Tight-sketch
A high-performance sketch for heavy item-oriented data stream mining with limited memory size

Citation for published version:

Digital Object Identifier (DOI):
10.1145/3583780.3615080

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
CIKM '23: Proceedings of the 32nd ACM International Conference on Information and Knowledge Management

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
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 items 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 over-estimation). 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 → Data stream mining.

KEYWORDS

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIKM ’23, Oct. 21–25, 2023, Birmingham, United Kingdom
© 2023 Copyright held by the owner/authors. Publication rights licensed to ACM.
ACM ISBN 978-8-4007-0124-5/23/10...$15.00
https://doi.org/10.1145/3583780.3615080

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 objective 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 refer 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, 20, 22, 23].

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 [25, 26] 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 [27, 28], 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 [29–31].

(ii) Moreover, items that appear in data streams usually follow highly skewed distributions [32, 33], 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, 34]. Overestimation is particularly detrimental and brings non-negligible performance degradation in many cases [29]. Consider two typical scenarios where overestimation has a negative impact: (i) 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 [35]; (2) in distributed machine learning, optimization approaches such as stochastic gradient descent (SGD) [36] 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 [29].

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 introduce 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 [37–40]. 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 \( \epsilon \), 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 [41, 42] employs 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 [34] 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 [25] uses a two-dimensional array with \( r \) rows; each row has \( b \) buckets for tracking items hashed to these buckets [26]. 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 items 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 [26] 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 [22] partitions the sketch into a heavy and a light part, to record the information of heavy and non-heavy items, respectively. CocoSketch [44] 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 \( \eta \), 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 [48]. 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 naive 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 \( \alpha \) and \( \beta \) 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 [46] 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 [56] trace with 0.64M items and set the thresholds \( \epsilon \) and \( \eta \) for heavy hitter detection and persistent item lookup as

![Figure 1: Wrong replacement events and detection accuracy with state-of-the-art sketches, under different memory sizes.](image)

(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 [27]). The impact of memory size on detection accuracy is illustrated in Figure 1(b), which shows that MV-Sketch’s F1 score is 5.4x 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.

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 [32, 33, 50, 51]. 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.

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 [32, 37, 50, 52, 53]. 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 [37]. 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...
PERSISTENCE.

Distribution

There mainly exist two types of data structures in current sketches:

- \( B \) of the candidate item;
- \( j \) of the row.

Each row is associated with a different pairwise-independent hash function, denoted as \( h_j \). The key field of the mapped bucket will be set as \( e.k \), and both counters will be increased by 1. However, if a different item already occupies the bucket, it indicates that item \( e \) is unable to be stored in the first row, due to hash collisions. In this case, Tight-Sketch will iteratively check the remaining rows using the hash functions \( h_2, \ldots, h_r \) 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, 26]. 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., \( M = 10 \)). In contrast, if the value counter of a bucket is less than \( M \), the counter is decreased with a higher rate of \( \frac{1}{(q+1)(q+2)} \) (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 \( \frac{1}{(q+1)(q+2)(q+3)} \) 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).
We apply Tight-Sketch to four different detection tasks: heavy hitter detection, heavy changer detection, persistent item lookup, and significant item lookup.

### 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.

#### 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 $\delta$, 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.

#### 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.

### 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 [21].

#### 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.

---

**Algorithm 1: Tight-Sketch’s Update Procedure**

**Input:** a newly incoming item $e$, hash function associated with each row $h_1, ..., h_n$, $min \leftarrow +\infty$

1. **Initialization:** Each bucket’s counters and item key are initialized to 0 and null, respectively.

2. **Stage I:** locating an available bucket
   - for $i = 1$ to $n$
     1. if $B(i,h_i(e)) == null$ | $B(i,h_i(e)) > min$ then
       1. $B(i,h_i(e)) == e,k$
   - return;

3. **Stage II:** probabilistic decay
   - if $B(i,h_i(e)) < min$ then
     1. $min \leftarrow B(i,h_i(e));$
     2. $p \leftarrow i; q \leftarrow h_i(e,k);$  
     3. $B(i,h_i(e)).a \leftarrow max(B(i,h_i(e)).a - 1, 0);$  
   - return;

4. **Initialization:** Each bucket’s counters and item key are initialized to 0 and null, respectively.

5. **Stage I:** locating an available bucket
   - for $i = 1$ to $n$
     1. if $B(i,h_i(e)) == null$ | $B(i,h_i(e)) > min$ then
       1. $B(i,h_i(e)) == e,k$
   - return;

6. **Stage II:** probabilistic decay
   - if $B(p,q).c < M$ then
     1. if random(0,1) < $\frac{1}{M(p,q).c}$ then
       1. $B(p,q).c = B(p,q).c - 1$
   - else if random(0,1) < $\frac{1}{M(p,q).c}$ then
     1. $B(p,q).c = B(p,q).c - 1$
   - return;

7. **Stage III:** probabilistic decay
   - if $B(p,q).c == 0$ then
     1. $B(p,q).k \leftarrow e,k;$
     2. $B(p,q).c \leftarrow B(p,q).c + 1;$
   - return;

8. **Stage IV:** probabilistic decay
   - if $B(p,q).c < 0$ then
     1. $B(p,q).c \leftarrow B(p,q).c - 1$
   - return;
until the detection task ends. Given a small positive number \( \sigma \) and a heavy item \( e \) with frequency \( S(e) \), \( \Pr \left[ S(e) - \hat{S}(e) \geq \lfloor \sigma N \rfloor \right] \leq \frac{\Delta}{N} \left[ \ln(S(e)) + L \right] \) holds, where \( \Delta \) is the fraction 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 \( e \) has entered the 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 \( 1 \leq x \leq Q_{i,j} \). 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 \( \sigma \), we attain

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

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

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

We use \( \psi \) to denote the value of the value counter of bucket \( B(i, j) \) when the detection task starts. 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,

\[
E(R_{i,j,x}|\psi) = \sum_{\psi} \frac{\psi}{\psi (\sigma + a)} + 1,
\]

where \( c \) and \( a \) represent the value counter and sustained arrival strength counter of bucket \( B(i, j) \), and \( \delta \) 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

\[
E(R_{i,j,x}|Q_{i,j}) = \sum_{\psi=0}^{S(e)-Q_{i,j}} \frac{\psi}{\psi (\sigma + a)} + 1.
\]

where \( b \) is the number of buckets in each row. Then we get

\[
\begin{align*}
\Pr \left[ \sum_{x=1}^{Q_{i,j}} R_{i,j,x} \leq \frac{S(e)}{\sigma} \right] & \leq \sum_{x=1}^{\frac{S(e)}{\sigma}} \frac{\sum_{x=1}^{Q_{i,j}} R_{i,j,x}}{\sigma N} = \exp \left[ -\frac{\sigma}{b} Q_{i,j} \right] + \sum_{x=1}^{Q_{i,j}} \frac{\exp \left[ -\frac{\sigma}{b} Q_{i,j} \right]}{Q_{i,j}!} \\
& \leq \sum_{x=1}^{Q_{i,j}} \frac{\exp \left[ -\frac{\sigma}{b} Q_{i,j} \right]}{Q_{i,j}!} + \sum_{x=1}^{Q_{i,j}} \frac{\exp \left[ -\frac{\sigma}{b} Q_{i,j} \right]}{Q_{i,j}!} \left( \frac{S(e)}{\sigma} - 1 \right)
\end{align*}
\]

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 [49]. Finally, we get the underestimation error bound as

\[
\Pr \left[ S(e) - \hat{S}(e) \geq \lfloor \sigma N \rfloor \right] \leq \frac{\sum_{x=1}^{Q_{i,j}} R_{i,j,x}}{\sigma N} \leq \frac{\delta}{\sigma N} \left[ \ln(S(e)) + L \right].
\]

\[\square\]

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 [56], 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 [57], which presents traffic traces collected by MAWI in Japan. We select a trace with 2.75M items from 2020; (iii) Campus [58], 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) [44],
For Tight-Sketch, we set the number of rows as 4 \cite{6, 60} and alter $b$ based on memory budgets. We default to select the threshold that keeps the hot items around 100 for each detection task \cite{46}. An analysis on configuring the parameter $M$ can be found in \cite{21}.

**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: $\frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}}$; (iv) Average Relative Error (ARE): $\frac{1}{|\Phi|} \sum_{e \in \Phi} \frac{|\hat{S}(e) - S(e)|}{S(e)}$, 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 \cite{46} 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 CocoSketch, which conduct admission operations based on probability, a series of advanced works also follow probabilistic replacement, namely HeavyKeeper [7], UA-Sketch [55], and PRECISION [61]. 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 [47] 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>
<thead>
<tr>
<th>Memory (KB)</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>MV-Sketch</td>
<td>161659</td>
<td>41690</td>
<td>5949</td>
<td>957</td>
<td>220</td>
</tr>
<tr>
<td>Tight-Sketch</td>
<td>64</td>
<td>34</td>
<td>15</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

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

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 $\alpha$ and $\beta$ 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 $\epsilon$ and $\eta$ 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 $\epsilon$ 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
decreases as \( \eta \) 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 \( \eta \) is set to 0.7. These results highlight the robustness of Tight-Sketch under a range of thresholds.

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

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

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Tight</th>
<th>MV</th>
<th>Coco</th>
<th>RAP</th>
<th>Elastic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Time (ms)</td>
<td>29.073</td>
<td>161.481</td>
<td>105.069</td>
<td>655.831</td>
<td></td>
</tr>
</tbody>
</table>

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.

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 4 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, CAIDA 2018).

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Tight</th>
<th>MV</th>
<th>Elastic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Time (ms)</td>
<td>29.073</td>
<td>161.481</td>
<td>105.069</td>
</tr>
</tbody>
</table>

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_cmove_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.

<table>
<thead>
<tr>
<th>Memory Size (KB)</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tight-SIMD</td>
<td>24.2</td>
<td>24.3</td>
<td>24.6</td>
<td>24.8</td>
<td>25.4</td>
</tr>
<tr>
<td>Tight-Sketch</td>
<td>17.8</td>
<td>18.1</td>
<td>18.5</td>
<td>19</td>
<td>19.4</td>
</tr>
</tbody>
</table>

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


