Tight-Sketch: A High-Performance Sketch for Heavy Item-Oriented Data Stream Mining with Limited Memory Size

Weihe Li
School of Informatics
The University of Edinburgh
Edinburgh, Scotland
weihe.li@ed.ac.uk

Paul Patras
School of Informatics
The University of Edinburgh
Edinburgh, Scotland
paul.patras@ed.ac.uk

ABSTRACT

Accurate and fast data stream mining is critical and fundamental to many tasks, including time series database handling, big data management and machine learning. Different heavy-based detection tasks, such as heavy hitter, heavy changer, persistent item and significant item detection, have drawn much attention from both the industry and academia. Unfortunately, due to the growing data stream speeds and limited memory (L1 cache) available for real-time processing, existing schemes face challenges in simultaneously achieving high detection accuracy, high memory efficiency, and fast update throughput, as we reveal. To tackle this conundrum, we propose a versatile and elegant sketch framework named Tight-Sketch, which supports a spectrum of heavy-based detection tasks. Considering that most items are cold (non-heavy/persistent/significant) in practice, we employ different eviction treatments for different types of items to discard these potentially cold ones as soon as possible, and offer more protection to those that are hot (heavy/persistent/significant). In addition, we propose an eviction method that follows a stochastic decay strategy, enabling Tight-Sketch to only bear small one-sided errors (*no overestimation*). We present a theoretical analysis of the error bounds and conduct extensive experiments on diverse detection tasks to demonstrate that Tight-Sketch significantly outperforms existing methods in terms of accuracy and update speed. Lastly, we accelerate Tight-Sketch's update throughput by up to 36% with Single Instruction Multiple Data (SIMD) instructions.

CCS CONCEPTS

• **Information systems applications → Data stream mining**.

KEYWORDS

data stream mining; heavy item; persistent item; significant item; sustained arrival strength

1 INTRODUCTION

In recent years, massive data transmission has become ubiquitous in social networks [1], financial services [2], and many other areas. Such data streams convey valuable information that can be useful to a range of applications, including business intelligence, anomaly detection [3, 4], recommendation systems [5], etc. One important objectives in stream mining is the identification of heavy items, which spans heavy hitter detection [6–9], heavy changer detection [10–12], persistent item lookup [13–15], and significant item lookup [16, 17]. *Heavy hitters* indicate items with large size or frequency. *Heavy changers* refers to items whose frequency changes dramatically in two contiguous time windows. *Persistent items* represent items which appear in multiple different time windows, while *significant items* are those that have both high frequency and persistence. Real-time detection of any of these is challenging, as high speeds and large volumes preclude recording information pertaining to each item in the detection process [18]. To overcome this obstacle, approximate stream mining leveraging probabilistic data structures such as ***sketches*** has attracted much interest [6, 7, 19–22].

Limitations of Existing Approaches: Even though many sketch-based approaches have been introduced for distinct detection tasks, ultra-fast data stream mining poses significant challenges to existing algorithms. In particular, **(i)** many sketches [24, 25] are non-invertible, meaning that they need to check every item in a stream to retrieve all hot ones, which yields considerable memory access overhead and low throughput. Most existing invertible sketches either track hot items with additional data structures (e.g., heaps) or involve further non-trivial processes (e.g., coding and decoding [15]), resulting in redundant memory access and high computational cost. To boost processing speed, a sketch's update and query process should be straightforward and ideally only access CPU caches when handling high-speed data streams [19]. CPU cache memory is divided into three levels: L1, L2 and L3, among which *the L1 cache is the fastest,* but of size restricted to between 8KB and 64KB in general [26, 27], forcing sketches to be compact enough. Sketches with small sizes bring benefits in many practical scenarios, e.g., to compress gradients and accelerate the training process in distributed machine learning [28–30].

(ii) Moreover, items that appear in data streams usually follow highly *skewed* distributions [31, 32], meaning that most appear infrequently and only a few items exhibit high frequency (or persistence). Unfortunately, most existing sketch-based approaches treat all items indiscriminately and make replacement decisions only based on item size (or persistence), resulting in the incorrect replacement of hot items by abundant cold ones. This problem is exacerbated under L1 cache memory constraints, as hash collisions are more severe, which further compromises detection accuracy.

(iii) Another common issue faced by existing sketches is two-sided estimation errors, i.e., both overestimation and underestimation of item values [6, 33]. Overestimation is particularly detrimental and brings non-negligible performance degradation in many cases [28]. Consider two typical scenarios where overestimation has a negative impact: *(1)* in detecting DDoS attacks, overestimating the malicious traffic volume can cause benign traffic to be wrongly identified as abnormal, resulting in its blocking and thus service disruptions for legitimate users, with reputation damage consequence and revenue loss [34]; *(2)* in distributed machine learning, optimization approaches such as stochastic gradient descent (SGD) [35] move towards minima by following steps in the opposite direction of gradients. However, if the scale of the steps is high, this can hamper convergence. Compared to sketches with overestimation, underestimating gradients might slow down the convergence rate, without harming the learning process [28].

Contributions: To tackle these shortcomings, we propose a new sketch framework named Tight-Sketch, which achieves high detection accuracy, memory efficiency and processing speed, even under tight memory size (L1 cache). Tight-Sketch can be deployed for many heavy-based detection tasks, including heavy hitter detection, heavy changer detection, persistent item lookup, significant item lookup, etc. Tight-Sketch encompasses three key techniques in its operation: (i) we attempt to evict an item tracked in a bucket with a *probabilistic decay* policy, when hash collisions happen during the update process. Precisely, we decrease bucket counters by one with a probability, when a new item arrives; if a bucket's counter reaches zero, the item recorded is discarded, and the newly arrived one will be stored. This way, we ensure Tight-Sketch only owns one-sided estimation errors, i.e., *only bounded underestimation error*, leading to high precision; (ii) considering the highly-skewed distributions of items in data streams, we employ *different eviction treatments for different item types.* For potentially cold items with small counter values, we adopt a higher eviction probability than for hot items, to evict the former quickly, leaving more space for the latter over time; and (iii) to avoid erroneously replacing hot items with cold ones, we introduces a new metric, *sustained arrival strength*, that delivers more protection for hot items based on multidimensional characteristics. This builds on the observation that most cold items are short-lived and arrive in a bursty manner [36–39]. By incorporating the arrival strength feature into the eviction probability, Tight-Sketch effectively circumvents the effortless ejection of hot items by cold ones, significantly improving detection accuracy.

We conduct extensive experiments to demonstrate that Tight-Sketch outperforms state-of-the-art approaches for different detection tasks in terms of accuracy and processing speed. For instance, the average F1 score for heavy hitter detection under extremely tight memories (16KB) is close to 1, up to 24× higher than that of existing methods. Furthermore, Tight-Sketch does not rely on pointers and additional data structures and abandons redundant hash operations once an item finds an available bucket during the update process, attaining higher update throughput than current solutions. Lastly, to accelerate Tight-Sketch's processing speed, we exploit SIMD instructions and parallelize the update process, which increases the update throughput by up to 36%.

2 PROBLEM DEFINITION AND BACKGROUND

2.1 Heavy Item Detection

2.1.1 Definition: Heavy items include heavy hitters and heavy changers. Let $S(e)$ denote the frequency or size of item e, S represent the frequency or total size of all items. Given a pre-defined threshold ϵ, if $S(e) \geq \epsilon S$, we consider item e to be a heavy hitter. Suppose we split the data stream into two equal-sized windows (W_1 and W_2) and use $D(e), D$ to respectively denote the absolute change of item e and all items in two adjacent periods. If $D(e) \geq \epsilon D$, we treat item e as a heavy changer.

2.1.2 Related Work: Existing work for heavy item detection can be divided into two categories: counter-based and sketch-based.

Counter-based algorithms leverage hash tables to record the information (explicit key and value) of heavy items. (Unbiased) Space-Saving [40, 41] employ a data structure named Stream-Summary to track heavy items. When the data structure is full and a newly-arrived item is not tracked, Space-Saving will discard the item with the lowest frequency. Unbiased Space-Saving substitutes the least frequent item based on variance minimization to attain unbiased estimation. RAP [33] expels the item with the smallest value via a probability computed by the frequency, when there is no space for newly arrived items. The replacement strategy of these methods is based solely on the estimated frequency, which cannot provide enough protection for heavy items under tight memory settings, resulting in modest detection accuracy. In addition, the update process of counter-based methods mainly relies on pointers, and many pointer operations for insertion significantly reduce update speeds.

Sketch-based algorithms harness a compact data structure to record the accumulated information of all items, attaining high update speeds and a small memory footprint by sacrificing a certain level of accuracy. Count-min Sketch [24] uses a two-dimensional array with r rows; each row has b buckets for tracking items hashed to these buckets [25]. When a new item arrives, Count-min Sketch hashes this item into r different buckets, and then the corresponding counter in each bucket is increased by one (or the item's size). Finally, the smallest value among r-hashed rows is regarded as the estimated size. Count-min Sketch is non-invertible, which means it involves considerable memory access operations that harm update speeds. It also has a significant overestimation issue under tight memories, leading to many non-heavy items being incorrectly recognized as heavy. Count-min Sketch Heap [25] introduces an additional heap to track heavy items. However, access to this slows the update speed. To improve detection accuracy and throughput, MV-Sketch [6] adopts the majority vote algorithm to track heavy items. HeavyKeeper [7] evicts items from the sketch by obeying an exponential decay strategy. Elastic Sketch [21] partitions the sketch into a heavy and a light part, to record the information of heavy and non-heavy items, respectively. CocoSketch [43] employs stochastic variance minimization to support arbitrary partial key queries. However, these methods mainly replace items only based on their frequency, which cannot protect heavy items adequately, leading to many heavy items being replaced by non-heavy ones.

2.2 Persistent Item Detection

2.2.1 Definition: Given a stream divided into N consecutive and non-overlapping time windows, the persistence of an item e is the number of discrete windows in which item e appears, denoted as $P(e)$. With a user-defined η, if $P(e) \geq \eta N$, item e is persistent.

2.2.2 Related Work: Existing solutions for persistent item detection can be divided into sample-, coding-, and sketch-based.

Sample-based methods such as Small-Space [14] record persistent items with a probability and track them into a hash table. Chen et al. introduce adaptive sampling to track persistent items without knowing the monitoring time horizon [47]. Even though such approaches seek to alleviate memory usage via sampling, they still track many non-persistent items, leading to poor memory efficiency. Moreover, the sample rate is configured according to the memory budget, and small values amplify detection errors when the memory is tight. To address this inefficiency, ***coding-based*** methods, like PIE [15], leverage Raptor codes to encode each item and store the code instead of the item ID. However, every item needs to be encoded in each window, which wastes resources for processing large volumes of non-persistent items. Also, encoding and decoding are additional operations that increase processing times and harm update speeds. ***Sketch-based*** methods such as On-Off Sketch [13] adopt a flag bit to increase the persistence periodically, and propose to separate persistent/non-persistent items. Unfortunately, the naïve partitioning causes persistent items to be mistakenly expelled by non-persistent ones, yielding inferior detection accuracy when memory size is limited.

2.3 Significant Item Detection

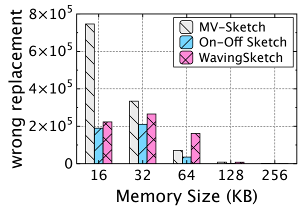
2.3.1 Definition: Suppose a data stream is partitioned into N equal-sized time windows. The significance $G(e)$ of an item e is a weighted sum of two metrics, the frequency $S(e)$ and persistence $P(e)$, and is computed as $G(e) = \alpha S(e) + \beta P(e)$, where α and β are user-defined [16, 17]. Given a threshold G, an item e is considered to be a significant item if $G(e) \geq G$.

2.3.2 Related Work: Long-Tail Clock (LTC) [16] leverages two essential techniques, Long-tail Restoring and an adapted CLOCK algorithm, for significant item lookup. Long-tail Restoring exploits the long-tail distribution feature of real datasets to mitigate the overestimation, and the adapted CLOCK algorithm periodically increases each item's persistence. Nonetheless, the complicated processing makes it hard for LTC to match high-speed data streams.

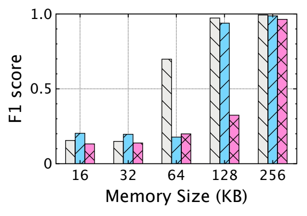
2.4 Summary

Limitations of Prior Art: Existing schemes for different detection tasks *struggle to concurrently maintain high accuracy, high memory efficiency and fast update speed under limited memory size.* To further illustrate the inefficiencies of current methods, we take three state-of-the-art approaches as examples: MV-Sketch [6] for heavy hitter detection, and On-Off Sketch [13] and WavingSketch [20] for persistent item lookup. We vary the memory size from 16KB to 256KB [45] to count the number of hot items being mistakenly substituted by cold ones during the update process, followed by evaluating their detection accuracy. We conduct these tests using a CAIDA 2016 [55] trace with 0.64M items and set the thresholds ϵ and η for heavy hitter detection and persistent item lookup as 0.0005 and 0.5, respectively. Figure 1(a) demonstrates that when the memory size is tight (≤64KB), the number of wrong replacement events increases significantly. This indicates that current methods are ineffective in protecting hot items, when using fast L1 cache memories (which typically range between 8KB and 64KB [26]). The impact of memory size on detection accuracy is illustrated in Figure 1(b), which shows that MV-Sketch's F1 score is 5.4× lower when the memory size is 16KB compared to when it is 256KB.

Motivation: Our analysis indicates that current methods perform poorly when the memory size is limited. The main reason is that under these conditions many hot items are mistakenly replaced by cold ones due to frequent hash collisions, resulting in low detection accuracy. In order to address this issue, we introduce a new sketch-based approach that uses more data stream features to better protect hot items from being replaced by cold ones, while maintaining fast update speeds.



(a) Number of wrong replacements.



(b) F1 score.

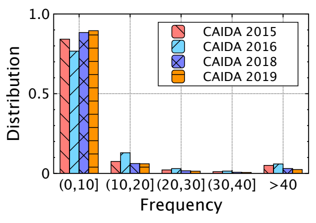
Figure 1: Wrong replacement events and detection accuracy with state-of-the-art sketches, under different memory sizes.

3 TIGHT-SKETCH DESIGN

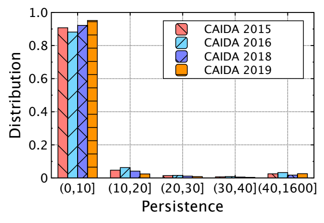
In this section, we first conduct a data analysis and reveal the two primary design rules behind Tight-Sketch, then introduce the data structure it employs and basic operations (update and query).

3.1 Design Rules

Rule 1: The distribution of items in real data streams is highly skewed, indicating that most are small and only a tiny fraction are large [31, 32, 49, 50]. We employ four datasets, CAIDA 2015, 2016, 2018, and 2019, to confirm this feature. Each trace consists of 0.45M, 0.64M, 1.29M, and 1.53M items. We divide traces into five parts, according to the frequency and persistence of items. Note that other number of partitions could be also used. As shown in Figure 2(a), we find that most items have a frequency of no more than 10, and only a tiny portion of items possess a frequency greater than 40. Similarly, we divide each trace into 1,600 time windows [13], and find that around 92% of items have a persistence of less than 10, while only 2.5% have a persistence greater than 40 on average (Figure 2(b)). These results reveal that most items are cold and only appear a few times. Therefore, it is appropriate to discard these cold items as soon as possible, to leave memory space for hot ones.



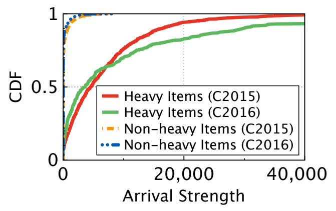
(a) Frequency.



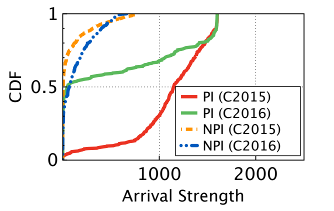
(b) Persistence.

Figure 2: Item frequency and persistence distributions in different real-world datasets.

Rule 2: The transmission of large amounts of items is often characterized by repeating patterns of active and inactive transmission, as already observed widely in practice [31, 36, 49, 51, 52]. In particular, unlike massive amounts of short-lived cold items with small frequencies and long inactive periods, the active periods for hot items are much longer, indicating that their arrival is more sustained than that of cold ones [36]. To verify this property, we utilize MV-Sketch [6] and WavingSketch [20] to observe the sustained arrival strength of items tracked in each bucket. We set the memory size to 64KB and divide the CAIDA 2015 and 2016 traces into 1,600 time windows [13]. When a new item arrives, its arrival strength is increased by one if it has already been tracked in the hashed bucket. Otherwise, the arrival strength of the item stored in the hashed bucket is reduced by 1, with a minimum value of 0. Figure 3 illustrates that the sustained arrival strength of hot items is significantly higher than that of cold items. Therefore, sustained arrival strength is a valuable metric for identifying hot items and can be employed in various detection tasks.



(a) Heavy item detection. C→CAIDA.



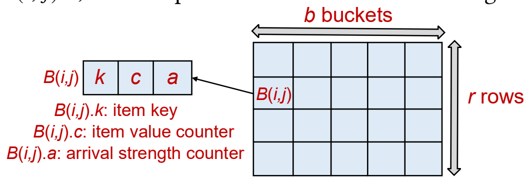
(b) Persistent item detection. (N)PI→(non-)persistent items.

Figure 3: Sustained arrival strength of hot and cold items.

Summary: Based on the above analysis, we find that hot items primarily have a higher frequency/persistence and a stronger sustained arrival strength than cold items. Thus, our Tight-Sketch harnesses these features to evict cold items as soon as possible and provide more protection for hot items, thereby significantly improving detection accuracy even under limited memory budgets.

3.2 Data Structure

There mainly exist two types of data structures in current sketches: flat [25] and hierarchical [44]. Instead of the sophisticated hierarchical structure with multiple layers, we choose the classic flat structure for Tight-Sketch, as it bears faster processing speed and it is easier to deploy in practice. As illustrated in Figure 4, Tight-Sketch's data structure consists of r rows, each containing b buckets. Each row is associated with a different pairwise-independent hash function, denoted as $h_1, h_2, \cdots, h_r$. $B(i,j)$ represents a bucket in the i-th row and j-th column, where $1 \le i \le r$ and $1 \le j \le b$. The bucket $B(i,j)$ has three fields: $B(i,j).k$, which stores the key of the candidate item; $B(i,j).c$, which maintains a statistic of the candidate item, such as its frequency, persistence, or significance; and $B(i,j).a$, which represents the item's arrival strength.



b buckets
r rows
B(i,j)
k
c
a
B(i,j)
B(i,j).k: item key
B(i,j).c: item value counter
B(i,j).a: arrival strength counter

Figure 4: Tight-Sketch's data structure.

3.3 Update and Query

Tight-Sketch supports two basic operations, *update* and *query*. Specifically, *update* is essential for inserting a newly arrived item into a bucket probabilistically. *Query* is for returning the hot items whose value is greater than a predefined threshold.

3.3.1 Update. The update process for each incoming item e is outlined in Algorithm 1, which consists of two stages. The first stage (Lines 2-10) involves determining whether the incoming item has already been recorded or if there is an empty bucket to store it. If not, the second stage (Lines 11-22) involves replacing the item currently tracked in a bucket with the incoming item using a probabilistic decay method.

Stage I. Upon the arrival of a new item e, Tight-Sketch first maps this item to a bucket with the hash function h_1 in the first row. If the bucket $B(1, h_1(e.k))$ is empty or has been occupied by item e, the key field of the mapped bucket will be set as $e.k$, and both counters will increase by 1. However, if a different item already occupies the bucket, it indicates that item e was unable to be stored in the first row, and a hash collision has occurred. In this case, Tight-Sketch will iteratively check the remaining rows using the hash functions $h_2, \cdots, h_r$ to locate an available bucket for item e. Once an available bucket is found, the hash operation terminates (Lines 2-7).

Compared to existing methods that hash an item across all rows, e.g., MV-Sketch [6] and HeavyKeeper [7], Tight-Sketch avoids redundant hashing operations and conserves memory usage, allowing more space to track hot items. Suppose hash collisions happen in all rows, indicating that item e cannot find an available bucket. In that case, Tight-Sketch will evaluate the bucket with the smallest value counter to determine if item e can be successfully stored by replacing the item currently therein (Lines 8-10). Also, the occurrence of hash collisions during the mapping process is an indication that the item recorded does not have a sustained presence. As a result, the sustained arrival strength counter for the hashed bucket can be decremented by 1 (Line 11). This decrease in the arrival strength counter allows for the potential eviction of the item in favor of incoming items with a more sustained presence −*recall that hot items tend to have stronger sustained arrival strength.*

Stage II. Tight-Sketch employs a finer grained approach to item eviction than many recent schemes that often expel items indiscriminately [14, 25, 40]. Given that in practice most items are cold, Tight-Sketch prioritizes the eviction of these items to conserve more space for hot ones. To achieve this, Tight-Sketch employs a threshold value M, which is usually set to a small value (e.g., $M = 10$). If the value counter of a hashed bucket is less than M, the counter is decreased with a higher rate of $\frac{1}{B(p,q).c+1}$ (Lines 12-13). In contrast, if the value counter is greater than or equal to M, the counter is decreased with a more conservative probability $\frac{1}{B(p,q).c \times B(p,q).a+1}$ that considers both the item's value and arrival strength (Lines 14-15). Hot items with high frequency and sustained arrival strength will quickly exceed the threshold M and will be harder to evict. We verify empirically that this process delivers better guarding of hot items than other probabilistic eviction strategies, such as probabilistic decay without considering the arrival strength. If the value counter is successfully decreased to 0, an incoming item e can replace the incumbent item in the bucket and set the value counter to 1 (Lines 16-19). Otherwise, Tight-Sketch will discard the incoming item (Lines 21-22).

3.3.2 Query. Unlike non-invertible approaches that require the examination of every item in the stream to return all hot items, Tight-Sketch only requires a scan of each bucket to determine which items are hot. Specifically, Tight-Sketch checks the value counter of each bucket to see if it is above a predefined threshold. If so, the item stored in that bucket is reported as hot.

3.4 Utilizing Tight-Sketch for Various Tasks

We apply Tight-Sketch to four different detection tasks: heavy hitter detection, heavy changer detection, persistent item lookup, and significant item lookup.

Algorithm 1: Tight-Sketch's Update Procedure

```
Input: a newly incoming item $e$, hash function associated with
       each row $h_1, ..., h_r$, $min \leftarrow +\infty$
Initialization: Each bucket's counters and item key are initialized
  to 0 and null, respectively.
// Stage I: locating an available bucket
for $i = 1$ to $r$ do
    if $B(i, h_i(e)).k == null \,||\, B(i, h_i(e)).k == e.k$ then
        $B(i, h_i(e)).k \leftarrow e.k$;
        $B(i, h_i(e)).c \leftarrow B(i, h_i(e)).c + 1$;
        $B(i, h_i(e)).a \leftarrow B(i, h_i(e)).a + 1$;
        return;
    else if $B(i, h_i(e)).c < min$ then
        $min \leftarrow B(i, h_i(e)).c$;
        $p \leftarrow i; q \leftarrow h_i(e.k)$;
    $B(i, h_i(e)).a \leftarrow max(B(i, h_i(e)).a - 1, 0)$;
// Stage II: probabilistic decay
if $B(p, q).c < M$ then
    if $random(0, 1) < \frac{1}{B(p,q).c+1}$ then
        $B(p, q).c = B(p, q).c - 1$
else if $random(0, 1) < \frac{1}{B(p,q).c \times B(p,q).a+1}$ then
    $B(p, q).c = B(p, q).c - 1$
if $B(p, q).c == 0$ then
    $B(p, q).k \leftarrow e.k$;
    $B(p, q).c \leftarrow B(p, q).c + 1$;
    return;
else
    Discard the incoming item $e$;
    return;
```

3.4.1 Heavy Hitter Detection. Since Tight-Sketch can be directly deployed for heavy hitter detection, the data structure, update and query operations are consistent with Sections 3.2 and 3.3.

3.4.2 Heavy Changer Detection. For each time window, we construct a Tight-Sketch to track the frequency of items and compare changes in their frequency in adjacent windows, to find heavy changers. When an incoming item e arrives, we insert it into Tight-Sketch based on its period. The insertion process is the same as in Section 3.3. Suppose the frequency of item e in the first and second time windows is $S_1(e)$ and $S_2(e)$. If the variation $|S_1(e) - S_2(e)|$ is greater than the threshold ϵD, item e is reported as a heavy changer.

3.4.3 Persistent Item Lookup. Each item's persistence only increases by 1 in a time window, no matter how many times it arrives. To eliminate duplicates, Tight-Sketch includes a flag field (*true or false*) in its data structure [13]. A *true* flag value indicates that a bucket has not been accessed in the current time window and is set to $false$ after access. At the beginning of each time window, the algorithm first checks the flag in each bucket. If the flag is *true*, indicating the recorded item does not appear in the last window, the arrival strength of that item will be decreased by 1. Then, all flag fields are reset to *true*. To optimize memory usage, the algorithm uses the highest bit of the arrival strength counter to store the flag field, instead of adding a separate field to the data structure. This allows Tight-Sketch to efficiently track and update the status of items while minimizing memory usage.

Upon the arrival of a new item e, Tight-Sketch first searches for an available bucket with a flag value *true*. If such a bucket is found, the value counter and arrival strength counter are incremented by 1, and the flag field is set to $false$. If an available bucket is not identified, Tight-Sketch attempts to evict the incumbent item with the smallest persistence counter across all rows. If the flag of the chosen bucket is $false$, indicating that the incumbent item arrived in the current time window, item e is discarded, and the eviction process is terminated. Otherwise, the replacement procedure is carried out according to Algorithm 1 (Lines 12-22). The query operation for retrieving persistent items is consistent with Section 3.3.

3.4.4 Significant Item Lookup. To identify significant items, Tight-Sketch needs to track the frequency and persistence of each item. To accomplish this, we modify the data structure in bucket $B(i, j)$ to include the following fields: k, which indicates the item identifier; fc, a value counter for item frequency; fa, a sustained arrival strength counter for frequency; pc, a persistence counter; and pa, an arrival strength counter for persistence. We also use the highest bit of pa to record the flag (*true/false*) for removing duplicates.

When an incoming item arrives, it will first search for an available bucket. If it fails, it will attempt to evict the tracked item with minimal significance among all mapped buckets in each row. Suppose the significance of the recorded item is smaller than the threshold M. In that case, Tight-Sketch will decrease the value counters by 1 with a probability that only considers the frequency and persistence values. Otherwise, Tight-Sketch will decay the value counters considering the arrival strength. Since the persistence value is no more than the frequency value, once the persistence counter is decreased to 0, the newly arrived item can successfully replace the tracked item in the bucket. After insertion, Tight-Sketch scans each bucket to return items with significance higher than G.

4 MATHEMATICAL ANALYSIS

In this section, we first prove that Tight-Sketch does not suffer overestimation errors. We then derive an underestimation error bound, using heavy hitter detection as a concrete example. Note that persistent item lookup can also be seen as a special case of heavy item detection, where the frequency of each item only increases by one within a given time window. Therefore, the analysis presented can be easily extended to hold for persistent item lookup as well.

4.1 No Overestimation Error

Theorem 4.1. *For an item e, let $S_t(e)$ and $\hat{S}_t(e)$ respectively denote the real frequency and estimated frequency at any given time t. We have $\hat{S}_t(e) \leq S_t(e)$.*

Proof. The proof is available in Appendix.A. □

4.2 Underestimation Error Bound

Theorem 4.2. *For a heavy item e, we assume that it will successfully enter the mapped bucket once it arrives and remain there until the detection task ends. Given a small positive number σ and a heavy item e with frequency $S(e)$, $\Pr\left\{S(e) - \hat{S(e)} \geq \lceil \sigma N \rceil\right\} \leq \frac{\delta}{\sigma N}\left[\ln(S(e)) + L\right]$ holds, where δ is the faction of non-heavy items among all items, L denotes the Euler-Mascheroni constant, N is the number of all entries for all items.*

Proof. When an item different from e arrives and is mapped into the same bucket $B(i, j)$ as e, the value counter of this bucket is either reduced by 1 or left unchanged. Let $Q_{i.j}$ denote how many times items that differ from e hashed into the same bucket, we attain $S(e) - Q_{i.j} \leq B(i, j).c \leq S(e)$. We utilize a random variable $R_{i,j,x}$

to denote whether the value counter of bucket $B(i,j)$ decreases by 1 when the x-th item arrives, where $1 \le x \le Q_{i.j}$. Thus, $B(i,j).c = S(e) - \sum_{x=1}^{Q_{i,j}} R_{i,j,x}$. According to the Markov inequality, with a small positive number σ, we attain

$$\Pr\left\{B(i,j).c \le S(e) - \sigma N\right\} = \Pr\left\{S(e) - \sum_{x=1}^{Q_{i,j}} R_{i,j,x} \le S(e) - \sigma N\right\}$$

$$= \Pr\left\{\sum_{x=1}^{Q_{i,j}} R_{i,j,x} \ge \sigma N\right\} \le \mathbb{E}\left[\sum_{x=1}^{Q_{i,j}} R_{i,j,x}\right]\frac{1}{\sigma N}.$$

Assume all entries follow a uniform distribution, with each arriving item having an equal probability to decay the tracked item's counter,

$$\mathbb{E}\left[\sum_{x=1}^{Q_{i,j}} R_{i,j,x}\right] = \mathbb{E}\left[Q_{i,j}R_{i,j,x}\right] = \sum_{Q_{i,j}=1}^{S(e)} p(Q_{i,j})\left[Q_{i,j}\mathbb{E}(R_{i,j,x}|Q_{i,j})\right].$$

We use ψ to denote the value of the value counter of bucket $B(i,j)$ when the detection task ends. Since the frequency of heavy items is much greater than the threshold M and the stronger arrival strength causes the heavy items to exceed M quickly, the decay of the value counter in the bucket holding heavy items is mainly based on item frequency and arrival strength. As we assume that a heavy item can successfully enter the bucket, the reduction operation will only occur if the incoming item is a non-heavy one. Therefore,

$$\mathbb{E}(R_{i,j,x}|\psi) = \sum_{c=1}^{\psi}\frac{\delta}{\psi}\frac{1}{(c\times a)+1},$$

where c and a represent the value counter and sustained arrival strength counter of bucket $B(i,j)$, and δ is the ratio of non-heavy items in all items.

Since a heavy item generally carries much more data than all other items that are hashed to the same bucket [6], we obtain

$$\mathbb{E}(R_{i,j,x}|Q_{i,j}) = \sum_{\psi=S(e)-Q_{i,j}}^{S(e)-1} p(\psi)\sum_{c=1}^{\psi}\frac{\delta}{\psi}\frac{1}{(c\times a)+1}$$

$$\le \sum_{\psi=S(e)-Q_{i,j}}^{S(e)-1} p(\psi)\sum_{c=1}^{\psi}\frac{\delta}{S(e)-Q_{i,j}}\frac{1}{c}$$

$$\le \sum_{\psi=S(e)-Q_{i,j}}^{S(e)-1} p(\psi)\sum_{c=1}^{S(e)}\frac{\delta}{S(e)-Q_{i,j}}\frac{1}{c} = \frac{\delta}{S(e)-Q_{i,j}}\sum_{c=1}^{S(e)}\frac{1}{c}.$$

Since e is a heavy item, it owns a large value and thus $p(Q_{i,j})$ obeys a Possion distribution with mean $\frac{N}{b}p(Q_{i,j}) = \frac{N}{b}e^{-\frac{N}{b}Q_{i,j}}$, where b is the number of buckets in each row. Then we get

$$\mathbb{E}\left[\sum_{x=1}^{Q_{i,j}} R_{i,j,x}\right] \le \sum_{Q_{i,j}=1}^{S(e)-1} p(Q_{i,j})\left(Q_{i,j}\frac{\delta}{S(e)-Q_{i,j}}\sum_{c=1}^{S(e)}\frac{1}{c}\right)$$

$$= \sum_{Q_{i,j}=1}^{S(e)-1}\frac{N}{b}e^{-\frac{N}{b}Q_{i,j}}\left(Q_{i,j}\frac{\delta}{S(e)-Q_{i,j}}\sum_{c=1}^{S(e)}\frac{1}{c}\right)$$

$$= \sum_{c=1}^{S(e)}\frac{1}{c}\left[\sum_{Q_{i,j}=1}^{\frac{S(e)}{2}}\frac{N}{b}e^{-\frac{N}{b}Q_{i,j}}\left(Q_{i,j}\frac{\delta}{S(e)-Q_{i,j}}\right)\right.$$

$$\left.+\sum_{Q_{i,j}=\frac{S(e)}{2}+1}^{S(e)-1}\frac{N}{b}e^{-\frac{N}{b}Q_{i,j}}\left(Q_{i,j}\frac{\delta}{S(e)-Q_{i,j}}\right)\right]$$

$$\le \sum_{c=1}^{S(e)}\frac{1}{c}\left[\sum_{Q_{i,j}=1}^{\frac{S(e)}{2}}\frac{\delta N}{b}e^{-\frac{N}{b}Q_{i,j}} + \sum_{Q_{i,j}=\frac{S(e)}{2}}^{S(e)-1}\frac{\delta N}{b}e^{-\frac{N}{b}\frac{S(e)}{2}}Q_{i,j}\frac{1}{S(e)-Q_{i,j}}\right]$$

$$\le \sum_{c=1}^{S(e)}\frac{1}{c}\left[\delta + \sum_{Q_{i,j}=\frac{S(e)}{2}}^{S(e)-1}\frac{\delta N}{b}e^{-\frac{N}{b}\frac{S(e)}{2}}(S(e)-1)\frac{1}{S(e)-(S(e)-1)}\right]$$

$$\le \sum_{c=1}^{S(e)}\frac{1}{c}\left[\delta + \sum_{Q_{i,j}=\frac{S(e)}{2}}^{S(e)-1}\frac{\delta N}{b}e^{-\frac{N}{b}\frac{S(e)}{2}}S(e)\right]$$

$$\le \sum_{c=1}^{S(e)}\frac{1}{c}\left[\delta + S(e)\frac{\delta N}{b}\frac{S(e)}{2}e^{-\frac{N}{b}\frac{S(e)}{2}}\right] \le \sum_{c=1}^{S(e)}\frac{\delta}{c}.$$

Generally, $S(e)$ is a large number. Thus, $\sum_{c=1}^{S(e)}\frac{\delta}{c}$ can be approximated as $\delta\left[\ln(S(e)) + L\right]$, where L denotes the Euler-Mascheroni constant [48]. Finally, we get the underestimation error bound as

$$\Pr\left\{S(e) - \hat{S(e)} \ge \lceil\sigma N\rceil\right\} \le \Pr\left\{B(i,j).c \le S(e) - \sigma N\right\}$$

$$\le \frac{\mathbb{E}\left[\sum_{x=1}^{Q_{i,j}} R_{i,j,x}\right]}{\sigma N} \le \frac{\delta}{\sigma N}\left[\ln(S(e)) + L\right].$$

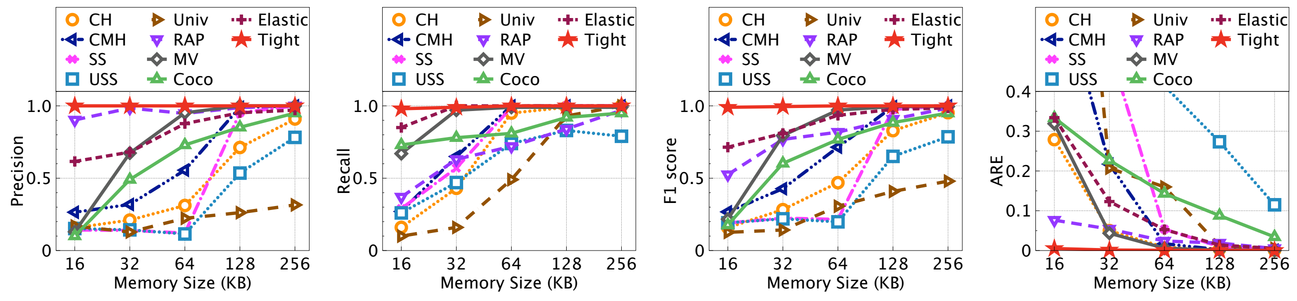
□

5 EVALUATION

To evaluate the performance of Tight-Sketch, we implement it as well as existing schemes in C++. We conduct experiments on a computer with 16GB DRAM memory, and an Intel(R) Core(TM) i5-1135G7 @ 2.40GHz CPU. Each core owns a 48KB L1 data cache and a 1,280KB L2 cache. All cores share a 8,192KB L3 cache.

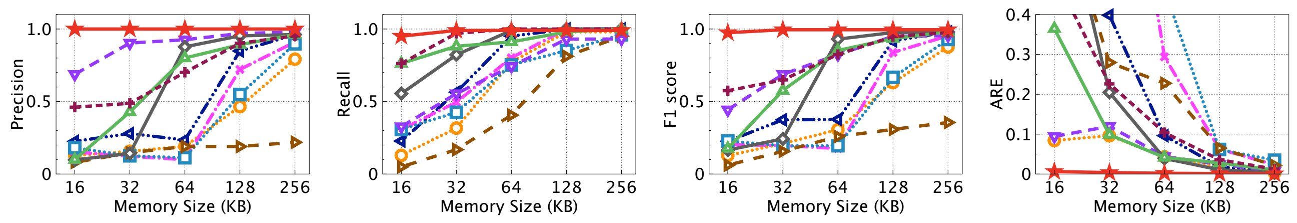
Datasets: We employ three datasets for evaluation: (i) CAIDA [55], which contains anonymized IP trace streams collected from CAIDA. We pick two traces from 2015 and 2018, with 0.52M and 0.77M items, respectively; (ii) MAWI [56], which presents traffic traces collected by MAWI in Japan. We select a trace with 2.75M items from 2020; (iii) Campus [57], a dataset consisting of campus network traffic collected over 10 days in 2016. We randomly pick a trace that contains 0.87M items for evaluation. For these traces, we regard source-destination pairs as item keys (8 bytes).

Methodology: For heavy item detection, we compare Tight-Sketch (Tight) with MV-Sketch (MV) [6], CocoSketch (Coco) [43],



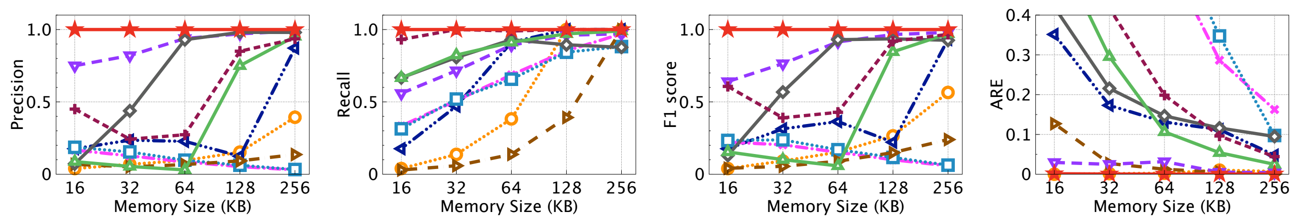
(a) Precision. **(b) Recall.** **(c) F1 score.** **(d) ARE.**

Figure 5: Heavy hitter detection with different approaches, as a function of memory size (CAIDA 2015).



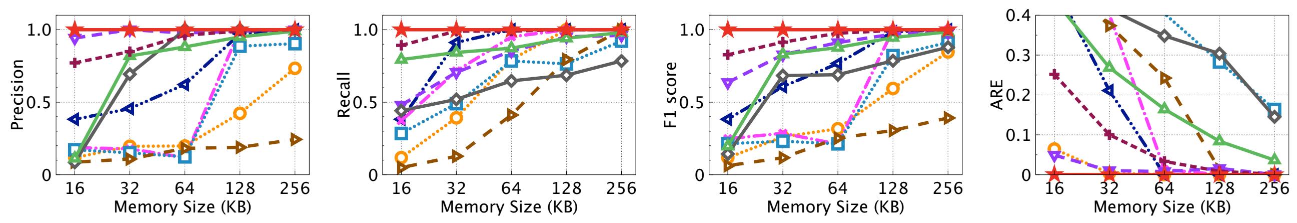
(a) Precision. **(b) Recall.** **(c) F1 score.** **(d) ARE.**

Figure 6: Heavy hitter detection with different approaches, as a function of memory size (CAIDA 2018).



(a) Precision. **(b) Recall.** **(c) F1 score.** **(d) ARE.**

Figure 7: Heavy hitter detection with different approaches, as a function of memory size (MAWI).



(a) Precision. **(b) Recall.** **(c) F1 score.** **(d) ARE.**

Figure 8: Heavy hitter detection with different approaches, as a function of memory size (Campus).

Elastic [21], RAP [33], USS [41], UnivMon (Univ) [58], CMHeap (CMH) [25], CountHeap (CH) [24] and Space-Saving (SS) [40]. For MV-Sketch, we configure the number of rows as 4 [6]. For RAP, we set the number of arrays as 2. The parameter settings of the rest of the schemes are consistent with [43]. In addition, for a comprehensive assessment, we also compare Tight-Sketch with the advanced probability-based methods HeavyKeeper [7], UA-Sketch [54], and PRECISION [60]. For persistent item lookup, we divide each trace into 1,600 time windows [13] and select two off-the-shelf benchmarks, On-Off Sketch (On-Off) [13] and WavingSketch (Waving) [20]. The number of cells for On-Off Sketch and WavingSketch is 16 [20]. For significant item lookup, we compare Tight-Sketch with the state-of-the-art approach LTC [16], using its default settings.

For Tight-Sketch, we set the number of rows r as 4 [6, 59] and alter b based on memory budgets. We default to select the threshold that keeps the hot items around 100 for each detection task [45]. An analysis on how to configure the parameter M can be found in Appendix.B.

Metrics: We use the following five performance metrics. (i) Recall: fraction of true reported items over all true items; (ii) Precision: fraction of true reported items over all reported items; (iii) F1 score: $\frac{2 \times recall \times precision}{recall + precision}$; (iv) Average Relative Error (ARE): $\frac{1}{|\Phi|}\sum_{e \in \Phi} \frac{|S(e) - \hat{S}(e)|}{S(e)}$, which evaluates the error rate of the estimated value; (v) Update throughput: the update speed of the algorithm, in millions of operations per second (Mops).

5.1 Performance on Heavy Hitter Detection

We vary the memory size from 16KB to 256KB [45] and compare the performance of Tight-Sketch with existing approaches on heavy hitter detection. Figures 5–8 detail this across different datasets.

Precision (Figures 5(a)–8(a)): We find that the precision of Tight-Sketch is always 1, outperforming existing approaches even under limited memory size (16KB). Specifically, Tight-Sketch ameliorates the precision by 4%-356%, 12%-506%, 12%-1106%, and 2%-518% on average under these datasets, respectively. The superiority of Tight-Sketch stems from its finer update operations, which avoid overestimation errors and effectively circumvent the effortless eviction of heavy items by non-heavy ones.

Recall (Figures 5(b)–8(b)): Tight-Sketch maintains its optimality in terms of recall on different traces, with an improvement of up to 85% across the CAIDA 2015 trace, 106% across the CAIDA 2018 trace, 209% across the MAWI trace, and 110% across the Campus trace. During the update process, Tight-Sketch effectively alleviates the interference of non-heavy items on heavy items with the help of stream characteristics (the heavy-tail feature helps to evict cold items with high probability; the arrival strength provides more protection to hot items). In addition, abandoning hash operations in time saves memory usage, leaving more space for Tight-Sketch to record heavy items and thus guaranteeing a high recall.

F1 score (Figures 5(c)–8(c)): Compared with current methods, Tight-Sketch attains the highest F1 score under different memory budgets. Even with 16KB of memory, the F1 score reaches around 1, enhancing the detection accuracy by 39%-6879%, 70%-1489%, 56%-2450%, and 21%-1500%, respectively, across different datasets.

ARE (Figures 5(d)–8(d)): We find that Tight-Sketch also obtains the lowest estimation error as compared to existing approaches. For instance, under the CAIDA 2015 trace, the ARE of Tight-Sketch is 23× and 72× smaller than that of RAP and Elastic on average, which demonstrates the effectiveness of Tight-Sketch.

Deep Dive: (i) We investigate the reasons behind Tight-Sketch's significant performance improvements by counting the number of incorrect replacement events during the update process. As observed in Table 1, Tight-Sketch efficiently mitigates the occurrence of mistakenly substituted heavy items by non-heavy ones, leading to high detection accuracy. Compared with MV-Sketch, when the memory size is 16KB, the number of wrong replacement events by Tight-Sketch is 2525× smaller. (ii) In addition to RAP and Coco-Sketch, which conduct admission operations based on probability, a series of advanced works also follow probabilistic replacement, namely HeavyKeeper [7], UA-Sketch [54], and PRECISION [60]. We also evaluate the performance of these methods under the CAIDA 2018 dataset. The results confirm that Tight-Sketch outperforms state-of-the-art probability-based sketch techniques. Specifically, when using a memory size of 16KB, Tight-Sketch, HeavyKeeper, UA-Sketch, and PRECISION achieve F1 scores of 0.99, 0.11, 0.86, and 0.21, respectively, verifying the outstanding performance of Tight-Sketch. Besides, unlike these methods which are exclusively designed for heavy item detection, Tight-Sketch also owns more versatility and can be deployed for various detection tasks. (iii) Although both schemes use the probability decay strategy to evict items stored in buckets, Tight-Sketch and HeavyGuardian [46] differ significantly. Firstly, HeavyGuardian uses exponential decay to decrease the value counter based solely on the item information, such as the item frequency. However, when cold items arrive in a bursty manner in a short period of time, they can increase the counter value quickly, making it difficult to evict them from the bucket. In contrast, for Tight-Sketch, the low sustained arrival strength of cold items accelerates their eviction, which mitigates the interference of cold items with hot ones, guaranteeing a high detection accuracy even under limited memory size. Secondly, HeavyGuardian uses an auxiliary list to record potential hot items, while Tight-Sketch avoids maintaining additional data structures, reducing the memory overhead. We conducted experiments using the CAIDA 2015 and MAWI traces to compare the performance of HeavyGuardian and Tight-Sketch in detecting heavy items with memory sizes ranging from 16KB to 256KB. The results reveal that, on average, Tight-Sketch outperforms HeavyGuardian by 7.69% and 36.79% in terms of F1 score for the two traces, respectively, confirming the superiority of Tight-Sketch. (iv) The above experiments involve detecting heavy items by considering their frequency. In addition, we evaluate the performance of Tight-Sketch in identifying heavy items based on their size. The experimental results demonstrate that despite tight memory constraints (16KB), Tight-Sketch still achieves an F1 score of around 1 across various datasets, indicating its excellent lookup accuracy (figure omitted due to space limitations).

Table 1: # of incorrect replacement events (CAIDA 2015).

Memory (KB)	16	32	64	128	256
Tight-Sketch	**64**	**34**	**15**	**6**	**6**
MV-Sketch	161,659	41,690	5,949	957	220

5.2 Performance on Other Detection Tasks

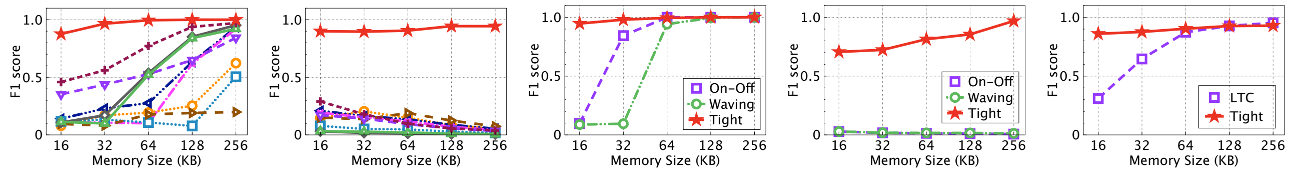
Heavy Changer Detection (Figures 9(a),(b)): The results of our analysis show that the F1 score of Tight-Sketch is on average 31% higher than the most competitive approach, Elastic, when applied to the CAIDA 2015 dataset. While the MAWI trace exhibits less skewness, the performance of the considered benchmarks is significantly diminished in comparison to the CAIDA trace. However, Tight-Sketch still maintains its high detection performance in this scenario, demonstrating its robustness and effectiveness.

Persistent Item Detection (Figures 9(c),(d)): Tight-Sketch demonstrates superior performance in persistent item lookup, in comparison to existing methods, with a 25% improvement and 5163% enhancement over On-Off Sketch on the CAIDA 2015 and MAWI traces, respectively.

Significant Item Detection (Figure 9(e)): We set the threshold values α and β to 1. Our results reveal that Tight-Sketch consistently achieves the highest detection accuracy, even when the available memory size is restricted. With a memory size of 16KB, the F1 score of Tight-Sketch is 178% higher than that of the state-of-the-art LTC.

5.3 Impact of Different Thresholds

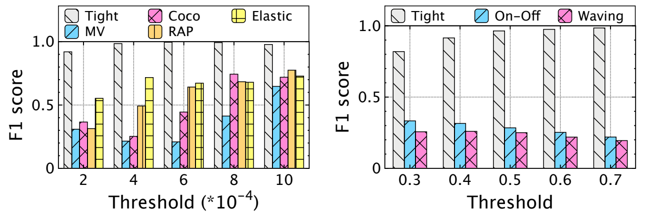
We sought to identify the top 100 hot items from high-speed streams in the above experiments. Here, we examine the impact of varying thresholds on the performance of different methods. To do so, we fix the memory size at 32KB and vary the ϵ and η threshold values for heavy hitter detection and persistent item lookup, respectively, in the range of 0.0002-0.001 and 0.3-0.7. As shown in Figure 10, Tight-Sketch is superior across a range of thresholds. In the case of heavy hitter detection, when ϵ is set to 0.0002, Tight-Sketch outperforms Elastic by 66%. For persistent item lookup, we observe that the performance of On-Off Sketch and WavingSketch



(a) Heavy changer (CAIDA'15). (b) Heavy changer (MAWI). (c) Persistent item (CAIDA'15). (d) Persistent item (MAWI). (e) Significant item (CAIDA'15).

Figure 9: F1 score for other tasks across different traces (the legend of heavy changer detection is the same as that in Figure 5).

decreases as η increases. This is due to the fact that the number of persistent items decreases with increasing thresholds, and the rough replacement strategies of On-Off Sketch and WavingSketch result in many persistent items being incorrectly replaced by non-persistent ones, leading to low detection accuracy. In contrast, Tight-Sketch achieves the highest detection performance, with a 349% improvement over On-Off Sketch when η is set to 0.7. These results highlight the robustness of Tight-Sketch under a range of thresholds.

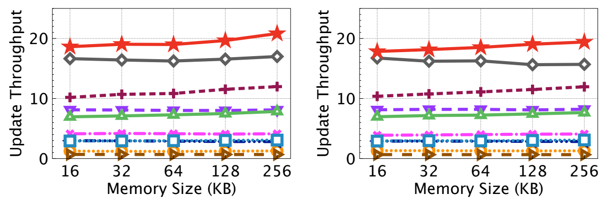


(a) Heavy hitter detection. (b) Persistent item lookup.

Figure 10: Detection accuracy under different thresholds (memory size: 32KB, CAIDA 2018).

5.4 Update Throughput and Query Time

5.4.1 Update Speed. We leverage heavy hitter detection as an example to investigate the update speed of Tight-Sketch. Figure 11 compares the update throughput of various algorithms under different memory sizes, revealing that Tight-Sketch yields the highest update speed, which is 17% and 15% higher than that of MV-Sketch on the CAIDA 2015 and 2018 traces, respectively. This can be attributed to Tight-Sketch's simple update rule and the elimination of unnecessary hash operations in time. We further assess Tight-Sketch's update throughput on other detection tasks and find that it consistently outperforms the considered benchmarks.



(a) CAIDA 2015. (b) CAIDA 2018.

Figure 11: Update Throughput (Mops) with different schemes across the CAIDA traces (legend as in Figure 5).

5.4.2 Query Time. Here, we utilize the CAIDA 2015 trace to evaluate the query time of Tight-Sketch for heavy hitter detection. Table 2 presents the query time of different algorithms, with our findings demonstrating that Tight-Sketch achieves the lowest query time among the tested algorithms. This can be attributed to the invertibility of Tight-Sketch and the fact that it doesn't require extra hash operations during the query process, resulting in a shorter query times than with existing schemes. Conversely, MV-Sketch required additional hash operations during querying, leading to longer query times. We observe a similar trend in the results for other detection tasks, such as persistent item lookup.

Table 2: Query time for heavy item detection (Memory: 32KB)

Scheme	Tight	MV	Elastic	USS
Query Time (ms)	29.073	161.481	105.069	655.831

5.4.3 Optimization with SIMD Instructions. During the update process, Tight-Sketch must sequentially check the buckets in each row to locate one available for an incoming item. In the worst case, Tight-Sketch must check all rows, which slows the update speed. To further increase performance, we employ SIMD instructions and process sequential operations in parallel. As an incoming item arrives, we first utilize the primitive `MurmurHash3_x64_128` to obtain the hash value based on the item key. Then, we divide the hash value into r parts, where r is the number of rows in the Tight-Sketch data structure. Next, we obtain the bucket positions in each row and track them into a register array and use `_mm256_cmpeq_epi64` to compare the newly arrived item's key with items recorded in r rows in parallel. With this method, Tight-Sketch with SIMD instructions can quickly locate an available bucket for a newly arrived item in a single step. Table 3 compares the update speed of Tight-Sketch with and without SIMD instructions, revealing up to 36% improvements.

Table 3: Tight-Sketch's update throughput (Mops) with SIMD.

Memory Size (KB)	16	32	64	128	256
Tight-SIMD	**24.2**	**24.3**	**24.6**	**24.8**	**25.4**
Tight-Sketch	17.8	18.1	18.5	19	19.4

6 CONCLUSIONS

This paper presents Tight-Sketch, a novel sketch that achieves high detection accuracy even with limited memory budgets while maintaining fast update speeds. Specifically, Tight-Sketch follows a probabilistic decay strategy to cautiously substitute incumbent items tracked in buckets based on multidimensional features. We apply Tight-Sketch on different heavy-based detection tasks and conduct extensive experiments with diverse datasets to confirm its superiority. Our results show that Tight-Sketch dramatically outperforms existing approaches in all scenarios. We further optimize Tight-Sketch with SIMD instructions, thereby enhancing its update throughput and enabling our solution to match very fast data streams.

ACKNOWLEDGEMENT

This work was partially supported by Cisco through the Cisco University Research Program Fund (Grant no. 2019-197006).

REFERENCES

[1] N. Tang, Q. Chen, and P. Mitra, "Graph Stream Summarization: From Big Bang to Big Crunch," in Proceedings of ACM SIGMOD, 2016.

[2] B. Ball, M. Flood, H.V. Jagadish, J. Langsam, L. Raschid, P. Wiriyathammabhum, "A Flexible and Extensible Contract Aggregation Framework (CAF) for Financial Data Stream Analytics," in Proceedings of ACM DSMM, 2014.

[3] Q. Xiao, Y. Qiao, M. Zhen, and S. Chen, "Estimating the Persistent Spreads in High-speed Networks," in Proceedings of IEEE ICNP, 2014.

[4] N. Immorlica, K. Jain, M. Mahdian, and K. Talwar, "Click Fraud Resistant Methods for Learning Click Through Rates," in Proceedings of ACM WINE, 2005.

[5] S. Gündüz, M.T. Özsu, "A Web Page Prediction Model Based on Click-Stream Tree Representation of User Behavior," in Proceedings of ACM KDD, 2003.

[6] L. Tang, Q. Huang, and P.P.C. Lee, "MV-Sketch: A Fast and Compact Invertible Sketch for Heavy Flow Detection in Network Data Streams," in Proceedings of IEEE INFOCOM, 2019.

[7] J. Gong, T. Yang, H. Zhang, H. Li, S. Uhlig, S. Chen, L.Uden, and X. Li, "Heavy-Keeper: An Accurate Algorithm for Finding Top-k Elephant Flows," in Proceedings of USENIX ATC, 2018.

[8] Q. Xiao, Z. Tang, and S. Chen, "Universal Online Sketch for Tracking Heavy Hitters and Estimating Moments of Data Streams," in Proceedings of IEEE INFOCOM, 2020.

[9] Q. Xiao, H. Wang, and G. Pan, "Accurately Identify Time-decaying Heavy Hitters by Decay-aware Cuckoo Filter along Kicking Path," in Proceedings of IEEE/ACM IWQoS, 2022.

[10] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A Better NetFlow for Data Centers," in Proceedings of USENIX NSDI, 2016.

[11] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based Change Detection: Methods, Evaluation, and Applications," in Proceedings of ACM IMC, 2003.

[12] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P.A. Dinda, M. Kao, and G. Memik, "Reversible Sketches: Enabling Monitoring and Analysis Over High-Speed Data Streams," IEEE/ACM Transactions on Networking, vol. 15, no. 5, pp. 1059-1072, 2007.

[13] Y. Zhang, J. Li, Y. Lei, T. Yang, Z. Li, G. Zhang, and B. Cui, "On-Off Sketch: A Fast and Accurate Sketch on Persistence," in Proceedings of VLDB Endowment, 2020.

[14] B. Lahiri, J. Chandrashekar, and S. Tirthapura, "Space-efficient Tracking of Persistent Items in a Massive Data Stream," in Proceedings of ACM DEBS, 2011.

[15] H. Dai, M. Shahzad, A.X. Liu, and Y. Zhong, "Finding Persistent Items in Data Streams," in Proceedings of VLDB Endowment, 2016.

[16] T. Yang, H. Zhang, D. Yang, Y. Huang, and X. Li, "Finding Significant Items in Data Streams," in Proceedings of IEEE ICDE, 2019.

[17] S. Cheng, D. Yang, T. Yang, H. Zhang, and B. Cui, "LTC: A Fast Algorithm to Accurately Find Significant Items in Data Streams," IEEE Transactions on Knowledge and Data Engineering, vol. 34, no. 9, pp. 4342-4356, 2022.

[18] T. Yang, Y. Zhou, H. Jin, S. Chen, X. Li, "Pyramid Sketch: a Sketch Framework for Frequency Estimation of Data Streams," in Proceedings of VLDB Endowment, 2017.

[19] Z. Zhong, S. Yan, Z. Li, D. Tan, T. Yang, and B. Cui, "BurstSketch: Finding Bursts in Data Streams," in Proceedings of ACM SIGMOD, 2021.

[20] J. Li, Z. Li, Y. Xu, S. Jiang, T. Yang, B. Cui, Y. Dai, and G. Zhang, "WavingSketch: An Unbiased and Generic Sketch for Finding Top-k Items in Data Streams," in Proceedings of ACM KDD, 2020.

[21] T. Yang, J. Jing, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic Sketch: Adaptive and Fast Network-wide Measurements," in Proceedings of ACM SIGCOMM, 2018.

[22] S. Sheng, Q. Huang, S. Wang, and Y. Bao, "PR-Sketch: Monitoring Per-key Aggregation of Streaming Data with Nearly Full Accuracy," in Proceedings of VLDB Endowment, 2021.

[23] G. Cormode, 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.

[24] M. Charikar, K. Chen, and M.F. Colton, "Finding Frequent Items in Data Streams," in Proceedings of Springer ICALP, 2002.

[25] G. Cormode, 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.

[26] "CPU and memory," https://www.bbc.co.uk/bitesize/guides/zmb9mp3/revision/3.

[27] J. Chen, C. Lu, J. Ni, X. Guo, P. Girard and Y. Cheng, "DOVA PRO: A Dynamic Overwriting Voltage Adjustment Technique for STT-MRAM L1 Cache Considering Dielectric Breakdown Effect," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 29, no. 7, pp. 1325-1334, 2021.

[28] J. Jiang, F. Fu, T. Yang, and B. Cui, "SketchML: Accelerating Distributed Machine Learning with Data Sketches," in Proceedings of ACM SIGMOD, 2018.

[29] K. Yang, et al., "FastSGD: A Fast Compressed SGD Framework for Distributed Machine Learning," arXiv preprint arXiv:2112.04291, 2021.

[30] D. Rothchild, A. Panda, E. Ullah, N. Ivkin, I. Stoica, V. Braverman, J. Gonzalez, and R. Arora, "FetchSGD: Communication-Efficient Federated Learning with Sketching," in Proceedings of ACM ICML, 2020.

[31] T. Benson, A. Akella, and D.A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in Proceedings of ACM SIGCOMM, 2010.

[32] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold Filter: A Meta-Framework for Faster and More Accurate Stream Processing," in Proceedings of ACM SIGMOD, 2018.

[33] 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.

[34] D. Ding, M. Savi, and D. Siracusa, "Tracking Normalized Network Traffic Entropy to Detect DDoS Attacks in P4," IEEE Transactions on Dependable and Secure Computing, vol. 19, no. 6, pp. 4019-4031, 2022.

[35] S.I. Amari, "Backpropagation and Stochastic Gradient Descent Method," Neurocomputing, vol. 5, no. 4-5, pp. 185-196, 1993.

[36] J. Liu, J. Huang, W. Lv, and J. Wang, "APS: Adaptive Packet Spraying to Isolate Mix-Flows in Data Center Network," IEEE Transactions on Cloud Computing, vol. 10, no. 2, pp. 1038-1051, 2022.

[37] H. Xu, and B. Li, "RepFlow: Minimizing Flow Completion Times with Replicated Flows in Data Centers," in Proceedings of IEEE INFOCOM, 2014.

[38] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding Data Center Traffic Characteristics," ACM SIGCOMM Computer Communication Review, vol. 40, no. 1, pp. 92-99, 2010.

[39] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron, "Better Never than Late: Meeting Deadlines in Datacenter Networks," in Proceedings of ACM SIGCOMM, 2011.

[40] A. Metwally, D. Agrawal, and A.E. Abbadi, "Efficient Computation of Frequent and Top-k Elements in Data Streams," in Proceedings of Springer ICDT, 2005.

[41] D. Ting, "Data Sketches for Disaggregated Subset Sum and Frequent Item Estimation," in Proceedings of ACM SIGMOD, 2018.

[42] C. Estan, G. Varghese, "New Directions in Traffic Measurement and Accounting," in Proceedings of ACM SIGCOMM, 2002.

[43] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Mao, P. Liu, R. Zhang, and J. Jiang, "CocoSketch: High-Performance Sketch-based Measurement over Arbitrary Partial Key Query," in Proceedings of ACM SIGCOMM, 2021.

[44] T. Yang, S. Gao, Z. Sun, Y. Wang, Y. Shen and X. Li, "Diamond Sketch: Accurate Per-flow Measurement for Big Streaming Data," in IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 12, pp. 2650-2662, 2019.

[45] J. Huang, W. Zhang, Y. Li, L. Li, Z. Li, J.Ye and, J. Wang, "ChainSketch: An Efficient and Accurate Sketch for Heavy Flow Detection," in IEEE/ACM Transactions on Networking, 2022, doi: 10.1109/TNET.2022.3199506.

[46] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "HeavyGuardian: Separate and Guard Hot Items in Data Streams," in Proceedings of ACM KDD, 2018.

[47] L. Chen, R. C.-W. Phan, Z. Chen, and D. Huang, "Persistent Items Tracking in Large Data Streams Based on Adaptive Sampling," in Proceedings of IEEE INFOCOM, 2022.

[48] R. Courant, F. John, AA. Blank, A. Solomon, "Introduction to calculus and analysis. Vol. 1," New York: Interscience Publishers, 1965.

[49] J. Hu, J. Huang, W. Lyu, W. Li, Z. Li, W. Jiang, J. Wang, and T. He, "Adjusting Switching Granularity of Load Balancing for Heterogeneous Datacenter Traffic," IEEE/ACM Transactions on Networking, vol. 29, no. 5, pp. 2367-2384, 2021.

[50] J. Huang, W. Li, Q. Li, T. Zhang, P. Dong, and J. Wang, "Tuning High Flow Concurrency for MPTCP in Data Center Networks," Journal of Cloud Computing, vol. 9, no. 13, pp. 1-15, 2020.

[51] T. Zhang, J. Wang, J. Huang, J. Chen, Y. Pan and G. Min, "Tuning the Aggressive TCP Behavior for Highly Concurrent HTTP Connections in Intra-Datacenter," IEEE/ACM Transactions on Networking, vol. 25, no. 6, pp. 3808-3822, 2017.

[52] Z. Liu et al., "BurstBalancer: Do Less, Better Balance for Large-scale Data Center Traffic," in Proceedings of IEEE ICNP, 2022.

[53] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities," in Proceedings of ACM SIGCOMM, 2018.

[54] J. Ye, L. Li, W. Zhang, G. Chen, Y. Shan, Y. Li, W. Li, and J. Huang, "UA-Sketch: An Accurate Approach to Detect Heavy Flow based on Uninterrupted Arrival," in Proceedings of ACM ICPP, 2022.

[55] "The CAIDA Anonymized Internet Traces," http://www.caida.org/data/overview/.

[56] "MAWI Working Group Traffic Archive," http://mawi.wide.ad.jp/mawi/.

[57] M. Singh; M. Singh, S. Kaur, "10 Days DNS Network Traffic from April-May, 2016," Mendeley Data, V2, 2019, doi: 10.17632/zh3wnddzxy.2.

[58] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in Proceedings of ACM SIGCOMM, 2016.

[59] L. Tang, Q. Huang and P.P.C. Lee, "SpreadSketch: Toward Invertible and Network-Wide Detection of Superspreaders," in Proceedings of IEEE INFOCOM, 2020.

[60] R.B. Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Designing Heavy-Hitter Detection Algorithms for Programmable Switches," IEEE/ACM Transactions on Networking, vol. 28, no. 3, pp. 1172-1185, 2020.

A NO OVERESTIMATION ERROR

THEOREM A.1. *For an item e, let $S_t(e)$ and $\hat{S}_t(e)$ respectively denote the real frequency and estimated frequency at any given time t. We have $\hat{S}_t(e) \leq S_t(e)$.*

PROOF. When the detection task starts ($t = 0$), $\hat{S}_t(e)$ and $S_t(e)$ are both zero, so the theorem holds. Suppose that the Theorem A.1 is valid at time $t - 1$, that is $\hat{S}_{t-1}(e) \leq S_{t-1}(e)$. At time t, there exist two cases: (i) if the incoming item is again e, we have $\hat{S}_t(e) = \hat{S}_{t-1}(e) + 1$ and $S_t(e) = S_{t-1}(e) + 1$. Thus, $\hat{S}_t(e) \leq S_t(e)$ holds in this scenario; (ii) if an item different from e arrives, the estimation of item e either is decreased by 1 or remains the same, which is $\hat{S}_t(e) = \hat{S}_{t-1}(e) - 1$ or $\hat{S}_t(e) = \hat{S}_{t-1}(e)$. Since $S_t(e) = S_{t-1}(e)$, we obtain $\hat{S}_t(e) \leq S_t(e)$.

Since the claim holds in all cases, Theorem A.1 is proven. □

B PARAMETER SETTING

Similar to existing work on parameter settings [19, 20], we conduct an experiment to investigate the impact of varying the value of M on the detection accuracy of significant item lookup. Specifically, we vary M from 1 to 100 and observe its effect on the F1 score. Our experimental results reveal that when we increase M from 0 to 10, the F1 score shows a rising trend. When the memory size is 16KB, the F1 score at $M = 10$ is 3.1% higher than that at $M = 0$. This is because most cold items have a low frequency or persistence and fall into this range, and setting M to a small value accelerates their eviction process. When M ranges between 10 and 50, the F1 score shows a similar trend. However, when we further increase the value of M, the F1 score decreases. This is because setting M to a larger value may increase the decay rate of hot items, which can negatively impact on the detection accuracy. Therefore, we configure the threshold M as 10. The experiment results on different detection tasks below demonstrate that such setting is robust and effective.