Stingy Sketch: A Sketch Framework for Accurate and Fast Frequency Estimation

Haoyu Li* Peking University lihy@pku.edu.cn

Qizhi Chen* Peking University hzyoi@pku.edu.cn

Yixin Zhang* Peking University yxzhangg@gmail.com

Tong Yang*† Peking University yang.tong@pku.edu.cn

Bin Cui* Peking University bin.cui@pku.edu.cn

ABSTRACT

Recording the frequency of items in highly skewed data streams is a fundamental and hot problem in recent years. The literature demonstrates that *sketch* is the most promising solution. The typical metrics to measure a sketch are accuracy and speed, but existing sketches make only trade-offs between the two dimensions. Our proposed solution is a new sketch framework called Stingy sketch with two key techniques: Bit-pinching Counter Tree (**BCTree**) and Prophet Queue (**PQueue**) which optimizes both the accuracy and speed. The key idea of **BCTree** is to split a large fixed-size counter into many small nodes of a tree structure, and to use a precise encoding to perform carry-in operations with low processing overhead. The key idea of **PQueue** is to use pipelined prefetch technique to make most memory accesses happen in L2 cache without losing precision. Importantly, the two techniques are cooperative so that Stingy sketch can improve accuracy and speed simultaneously. Extensive experimental results show that Stingy sketch is up to 50% more accurate than the SOTA of accuracy-oriented sketches and is up to 33% faster than the SOTA of speed-oriented sketches.

PVLDB Reference Format:
Haoyu Li, Qizhi Chen, Yixin Zhang, Tong Yang, and Bin Cui. Stingy Sketch: A Sketch Framework for Accurate and Fast Frequency Estimation. PVLDB, 15(7): XXX-XXX, 2022.
doi:10.14778/3523210.3523220

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/StingySketch/Stingy-Sketch.

1 INTRODUCTION

1.1 Background and Motivation

Recording the frequency of items in highly skewed data streams is a fundamental and hot problem in recent years [1–3]. And it is also the basis of many applications including finding top-k items [4–7], joining tables [8, 9], multi-set querying [10], and more [11–14]. Sketch is widely acknowledged as the most promising probabilistic algorithm for frequency estimation. The classical sketch, Count-Min [15], hashes items into d counters of d arrays, and reports the minimal value as the estimation. At an expense of determinacy, sketch summarizes data streams in a compact space and $O(1)$ time, making the item processing efficient.

The typical metrics to measure a sketch are accuracy and speed. Unfortunately, there is an inherent conflict between these two metrics. On the one hand, when optimizing the accuracy, sketch often needs to introduce complex operations which consumes much time; On the other hand, when optimizing the speed, sketch often needs to reduce complex operations which in turn degrades the accuracy. *The ideal goal is to design a sketch that simultaneously reaches the highest accuracy and fastest speed.*

1.2 State of the Art and Their Limitations

We divide the existing sketches into 3 categories: the *classical*, the *accuracy-oriented*, and the *speed-oriented*. The classical sketches include Count-Min [CMS] [15], Conservative Update [CUS] [16], and Count [CS] [17], which are simple to implement but have poor accuracy and low speed. A number of improvements have emerged to address these 2 drawbacks, and they are classified as accuracy-oriented and speed-oriented algorithms respectively.

The reason for accuracy drawback is that classical sketches use fixed-size counters, which mismatch the frequency distribution in practice. It is known that item frequency often obeys a highly skewed distribution [18–20]: Almost all items in a data stream appear only once or a couple of times, while very few items appear kilos or even millions of times. In other words, if we use only 32-bit counters, most of the significant bits are wasted. The state of the art (SOTA) of accuracy-oriented sketches is Self-Adjusting Lean Streaming Analytics [SALSA] [3] which uses small counters at first, and merges adjacent counters when they overflow. But in turn, SALSA needs an additional bitmap and complex operations to indicate the overflowing counters, making the speed very slow. Further, SALSA is a flattened (rather than hierarchical) structure and thus there is still room for accuracy improvement.

The reason for speed drawback is that classical sketches do not exploit the L2 cache acceleration function. In general, the classical sketches need to access d counters totally randomly and such randomness is cache-unfriendly. The ideal goal is to always keep all counters in L2 cache [21]. But it seems unrealistic when the sketch size is too large. To solve this problem, the mainstream solution is to utilize temporal or spatial locality. The Augment sketch [22] is the SOTA of speed-oriented work which utilizes temporal locality. It creates a small additional filter to store hot items which can be fully kept in L2 cache. The Pyramid sketch [19] is the SOTA

---

*School of Computer Science, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University.
†Peng Cheng Laboratory, Shenzhen, China.
Haoyu Li and Qizhi Chen are co-first authors, and the corresponding author is Tong Yang. This work is supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, National Natural Science Foundation of China (NSFC) (No. U20A20179, 61832001).

Proceedings of the VLDB Endowment, Vol. 15, No. 7 ISSN 2150-8097.
doi:10.14778/3523210.3523220

of speed-oriented work which utilizes spatial locality. It forces all d mapped counters to converge in one word so that they can be fetched in one sitting and uses one hashing technique to reduce the hash computation overhead. Although Augment and Pyramid can speed up the counting process to some extends, both of them have significant limitations: (1) Augment cannot deal with cold items. For each incoming cold item, the traversal of the small filter is totally an extra work. (2) Pyramid costs much precision. Once two items are mapped into the same word, this method may lead to severe hash collision. Further, Augment and Pyramid both need complex operations, and thus there is room for speed improvement.

In brief, classical sketches have two serious drawbacks on accuracy and speed. Existing sketches make only trade-offs, but cannot effectively optimize both of the 2 dimensions. **Our goal is to propose a new sketch framework which performs more accurate than accuracy-oriented work SALSA, as well as faster than speed-oriented works (Augment and Pyramid).**

1.3 Our Proposed Solution

Towards the ideal goal, we propose a new sketch framework called Stingy sketch. The Stingy sketch is a completely stingy guy who budgets every penny of computing resources. It consists of two cooperative techniques: Bit-pinching Counter Tree (**BCTree**) and Prophet Queue (**PQueue**). **BCTree** makes most bits to count with low processing overhead, while **PQueue** makes most memory accesses finish in L2 cache without losing precision. Unlike other trade-off works, the Stingy sketch miserly combines the accuracy and speed techniques, achieving high precision and throughput simultaneously. Further, Stingy sketch is a generic and fundamental sketch framework which can extend to many popular sketches such as CMS, CUS, and CS (We call them S_{CM}, S_{CU}, and S_C). Next we briefly introduce the key techniques of the Stingy sketch.

• **Key Technique I: Bit-pinching Counter Tree (BCTree, Section 3.1).** In a nutshell, **BCTree** makes most bits to count by using a well encoded tree-structure, achieving higher accuracy than the SOTA of accuracy-oriented work - SALSA. SALSA uses a flatted structure, and intuitively hierarchical structure has higher potential than flattened structure to achieve memory efficiency. In our tree-structure, each node has very few number of bits, such as 2 bits and 6 bits. To insert an item, we first map the item to the leaf node (6-bit counter), and if it overflows, we will perform carry-in operation to its parent node (2-bit counter). In **BCTree**, we have the following two key designs. First, we manage to minimize the number of memory accesses for each insertion/query when carry-in operations happen. In SALSA and Pyramid, once a carry-in operation happens, one additional memory access is inevitable. Differently, in **BCTree**, the child nodes are arranged near their parents, and thus they can be read in one memory access. To achieve this, we physically organize the counters in an in-order traversal of the binary tree (see details in **Section 3.1.1**). In this way, given a frequent item overflowing 4 times to the 4th level, we need only 1 memory accesses rather than 4 memory accesses. Second, although the flags to indicate whether overflows happen are inevitable, these flags do not contribute to the accuracy. Therefore, we manage to minimize the memory usage of such flags. In the tree structure - Pyramid, it uses around 25% memory for flags. In contrast, we only use around 5.5% memory for flags (see details in **Section 3.1.2**). In other words, 94.5% bits are used for counting in BCTree.

• **Key Technique II: Prophet Queue (PQueue, Section 3.2).** *In a nutshell,* ***PQueue*** *foreknows the coming items like a prophet and prefetches their addresses into L2 cache.* Using **PQueue**, we can make most memory accesses happen in L2 cache, though the size of Stingy sketch may be much larger. Traditional insertion design updates the incoming counters instantly. CPU has to wait until the addresses of counters are fetched from L3 cache or even memory. Our design seems like late update: Once a counter is required to be updated, We only prefetch its address instantly, but update its value after a short period. Specifically, **PQueue** is a variable-length queue structure. It enqueues the counter's address in the current insertion, and dequeues it to update counters after a short period.

We make 3 further contributions. First, we propose **Bounded Hash Split (BHS, Section 3.3)** to reduce the hash computation cost. It computes only one hash function and split it into one index and d offset parts to find d mapped counters. Compared with word acceleration technique, **BHS** uses variable-length offset parts and provides a tight error bound. Second, we prove the unbiasedness[1] of S_C (the Stingy sketch extended to the Count sketch), so Stingy sketch can be applied to extensive tasks like unbiased top-k detection. Third, we conduct comprehensive C++ simulation experiments on multiple datasets, and deploy Stingy sketch on top of a modern stream processing framework, Apache Flink [23], to show its performance in distributed environment.

1.4 Key Contributions

• We propose the Stingy sketch, a new sketch framework that achieves high accuracy, high speed and unbiasedness.
• We give detailed mathematical analyses of unbiasedness, time complexity and error bound of our techniques.
• We conduct simulation experiments on frequency estimation and unbiased top-k detection tasks and deploy Stingy sketch on top of Apache Flink framework. In frequency estimation task, the Stingy sketch is up to 50% more accurate than SOTA accuracy-oriented sketch SALSA and 123% faster than speed-oriented sketch Pyramid. In unbiased top-k detection task, Stingy sketch achieves up to 19 times more accurate than SOTA algorithm USS [5]. The integration into Apache Flink shows that Stingy sketch also works well on modern distributed stream processing framework.

2 RELATED WORK

2.1 Preliminaries

• **Data Stream:** A data stream $\mathcal{S}$ 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, but the algorithm should process items in order to support online query. A more formal definition of the data stream is a sequence of N pairs $\langle(e_1, w_1), (e_2, w_2), ..., (e_N, w_N)\rangle$, where w_i represents weights of e_i. We only consider the first definition because some comparison algorithms don't support the second one.
• **Frequency Estimation:** Given a data stream $\mathcal{S} = \langle e_1, e_2, ..., e_N\rangle$, we use an algorithm $\hat{f}(e)(\forall e \in E)$ to measure $f(e) := \sum_i \mathcal{I}\{e = e_i\}$.

[1]Given an item e, we say the estimation $\hat{f}(e)$ is unbiased if and only if $\mathbb{E}\hat{f}(e) = \mathbb{E}f(e)$.

Of course, we don't know e until we finish processing $\mathcal{S}$. So we should record the frequency of all items during the procedure.
• **Unbiased Top-k Detection:** Given a data stream $\mathcal{S}$ and an integer k, we want to record the most frequent k items with its frequency. If an algorithm not only records the top-k items, but also reports the unbiased frequency [i.e. $\mathbb{E}\hat{f}(\cdot) = \mathbb{E}f(\cdot)$], we say the algorithm detects unbiased top-k items.

2.2 Related Sketches

The sketch is widely used in many domains, including real-time IP traffic [15–17, 21], natural language processing (NLP) [24], graph stream [25], sensor database [26], and more [27, 28]. For frequency estimation problem, classic sketches are Count-Min sketch [CMS] [15], Conservative Update sketch [CUS] [16] and Count sketch [CS] [17]. CMS consists of d arrays $A_0[], A_1[], ..., A_{d-1}[]$. Every array is L fixed-size counters. For each item e, CMS picks one counter per array by independent hash functions $h_i(\cdot)$. When counting frequency, CMS increases all mapped counters $A_i[h_i(e)]$ by 1; When querying frequency, CMS reports the min value of all mapped counters. CMS has zero underestimate rate, low overestimate rate, and keeps the optimal theoretical result: Within $O(1/\epsilon \times \log(1/\delta))$ space, CMS ensures $\mathbb{P}\left(|\hat{f}(e) - f(e)| \leqslant \epsilon N\right) \geqslant 1 - \delta$. CMS has more accurate variants such as Conservative Update sketch [CUS] [16], but it loses the ability to delete. Another classic sketch is CS. It increases the mapped counters by +1 or -1 with equal probability. When querying frequency, CS reports the mean or median value of all mapped counters. In this way, $\hat{f}(\cdot)$ is proved to be the unbiased estimation of $f(\cdot)$. Within $O(1/\epsilon^2 \times \log(1/\delta))$ space, CS ensures $\mathbb{P}\left(|\hat{f}(e) - f(e)| \leqslant \epsilon \sum_{e \in E} f^2(e)\right) \geqslant 1-\delta$. **With an additional heap to record hot items, CS can also be used to find unbiased top-k items.**

Most existing works are optimizations from CMS or CS. But they only make trade-offs among space, accuracy and speed. In some scenarios, sketches are designed to achieve high throughput (e.g. Nitro sketch [29], Morton Filter [30], Cache Assisted Randomized Sharing Counters [31], and Additive-Error Counters [2]) at cost of accuracy. While in some other scenarios, sketches would rather cost time in exchange for accuracy or compact space (e.g. Counter Braids [32], Counter Tree [33], Hyper Log Log [34], and Diamond sketch [20]). We introduce some typical and most related sketches in the following parts and use them as comparison algorithms in **Section 5**.
• **Self-Adjusting Lean Streaming Analytics [SALSA] [3].** SALSA is a typical accuracy-oriented sketch framework which is a flattened and simplified version of ABC sketch [35]. Initially, SALSA uses only one 8-bit *char* rather than 32-bit *int* to count frequency[2]. It establishes an extra bitmap to tag overflowing counters, and merges small neighboring counters to dynamically form a bigger one. SALSA achieves high accuracy but its shortcomings is also obvious: The extra bitmap is not only space consuming, but also decreases the speed. That's because SALSA always looks up the bitmap at every operation, to check if the corresponding counter merges with others.

[2]In fact, SALSA has other versions such as the 2-, the 4-, and the 16-bit ones, but the 8-bit version is recommended by the authors.

• **Augment Sketch [AS] [22] and Pyramid Sketch [PS] [19].** Augment and Pyramid are typical speed-oriented works. Augment adds a pre-filtering stage when inserting items. The filter can be totally loaded in L2 cache, so a hot item can be sought and updated efficiently. However, for cold items, this sought process is just a waste of time. Pyramid has two main contributions: Counter sharing and word acceleration. Counter sharing is similar to Counter Tree but uses 2 flag bits per counter to accelerate query process. Word acceleration forces counters map in the same word to utilize spatial locality. Further, because these counters locate near, Pyramid can use only one hash function to find d mapped counters. On the one hand, Pyramid is much faster than existing works. But on the other hand, the extra flag bits make 3/4 states of a counter be sentinels, and the word acceleration technique causes serious collisions. Both of them greatly limit the accuracy. Further, the cumbersome encoding also leaves room for speed.
• **Self-adaptive Counters [SAC] [36].** SAC is a trade-off between accuracy-oriented and speed-oriented sketches. It designs a new counter that switches between normalized and denormalized numbers and updates them with a certain probability. SAC is less accurate and faster than SALSA, but is more accurate and slower than Pyramid.
• **Unbiased Space Saving [USS] [5] and Waving Sketch [WS] [6].** USS and WS are typical unbiased top-k detection algorithms. USS is an extension of Space Saving [37]. It guesses top-k items in advance and organizes them in a minimum heap $\mathcal{H}$. When the coming item e is an element of $\mathcal{H}$, USS increases its frequency by 1. Otherwise, USS increases the frequency of the top element f_{min} by 1 and exchange e_{min} by e with probability $1/(f_{min} + 1)$. WS is another SOTA work on finding unbiased top-k items. It identifies hot items like traditional CS+Heap (Count sketch + Minheap). But it uses a more delicate heavy part to record the top-k items.

3 THE STINGY SKETCH FRAMEWORK

In this section, we briefly introduce the data structure of the Stingy sketch. For convenience, we use S_C (the Stingy sketch extended to Count sketch) as an example.

3.1 Bit-pinching Counter Tree (BCTree)

BCTree is a specialized technique to make full use of every bit of a sketch. In this subsection, we implement **BCTree** in three steps. In the first step, we give an explicit carry direction if a counter overflows; In the second step, we demonstrate how to deal with multilayered carry chains and query a counter's value; And in the last step, we "kick" the conflict counters off as a further optimization, by which we can ultimately report an unbiased estimation. We call the three steps as **Inlay Carry Mode**, **Counting State Machine**, and **Open Addressing Method** respectively.

3.1.1 *Inlay Carry Mode (ICM) (Fig. 1).*

The idea of **ICM** is derived from the in-order traversal of the binary tree. Unlike the existing works (e.g. Pyramid), **ICM** ensures a counter overflow to a relative near address, so we can fetch all counters together into L2 cache. As preliminaries, we suppose every array has L bytes and every byte consists of two counters: The least significant 6 bits form a 0-level counter and the most significant 2 bits form a nonzero-level counter. *The key contribution of* ***ICM*** *is,* *once a k-level counter x_k^i ($k \geqslant 0$) overflows, it carries to a (k+1)-level counter x_{k+1}^i whose address is*

$$x_{k+1}^i := \begin{cases} x_k^i|1, & k=0; \\ (x_k^i|(2\times b))\oplus b, & k>0. \end{cases} \quad \text{where } b := x_k^i \& (-x_k^i). \quad (1)$$

In this formula, "|" represents **OR**, "⊕" represents **XOR** and &" represents **AND**.

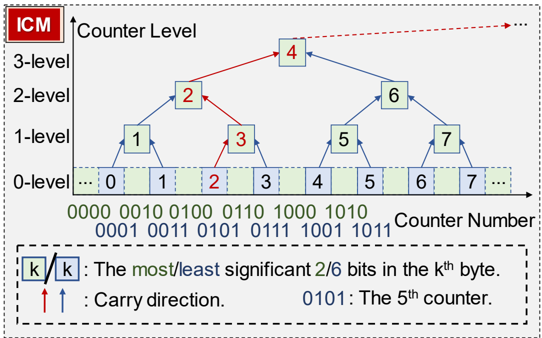

Figure 1: Inlay Carry Mode (ICM).

3.1.2 *Counting State Machine (CSM) (Fig. 2).*

***CSM** uses Finite State Machine (FSM) to describe the counting process.* Because every counter of S_C is either 6 bits (0-level counter) or 2 bits (nonzero-level counter), we use 2 kinds of FSM (FSM_0 and FSM_1) to describe them. To be more specific, FSM_0 represent the true form of a 0-level counter which has $2^6 = 64$ states $\pm 0, \pm 1, ..., \pm 31$. Except the state "-0" is used as Kick Tag[3], the states transfer from one to another according to **Fig. 2**. For instance, if the initial state of FSM_0 is "+31", it will overflow to a 1-level counter and turns to state "+1" after an insertion. Similarly, we use FSM_1 to represent a nonzero-level counter in **Fig. 2**. FSM_1 has exactly 4 states 0, 1, 2, 3. For state 0, 1, 2, their successor is naturally 1, 2, 3; And for state 3, its successor is 1 rather than 0. In this way, a nonzero-level counter can not transfer to state 0 once it is inserted.

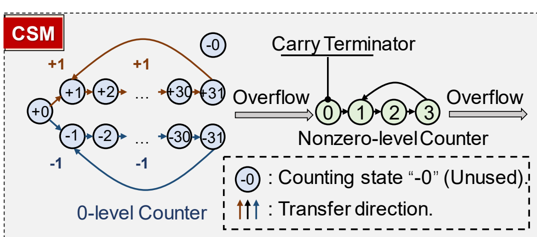

Figure 2: Counting State Machine (CSM).

CSM helps to terminate the query process: Since the query process is bottom-up, if we meet state "0", we can simply stop the query process because the counter is never carried. For instance, if we inductively consider a carry chain including k counters $(-1)^S \times v_0, v_1, v_2, ..., v_{k-1}(v_{k-1} = 0)$, we can quickly calculate its value by Horner scheme:

$$V := (-1)^S \times (v_0 + \sum_{i=1}^{k-2} 3^{i-1} \times 31 v_i) = (-1)^S \times (v_0 + 31V_1), \quad (2)$$

$$\text{where } V_j := \begin{cases} v_j + 3\times V_{j+1}, & j = 1,2,...k-2; \\ v_j = 0, & j = k-1. \end{cases}$$

So we call a nonzero-level counter in state 0 as a carry terminator (We show a more explicit example in **Section 3.4**). For S_{CM}, S_{CU}, FSM_0 doesn't need a sign bit thus $V = v_0 + 62V_1$.

3.1.3 *Open Addressing Method (OAM) (Fig. 3).*

OAM further reduce the error and ensure the unbiasedness of S_C[4]. From **Section 3.1.1** we find **ICM** divides the estimation error into 2 kinds: One is caused by hash collision, which is a common error of all sketches; The other is caused by carry conflict, which means both of two child nodes overflows to the same parent node. Carry conflict error prevents S_C giving an unbiased estimation. To address this problem, we borrow a concept from the hash table and propose a technique called **OAM**.

*The main idea of **OAM** is, if two carry chains overflow to the same parent node, we kick the smaller one to another place.* We use the reserved state "-0" to tag a kicked counter (That's why we name it Kick Tag), and simply use $p(x)$ to represent the kicked place[5] of the original carry chain x. That is, if x is kicked to $p(x)$, we tag chain x with "-0" and use the sum value of x and $p(x)$ to replace the original value of $p(x)$. If carry chain $p(x)$ overflows as well, we adopt the following steps to handle carry conflict (**Fig. 3**): (a) If the overflow doesn't cause a conflict error, we allow it to overflow and terminate the kick process; (b) If it overflows to a counter, but this counter has a nonempty parent, we kick $p(x)$ to $p(p(x))$; (c) If it overflows to a carry terminator (Recall a carry terminator is a nonzero-level counter in state 0, see **Section 3.1.2**) of another item y, we kick y to $p(y)$ and start a new kick process. Based on the 3 steps, we can completely avoid carry conflict. It seems time consuming because we may kick 2^k derived nodes from a k-level carry terminator in the worse case. But in fact, our worry is unwarranted: According to **OAM**, every node has at most one nonempty child in **BCTree**. Therefore, the branch of a k-level counter has only $(k+1)$ nodes and the kick process doesn't cost much time. In **Theorem 4.4**, we prove the kick process is a small probability event.

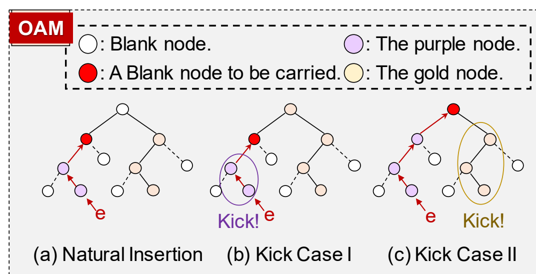

Figure 3: Open Addressing Method (OAM).

3.2 Prophet Queue (PQueue)

PQueue significantly reduces L3 cache misses during the operations. In general, the sketch size often exceeds the capacity of L2, which leads to severe cache misses. A common sense solution is to

[3]We explain the Kick Tag in **Section 3.1.3**.

[4]For simplicity, we do not apply **OAM** to biased algorithms S_{CM} and S_{CU}.

[5]We use linear probing $p(x)$=x+31 in our experiment. But other settings such as quadratic probing are also recommended.

utilize locality. But in this subsection, We take a different approach. *That is, **PQueue** makes L2 cache prefetch all addresses before they are really updated.*

The data structure of **PQueue** is a queue that contains Z slots, where Z can be dynamically adjusted based on the data stream speed. We describe PQueue in two steps: (a) Given a series counters $c_1, c_2, ..., c_n$ to update, we first prefetch the coming counter's address a_i, then enqueue a_i into **PQueue** but do not really modify c_i. Instead, we dequeue the tail address a_{i-Z+1} and update the tail counter c_{i-Z+1} (We do nothing when $i \leqslant Z$). We repeat above process for Z times, thus **PQueue** has enqueued and dequeued for Z times. (b) After Z such operations, c_i has already been fetched in L2 and a_i has been moved to **PQueue**'s tail. Then can simply dequeue a_i and modify c_i without worrying about L3 cache misses.

PQueue consumes only 0.2KB memory (when $Z = 16$) but explores the cache ability. As a price, we need to flush all items out of **PQueue** before querying an item. This latency is a drawback when Z is getting larger. Let t be the average insertion time thus Zt be the expected latency per insertion. We further want to dynamically change Z based on data stream speed to minimize $u(t, Z) := (1 + \gamma Z)t$, where $\gamma \approx 0.001$ is a hyper-parameter. To address this problem, we divide all items into windows of size n, and periodically measure t to calculate $u(t, Z)$ per n continuous insertions. When a new window comes, we either *increase* or *decrease* Z by one[6] ($Z = 1$ at the beginning): When $u(t, Z)$ increases at this window, we *follow* the operation of the previous window; Else we do the *opposite* operation. In this way, we can automatically adjust the queue size as the data stream speed changes.

3.3 Bounded Hash Split (BHS)

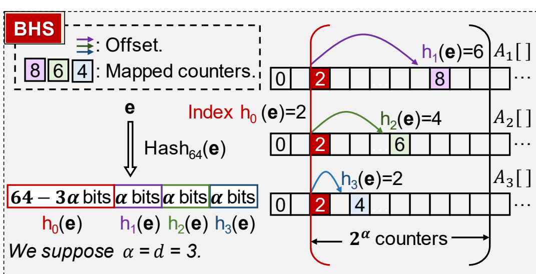

Figure 4: Bounded Hash Split (BHS).

BHS is a useful technique to save hash computation overhead without losing much accuracy. We assume the sketch has d arrays and every array has L bytes. Originally, given an item e, we need d hash functions to calculate the counters $A_i[x_0^i](i \in [0..d-1], x_0^i \in [0..(L-1)])$. This calculation needs $d\lfloor \log L \rfloor$ hash bits. **BHS** saves hash bits by dividing one hash value into one index part $h_0(e)$ and d offset parts $h_1(e) \sim h_d(e)$ (**Fig. 4**). At every operation, we call the hash function only once, and use formula

$$x_0^i := h_0(e) + h_i(e) \quad (mod\ L)$$

to calculate the address of x_0^i. In other words, the d addresses have the same index but have their own offsets. In particular, when $\alpha = \lceil \log L \rceil$, this optimization has no difference from the original method; when $\alpha = 0$, the d array is exactly one array copied d times.

[6]Specially, we can not decrease Z when $Z = 0$.

Although **BHS** resembles the word acceleration technique in Pyramid, they still have differences. Word acceleration aims at reducing the average number of memory accesses, so it strictly limits all mapped counters in one word. As a by-product, it saves the number of hash bits from $d\lceil \log L \rceil$ to $(\lceil \log dL \rceil + (d-1)\log W)$[7] and uses one hashing technique to accelerate. However, such a small number of hash bits inevitably leads to severe hash collisions. Differently, ***BHS** is a simple technique that reduces the number of hash bits from $\lceil \log L \rceil d$ to $(\lceil \log L \rceil + (d-1)\alpha)$. We can prove that an appropriate α (e.g. $\alpha = 8$) of **BHS** only brings a slight extra error rate (**Theorem 4.5**) which can be almost ignored.* **BHS** do not reduce the number of memory accesses, but with the help of **PQueue**, Stingy sketch can be even faster than the cumbersome Pyramid.

3.4 Operations and Example

In this subsection, we show the **insertion** and **query** operations of the Stingy sketch. We omit the **deletion** operation because it is only the inverse process of insertion.

Insertion: For each item e, we first use Hash(e) to map it to d counters. Then we enqueue their address and prefetch them into L2 cache. At the same time, we update the tail counter of **PQueue** and update the old counter. If the carry chain x is kicked away, we search $p(x), p(p(x))$... until we find a counter that isn't in the Kick Tag state. Then we immigrate the carry chain to the new counter and terminate the insertion.

Query: To query an item e, we first use Hash(e) to map it to d 0-level counters. For each counter x, If it is kicked away, we search $p(x), p(p(x))$... until we find a counter that is not in the Kick Tag state. Next we call *Calculate(x)* to trace all ancestor counters until the Carry Terminator calculate its value. Finally we compare the values of its d counters and report the mean or median value as its unbiased estimation. Note that we should flush **PQueue** to make sure all items are inserted before a formal query.

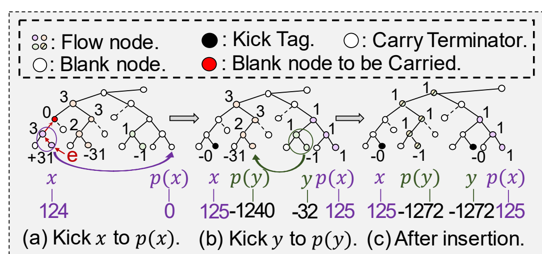

Figure 5: Basic Operations of Stingy Sketch Framework.

Example: Here we suppose $d = 1$ and give a simple example of Stingy Sketch Framework (**Fig. 5**). To start with, there are 3 distinct kinds of items marked as purple, orange and green respectively. Based on **Formula 2**, the purple items has frequency $V_P = (-1)^0 \times (31 + 31 \times 3) = 124$, the orange items has frequency $V_O = (-1)^1 \times (31 + 31 \times (2 + 3 \times (3 + 3 \times 3))) = -1209$, and the green items has frequency $V_G = (-1)^1 \times (1 + 31) = -32$. We demonstrate how the

[7]In Pyramid, all mapped counters are in one rather than d arrays, so it uses $(\lceil \log dL \rceil - \log W)$ hash bits to identify a word. W is the number of counters in a word whose value is about 16 on many of today's CPUs.

Algorithm 1: Insertion for S_C

Input: Item e.
Function Insert (e):
Address[0..d) $\leftarrow$ Hash(e);
for i in $[0..d)$ **do**
  Prefetch Address[i] and put Address[i] in **PQueue**;
  $x \leftarrow$ the tail counter of **PQueue**;
  $x \leftarrow p(x)$ **while** x is in state "-0";
  Increase x by $(-1)^{\text{Sign}}$; //Sign $\leftarrow$ Hash$(x, e) \in \{0, 1\}$
  if Abs(x) changes from T to 0 **then**
    $y \leftarrow x's$ highest ancestor counter;
    if $y's$ grandfather $\neq 0$ (We say *the carry chain cannot hold this value*): Kick(x); //Kick Case I
    else if ($y_s \leftarrow y's$ sibling) $\neq 0$ **then**
      while y_s is not a 0-level counter **do**
        $y_s \leftarrow$ the nonempty child of y_s;
      Kick(y_s); //Kick Case II
    else $x \leftarrow (-1)^{\text{Sign}}$ **and** Carry ($x_p \leftarrow x's$ parent, +1);
  if Abs(x) changes from 1 to 0 **then**
    if ($x_p \leftarrow x's$ parent) $\neq 0$ **then**
      $x \leftarrow (-1)^{\text{Sign}}T$ **and** Carry ($x_p \leftarrow x's$ parent, -1);
    else Set x to 0;
Function Carry (x, v): //$v \in \{+1, -1\}$
Increase x by v **and** $x_p \leftarrow x's$ parent;
if x changes from τ to 0: Set x to 1 and Carry (x_p, v);
if x changes from 1 to 0: **then**
  if $x_p \neq$ **0:** Set x to τ **and** Carry (x_p, v) **else** Set x to 0;
Function Kick (x):
$C \leftarrow$ Calculate(x) + 1 + $(-1)^{\text{Sign}}T$; //Calculate is in **Alg. 2**
while This carry chain cannot hold C items **do**
  Set x to "-0" **and** set all father counters of x to "0";
  $x \leftarrow p(x)$ **and** $C \leftarrow C$ + Calculate(x); //Kick Case I
$\forall$ 0-level node y s.t. the carry chain of y shares counters with the carry chain of x: Kick(y); //Kick Case II
Set the carry chain of x to C;

Algorithm 2: Query for S_C

Input: Item e.
Output: Query result Q_e
Function Query (e):
$Q_e \leftarrow +\infty$;
Address[0..d) $\leftarrow$ Hash(e);
for i in $[0..d)$ **do**
  $x \leftarrow$ the counter at Address[i];
  $x \leftarrow p(x)$ **while** x is in state "-0";
  $Q_e \leftarrow \min\{Q_e, \text{Abs(Calculate}(x))\}$; //Absolute value
return Q_e;
Function Calculate(x): //return a signed integer
$v \leftarrow$ the value of x;
if $v = \pm 0$: **return** 0;

$$\gamma \leftarrow \begin{cases} T = 2^5 - 1, & x \text{ is a 0-level counter and } v > 0; \\ -T, & x \text{ is a 0-level counter and } v < 0; \\ \tau = 3, & x \text{ is a nonzero-level counter.} \end{cases}$$

return $v + \gamma$ Calculate ($x_p \leftarrow x's$ parent);

Stingy sketch react to the insertion of purple item e. (a) First of all, we find the two purple nodes are full thus the insertion overflows to the red carry terminator. Unfortunately, the red node's parent has already been taken by the orange items. To eliminate the carry conflict, we set the counter to state "-0" and kick the whole purple branch to $p(x)$. (b) However, this kick also leads to another carry conflict because the green's carry terminator is taken. So we kick the green items to $p(y)$ and merge the orange and green items. (c) Finally, the mixed items overflows to a 4-level counter and causes no more carry conflicts. So we can end the kicking process and the query results ultimately become $V'_P = (-1)^0 \times (1 + 31 \times (1 + 3)) = 125$, $V'_O = V'_G = (-1)^1 \times (1 + 31 \times (1 + 3 \times (1 + 3 \times (1 + 3)))) = -1241$.

4 MATHEMATICAL ANALYSIS

In general, the Stingy sketch can extend to many popular sketches such as CMS, CUS and CS. But for convenience, we only conduct theoretical analysis on S_C in this section. The only one exception is **Theorem 4.5**, since **BHS** can be directly applied to CMS. For space constraints, we only list the conclusions in this section, but leave detailed proofs in a technical report on GitHub [38].

Theorem 4.1. *S_C reports the unbiased frequency estimation. In other words, $\forall e \in E$, we have $\mathbb{E}\hat{f}(e) = \mathbb{E}f(e)$.*

Proof. See Appendix A.1 in the technical report. □

Theorem 4.1 shows the unbiasedness of S_C, so it can be applied to more tasks such as unbiased top-k detection. In following parts, we analysis the speed and accuracy of S_C.

According to **ICM**, every counter consists of 6 bits (0-level counter) or 2 bits (nonzero-level counter). So the capacities of the two kind are $T := 2^5 - 1$ (-31 ~ +31) and $\tau := 2^2 - 1$ respectively. In this way, an item with value $C\,(C > T)$ occupies $\lceil \log_\tau (C\tau/T) \rceil$ counters. So increasing items seems to cost much time. However, **Theorem 4.2** shows that although we need to traverse all $\lceil \log_\tau (C\tau/T) \rceil$ counters in the worse case, the amortized cost of inserting an item is no more than 1.05 counters.

Theorem 4.2. *If a natural insertion (**Fig. 3 (a)**) costs* 1 *dollar, the price of inserting N items won't exceed* $1.05N$ *dollars.*

Proof. See Appendix A.2 in the technical report. □

Remark. In one carry chain, 0~7-level counters are in the same 8-byte word. We can similarly prove (1) Less than 0.12% insertions cause extra memory accesses. (2) As long as $\log L \in \mathbb{N}$ and $|\mathcal{S}| < T\tau^{\log L} (= 1.1 \times 10^{11}$ when $L = 1\text{MB})$, Stingy sketch won't lead to an out-of-range error.

Next we analysis the accuracy of S_C. Initially, we provide the number of carry chains of S_C is more than the number of counters of original CS.

Theorem 4.3. *When $|\mathcal{S}| \leqslant TL$, the number of carry chains of S_C is larger than the number of counters of CS under the same memory cost.*

Proof. See Appendix A.3 in the technical report. □

Remark. Under certain conditions, we can prove S_{CM}, S_{CU} are also better than original sketches in terms of accuracy. Thus we can simply take their theoretical error bounds in **Section 2.2** as ours.

Then we give a more precise result of kick rate. Suppose N distinct kinds of items are independent identically distributed, we have a more precise upper bound:

THEOREM 4.4. *When $TL/\mathbb{E}|\mathcal{S}| > 3$, the kick rate $K < \frac{\mathbb{E}|\mathcal{S}|}{LT-\mathbb{E}|\mathcal{S}|}$.*

PROOF. See Appendix A.4 in the technical report. □

REMARK. In reality, the data stream is highly skewed and $|\mathcal{S}|$ is usually less than 5 L ($K < 9\%$). So the kicking process is a really small probability event.

Finally we estimate the error rate of **BHS** optimization. Error rate is defined as the not correctly estimated items proportion, i.e. $ER := \mathbb{P}_{e \in E}[\hat{f}(e) \neq f(e)]$. For convenience, we analysis ER on CMS rather than S_C.

THEOREM 4.5. *Let Δ be the expectation of extra error rate caused by **Bounded Hash Split**, we have*

$$\Delta \leqslant \overline{\Delta} := \phi(2^\alpha) - \phi(L), \ \text{where } \phi(\zeta) := \frac{N}{L} \times \left(\frac{1}{\zeta}\right)^{d-1} + \frac{N^2}{L^2} \times \left\{\frac{d}{2}(\zeta - 1)\left(\frac{1}{\zeta}\right)^d + 2^{d-2}\left[\left(1-\frac{1}{\zeta}\right)\left(1-\frac{1}{2\zeta}\right)^{d-1} - 1\right]\right\}.$$

PROOF. See Appendix A.5 in the technical report. □

REMARK. We point out that although **Theorem 4.5** seems somewhat complex, this formula indeed provides a quite tight error bound of error rate. For example, when $d = 2, 3$, $\overline{\Delta}$ can be written as

$$\overline{\Delta} = \begin{cases} \frac{N}{L}\left(\frac{1}{2^\alpha} - \frac{1}{L}\right), & d = 2; \\ \frac{N}{L}\left(\frac{1}{2^{2\alpha}} - \frac{1}{L^2}\right) + \frac{N^2}{L^2}\left[\left(\frac{9}{2^\alpha} - \frac{3}{2^{2\alpha}}\right) - \left(\frac{9}{L} - \frac{3}{L^2}\right)\right], & d = 3. \end{cases}$$

We conduct experiments on 10 CAIDA real IP datasets with 10,0000 distinct items each, finding that the experimental results agree well with the theory (**Fig. 6**, L = 4MB).

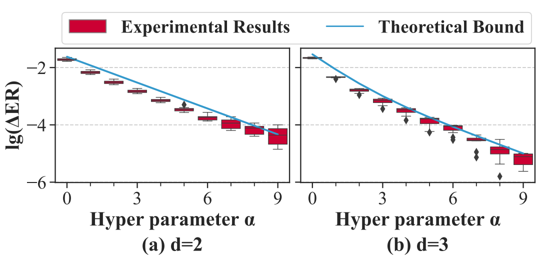

Figure 6: Empirical Result vs Theoretical Bound.

5 PERFORMANCE EVALUATION

In this section, we extend popular sketches to Stingy Sketch Framework and evaluate its performance. In **Subsection 5.1** to **5.5**, we conduct simulation experiments to evaluate parameter setting, memory efficiency, accuracy and throughput of Stingy sketch on frequency estimation and top-k detection tasks[8]. And in **Subsection 5.6**, we deploy Stingy sketch into Apache Flink framework and evaluate its throughput in distributed environment. **For convenience, we use SS to represent the Stingy sketch**.

[8]In **Fig. 8, 9, 10, 16, 17, 18, 22, 23** in this paper, we merely show the experimental results on Campus and Synthetic datasets for space constraints. We put the complete results in Appendix B of our technical report on GitHub [38].

5.1 Experimental Setup

5.1.1 Implementation: For **simulation experiments**, we implement CMS, CUS, CS, SS, SALSA, PS, AS, SAC, USS, and WS in C++[9] and equip them with **PQueue** (**Section 3.2**) and **BHS** (**Section 3.3**). We only apply **PQueue** to accelerate insertion process since the query process may not be continuous. For PS and AS, we simply use the provided open-source code at [39]. We equip every sketch with a well known fast hash function, MurmurHash [40], to compute indices. Because **PQueue** and **BHS** are separate accelerate techniques from **BCTree**, we also apply them to CMS, CUS, CS, SALSA, SAC and form 3 subversions: The **Basic** version, the **BHS** version, and the **BHS+PQueue** version. To form the **BHS** version, we make the mapped counters share the common index to save hash bits, and use a 64-bit hash function to calculate them (we have verified it is enough for our experiments); To form the **BHS+PQueue** version, we add **PQueue** to the BHS version to further accelerate the insertion process[10]. In fact, the only difference between S_{CM} and the **BHS+PQueue** version of CMS is that S_{CM} uses **BCTree** to increase accuracy. We perform all simulation experiments on a machine with 4 core CPUs (Intel (R) Core (TM) i7-10510U CPU @ 1.80GHz) and 16 GB DRAM memory. The CPU core has 256KB L1 cache, 1.0MB L2 cache, and 8.0 MB L3 cache.

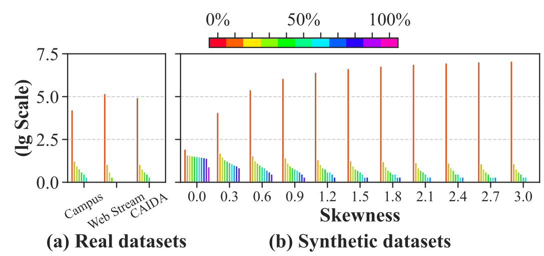

Figure 7: Data Distribution.

5.1.2 Datasets: We use 4 kinds of datasets during the experiments.

- **Campus.** 10 real IP trace datasets collected from the gateway of our campus. Each dataset have about 180 Kilo kinds of items and 2.4 million items in total.
- **Synthetic.** To demonstrate the adaptability of Stingy sketch over a wide range of distributions, we generate 11 synthetic datasets with the skewness[11] varies from 0.3 to 3.0. Every dataset has 32 million items with 4 bytes item ID. Unless otherwise stated, we use the dataset with skewness of 1.5 for general experiments.
- **Web Stream.** 8 real datasets downloaded from [41]. Every dataset has 0.9 million kinds of items (32 million items in total). The item in this dataset represents the number of different terms in web pages.

[9]SS, SALSA and PS are sketch frameworks so they represents 9 explicit algorithms $S_{CM}, S_{CU}, S_C, SALSA_{CM}, SALSA_{CU}, SALSA_C, P_{CM}, P_{CU}$ and P_C. For SS, we change their d arrays into d **BCTree**s (**Section 3.1**).

[10]We do not extend these **PQueue** and **BHS** to speed-oriented sketches Augment and Pyramid because they conflict to their builtin optimizations (i.e. the pre-filter stage in Augment and the word acceleration technique in Pyramid).

[11]Skewness is a measure of the asymmetry of the item count distribution of a flow about its mean. Let X be the random variable representing the number of items of a flow, the skewness is defined as $\mathbb{E}\left[\frac{(X-\mathbb{E}X)^3}{\mathbb{D}^{3/2}X}\right]$ where $\mathbb{E}X$ and $\mathbb{D}X$ are the expectation and variance of X.

• **CAIDA.** 10 real IP trace datasets collected by CAIDA 2018 [42]. Each item has 13 bytes IP address and 8 bytes time stamp. Every dataset has about 1.3 million kinds of items and 26 million anonymized IP traces in total.

Summary: Here we list some key characteristics of all the 4 kinds of datasets: (1) Campus, Web stream and CAIDA are real datasets, while Synthetic are a series of generated datasets. (2) The number of items of Synthetic (32 million), Web Stream (26 million) and CAIDA (32 million) are large, while the number of items of a Campus dataset (2.4 million) is relatively small. (3) All these datasets are highly skewed, while the Web Stream dataset has the highest skewness. Further, the skewness of Synthetic datasets could vary from 0.0 to 3.0. **Fig. 7** shows the numbers (The ordinate is the logarithm to base 10) of items of the top 0% (the max), 10%, 20%, ... , 90%, 100% (the min) flows in each dataset which provides some intuition about the item count distribution of the datasets.

5.1.3 Evaluation Metrics: We measure the following metrics for frequency estimation and top-k detection:

• **Tree Occupation Ratio (TOR)**: The number of occupied counters versus the number of all counters, where *occupied* means the counters have been inserted and are not Kick Tags. Similarly, we use **TOR**$[i]$ ($k = 0, 1, 2, ..., \lfloor \log L \rfloor$) to represent the number of occupied i-level counters versus the number of all i-level counters, and use **Waste Rate (WR)** to represent the number of Kick Tags versus the number of all 0-level counters.

• **Average Absolute Error (AAE)**: $\frac{1}{|E|} \sum_{e \in E} |\hat{f}(e) - f(e)|$, where $|E|$ is the number of distinct items, $f(\cdot)$ and $\hat{f}(\cdot)$ are real and estimated frequency respectively. For top-k detection task, we regard the estimated value of a misreported item e' as 0.

• **F1-Score**: $\frac{2RR \times PR}{RR+PR}$, where $RR := \frac{\text{Reported top-}k}{k}$, $PR := \frac{\text{Reported top-}k}{\text{Reported items}}$. F1-Score is only for top-k detection.

• **Average Insert Throughput (AIT)**: $\frac{T_I}{|\mathcal{S}|}$, where T_I is the total time to insert all items in $\mathcal{S}$. The unit of measurement of AIT is Million operations per second (Mops).

• **Average Query Throughput (AQT)**: $\frac{T_Q}{|E|}$ (Mops), where T_Q is total time to query all items in E. AQT is only for frequency estimation, because the query process in top-k detection isn't that important in real applications.

When we measure speed, we repeat every experiment for 10 times and record the mean value as our result.

5.2 Parameters of the Stingy sketch

In this subsection, we use S_{CM} as an example to explore suitable hyper parameters for the Stingy sketch. For convenience, we list related parameters in **Table 1**.

Table 1: Parameters and their meanings.

Hyper Parameters		*General Parameters*	
α	The offset of **BHS**	d	The number of arrays
Z	The length of **PQueue**	L	Array size
T/τ	The capacity of counter	k	The number of top items

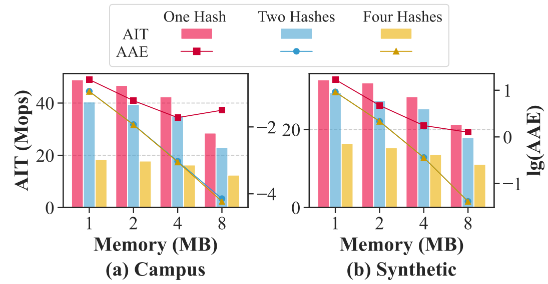

Figure 8: Impact of α for Different Memory.

5.2.1 Impact of parameter α.

• **Analysis:** In **Section 3.3** and **Theorem 4.5**, we have respectively proved (1) **BHS** uses $(\lfloor \log L \rfloor + (d-1)\alpha)$ hash bits to find d mapped counters, (2) An adequate α only leads to low extra error rate. In this part, we show that as long as α is not too small (e.g. $\alpha \geqslant 8$), using 64 hash bits is optimal after weighing accuracy and speed.

• **Experiment:** We fix $d = 4$, using one, two, and four 32-bit MurmurHash functions ($\alpha = \lfloor \frac{32 - \lfloor \log L \rfloor}{d-1} \rfloor, \lfloor \frac{32 - \lfloor \log L \rfloor}{d-1} \rfloor, \lceil \log L \rceil$ respectively to make full use of all hash bits) to conduct the experiments. The experimental result is shown in **Fig. 8**: On the campus and Synthetic datasets, we find that 32-bit **BHS** achieves up to 167% and 100% faster speed, but leads to about 52470% and 1809% extra AAE; while 64-bit **BHS** achieves about 121% and 87% faster than the original version, but leads to less than 5% and 0.02% extra AAE.We reckon the Stingy sketch performs the optimal performance when $\alpha = \lfloor \frac{64 - \lfloor \log L \rfloor}{d-1} \rfloor$. So we use a 64-bit MurmurHash for further experiments. However, when $(\lfloor \frac{64 - \lfloor \log L \rfloor}{d-1} \rfloor)$ is very small, we have to use more than 64 hash bits. For example, when $d = 9$ and $L = 2^{16}$, we should combine a 32-bit function and a 64-bit hash function to calculate 96 hash bits, and use $\alpha = \lfloor \frac{96 - \lfloor \log L \rfloor}{d-1} \rfloor$ = 10-bit offset.

5.2.2 Impact of parameter Z.

• **Analysis:** When Z is small (e.g. $Z = 1$), the time interval between the prefetch instruction and the actual change of counters is less than the addressing time. Thus **PQueue** cannot show its full power. When Z is too large (e.g. $Z = 2^{20}$), the prefetched counters may be frequently swapped out from L2 cache. Thus **PQueue** performs poorly either. Our induction is, there *exists* an appropriate constant $Z \in [1..2^{20}]$ that achieves the highest throughput.

• **Experiment:** We conduct 3 experiments to find the optimal Z and the results are shown in **Fig. 9, 10, 11**. In the first experiment we fix $d = 2$, finding that when the sketch can be loaded in L2 cache (1 MB), the improvement of **PQueue** is small. When the space is getting larger (e.g. *memory* = 8MB), **PQueue** accelerates 56% and 72% on Campus and Synthetic datasets. In the second experiment we fix *memory* = 8MB, finding that AIT is the highest when $Z \approx 16$, so we fix Z to 16 for further (except the next) experiments. We note that the recommended Z may be different based on data stream speed. So we set $u(t, Z) = (1 + \frac{1}{1024} Z)t$ and dynamically change the **PQueue** length to minimize $u(t, Z)$ per $n = 256$ insertions as described in **Section 3.2**. In the third experiment, we manually prolong the item reading time for about 100 ns when we read data that between 25% and 75% in the dataset, finding that higher reading speed often leads to longer **PQueue** length (**Fig. 11 (a)**) because a

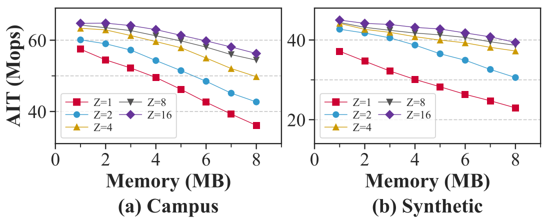

Figure 9: Impact of Z for Different Memory.

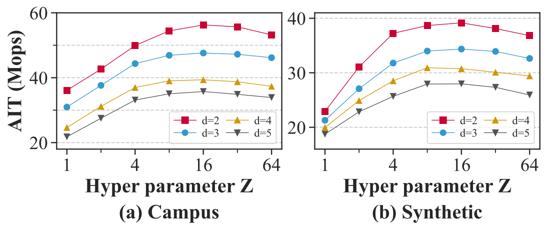

Figure 10: Impact of Z for Different Array Number d.

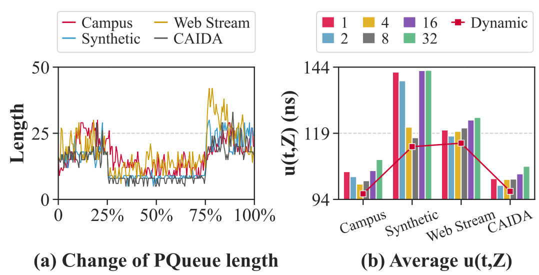

Figure 11: Dynamic PQueue Length.

longer **PQueue** can increase the time interval between prefetching and addressing. **Fig. 11 (b)** shows that the dynamically changing Z do helps to minimize the average $u(t, Z)$ when the data stream speed is constantly changing[12].

5.2.3 Impact of parameters T/τ.

• **Analysis:** In all existing hierarchical works (e.g. Pyramid), every level counter uses the same number of bits. In the Stingy sketch, however, we use 6-bit 0-level counter ($T = 2^6 - 1$) and 2-bit nonzero-level counter ($\tau = 2^2 - 1$) instead. An imprecise reason has been written in **Theorem 4.4**: As long as $\tau \geqslant 2$, the loose upper bound of kick rate $|\mathcal{S}|/T$. So we decrease τ and increase T to optimize this upper bound. In this part, we conduct a more solid experiment on the size of the 0-level counter, finding that a 6-bit 0-level counter is really better than a 4-bit version.

• **Experiment:** The experimental result is shown in **Fig. 12** ($d = 2$ and *memory* =8MB). We find that a 6-bit 0-level counter brings 15% and 37% reduction of AAE than 4-bit one on Campus and Synthetic datasets. Furthermore, since a 6-bit 0-level counter can hold more items, most insertions can finish in the 0-level counter rather than

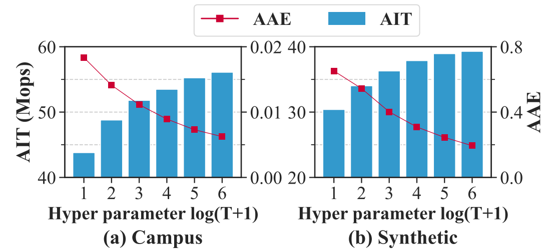

Figure 12: Impact of T/τ.

carry to a higher level counter[13], so AIT of S_{CM} slightly increases as well.

5.3 Memory Efficiency Evaluation

In this subsection, we use S_C as an example to illustrate the memory efficiency of Stingy sketch on TOR, TOR[i] and WR. We give upper bounds of these metrics before experiments.

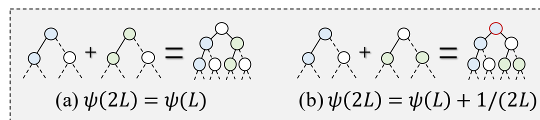

Figure 13: Relation between $\psi(L)$ and $\psi(2L)$.

• **Upper bounds:** First, we point out TOR is always less than 2/3. Suppose L is a power of 2, and the max TOR is defined as $\psi(L)$. We have (1) If $\psi(L)$ is taken *only when* the highest-level (i.e. the ($\log L$)-level) counter is occupied, then $\psi(2L) = \psi(L)$ and the new highest-level counter is not necessarily occupied to reach $\psi(2L)$ (**Fig. 13 (a)**). (2) If the highest-level counter is not necessarily occupied to reach $\psi(L)$, then $\psi(2L) = \psi(L) + 1/(2L)$ and the new highest-level counter must be occupied to reach $\psi(2L)$ (**Fig. 13 (b)**). Note $\psi(1) = \psi(2) = 1/2$, so we have $TOR < \lim\limits_{L \to +\infty} \psi(L) = \sum\limits_{n=0}^{+\infty} 4^{-n}/2 = 2/3$[14]. Second, we point out the upper bound of TOR[i] is simply 1 since all k-level counters may be occupied at the same time. Third, let $\phi(x) := \left(T \frac{\tau^x - 1}{\tau - 1} \frac{L}{2^x} \right), x \in [0, +\infty)$. Then according to **Theorem 4.3**, we have WR $< 1 - 2^{-\phi^{-1}(|\mathcal{S}|)}$ where $|\mathcal{S}|$ is the number of items in the dataset.

• **Experiments and analysis:** Generally speaking, a high and stable TOR often means a high and stable memory efficiency. In this part, we conduct 2 experiments on all the 4 kinds of datasets. In the first experiment, we fix $d = 4$ and change *memory* from 0.25 MB to 8 MB (**Fig. 14**). We find that except on Campus, when memory is very small (e.g. 0.25 MB), TOR is low since most 0-level counters are kicked away and gathered to the higher layers; When memory is very large (e.g. 8 MB), TOR is also low because the number of occupied counters is limited (less than flow number N). We also find that although TOR[i] continuously changes, TOR keeps high and stable in a wide range of memory (e.g. 1~4 MB). In the second experiment, we fix *memory* = 1 MB and 8 MB, changing

[12] To control variables, when Z is fixed, we still periodically measure $u(t, Z)$ which takes some time.

[13] Following the proof in **Theorem 4.2**, we can prove that if we use 4-bit 0-level counter, inserting an item modifies $(1 + [15 \times (1 - 1/15)^{-1} \approx 1.07)$ counters on average, which is more than 6-bit 0-level counter (which modifies $(1 + [T(1 - 1/\tau)^{-1} \approx 1.03)$ counters on average).

[14] Above formula still holds when L is any positive integer.

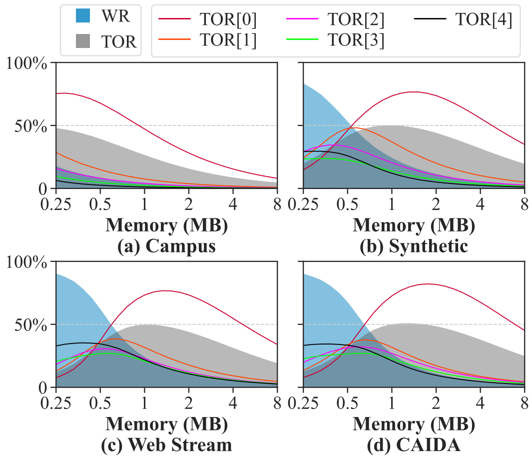

Figure 14: TOR for Different Memory.

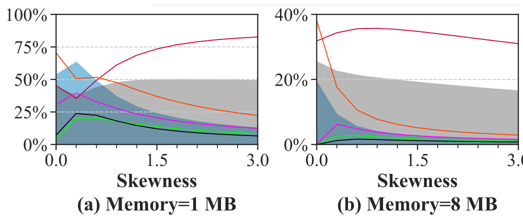

Figure 15: TOR for Different Skewness of Datasets.

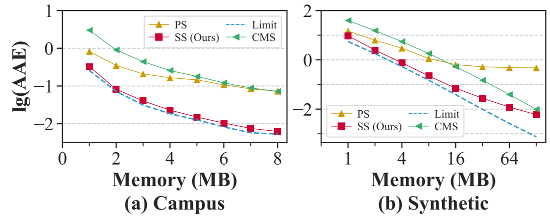

Figure 16: AAE of S_{CM}, PS_{CM} and CMS.

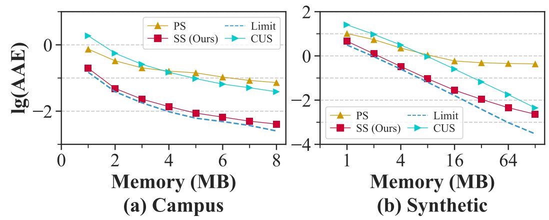

Figure 17: AAE of S_{CU}, PS_{CU} and CUS.

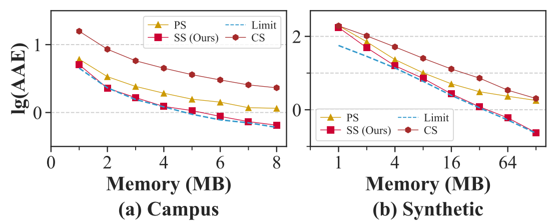

Figure 18: AAE of S_C, PS_C and CS.

the skewness from 0.0 to 3.0 (**Fig. 15**). We find that as the skewness getting larger, TOR becomes higher when $memory = 1$ MB[15], but decreases when $memory = 8$ MB. That's because when $memory = 1$ MB, a higher skewness often reduces the number of large flows even though a large flow may kick more counters. So the waste rate (WR) drops and TOR even increases. However, when $memory = 8$ MB, the increment of skewness greatly reduces the number of overflows and thus decreases TOR. We also find that when a carry chain overflows to a 1-level counter, it probably kicks another one 0-level counter away, so TOR keeps stable in a wide range of skewness. Therefore, we say Stingy sketch has a sound adaptability towards different item count distributions and has a sound memory efficiency.

5.4 Comparison on Accuracy

In this subsection, we conduct experiments to illustrate the accuracy of Stingy Sketch based on AAE and F1-Score.

5.4.1 *Accuracy of Frequency Estimation.*

• **Compare with limestone algorithms:** We fix $d = 2$ and conduct experiments on S_{CM}, S_{CU} and S_C. The experimental result (**Fig. 16, 17, 18**) shows that *SS* significantly outperforms original sketches. The blue dotted line is the AAE of a quadrupled original sketch. Because **BCTree** compress four bytes into one, its accuracy has no reason to exceed an original sketch that consumes 4 times memory. So the blue dotted line is the theoretical limit of accuracy. In this experiment, we find that the Stingy sketch has almost *approached the limit*.

• **Compare with other SOTA algorithms:** We fix $memory = 2$MB and conduct 3 comprehensive experiments (**Fig. 19, 20, 21**). In the first experiment, we find AAE of S_{CM} is up to 50% (Campus), 17% (Synthetic), 41% (Web Stream), and 39% (CAIDA) lower than SOTA accuracy-oriented work SALSA$_{CM}$[16]. We also find when the *memory* is constant, blindly increasing d may increase AAE. So we wonder given *memory* and the flow number N, how to choose an appropriate d to reach optimal accuracy. For AAE, we cannot give such a d because it depends on the item count distribution. But for error rate ER, we can give a recommended d on CMS: Let $dL = C$ is a constant, where L represents the number of counters in one array. According to **Theorem 4.5**, we have $ER := \left(1-(1-1/L)^{N-1}\right)^d = \left(1-(1-1/L)^{L\frac{N-1}{L}}\right)^{\frac{N-1}{L}dL} \xrightarrow[dL=C]{(1-1/L)^L\approx 1/e} \left[\left(1-e^{-\frac{N-1}{C}d}\right)^{\frac{N-1}{C}d}\right]^{\frac{C}{N-1}}$. Note $h(t) := \left(1-e^{-t}\right)^t$ takes the max value when $t = t_0 := 0.693$. So the recommended d is around $\frac{t_0 C}{N-1}$. In the second experiment, we fix $d = 2$ and change the skewness of Synthetic datasets. We find that unless the skewness of the dataset is extremely low (skewness$\leqslant 0.3$), S_{CM} is the optimal algorithm on metric AAE. In the third experiment, we also fix $d = 2$ but compare S_C (not S_{CM}) with SALSA$_C$. The experimental result shows that AAE of S_C is up to 49% lower than SALSA$_C$.

[15]Skewness = 0.0 is a fortunate exception, that's because a Synthetic dataset has 32 MB items and about 1 MB flows. So most flows take exactly 1 or 2 counters and thus TOR is high.

[16]Because PS (Pyramid) maps all counters into one cache line, the most feasible d of PS is 5. So AAE of PS does not change when $d > 5$.

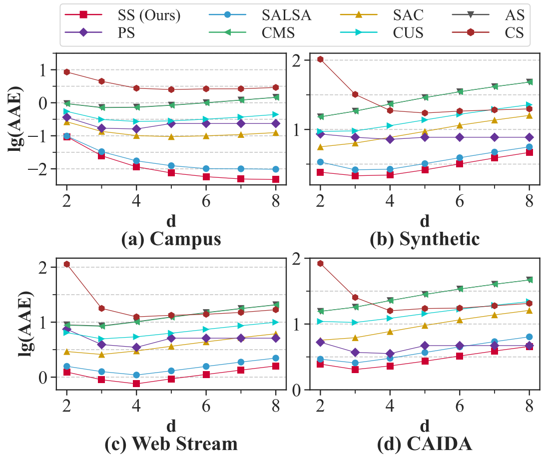

Figure 19: AAE of S_{CM} and Existing Algorithms.

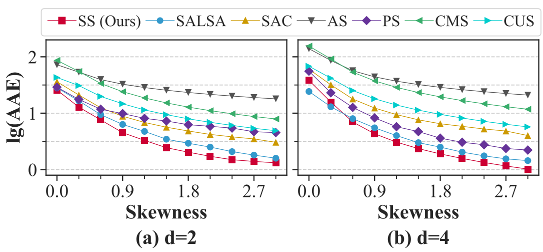

Figure 20: AAE for Different Skewness of Datasets.

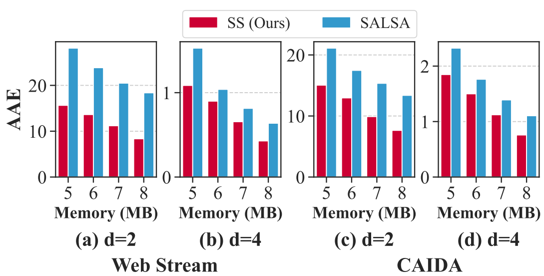

Figure 21: AAE of S_C and $SALSA_C$.

5.4.2 *Accuracy of Unbiased Top-k Detection.*

Top-k detection can be regarded as an extension of frequency estimation. It has many applications such as finding top-k frequent items [43–47], heavy changes [48–50], persistent items [51, 52], and super-spreaders [53]. Although there are many existing algorithms (e.g. Space Saving [37], Lossy Counting [54], Heavy Guardian [4], Heavy Keeper [55], Elastic Sketch [56], CS+Heap, USS, WS), only CS+Heap, USS and WS among them can keep the unbiasedness. In this part, we fix $d = 3$ and conduct experiments on finding top-k frequent items task to show the performance of $S_C + Heap$. Note the *heap* part in CS+Heap and S_C+Heap is a simple minimum heap that records and updates the top-k items.

• **Compare with CS+Heap (Fig. 22, 23):** We find that no matter $k = 500, 1000$, or 2000, $S_C + Heap$ is more accurate on both F1-Score and AAE metrics. Specially, AAE of $S_C + Heap$ is up to 275 and 36 times lower than $CS + Heap$ on Campus and Synthetic datasets.

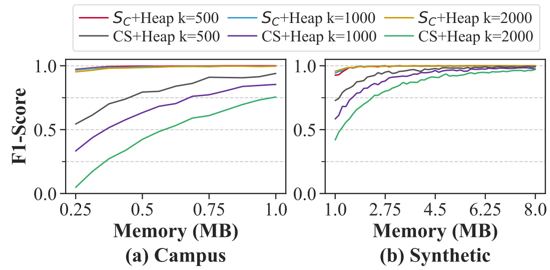

Figure 22: F1-Score of $S_C + Heap$ and $CS + Heap$.

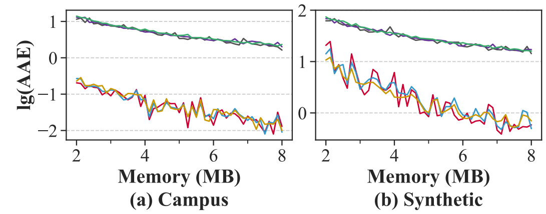

Figure 23: AAE of $S_C + Heap$ and $CS + Heap$.

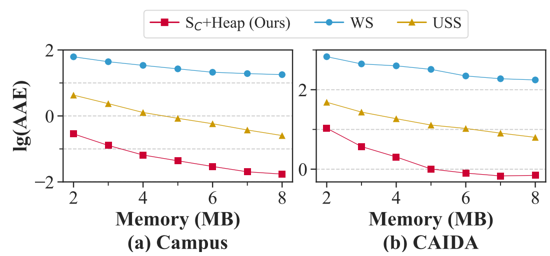

Figure 24: AAE of $S_C + Heap$ and Existing Algorithms.

• **Compare with other SOTA algorithms:** In this part, we compare S_C with other SOTA algorithms USS and WS for different *memory*. Because we reckon a larger k helps to reduce contingency, we fix $d = 2$ and $k = 2000$. The experimental results are shown in **Fig. 24**. We find that AAE of $S_C + Heap$ is up to 19 times lower than USS, as well as 224 times lower than existing SOTA work WS[17]. So we reckon $S_C + Heap$ achieves comparably higher accuracy than existing SOTA algorithms.

5.5 Comparison on Throughput

In this subsection, we conduct experiments to illustrate the throughput of the Stingy sketch based on AIT and AQT.

5.5.1 *Throughput of Frequency Estimation.*

•**Overview:** In this part, we fix $d = 2$ and change *memory* from 1 to 8 MB and conduct 2 experiments on all the 4 kinds of datasets. The experimental results are shown in **Fig. 25, 26** respectively. The first experiment shows that AIT of S_{CM} is up to 343% (Campus), 331% (Synthetic), 266% (Web Stream), and 327% (CAIDA) faster than accuracy-oriented work $SALSA_{CM}$, as well as 100%, 114%, 123%,

[17] In the paper of Waving Sketch [6], the authors provide the unbiased and biased versions of WS. They prove the unbiasedness of the unbiased WS but conduct experiments on the biased WS. So the experimental result seems very accurate. In our paper, however, we use the truly unbiased version of WS as a comparison algorithm, that's why our experimental results seem inconsistent with the original paper.

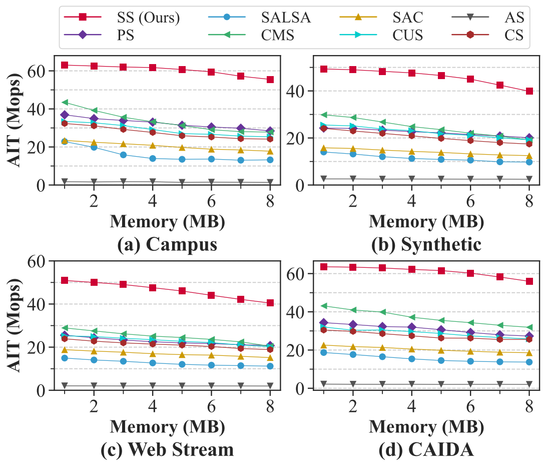

Figure 25: AIT of S_{CM} and Existing Algorithms.

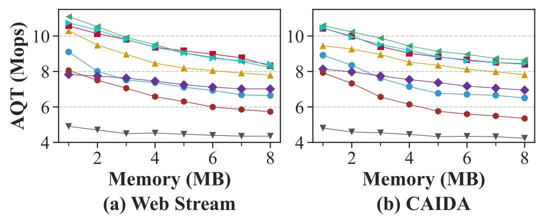

Figure 26: AQT of S_{CM} and Existing Algorithms.

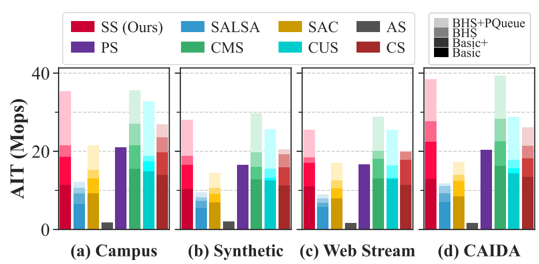

Figure 27: Improvement of BHS and PQ.

and 107% faster than speed-oriented work PS_{CM}. The second experiment shows that AQT of the Stingy sketch also belongs to the first team, which is up to 70%, 28%, 31%, and 30% faster than SALSA$_{CM}$, as well as 33%, 24%, 32%, and 29% faster than PS$_{CM}$. Such a high AQT owes to the Carry Terminator of **CSM** (**Section 3.1.2**), because it lets the query process terminate once it searches to a blank node. Summarizing the above 2 experiments, we reckon Stingy Sketch outperforms existing SOTA algorithms on throughput.

•**Improvement of BHS and PQ:** In this part, we give a more detailed comparison about our speed up techniques: **BHS** and **PQueue**. As we said before, **BHS** and **PQueue** are two generic and fundamental techniques that can be used in a wide range of sketches. So we fix $d = 4$, $memory$ = 8MB, and equip SALSA, CMS, CUS, CS, SAC with (1) **BHS**, (2) **BHS+PQueue** to demonstrate their effects on AIT[18]. Further, we notice that a **Basic** version uses only $d\lceil\log L\rceil = 92$ hash bits. So we use a 64-bit hash function and

[18]We do not apply **BHS** and **PQueue** to AS and PS because AS and PS use their own acceleration techniques (**Section 2.2**) which conflict to ours.

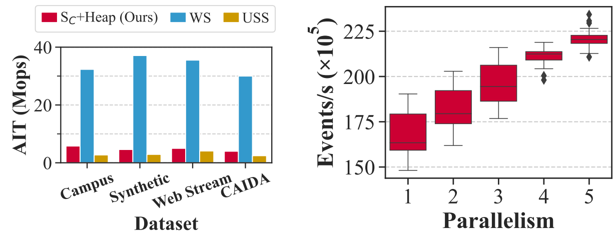

Figure 28

a 32-bit hash function and split them to form a **Basic+** version for fair comparison. The experimental result is shown in **Fig. 27**. We find that except a tiny distance from CMS, S_{CM} is comparably faster than all existing algorithms even if we use the same speed up techniques. For example, AIT of S_{CM} is up to 228% faster than the **BHS+PQueue** version of $SALSA_{CM}$.

5.5.2 Throughput of Top-k Detection.

We fix $memory = 8MB, d = 3, k = 2000$ and compare the AIT of $S_C + Heap$, USS, and WS. The experimental result is shown in the left subfigure of **Fig. 28**. We find that although $S_C + Heap$ cannot exceed WS, the AIT of S_C is up to 114% faster than the Unbiased Space Saving (USS).

5.6 Integration into Apache Flink

We implement Stingy sketch on top of Apache Flink [23], a modern data stream processing framework to show its throughput in distributed environment. To finish the experiment, we rewrite S_{CM} in Java and deploy a Hadoop Distributed File System (HDFS) to feed data into the application where insertion and query are equally seen as *events*. We use a Flink cluster with 1 master node and 4 worker nodes, each of them has 4 Intel XEON Platinum 8369B vCPU cores and 16 GB DRAM. Every Task Manager uses Flink 1.13.1, Java 11 and Hadoop 2.8.3 running on Ubuntu 20.04 LTS, providing 4 available slots and is configured with 1GB memory. We fix $memory = 8MB, d = 4$, change $parallelism$ from 1 to 5 and repeat the test for 20 times on CAIDA. The experimental result is shown in the right subfigure of **Fig. 28**. In this experiment, we find that Stingy sketch works smoothly on top of Flink, and the overall running speed grows linearly with the growth of parallelism.

6 CONCLUSION

In this paper, we propose a sketch framework called Stingy sketch which budgets every penny of computing resource. The Stingy sketch uses **BCTree** which splits large counters into small nodes of a tree structure to reduce error, and uses pipelined prefetch technique **PQueue** to reduce memory access without losing precision. Theoretical and experimental results show that the Stingy sketch outperforms existing works on both accuracy and speed. We believe that the Stingy sketch is a generic and fundamental contribution that can be used in many domains (e.g. data mining and database) and problems (e.g. top-k detection and joining tables). We have released our code at GitHub [38].

REFERENCES

[1] Ran Ben-Basat, Gil Einziger, Isaac Keslassy, Ariel Orda, Shay Vargaftik, and Erez Waisbard. Memento: making sliding windows efficient for heavy hitters. In *CoNEXT*, pages 254–266. ACM, 2018.

[2] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. Faster and more accurate measurement through additive-error counters. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020.

[3] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. SALSA: self-adjusting lean streaming analytics. In *ICDE*, pages 864–875. IEEE, 2021.

[4] Y. Tong, J. Gong, H. Zhang, Z. Lei, and X. Li. Heavyguardian: Separate and guard hot items in data streams. In *the 24th ACM SIGKDD International Conference*, 2018.

[5] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[6] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In *KDD*, pages 1574–1584. ACM, 2020.

[7] B. Shi, Z. Zhao, Y. Peng, F. Li, and J. M. Phillips. At-the-time and back-in-time persistent sketches. In *SIGMOD/PODS '21: International Conference on Management of Data*, 2021.

[8] Y. Izenov, A. Datta, F. Rusu, and J. H. Shin. Compass: Online sketch-based query optimization for in-memory databases. In *SIGMOD/PODS '21: International Conference on Management of Data*, 2021.

[9] A. Santos, A. Bessa, F. Chirigati, C. Musco, and J. Freire. Correlation sketches for approximate join-correlation queries. In *SIGMOD/PODS '21: International Conference on Management of Data*, 2021.

[10] Rundong Li, Pinghui Wang, Jiongli Zhu, Junzhou Zhao, and Kai Ye. Building fast and compact sketches for approximately multi-set multi-membership querying. In *SIGMOD/PODS '21: International Conference on Management of Data*, 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 *SIGMOD/PODS '21: International Conference on Management of Data*, 2021.

[12] Peng Jia, Pinghui Wang, Junzhou Zhao, Shuo Zhang, Yiyan Qi, Min Hu, Chao Deng, and Xiaohong Guan. Bidirectionally densifying LSH sketches with empty bins. In *SIGMOD Conference*, pages 830–842. ACM, 2021.

[13] Pinghui Wang, Yiyan Qi, Yuanming Zhang, Qiaozhu Zhai, Chenxu Wang, John C. S. Lui, and Xiaohong Guan. A memory-efficient sketch method for estimating high similarities in streaming sets. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD, 2019*, pages 25–33. ACM, 2019.

[14] Yang Yang, Ying Zhang, Wenjie Zhang, and Zengfeng Huang. GB-KMV: an augmented KMV sketch for approximate containment similarity search. In *ICDE*, pages 458–469. IEEE, 2019.

[15] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In *LATIN*, Lecture Notes in Computer Science, pages 29–38, 2004.

[16] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, pages 270–313, 2003.

[17] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *ICALP*, pages 693–703, 2002.

[18] Kai Cheng, Limin Xiang, Mizuho Iwaihara, Haiyan Xu, and Mukesh K. Mohania. Time-decaying bloom filters for data streams with skewed distributions. In *RIDE*, pages 63–69. IEEE Computer Society, 2005.

[19] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: a sketch framework for frequency estimation of data streams. *Proc. VLDB Endow.*, 10(11):1442–1453, 2017.

[20] Tong Yang, Siang Gao, Zhouyi Sun, Yufei Wang, Yulong Shen, and Xiaoming Li. Diamond sketch: Accurate per-flow measurement for big streaming data. *IEEE Trans. Parallel Distributed Syst.*, pages 2650–2662, 2019.

[21] Tao Li, Shigang Chen, and Yibei Ling. Per-flow traffic measurement through randomized counter sharing. *IEEE/ACM Trans. Netw.*, pages 1622–1634, 2012.

[22] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *SIGMOD Conference*, pages 1449–1463. ACM, 2016.

[23] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[24] Amit Goyal, Hal Daumé III, and Graham Cormode. Sketch algorithms for estimating point queries in NLP. In *EMNLP-CoNLL*, pages 1093–1103. ACL, 2012.

[25] Peixiang Zhao, Charu C. Aggarwal, and Min Wang. gsketch: On query estimation in graph streams. *Proc. VLDB Endow.*, pages 193–204, 2011.

[26] George Kollios, John W. Byers, Jeffrey Considine, Marios Hadjieleftheriou, and Feifei Li. Robust aggregation in sensor networks. *IEEE Data Eng. Bull.*, pages 26–32, 2005.

[27] Gero Dittmann and Andreas Herkersdorf. Network processor load balancing for high-speed links. In *Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2002.

[28] Atul Kant Kaushik, Emmanuel S. Pilli, and Ramesh C. Joshi. Network forensic analysis by correlation of attacks with network attributes. In *ICT*, pages 124–128, 2010.

[29] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: robust and general sketch-based monitoring in software switches. In *SIGCOMM*, pages 334–350. ACM, 2019.

[30] Alex D Breslow and Nuwan S Jayasena. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment*, pages 1041–1055, 2018.

[31] Qian Liu, Haipeng Dai, Alex X. Liu, Qi Li, Xiaoyu Wang, and Jiaqi Zheng. Cache assisted randomized sharing counters in network measurement. In *ICPP*, pages 40:1–40:10. ACM, 2018.

[32] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter braids: a novel counter architecture for per-flow measurement. In *SIGMETRICS*, pages 121–132. ACM, 2008.

[33] Min Chen, Shigang Chen, and Zhiping Cai. Counter tree: A scalable counter architecture for per-flow traffic measurement. *IEEE/ACM Trans. Netw.*, pages 1249–1262, 2017.

[34] Yun William Yu and Griffin Weber. Hyperminhash: Jaccard index sketching in loglog space. *CoRR*, 2017.

[35] Junzhi Gong, Tong Yang, Yang Zhou, Dongsheng Yang, Shigang Chen, Bin Cui, and Xiaoming Li. ABC: A practicable sketch framework for non-uniform multisets. In *IEEE BigData*, pages 2380–2389. IEEE Computer Society, 2017.

[36] Tong Yang, Jiaqi Xu, Xilai Liu, Peng Liu, Lun Wang, Jun Bi, and Xiaoming Li. A generic technique for sketches to adapt to different counting ranges. In *INFOCOM*, pages 2017–2025, 2019.

[37] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, pages 398–412. Springer, 2005.

[38] Related source code. https://github.com/StingySketch/Stingy-Sketch.

[39] Open source code of augment and pyramid. https://github.com/zhouyangpkuer/Pyramid_Sketch_Framework, 2017.

[40] Murmur hashing source code. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.

[41] The web stream dataset. http://fimi.ua.ac.be/data/.

[42] The caida anonymized internet traces dataset. http://www.caida.org/data/overview/.

[43] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 693–703. Springer, 2002.

[44] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, pages 51–55, 2003.

[45] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003.

[46] Lukasz Golab, David DeHaan, Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Internet Measurement Conference*, pages 173–178. ACM, 2003.

[47] Nishad Manerikar and Themis Palpanas. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data Knowl.*, 2009.

[48] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: methods, evaluation, and applications. In *Internet Measurement Conference*, pages 234–247. ACM, 2003.

[49] Robert T. Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter A. Dinda, Ming-Yang Kao, and Gokhan Memik. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Trans. Netw.*, 15(5):1059–1072, 2007.

[50] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, pages 311–324. USENIX Association, 2016.

[51] Zhewei Wei, Ge Luo, Ke Yi, Xiaoyong Du, and Ji-Rong Wen. Persistent data sketching. In *SIGMOD Conference*, pages 795–810. ACM, 2015.

[52] Haipeng Dai, Muhammad Shahzad, Alex X. Liu, and Yuankun Zhong. Finding persistent items in data streams. *Proc. VLDB Endow.*, pages 289–300, 2016.

[53] Shobha Venkataraman, Dawn Xiaodong Song, Phillip B. Gibbons, and Avrim Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*. The Internet Society, 2005.

[54] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. *Proc. VLDB Endow.*, page 1699, 2012.

[55] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. Heavykeeper: An accurate algorithm for finding top-k elephant flows. *IEEE/ACM Trans. Netw.*, pages 1845–1858, 2019.

[56] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: adaptive and fast network-wide measurements. In *SIGCOMM*, pages 561–575. ACM, 2018.