[i] = bars[b].start + bars[b].length \times$
    $(1 - surplus/(|bars[b]|/effectSmp[b]))$
  $count = surplus$;
  $curBuckSize = curBuckSize \,/\, \phi$;
}
return $boundaries$;

3.3 Biased Sampling

Sampling provides a simple technique for scaling many algorithms to larger data sets. The most common way to sample is to uniformly drop items from the input data set and reduce its volume. Although this is very easy to implement, it may not be the best practice for biased histograms, since uniformly removing items from smaller bars has a more negative effect in estimating the boundaries than in larger bars. Thus, a good sampling technique needs to sample out fewer items around the biased point(s) than from the rest of the data distribution.

Random non-uniform sampling was also used by Zhang and Wang [19] who first collect and sort the whole input and then sample from the ordered input at non-uniform rate. BSBH instead introduces a faster non-blind sampling technique, called *biased sampling*, which does not need to see the entire data set in advance. This, makes BSBH a much more practical approach for data streams. The goal in biased sampling is to sample (keep) at most σ items from each window. Thus for a low-biased histogram, knowing the new items value ($next$) and its associated bar index (i), we sample $next$ with probability μ^i, such that the following equation holds:

$$\sum_{i=0}^{S_m-1} \mu^{S_m-1-i}|bar_i| = \sigma \times W_t$$

As you can see, the sampling occurs such that we keep all the items in the smallest bar (bar_{S_m-1}), sample items from bar_{S_m-2} with rate $\mu \leq 1$, and exponentially decrease the sampling rate to a more selective one for larger bars. In the rest of the paper, we will refer to μ as the *biased sampling rate*. Note that it is not plausible to sample in a way that after sampling the size of bar_{i+1} becomes smaller than the size of bar_i, which means μ should be less than or equal to ρ ($\mu \leq \rho$). If smaller sample rates are desired, uniform sampling should be used. For the case of $\mu = \rho$, it is easy to see that after observing at most $2W$ items all bars will have equal sizes (which is $\alpha_\rho \rho^{S_m-1} W_t$) and the biased histogram computation is reduced to an equi-depth histogram computation. This basically means that BSBH with biased sampling rate ρ will be eventually reduced to BASH after the sampling phase is completed.

One challenge most of the sampling-based techniques need to address is the initiation phase. Since the system is initially empty, to quickly, and more accurately report the first set of results, the algorithms need to keep more data items and drop less. Later when the structures are initiated, the techniques can sample more data items to reduce the load. Moreover in our case, the sampling is not blind and needs to know the boundaries of the histogram's bars. Thus, we need to first generate the boundaries so we can start the sampling phase. In order to alleviate the effect of switching to the sampling stage, we gradually decrease biased sampling rate from 1 (no sampling) to μ for the first X data items (lines 14 to 16 in Algorithm 1). For the physical windows X could be set to the window size W. For the logical windows, X can be set to estimated size of the window at the time in which the first item expires.

Since we can not start full sampling form the beginning, the effective biased sampling rate for bar_i is not going to be μ^i for the first $2X$ data items. Thus, in lines 17 and 18 of Algorithm 1, we update the effective sampling rate for each bar in an array called $effectSmp$ based on the history of changes on sampling rates. $effectSmp$ will be used later to estimate the actual size of bars.

4. FORMAL ANALYSIS

In this section, we show that BSBH can provide an expected ϵ-approximate biased histogram for sliding windows on data streams with no concept shift for any given $\epsilon > 0$. We also compute the per-data item delay and space complexity of BSBH for this case. Notice that data streams with no concept shift can be modeled as the case that every incoming item from the data stream is taken form a fixed data distribution. This consequently means that items are arriving in random order. To ease our discussion, we consider that the window size is fixed at W. First, we compute the expected number of splits in each sliding window.

Let $E_{sp}\{N\}$ be the expected number of splits after processing N data items from the data streams. Observe that the expected number of splits for bar_i is $E_{sp}\{N\}/S_m$ since the probability of insert to (and expire from) bars for randomly ordered inputs are the same for each bars. Let $p_{t_1,t_2,i,j}$ be the probability that the current bar at i'th index/position (bar_i) in time t_1 is moved to position j at time t_2 without being split or merged (That is, the change of position is only due to splitting and merging of other bars). Also, let $q_{t,i}$ be the probability of having bar_i split at time t. Thus, the probability of having two consecutive splits on a bar at times t_1 and t_2 and at the starting position i and the finishing position j is $P_{t_1,t_2,i,j}$ = $q_{t_1,i}p_{t_1+1,t_2,i,j}q_{t_2,j}$. Knowing this, we can now calculate the expected number of inserts into bar_i (called $E_{in}\{i\}$) between two consecutive splits using the following formula:

$$\begin{aligned}
E_{in}\{i\} &= \frac{S_m}{E_{sp}\{N\}} \sum_{t}^{N}\sum_{t'=t}^{N}\sum_{j=0}^{S_m-1} P_{t-1,t',i,j}(x_{t,t',i} + m_j - \frac{m_i}{2}) \\
&= \frac{S_m}{E_{sp}\{N\}} \sum_{t}^{N}\sum_{t'=t}^{N}\sum_{j=0}^{S_m-1} P_{t-1,t',i,j}(m_j - \frac{m_i}{2}) \\
&\quad + \frac{S_m}{E_{sp}\{N\}} \sum_{t}^{N}\sum_{t'=t}^{N}\sum_{j=0}^{S_m-1} P_{t-1,t',i,j}(x_{t,t',i}) \\
&= I_i + X_i
\end{aligned}$$

where m_i is the abbreviated form of $maxSize_i$ and $x_{t,t',i}$ is the expected number of expired items from bar_i in between the time interval $[t, t']$. Basically, for two consecutive splits on bar_i at times $t-1$ and t', we need to insert $m_j - \frac{m_i}{2}$ items to reach to the split threshold at position j if nothing expires from the bar. Otherwise, we need to insert and additional $x_{t,t',i}$ of items to compensate for the expired items from the bar. The above equation simply adds up all of such values for all possible t, t', and j. Then, the expected number of inserts among all bars between two consecutive splits can be computed with the following formula:

$$E_{in} = \sum_{i=0}^{S_m-1} E_{in}\{i\} = \sum_{i=0}^{S_m-1} I_i + \sum_{i=0}^{S_m-1} X_i = I + X \qquad (1)$$

Let us start with I:

$$\begin{aligned}
I &= \sum_{i=0}^{S_m-1} \frac{S_m}{E_{sp}\{N\}} \sum_{t}^{N}\sum_{t'=t}^{N}\sum_{j=0}^{S_m-1} P_{t-1,t',i,j}(m_j - \frac{m_i}{2}) \\
&= \frac{S_m}{E_{sp}\{N\}} \sum_{t}^{N}\sum_{t'=t}^{N}\sum_{i=0}^{S_m-1}\sum_{j=0}^{S_m-1} P_{t-1,t',i,j}(m_j - \frac{m_i}{2}) \\
&= \frac{S_m}{2E_{sp}\{N\}} (\sum_{t}^{N}\sum_{t'=t}^{N}\sum_{i=0}^{S_m-1}\sum_{j=0}^{S_m-1} P_{t-1,t',i,j}(m_j - \frac{m_i}{2}) \\
&\quad + \sum_{t}^{N}\sum_{t'=t}^{N}\sum_{i=0}^{S_m-1}\sum_{j=0}^{S_m-1} P_{t-1,t',j,i}(m_i - \frac{m_j}{2})) \\
&= \frac{S_m}{2E_{sp}\{N\}} \sum_{t}^{N}\sum_{t'=t}^{N}\sum_{i=0}^{S_m-1}\sum_{j=0}^{S_m-1} P_{t-1,t',i,j}(\frac{m_i + m_j}{2})
\end{aligned}$$

In the last line of the above equations, we have used the fact that probabilities $q_{t,i}$ and $q_{t,j}$ are equal for any given t for randomly ordered inputs with no concept shift. This also indicates that $P_{t_1,t_2,i,j}$ and $P_{t_1,t_2,j,i}$ are equal. To understand this fact, let $P_{t_1,t_2,i,j} > P_{t_1,t_2,j,i}$ (or $P_{t_1,t_2,i,j} > P_{t_1,t_2,j,i}$). This means that the bars between i and j indices are constantly moving toward the index j (or i) due to splitting and merging which is in contradiction with the fact that the data stream contains no concept shift.

$$\begin{aligned}
I &= \frac{S_m}{2E_{sp}\{N\}} \sum_{t}^{N}\sum_{t'=t}^{N}\sum_{i=0}^{S_m-1}\sum_{j=0}^{S_m-1} P_{t-1,t',i,j}(\frac{m_i + m_j}{2}) \\
&= \frac{S_m}{2E_{sp}\{N\}} \sum_{t}^{N}\sum_{t'=t}^{N}\sum_{i=0}^{S_m-1}\sum_{j=0}^{S_m-1} P_{t-1,t',i,j} m_i \\
&= \frac{S_m}{2E_{sp}\{N\}} \sum_{i=0}^{S_m-1} m_i \sum_{t}^{N}\sum_{t'=t}^{N}\sum_{j=0}^{S_m-1} P_{t-1,t',i,j}
\end{aligned}$$

Note that $\sum_{t}^{N}\sum_{t'=t}^{N}\sum_{j=0}^{S_m-1} P_{t-1,t',i,j}$ computes the expected number of splits for bar_i for the first N data items of the stream. As already discussed this value is equal to $E_{sp}\{N\}/S_m$. Therefore,

$$\begin{aligned}
I &= \frac{S_m}{2E_{sp}\{N\}} \sum_{i=0}^{S_m-1} m_i (\frac{E_{sp}\{N\}}{S_m}) \\
&= \frac{1}{2}\sum_{i=0}^{S_m-1} m_i = \frac{maxCoefW}{2}
\end{aligned} \qquad (2)$$

The above formula simply indicates that the expected number of inserts per split, or E_{in}, is greater than $maxCoefW/2$. This leads us to the following lemma which is very important to prove the bound on memory requirements of BSBH.

LEMMA 1. *For $N \geq W$, the expected number of blocked EHs in BSBH is $O(1)$.*

Proof: Equation 2 indicates that for $N \geq W$, the expected number of inserts before seeing the next split in BSBH is greater than

$maxCoefW/2$ (equation $\sum_{i=0}^{S_m-1} m_i = maxCoefW$ only holds for $N \geq W$.) That is, in each window the number of splits is $O(1)$ on the average. On the other hand, the number of blocked EHs is proportional to the number of splits as proved in Lemma 2 of [13]. This completes the proof. □

Note that after first W items have arrived, older items start to expire with the same rate as new-arriving items. This essentially means that the expected number of items expiring in a given time interval is the same as the expected number of arriving items ($X = I$). That is after the first window:

$$E_{in} = I + X = 2I = maxCoef \times W$$

This essentially means that for $maxCoef > 1$, E_{in} would be more than W. In other words, the expected number of split in each window is less than 1.

THEOREM 1. *For data stream S with no concept shift and for any given $0 < \epsilon$ and $0 < \phi < 1$, the BSBH algorithm can provide a size-based expected ϵ-approximate biased histogram with bias factor ϕ for sliding windows on S.*

Proof: As stated above in each window there can be at most one split, and no matter how poorly this split operation is performed, before the next split all the wrongly split items will have been expired. Thus at any moment, only two adjacent bars may contain incorrect items due to the split operation. We refer to these two bars as the split bars. Let us first compute a bound on the expected error imposed by these two bars. The final boundaries that Algorithm 3 generates may cut at most $B - 1$ bars. We refer to the boundaries as bn_1, bn_2, ..., bn_{B-1} and to the bars they cut as the cut bars (b_1, b_2, ..., b_{B-1}). If both split bars are among un-cut bars they do not impose any error since they compensate for each other in the final bucket. Otherwise, one of them should be among b_i's. We will return to the error caused by the b_is after we consider the error caused by the second split bar. Let the second split bar be in interval [bn_i, bn_{i+1}). The maximum error it generates (called err_{sp}) can be computed as:

$$\frac{|bar_{ip+1}|}{|B_i|} \leq \frac{maxCoef.\alpha_\rho.\rho^{ip+1}.W}{\alpha_\phi.\phi^i.W} = \frac{maxCoef(1-\rho)\rho}{(1-\phi)}$$

The other part of the error is because of the the position of boundaries in the cut bars. Considering the estimated number of items at the left and right sides of boundary bn_i in bar b_i are respectively called l_i and r_i ($l_i + r_i = |b_i|$), the maximum error imposing by wrongly selecting the boundaries for B_i is then $(r_i + l_{i+1})/|B_i|$. Thus on aggregate for the cut bars error (err_{cut}), we have:

$$\begin{aligned}
\sum_{i=1}^{B-1} \frac{(r_i + l_{i+1})}{|B_i|} &\leq \sum_i \frac{(r_i + l_i)}{|B_i|} = \sum_i \frac{|b_i|}{|B_i|} \\
&\leq \sum_{i=1}^{B-1} \frac{maxCoef.\alpha_\rho.\rho^{ip}.W}{\alpha_\phi.\phi^i.W} \\
&= \sum_{i=1}^{B-1} \frac{maxCoef(1-\rho)}{(1-\phi)} \\
&= (B-1)\frac{maxCoef(1-\rho)}{(1-\phi)}
\end{aligned}$$

Knowing that each bar uses an EH sketch with approximation error δ, we can compute the following bound for the final expected approximation error for the buckets in our BSBH algorithm:

$$\begin{aligned}
ErrRate &= \delta + (err_{sp} + err_{cut})/B \\
&\leq \delta + \frac{maxCoef(1-\rho)\rho}{(1-\phi)B} + \frac{maxCoef(1-\rho)}{(1-\phi)} \\
&\leq \delta + c(\frac{\rho}{B} + 1)(1-\rho) < \delta + 2c(1-\rho) \qquad (3)
\end{aligned}$$

where c=$maxCoef/(1-\phi)$ is a constant. This proves that the expected approximate error is bounded. Calling this bound ϵ and noting that $\rho = \phi^{1/p}$, one can say that by setting $\delta = \epsilon/2$ ($k = \lceil 2/\epsilon \rceil$) and $p = \lceil -2c.log(\phi)/\epsilon \rceil$, we obtain a size-based expected ϵ-approximate low-biased histogram. To see this observe that:

$$p = \lceil \frac{-2c.log(\phi)}{\epsilon} \rceil \geq \frac{-log(\phi)}{\epsilon/2c} \geq \frac{log(\phi)}{log(1-\epsilon/2c)}$$

The last part of the above inequality is derived from the fact $e^x > 1 + x$ (where x can be set to $1 - \epsilon/2c$). Now by replacing this lower bound for p in Formula 3, the proofs will be completed. □

Empirical results show that even much smaller values for k and p can provide the same accuracy level. However, it is necessary to mention that the larger the expansion factor p is, the quicker the convergence of the algorithm will be especially for those data sets with concept shifts. Lemma 1 shows that the total number of bars (active EHs plus blocked EHs) in BSBH is $O(S_m)$. Additionally, the size of all of these bars is smaller than the window size, which indicates that each bar needs at most $\frac{1}{\delta}log(\delta W)$ of space. These two facts lead us to the following bound on the expected memory usage of BSBH:

THEOREM 2. *The expected memory usage of the BSBH algorithm is bounded by $O(\frac{S_m}{\delta}log(\delta W)) = O(\frac{B}{\epsilon^2}log(\epsilon W))$, where W is the sliding window size and ϵ is the approximation factor of BSBH algorithm.*

We can provide a bound on the per item CPU usage in the following theorem. Since the proof of this theorem is identical to the one in [13], we skip it in this paper.

THEOREM 3. *BSBH on average spends $O(log(S_m) + \frac{S_m}{S}) = O(log(\frac{B}{\epsilon}) + \frac{B}{\epsilon S})$ time for each input data item, where S is the slide size.*

Note that in practice and as shown in the experimental results, this time complexity is very close to a constant time, since S is usually much larger than S_m, and $log(S_m)$ is usually very small.

5. EXPERIMENTAL RESULTS

We implemented both versions of the BSBH algorithm in C++. The version of BSBH which blocks the bars in the merge operation is referred to as BSBH-BL, and the alternative approach is called BSBH-AL. We also compared BSBH with the CKMS algorithm proposed by Cormode et al. [3] for the case of no sliding windows, since CKMS does not support it. All the errors were computed using average size error of the histograms' bars as defined in Definition 3. For all the experiments shown in this section, we fixed the slide size at 1000 tuples and number of bars at 20. No prior knowledge of the minimum and maximum values of the incoming data was considered. We also set δ to 0.1, p to 7, and $maxCoef$ to 1.7 which are experimentally proved to be optimum in [13]. All the experiments were run on a 64bit, 2.27 GHz machine running CentOS with 4GB of main memory (RAM) and 8MB of cache.

5.1 Data sets

We have considered 10 data sets wherein one is a real-world data set (DS10), and the rest are synthesized data sets with different distributions as shown in Table 1. As you can see, the first six data sets

Table 1: Data sets used for the experimental results

Name	Dist.	Size	Shifts No.	Parameters
DS1	Uniform	$1m$	0	$min 0, max 10k$
DS2	Normal	$1m$	0	$\mu 5k, \sigma 500$
DS3	Zipfian	$1m$	0	$\alpha 1.5$
DS4	Zipfian	$1m$	0	$\alpha 1.8$
DS5	Exponential	$1m$	0	$\lambda 10^{-3}$
DS6	Exponential	$1m$	0	$\lambda 10^{-4}$
DS7	Zipfian	$1m$	100	starting $\alpha 1.5$
DS8	Zipfian	$1m$	10	alt. $\alpha 3$ & $\alpha 1.1$
DS9	Exponential	$1m$	100	starting $\lambda 10^{-3}$
DS10	Precipitation	$21m$	-	-

(DS1 to DS6) contain no concept shifts which means their distribution does not change through time. In DS7 and DS9, we shift the concept by randomly increasing or decreasing the distribution's parameters (respectively α and λ) by 10% for every 10K data items. We have also generated a data set with 10 unrealistically large concept shifts, called DS8. In this data set, the distribution alternates between Zipfian with $\alpha = 3.0$ and $\alpha = 1.1$. Our real data set contains the amount of daily precipitation recorded in different stations across the United States for about 400 years[5]. Although, there are many missing records for early years, this data set contains more than 21 million records. An estimation for the distribution of data in this data set (Figure 1) indicates that the distribution has a very long tail.

5.2 BSBH versus CKMS

Figures 5 and 6 provides the execution time, memory usage, and size error of the BSBH algorithms as well as CKMS algorithm for data sets DS3 and DS7 respectively. Both of these two data sets have Zipfian distribution (with α=1.5), but DS7 contains 100 random concept shifts. For CKMS, we set ϵ and ϵ_{min} respectively to 0.01 and 0.001 as suggested in [3]. As can be seen in the figure, both BSBH-BL and BSBH-AL are constantly faster, lighter (in the sense of memory usage), and more accurate than CKMS, even when the results for BSBH algorithms are taken over the whole history of data streams. For the current setting, BSBH is at least twice as fast as CKMS while using more than six times less memory and providing results with twice or more better accuracy. We should mention that CKMS provides a worst-case error guarantee while BSBH provides an expected error guarantee. This mainly explains the gap between memory and CPU usage of the two approaches. Note that the error increases for smaller ϕ's in both algorithms. This is mainly because for smaller ϕ's, bars closer to the biased regions get even smaller. As a result, even one misplaced item in those small bar may result in a very high error rate. This is actually why CKMS uses the flat error rate ϵ_{min} at the biased points.

5.3 Scalability

As discussed in previous sections, BSBH uses a constant number of Exponential Histograms and as a result, we claimed BSBH will inherit the scalability of the Exponential Histograms. To verify our claim, we tested the execution time, memory usage, and size error of our two BSBH algorithms over data sets DS3 and DS7 for different window sizes in Figure 7. Parts (a) and (b) of this figure depict that both CPU and memory usage of BSBH algorithms have logarithmic relation with respect to the size of the windows which proves the scalability of our structure for larger window sizes. In part (c) of the figure, we included the error which is steady at 2% for larger than 16K window sizes.

[5]Taken from http://www.ncdc.noaa.gov.

It is also worthy to compare the results for DS3 and DS7, in which the only difference is the presence of concept shift in DS7. For smaller window sizes, the errors (part (c) in Figure 7) are higher for both data sets, since there is not enough data in each bar to make a good estimation. DS7's error is even slightly higher due to several concept shifts. For larger window sizes, which are more probable in fast data streams, the accuracy for both data sets is almost the same. However, DS7 needs slightly more memory and time due to the larger number of split/merge operations. We discuss more results on data sets with concept shifts in Subsection 5.5.

5.4 Biased Sampling versus Uniform Sampling

To evaluate the effect of our proposed biased sampling technique, we compared the execution time, memory usage, and size error of BSBH algorithms for two types of sampling techniques; uniform sampling and biased sampling. In uniform sampling, we simply dropped items with a fixed probability. The results for data sets DS7 and DS10 are shown in Figures 8 and 9. The biased sampling rate (μ) and its equivalent uniform sampling rates (σ) are shown in the horizontal axis in the figures. As part (c)'s of the figures depict, the accuracy of the biased sampling is superior to that of uniform sampling, especially when the sampling rate increases. However, biased sampling needs slightly more memory than uniform sampling (part (b) of the figures) because fewer items from small bars are sampled out. This increases the number of boxes in small bars. Notice that small changes in the number of items in larger bars usually does not significantly change the number of boxes in them. As for the CPU usage s(part (a) of the figures), both techniques performs similarly. However, the interesting result is that by biasing the sampling rate (μ) to less than .98 none of these techniques significantly improve the CPU performance.

5.5 The Effect of Concept Shifts

As already shown in Figure 7, moderate concept shifts in the data stream have a very small effect on the performance of the BSBH algorithms. To take a closer look at the effect of concept shifts on the performance of BSBH, we compared the evolution of the size error through time for DS7, DS8, and DS9. The results (Figure 10) indicate that concept shifts have insignificant effect on the size error for data sets DS7 and DS9 in which we have reasonable concept shifts. This mainly shows that concept shifts of such as in DS7 and DS9 can be quickly (only in couple of slides a our experiments shows) resolved by the BSBH. However for DS8, the concept shifts effects are visible due to the unrealistic changes in the data distribution at each concept shifts. As it can be seen in the figure, the error increases right before the distribution changes from Zipf with $\alpha = 3.0$ to Zipf with $\alpha = 1.1$. The reason that the error is lower for $\alpha = 3.0$ is that BSBH receives less data items at the tail of the distribution of the current window under this case. This basically means that the boxes at the tail gradually shrink down in their size, and consequently they will not split but may merge. Since the merge operation does not impose any extra error, the overall error remains low for this case.

5.6 Discussion

Figure 11 depicts the performance of BSBH-BL without sampling as well as with biased sampling rates (μ) of 0.98 and 0.96 for all the data sets introduced in Table 1. There are some important points which can be understood from the figure. Perhaps the most prominent element is that the execution time for BSBH is almost independent of the distribution of the data sets. As you can see in part (a) of Figure 11, the execution time of all the data sets is proportional to the size of the data sets. This verifies our Theorem 3 in

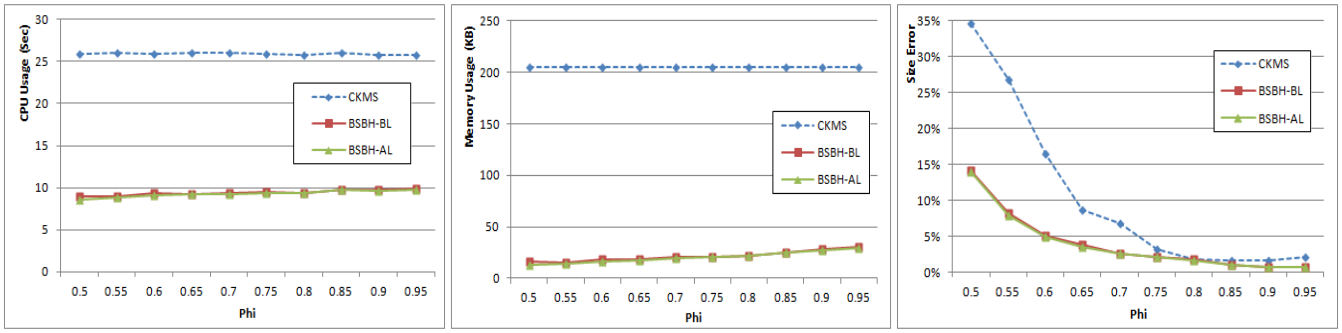

Figure 5: a) The execution time, b) memory usage, and c) size error of BSBH and CKMS algorithms on DS3. ($k = 10$, $p = 7$, $\epsilon = 0.01$, **and** $\epsilon_{min} = 0.001$.)

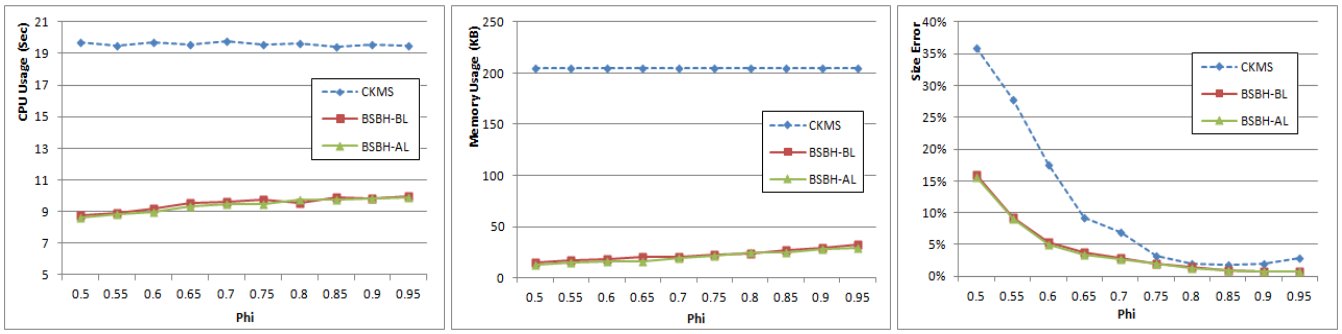

Figure 6: a) The execution time, b) memory usage, and c) size error of BSBH and CKMS algorithms on DS7. ($k = 10$, $p = 7$, $\epsilon = 0.01$, **and** $\epsilon_{min} = 0.001$.)

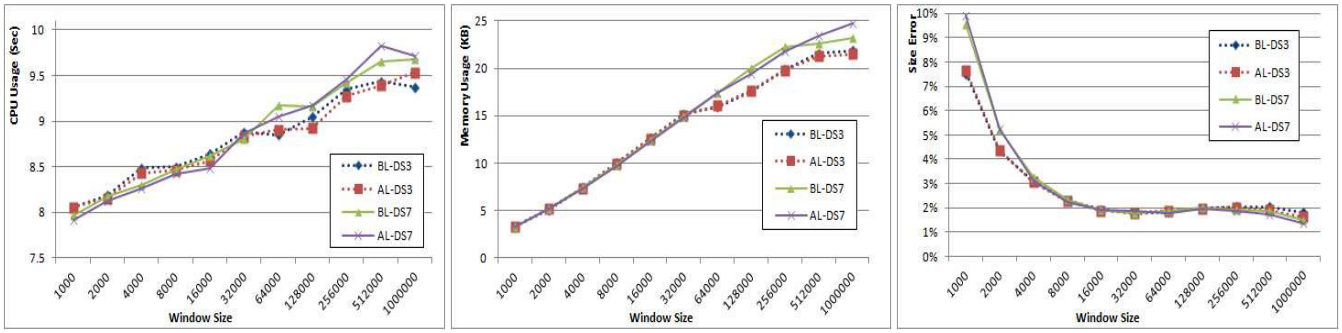

Figure 7: a) The execution time, b) memory usage, and c) size error of BSBH algorithms on DS3 and DS7 for different window sizes. ($\phi = .8$, $k = 10$, $p = 7$, $\epsilon = 0.01$, **and** $\epsilon_{min} = 0.001$.)

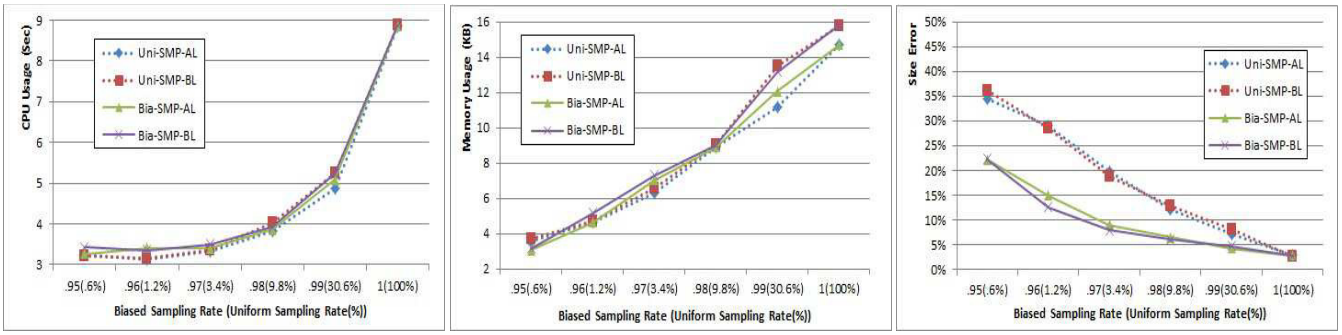

Figure 8: a) The execution time, b) memory usage, and c) size error of BSBH algorithms on DS7 for different sampling rates. ($\phi = .7$, $W = 100K$, $k = 10$, **and** $p = 7$.)

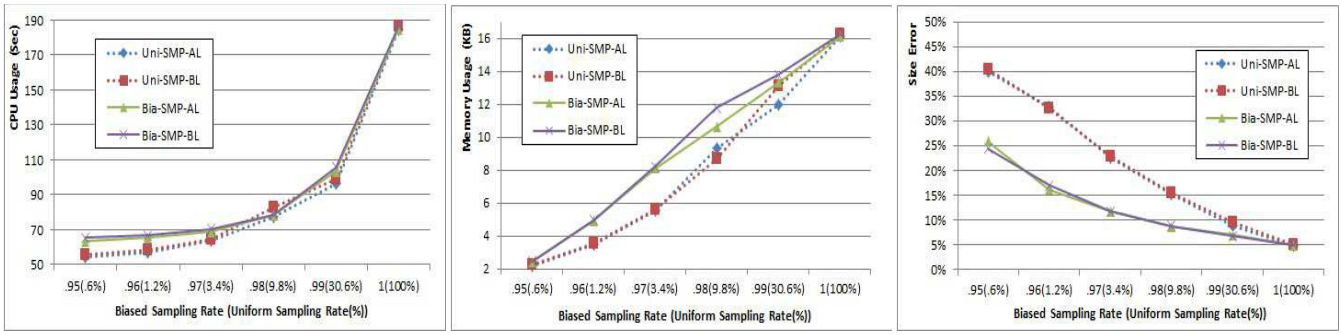

Figure 9: a) The execution time, b) memory usage, and c) size error of BSBH algorithms on DS10 for different sampling rates. ($\phi = .7$, $W = 100K$, $k = 10$, and $p = 7$.)

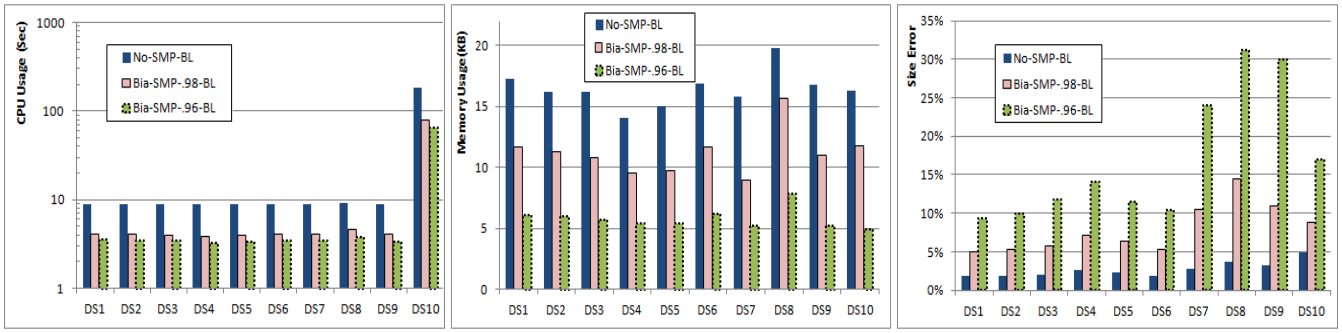

Figure 11: a) The execution time, b) memory usage, and c) size error of BSBH algorithms with different sampling rate for all data sets. ($\phi = .7$, $W = 100K$, $k = 10$, $p = 7$)

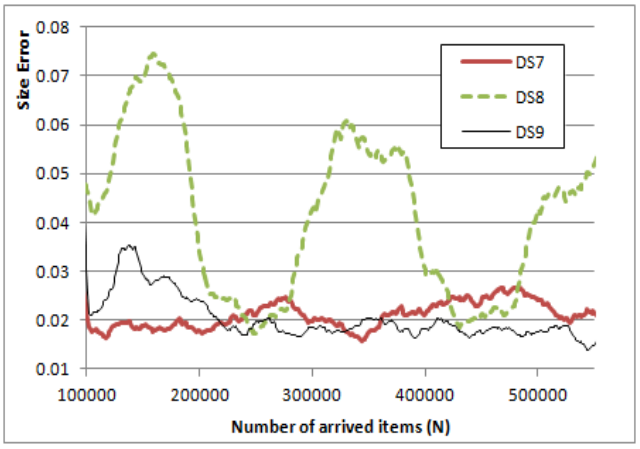

Figure 10: The size error of running BSBH-BL on data sets DS7 to DS9 through time. ($\phi = .7$, $W = 100K$, $k = 10$, $p = 7$)

previous section. As for the memory usage, part (b) of the figure suggests that the data distribution has a slight effect on the memory usage of BSBH. This is also in accordance with Theorem 2.

The other important point is that although the execution time and the memory usage of BSBH drops after sampling, this improvement is not very impressive especially for smaller sample rates. For instance with biased sampling rate of 0.96, which is equivalent to uniform sampling rate of 1.2%, we improve the memory usage and execution time at most by factors of 3 and 2 respectively, while degrading the accuracy by at least a factor of 7. This is actually not an unexpected result considering the compactness of our data structure. According to our experiments, partially shown in Figures 8, 9, and 11, not much is gained when we biased the sampling rate μ to be less than 0.98. Moreover, sampling on the data sets with concepts shifts has a worse effect on the accuracy of the results, as shown in part (c) of Figure 11.

6. RELATED WORK

The problem of designing quantiles and equi-depth histograms in databases has been studied for a long time [16], [5], [6]. In 1984, Shapiro and Connel introduced a method to estimate the selectivity of conditions in the form of $attribute\ \theta\ constant$ in a database system where θ can be one of $=$, $<$, $>$, $\leq$, and $\geq$ [16]. In 2002, Gilbert *et al.* used *Random Subset Sums* (RSSs) as a sketch to store summarized information about the whole database and estimate the quantile with a one-pass algorithm [6]. Gibbons *et al.* have also presented a sampling-based technique for maintaining approximate equi-depth histograms on relational databases [5].

Existing works on designing equi-depth histograms over data streams have mainly focused on the related problem of quantiles. Since computing exact quantiles with a single-pass algorithm requires to store all the data [14], most of these works try to approximately answer quantile queries with low space complexity. Manku *et al.* introduced an ϵ-approximate algorithm to answer any quantile query over the entire history of the data stream [11]. Later, they used non-uniform random sampling to improve their quantile computation for the case where the data set size is not known in advance [12]. Greenwald and Khanna, in [8], improved the memory usage

of the previously mentioned approach. Their work , which is sometimes referred to as the GK algorithm, was used in [10] and [1] to answer quantile queries over data streams with sliding windows. Both of these two approaches run several copies of the GK, and as a result they suffer from high time complexity.

Cormode *et al.* introduced the idea of biased quantiles in [2]. Based on the GK algorithm, they proposed a deterministic ϵ-approximate algorithm to construct biased quantiles over the whole history of a data stream. Their approach needs $O(\frac{1}{\epsilon}B \times log(1/\phi)log(\epsilon N))$ space, where N is the data stream size and B is the number of boundaries. Note that as shown in [21], the worst case behavior of this type of algorithms is linear in the universe size (The size of the items' range). Later, in [3], the same authors proposed a faster and more space-efficient deterministic algorithm for this problem, based on a binary tree structure idea borrowed from [18]. Needing $O(\frac{1}{\epsilon}logUlog(\epsilon N))$ space and an almost constant amortized cost of actions per new entry, the algorithm is best suited for high speed data streams. However, it can not be easily employed in the sliding window model for data streams. Moreover, their algorithm requires prior knowledge of U and the bound on space requirement depends on the U which is undesirable. Zhang and Wang have used a decomposable structure to construct ϵ-approximate biased quantiles using $O(\frac{log^3(\epsilon N)}{\epsilon})$ space and $O(log(\frac{log(\epsilon N)}{\epsilon}))$ time. However, they used a naive non-uniform sampling technique which needs to sort thse entire data set in advance.

Unfortunately, the aforementioned biased quantile computation algorithms can not be easily used for the sliding window case, since their underlying structures do not support the idea of expirations. The usual solution for this issue is to run several copies of the same algorithms for different-sized chunks of the most recent part of the current window, and combine the results for the biggest possible parts which are not yet expired. A similar method is used in [1] for regular quantiles that is proven to be very slow [20] [13].

7. CONCLUSION

In this paper, we proposed an expected ϵ-approximate biased histogram algorithm, called Bar Splitting Biased Histogram (BSBH). We formally proved that BSBH time and memory requirements are bounded and independent from the data streams' size when no concept shift is present. The experimental results confirmed said theoretical bounds and showed that the approach is scalable with respect to the sliding window size. The paper also introduced a new biased sampling technique which outperforms the uniform sampling technique in terms of accuracy, while using approximately the same amount of CPU and memory. The experimental results also show that BSBH handles quite effectively concept shift, although formal characterization for this case was left for the future. Future work will also address the problem of automatically detecting the biased points in data distributions, according to the distribution of the data, the distribution of the users' queries, or both.

8. ACKNOWLEDGMENTS

We would like to sincerely thank Graham Cormode and Flip Korn for providing us with the source code of their biased quantile algorithm proposed in [3].

9. REFERENCES

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

[2] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Effective computation of biased quantiles over data streams. In *ICDE*, pages 20–31, 2005.

[3] 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.

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

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

[6] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *VLDB*, pages 454–465, 2002.

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

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

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

[10] 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.

[11] 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.

[12] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *SIGMOD Conference*, pages 251–262, 1999.

[13] H. Mousavi and C. Zaniolo. Fast and accurate computation of equi-depth histograms over data streams. In *EDBT*, pages 69–80, 2011.

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

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

[16] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *SIGMOD Conference*, pages 256–276, 1984.

[17] 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.

[18] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: New aggregation techniques for sensor networks. *CoRR*, cs.DC/0408039, 2004.

[19] Q. Zhang and W. Wang. An efficient algorithm for approximate biased quantile computation in data streams. In *CIKM*, pages 1023–1026, 2007.

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

[21] Y. Zhang, X. Lin, J. Xu, F. Korn, and W. Wang. Space-efficient relative error order sketch over data streams. In *ICDE*, page 51, 2006.