Augmented Sketch: Faster and More Accurate Stream Processing

Pratanu Roy*
Systems Group Computer Science
ETH Zurich
pratanu@inf.ethz.ch

Arijit Khan†
School of Computer Engineering
NTU Singapore
arijit.khan@ntu.edu.sg

Gustavo Alonso
Systems Group Computer Science
ETH Zurich
alonso@inf.ethz.ch

ABSTRACT

Approximated algorithms are often used to estimate the frequency of items on high volume, fast data streams. The most common ones are variations of Count-Min sketch, which use sub-linear space for the count, but can produce errors in the counts of the most frequent items and can misclassify low-frequency items. In this paper, we improve the accuracy of sketch-based algorithms by increasing the frequency estimation accuracy of the most frequent items and reducing the possible misclassification of low-frequency items, while also improving the overall throughput.

Our solution, called Augmented Sketch (ASketch), is based on a pre-filtering stage that dynamically identifies and aggregates the most frequent items. Items overflowing the pre-filtering stage are processed using a conventional sketch algorithm, thereby making the solution general and applicable in a wide range of contexts. The pre-filtering stage can be efficiently implemented with SIMD instructions on multi-core machines and can be further parallelized through pipeline parallelism where the filtering stage runs in one core and the sketch algorithm runs in another core.

Keywords

data streams; sketch; approximated algorithms; data structures; stream summary

1. INTRODUCTION

In scenarios such as real-time IP traffic, phone calls, sensor measurements, web clicks and crawls, massive amounts of data are generated as a high-rate stream [10, 26]. Processing of such streams often requires approximation through succinct synopses created in a single-pass [8]. These synopses summarize the streams to give an idea of the frequency of items, using a small amount of space, while allowing to process the stream fast enough. Due to the smaller size of the synopsis compared to the original stream size, there is a trade-off among space, accuracy, and efficiency: reducing the space increases the efficiency but affects accuracy.

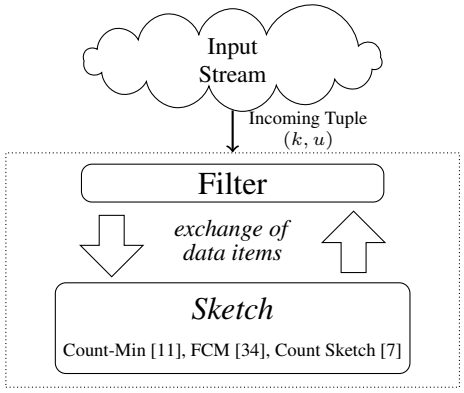


Figure 1: ASketch exchanges data items between the filter and the sketch such that high-frequency items are kept in the filter.

In this paper, we study the problem of frequency estimation over data streams that require summarization: given a data item as a query, we estimate the frequency count of that item in the input stream. In the literature, sketch data structures [3, 7, 11, 12, 18, 22] are the solution of choice to address this problem. By using multiple hash functions, sketches summarize massive data streams within a limited space. However, due to summarization, the approach can give inaccurate results. First, it can give an inaccurate count for the most frequent items. Second, it can misclassify low-frequency items and report them as high-frequency ones. Since the high-frequency items are usually more relevant in many applications including load-balancing and heavy-hitter queries [9, 35], it is important to estimate the frequency of the most frequent items *as accurately as possible*. In the literature, there are many examples of where lower accuracy on the frequency estimation impacts the subsequent analysis of the data. For instance, it was shown in [35] that due to the lower accuracy provided by traditional sketches, an algorithm trying to balance the load will result in biases and unbalanced job distribution. Similarly, in case of natural language processing (NLP) applications such as sentiment analysis, a typical application of sketches is to compute the pointwise mutual information (PMI), and subsequently find the top-k positive and negative words [19]. Without an accurate ranking, words can easily be misclassified, leading to wrong interpretations. To find out the level of accuracy of existing solutions, we constructed a Count-Min sketch summary of size 128KB from a real-world *IP-trace* stream data, originally containing 461M tuples. This results in the estimated frequency of the most frequent item to be 18 018 217, whereas its

*Author's new affiliation: Oracle Labs, Zurich, Email: pratanu.r.roy@oracle.com.

†This work has been done while the author was at ETH Zurich.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882948

	Stream Processing Throughput Updates/milisecond	*Frequency-Estimation Query Throughput* Queries/milisecond	*Frequency-Estimation Error* Observed error (%)
Count-Min [11]	6481	6892	0.0024
Frequency-Aware Count [34]	6165	7551	0.0013
Holistic UDAFs [10]	17508	6319	0.0025
ASketch [this work]	**26739**	**30795**	**0.0004**

Table 1: ASketch vs. other sketch-based methods: ASketch is implemented on top of Count-Min with a filter capacity of 32 items. All methods are allocated 128KB of space. The stream size is 32M, and the number of distinct data items is 8M. The stream data has a Zipf frequency distribution with skew 1.5. For details about various parameters, query setting, and evaluation metrics, see Section 7.

true frequency is 17 978 588. Similar margin of errors also occur for other high-frequency items in the *IP-trace* stream, as well as in other stream datasets that we experimented with. Although the error may seem small, it can create problems in applications checking thresholds, doing bookkeeping, and establishing ranking as just discussed.

In addition, due to the *one-sided error guarantee* (i.e., the estimated frequency of an item is at least as high as its true frequency) ensured by many sketches, a low-frequency item can misleadingly appear as a high-frequency item in the sketch synopsis. This is indeed the case found in our empirical evaluation. When we constructed a 16KB Count-Min synopsis of a synthetic stream having a Zipf frequency distribution, we find that 27 low-frequency items are misclassified as very high-frequency items, with an average relative error in the order of 10^5 for these misclassified items. Such misclassifications can be detrimental to various machine learning applications. For instance, it was shown in [36] that since frequent items mining is the first step of frequent itemsets mining, even a small number of misclassified items leads to a large number of false-positive itemsets, which makes effective frequent itemsets mining from summarized streams difficult. A similar situation arises in sketch-based clustering and classifications, often used in suspicious events detection, where a slight improvement in prediction accuracy can result in a huge improvement [1, 2].

In this paper, we propose a way to solve these problems by complementing existing techniques and without requiring a radically different solution. ASketch (Augmented Sketch) is a stream processing framework, applicable over many sketch-based synopses, and coupled with an adaptive pre-filtering and early-aggregation strategy. It utilizes the skew[1] of the underlying stream data to improve the frequency estimation accuracy for the most frequent items by filtering them out earlier. It also improves the overall throughput — while using *exactly* the same amount of space as traditional sketches. The idea of ASketch is illustrated in Figure 1. All items in the stream pass through a filtering stage. The filter retains the high-frequency items and the sketch processes the tail of the distribution. The throughput improvement comes from the skewed distribution and extremely fast lookup time of the filter (e.g., with *vector instructions*). Catching high-frequency items with a filtering stage also improves the accuracy of the frequency estimation for the frequent items. For example, ASketch of size 128KB reports the estimated frequency of the most frequent item to be exactly 17 978 588 in the *IP-trace* stream. At the same time, separation of high frequency items from the sketch reduces misclassifications. In our experiments, we do not find any instance of misclassified low frequency items, when using an ASketch synopsis of the same size.

Our contributions can be summarized as follows.

- We explore the use of a filter to catch heavy hitters earlier and remove them from the subsequent sketch data structure. The resulting design, ASketch, shows how to implement such filter efficiently so that not only *accuracy improves* but also *overall throughput* (Section 5).
- Since introducing a filter in front of a sketch data structure changes the properties of the latter, we provide an extensive analysis of ASketch, and how the different parameters of the design affect accuracy and performance (Section 4).
- Modern hardware offers several possibilities for increasing the efficiency of the filtering stage. In the paper, we explore possible parallel implementations using separate cores, using SIMD instructions, and in an SPMD (single program, multiple data) parallel model (Section 6).
- The performance analysis showing the advantages of ASketch, also shed light on the differences between existing solutions and how the filter interacts with different forms of sketches. We compare ASketch with Count-Min [11], Holistic UDAFs [10], Frequency-Aware Counting [34], and Space Saving [27] (Section 7).

[1] Manerikar et. al. [26] reported Zipf skew $z \geq 1.4$ in real-world datasets.

2. RELATED WORK

The problem of synopsis construction has been studied extensively [8] in the context of a variety of techniques such as sampling [15], wavelets [16, 17], histograms [20], sketches [3, 7, 11, 12, 18, 22], and counter-based methods, e.g., Space Saving [27] and Frequent [9]. Among all these techniques, sketches and counter-based approaches are widely used for stream data summarization. Sketches are typically used for frequency estimation. Counter-based data structures are designed for finding the top-k frequent items. Sketches keep approximate counts for all items, counter-based approaches maintain approximate counts only for the frequent items. Sketches can support top-k queries with an additional heap [7] or a hierarchical data structure [8]. However, counter-based approaches achieve faster update throughput than sketches for the top-k estimation [9], as they do not count the frequency of all items.

Several systems have been proposed either to improve the stream processing throughput of sketches, e.g., Holistic UDAFs [10], or to improve their query processing accuracy, e.g., Frequency-Aware Counting [34] and gSketch [37]. ASketch improves both accuracy and throughput with the help of pre-filtering and early-aggregation of high frequency items. Approaches like Holistic UDAFs and Frequency-Aware Counting use additional data structures, increasing the overall storage requirement, which is often significant with respect to small-space sketch data structures. As an example, we empirically compare in Table 1 the efficiency and accuracy of ASketch with several existing techniques, e.g., Count-Min [11], Holistic UDAFs [10], and Frequency-Aware Counting [34], while allocating the same amount of space. The results demonstrate the advantages of ASketch.

The idea of separating high-frequency items from skewed data distribution is not new [15]. Modern processors employ caches on top of main memory to capture the locality of access and to improve

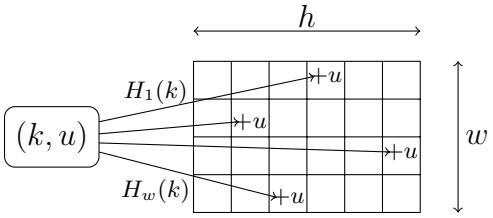


Figure 2: Count-Min.

the overall throughput. Skimmed Sketch [14] observed that major part of the error in join size estimation with sketches appears due to the collisions of high frequency items with low frequency items, and therefore, proposed to skim-off the high frequency items in a post-processing manner with the help of a heap [7].

The pre-filtering and early-aggregation strategy has been effectively used in many applications, e.g., duplicate elimination [23], external merge sort [5], Space Saving synopsis construction [32], etc. String matching techniques are utilized to improve XML pre-filtering so that the memory consumption and processing time decreases [21]. In [24], pre-filtering has been effectively used to prune irrelevant events from the ones that are being monitored in a complex event detection application. In both of these cases [21, 24], pre-filtering is used to eliminate data. ASketch uses a pre-filtering strategy to separate common items from the less common ones for sketch data structures.

3. PRELIMINARIES

A data stream of running length n is a sequence of n tuples. The t-th tuple is denoted as (k_t, u_t), where k_t is a data-item key used for hashing and u_t is a frequency value associated with the data item. Usually the values for k_t are drawn from a large domain (e.g, IP addresses, URLs, etc.) and the value u_t is a positive number[2,3]; hence, the frequency counts of data items increase monotonically.

Sketches are a family of data structures for summarizing data streams. Sketches bear similarities to Bloom filters [6, 31] in that both employ hashing to summarize data; however, they differ in how they update the hash buckets and use these hashed data to derive estimates. Among the various sketches available [3, 7, 11, 13, 18, 22, 25], it has been shown [9, 33, 34] that Count-Min [11] achieves the best update throughput in general, as well as high accuracy on skewed distributions. Therefore, we shall discuss ASketch on top of Count-Min. Nevertheless, ASketch *is generic and can be applied in combination with other sketches*. To demonstrate it, we also evaluate the performance of ASketch with other underlying sketches. Next, for the ease of presentation, we introduce Count-Min.

Count-Min. In Count-Min, hashing is used to approximately maintain the frequency counts of a large number of distinct items in a data stream. We use w pairwise independent hash functions, each of which maps onto uniformly random integers in the range $[0 \ldots h-1]$. The data structure itself consists of a 2-dimensional array with $w \cdot h$ cells of length h and width w. Each hash function corresponds to one of w 1-dimensional arrays with h cells. The hash functions are used to update the counts of different cells in this 2-dimensional data structure (Figure 2).

In order to *estimate* the count of an item, we determine the set of w cells to which each of the w hash-functions maps onto, and compute the minimum value among all these cells. Let c_t be the true value of the count being estimated. The estimated count is at least equal to c_t, since we are dealing with non-negative counts only, and there may be an over-estimation because of collisions among hash cells. As it turns out, a probabilistic upper bound to the estimate can be determined [11]. For a data stream with N as the sum of the counts of the items received so far, the estimated count is at most $c_t + (\frac{e}{h})N$ with probability at least $1 - e^{-w}$. Here, e is the base of the natural logarithm.

[2]Sketches, as well as ASketch, also allow us to model the removal of data items (see Appendix). However, for simplicity, we shall currently assume a *strict distributions* [29], that is, the stream updates are positive.

[3]For simplicity, we assume $u_t = 1$. Our analysis can easily be extended for larger values of u_t.

4. MOTIVATION FOR FILTERING AND AGGREGATING FREQUENT ITEMS

We analytically compare the frequency estimation error, frequency estimation time, and stream processing throughput of both Count-Min and ASketch. Let us denote by N the aggregate of the counts of all items in the data stream.

Count-Min Properties. Consider a Count-Min with w pairwise independent hash functions, each of which maps onto uniformly random integers in the range $[0 \ldots h-1]$. The time required to insert an item in Count-Min is $t_s = \mathcal{O}(w)$. This is also same as the time required to answer a frequency estimation query. The expected error in the frequency estimation [11] is at most $(\frac{e}{h})N$, with probability at most e^{-w}, where e is the base of the natural logarithm.

How is a Filter Accommodated in an ASketch? ASketch augments the traditional Count-Min with a filter. Let us assume that the filter consumes s_f space, while the time required to insert an item (as well as query an item) in the filter is t_f. Usually, s_f is very small and due to the hardware conscious implementation of the filter (explained in Section 6.1), $t_f << t_s$. To accommodate the filter in ASketch, the space from the underlying sketch data structure is reduced so that the overall space for ASketch stays the same as that of the Count-Min alone. Reduction of space from the underlying sketch can be achieved either by reducing the number of hash functions, or by reducing the range of each hash function, or by both. More specifically, let w' and h' be the number of hash functions and the range of each hash function in ASketch, respectively. Then, we have: $s_f + w'h' = wh$. In our implementation, we fix $w' = w$, and reduce h' to $(h - \frac{s_f}{w})$, due to two reasons: (1) Usually, $h > w$. Hence, for the same amount of change in h or w, the former will have a lower impact in reduction of storage space consumed by the underlying sketch. Therefore, updating h is more flexible to accommodate various sizes of filter in ASketch. (2) By having the same w number of hash functions for both Count-Min and ASketch, we keep the error bound probability e^{-w} identical.

Impact of the Filter on Throughput of ASketch. ASketch update time (as well as query time) can be expressed as: $t_f + \mathsf{filter}_{\mathsf{selectivity}} * t_s$. Here, t_f and t_s are update times of the filter and the sketch, respectively. We define $\mathsf{filter}_{\mathsf{selectivity}}$ as the ratio of the count of data items that overflow the filter with respect to the total count of all data items. In the case of ASketch updates, there is an additional factor which involves *exchanging of data items* between the filter and sketch (described in our Algorithm 1). Experiments show that the number of exchanges is rather small with respect to the overall stream size and hence, we ignored it here, but analyze it in more detail in the Appendix. We evaluate the impact of this factor empirically in Section 7.

Clearly, for ASketch, we increase the processing time by t_f; however, in return, we reduce the sketch processing time by a factor of $\mathsf{filter}_{\mathsf{selectivity}}$. A small size of the filter (i.e., s_f) and vectorized

	Count-Min	ASketch
Freq. Estimation Time	t_s	$t_f + \left(\frac{N_2}{N}\right) t_s$
Stream Processing Throughput	$\frac{1}{t_s}$	$\frac{1}{t_f + \left(\frac{N_2}{N}\right) t_s}$
Freq. Estimation Error	$\left(\frac{e}{h}\right) N$ with prob. e^{-w}	$\left(\frac{e}{h - \frac{s_f}{w}}\right) N_2 \left(\frac{N_2}{N}\right)$ with prob. e^{-w}
Supported Queries	Frequency Estimation	Frequency Estimation, Top-k Frequent Items

Table 2: Analytic comparison between Count-Min and ASketch: w the number of hash functions, h the range of each hash function, s_f the size of the filter, t_s data-item insertion time in Count-Min, t_f data-item insertion time in the filter, N total aggregated stream size, N_1 stream size stored in the filter, $N_2 = N - N_1$, e base of natural logarithm.

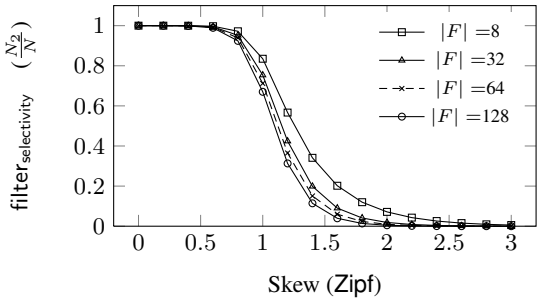


Figure 3: Filter selectivity: data stream with Zipf distribution, stream size 32M, number of distinct data items 8M. We vary the filter size $|F|$, i.e., the number of high-frequency items stored in the filter.

execution (explained in Section 6.1) keeps t_f small, in the range of a few assembly instructions. Therefore, $\text{filter}_{\text{selectivity}}$ essentially determines the overall effectiveness of ASketch.

Let us assume that out of N aggregated stream-counts, N_1 counts are processed by the filter and $(N - N_1) = N_2$ counts are processed by the sketch. Therefore, $\text{filter}_{\text{selectivity}} = \frac{N_2}{N}$. However, most real world streams exhibit skew [26]. For a Zipf distribution, we can measure the $\text{filter}_{\text{selectivity}}$ easily in the closed form by summing up the frequency counts of the top-k items [4, 32]. To demonstrate the potential effectiveness[4] of the ASketch framework, we have plotted the filter size ($|F|$) and the $\text{filter}_{\text{selectivity}}$, i.e., $\frac{N_2}{N}$ in Figure 3 on a data stream of 32M items with Zipf distribution. For a skew of 1.5, the top-32 data items account for 80% of all frequency counts, i.e., only $\frac{N_2}{N} = 20\%$ items are forwarded to the underlying sketch data structure. Therefore, we expect that the overall stream processing time of ASketch would be smaller than traditional sketches. One interesting observation from Figure 3 is that increasing the filter size beyond a certain threshold does not reduce the $\text{filter}_{\text{selectivity}}$ significantly. However, increasing the filter size, depending on the filter implementation (explained later in Section 6.1), may increase the filter processing time (t_f). Therefore, the filter in ASketch must consume a small space in order to achieve the maximum throughput gain.

Impact of the Filter on Accuracy of ASketch. ASketch improves the frequency estimation accuracy for the frequent items that are stored in the filter. In fact, if one can store the *exact* frequency counts for the items in the filter, their frequency estimation error would be zero. For the items that are stored in the sketch, we reduce their collisions with the high-frequency items, as the items stored in the filter are no longer hashed into the sketch. This reduces the possibility that a low-frequency item would appear as a high-frequency item, therefore reducing the misclassification error. However, in order to accommodate space for the filter, we now utilize a sketch structure which has a smaller size than that of the original Count-Min. This increases the frequency estimation error for low-frequency items stored in the sketch, often within a tolerable margin (see Theorem 1). Nevertheless, as mentioned earlier, it is often the frequency estimation accuracy for the high-frequency items which is more critical in actual applications (e.g., frequent pattern mining, outlier detection, load-balancing, and heavy-hitter detection) and ASketch significantly improves the frequency estimation accuracy for the high-frequency items stored in the filter. Let us assume that the frequency estimation queries are obtained by sampling the data items based on their frequencies, that is, the high-frequency items are queried more than the low-frequency items in a skewed stream. Then, the expected error in frequency estimation due to ASketch is given by: $(\frac{e}{h - \frac{s_f}{w}}) N_2 (\frac{N_2}{N})$, whereas the expected frequency estimation error due to Count-Min will be: $(\frac{e}{h})N$ for the same amount of storage space. Clearly, ASketch will result in a smaller expected error, since $N_2(\frac{N_2}{N}) << N$ in a skewed stream.

In the following theorem, we provide a probabilistic bound of the increase in error for the low-frequency items that reside in the underlying Count-Min sketch of ASketch, due to the reduction in Count-Min sketch size.

THEOREM 1. *The increase in error ΔE for a low-frequency item that resides in the underlying* Count-Min *sketch of* ASketch*, due to the reduction in* Count-Min *sketch size, is bounded with probability of at least $1 - e^{-w}$ as follows:*

$$\Delta E \leq (\frac{e s_f}{w h (h - \frac{s_f}{w})}) N \quad (1)$$

The proof can be found in the Appendix. However, one can verify that the upper bound is reasonably small even for a large size stream. In reality, the increase in error for the low-frequency items is even smaller, because the high-frequency items which constitute a significant fraction of the aggregated stream frequency, are stored in the filter, and they are not hashed into the sketch.

Summary. For a fixed ASketch size, we summarize the *trade-offs among filter size, throughput, and frequency estimation accuracy*.

- By having a larger filter, one can potentially store more high-frequency items in the filter; and this, in turn, will result in better frequency estimation accuracy for more high-frequency items. However, if we consider the expected frequency estimation error (given in Table 2), it consists of the errors due to both high-frequency and low-frequency items in the data stream. Since the high-frequency items are queried more than the low-frequency items, we find that the expected error initially decreases with the increasing filter size; but beyond a threshold filter size, it stops decreasing. This is because with increasing filter size, while we have better accuracy for more high-frequency items, we also introduce more error over the low-frequency items due to the reduced size of the underlying sketch.
- For the implementations considered, increasing the filter size increases the filter processing time t_f, without too much gain in the $\text{filter}_{\text{selectivity}}$. Therefore, beyond a threshold filter size, the throughput starts decreasing.

[4]Note that the k-items in the filter are not always guranteed to be the top-k frequent items; but based on the empirical results (Table 5), we achieve high accuracy for the top-k frequent items.

We empirically explore these trade-offs in Section 7.

5. ALGORITHM FOR ASKETCH

ASketch exchanges data items between the filter and the sketch, so that the filter stores the most frequent items of the observed input stream. Such movement of data items is a challenging problem for several reasons. First, one needs to ensure that the exchange procedure does not incur too much overhead on the normal stream processing rate. Second, it is difficult to remove the item, along with its frequency count, from the underlying sketch. This is because Count-Min only provides an over-estimation of the true frequency of an item; if we remove the over-estimated frequency count of an item from the sketch, it may violate the *one-sided accuracy guarantee* provided by the sketch. We illustrate this problem with an example:

EXAMPLE 1. *Let us assume that the data item A is currently stored in* Count-Min *and the hash cell corresponding to some hash function H_i provides the smallest frequency estimation for A. The removal of A from the sketch, along with its estimated frequency, will result in a count value of 0 in the hash cell corresponding to $H_i(A)$. Now, consider all items B that are also currently stored in* Count-Min*, such that $H_i(B) = H_i(A)$. After the removal of A from the sketch, the estimated frequency of all such B will become 0, an underestimation of their true frequencies.*

In ASketch, we want to maintain the one-sided accuracy guarantee of Count-Min, i.e., *the estimated frequency of an item reported by* ASketch *must always be an over-estimation of its true frequency*. We achieve this accuracy guarantee by introducing two different counts in the filter as described below.

Algorithm Description. ASketch consists of two data structures: *filter* (F) and *sketch* (CMS). The filter stores a few high-frequency items and two associated counts, namely new_count and old_count for each of the items that it monitors. The new_count denotes the estimated (over-estimation) frequency of an item that is currently stored in the filter. The difference between new_count and old_count represents the (exact) aggregated frequency that have been accumulated during the time a particular item resides in the filter. The sketch is implemented as the classical Count-Min, which is a two dimensional array with length (i.e., range of each hash function) h and width (number of hash functions) w, as shown in Figure 2.

The *stream processing algorithm* is shown in Algorithm 1. Every incoming tuple in the stream is represented as (k, u), where k is the key of the data item, and u is a frequency value. We first perform a lookup for the key k in the filter (lines 1-6, Algorithm 1). If k does not exist in the filter and there is an empty cell in the filter, then k is inserted in the filter with its old_count set to 0, and new_count as u (lines 4-6). Next time, whenever we find the key in the filter, we only update its new_count value without changing the old_count value (lines 2-3). If k is not found in the filter, and the filter is full (lines 7-18, Algorithm 1), then the data item is sent to the sketch. We insert the item in the sketch with the traditional sketch updating protocol, that is, k is hashed into w different rows by w different hash functions, where every hash function maps k in a range of h hash values (line 8).

Whenever we insert an item in the sketch, we get an estimate (over-estimation) of its true frequency. Now, we additionally keep track of the smallest frequency value (i.e., new_count) of any item currently stored in the filter. Thus, if the estimated frequency of the last item inserted into Count-Min is higher than the smallest

Algorithm 1 Stream processing algorithm for ASketch

Ensure: insert tuple (k, u) into ASketch
1: lookup k in filter
2: **if** item found **then**
3: new_count[k] $\rightarrow$ new_count[k] + u
4: **else if** filter not full **then**
5: new_count[k] $\rightarrow u$
6: old_count[k] $\rightarrow 0$
7: **else**
8: update sketch with (k, u)
9: **if** (estimated freq[k] $>$ min freq[F]) **then**
10: find minimum freq item k_i in F
11: **if** (new_count[k_i]-old_count[k_i])>0 **then**
12: update sketch with (k_i,new_count[k_i]-old_count[k_i])
13: **end if**
14: add k to filter
15: new_count[k] $\rightarrow$ estimated freq[k]
16: old_count[k] $\rightarrow$ estimated freq[k]
17: **end if**
18: **end if**

Algorithm 2 Query processing algorithm for ASketch

Ensure: estimate frequency of item k using ASketch
1: lookup item k in filter
2: **if** item found **then**
3: return new_count[k] from filter
4: **else**
5: return estimated freq[k] from sketch
6: **end if**

frequency value in the filter, we initiate an exchange (lines 9-17, Algorithm 1). The last item that was hashed into the sketch is now moved to the filter with both its old_count and new_count set as its estimated count obtained from the sketch. However, we do not make any change in the values inside the sketch. To accommodate this item in the filter, the item with the smallest frequency in the filter is now inserted into the sketch; but we only hash its (new_count-old_count) frequency into the sketch (lines 11-12). This is because its old_count frequency can either be 0, which indicates that the item was never placed into the sketch; otherwise, its old_count frequency is already contained in the sketch since the time the item was last moved from the sketch.

The key insight here is that we can *exactly* capture the hits for an item that is currently stored in the filter by using its (new_count-old_count) value. This has three benefits: (1) By accumulating this count in the filter, we save the cost of applying multiple hash functions in the sketch, and thus, improve the *throughput*. (2) As high-frequency items in a skewed data stream remain in the filter most of the time, we get more *accurate* frequency counts for these items. (3) The *misclassification rate* for the low-frequency items as high-frequency items also decreases, since the aggregate of the (new_count-old_count) values of the high-frequency items are not hashed into the underlying sketch .

The *query processing algorithm* is shown in Algorithm 2. For an incoming query requesting for the frequency of an item k, the algorithm first performs a lookup in the filter, and if finds the item, then returns the new_count (line 2-3); otherwise, the estimated frequency from the underlying sketch is returned (line 5).

Exchange Policy. In this section, we discuss the efficiency of the exchange policy. First, we need to keep track of the smallest frequency count in the filter. The efficiency of this step depends on the underlying data structure and the hardware implementation of the filter, which will be discussed shortly in Section 6.1. Second, we empirically found that we require a relatively small number of

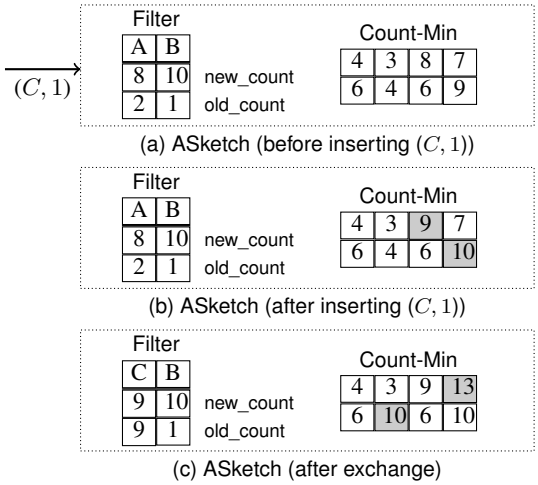


Figure 4: Example of stream processing by ASketch.

exchanges compared to the overall stream size in order to ensure that the high-frequency items are stored and early aggregated in the filter. As an example, even for the Uniform distribution (i.e., skew is 0), we need only about 40K exchanges for a stream of size 32M, ASketch size of 128KB, and filter size of 32 items (Figure 9). When the skew increases, the number of exchanges decreases as shown in our experimental results. Third, and perhaps more importantly, can our exchange policy trigger *multiple exchanges* between the filter and the sketch? Our exchange mechanism may indeed initiate multiple exchanges; for example, when the lowest-frequency item from the filter is moved to Count-Min, its estimated frequency (which will now be an over-estimation due to hash-based collisions) could be higher than that of the second least-frequent item in the filter. This will initiate yet another exchange. However, a careful examination reveals that multiple exchanges that happen as an after-effect of the first exchange are unnecessary and they introduce additional errors in the frequency estimation. Hence, whenever we hash an item in Count-Min because it was not found in the filter (or because the filter was full), we always restrict ourselves to *at most one exchange* between the filter and the sketch. The following lemma guarantees an upper bound on the number of times an item is inserted into the underlying sketch in the presence of our exchange policy, thereby ensuring the overall efficiency of ASketch.

LEMMA 1. *Let us assume that some data item has been appeared total t times in the stream. Then, the number of times we insert that data item in the underlying sketch by following* ASketch *algorithm is no more than t.*

Proof of the lemma follows from the fact that, in the best case, data items are early-aggregated in the filter before being inserted into the sketch. Coupled with the characteristics of our exchange policy, that is, (1) we do not perform multiple exchanges between filter and sketch for each insertion in the sketch, and (2) if the difference between the new_count and the old_count for the lowest-frequency item in the filter is zero, we do not perform any update in the sketch (line 11, Algorithm 1); we ensure that the above lemma is satisfied. While we demonstrate in our experiments (Section 7) that the number of exchanges is usually very small, we also provide theoretical bounds on the exchange policy in the Appendix.

Below, we demonstrate the key steps of our stream processing algorithm (Algorithm 1) with an example.

EXAMPLE 2. *In Figure 4, we show in three steps how a data tuple $(C, 1)$ is inserted into* ASketch. *Since the filter is full and it does not contain the data item C, the sketch is updated. We shaded the affected cells due to this update in Figure 4 (b). However, the estimated frequency of C, which is 9, is higher than the smallest frequency stored in the filter (i.e., 8 for item A). Hence, we move C into the filter with both its* old_count *and* new_count *assigned to its estimated frequency (is shown in Filter in Figure 4 (c)). We do not change anything in the sketch cells due to the movement of C. Finally, we insert A into the sketch; however, only the difference of its* (new_count-old_count) 6*, is inserted into the sketch (is shown in* Count-Min *in Figure 4 (c); affected cells are highlighted). Although, the current estimated frequency of A is 10, which is larger than the current smallest count in the filter, we do not initiate any other exchanges.*

6. ASKETCH ON MULTI-CORE

We now discuss the implementation of ASketch in modern multi-core hardware with an emphasis on (1) SIMD parallelism in the filter implementation, (2) pipeline parallelism in the ASketch framework, as well as (3) SPMD parallelism by implementing ASketch as a counting kernel in a multi-core machine.

6.1 Filter with SIMD Parallelism

Continued from the discussion in Section 4, we are aiming for a filter that is small in size. It must support the following two operations efficiently: (1) *lookup* based on the key of the data item; (2) *find* the item with the minimum new_count value. We considered several alternatives (explained below). An empirical comparison can be found in Section 7.

The Space Saving [27] algorithm also requires efficient support of the aforementioned two operations. The authors used a Stream-Summary data structure coupled with a hash table. The hash table is used for the lookup operation and the stream-summary, a linked list which keeps all the items sorted based on their count values, helps the second operation. Therefore, we used it as the first design alternative. However, this implementation has a high space overhead as it may require up to four pointers per item. Due to the large memory overhead, for the same filter size budget, it can monitor fewer items and thus performs poorly.

On modern hardware, often for small data structures, a *linear scan* performs better than a hash-based lookup, as it avoids random access and pointer chasing [32]. A linear scan is also amenable to *vectorized execution*. Therefore, we considered two array-based implementations, namely, Vector and Heap. For both implementations, we use three arrays with (id, new_count, old_count) values. The vectorized search on id to find the location of an item is shown in Algorithm 3. This is an implementation of *linear scan* with SIMD instructions[5]. The __builtin_ctz is a hardware specific high performance function implemented in assembly provided by GCC.[6] This particular function returns the count of trailing zeros in a binary representation of a number which is used to find the location of the hit. Vector also uses a linear scan for finding the element with the minimum count. We find that the Vector performs very efficiently with a skew more than 2 in a Zipf-distribution.

The Heap incurs additional overhead for maintaining the heap. However, the benefit is that the lookup of the item of minimum count becomes inexpensive. We evaluated two min-heap implementations, namely, Strict and Relaxed. For every hit in the filter, Strict-Heap rearranges the data structure so that it maintains the

[5]Details about SSE2 intrinsics can be found in https://software.intel.com/en-us/node/514275.

[6]Details can be found in https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html.

Algorithm 3 Filtering with SSE2 SIMD intrinsics for a filter of 16 items. Implementation of linear search on the $filter\_id$ array.

```
/*load the input item*/
const __m128i s_item = _mm_set1_epi32(item);
/*point to the filter id array*/
__m128i *filter = (__m128i *)filter_id;
/*search 16 elements*/
__m128i f_comp = _mm_cmpeq_epi32(s_item, filter[0]);
__m128i s_comp = _mm_cmpeq_epi32(s_item, filter[1]);
__m128i t_comp = _mm_cmpeq_epi32(s_item, filter[2]);
__m128i r_comp = _mm_cmpeq_epi32(s_item, filter[3]);
/*check whether item is found*/
f_comp = _mm_packs_epi32 (f_comp, s_comp);
t_comp = _mm_packs_epi32 (t_comp, r_comp);
f_comp = _mm_packs_epi32 (f_comp, t_comp);
int found = _mm_movemask_epi8 (f_comp);
/*if found return the position*/
if (found) return __builtin_ctz(found);
else return -1;
```

heap. We do not find it as efficient as the Relaxed-Heap alternative which reconstructs the heap only when there is a hit on the item with the minimum count. Based on our empirical evaluation, the Relaxed-Heap provides the best performance among all design choices when the skew is low or even moderate (e.g., skew < 2 in a Zipf distribution).

6.2 Pipeline Parallelism: Filter and Sketch

The benefits of filtering are more pronounced when the filter and sketch are decoupled and run on separate cores. In such an implementation, shared memory access is replaced with a message passing interface between the two processors. As shown in Figure 1, the exchange of data items between the two data structures now happens through messages. Such design has been shown to be a good fit for modern hardware [32], as it avoids locking in the context of shared memory access. In ASketch, core C_0 contains the filter and consumes the input tuples. As soon as there is a miss in the filter (line 7 in Algorithm 1), it forwards the item to core C_1 that maintains the sketch. C_0 also forwards the minimum count whenever the minimum count changes. When the estimated frequency in C_1 is higher than the minimum count in C_0 (line 9 in Algorithm 1), C_1 sends the item back to C_0 and the filter is updated. The key benefit of this form of parallelism is that, as soon as C_0 forwards an item to C_1, C_0 can process new items. Therefore, we expect to see further improvements in both update and query throughput. This decoupling also allows the filter to have a small memory footprint and, hence, it may as well happen that the filter can even fit into the registers of the processor C_0 and may further improve the throughput.

6.3 SPMD Parallelism: ASketch as a Kernel

ASketch can be parallelized in an SPMD model where each processor will execute ASketch as a sequential counting kernel. Similar form of parallelism was considered by Thomas et al. [34] in the context of cell processor. We want to demonstrate the scalability of ASketch and compare its throughput with that of Count-Min when used as a kernel. We assume a multi-stream scenario where every core is consuming a different stream and applying the tuples to its own kernel. As frequency estimation is commutative in nature, when there is a query, each kernel replies its response which is later combined to produce the final result. As it is a point query, such a combination from multiple kernels is quite inexpensive. As pointed out in [34], there are also other forms of parallelism possible for sketches. For example, the sketch can be partitioned which would allow us to accommodate larger sketch sizes. However, it would also require all the sketch to process every incoming tuple both at update and query time. Therefore, we do not explore this form of parallelism in this paper.

7. EXPERIMENTAL RESULTS

We present experimental results with two real-world and one synthetic dataset. We evaluate ASketch *update efficiency*, *query processing throughput*, and *accuracy* of frequency estimation queries and compare it with that of Count-Min [11], Frequency-Aware Counting [34], Holistic UDAFs [10], and Space Saving [27]. We also provide a sensitivity analysis of ASketch by varying several parameters such as skew of the frequency distribution, filter type, and filter size.

7.1 Experimental Setup

Datasets. We summarize the datasets used below.

IP-Trace Network Stream. We used the IP-packet trace in an anonymous LAN for generating edge streams. The edge frequency is the total number of communications between two corresponding IP-addresses. Total stream size is 461 millions with 13 millions distinct edges. The maximum frequency of an edge is 17 978 588. This dataset is similar to a Zipf distribution of skew 0.9.

Kosarak Click Stream. It is an anonymized click-stream data of a hungarian on-line news portal.[7] The data set is relatively small containing 8M clicks with 40 270 distinct items. The maximum frequency of an item is 601 374. This dataset has a skew similar to a Zipf distribution of 1.0.

Synthetic Dataset. We generate a synthetic stream dataset from the skewed Zipf distribution with a stream size of 32M and the number of distinct items of 8M. We vary the skew from 0 to 3. For ease of discussion, we divide the skew range into three parts [26], and we refer to them as follows: low-skew range (0 to 1), mid-skew range or *real-world* skew range (1 to 2), and high-skew range (2 to 3).

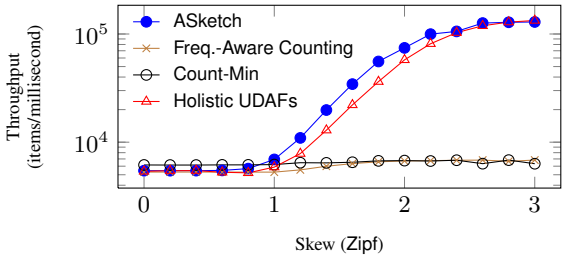
Query and Parameters Setting. We evaluate *frequency estimation queries*: given a data item, estimate its frequency. The queries are obtained by sampling the data items based on their frequencies, that is, the high-frequency items are queried more than the low-frequency items in a skewed stream. We also perform an evaluation of the *top-k frequent items query*: determine the top-k frequent items, where k is given by the filter size of our ASketch synopsis.

We vary ASketch size from 16KB to 128KB [9, 11]. Our main focus is to operate from either the L1 or the L2 cache. Therefore, most of the experiments are done with an ASketch size of 128KB. The filter size ($|F|$) is varied from 0.1KB (8 items) to 12KB (1024 items). To keep the total synopsis size constant, we also varied the underlying sketch size. However, for a specific size of the sketch, we find that the accuracy is more sensitive towards h (i.e., range of each hash function) than towards w (i.e., the number of hash functions), especially with $w \leq 8$. Therefore, in most of our experiments, we have set $w = 8$ for all the sketch data structures, and varied h to compensate for the space of our filter. Finally, we compare the performance of four different filter implementations of ASketch as discussed in Section 6.1.

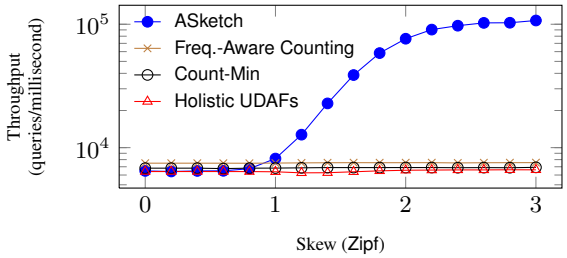
Evaluation Metrics. We measure efficiency using two metrics.

- *Stream processing throughput*, expressed in items per millisecond, measures the average number of incoming data tuples that are processed per millisecond.

[7] Obtained from http://fimi.ua.ac.be/data/.



(a) Stream processing throughput with varying skew (Zipf)



(b) Query processing throughput with varying skew (Zipf)

Figure 5: Experiments on the *Synthetic* dataset; ASketch with Relaxed-Heap filter; filter size $|F|$ =32 items (∼0.4KB); total synopsis size 128KB.

- *Query processing throughput*, expressed in queries per millisecond, counts the average number of frequency estimation queries that can be answered per millisecond.

We evaluate the accuracy of frequency estimation queries with the two following metrics [11].

- *Observed error* is measured as the difference between the estimated frequency and the true frequency, accumulated over all the items queried. We express it as a ratio over the aggregated true frequency of all the data items queried, i.e.,

$$\text{Observed Error} = \frac{\sum_{i \in \text{Query}} |\text{estimated freq.}_i - \text{true freq.}_i|}{\sum_{i \in \text{Query}} \text{true freq.}_i}$$

- *Relative error* for one frequency estimation query is defined as the difference between the estimated frequency and the true frequency, divided by the true frequency of that data item. In our results, we report the average relative error over a pre-specified number of data items queried. Formally,

$$\text{Avg. Rel. Error} = \frac{1}{|\text{Query}|} \sum_{i \in \text{Query}} \frac{|\text{estimated freq.}_i - \text{true freq.}_i|}{\text{true freq.}_i}$$

  One may note that the average relative error is biased towards the low-frequency items, due to their lower true frequency values in the denominators of the above equation.

Finally, we analyze the effectiveness of our top-k frequent items query by using *Precision-at-k*:

$$\text{Precision-at-k} = \frac{\text{true top-}k\text{ frequent items in the top-}k\text{ reported items}}{k}$$

Comparing Methods. We compare the accuracy and efficiency of ASketch with three state-of-the-art stream processing approaches[8], by assigning the same total space to all these methods.

- Count-Min (CMS) [11] applies w pairwise independent hash functions on the key k of an incoming tuple (k, u). In most of the experiments, we fixed $w = 8$ and varied h based on the total size of the synopsis.
- Frequency-Aware Counting (FCM) [34] improves the accuracy of Count-Min by using only a subset of the w hash functions for hashing each data item. Given a data item, the algorithm first applies two separate hash functions to compute an *offset* and a *gap*, which determines the subset of Count-Min hash functions to be used for hashing the data item. In addition, FCM varies the number of hash functions used for a high-frequency and a low-frequency item (e.g., $\frac{w}{2}$ hash functions for a high-frequency item and $\frac{4w}{5}$ hash functions for a low-frequency item as considered in [34]). Finally, it also uses a MG counter data structure [28] to identify the high-frequency items. For fairness, in our experiments, we have fixed the MG counter size in such a way that it stores the same number of high-frequency items as that in our filter. For lookup in the MG counter, we use the same hardware-conscious SIMD-enabled lookup code that we use for the filter lookup in ASketch.
- Holistic UDAFs (H-UDAF) [10] performs run-length aggregation in a low-level table before flushing the items into a sketch. We use a low-level table that is capable of storing the same number of data items as that in our filter, whereas the underlying sketch is implemented as Count-Min sketch. For the lookup in the low-level table, we use the same code that we use for the filter lookup.

We used publicly available source code of CMS from [11]. We implemented the codes of FCM and H-UDAF, since they are not publicly available.

Machine description. We performed all the experiments on a 2.27GHz Intel Xeon L5520 machine with 8 cores. The CPU core has 32KB L1 cache, 256KB L2 cache, and 8MB L3 cache. The operating system was Linux kernel 3.8.5. All the algorithms are implemented in C and compiled using gcc 4.7.2 with -O3 optimization. Each experimental result is averaged over 100 runs.

7.2 ASketch Performance Comparison

7.2.1 *Frequency Estimation queries*

We first compare ASketch with Count-Min, Frequency-Aware Counting, and Holistic UDAFs. We consider the Relaxed-Heap filter implementation for ASketch since it performs well over all skew ranges, as illustrated later in our sensitivity analysis experiments (Section 7.5)

■ **Stream Processing Throughput.** The result of the stream processing throughput comparison for all the approaches is shown in Figure 5(a). Count-Min performs almost the same over all skew ranges, since the algorithm applies the same number of hash functions irrespective of the skew. Frequency-Aware Counting throughput is lower than that of the Count-Min in low-skew range; which is caused by the overhead of MG counter and the two additional hash functions. However, for high-skew, hashing frequent items to a smaller number of hash cells compensates this overhead. Therefore, Frequency-Aware Counting throughput catches up with that of Count-Min when the skew increases. Benefits

[8] While Space Saving [27] can be adapted to process frequency estimation queries, usually it does not perform well in frequency estimation. We show comparision with Space Saving separately in Figure 11.

Count-Min Size	Max. # Misclassifications
16KB	27
24KB	5
32KB	8

Table 3: Misclassification statistics of Count-Min on the *Synthetic* data (Zipf skew 1.5). We report the maximum number of low-frequency items misleadingly appearing as high-frequency items over 100 runs. In our experiments with ASketch, such misclassifications did not occur.

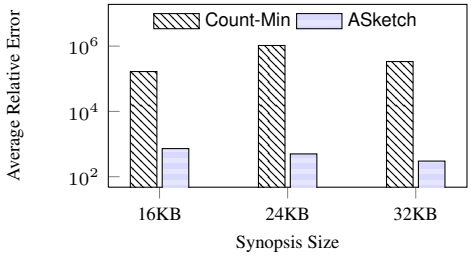


Figure 6: Average relative error over non-frequent items that appeared highly-frequent in Count-Min: *Synthetic* data with Zipf skew 1.5.

of early-aggregation is clearly visible in the throughput graph of Holistic UDAFs. In low and moderate skew ranges, the low-level aggregate table in Holistic UDAFs works as an overhead, and early-aggregation does not provide any benefit. Hence, its performance drops below that of Count-Min. As the skew increases, the throughput of the algorithm also increases. At a high-skew > 2.5, a few flushings are required and thus the algorithm achieves high throughput. ASketch also behaves similarly. Even at low skew, the added overhead of filter does not deteriorate the performance significantly, compared to Count-Min. ASketch starts superseding Count-Min as soon as the skew reaches 0.8. With further increase in skew, the throughput of ASketch increases by almost an order of magnitude compared to Count-Min. In addition, we find that ASketch outperforms Holistic UDAFs in the *real-world* skew range (1∼2), due to the overhead of flushing from the low-level aggregate table for Holistic UDAFs.

■ **Query Processing Throughput.** The query processing throughput is shown in Figure 5(b). The query items are selected uniformly from the incoming stream, that is, in a skewed data stream, the high-frequency items are queried more often compared to their low-frequency counterparts. Our results attest to the benefits of ASketch. Our filter is capable of answering most of the queries. But for Holistic UDAFs, the low-level aggregator alone is not capable of answering any query. It always requires to retrieve the frequency estimate from the underlying sketch. One may also note that Frequency-Aware Counting performs slightly better than Count-Min, since the former uses a smaller number of hash functions for answering frequency estimation queries [34]. Figure 5(b) shows that ASketch improves the query processing throughput by an order of magnitude over the existing approaches when the skew is more than 1.

■ **Avoiding Large Estimation Error.** One negative aspect of Count-Min is that if a low-frequency item always collides with the high-frequency items by applying each one of the hash functions, it then incurs a significant frequency estimation error for that low-frequency item. In such cases, the low-frequency item can be misclassified as a high-frequency item, which is often detrimental in many applications [2, 19, 30]. Although increasing the size of the sketch reduces the probability of such misclassification, we find that for relatively small-sized sketches inside the L1 cache, this situation occurs consistently when a significant number of non-frequent items get misclassified as a high-frequency item.

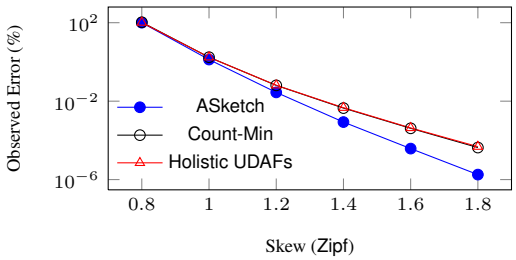


Figure 7: Observed error among ASketch, Count-Min, and Holistic UDAFs: synopsis size is 128KB.

Skew	× Improvement (Synopsis 64KB)	× Improvement (Synopsis 128KB)
0.8	1.0	1.0
1.0	1.3	1.3
1.2	2.3	2.2
1.4	5.3	5.2
1.6	11.0	10.8
1.8	28.0	23.9

Table 4: Times improvement in observed error for ASketch over Count-Min: total synopsis sizes are 64KB and 128KB.

We present the maximum number of such misclassifications over 100 runs in Table 3. The corresponding average relative error introduced by those misclassified items are shown in Figure 6. Similar situations are very unlikely to occur in ASketch as it separates the high-frequency items from the low-frequency items. This is clearly visible in Figure 6. We can see that the average relative error of Count-Min can be three order of magnitude higher than that of ASketch.

■ **Error Comparison with Count-Min and Holistic UDAFs.** We recall that any improvement by the ASketch is orthogonal to the underlying sketch. Therefore, we first compare our ASketch technique with Count-Min and Holistic UDAFs, since we have used Count-Min as the underlying sketch for both ASketch and Holistic UDAFs. The error comparison results using the percentage observed error are presented in Figure 7. We focus mainly on the skew range of 0.8 to 1.8 due to two reasons. (1) This is almost same as the *real-world* skew range. (2) For a skew higher than 1.8, the observed error by ASketch is often zero; and for a skew below 0.8, the observed errors by these three methods are very much comparable. One may observe that as we increase the skew, ASketch starts performing better than both Count-Min and Holistic UDAFs. Holistic UDAFs relies on the underlying sketch for answering the queries, therefore the performance is almost the same as that of Count-Min. For example, with skew 1.4, the observed error by both Count-Min and Holistic UDAFs is $(4 \times 10^{-3})\%$, while the observed error by ASketch is $(9 \times 10^{-4})\%$. Considering the fact that the true frequencies of the most frequent items are in the order of $10^6 \sim 10^7$, which are aggregated in the denominator to compute the observed error percentage, ASketch significantly improves the frequency estimation accuracy for the high-frequency items. In Table 4, we further show the times of improvement in observed error that ASketch achieves over Count-Min for two different synopsis sizes, i.e. 64KB and 128KB. ASketch *improves the observed error about* 25 *times compared to* Count-Min *and* Holistic UDAFs*, while using the same amount of storage space*.

■ **Error Comparison with Frequency-Aware Counting (FCM).** To demonstrate the *generality* of ASketch, we implement it on top of a sketch that uses the FCM principle. While running the experiments with FCM, we found that FCM performs much better in terms of the accuracy of frequency estimation queries, in comparison to the Count-Min. The main error improvement of FCM

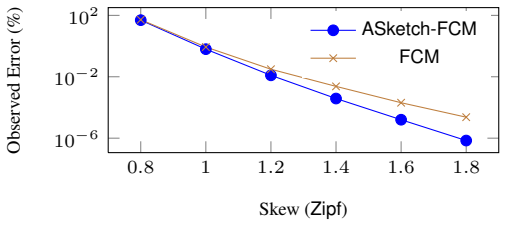


Figure 8: Observed error comparison between ASketch and FCM: Synopsis size is 128KB, ASketch-FCM is the ASketch that uses an underlying sketch based on FCM.

Skew (Zipf)	*Precision-at-k*
0.4	0.74
0.6	0.96
0.8	0.99
≥1.0	1.0

Table 5: Accuracy comparison of ASketch for top-k frequent item queries. Here, ASketch size is 128 KB, filter size is $|F| = 0.4$KB.

comes from the fact that it uses two additional hash functions to compute an *offset* and a *gap*, and picks a $w_1 < w$ number of sketch hash functions to apply for a particular data item. This reduces the number of collisions and improves the overall accuracy of the sketch. However, this approach is also compatible with ASketch. Therefore, we replace the underlying Count-Min implementation in ASketch, and incorporated an FCM-like sketch. The result of this comparison is shown in Figure 8. We find that we achieve similar levels of improvement as we have achieved before with respect to Count-Min. For example, with skew 1.6, our observed error is 13 times smaller than that of FCM. The result does emphasize the generality of ASketch as it can take advantage of any improvement in the underlying sketch.

7.2.2 *Top-k Frequent Items Queries*

For strict distributions [29], ASketch directly supports top-k queries where k is defined by the size of the filter. The result of *precision-at-k* for ASketch is shown in Table 5. As shown in the table, ASketch achieves high accuracy. For a skew (Zipf) of 1.0 and higher, the algorithm achieves a *precision-at-k* value of 1.0. The algorithm also exhibits high precision even at relatively small skew.

7.2.3 *Number of Exchanges for ASketch.*

In Figure 9, we present the statistics about the number of exchanges that happens between the filter and the sketch data structures in ASketch. As expected, the number of exchanges reduces as the skew increases. This indeed helps in improving the ASketch throughput with increasing skew in the input stream, as shown earlier in Figures 5(a) and 14. However, even with Uniform distribution (i.e., $zipf = 0$), we need only about 40K exchanges for a stream of size 32M. Our experimental evaluation indicates that the exchange of data items is not an issue.

7.3 Experiments on Real-World Datasets

■ **Stream Processing Throughput Comparison.** The stream processing throughput on the *IP-trace* data is shown in Figure 10(a). Here, ASketch-FCM is the ASketch implementation where the underlying sketch is implemented by following the FCM principle. The throughput of ASketch is 5% higher than that of the Count-Min. Since *IP-trace* data is in the low-skew range, it is expected that ASketch will not outperform Count-Min by a significant margin. However, ASketch-FCM achieves almost 30% more throughput than that of Count-Min. It also outperforms Holistic UDAFs by almost 40%, since with low skew, the low-level aggregate table in Holistic UDAFs works as a performance bottleneck due to the excessive flushing from the table. In comparison to FCM, ASketch-FCM also achieves 40% more throughput. This is because of two reasons. (1) As explained earlier in Section 7.2.1, we consider a modified version of the FCM where we do not use the MG counter — the MG counter incurs a significant performance overhead for the original FCM technique [34]. (2) We accumulate a large number of frequency counts in the filter that are not hashed into the underlying sketch. In case of FCM, all the frequency counts are hashed into the sketch data structure.

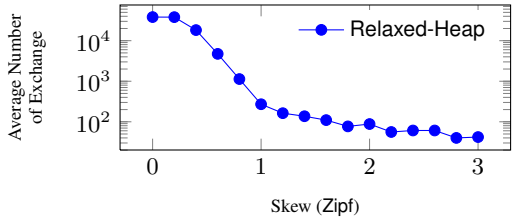


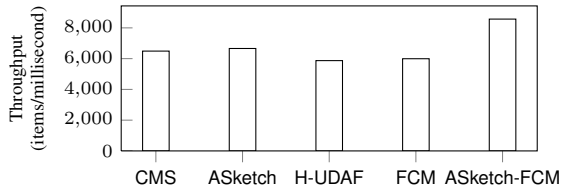
Figure 9: Average number of exchange with varying skew for ASketch: *Synthetic* dataset; Relaxed-Heap filter of size $|F|$ =32 items (∼0.4KB); ASketch size is 128KB.

The stream processing throughput for the Kosarak click stream is shown in Figure 10(c). The Kosarak click stream has slightly higher skew than the IP-trace stream. As a result, ASketch achieves almost 20% more throughput than Count-Min. The performance of Count-Min is skew independent, therefore, the throughput numbers are similar to that of IP-trace stream. Skew also helps Holistic UDAFs but ASketch achieves almost 10% better throughput than that of Holistic UDAFs. Similar to the IP-trace stream, FCM throughput is slightly lower than that of Count-Min due to the overhead of the MG counter. Finally, ASketch-FCM outperforms all other approaches by accumulating larger number of frequency counts in the filter. It achieves almost 70% better throughput than that of FCM.

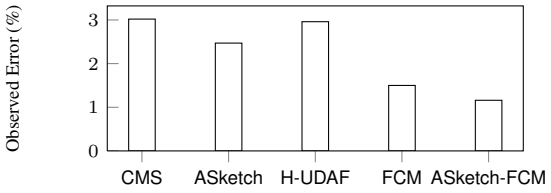
■ **Frequency Estimation Error Comparison.** The result of frequency estimation error comparison with the *IP-trace* stream is shown in Figure 10(b). We show both implementations of ASketch — one with Count-Min as the underlying sketch; another one with FCM as the underlying sketch (abbreviated as ASketch-FCM). ASketch achieves almost 20% better observed error than both Count-Min and Holistic UDAFs. Holistic UDAFs achieves almost the same error as that of Count-Min, since it uses the underlying sketch to answer all queries. FCM achieves better accuracy than that of Count-Min as well as from ASketch. However, ASketch-FCM improves the accuracy of FCM by more than 22%.

The frequency estimation results over the Kosarak click stream are shown in Figure 10(d). ASketch outperforms both Count-Min and Holistic UDAFs by achieving almost 32% less observed error. Holistic UDAFs achieves almost the same error as that of Count-Min because it always uses the underlying sketch for answering queries. As expected, FCM achieves better accuracy than that of both ASketch and Count-Min. However, ASketch-FCM outperforms FCM by achieving almost 48% less observed error. It illustrates that the improvement of ASketch is orthogonal to the underlying sketching technique.

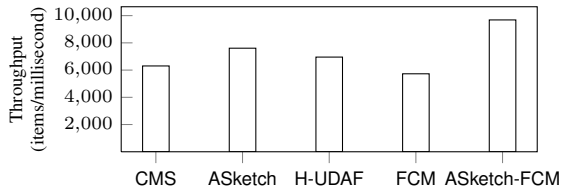
■ **Space Saving for Frequency Estimation.** The Space Saving [27] can be modified to answer frequency estimation queries as follows. If an item is currently monitored by the algorithm, its count can directly be used as the corresponding estimate. Oth-



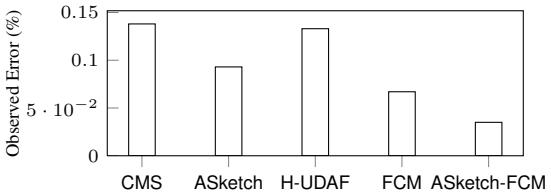
(a) Stream processing throughput on the *IP-trace* stream



(b) Observed error comparison on the *IP-trace* stream



(c) Stream processing throughput on the *Kosarak* click stream



(d) Observed error comparison on the *Kosarak* click stream

Figure 10: Experiments on real-world datasets: ASketch with Relaxed-Heap filter implementation; filter size $|F|$ =32 items (∼0.4KB); ASketch size is 128KB. ASketch-FCM is the ASketch that uses an underlying sketch based on FCM.

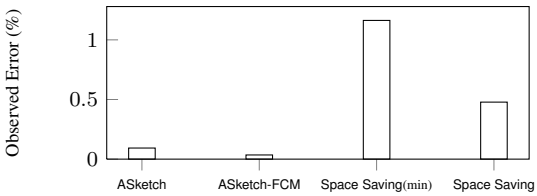


Figure 11: Observed error comparison of ASketch with Space Saving on the Kosarak click stream: ASketch with Relaxed-Heap filter; filter size $|F|$ =32 items (∼0.4KB); ASketch size is 128KB. ASketch-FCM is ASketch with FCM based underlying sketch.

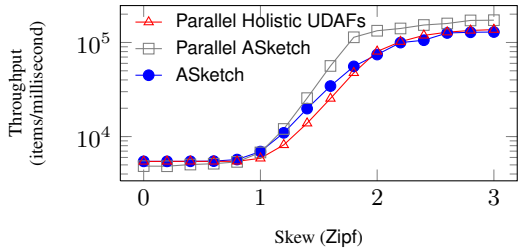


Figure 12: Stream processing throughput comparison of Parallel ASketch and Parallel Holistic UDAFs with varying skew (Zipf) on the *Synthetic* dataset; ASketch with Relaxed-Heap filter is used as a baseline; filter size $|F|$ =32 items (∼0.4KB); total synopsis size is 128KB.

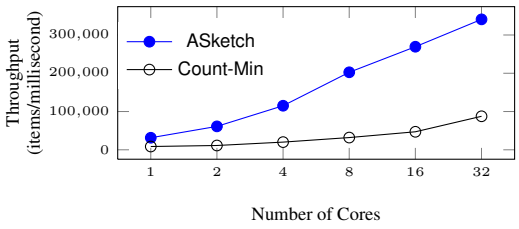


Figure 13: Throughput comparison between ASketch and Count-Min when they are used as counting kernel with varying number of cores: each synopsis size is 128KB. Here, the stream size is 1 billion tuples and the number of distinct items is 100 million, Zipf skew 1.5.

Filter Type	Observed Error (%)
Stream Summary	0.0005
Vector	0.0002
Relaxed-Heap	0.0002
Strict-Heap	0.0002

Table 6: Accuracy comparison with various filter implementations in ASketch: Vector, Stream-Summary, Strict-Heap, and Relaxed-Heap. ASketch size is 128KB, filter size $|F|$ = 0.4KB, skew (Zipf)=1.5.

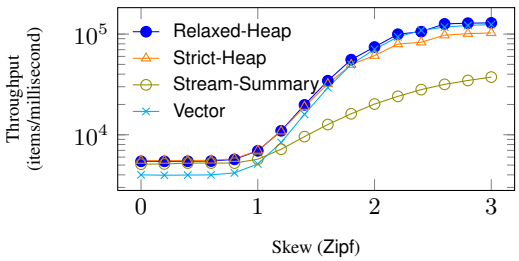
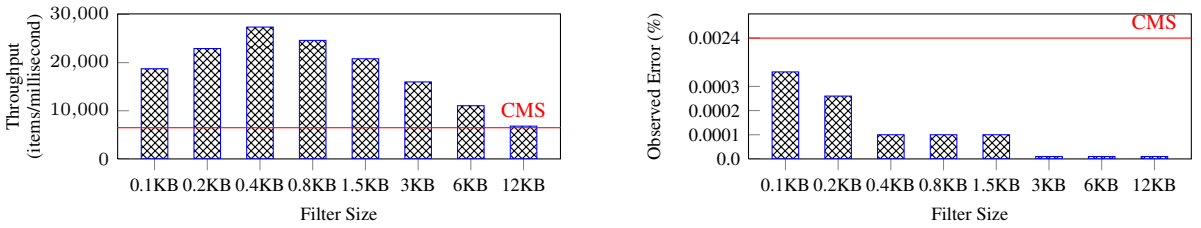


Figure 14: Stream processing throughput comparison with various filter implementations in ASketch: Vector, Stream-Summary, Strict-Heap, and Relaxed-Heap. ASketch size is 128 KB, filter size $|F|$ = 0.4KB.

erwise, there are two approaches that can be followed. The authors in [27] suggested not to underestimate the value of the frequency and therefore, we can use the frequency of the item with the minimum count as an estimate. However, the authors in [9] suggested to use 0 as the estimate of frequency for such items. In both cases, Space Saving performs poorly for frequency estimation in comparison to a same-sized sketch. To demonstrate this, we have used Space Saving to process frequency estimation queries on the Kosarak dataset, and compare this result with ASketch (presented earlier in Figure 10(d)). The result of this comparison is shown in Figure 11. Here, Space Saving(min) is the one where we use the minimum value as the estimate, and Space Saving is the one where we use 0 as the estimate. Clearly when 0 is used as estimate, Space Saving performs better. However, both implementations of ASketch achieve a much lower error in comparison.



(a) Stream processing throughput

(b) Observed error comparison

Figure 15: Sensitivity analysis, w.r.t., filter sizes in ASketch: ASketch size is 128KB; *Synthetic* data; Zipf skew 1.5; Relaxed-Heap filter. Here, CMS refers to the throughput and error of Count-Min alone, shown as a point of reference for other measurements.

7.4 Experiments on Parallel ASketch

■ **Throughput Benefit of Pipeline Parallelism.** ASketch with pipeline parallelism (we refer to as Parallel ASketch), is the one where one core runs the filter and another core runs Count-Min algorithm. The throughput benefit of this algorithm is visible in Figure 12, especially in the skew range of 1.2 and higher. The core running the filter can start processing the next item as soon as it has forwarded the item to the core running the sketch. In fact, Parallel ASketch achieves almost twice the throughput of ASketch at a skew of 1.8. However, for a skew higher than 2.4, the throughput advantages diminish as a few items overflow. Parallel Hollistic UDAFs in Figure 12 is the implementation that is parallelized in a pipeline manner. Pipeline parallelism also benefits Parallel Hollistic UDAFs as the first core, after flushing the low-level aggregator table, can immediately start processing next items from the input stream. However, Parallel ASketch achieves almost twice the throughput of parallel Parallel Hollistic UDAFs at a skew of 1.8.

■ **ASketch as a Counting Kernel.** We run this experiment on an Intel Sandy-Bridge machine with 32 cores arranged in a four-socket configuration. The clock frequency is 2.40 GHz; therefore, single thread throughput is slightly higher in Figure 13 than those presented earlier in Figure 5(a). Although we do not carefully optimize the implementation for NUMA architecture, the linear scalability with increasing number of cores is clearly visible. The throughput advantage of ASketch over Count-Min is also obvious from the figure. For example, a 32 core configuration for ASketch achieves almost 4 times better throughput.

7.5 ASketch Sensitivity Analysis

Here, we analyze the performance of ASketch by varying two important parameters: the filter implementation and the filter size.

■ **Impact of the Filter Implementation.** We compare the tradeoffs of four different filter implementations: Vector, Stream-Summary with hash table, Strict-Heap, and Relaxed-Heap. Here, we fix the total ASketch size to 128KB and fix the filter size as 0.4KB. We present the impact of varying the filter implementations on ASketch accuracy for frequency estimation queries in Table 6. Since Stream-Summary has more space overhead per item, it can accommodate only 4 items with a 0.4KB filter size; the other three filter implementations have exactly the same space overhead per item, and each of them can store 32 items within 0.4KB filter. Therefore, the Vector and both the heap-based implementations have the same accuracy; while the Stream Summary has lower accuracy.

The stream processing throughput of these four implementations are given in Figure 14. Vector performs better at a higher skew range (>2.0), since there is no overhead of maintaining the data structure, and almost all the hits go to the filter. However, it does not perform well for a skew <2.0, due to the lookup cost of the smallest count present in the filter (see line 9, Algorithm 1), which requires a linear scan on the filter.

Stream-Summary, in general, does not perform very well over all skew ranges due to its expensive data structure maintenance costs, e.g., hash evaluations, pointer chasing, collision resolutions, and also the maintenance of a doubly linked list. Nevertheless, because of the constant time access to the item with the minimum frequency count, Stream-Summary achieves better throughput at a lower skew than Vector.

Heap implementations perform better with skew <2.0, which also includes the *real-world* skew range. Particularly, Relaxed-Heap performs better than Strict-Heap, since less maintenance cost is required in the former. Both heap implementations perform better than Vector in low and medium skew ranges due to a fast access to the item with the minimum frequency count at the root of the heap. This is why we used Relaxed-Heap based ASketch while comparing with existing approaches.

■ **Impact of the Filter Size.** We present the impact of filter size over ASketch throughput and accuracy in Figures 15(a) and 15(b), respectively. Here, we used Relaxed-Heap. As explained earlier in Section 4, increasing the filter size beyond a threshold size (e.g., 0.4KB in our current setting) increases the filter processing time without too much gain in the filtering rate. Thus, beyond a threshold filter size, the overall ASketch throughput decreases. On the other hand, with increasing filter size, while we have better accuracy for more high-frequency items, we also introduce more error over the low-frequency items due to the reduced size of the underlying sketch data structure. Therefore, we find that the observed error stops improving after a threshold filter size, e.g., 3KB in our current setting (Figure 15(b)).

8. CONCLUSIONS

In this paper, we present ASketch — a filtering technique complementing sketches. It improves the accuracy of sketches by increasing the frequency estimation accuracy for the most frequent items and by reducing the possible misclassification of the low-frequency items. It also improves the overall throughput. ASketch outperforms the existing stream processing algorithms with an order of magnitude higher throughput and about 25 times higher accuracy, while using exactly the same amount of space. We further incorporate parallelism in ASketch to support even higher streaming rate, e.g., pipeline parallelism improves the ASketch throughput almost twice in the *real-world* skew range. ASketch also exhibits linear scalability in the case SPMD parallelism. In future work, it would be interesting to employ ASketch in a wide variety of machine learning and data mining applications where sketches are currently used.

9. REFERENCES

[1] C. Aggarwal and K. Subbian. Event Detection in Social Streams. In *SDM*, 2012.

[2] C. Aggarwal and P. S. Yu. On Classification of High-Cardinality Data Streams. In *SDM*, 2010.

[3] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *STOC*, 1996.

[4] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss. Space-optimal Heavy Hitters with Strong Error Bounds. *ACM TODS*, 35(4):26, 2010.

[5] D. Bitton and D. J. DeWitt. Duplicate Record Elimination in Large Data Files. *ACM TODS*, 8(2):255–265, 1983.

[6] B. H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Comm. of ACM*, 13:422–426, 1970.

[7] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *ICALP*, 2002.

[8] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[9] G. Cormode and M. Hadjieleftheriou. Finding Frequent Items in Data Streams. In *VLDB*, 2008.

[10] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic UDAFs at Streaming Speeds. In *SIGMOD*, 2004.

[11] G. Cormode and S. Muthukrishnan. An Improved Data-Stream Summary: The Count-min Sketch and its Applications. *J. of Algorithms*, 55(1), 2005.

[12] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing Complex Aggregate Queries over Data Streams. In *SIGMOD*, 2002.

[13] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.

[14] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing data-stream join aggregates using skimmed sketches. In *Advances in Database Technology-EDBT 2004*, pages 569–586. Springer, 2004.

[15] S. Ganguly, P. B. Gibbons, Y. Matias, and A. Silberschatz. Bifocal Sampling for Skew-Resistant Join Size Estimation. In *SIGMOD*, pages 271–281, 1996.

[16] M. Garofalakis and P. B. Gibbons. Wavelet Synopses with Error Guarantees. In *SIGMOD*, 2002.

[17] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In *VLDB*, 2001.

[18] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to Summarize the Universe: Dynamic Maintenance of Quantiles. In *VLDB*, 2002.

[19] A. Goyal, H. D. III, and G. Cormode. Sketch Algorithms for Estimating Point Queries in NLP. In *EMNLP-CoNLL*, 2012.

[20] S. Guha, N. Koudas, and K. Shim. Data-streams and Histograms. In *STOC*, 2001.

[21] C. Koch, S. Scherzinger, and M. Schmidt. XML Prefiltering as a String Matching Problem. In *ICDE*, pages 626–635, 2008.

[22] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based Change Detection: Methods, Evaluation, and Applications. In *IMC*, 2003.

[23] P. Larson. Grouping and Duplicate Elimination: Benefits of Early Aggregation, 1997. Technical Report, MSR-TR-97-36.

[24] T. Liu, Y. Sun, A. X. Liu, L. Guo, and B. Fang. A Prefiltering Approach to Regular Expression Matching for Network Security Systems. In *Applied Cryptography and Network Security*, pages 363–380, 2012.

[25] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter braids: a novel counter architecture for per-flow measurement. In *SIGMETRICS*, 2008.

[26] N. Manerikar and T. Palpanas. Frequent Items in Streaming Data: An Experimental Evaluation of the State-of-the-art. *Data Knowl. Eng.*, 68(4):415–430, 2009.

[27] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *ICDT*, 2005.

[28] J. Misra and D. Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

[29] S. Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

[30] A. Pietracaprina, M. Riondato, E. Upfal, and F. Vandin. Mining top-K Frequent Itemsets Through Progressive Sampling. *Data Min. Knowl. Discov.*, 21(2):310–326, 2010.

[31] O. Rottenstreich, Y. Kanizo, and I. Keslassy. The Variable-Increment Counting Bloom Filter. *IEEE/ACM Trans. Netw.*, 22(4):1092–1105, 2014.

[32] P. Roy, J. Teubner, and G. Alonso. Efficient Frequent Item Counting in Multi-Core Hardware. In *KDD*, 2012.

[33] F. Rusu and A. Dobra. Sketches for Size of Join Estimation. *ACM TODS*, 33(3):15, 2008.

[34] D. Thomas, R. Bordawekar, C. Aggarwal, and P. S. Yu. On Efficient Query Processing of Stream Counts on the Cell Processor. In *ICDE*, 2009.

[35] W. Yan and Y. X. B. Malin. Scalable Load Balancing for MapReduce-based Record Linkage. In *IPCCC*, 2013.

[36] J. Yu, Z. Chong, H. Lu, and A. Zhou. False Positive or False Negative: Mining Frequent Itemsets from High Speed Transactional Data Streams. In *VLDB*, 2004.

[37] P. Zhao, C. C. Aggarwal, and M. Wang. gSketch: On Query Estimation in Graph Streams. In *VLDB*, 2012.

APPENDIX

A. DELETION OF ITEMS

Removal of an item from sketches is often necessary in many applications such as the conclusion of a packet flow, or the return of a previously bought item. Traditional sketches allow us to model removal of items as an update with negative counts [9]. Particularly, in the Count-Min, a negative-count-update can be performed in the same way as a positive-count-update, as long as the overall count of an item never goes negative. In the following, we shall illustrate how ASketch supports negative-count-updates.

Let us assume that the negative-count-update with the current tuple is k, that is, the item's overall frequency count needs to be reduced by an amount of k. If the item is not found in the filter, we directly apply the Count-Min technique to reduce its count by k from the sketch. On the other hand, if the item is found in the filter and its (new_count-old_count) is greater than or equal to k, then we only reduce its new_count by k. However, complexity arises if the item is found in the filter, but its (new_count-old_count) is smaller than k. Let us denote by (new_count-old_count)$= \triangle$ for that item. In such cases, we subtract k from its new_count, $(k - \triangle)$ from its old_count, and finally we also reduce its count by $(k - \triangle)$ in the underlying sketch. It is important to note that after processing a negative-count-update, we do not initiate any exchange of items between the filter and the sketch. The exchange is performed only in the successive positive-count-updates, if necessary.

B. ADDITIONAL EXPERIMENTS

B.1 Accuracy of Low-Frequency Items for ASketch

One of the key design issues of ASketch data structure is the size of the *Filter*. We have evaluated the impact of the filter size in Section 4, analytically and in Section 7.5, experimentally. We have demonstrated that increasing the filter size beyond certain threshold will decrease the stream processing throughput without any gain in accuracy of high frequency items. Therefore, we have used a small sized filter. To accommodate the filter data structure in ASketch, the same amount of space is reduced from the underlying *sketch*. As pointed out earlier, this reduction of space is very small to have any significant effect over accuracy of the low-frequency items. To demonstrate this, we have plotted the average relative error over all the low-frequency items in Figure 16. Even though the average relative error is more biased towards the low-frequency items, we do not see any differences in between Count-Min and ASketch.

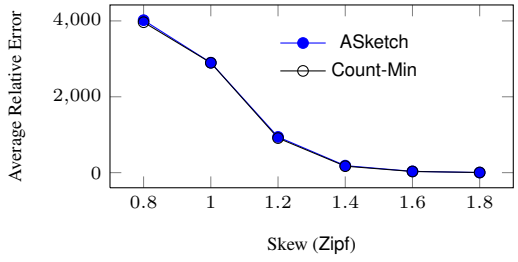


Figure 16: Average relative error comparison over all low-frequency items between ASketch and Count-Min. Stream size is 32M and alphabet size is 8M; synopsis size is 128KB.

To further emphasize that we do not introduce a high error in a few low-frequency items, we find the top-10 items with the highest accumulative error (i.e., difference between the actual count and estimated count). We show the average accumulative error over these top-10 items for both Count-Min and ASketch in Table 7. As shown in the table, there is almost no difference in the accumulative error of the top-10 error items.

Skew	Count-Min	ASketch
0.8	8013	8088
1.0	6761	6750
1.2	3541	3567
1.4	1399	1280
1.6	499	441
1.8	156	122

Table 7: Average accumulative error for top-10 error items between ASketch and Count-Min. Stream size is 32M and alphabet size is 8M; synopsis size is 128KB.

B.2 Comparison of Predicted vs. Achieved Filter Selectivity

Analytically, we have predicted the $\text{filter}_{\text{selectivity}}$ on a Zipf distribution in Section 4. To demonstrate that we achieve high $\text{filter}_{\text{selectivity}}$ with ASketch, we ran the experiment on Zipf distribution with 32M input items and tracked the number of items that are processed by the underlying sketch data structure (N_2). We compute the $\text{filter}_{\text{selectivity}}$ as the ratio of N_2 over the total number of input items (N), and plotted it in Figure 17. There are slight differences among the values which are almost not visible in the figure (e.g., for skew of 1.0, predicted is 0.75, and achieved is 0.76). For highly skewed distributions, the high-frequency items, after few updates in the sketch, gets exchanged into the filter, and then regular updates to the filter keeps them in the filter.

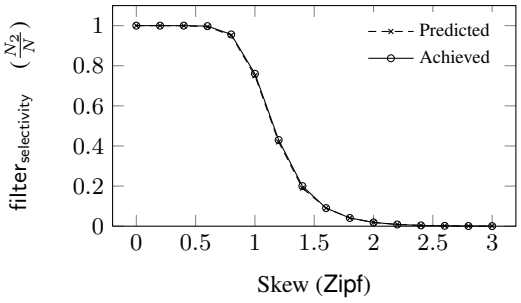


Figure 17: Filter selectivity: *Predicted* vs. *Achieved* $\text{filter}_{\text{selectivity}}$; data stream with Zipf distribution, stream size is 32M, the number of distinct data items is 8M. Here, we fixed the filter size $|F| = 32$ high frequency items.

C. ADDITIONAL ANALYSIS

C.1 Proof of Theorem 1

When the range of each hash function is h, on an average N/h items are hashed into each cell. When we allocate a size s_f to the filter, the range of each hash function reduces by s_f/w. Therefore, Ns_f/hw items get accumulated in the remaining $h - s_f/w$ cells. In other words, the expected value of the increase in the count for low-frequency items is given by $\Delta E = \frac{Ns_f}{wh(h-s_f/w)}$. Hence, by applying Markov inequality, we get:

$$P(\Delta E \geq (\frac{es_f}{wh(h - \frac{s_f}{w})})N) \leq 1/e \quad (2)$$

Since we use w pairwise independent hash functions, the overall probability that the above happens is smaller than e^{-w}. Hence, the theorem follows.

C.2 Exchange Policy

As depicted in Figure 9, if there exists skew in input data, the number of exchanges drops significantly (less than 100 exchanges for a skew of 3). In order to construct an average case scenario with uniform distribution, let us assume that no hit happens in the filter. Given that the filter monitors $|F|$ items, and the range of hash function in the sketch is h, one may require $|F|$ exchanges on an average for every h items. The intuition behind this is that once $|F|$ items are exchanged, the count of each cell in the hash function needs to be increased by 1 for the next set of $|F|$ exchanges. Therefore, with a stream size of N, the average number of exchange would be $N|F|/h$. For example, in a setup where $|F| = 32$, $h = 4084$, and $w = 1$, with 32M stream size in uniform distribution, one may require up to ∼250K exchanges. However, in practice, we consider multiple hash functions in the sketch (i.e, $w > 1$), and minimum of those w values are used to estimate the frequency of an item. Also, if there are hits in the filter, the number of exchange would further decrease. Thus, the observed number of exchanges shown in Figure 9 is much lower than ∼250K, more specifically ∼40K for the uniform distribution.

Next, we construct a scenario with uniform distribution such that minimum number of exchanges will occur. Let us assume, $|F| = 32$, $h = 4084$, $w = 1$, stream size: $N = 32$M, and domain size: $M = 8$M. Therefore, each item occurs $\frac{N}{M} = 4$ times. Assuming these N counts get evenly distributed in each cell of the underlying sketch, the count of each cell will be $\frac{N}{h} = 7832$. The counts of each cell will come from $\frac{M}{h} = 1958$ different items. Now, we consider a scenario where these 1958 different items constitute the first 7832 counts in the stream, and they all get mapped to one particular cell in the sketch. Assuming no hits in the filter, there will be about 7832 exchanges for these items, and each item in the filter will have a count close to 7832. Now, assuming that the rest $(N - \frac{N}{h})$ counts get evenly distributed in rest of the $h-1 = 4083$ cells, there will be only few more additional exchanges due to the items with counts less than 7832 in filter. In summary, in best case, there will be in the order of $\frac{N}{h} = 7832$ exchanges. Our achieved number of exchanges are ∼40K, which is much higher than the best case; however, much lower than the average case.

Below, we provide two additional lemma to derive the worst case bounds on the number of exchanges for ASketch. However, the probability of occurring such worst case scenarios are very small because the stream requires to be particularly ordered.

LEMMA 2. *Assuming no collision inside the* sketch *(i.e., the sketch is big enough to hold accurate count for all distinct keys), total number of exchanges cannot be more than $\frac{N}{2}$, where N is the total stream size.*

First, we construct an example when this happens. Let there be only two distinct items in a stream, and they are coming in the following sequence: $A\,\underline{BB}\,\underline{AA}\,\underline{BB}\,\underline{AA}\,\underline{BB}$. Assume that the filter size is 1, and the two items A and B do not collide in the sketch. This will result in $\lfloor (N-1)/2 \rfloor$ exchanges for a stream of size N. In particular, every item has an exact count in the sketch based on our assumption. We move an item from the sketch to the filter only when its count obtained from the sketch is larger than the current minimum count stored in the filter. Therefore, after an item is evicted from the filter and before it moves back again to the filter, the item will require at least 2 more hits while staying in the sketch. This is true for any item in the stream. Therefore, the maximum number of exchanges is $\frac{N}{2}$.

LEMMA 3. *Assuming there are collisions in the* sketch *(i.e., the sketch width $h << M$, the number of distinct items), total number of exchanges can be N, where N is the stream size.*

We again construct a scenario when this happens. Let there be only two distinct items in a stream, and they are coming in the following sequence: $AB\,\underline{BA}\,\underline{BA}\,\underline{BA}\,\underline{BA}\,\underline{BA}$. Assume that the filter size is 1, and each hash function in the sketch hashes both A and B to the same cell, i.e., $H_i(A) = H_i(B)$ for all $i \in (1, w)$. Hence, A and B always collide in the sketch. This results in $N-2$ exchanges for a stream of size N. We ensure that there will be no more than N exchanges, because at most one exchange can happen after one insertion in the sketch. Since the total number of insertion in the sketch is bounded by N following Lemma 1, the maximum number of exchanges cannot be more than N.