BitMatcher: Bit-level Counter Adjustment for Sketches

Qilong Shi
Tsinghua University
Peng Cheng Laboratory

Chengjun Jia
Tsinghua University

Wenjun Li*
Peng Cheng Laboratory
Harvard University

Zaoxing Liu
University of Maryland

Tong Yang
Peking University
Peng Cheng Laboratory

Jianan Ji
Peking University

Gaogang Xie
Chinese Academy of Sciences

Weizhe Zhang
Peng Cheng Laboratory

Minlan Yu
Harvard University

***Abstract*—Sketch has been widely used in the field of large-scale data stream processing. However, common fixed-counter algorithms such as Count-Min Sketch have to allocate larger counters, which wastes a lot of memory due to the high skewness of real-world data streams. To reduce memory usage, we propose to dynamically adjust the counter size that matches the distribution of the data stream. We introduce BitMatcher, a fast global-adjusting algorithm that automatically adjusts the counter to the appropriate size to match the data stream. During stream processing, BitMatcher identifies items hashed into a bucket based on isolated fingerprints. If it overflows, BitMatcher changes the flag bits in the bucket and dynamically increases or shrinks the size of some counters in a fine-grained manner. BitMatcher can also relocate a cold item in the bucket with the idea of cuckoo hashing to preserve the potential hot item while achieving global load balancing. Through the above way of dealing with overflow caused by skewed data, BitMatcher precisely manipulates allocated bits and maximizes memory utilization. The experiments show that BitMatcher has high throughput and can outperform SOTA by up to 4 orders of magnitude in terms of accuracy. We also deployed BitMatcher on several platforms, showing its software and hardware scalability.**

***Index Terms*—Data stream, Approximate algorithm, Sketch.**

I. Introduction

Recent years have witnessed that large-scale data stream processing plays an important role in various applications such as network monitoring [2], [3], [4], [5], recommendation systems [6], stock tickers [7], joining tables [8], [9], and more [10], [11], [12], [13]. In these applications, data are often received and processed in a streaming fashion with varying high rates and over the long term [14], [15]; we need algorithmic approaches that can process these data accurately using small memory. For instance, network operators are interested in characterizing various flow events (e.g., heavy hitters and flow size distribution) among Terabits-level network traffic, given resource-constrained network devices.

*Corresponding author: Wenjun Li (wenjunli@pku.org.cn).

Qilong Shi and Chengjun Jia are co-first authors of this paper, and they conducted this work under the guidance of Wenjun Li and Tong Yang. Qilong Shi finished this work when he was a research assistant at Peking University. Wenjun Li finished this work when he conducted his postdoctoral research at Harvard University. The source code of this paper can be downloaded from the website [1].

To support these data applications, (one-pass) sketch algorithms are promising. There are significant recent efforts in optimizing sketch algorithm designs in various aspects, such as memory, accuracy, and processing speed. However, these prior efforts fell short of optimizing the memory-accuracy tradeoffs to cope with the ever-increasing data volume and limited compute/memory resources. A fundamental issue behind these traditional designs is that they consider counters to be fixed in length. With varying data streams, this fixed-length counter design can lead to undesirable wasted memory space. For instance, CM-Sketch [16] is implemented with fixed-size counters and cannot easily adapt to real data streams when the workload distribution varies over time. When there are only a few hot items whose sizes are $\lesssim 2^{32}$, but we use 32 bits for all counters, most of the significant counter bits are wasted.

To adapt to the high-skewness characteristics of real data streams, researchers have developed various algorithms to improve the tradeoff between memory usage and accuracy [17], [18], [19], [20], [21]. These algorithms fall into two main categories: *hierarchical-based* and *self-adjusting based*. *Hierarchical-based* algorithms, such as ASketch [14], structure data into layers to distinguish between frequently and infrequently accessed items. While ASketch employs a filter to collect frequently accessed items, its interaction with the main sketch structure results in only marginal accuracy gains at the expense of speed. Elastic Sketch [22] further develops this approach, yet it struggles to maintain accuracy with limited memory. *Self-adjusting based* algorithms dynamically resize counters based on item frequency, optimizing space without compromising accuracy. SALSA [23] begins with small counters and combines them as needed, although its decoding process is inefficient. DHS [24], on the other hand, uses fixed-size buckets and adjusts counters within a single bucket when hot items overflow. This method, however, faces performance issues as the granularity of adjustments increases.

In summary, existing algorithms are compromised in either performance or accuracy, and fail to adjust the counter size properly for the underlying data distribution. Motivated by this, we ask ***can we design a sketch algorithm that can adjust the counter size in a very appropriate way to accommodate different distributions, while retaining high performance?***

To answer this question, we propose a new sketch algorithm called **BitMatcher**. BitMatcher is able to **match** the suitable **bit**-size counter for the vast majority of items according to their frequency to achieve precise memory allocation under arbitrary data distribution. To achieve this, BitMatcher's design ideas revolve around the following points:

1) We adopt a cuckoo filter-like structure [25] and treat a single bucket as the basic hash unit. Cuckoo kick can be used to balance the load among buckets to achieve the goal of "global coordination".
2) In a single bucket, we divide multiple entries to store multiple items, and each entry is isolated by fingerprints to achieve high accuracy.
3) $\lceil \log_2(state\ number) \rceil$ bits are reserved in a bucket as flag bits to indicate how the remaining bits are allocated by each entry (called the "state" of the bucket). Decoding is very simple to achieve high processing speed.
4) Most importantly, when dealing with overflow caused by highly skewed data, we increase or shrink the size of each entry in a fine-grained manner (changing the state in the bucket). Combined with the global coordination in (1), the precise allocation of counters can finally be achieved.

To evaluate real-world performance, we implement BitMatcher in both software and hardware platforms (e.g., CPU, and FPGA) and perform various measurement tasks including *frequency estimation*, *heavy hitters*, *heavy changes*, *frequency distribution*, and *entropy estimation*. BitMatcher achieves significantly better results than SOTA (i.e., DHS) in large network traffic datasets (errors improved by up to 4 orders of magnitude) and still beats SOTA in small datasets. The experimental results on software and hardware show that BitMatcher reaches high speeds and demonstrate that BitMatcher has real-world feasibility and scalability.

The rest of the paper is organized as follows. Section II introduces the background and latest work of data stream processing. Section III shows the data structure and algorithm of BitMatcher. Section IV gives the mathematical analysis and Section V presents the experimental results. Finally, the whole paper is summarized in Section VI.

II. Background and Motivation

In this section, we first define the data stream model and the tasks we address in this paper. We then discuss the existing works and their limitations. Finally, we give a more intuitive motivation.

A. Problem Statement

• **Data Stream Model:** A data stream S is a sequence of N items $\langle e_1, e_2, ..., e_N \rangle (e_i \in E)$, where E is the item set. Items in E can appear more than once, and the algorithm should process items in order to support online queries. The frequency of an item $e_i(e_i \in E)$ is defined as $f_i \triangleq \sum_j I\{e_i = e_j\}$ where I is the indicator variable (either 0 or 1). As shown in Fig. 1, $f_1 \sim f_4$ can be calculated by counting the number of times they appear in the data stream.

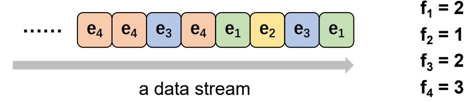


Fig. 1: A data stream.

Measurement Tasks: We aim to design a sketch algorithm that tackles the following measurement tasks.

• *Frequency Estimation:* Given a data stream $S = \langle e_1, e_2, ..., e_N \rangle$, we want to (approximately) measure the frequency of each item as accurately as possible (i.e., $f_i(\forall e_i \in E)$). Frequency estimation is the basis of many applications such as joining tables [8], [9] and multi-set querying [10]. There are many sketches that can be applied to this task, including the 8 sketches mentioned in this paper, as well as Cuckoo Counter [13] and Stingy Sketch [26].

• *Heavy Hitter Detection:* Finding an item set E_{hh} satisfying that $\forall e_i \in E_{hh}, f_i > \theta_{hh} \times N$, where θ_{hh} is a predefined threshold. Heavy hitter detection is important in data-intensive applications like recommendation systems and information retrieval [24]. In addition to the 8 algorithms mentioned in the article, Tight-Sketch [27] can also be applied to this task.

• *Heavy Changes:* For each item appearing in the time window T or $T-1$ (a data stream can be divided into several time windows according to each item's arrival time), its change degree can be denoted as $\Delta = |f_i^T - f_i^{T-1}|$. Heavy change detection aims to find the items whose Δ is larger than $\theta_{hc} \times N$ (θ_{hc} is a predefined threshold). Generally speaking, all sketches that can be applied to heavy hitter detection can be applied to heavy changes.

• *Item Size Distribution:* Estimating the number of items belonging to each specific size. Item size distribution estimation is widely used in database query load balancing [28] and anomaly detection [29] by distribution. The sketches that can be applied to this task in recent years are the DHS [24] and Elastic Sketch [22] compared in this paper.

• *Entropy Estimation:* It returns the entropy of the data stream to describe its distribution and uncertainty. Data stream entropy is applied in data mining work, including clustering [30] and data quality evaluation [31]. Similarly, the sketches that can be applied to this task in recent years are DHS [24] and Elastic Sketch [22].

B. Related Work

The sketch is widely used in data stream processing. The most widely used sketch is the Count-Min sketch (CM) [16]. It consists of d arrays A_1, A_2, ..., A_d. Every array is a fixed-size counter. For each item e, CM picks one counter per array by independent hash functions $h_i(\bullet)$, and increases all mapped counters $A_i[h_i(e)]$ by 1. When querying frequency, CM reports the minimum value of all mapped counters. As an improvement, NitroSketch [32] uses geometric sampling to exchange a small amount of accuracy for a very high update speed. However, due to the high skewness of real data streams, common CM-based methods have much memory waste. To overcome this, hierarchical and self-adjusting sketches were invented.

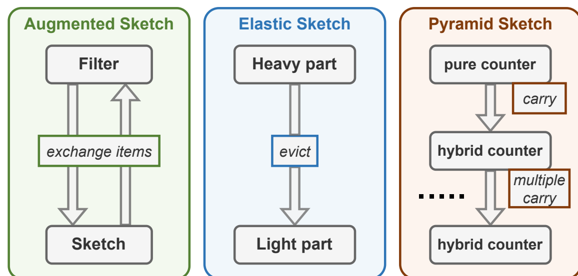


Fig. 2: The latest hierarchical sketches.

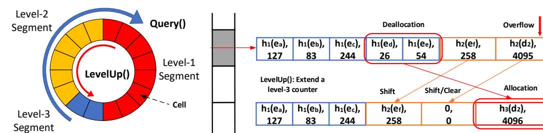
1) Hierarchical Sketches: Fig. 2 shows a typical sketch of three hierarchical structures. It can be found that they all divide their structure into multiple layers, which are used to store hot and cold items respectively, so as to achieve the effect of adapting to the real data stream.

Augmented Sketch (AS) [14] uses an additional filter to aggregate the hot items in advance. When an item arrives, AS first looks for it in the filter, and if it is not found, AS inserts it into the subsequent normal sketch (such as CM). After that, it checks whether the frequency of this item is greater than the smallest in the filter, and if so, exchanges them. This ensures that there are always hot items in the filter. However, this exchange process can significantly slow down processing.

Pyramid Sketch [19] divides its structure into a pure counter at the bottom and a mixed counter above it, with a total of λ layers. The number of counters of layer i is half of that of layer $i-1$. The first layer is a normal sketch, and the second layer and above are used for automatic carry. With this structure, it not only prevents counters from overflowing without the need of knowing the frequency of the hot items in advance, but also achieves high accuracy and high throughput at the same time. But whenever querying the hot item, it will access multiple layers and thus decrease the speed, making it difficult to perform tasks that focus on hot items.

Elastic Sketch (EL) [22] is the state-of-the-art hierarchical algorithm. It divides the structure into a heavy part and a light part to accommodate hot items and cold items respectively. It uses an algorithm similar to "Ostracism" to ensure the accuracy of the separation of hot/cold items. Specifically, each bucket in the heavy part stores three fields: item ID, positive votes, and negative votes. Given an incoming item with ID e_1, if it is the same as the item in the bucket, EL increments the positive votes. Otherwise, EL increment the negative votes, and if $\frac{\#negative\ votes}{\#positive\ votes} \geq \lambda$, where λ is a predefined threshold, EL expels the item from the heavy part, and insert e_1 into the light. Unfortunately, the hot item can be accidentally expelled in the process.

From the above, we can see that the existing hierarchical algorithms have problems and it is actually because a data stream cannot be easily divided into multiple classes. There are many types of items in a data stream, and their frequency ranges are very different. Therefore, other algorithms, based on self-adjustment, are proposed to adjust the size of the counter as the data stream comes, and thus be able to match its distribution.



(a) The structure of SALSA [23].

(b) The structure of DHS [24].

Fig. 3: The latest self-adjusting sketches.

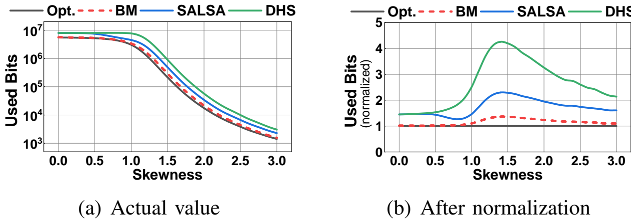
2) Self-adjusting Sketches: Fig. 3 shows the two most famous self-adjusting sketch algorithms, and our BitMatcher is also classified into this type. Fig. 3(a) shows the structure of **SALSA** [23], a typical self-adjusting sketch. Initially, SALSA uses only one 4-bit $char$ rather than 32-bit int to estimate frequency. It establishes an extra bitmap to tag overflowing counters and merges small neighboring counters to form a bigger one dynamically. As shown in Fig. 3(a), $Values[4] \sim Values[7]$ have been merged into one large counter (because a hot item has entered one of these four counters). At this time, $Merges[4] \sim Merges[6]$ (the extra bitmap) will be set to 1, and $Merges[7]$ will be set to 0 (that is, all except the last bit are set to 1). SALSA achieves high accuracy, but its shortcoming is obvious: The extra bitmap is space-consuming and decreases the speed.

Fig. 3(b) shows the state-of-the-art self-adjusting algorithm, **Dynamic Hierarchical Sketch** (DHS) [24]. It consists of many fixed-size buckets (128 bits), and the buckets contain three kinds of counters (the size of 8, 12, and 16 bits, respectively). When an item in a smaller counter overflows, DHS will reallocate the space in the bucket and move the item to the larger counter (achieved by sacrificing the small counter). As shown in Fig. 3(b), when the item in a 12-bit counter overflows, it sacrifices two 8-bit counters and merges them into a 16-bit counter, and then puts the item into it. DHS's counter adjustments are all within a single bucket, which is fast. But it cannot record items with frequencies greater than 2^{16}, which are common in the real world. This is because the coarse-grained adjusting strategy makes DHS hard to match streams with high skewness in practice. But in general, self-adjusting algorithms are better able to adapt to highly skewed data streams than hierarchical algorithms.

C. Insights about Motivation

Assuming that there is an item e_1 with a frequency of 500 ($f_1 = 500$), the ideal situation would be to assign it a 9-bit counter (since it can record the frequency between 0 and $2^9 = 512$). However, DHS only has counters with 8, 12, or 16 bits, so it can only allocate a 12-bit counter; SALSA has counters with 4, 8, 16, or 32 bits, so it must allocate a 16-bit counter. To quantify the effectiveness of self-adjusting algorithms on counter allocation, "Optimal counter size" will be formally defined here, and we will look at how far existing algorithms are from this "optimum", to give more insights about motivation.

For a data stream S and its item set E, the number of bits we need is: $\sum_{e_i \in E}(\lfloor \log(f_i) \rfloor + 1)$. This is because, from the perspective of information theory, to record an item with frequency f_0, the minimum number of bits required will not be lower than $\log(f_0)$. But since the minimum unit of memory operation is 1 bit, $(\lfloor \log(f_i) \rfloor + 1)$ bits are necessary for this item. Similarly, the used bit of an algorithm is *[counter bits used to store f_i]*. For example, the item e_1 with $f_1 = 500$ mentioned in the previous paragraph should ideally be loaded with a 9-bit counter (since $\lfloor \log(500) \rfloor + 1 = 9$). BitMatcher (i.e., BM) is more likely to reach the optimum, while DHS has to use a 12-bit counter, and SALSA uses a 16-bit counter. Adding up the required bits of all items in a data stream yields the total used bits.



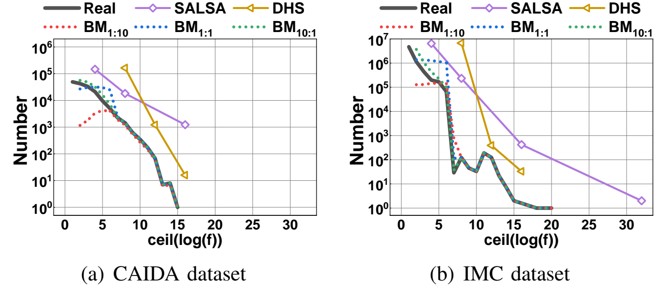
(a) Actual value (b) After normalization

Fig. 4: Used bits and its normalized form.

As shown in Fig. 4(a), we compared the used bits of SALSA and DHS under various skewness datasets. The *Opt.* curve shows a decreasing trend with increasing skewness, which is because our dataset has the same total number of items (i.e., $|S| = \sum_{e_i \in E} f_i$), and the skewness of the data leads to a decrease in item types (i.e., $|E|$), and then reduced *Opt.*. Note that the used bits of SALSA and DHS are far from optimal. The normalized results are shown in Fig. 4(b). When the skewness > 1.2 (common in real data streams), SALSA uses more than 2x the optimal value, and DHS is as high as 4x, while our algorithm does not exceed 1.5x. This means that if storing a data stream optimally needs to allocate 10MB of counters, then SALSA needs 20MB, DHS needs 40MB, and BM only needs $<$ 15MB. Typical hardware platforms and switches usually only have $<$ 30MB of memory, so optimizing the memory utilization of sketch is very important.

Fig. 5 shows the real counter distribution (based on the real item distribution), the counter distribution for SALSA, DHS and BM under different memories. The lines in the figure are all discrete and only meaningful when the x-coordinate is an integer (such as $x = 1, 2...$). The black line ($Real$) represents the optimal counter distribution for a data stream S. Assuming that there are y kinds of items (i.e., $\{e_1, e_2, ..., e_y\}$) in S with frequencies between 2^{x-1} and 2^x (i.e., $2^{x-1} \leq f_1, f_2, ..., f_y < 2^x$), then the best case is to accommodate them with y x-bit counters, so the black line will pass through the point (x, y). For example, as shown in Fig. 5(a), since the CAIDA dataset has 9 items with a frequency between 2^{13} and 2^{14} so the black line passes through $(14, 9)$. Simply put, if a data stream S is a sequence of N items $\langle e_1, e_2, ..., e_N \rangle (e_i \in E)$, where E is the item set. Then its black line in Figure 5 will pass through all such $(X, Y) \in \{(x, y) | y = |\{e_i | e_i \in E, 2^{x-1} \leq f_i < 2^x, x \in \mathbb{N}^+\}|\}$. For each algorithm, their lines will pass through (X, Y) if and only if there are Y non-empty counters of X bits in their data structures.

For DHS (yellow line), because it has only three types of counters (8, 12, and 16 bits), its x-coordinate can only take 8, 12, and 16. Each yellow line shows the optimal counter distribution of DHS when the memory is large enough. That is, DHS stores all items with frequency $< 2^8$ in an 8-bit counter, and the items with frequency between $2^8 \sim 2^{12}$ are stored in 12-bit counters, and so on. The same goes for SALSA (purple line). For BM, we plotted it for different relative memory. $BM_{1:10}$ in the figure indicates that $\frac{number\ of\ counters}{item\ types} = \frac{1}{10}$, which is a very tight memory.



(a) CAIDA dataset (b) IMC dataset

Fig. 5: Distribution of various-sized counters.

As shown in Fig. 5, we measured two real-world datasets (i.e., CAIDA and IMC). Details of these datasets can be found in § V-A. We can find that the optimal counter lines of SALSA and DHS deviate significantly from the Real line, while BM fits the hot item part (the right part of the black line) well under different relative memories. For the cold item part (the left side), when the memory changes from $BM_{1:10}$ to $BM_{1:1}$, BM will waste a small amount of space, and it will gradually fit when the memory changes to $BM_{10:1}$. From the above discussion, we find that the memory utilization of DHS and SALSA is low. This can also be seen in the experimental part of § V. When memory size is tight, the accuracy of BM is much ahead of DHS and SALSA. BitMatcher achieves extremely fine-grained memory allocation due to its bit-level counter adjustment, which makes it superior to the two current self-adjustment algorithms.

III. BitMatcher Framework

In this section, we describe the data structure and algorithm of BitMatcher. A data stream processing algorithm should provide two fundamental interfaces: *Insert()* and *Query()* in order to support each processing task.

A. Data Structure

As shown in Fig. 6, BitMatcher consists of two arrays, A_1 and A_2. We assume that each array has w buckets, and each bucket consists of B entries with different sizes: $\{entry_1, entry_2, ..., entry_B\}$ (the size of $entry_i$ increases as i increases, and B may change as the data stream comes). An entry is the smallest unit of BitMatcher, and a bucket is the unit of one access. As a result, we want the length of a bucket to be an integer multiple of a machine word (e.g., 64bits, 128bits...), and there will be no waste of memory

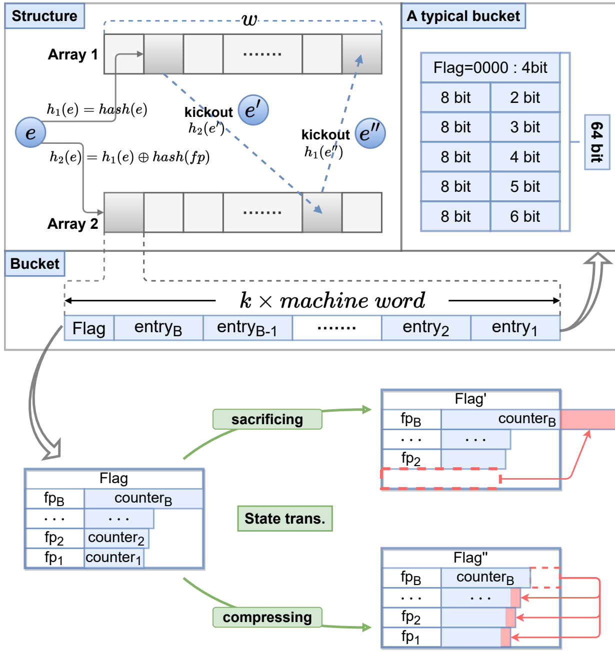


Fig. 6: The data structure of BitMatcher.

access. Each entry consists of two parts, $\underline{fingerprint}$ and $\underline{counter}$. We employ the fingerprint to identify the item (e.g., e_x); it is derived from an independent hash function $fp(e_x)$. All fingerprints must occupy the same size space to meet the requirements of our algorithm. Note that the distribution of each entry (or counter) in the bucket will change with the processing of the data stream. We call this the "state" of a bucket. The $\underline{Flag}$ bit in the bucket shows the state of the bucket at this time. The transition of buckets between states follows two core ideas: *sacrificing* (the smallest counter) and *compressing* (the largest counter). We will introduce it in detail in the following subsection. Our algorithm will naturally place hot items in larger entries and cold items in smaller entries. The upper right corner of Fig. 6 is a typical 64-bit size bucket in its initial state.

Notice that we introduce partial-key cuckoo hashing [25] to derive an item's alternate location based on its fingerprint. For an item e, the details of calculating the index of two candidate buckets are as follows, $h_1(e) = hash(e)$, $h_2(e) = h_1(e) \oplus hash(e's\ fingerprint)$. The $\oplus$ (XOR) in the formula guarantees that $h_1(e)$ can also be computed from $h_2(e)$ and $e's\ fingerprint$, which means that $h_1(e) = h_2(e) \oplus hash(e's\ fingerprint)$. Hence, no matter which array the item is now in, we can calculate the location of the item in the other array by its current position and fingerprint:

$$h_{another} = h_{current} \oplus hash(item's fingerprint)$$

It should be noted that our data structure of 2 arrays and one cuckoo hashing function may be similar to that of the Cuckoo filter [25], but the similarity stops there. First, these two algorithms have different focuses. Cuckoo filter is only used for set membership query, that is, to determine whether an item is in a given set. BitMatcher focuses on generality. It

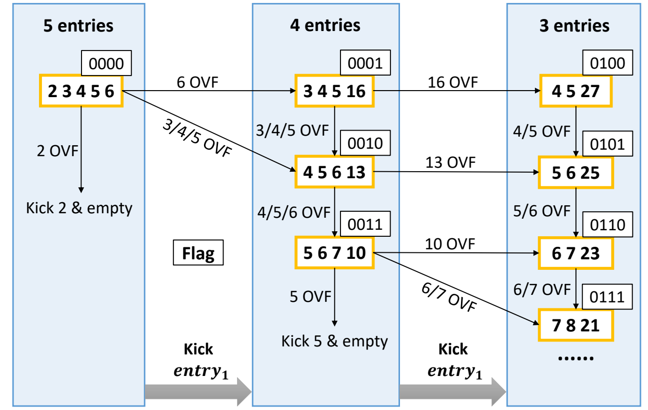


Fig. 7: State transition table and rules.

can handle five different tasks. Second, and most importantly, the core innovation of BitMatcher is the dynamic bit-level counter adjustment based on "state transition" in a single bucket (which will be introduced shortly). The reason for introducing cuckoo hashing is to deal with some special cases that state transitions cannot handle.

B. Algorithm and Operations

Insert: Pseudocode can be found in Algorithm 1. For brevity, we use $A_i[j][k]$ to refer to $array_i[bucket_j][entry_k]$. Initially, all entries are set to 0. When inserting an item e, we first compute two indexes by hashing, $h_1(e)$ and $h_2(e)$ to find two candidate buckets, $A_1[h_1(e)]$ and $A_2[h_2(e)]$. Then we scan all entries in these two buckets. If item e exists, we increment the counter of the corresponding entry by 1. If item e is new, we insert it into the empty entry and set the counter value to 1. If two buckets are full, we randomly select $A_1[h_1(e)][1]$ or $A_2[h_2(e)][1]$ and decrease it by 1 (we can also use other replacement strategies here, such as "Exponentially -1 [33]", "RAP [34]" or "USS [35]", but experiments have shown that "Directly -1" has the best effect). After insertion, if an overflow occurs, we first try to place or exchange it into a larger entry, just like line 9 of Algorithm 1. If it fails, the following in-bucket state transitions are required.

State transition: Pseudocode can be found in Algorithm 2. The main part of our algorithm is described next. For convenience, we assume that the bucket size is fixed at 64 bits (other sizes are similar), the fingerprint size is fixed at 8 bits, and the flag bit is 4 bits. As shown in Fig. 7, $\langle x_1\ x_2\ ...\ x_B \rangle$ in each yellow box represents a state of the bucket, and the black boxes adjacent to them are the corresponding flag bits. When accessing a bucket, we first use the flag bits to decode the number of entries and the counter size of each entry. For example, the initial state $\langle 2\ 3\ 4\ 5\ 6 \rangle$ on the left represents that there are 5 entries in the bucket at this time while the flag bits are $\boxed{0000}$, and the counter fields occupy 2/3/4/5/6 bits respectively. It can be verified that fingerprints occupy 8 bits$\times 5 = 40$ bits, counters occupy $2+3+4+5+6 = 20$ bits, and flag occupies 4 bits, so the total size is 64 bits. Similarly, if we find that the flag of a bucket is $\boxed{0001}$, we can decode the state in the bucket as $\langle 3\ 4\ 5\ 16 \rangle$ according to Fig. 7.

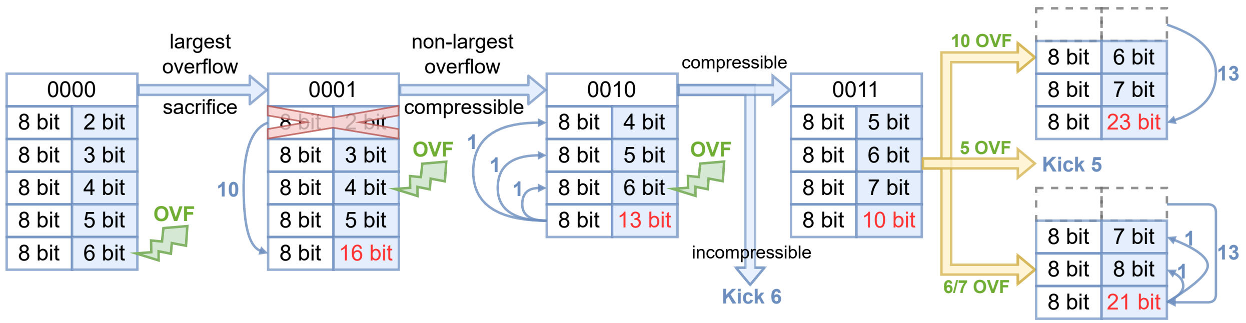


Fig. 8: A running example of BitMatcher.

Next, we introduce the state transition rules with a running example. As mentioned in the previous subsection, the core idea of BitMatcher can be summed up in two points: **sacrificing** the smallest counter and **compressing** the largest counter. During the process, there are mainly the following two cases.

Case 1: If the largest $entry_B$ in the bucket overflows (it contains the hottest item), we **sacrifice** the smallest $entry_1$ in the bucket and allocate all its space to the counter field of $entry_B$. As shown in the first frame of Fig. 8, if the 6-bit counter ($entry_5$) in $\langle 2\ 3\ 4\ 5\ 6\rangle$ overflows, we sacrifice the smallest $entry_1$ and assign all these 10 bits to $entry_5$'s counter, then the state becomes $\langle 3\ 4\ 5\ 16\rangle$, and its flag bit changes from $\boxed{0000}$ to $\boxed{0001}$. "OVF" is short for "overflow".

Case 2: If a non-maximum $entry_i$ $(i < B)$ in the bucket overflows, we try to **compress** the counter field of $entry_B$, reduce its space by $(B-1)$ bits and divide it equally among the remaining $B-1$ entries (i.e., $entry_1 \sim entry_{B-1}$). As shown in the second frame of Fig. 8, if the 4-bit counter ($entry_2$) overflows, we try to compress the largest 16-bit counter by 3 bits and divide it equally among the smaller counters. If the compression is successful, the status becomes $\langle 4\ 5\ 6\ 13\rangle$, and the flag bit changes from $\boxed{0001}$ to $\boxed{0010}$.

If the smaller counter (e.g., the 6-bit) of $\langle 4\ 5\ 6\ 13\rangle$ continues to overflow, keep trying to change the state to $\langle 5\ 6\ 7\ 10\rangle$. There are two cases here: ① If it is incompressible (that is, the item frequency in the 13-bit counter is greater than 2^{10} and less than 2^{13}), we use the cuckoo hash to kick out the overflowed small counter. ② Otherwise, we just compress it. Examples are in the 3rd $\sim$ 4th frame in Fig. 8.

At this point, the largest counter is no longer compressible (otherwise the state would be $\langle 6\ 7\ 8\ 7\rangle$, which breaks our rule). There are three cases here: ① If the largest counter overflows, we sacrifice the smallest entry as before. ② If the smallest counter overflows, we will kick it out with cuckoo hash and empty it (the maximum number of kicks is limited by a parameter named $maxloop$, generally $= 1$ to meet the loading rate). ③ Otherwise, we sacrifice the smallest entry to the largest counter and then compress it. Examples are in the 4th $\sim$ 5th frame in Fig. 8.

General state transition rules are shown in Fig. 9, where $C[X, Y]$ represents the Y_{th} state containing X entries and $cnt_{min/max}$ refers to the smallest/largest counter. For example, $\langle 4\ 5\ 6\ 13\rangle$ in Fig. 7 is $C[4, 2]$, because it is the second state at the "4 entries" frame. Similarly, $\langle 5\ 6\ 25\rangle$ is $C[3, 2]$.

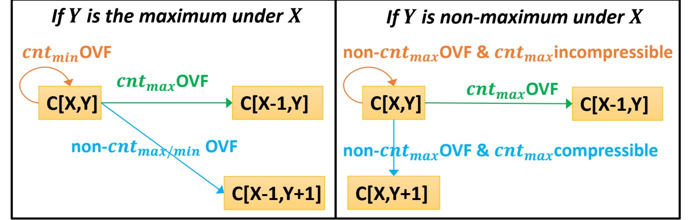


Fig. 9: General state transition rules in BitMatcher.

Algorithm 1: Insert(e)

Input: an item p belonging to $e \in E$

1 $fp = fingerprint(e)$, $maxloop \geq 0$;
2 $h_1 = hash(e)$, $h_2 = h_1(e) \oplus hash(fp)$;
3 $entry.fp/cnt$ to refer to $entry.fingerprint/counter$;
4 Decode current state $C[X, Y]$ by $flag$, X is the number of entries in a bucket;
5 i, $i' \in \{1, 2\}$, $i + i' = 3$, j from X down to 1;
6 **if** $fp == A_i[h_i(e)][j].fp$ **then**
7 $\quad A_i[h_i(e)][j].cnt{+}{+}$;
8 $\quad$ **if** $A_i[h_i(e)][j]$ *overflow* **then**
9 $\quad\quad$ **Stay_overflow**($A_i[h_i(e)][j]$);
10 $\quad\quad$ **if** *failed* **then**
11 $\quad\quad\quad$ **State_transition**($C[X, Y], j$);
12 **else if** $A_i[h_i(e)]$ *has an empty entry* **then**
13 $\quad$ insert e into the entry;
14 **else if** $A_i[h_i(e)][1] -- == 0$ **then**
15 $\quad$ put $\{fp, 1\}$ to $A_i[h_i(e)][1]$;
16 **Function** `Stay_overflow`($A_i[h][j]$)**:**
17 $\quad$ **if** *has* $A_i[h][k].cnt$, $(k > j)$ *is smaller* **then**
18 $\quad\quad$ swap($A_i[h][j]$, $A_i[h][k]$);
19 $\quad\quad$ **return** 1;
20 $\quad$ **return** 0;

Query: When querying an item e, we calculate two indexes firstly, $h_1(e)$ and $h_2(e)$, by partial-key cuckoo hashing. Then we match the fingerprint of e with these fingerprints in $A_i[h_1(e)]$ $(i \in \{1, 2\},\ 1 \leq j \leq B)$. If matched, we return the counter of the corresponding entry. Or if there is at least 1 empty entry, we return 0. Otherwise, we just return the minimum of the two buckets.

Algorithm 2: State transition($C[X,Y],j$)

1 **if** *Y is the maximum under current X* **then**
2   Kickout($maxloop$, $A_i[h][1]$);
3   **if** *j==X* **then**
4   change from $C[X,Y]$ to $C[X-1,Y]$;
5   **else if** *j==1* **then**
6   stay in $C[X,Y]$;
7   **else**
8   change from $C[X,Y]$ to $C[X-1,Y+1]$;
9 **else**
10   **if** *j==X* **then**
11   Kickout(maxloop, $A_i[h][1]$);
12   change from $C[X,Y]$ to $C[X-1,Y]$;
13   **else if** *$A_i[h][X]$ can be compressed* **then**
14   change from $C[X,Y]$ to $C[X,Y+1]$;
15   **else**
16   Kickout($maxloop$, $A_i[h][j]$);
17 **Function** `Kickout` ($maxloop$, $A_i[h][j]$)**:**
18     $rh = h \oplus hash(A_i[h][j].fp)$;
19   **if** *$A_{i'}[rh]$ has an **empty** and **capable** entry* **then**
20   put $A_i[h][j]$ into this entry; **return**;
21   choose a smallest capable entry $A_{i'}[rh][k]$;
22   **if** *$maxloop--> 0$* **then**
23   Kickout($maxloop$, $A_{i'}[rh][k]$);
24   put $A_{i'}[h][j]$ into $A_{i'}[rh][k]$;

IV. Mathematical Analysis

In this section, we analyze the expected AAE for the frequency estimation of BM. Then we define the "loading rate" to represent the load status of BM at a certain time.

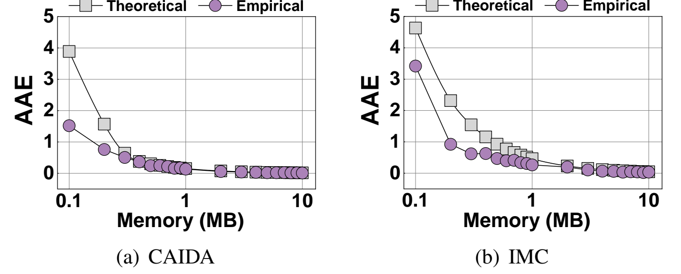
A. Average Absolute Error (AAE)

The calculation formula of AAE is defined as $\frac{1}{|\Psi|}\sum_{(e_i \in \Psi)}|f_i - \tilde{f}_i|$, where f_i is the real frequency of item e_i, $\tilde{f}_i$ is the estimated frequency, and the Ψ is the query set. We have the theorem: $E(AAE) \approx \frac{N}{w2^{\mathcal{F}}}$, where N is the total number of items, w is the width of the BM, and $\mathcal{F}$ is the length of the fingerprint. Detailed proofs can be found in supplemental material [36].

We use the CAIDA and IMC datasets. For detailed information, please refer to § V-A. The experimental results are shown in Fig. 10 with $0.1 \sim 10$ MB memory. The black line is the empirical AAE, and the red line is the theoretical expectation of AAE. $\mathcal{F} = 8$ bits. We can find that the theoretical values fit fairly well.

B. Space and Time Complexity

We can obtain the following time and space complexity in Table I. Notice that the d and w are the depth and width.



(a) CAIDA (b) IMC

Fig. 10: Empirical and theoretical AAE.

Sketch	d	w	Space	Insert	Query
CM[1]	$\log\frac{1}{\delta}$	$\frac{2}{\epsilon}$	$O(\frac{1}{\epsilon}\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
NI[2]	$\log\frac{1}{\delta}$	$O(\frac{1}{\epsilon^2 p}+\frac{\sqrt{\log\frac{1}{\delta}}}{\epsilon^2 p^{1.5}\sqrt{m}})$	$O(\frac{\log\frac{1}{\delta}}{\epsilon^2 p}+\frac{\log^{1.5}\frac{1}{\delta}}{\epsilon^2 p^{1.5}\sqrt{m}})$	$O(p\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
DHS[3]	1	$\sum_{k=1}^{3}\frac{\lambda_k}{\delta\epsilon 2^{l_k}}$	$O(\sum_{k=1}^{3}\frac{\lambda_k}{\delta\epsilon 2^{l_k}})$	$O(1)$	$O(1)$
BitMatcher	2	$\frac{1}{\delta\epsilon 2^{\mathcal{F}}}$	$O(\frac{1}{\delta\epsilon 2^{\mathcal{F}}})$	$O(1)$	$O(1)$

TABLE I: Comparison of BitMatcher with SOTA.

Note that some algorithms in the above table have self-contained parameters, so please go to the corresponding original paper if necessary.

C. Loading Rate

First, we introduce the definition of the loading rate.

Definition IV.1. *If the BitMatcher contains $2w$ buckets, of which x buckets are full (no empty entries), then the **loading rate** at this time is $\frac{x}{2w}$.*

Then we have the following theorems. Detailed proofs are in supplemental material [36].

Theorem IV.1. *Assuming that the BitMatcher has $2w$ buckets, each bucket has an average of B entries, each array has 1 candidate bucket, and n types of items arrive at this time. Then for the loading rate LR at this time, we have:*

$$LR \geq 1 - \left(\sum_{i=0}^{\lceil B \rceil - 1}\frac{\left(\frac{n}{2w}\right)^i}{i!}\right)e^{-\frac{n}{2w}} \quad (1)$$

Theorem IV.2. *Under the conditions of Theorem IV.1, and we consider the fingerprint effect. Assuming that the theoretical upper bound of the loading rate is LR_{opt}, we have:*

$$LR \leq LR_{opt} = min(\frac{n}{2w\sum_{i=0}^{B-1}\frac{2^F}{2^F - i}}, 1) \quad (2)$$

We use the CAIDA dataset to validate our conclusions. As shown in Fig. 11, we show the theoretical optimal value and theoretical lower bound of the loading rate of BM, and the empirical value is recorded for every 5K items. The experimental results validate our theory. This also shows that BitMatcher has very few hash collisions.

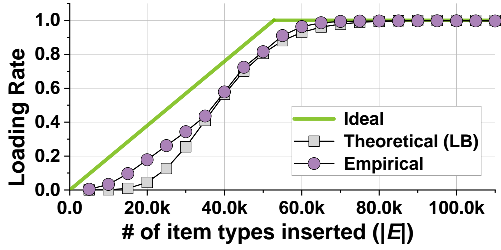


Fig. 11: Empirical and theoretical LR.

V. Performance Evaluation

In this section, we conduct experiments. We have released our source code on the website [1], as well as the GitHub [36].

A. Experiment Setup

1) Test Platform:

We performed all experiments on a machine with Intel $i7-9700CPU@3.0GHz$ and 16G DRAM. The OS is Ubuntu 20.04. To reduce the CPU jitter error, we take the average results by running 10 times for each evaluation circularly.

2) Datasets:

1) CAIDA Datasets: We use the CAIDA trace which was collected in Equinix-Chicago monitor from CAIDA [37]. It contains 165K kinds of items and 2.49M items in total. The maximum item size is 17K.

2) Campus Datasets: We obtain the real IP traces from the main gateway at our campus. The dataset is the same as that used in the Pyramid Sketch [19]. It contains 1M kinds of items and 10M items in total. The maximum item size is 25K.

3) IMC DC Trace: IMC Data Center Trace [38] is collected from the data centers studied in [39]. It contains 6.71M kinds of items and 19.86M items in total. The maximum item size is 0.69M.

4) Zipf Datasets (synthetic): We generate a series of synthetic traces that follow the Zipf [40] distribution using Web Polygraph [41]. The skewness of the traces ranges from 0.0 to 3.0. Each trace contains 32.0M items. The number of item types decreases as the skewness increases. When the skewness = 0.0, there are 1M kinds of items; when the skewness = 3.0, there are 350 kinds of items. The maximum item size ranges from 123K to 18.1M.

We consider CAIDA and Campus as a common dataset and IMC as a large dataset. Because the number of item types of the latter is much larger that of the former. The Zipf dataset is artificially synthesized to test the performance of each algorithm under different skewness.

B. Algorithms and Tasks

1) Comparing Algorithms:

We implement our BitMatcher (BM) in C++, and compare our results with CM sketch (CM) [16], Augmented sketch (AS) [14], Dynamic Hierarchical Sketch (DHS) [24], Pyramid sketch [19], Elastic sketch (EL) [22], Nitro sketch (NI) [32] and Self-Adjusting Lean Streaming Analytics (SALSA) [23]. Because the Pyramid CU sketch (PCU) has the highest accuracy [19], we use PCU sketch as the representative of Pyramid sketch. And for SALSA we choose $SALSA_{CM}$. For CM, AS, DHS and PCU, we use their open-source codes; For EL, NI, and SALSA, we implemented them by ourselves.

The entry sizes in CM and AS are 16 or 32 bits, depending on the maximum item size in datasets. CM and CU allocate 4 arrays and use 4 32-bit Bob hash [42] functions for items mapping. AS consists of the widely used CM sketch and a filter. The filter will allocate about 0.4KB of additional memory, and the CM sketch of AS also includes 4 arrays and 4 32-bit Bob hash functions. The size of a single bucket of DHS is 128 bits, which contains three levels of entries, whose fingerprints occupy 8 bits, and counters occupy 8, 12, and 16 bits respectively. All entries of the PCU are 4 bits, and the number of mapped entries is 4. The PCU uses one 64-bit Bob hash function. EL's heavy part contains 8 entries in each bucket, and the depth of the CM sketch in the light part is 1. The depth of the count sketch of NI is 4, and the geometric sampling rate is $p = 0.01$. The initial counter size of SALSA is 4 bits and the maximum is 32 bits. The bucket size of BitMatcher is 64 bits, the initial state is $\langle 2\ 3\ 4\ 5\ 6 \rangle$. The state transition rules are the same as in Fig. 7.

2) Measurement Tasks:

Five tasks are used to measure the algorithms.

Frequency Estimation: Reporting the size of each item. It is implemented by using the $Query()$ function to query each item.

Heavy Hitter (HH) Detection: It is implemented by using an auxiliary counter to record the total item number within a measurement interval and computing the threshold $\theta_{hh} \times N$, and storing and returning the items whose frequency is larger than $\theta_{hh} \times N$ into a set F_{hh}.

Heavy Change (HC) Detection: It needs two same data stream processing structures to store items in two neighboring time windows $T-1$ and T respectively. For each target item e, its change degree is $\Delta = |query^T(e.id) - query^{T-1}(e.id)|$. Items whose change degrees are larger than a threshold will be identified as heavy changers and stored in a set F_{hc}.

Item size distribution: It is implemented by using auxiliary memory: using $Query()$ to get the size of each item and counting the number of items of each size N_i $(|\{e|e \in E, e.f = N_i\}|)$ in the auxiliary memory.

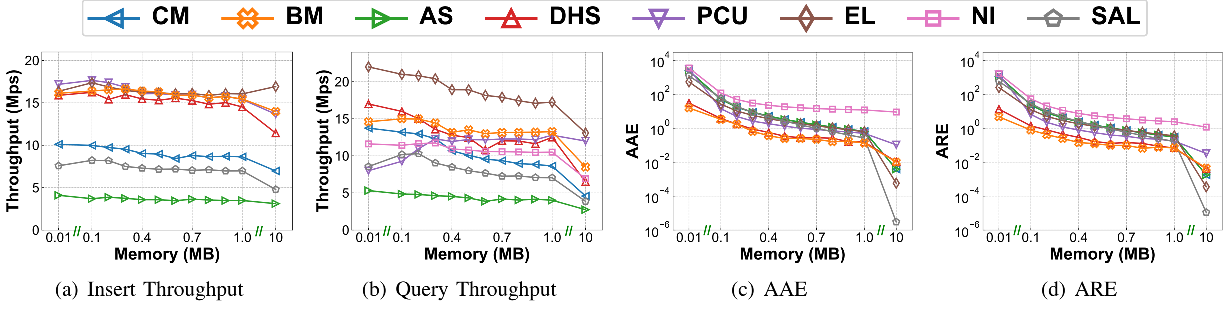
Entropy Estimation: We compute the entropy based on the item size distribution as $-\sum(i * \frac{n_i}{m} log\frac{n_i}{m})$, where m is the sum of n_i, and n_i is the number of items with a size of i.

C. Metrics

We consider the following metrics.

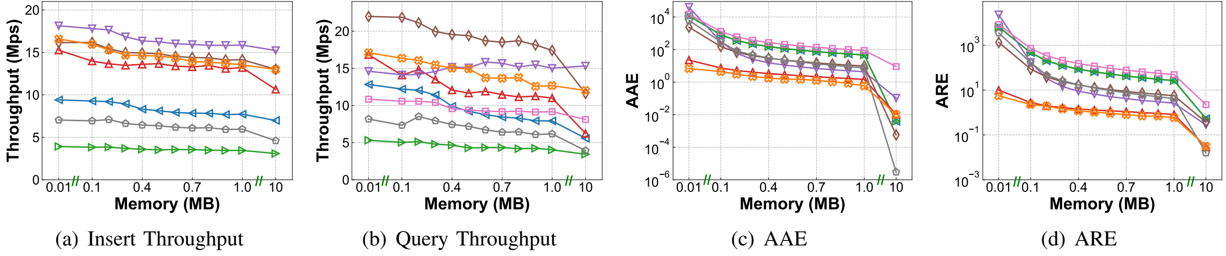
Throughput: Throughput is used to measure the processing speed of the insertion and query. The formula is $\frac{N}{T}$, where N is the number of items, and T is the running time. We use a million insertions per second (Mps) to represent throughput.

AAE: AAE is defined as $\frac{1}{|E|}\sum_{(e_i \in E)} |f_i - \tilde{f_i}|$, where f_i is the real frequency of item e_i, $\tilde{f_i}$ is the estimated frequency, and the E is the query set. We query each distinct item in the dataset once.



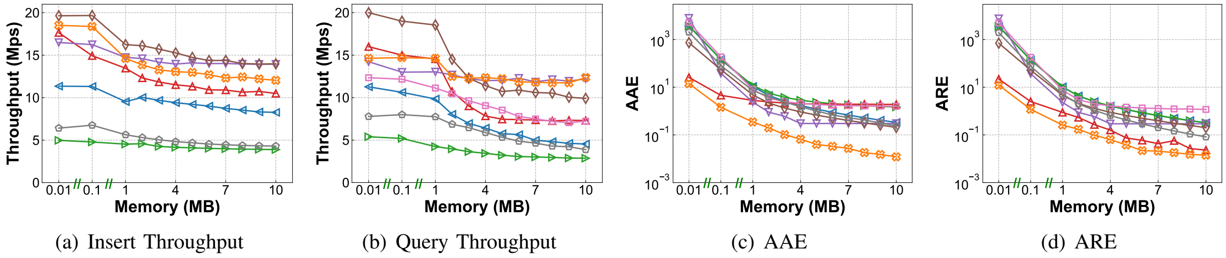
(a) Insert Throughput (b) Query Throughput (c) AAE (d) ARE

Fig. 12: Frequency estimation - Common dataset - CAIDA.



(a) Insert Throughput (b) Query Throughput (c) AAE (d) ARE

Fig. 13: Frequency estimation - Common dataset - Campus.



(a) Insert Throughput (b) Query Throughput (c) AAE (d) ARE

Fig. 14: Frequency estimation - Large dataset - IMC.

ARE: ARE is defined as $\frac{1}{|E|}\sum_{(e_i \in E)} \frac{|f_i - \tilde{f}_i|}{f_i}$. These parameters in the formula have the same meaning as in AAE.

F1-score: $\frac{2 \times PR \times RR}{PR+RR}$, where PR (Precision Rate) refers to the ratio of true instances reported and RR (Recall Rate) refers to the ratio of reported true instances. We use the F1-score to evaluate the accuracy of heavy hitter and heavy change detection.

WMRE (Weighted Mean Relative Error) [2], [43]: $\frac{\sum_{i=1}^{z} |n_i - \tilde{n_i}|}{\sum_{i=1}^{z} (\frac{n_i + \tilde{n_i}}{2})}$, where z is the maximum item size, and n_i and $\tilde{n_i}$ are the true and estimated numbers of items of size i respectively. We use WMRE to evaluate the accuracy of the item size distribution (ISD).

RE (Relative Error): $\frac{|True - Estimate|}{True}$, where $True$ and $Estimate$ are the true and estimated values, respectively. We use RE to evaluate the accuracy of entropy estimations.

D. Frequency Estimation

In this part, we use three datasets to measure the performance of BM in the common case (CAIDA), the large data case (IMC), and different skewness (Zipf) to verify BM's adaptivity in various scenarios. We illustrate the performance of our BM by *insert throughput*, *query throughput*, *AAE*, *ARE*. In the following content, we use the abbreviation of each algorithm to call them (i.e., CM, AS, DHS, PCU, BM, EL, NI, SAL). For the correspondence between the abbreviation and the original name, please refer to § V-B1.

1) Common Dataset - CAIDA and Campus:

This part is to verify the effectiveness of BM when data pressure is low. Fig. 12 and 13 show the experimental results under the CAIDA and Campus dataset. The memory range we selected is $0.1 \sim 1.0$MB, and in order to better see the trend of each algorithm, we took two extreme points on both sides of this range: 0.01MB and 10MB. At the same time, we also give the counter distribution in this memory range, so that the reader can compare it with Fig. 5.

1) For throughput: Fig. 12(a), 12(b), 13(a), and 13(b) depict insert and query speeds. Despite high insertion speeds ($>$ 100Mps) due to a low sampling rate ($p = 0.01$), NI's accuracy is compromised, thus it's excluded from insert throughput analysis. BM, PCU, EL, and DHS exhibit top insertion rates as they calculate fewer hash functions compared to others like CM, AS, and SAL, which need four times more. SAL is slower than CM due to extra decoding, and AS is the least efficient due to frequent data exchange between its filter and sketch.

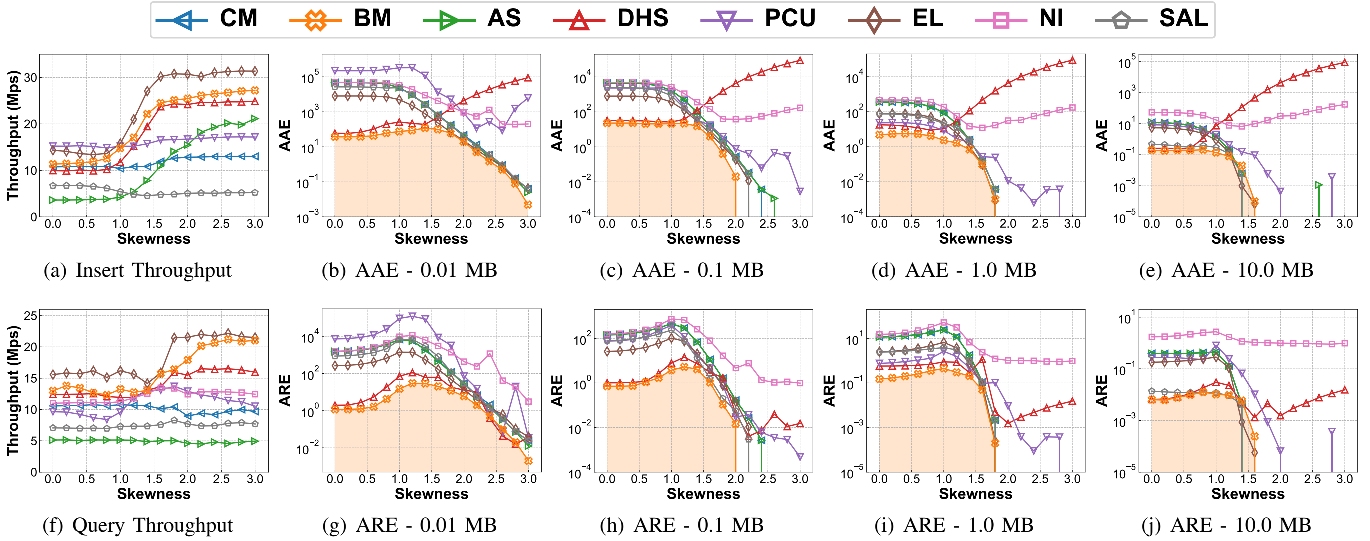


Fig. 15: Frequency estimation - Skewed dataset - Zipf.

For query speed, BM ranks second with DHS and PCU, all surpassed by EL. As memory usage grows, the throughput for all sketches declines slightly due to increased cache misses and memory access delays.

2) For error: Fig. 12(c), 12(d), 13(c), and 13(d) compare the AAE and ARE across algorithms. With less than 1MB of memory, BM's error is comparable to, or slightly better than, DHS, and it outperforms all other algorithms by more than half an order of magnitude. However, with 10MB of memory, BM and DHS fall behind EL and SAL. This is because BM and DHS use 8-bit fingerprints, which are less accurate with more memory, while EL's larger memory accommodates complete item IDs and SAL's structure allows for lower error rates, making them more accurate in high memory scenarios. Nonetheless, such high memory conditions are rare in real-world datasets like CAIDA and Campus. Additionally, NI's error grows with more memory, confirming that its high insertion throughput comes at the cost of accuracy.

In summary, BM delivers relatively high throughput and better accuracy than DHS and other algorithms, particularly noticeable when handling smaller datasets.

2) Large Dataset - IMC:

This subsection is to verify the effectiveness of BM under high data pressure. Fig. 14 shows the experimental results under the IMC dataset. The memory range we choose is $1 \sim 10$MB since the number of item types of IMC is about 10 times that of CAIDA. Similarly, we tested 0.01MB and 0.1MB extreme cases (we did not choose 100MB because it is too large and will not be used in reality).

1) For throughput: Fig. 14(a) and 14(b) reveal that BM generally has top-tier insertion and querying speeds, often surpassing DHS. With over 4MB of memory, BM and PCU share the highest query throughput.

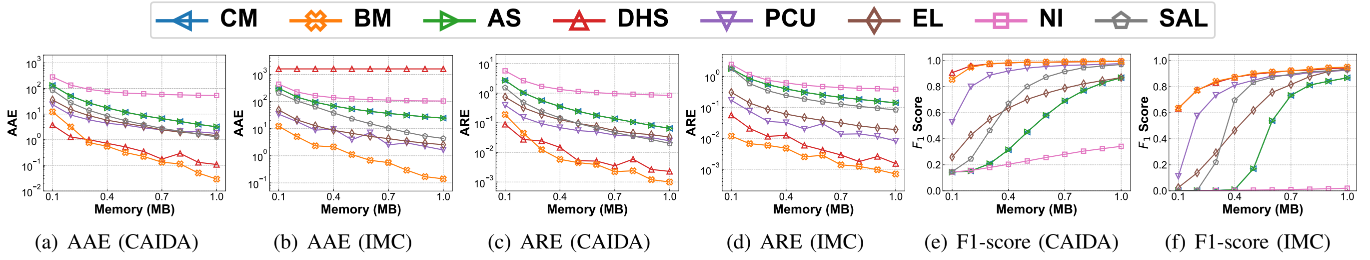
2) For error: In high data volume scenarios, BM significantly outperforms other algorithms in terms of AAE and ARE, as indicated by Figures 14(c) and 14(d). For normal memory ranges ($1 \sim 10$MB), BM's AAE is more than 10x lower than other algorithms, and nearly 100x lower than DHS. This superior performance is due to BM's ability to better handle frequent items, which DHS's lower granularity self-adjusting algorithm struggles with, leading to higher AAE.

In the ARE metric, BM maintains a slight edge over DHS and is more than 10x better than the competition. The slight difference between BM and DHS in ARE can be attributed to the nature of ARE, which is less impacted by frequent items. DHS inaccurately estimates hot items but remains accurate for cold ones. Hence, DHS's ARE is closer to BM, but its AAE is much higher. Overall, BM showcases outstanding accuracy under heavy data loads and maintains competitive throughput.

3) Skewed Dataset - Zipf:

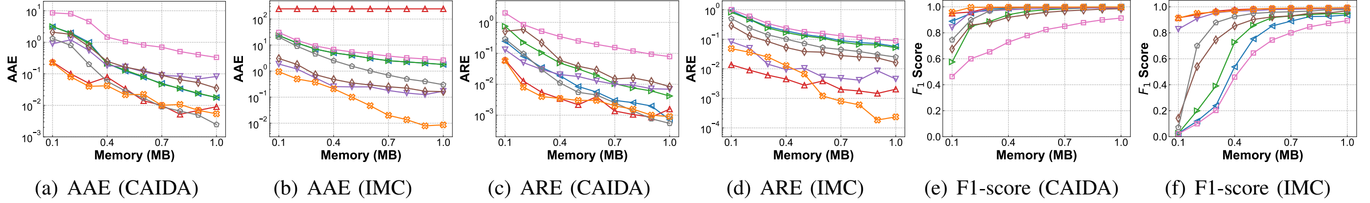
This subsection verifies BM's adaptability to data streams with different skewness. Fig. 15 shows the experimental results under the Zipf dataset. The range of skewness is $0.0 \sim 3.0$. We measure the AAE and ARE under 4 special memories. Since the trends in throughput are similar across different memory sizes, we plot their averages.

1) For throughput: Fig. 15(a) and 15(f) show the results of insert and query throughput. We can find that BM has higher insert and query throughput than DHS, ranking second, only behind EL. It is worth noting that the insertion throughput of the four algorithms BM, EL, DHS, AS increases as the skewness increases. This is because they are all self-adjusting (BM/DHS) or filter-containing (EL/AS) structures, which record the item's ID or $fingerprint$, which allows a quick memory hit when a hit item arrives without performing additional operations. Thereby increasing the speed. CM, PCU, and SAL have the same number of hashes for each item, so the skewness of the data does not affect their insertion speed. For query throughput, the results are mostly similar to insertion. It is worth noting that the query speed of AS becomes constantly low at this time. This is because the filter is traversed for each query.



(a) AAE (CAIDA) (b) AAE (IMC) (c) ARE (CAIDA) (d) ARE (IMC) (e) F1-score (CAIDA) (f) F1-score (IMC)

Fig. 16: Heavy hitter detection - Accuracy.



(a) AAE (CAIDA) (b) AAE (IMC) (c) ARE (CAIDA) (d) ARE (IMC) (e) F1-score (CAIDA) (f) F1-score (IMC)

Fig. 17: Heavy change detection - Accuracy.

2) For error: Fig. 15(b)∼15(e) and 15(g)∼15(j) shows the results of AAE and ARE. Note that in most cases, under the premise that the total number of items is the same, the higher the skewness, the smaller the estimation error of the algorithm should be. Because this means that the “hot item effect” of the data stream will be more significant, that is, fewer hot items occupy most of the frequency. Therefore, the number of item types will decrease as the skewness increases, so that the data structure of the algorithm can fully accommodate each kind of item. For example, the total number of items in the Zipf dataset used in our experiments is $32M$, the maximum item frequency of the 0.0-skewness dataset is 123K, including 1M items, and the average size of each item is 32. The maximum frequency of the 3.0-skewness dataset is as high as 18.1M, it contains only 350 items, and the average size of each item is 90K. This trend is not absolute. For example, in Fig. 15(g)∼15(i), the ARE increases when the skewness is $0.0 \sim 1.0$. This is due to some characteristics of this dataset, which we will not go into here. Note that the AAE of the DHS increases sharply when the skewness > 1.0, as it can no longer accommodate the hottest item.

We found that in the figure, the line of BM is in the bottom left of the figure in most cases, which means that it has the smallest error for the same skewness. Noting that when the skewness increases, the error of each algorithm may drop to 0, we focus on the point where the error of each algorithm drops to 0. When the memory is 0.01MB, the error of each algorithm is always greater than 0, because the space is too compact at this time. However, BM's AAE and ARE are always the best at this time. When memory is 0.1MB, BM first drops to error=0 at skewness=2.2, while {EL, SAL, CM, AS} are at {2.3, 2.4, 2.6, 2.7} respectively. When the memory is 1.0MB, the error of both BM and EL first drops to 0 at skewness=1.9, while SAL, CM, AS are all at 2.0. When the memory is 10.0MB, the error of SAL, CM, AS first drops to 0 when at skewness=1.6, while BM and EL are both at 1.7.

From Fig. 15(b)∼15(e) and 15(g)∼15(j), we can intuitively see that this point tends to move to the left as the memory increases, which is very reasonable. And different algorithms move at different speeds, BM changed from the original first place (0.1MB) to the second place (10.0MB). As we mentioned above, since the fingerprint length of EL is much larger than that of BM, the error bound of CM, AS, and SAL will be ahead of BM when the memory is large. Therefore, their progress is accelerated with the increase in memory. However, as we said earlier, in reality, we usually focus on the case where the memory is relatively compact, and the performance of BM is generally the best at this time.

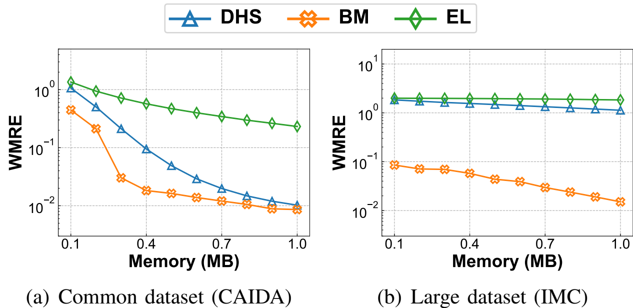
E. Heavy Hitter Detection

Fig. 16 compares the error and accuracy of BM and other algorithms in heavy hitter detection. With a memory range of $0.1 \sim 1$MB and a threshold set at 0.002% of total items, BM outperforms other algorithms. With at least 0.3MB of memory, BM's error rates are significantly lower than DHS's and are much better than those of other algorithms, achieving the highest F1-score. However, with 0.2MB or less, BM slightly falls behind DHS due to not being fully effective at lower data volumes.

For the IMC dataset, BM is far ahead in accuracy, with its error rates being notably lower than those of EL, PCU, and DHS—up to four times better compared to DHS. This advantage is especially pronounced in heavy hitter detection, where BM's design to preserve information about the most active items pays off, resulting in leading ARE and F1-scores.

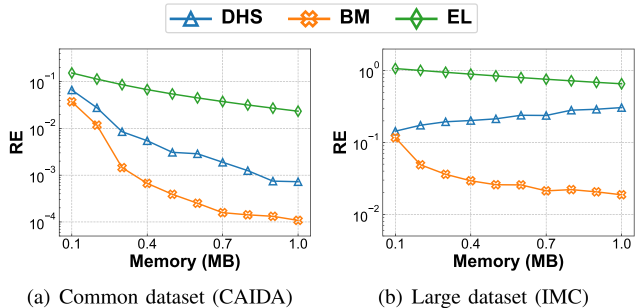
F. Heavy Change Detection

In the heavy hitter detection comparison (Fig. 17), using $0.1 \sim 1$MB of memory and a detection threshold of 0.001% of total items, BM's error rate is similar to that of DHS, SAL, and CM, placing them in the top tier. The superiority of BM is particularly evident at 0.2MB of memory in the CAIDA dataset, though its full potential isn't realized due to lower data volumes.



(a) Common dataset (CAIDA) (b) Large dataset (IMC)

Fig. 18: Item size distribution.



(a) Common dataset (CAIDA) (b) Large dataset (IMC)

Fig. 19: Entropy.

With the IMC dataset, BM's AAE is significantly lower than EL and PCU, and it's $2 \sim 4$ orders of magnitude better than DHS. However, BM's ARE is slightly worse than DHS at memory levels of 0.5MB or less because DHS can allocate more space for infrequent items by not adjusting for hot items as BM does. Above 0.6MB of memory, BM surpasses DHS and all competitors.

G. Item Size Distribution

Fig. 18 compares the WMRE of the BM algorithm with others for item size distribution, within a memory range of $0.1 \sim 1$MB. The comparison is limited to DHS and EL due to data availability from their original studies. BM performs slightly better than DHS and is notably more accurate than EL by $0.5 \sim 1.5$ orders of magnitude in the CAIDA dataset. For the IMC dataset, BM's accuracy exceeds both DHS and EL by $1 \sim 2$ orders of magnitude.

H. Entropy Estimation

Fig. 19 shows that BM has a lower RE in entropy estimation compared to DHS and EL across a memory range of $0.1 \sim 1$MB. Specifically, with at least 0.3MB of memory, BM's accuracy surpasses DHS by an order of magnitude and EL by two orders. In the IMC dataset, BM's lead is even more pronounced.

Interestingly, as memory increases, DHS's RE grows due to its entropy estimation method. With limited memory, DHS overestimates the frequency of rare items, inadvertently creating a more uniform distribution that appears to have higher entropy, which is closer to the actual value. As memory size increases, DHS's estimates become more precise, leading to less overestimation and a lower calculated entropy, thus increasing the RE.

I. FPGA Platform Integration

FPGA is a widely used commodity hardware. Table II compares the resource usage of BitMatcher on FPGA with Elastic Sketch with the same error rate for frequency estimation on CAIDA. Although BM needs more logic resources due to the algorithm complexity, BM saves 38.5% bits for RAM, and the max frequency is 18.3% higher, which means that the processing speed on the FPGA is higher for BM. The excess logics usage does not matter because the common FPGA chip [22] has more than 359K logics, and BM only takes up ˜3%. The FPGA source code of BM can refer to [36]. Table II demonstrates BM's advantage in memory utilization again and the hardware feasibility. Due to space limitations, we will give design details and continue to improve the hardware implementation in our follow-up work.

TABLE II: FPGA resource usage comparison

Algorithms	Logics	RAM	Max Frequency
Elastic Sketch	2,939	1,978,368 bits	162.6 MHz
BitMatcher	11,639	1,216,512 bits	192.3 MHz

VI. Conclusion

Data stream processing plays an essential role in various applications. To fully adapt to the high skewness of real data, in this paper, we propose BitMatcher, a sketch algorithm that can dynamically adjust the counter size in a bit-level way to accommodate different distributions while retaining high processing performance. To demonstrate this, we implement our sketch on several platforms and evaluate five typical measurement tasks. Extensive experiments show that BitMatcher can achieve excellent accuracy and maintain high speed while fully utilizing memory, demonstrating its real-world feasibility and scalability. In the future, we will improve BitMatcher in the following aspects: (1) Further optimize the algorithm on FPGA (and more hardware platforms), focusing on the integration of software and hardware; (2) Integrate into commonly used systems; (3) Use mathematical theory to analyze how well the counter adjustment strategy of BitMatcher can do.

VII. Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. This work was supported in part by the National Key Research and Development Program of China (No. 2022ZD0115303), in part by the National Natural Science Foundation of China (No. 62102203, U20A20179, 62372009, 62072430), in part by the Basic Research Enhancement Program of China (No. 2021-JCJQ-JJ-0483), in part by the Major Key Project of Peng Cheng Laboratory (No. PCL2023A06), in part by the China Postdoctoral Science Foundation (No. 2020TQ0158, 2020M682825), and in part by the International Post-Doctoral Exchange Fellowship Program of China (No. PC2021037).

REFERENCES

[1] "Our open source website," https://www.wenjunli.com/BitMatcher.

[2] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," *in ACM SIGCOMM*, 2017.

[3] M. Yu, "Network telemetry: towards a top-down approach," *in ACM SIGCOMM Computer Communication Review*, vol. 49, no. 1, pp. 11–17, 2019.

[4] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," *in USENIX NSDI*, 2013.

[5] Y. Zhou, J. Bi, T. Yang, K. Gao, J. Cao, D. Zhang, Y. Wang, and C. Zhang, "Hypersight: Towards scalable, high-coverage, and dynamic network monitoring queries," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 6, pp. 1147–1160, 2020.

[6] Ş. Gündüz and M. T. Özsu, "A web page prediction model based on click-stream tree representation of user behavior," *in ACM SIGKDD*, 2003.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "Niagaracq: A scalable continuous query system for internet databases," *in ACM SIGMOD*, 2000.

[8] Y. Izenov, A. Datta, F. Rusu, and J. H. Shin, "Compass: Online sketch-based query optimization for in-memory databases," *in ACM SIGMOD*, 2021.

[9] A. Santos, A. Bessa, F. Chirigati, C. Musco, and J. Freire, "Correlation sketches for approximate join-correlation queries," *in ACM SIGMOD*, 2021.

[10] R. Li, P. Wang, J. Zhu, J. Zhao, J. Di, X. Yang, and K. Ye, "Building fast and compact sketches for approximately multi-set multi-membership querying," *in ACM SIGMOD*, 2021.

[11] Z. Dai, A. Desai, R. Heckel, and A. Shrivastava, "Active sampling count sketch (ascs) for online sparse estimation of a trillion scale covariance matrix," *in ACM SIGMOD*, 2021.

[12] P. Jia, P. Wang, J. Zhao, S. Zhang, Y. Qi, M. Hu, C. Deng, and X. Guan, "Bidirectionally densifying lsh sketches with empty bins," *Proceedings of the 2021 International Conference on Management of Data*, 2021.

[13] Q. Shi, Y. Xu, J. Qi, W. Li, T. Yang, Y. Xu, and Y. Wang, "Cuckoo counter: Adaptive structure of counters for accurate frequency and top-k estimation," *IEEE/ACM Transactions on Networking*, 2023.

[14] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," *in ACM SIGMOD*, 2016.

[15] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.

[16] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[17] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[18] K. Cheng, L. Xiang, and M. Iwaihara, "Time-decaying bloom filters for data streams with skewed distributions," *in IEEE RIDE-SDMA*, 2005.

[19] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: A sketch framework for frequency estimation of data streams," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1442–1453, 2017.

[20] T. Yang, S. Gao, Z. Sun, Y. Wang, Y. Shen, and X. Li, "Diamond sketch: Accurate per-flow measurement for big streaming data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 12, pp. 2650–2662, 2019.

[21] J. Qi, W. Li, T. Yang, D. Li, and H. Li, "Cuckoo counter: A novel framework for accurate per-flow frequency estimation in network measurement," *in ACM/IEEE ANCS*, 2019.

[22] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," *in ACM SIGCOMM*, 2018.

[23] R. B. Basat, G. Einziger, M. Mitzenmacher, and S. Vargaftik, "Salsa: Self-adjusting lean streaming analytics," *in IEEE ICDE*, 2021.

[24] B. Zhao, X. Li, B. Tian, Z. Mei, and W. Wu, "Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing," *in ACM SIGKDD*, 2021.

[25] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," *in ACM CoNEXT*, 2014.

[26] H. Li, Q. Chen, Y. Zhang, T. Yang, and B. Cui, "Stingy sketch: a sketch framework for accurate and fast frequency estimation," *in ACM VLDB*, 2022.

[27] W. Li and P. Patras, "Tight-sketch: A high-performance sketch for heavy item-oriented data stream mining with limited memory size," *in ACM CIKM*, 2023.

[28] V. Poosala, Y. E. Ioannidis *et al.*, "Estimation of query-result distribution and its application in parallel-join load balancing," *in ACM VLDB*, 1996.

[29] Q. Huang and P. P. Lee, "Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams," *IEEE INFOCOM*, 2014.

[30] C.-H. Cheng, A. W. Fu, and Y. Zhang, "Entropy-based subspace clustering for mining numerical data," *in ACM SIGKDD*, 1999.

[31] Z. Li, F. Xiao, S. Wang, T. Pei, and J. Li, "Achievable rate maximization for cognitive hybrid satellite-terrestrial networks with af-relays," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 2, pp. 304–313, 2018.

[32] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: Robust and general sketch-based monitoring in software switches," *in ACM SIGCOMM*, 2019.

[33] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[34] R. B. Basat, X. Chen, G. Einziger, R. Friedman, and Y. Kassner, "Randomized admission policy for efficient top-k, frequency, and volume estimation," *IEEE/ACM Transactions on Networking*, vol. 27, no. 4, pp. 1432–1445, 2019.

[35] D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," *in ACM SIGMOD*, 2018.

[36] "Our open source github," https://github.com/wenjunpaper/BitMatcher.

[37] "The caida traces," http://www.caida.org/data/overview/.

[38] "Data set for imc 2010 data center measurement." https://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.

[39] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," *in ACM SIGCOMM*, 2010.

[40] D. M. Powers, "Applications and explanations of zipf's law," *in EMNLP-CoNLL*, 1998.

[41] A. Rousskov and D. Wessels, "High-performance benchmarking with web polygraph," *Software: Practice and Experience*, vol. 34, no. 2, pp. 187–211, 2004.

[42] "Hash website," http://burtleburtle.net/bob/hash/evahash.html.

[43] A. Kumar, M. Sung, J. Xu, and J. Wang, "Data streaming algorithms for efficient and accurate estimation of flow size distribution," *in ACM SIGMETRICS*, 2004.