Efficient Sketching for Heavy Item-Oriented Data Stream Mining With Memory Constraints

Weihe Li and Paul Patras, *Senior Member, IEEE*

***Abstract*—Accurate and fast data stream mining is critical to many tasks, including real-time series analysis for mobile sensor data, big data management and machine learning. Various heavy-oriented item detection tasks, such as identifying heavy hitters, heavy changers, persistent items, and significant items, have garnered considerable attention from both industry and academia. Unfortunately, as data stream speeds continue to increase and the available memory, particularly in L1 cache, remains limited for real-time processing, existing schemes face challenges in simultaneously achieving high detection accuracy, 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. Recognizing that, in practice, most items are cold (non-heavy/persistent/significant), we implement distinct eviction strategies for different item types. This approach allows us to swiftly discard potentially cold items while offering enhanced protection to hot ones (heavy/persistent/significant). Additionally, we introduce an eviction method based on stochastic decay, ensuring that Tight-Sketch incurs only small one-sided errors without overestimation. To further enhance detection accuracy under extremely constrained memory allocations, we introduce Tight-Opt, a variant incorporating two optimization strategies. We conduct extensive experiments across various detection tasks to demonstrate that Tight-Sketch significantly outperforms existing methods in terms of both accuracy and update speed. Furthermore, by utilizing Single Instruction Multiple Data (SIMD) instructions, we enhance Tight-Sketch's update throughput by up to 36%. We also implement Tight-Sketch on FPGA to validate its practicality and low resource overhead in hardware deployments.**

***Index Terms*—Data stream mining, heavy item, persistent item, significant item, sustained arrival strength, network measurements.**

I. Introduction

MASSIVE data transmission has become a salient characteristic of social networks [2], financial services [3], healthcare systems [4], autonomous vehicles [5], smart city infrastructures [6], and many other areas. Such data streams convey valuable information that can be useful to a range of applications, including business intelligence [7], anomaly detection [8], and recommendation systems [9]. One important objective in stream mining is the identification of heavy items, which spans heavy hitter detection [10], [11], [12], [13], [14], heavy changer detection [15], persistent item lookup [16], [17], [18], and significant item lookup [19]. *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.

Detecting these distinct item types is of paramount importance in real-world scenarios. For example, heavy hitter detection can play an essential role in traffic engineering. By identifying the most resource-intensive services, network operators can optimize traffic routing for enhanced network efficiency. Additionally, real-time identification of heavy changers enables operators to effectively allocate network resources, ensuring that high-priority services receive sufficient bandwidth while minimizing service disruptions during network fluctuations. Moreover, persistent item lookup can significantly enhance the overall user experience. For example, when it is recognized that a particular application is persistently consuming data in the background, network providers can offer users the option to control or limit their data usage, which helps users avoid unexpected overcharging. Besides, in network security, significant item lookup allows operators to quickly detect and respond to potential DDoS attacks and abnormal network behavior [19].

However, 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. To overcome this obstacle, approximate stream mining leveraging probabilistic data structures such as ***sketches*** has attracted much interest [10], [12], [20], [21], [22], [23].

A. Limitations of Existing Approaches

Although numerous sketch-based approaches have been introduced for various detection tasks, mining contemporary ultra-fast data streams still presents substantial challenges to existing algorithms. The primary limitations of these current approaches are outlined as follows:

i) Many sketches, such as Count Sketch [24] and Count-Min Sketch [25], are non-invertible, requiring a full scan of the item stream to retrieve all hot items. This approach

Received 3 November 2024; revised 16 April 2025; accepted 24 August 2025. Date of publication 2 September 2025; date of current version 10 October 2025. An earlier version of this paper was presented at the ACM Conference on Information and Knowledge Management (CIKM), 2023 [DOI: 10.1145/3583780.3615080]. Recommended for acceptance by S. He. *(Corresponding author: Weihe Li.)*

The authors are with the School of Informatics, University of Edinburgh, EH8 9YL Edinburgh, U.K. (e-mail: weihe.li@ed.ac.uk; paul.patras@ed.ac.uk).

Digital Object Identifier 10.1109/TC.2025.3604467

0018-9340 © 2025 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.

incurs significant memory access overhead and reduces throughput. Existing invertible sketches often rely on additional data structures, like heaps, or complex coding and decoding processes [18], leading to redundant memory operations and high computational costs.

ii) For faster processing, an ideal sketch should have efficient update and query mechanisms, utilizing CPU caches effectively when handling high-speed data streams [20]. CPU cache memory is divided into three levels: L1, L2, and L3, with L1 being the fastest but smallest, typically between 8KB and 64KB [26]. Thus, compact sketches that fit within these cache constraints are essential.

Compact and fast sketches are highly beneficial in practical applications. For instance, in network traffic monitoring, they enable quick packet classification and counting, boosting throughput and response time when stored in the L1 cache. Similarly, in spam filtering, an efficient sketch in the L1 cache enhances accuracy and speed in spam detection. Since L1 caches have strict memory limitations, algorithms optimized for these conditions demonstrate robustness across various memory configurations. Using smaller memory portions not only conserves resources but also speeds up query times for retrieving heavy items.

iii) Moreover, items that appear in data streams usually follow highly *skewed* distributions [27], 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.

B. The Proposed Method

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. 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. Specifically, Tight-Sketch encompasses four 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.

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 [28]. 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.

iv) To further enhance detection accuracy, we introduce a new variant of Tight-Sketch called Tight-Opt, which incorporates two optimization policies. The first policy involves using fingerprints to compress item keys, while the second policy provides an additional opportunities for items to retain their position in the sketch even when displaced by incoming items, rather than immediately discarding them. By implementing these optimizations, Tight-Opt improves detection accuracy, particularly in scenarios with extremely limited memory (16KB), at the cost of decreased update speeds. We explore accuracy and update speed tradeoffs that inform users' decisions on selecting the most suitable sketch version.

Key Advantages: Backed by extensive experiments we conducted to validate the effectiveness of our approach, we summarize its key benefits as follows: (i) Tight-Sketch excels in accuracy and processing speed for various detection tasks. For instance, heavy hitter detection with 16KB memory achieves an average F1 score close to 1, up to 24$\times$ higher than that of existing methods; (ii) Tight-Sketch achieves high update throughput without relying on pointers or additional data structures. Redundant hash operations are abandoned once an item secures an available bucket during the update process; (iii) To further enhance processing speed, Tight-Sketch leverages SIMD instructions, leading to a 36% increase in update throughput; (iv) By incorporating two optimization methods, Tight-Opt enhances detection accuracy, especially under extremely limited memory constraints (16KB), though at the cost of reduced processing speed. In practical scenarios where accuracy is prioritized over speed, users may prefer Tight-Opt as their default choice. (v) Finally, we implement Tight-Sketch on FPGA to demonstrate its low resource overhead and practical viability in hardware environments.

II. Problem Definition and Background

In this section, we introduce the definition and existing work for four typical detection tasks: heavy hitter detection, heavy changer detection, persistent item lookup, and significant item lookup. Then, we reveal that existing methods are challenging to achieve high accuracy and fast update speed under tight memory size, which in turn motivates our design.

A. Heavy Item Detection

1) Definition: Heavy items include heavy hitters and heavy changers. Let $S(e)$ denote the frequency or size of item e, S

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.

represent the frequency or total size of all items. Given a predefined threshold ϵ, if $S(e) \geq \epsilon S$ $(0 < \epsilon < 1)$, 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) 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 [29], [30] 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. Lossy counting [31] first separates the data stream into fixed-size windows. Then it processes each window sequentially and maintains a counter for each item. The algorithm evicts items of minor frequency at the end of each window from the table. RAP [32] 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 [10] adopts the majority vote algorithm to track heavy items. HeavyKeeper [12] evicts items from the sketch by obeying an exponential decay strategy. Literature [11] proposes an efficient and optimal ϵ-LDP mechanism, known as the Wheel mechanism, for set-valued distribution estimation and heavy-hitter identification. 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 [33] 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.

B. Persistent Item Detection

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 $P(e) \geq \eta N$ $(0 < \eta \leq 1)$, item e is persistent.

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

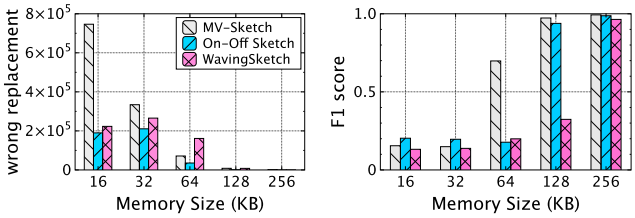
Sample-based methods such as Small-Space [17] record persistent items with a probability and track them into a hash table. Chen et al. introduce adaptive sampling to track persistent items without knowing the monitoring time horizon [34]. Even though such approaches seek to alleviate memory usage via 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 [18], 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 Count-min Sketch with a Bloom filter [35], leverage the Bloom filter to eliminate duplicates within a time window and then employ Count-min Sketch to track each item's persistence. However, the Bloom filter introduces significant false positive errors in tight memory settings, and the non-invertibility of Count-min Sketch results in slow update speeds. On-Off Sketch [16] 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.

C. Significant Item Detection

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 [19]. Given a threshold G $(G > 0)$, an item e is considered to be a significant item if $G(e) \geq G$.

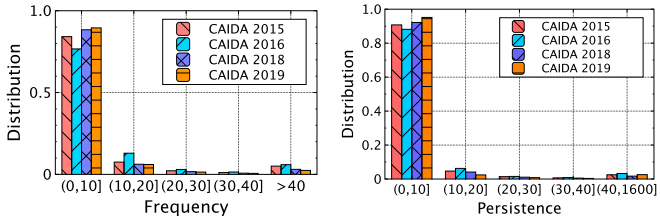
2) Related Work: A conventional approach to identifying significant items involves using two separate algorithms for tracking frequent and persistent items. However, this method comes with substantial time and space overhead, which is the combined overhead of both algorithms. Moreover, because these two data structures are independent, this approach tends to record many items that are either frequent or persistent, leading to inefficient use of limited memory. Long-Tail Clock (LTC) [19] leverages two essential techniques, Long-tail Restoring

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.



(a) Number of wrong replacements. (b) F1 score.

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



(a) Frequency. (b) Persistence.

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

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.

D. 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 [10] for heavy hitter detection, and On-Off Sketch [16] and WavingSketch [21] for persistent item lookup. We vary the memory size from 16KB to 256KB [36] 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 [37] 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. Fig. 1(a) demonstrates that when the memory size is tight ($\leq$ 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). The impact of memory size on detection accuracy is illustrated in Fig. 1(b), which shows that MV-Sketch's F1 score is 5.4$\times$ 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, while maintaining fast update speeds.

III. Tight-Sketch Design

In this section, we begin with a data analysis, uncovering the two fundamental design principles that underlie Tight-Sketch. We then proceed to introduce the data structure employed by Tight-Sketch and its core operations, including update and query. Additionally, we propose two optimization strategies aimed at further enhancing the detection accuracy of our proposed method.

A. Design Rules

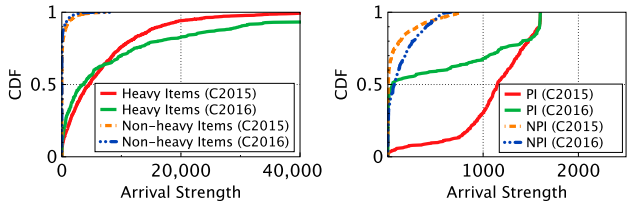
Rule 1: The distribution of items within real data streams is often heavily skewed, with the majority being small and only a minuscule fraction being large [27].

Here, 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 Fig. 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 [16], 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 (Fig. 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 [27]. 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. To verify this property, we utilize MV-Sketch [10] and WavingSketch [21] 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 [16]. 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. Fig. 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.

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

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.



(a) Heavy item detection. C→CAIDA.(b) Persistent item detection. (N)PI→(non-)persistent items.

Fig. 3. Sustained arrival strength of hot and cold items.

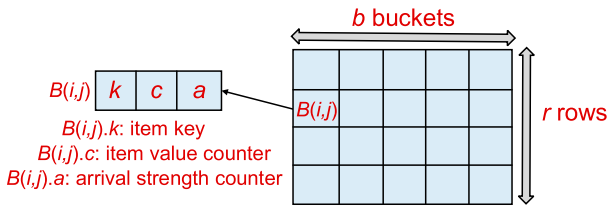


Fig. 4. Tight-Sketch's data structure.

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.

B. Data Structure

There mainly exist two types of data structures in current sketches: flat [25] and hierarchical [38]. Instead of the sophisticated hierarchical structure with multiple layers, we choose the classic flat structure for Tight-Sketch, since it bears faster processing speed and it is easier to deploy in practice.

As illustrated in Fig. 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.

C. 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.

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.

Algorithm 1: Tight-Sketch's Update Procedure.

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

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

`// Stage I: locating an available bucket`

2 **for** $i = 1$ *to* r **do**

3 **if** $B(i,h_i(e)).k == null \ || \ B(i,h_i(e)).k == e.k$ **then**

4 $B(i,h_i(e)).k \leftarrow e.k$;

5 $B(i,h_i(e)).c \leftarrow B(i,h_i(e)).c + 1$;

6 $B(i,h_i(e)).a \leftarrow B(i,h_i(e)).a + 1$;

7 **return**;

8 **else if** $B(i,h_i(e)).c < min$ **then**

9 $min \leftarrow B(i,h_i(e)).c$;

10 $p \leftarrow i$; $q \leftarrow h_i(e)$;

11 $B(i,h_i(e)).a \leftarrow max(B(i,h_i(e)).a - 1, 0)$;

`// Stage II: probabilistic decay`

12 **if** $B(p,q).c < M$ **then**

13 **if** $random(0,1) < \frac{1}{B(p,q).c+1}$ **then**

14 $B(p,q).c = B(p,q).c - 1$

15 **else if** $random(0,1) < \frac{1}{B(p,q).c \times B(p,q).a+1}$ **then**

16 $B(p,q).c = B(p,q).c - 1$

17 **if** $B(p,q).c == 0$ **then**

18 $B(p,q).k \leftarrow e.k$;

19 $B(p,q).c \leftarrow B(p,q).c + 1$;

20 **return**;

21 **else**

22 Discard the incoming item e;

23 **return**;

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 [10] and HeavyKeeper [12], 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

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.

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 [17], [25], [29]. 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 (see Section V-G). 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).

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.

D. Utilizing Tight-Sketch for Various Tasks

We employ Tight-Sketch for four distinct detection tasks: heavy hitter detection, heavy changer detection, persistent item lookup, and significant item lookup.

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 III-B and III-C.

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 III-C. 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) 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 [16]. 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 III-C.

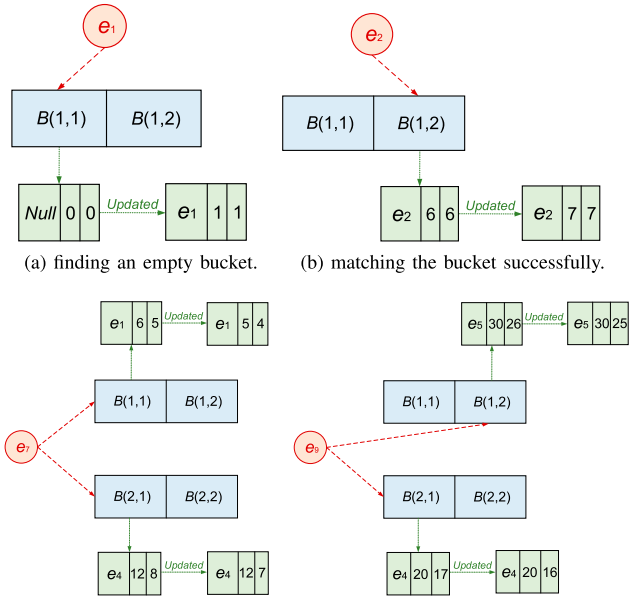
4) Significant Item Lookup: To identify significant items, Tight-Sketch needs to track the frequency and persistence of each item. To accomplish this, we modify the data structure in bucket $B(i, j)$ to include the following fields: k, which indicates the item identifier; fc, a value counter for item 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.

5) Running Examples: We use heavy hitter detection as an example and present multiple scenarios to illustrate the update process of Tight-Sketch. In these examples, we set M to 10.

Case 1: As illustrated in Fig. 5(a), when a new item e_1 arrives, it initially employs the hash function h_1 to determine the location of a bucket in the first row. In this case, the hash result directs item e_1 to bucket $B(1, 1)$. Upon inspection, it is found that the targeted bucket is empty. Consequently, item e_1 is successfully inserted into the bucket, and the bucket's information is updated from $(Null, 0, 0)$ to $(e_1, 1, 1)$.

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.



(a) finding an empty bucket. (b) matching the bucket successfully.

(c) attempting to replace (without considering the arrival strength). (d) attempting to replace (considering the arrival strength).

Fig. 5. Running examples.

Case 2: As illustrated in Fig. 5(b), when an incoming item e_2 locates a bucket $B(1,2)$ in the first row, it is determined that e_2 is already being tracked there. As a result, the bucket's information is updated from $(e_2, 6, 6)$ to $(e_2, 7, 7)$. The equivalence between the item's counter value and the arrival strength counter indicates that the item e_2 is entering the bucket in a continuous manner.

Case 3: As observed in Fig. 5(c), when item e_7 arrives, it is hashed to two buckets, $B(1,1)$ and $B(2,1)$, in the respective rows using hash functions h_1 and h_2. However, these hashed buckets are already occupied by items e_1 and e_4. Therefore, e_7 disrupts the continuous arrival of the tracked items, causing their individual arrival strength counters to decrement by 1. For instance, the arrival strength counter of item e_1 decreases from 5 to 4. Subsequently, e_7 seeks the bucket with the lowest counter value, which in this case is $B(1,1)$. Given that the value counter is smaller than the threshold M, the decay probability is computed as $\frac{1}{6+1}$ without taking into account the additional protection offered by the arrival strength. Fortunately, the decay operation proves successful, resulting in the value counter of e_1 being updated to 5. In the event of unsuccessful decay, the respective value counter remains unaltered. However, since the value counter of e_1 does not reach 0 due to decay, item e_7 is discarded.

Case 4: When item e_9 arrives (Fig. 5(d)), it encounters a situation where no available bucket is vacant. In this scenario, it selects the bucket with the lowest value counter to undergo decay, which is $B(2,1)$. Since the value counter exceeds the threshold M, the probability of decay is calculated as $\frac{1}{20\times17+1}$. Consequently, it becomes challenging for items with higher value counters to be evicted from the bucket.

Furthermore, the incorporation of arrival strength offers the additional advantage of mitigating the interference from bursty items. Typically, those exhibit a high-frequency arrival pattern within a short time frame, with most bursts belonging to cold items. With the aid of the arrival strength counter, even highly bursty items can be effectively removed from the bucket, leaving more space for genuine heavy items.

TABLE I
COLLISION RATE WITH VARYING FINGERPRINT (FP) LENGTHS

# of Items \ FP Lengths (Bits)	8	16	32
0.5M	0.986	0.018	3.41×10^{-7}
1M	0.999	0.037	6.83×10^{-7}
2M	1	0.071	1.37×10^{-6}
3M	1	0.107	2.05×10^{-6}

E. Tight-Sketch Optimization

Here, we introduce a variant of Tight-Sketch called Tight-Opt, which incorporates the following two optimizations.

1) Optimization 1: Compressing Keys With Fingerprints: As the default configuration, Tight-Sketch employs a key-centric approach for item tracking within each bucket. However, this approach can lead to increased memory usage in scenarios involving longer keys, such as 5-tuples in network data. To address this challenge, we introduce a fingerprint-centric optimization method that utilizes a hash function to compute a short sequence of bits based on the key, referred to as the *fingerprint* [39]. This approach helps conserve memory and increase the number of available buckets for item recording, but introduces the possibility of false detection when hash collisions occur between items. When hashing n items into b buckets, each associated with an x-bit fingerprint, as described in [39], [40], the probability of a hash collision can be expressed as:

$$Pr\{fingerprint\ collision\} = 1 - (1 - 2^{-x})^{\frac{n}{b}}.$$

With a configured memory size of 16KB for our method, Table I presents the collision rates for fingerprints with varying numbers of items and lengths. The results indicate that, when using a 32-bit fingerprint, the collision rate remains negligible even when processing 3M distinct items.

2) Optimization 2: Giving Items One More Chance: A heavy item may find all of its r hashed buckets already occupied by other heavy items with large counter fields, especially when the memory is limited. Hash collisions can also result in tracked heavy items being evicted from a bucket and thus a decline in detection accuracy. While Optimization 1 addresses the first issue by using fingerprints, which leaves space for more buckets, here we focus on the latter.

Table II illustrates the number of hash collisions between heavy items observed during the update process when employing the CAIDA 2015 trace, which encompasses 0.52 million distinct items, and where we adjust the threshold to regulate the quantity of heavy items in the 100–300 range. We observe that as the count of heavy items rises, the occurrence of hash collisions among heavy items also increases, particularly when memory resources are limited, which could potentially result in heavy items displacing each other. In particular, under a 16KB

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.

TABLE II
NUMBER OF HASH COLLISIONS BETWEEN HEAVY ITEMS, AS A FUNCTION OF MEMORY SIZE

# of Heavy Items \ Memory Size	16KB	32KB	64KB
100	62	0	0
200	277	1	0
300	2141	335	21

memory budget, the F1 score of our method decreases from 0.99 to 0.958 in this scenario.

As noted in [36], the majority of buckets are dedicated to tracking non-heavy items, constituting approximately 70% of the buckets when employing a 16KB memory allocation with the CAIDA trace. Consequently, to mitigate this second issue stemming from hash collisions among heavy items, we provide more opportunities to handle collision-heavy items by utilizing the remaining unused buckets or buckets that track potential non-heavy items. Specifically, when an incoming item displaces the tracked item within the bucket, we select its neighboring bucket as the alternative candidate, rather than immediately discarding it. If its neighboring bucket is vacant, the tracked item can be relocated to this new location. However, if this already contains another item, the tracked item makes an attempt to decrement the counter associated with that bucket. If the counter value of the neighboring bucket reaches 0 as a result, the tracked item can replace the item currently stored there. If unsuccessful, the tracked item will be evicted.

3) Trade-off Analysis: While the above optimizations improve the detection accuracy of the basic Tight-Sketch (see results in Section V-K for a quantitative analysis), they introduce additional complexity to the update process, resulting in a reduction in processing speed. Specifically, generating fingerprints requires additional hash operations, and rehashing items into different buckets further slows down the update throughput. In practical scenarios, if users prioritize processing speed, it may be preferable to use the basic Tight-Sketch over Tight-Opt, while when accuracy is to be prioritized over speed, Tight-Opt is a viable choice.

IV. 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 an example.

A. No Overestimation Error

Theorem 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 detailed proof can be found in [1]. □

B. Underestimation Error Bound

Theorem 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 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: The detailed proof can be found in [1]. □

V. EVALUATION

A. Setup

Implementation Platform: 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:

- CAIDA [37], 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.
- MAWI [41], which presents traffic traces collected by MAWI in Japan. We select a trace with 2.75M items from 2020.
- Campus [42], 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).

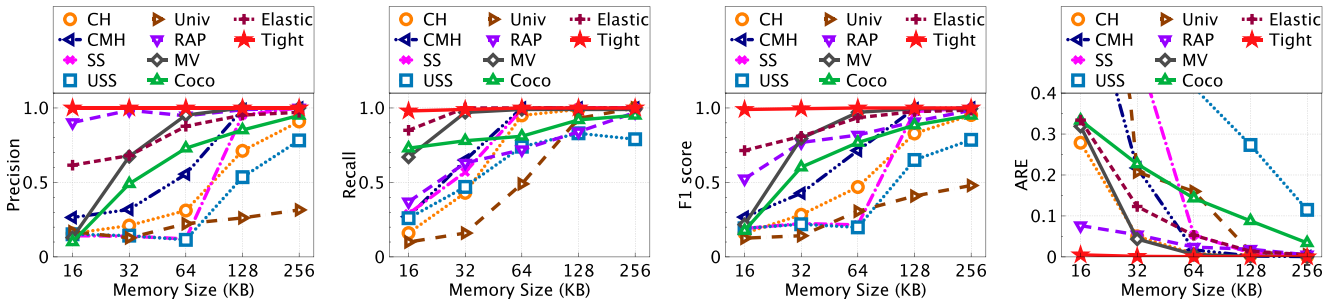
Benchmarks: For heavy item detection, we compare Tight-Sketch (Tight) with MV-Sketch (MV) [10], CocoSketch (Coco) [33], Elastic [22], RAP [32], USS [30], UnivMon (Univ) [43], CMHeap (CMH) [25], CountHeap (CH) [24] and Space-Saving (SS) [29]. For MV-Sketch, we configure the number of rows as 4 [10]. For RAP, we set the number of arrays as 2. The parameter settings of the rest of the schemes are consistent with [33]. In addition, for a comprehensive assessment, we also compare Tight-Sketch with the advanced probability-based methods HeavyKeeper [12], and PRECISION [44] in Section V-C1.

For persistent item lookup, we divide each trace into 1,600 time windows [16] and select two off-the-shelf benchmarks, On-Off Sketch (On-Off) [16] and WavingSketch (Waving) [21]. The number of cells for On-Off Sketch and WavingSketch is 16 [21].

For significant item lookup, we compare Tight-Sketch with LTC [19], using its default settings.

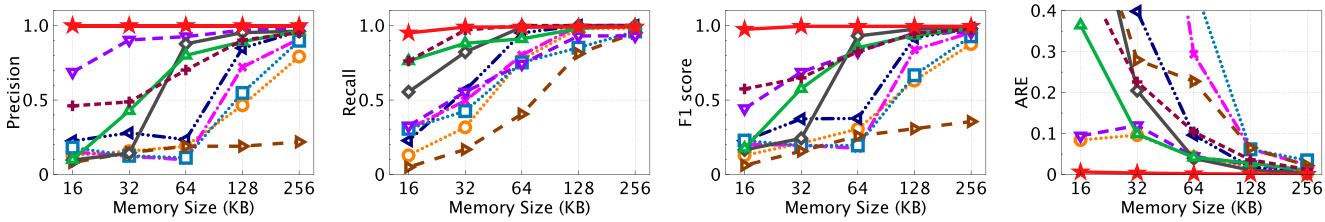
Memory Resource Allocation: Each algorithm has a specific bucket or slot size determined by its data structure design and the type of information it needs to store. In our experiments, we allocate different numbers of buckets to different algorithms under a fixed total memory constraint, ensuring that all methods operate within the same memory footprint for a fair comparison. This approach aligns with recent established practices in the field, such as those seen in [10], [12], [20]. We follow the standard practice of varying memory sizes from 16KB to 256KB, as recommended by [36], [45]. Specifically, in Tight-Sketch, each bucket has a fixed size. By specifying the number of rows (r)

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.



(a) Precision. (b) Recall. (c) F1 Score. (d) ARE.

Fig. 6. Heavy hitter detection with different approaches, as a function of memory size (CAIDA 2015).



(a) Precision. (b) Recall. (c) F1 Score. (d) ARE.

Fig. 7. Heavy hitter detection with different approaches, as a function of memory size (CAIDA 2018).

and the total available memory, we can accurately determine the number of buckets per row based on these parameters.

For methods like Space Saving [29], which require a minimum number of counters to avoid false negatives, we recognize this as an inherent limitation of the algorithm, especially under constrained memory conditions. Our experiments are designed to evaluate the performance of different algorithms under realistic memory constraints, which may not always permit ideal conditions for every solution. Hence, we consider this a characteristic of the algorithm. To offer a more balanced comparison, we also evaluate performance per KB in Section V-C, demonstrating the effectiveness of our method.

Number of Hot Items: In line with [10], we adjust the threshold for each trace to consistently maintain around 100 hot items for each detection task. Additionally, we analyze the effect of varying this threshold to assess the robustness of our approach in Section V-H.

Metrics: We use the following five performance metrics.

- Recall: fraction of true reported items over all true items.
- Precision: fraction of true reported items over all reported items.
- F1 score: $\frac{2\times recall\times precision}{recall+precision}$.
- 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.
- Update throughput: the update speed of the algorithm, in millions of operations per second (Mops).

B. Parameter Setting

Similar to existing work on parameter settings [20], [21], [45], 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.

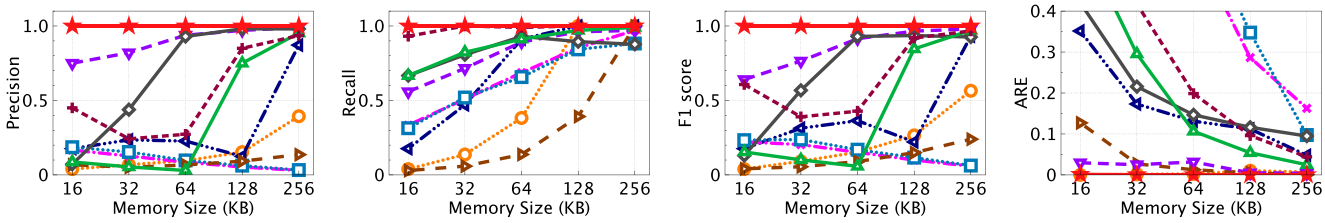
C. Performance on Heavy Hitter Detection

We compare the performance of Tight-Sketch with existing approaches on heavy hitter detection. Figs. 6–9 detail this across different datasets.

Precision (Figs. 6(a)–9(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.

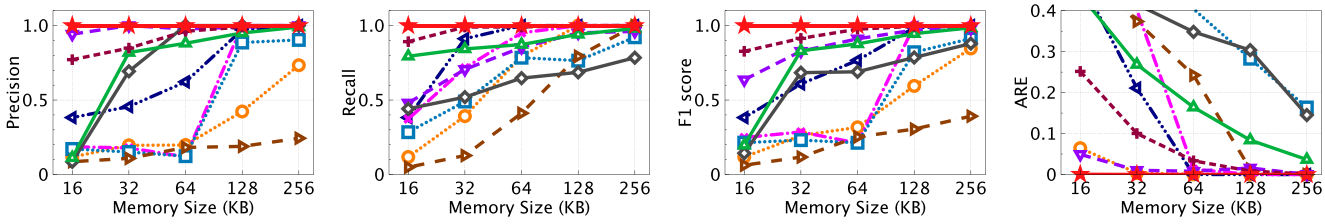
Recall (Figs. 6(b)–9(b)): Tight-Sketch maintains its optimality in terms of recall on different traces, with an improvement of up to 85% across the CAIDA 2015 trace, 106% across the CAIDA 2018 trace, 209% across the MAWI trace,

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.



(a) Precision. (b) Recall. (c) F1 Score. (d) ARE.

Fig. 8. Heavy hitter detection with different approaches, as a function of memory size (MAWI).

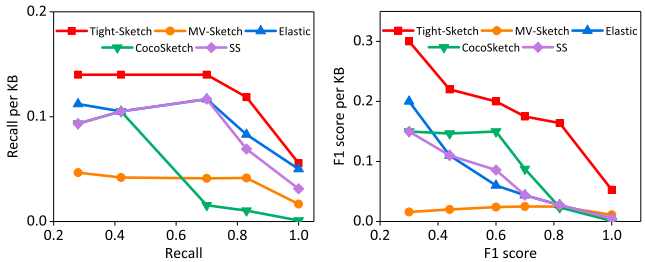


(a) Precision. (b) Recall. (c) F1 Score. (d) ARE.

Fig. 9. Heavy hitter detection with different approaches, as a function of memory size (campus).

TABLE III
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



(a) Recall per KB. (b) F1 score per KB

Fig. 10. Performance metrics per KB for various methods on the CAIDA 2015 dataset.

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 (Figs. 6(c)–9(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 (Figs. 6(d)–9(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.

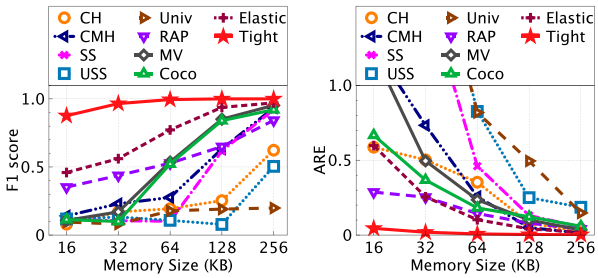
1) Deep Dive:

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 III, 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) We assess the performance of various methods using the accuracy per byte metric, for heavy item lookup. Specifically, we use the CAIDA 2015 trace and compare our Tight-Sketch method with MV-Sketch, Elastic, CocoSketch, and Space Saving for identifying the top-100 heavy items.

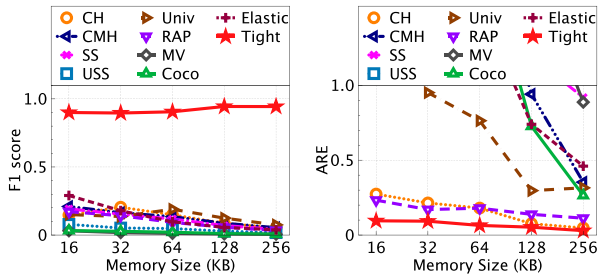
As illustrated in Fig. 10(a), our method consistently maintains the highest recall per KB across various target recall values, demonstrating its effectiveness and memory efficiency. We exclude the precision per KB from this analysis since our method only experiences underestimation errors. This ensures that the heavy items we capture are always the real ones, resulting in a constant precision of 1 across all memory budgets, which is superior to all baselines. Fig. 10(b) shows that our method also achieves the highest F1 score per KB. Similar trends are observed for other metrics, such as (1 - ARE)/byte.

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.



(a) F1 Score. (b) ARE.

Fig. 11. Detailed performance for heavy changer detection across the CAIDA 2015 dataset.



(a) F1 Score. (b) ARE.

Fig. 12. Detailed performance for heavy changer detection across the CAIDA 2015 dataset.

iii) To validate Tight-Sketch's superior performance and verify statistical significance of the results, we conduct further evaluations comparing its performance to that of established baselines across five additional CAIDA datasets, labeled as Traces #1 through #5. These traces vary in size: Trace #1 contains 730,000 items, Trace #2 comprises 1.53 million items, while Traces #3, #4, and #5 include 450,000, 640,000, and 1.29 million items, respectively. For these evaluations, we set the target recall and F1 score to 0.9 and assess the performance per KB of various methods. We exclude precision from our analysis, as our method consistently achieves a precision of 1 regardless of memory constraints.

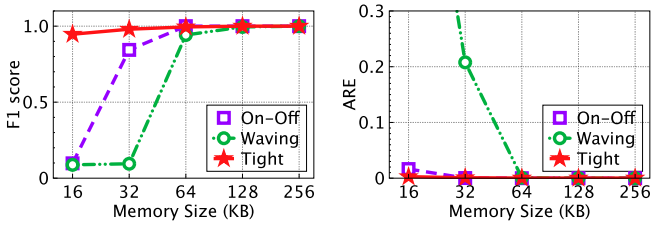
Our method achieves recall per KB of 0.06, 0.1, 0.09, 0.09, and 0.08 for Traces #1–#5, respectively. In comparison, MV-Sketch's recall per KB is 0.013, 0.014, 0.026, 0.012, and 0.0072 for the same traces. These results demonstrate our method's significantly higher recall per KB. This superiority is maintained when compared to other baselines. Regarding F1 score per KB, our method achieves 0.1, 0.1125, 0.1125, 0.1125, and 0.1, while MV-Sketch attains 0.013, 0.014, 0.026, 0.012, and 0.0072, which are lower than our method's.

We further conduct paired t-tests comparing Tight-Sketch's performance against that of MV-Sketch across five CAIDA datasets. For recall per KB, Tight-Sketch consistently outperforms MV-Sketch, with a mean difference of 0.06956. The t-test yields a t-statistic of 10.25 (df = 4), resulting in a p-value less than 0.001. Similarly, for F1 score per KB, Tight-Sketch demonstrates significantly better performance, namely a mean difference of 0.09306, with a t-statistic of 31.69 (df = 4), yielding a p-value less than 0.0001. The extremely low p-values for both metrics indicate that the observed differences in performance are statistically significant.

D. Performance on Heavy Changer Detection

Figs. 11 and 12 provide a detailed performance analysis of different approaches for heavy changer detection across various network traces.

The results depicted in Fig. 11(a) and Fig. 12(a) demonstrate that Tight-Sketch achieves an average F1 score that is 30.69% and 594.04% higher than that of the most competitive approach, Elastic, when applied to the CAIDA 2015 and MAWI datasets, respectively. Since 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. Moreover, as illustrated in Fig. 11(b) and Fig. 12(b), Tight-Sketch significantly reduces estimation errors, further confirming its superiority.



(a) F1 Score. (b) ARE.

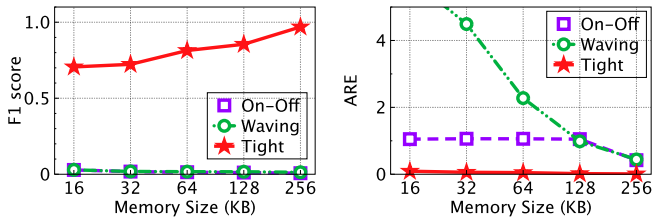
Fig. 13. Detailed performance for heavy changer detection across the MAWI dataset.

E. Performance on Persistent Item Detection

Here, we assess the performance of various approaches for persistent item lookup. Figs. 13(a) and 14(a) highlight Tight-Sketch's superior F1 score. Specifically, we observe a 25% improvement over On-Off Sketch on the CAIDA 2015 trace and a 5163% enhancement on the MAWI trace. Furthermore, Tight-Sketch achieves the lowest estimation errors, as Figs. 13(b) and 14(b) reveal. For instance, when applied to the MAWI trace, Tight-Sketch reduces the estimation error by an average factor of 22.31 compared to On-Off Sketch.

Besides, we employ an additional MAWI trace to conduct an extended assessment of Tight-Sketch and On-Off Sketch's F1 scores for detecting the top-100 persistent items under *larger memory allocations*. Table IV reveals that Tight-Sketch consistently outperforms On-Off Sketch in scenarios where the memory size ranges from 300KB to 500KB. This performance difference arises because the majority of items within the MAWI trace exhibit non-persistent behavior. The simplistic replacement strategy employed by On-Off Sketch results in the erroneous replacement of many persistent items with non-persistent ones, thereby reducing detection accuracy. Notably,

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.



(a) F1 Score. (b) ARE.

Fig. 14. Detailed performance for persistent item detection across the CAIDA 2015 dataset.

TABLE IV
F1 SCORE OF PERSISTENT ITEM LOOKUP UNDER LARGER MEMORY ALLOCATIONS

Memory Size (KB)	300	350	400	450	500
Tight-Sketch	0.958	0.974	0.99	0.99	0.99
On-Off Sketch	0.006	0.0056	0.005	0.005	0.0045

we observe that On-Off Sketch attains an F1 score of 0.95 only when the memory size is increased to approximately 1.2MB.

F. Performance on Significant Item Detection

We configure the threshold values α and β to 1. Our findings in Fig. 15 demonstrate that Tight-Sketch consistently attains the highest level of detection accuracy, even when working with limited available memory. Remarkably, with a memory size constraint of 16KB, Tight-Sketch achieves an F1 score that surpasses that of the state-of-the-art LTC by an impressive margin of 178%, solidifying its superiority.

G. Analysis of Item Eviction Strategy

We investigate three alternative eviction strategies: minus, probability decay without considering arrival strength, and probability replacement. The minus method involves decreasing both the value and arrival strength counters by 1 when a hash collision occurs with an incoming item. When the value counter decays to 0, a new arrival replaces the current item in the bucket. Probability decay without considering arrival strength involves decaying the counter based only on the probability of $\frac{1}{B(p,q).c+1}$. Probability replacement directly replaces the incumbent item with a probability of $\frac{1}{B(p,q).c+1}$.

We take persistent item lookup as an example and utilize the MAWI trace to conduct a comparison. Results demonstrate that our approach achieves the highest F1 score across a range of memory sizes (16KB to 256KB). Compared with the minus, probability decay without considering arrival strength, and probability replacement methods, our Tight-Sketch approach shows improvements of 12.96%, 2.47%, and 16.5%, respectively, when the memory size is 16KB.

H. 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.

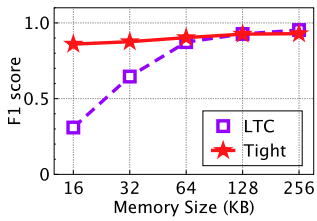
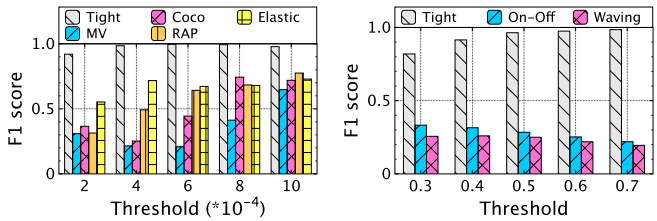


Fig. 15. F1 Score for significant item detection over the CAIDA 2015 trace.



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

Fig. 16. Detection accuracy under different thresholds (memory size: 32KB, CAIDA 2018).

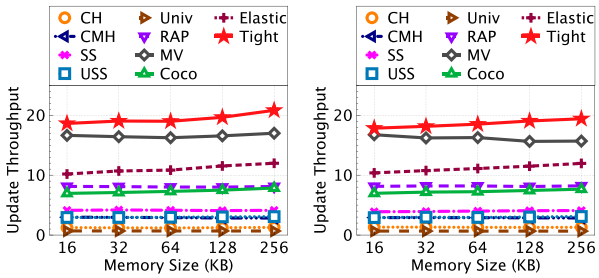
To do so, we fix the memory size at 32KB and vary the ϵ and η threshold values for heavy hitter detection and persistent item lookup, respectively, in the range of 0.0002-0.001 and 0.3-0.7. As shown in Fig. 16, Tight-Sketch is superior across a range of thresholds. In the case of heavy hitter detection, 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 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.

I. Update Throughput and Query Time

1) Update Speed: **Heavy Hitter Detection.** To assess the update speed of Tight-Sketch, we begin by examining its performance in heavy hitter detection. In Fig. 17, we compare the update throughput of various algorithms across different memory sizes. The results clearly indicate that Tight-Sketch achieves the highest update speed, surpassing MV-Sketch by 17% on the CAIDA 2015 trace and 15% on the CAIDA 2018 trace. This superior performance can be attributed to Tight-Sketch's straightforward update rule and the elimination of unnecessary hash operations, resulting in faster execution.

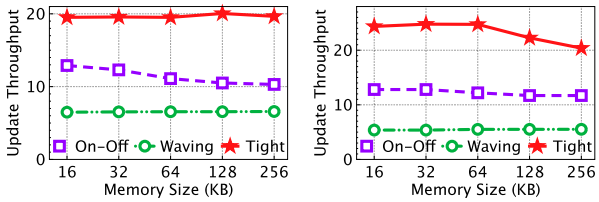
Persistent Item Lookup. Fig. 18 illustrates that Tight-Sketch sustains its rapid update speed for persistent item detection when compared to established approaches. Notably, when compared to On-Off Sketch, our method enhances the update throughput on average by 72% over the CAIDA 2015 trace and by 90.36% over the MAWI trace.

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.



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

Fig. 17. Update throughput (mops) for heavy item detection with different schemes across the CAIDA traces.



(a) CAIDA 2015. (b) MAWI.

Fig. 18. Update throughput (mops) for persistent item lookup with different schemes across different traces.

2) Query Time: In practice, long query times can result in a backlog of data, leading to performance degradation and an increased likelihood of errors. This is particularly true when dealing with real-time applications, where data must be processed and acted upon in real time. Delays in processing can lead to inaccurate or outdated results, which can significantly impact the performance and effectiveness of the application.

Here, we utilize the CAIDA 2015 trace to evaluate the query time of Tight-Sketch for heavy hitter detection. Table V 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.

J. 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. SIMD instructions are widely supported across modern CPU architectures, including x86 (Intel and AMD) and ARM platforms, making these optimizations applicable to a wide range of computing environments. In our implementation, we use Intel's AVX2 instruction set.

TABLE V
QUERY TIME FOR HEAVY ITEM DETECTION (32KB)

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

TABLE VI
TIGHT-SKETCH'S UPDATE THROUGHPUT (MOPS) FOR HEAVY ITEM DETECTION WITH SIMD OPTIMIZATION (CAIDA 2018)

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

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 VI presents a comparison of the update speed for heavy item detection using Tight-Sketch, both with and without SIMD instructions. The results reveal notable improvements, with up to a 36% boost in performance.

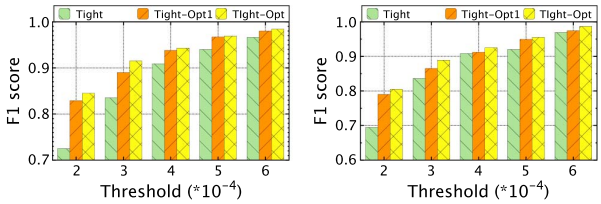
K. Performance of the Optimized Tight-Sketch

We evaluate the effectiveness of each optimization component by comparing different variants. Tight-Opt.1 represents Tight-Sketch with only the fingerprint-based optimization, while Tight-Opt includes both optimizations. We vary the heavy-hitter threshold from 2×10^{-4} to 6×10^{-4} and use the CAIDA 2015 and CAIDA 2018 traces for testing.

Fig. 19 shows that Tight-Opt consistently achieves higher detection accuracy compared to the original Tight-Sketch. For example, when the threshold is set to 3×10^{-4}, Tight-Opt achieves F1 scores that are 9.58% and 6.34% higher than Tight-Sketch on the CAIDA 2015 and 2018 traces, respectively. We also observe that both optimization strategies contribute to this improvement. Specifically, with a threshold of 2×10^{-4} on the CAIDA 2015 trace, the fingerprint compression increases the F1 score by 14.34%, while the additional bucket mechanism further improves it by 1.93%. These results highlight the effectiveness of the optimization strategies incorporated in Tight-Opt.

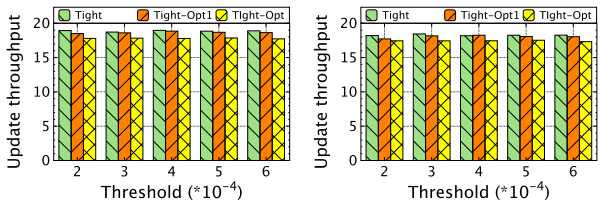
However, the performance increase in the update process of Tight-Opt comes at the cost of reduced update speed. As shown in Fig. 20, we compare the update speeds of Tight-Opt, Tight-Opt-1, and the original Tight-Sketch. The results indicate that Tight-Sketch achieves faster update speeds than Tight-Opt. For example, on the CAIDA 2015 trace with a threshold of 2×10^{-4}, Tight-Opt is around 6% slower than Tight-Sketch. Therefore, for scenarios where accuracy is prioritized over speed, Tight-Opt is a more suitable choice.

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.



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

Fig. 19. Comparison of the detection accuracy under different thresholds (memory size: 16KB).



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

Fig. 20. Comparison of the update speed under different thresholds (memory size: 16KB).

Deep Dive. We further compare the detection accuracy and update speed of Tight-Sketch and Tight-Opt with several recent approaches for heavy hitter detection, including HeavyGuardian, WavingSketch, and DHS. The evaluation is conducted using the CAIDA 2015 trace, with a memory size of 16KB and a heavy-hitter threshold set to 2×10^{-4}. For HeavyGuardian [39], WavingSketch [21], and DHS [40], we use the code provided in [40].

Table VII presents the F1 score and update throughput of various approaches. As shown, Tight-Opt achieves the highest F1 score, attributed to its fingerprint-based key compression and the strategy that gives items an additional chance to remain in the buckets. Regarding update throughput, the original Tight-Sketch is the fastest, benefiting from its elegant update strategy. The reduction in throughput for Tight-Opt is due to the added overhead of fingerprint computation and selecting an extra bucket upon successful eviction.

Additionally, we compare the performance of Tight-Opt with P-Sketch [45] for persistent flow detection. The number of rows in P-Sketch is set to 4. We divide the CAIDA 2015 trace into 1600 time windows and use a persistence threshold of 0.5. Under a memory constraint of 16KB, Tight-Opt achieves an F1 score of 0.937, compared to 0.856 for P-Sketch, resulting in a 9.46% improvement in accuracy. Overall, these results demonstrate the effectiveness of our methods.

L. Practical Deployment of Tight-Sketch

We implement a 2-row Tight-Sketch in Verilog and target the UltraScale+ XCU200-L2FSGD2104E FPGA. Due to the parallelism of FPGAs, we process incoming items hashed into two rows concurrently during deployment.

Specifically, Tight-Sketch's FPGA implementation employs a fully pipelined update process divided into four stages. In stage one, two independent CRC32 hash modules compute two distinct 32-bit hash values in parallel from the incoming item key to generate bucket addresses. In stage two, these addresses are applied to dual-port 128-bit BRAMs that store each bucket's content, which includes the key, a 32-bit value counter, and a 32-bit arrival continuity metric; note that each BRAM read requires two clock cycles. In stage three, the update logic evaluates each accessed bucket. If the bucket is empty or already contains the same key, both the counter and the arrival continuity are directly incremented. Otherwise, similar to [33], a 32-bit random number is generated and, using a multiplication-based probability check (that is, if the product of the random number and the counter, or the counter multiplied by the arrival continuity when the counter exceeds the threshold M, is less than 2^{32}), the counter is decremented. If the counter falls to zero, the bucket is updated with the new key and the counters are reset to one. Finally, in stage four, each row independently writes back the updated bucket content to its memory. Overall, this design is implemented in around 400 lines of Verilog code.

TABLE VII
COMPARISON OF F1 SCORE AND SPEED (IN MOPS) ACROSS DIFFERENT SCHEMES FOR HEAVY HITTER DETECTION

Scheme	Tight-Sketch	Tight-Opt	DHS	Waving	HeavyGuardian
F1 score	0.725	**0.845**	0.746	0.435	0.446
Speed	**18.716**	17.825	4.987	5.478	5.343

TABLE VIII
FPGA RESOURCE USAGE

Component	Used Quantity	Utilization
Look-Up Tables	2815	0.24%
LUTRAM	33	0.01%
Flip-Flops	1719	0.07%
Block RAM	384	17.78%
I/O Pins	67	9.91%

Table VIII lists the resource utilization of Tight-Sketch on the FPGA platform. We observe that Tight-Sketch consumes only a small fraction of the available Look-Up Tables (0.24%), LUTRAM (0.01%), and Flip-Flops (0.07%). Although 384 Block RAM blocks are used, corresponding to 17.78% utilization, and 67 I/O pins account for 9.91% utilization, these resources remain well within the device's limits. This low overhead leaves sufficient resources available for other tasks and confirms that Tight-Sketch is well-suited for high-speed, real-time item processing in practice.

VI. CONCLUSION

This paper presents Tight-Sketch, a new sketch designed for heavy-oriented tasks, offering high detection accuracy under limited memory and fast update speed. It uses a probabilistic decay strategy to selectively replace items in buckets based on bi-dimensional features. We apply Tight-Sketch to various heavy-based detection tasks and validate its effectiveness through extensive experiments on diverse datasets. Results show that Tight-Sketch consistently outperforms existing methods. We further improve its accuracy with two optimizations and boost update speed using SIMD instructions, making it well-suited for

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.

high-speed data streams. Moreover, our FPGA implementation confirms its low resource overhead in practice.

REFERENCES

[1] W. Li and P. Patras, "Tight-sketch: A high-performance sketch for heavy item-oriented data stream mining with limited memory size," in *Proc. ACM CIKM*, 2023, pp. 1328–1337.

[2] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *Proc. ACM SIGMOD*, 2016, pp. 1481–1496.

[3] B. Ball, M. Flood, H. V. Jagadish, J. Langsam, L. Raschid, and P. Wiriyathammabhum, "A flexible and extensible contract aggregation framework (CAF) for financial data stream analytics," in *Proc. ACM DSMM*, 2014, pp. 1–6.

[4] H. Huang, T. Gong, N. Ye, R. Wang, and Y. Dou, "Private and secured medical data transmission and analysis for wireless sensing healthcare system," *IEEE Trans. Ind. Informat.*, vol. 13, no. 3, pp. 1227–1237, Jun. 2017.

[5] C. Zhang, K. Ota, J. Jia, and M. Dong, "Breaking the blockage for big data transmission: Gigabit road communication in autonomous vehicles," *IEEE Commun. Mag.*, vol. 56, no. 6, pp. 152–157, Jun. 2018.

[6] J. J. Astrain, F. Falcone, A. J. Lopez-Martin, P. Sanchis, J. Villadangos, and I. R. Matias, "Monitoring of electric buses within an urban smart city environment," *IEEE Sensors J.*, vol. 22, no. 12, pp. 11364–11372, Jun. 2022.

[7] M. B. Schrettenbrunnner, "Artificial-intelligence-driven management," *IEEE Eng. Manag. Rev.*, vol. 48, no. 2, pp. 15–19, 2nd Quart. 2020.

[8] Q. Xiao, Y. Qiao, M. Zhen, and S. Chen, "Estimating the persistent spreads in high-speed networks," in *Proc. IEEE 22nd Int. Conf. Netw. Protocols*, 2014, pp. 131–142.

[9] S. Gündüz and M. T. Özsu, "A web page prediction model based on click-stream tree representation of user behavior," in *Proc. ACM KDD*, 2003, pp. 535–540.

[10] 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 *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 2026–2034.

[11] S. Wang et al., "Locally private set-valued data analyses: Distribution and heavy hitters estimation," *IEEE Trans. Mobile Comput.*, vol. 23, no. 8, pp. 8050–8065, Aug. 2024.

[12] J. Gong et al., "HeavyKeeper: An accurate algorithm for finding top-*k* elephant flows," in *Proc. USENIX ATC*, 2018, pp. 909–921.

[13] S. Sheng, Q. Huang, S. Wang, and Y. Bao, "PR-sketch: Monitoring per-key aggregation of streaming data with nearly full accuracy," in *Proc. VLDB Endowment*, 2021, pp. 1783–1796.

[14] C. H. Song, Pravein Govindan Kannan, B. K. H. Low, and M. C. Chan, "FCM-sketch: Generic network measurements with data plane support," in *Proc. 16th Int. Conf. Emerg. Netw. EXperiments Technol.*, 2020, pp. 78–92.

[15] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: Methods, evaluation, and applications," in *Proc. ACM IMC*, 2003, pp. 234–247.

[16] Y. Zhang et al., "On-off sketch: A fast and accurate sketch on persistence," in *Proc. VLDB Endowment*, 2020, pp. 128–140.

[17] B. Lahiri, J. Chandrashekar, and S. Tirthapura, "Space-efficient tracking of persistent items in a massive data stream," in *Proc. ACM DEBS*, 2011, pp. 255–266.

[18] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong, "Finding persistent items in data streams," in *Proc. VLDB Endowment*, 2016, pp. 289–300.

[19] T. Yang, H. Zhang, D. Yang, Y. Huang, and X. Li, "Finding significant items in data streams," in *Proc. IEEE 35th Int. Conf. Data Eng. (ICDE)*, 2019, pp. 1394–1405.

[20] Z. Zhong, S. Yan, Z. Li, D. Tan, T. Yang, and B. Cui, "BurstSketch: Finding bursts in data streams," in *Proc. ACM SIGMOD*, 2021, pp. 2375–2383.

[21] J. Li et al., "WavingSketch: An unbiased and generic sketch for finding top-k items in data streams," in *Proc. ACM KDD*, 2020, pp. 1574–1584.

[22] T. Yang et al., "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. ACM SIGCOMM*, 2018, pp. 561–575.

[23] J. Zhu, Z. Gao, P. Reviriego, S. Liu, and F. Lombardi, "Dependability of the k minimum values sketch: Protection and comparative analysis," *IEEE Trans. Comput.*, vol. 74, no. 1, pp. 210–221, Jan. 2025.

[24] M. Charikar, K. Chen, and M. F. Colton, "Finding frequent items in data streams," in *Proc. ICALP*, New York, NY, USA: Springer-Verlag, 2002.

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

[26] W. Li and P. Patras, "Stable-sketch: A versatile sketch for accurate, fast, web-scale data stream processing," in *Proc. ACM Web Conf.*, 2024, pp. 4227–4238.

[27] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM SIGCOMM*, 2010, pp. 267–280.

[28] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, pp. 92–99, 2010.

[29] A. Metwally and D. A. E. Abbadi, Agrawal "Efficient computation of frequent and top-k elements in data streams," in *Proc. ICDT*, New York, NY, USA: Springer-Verlag, 2005, pp. 398–412.

[30] D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," in *Proc. ACM SIGMOD*, 2018, pp. 1129–1140.

[31] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. VLDB Endowment*, 2002, pp. 346–357.

[32] 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 Trans. Netw.*, vol. 27, no. 4, pp. 1432–1445, Aug. 2019.

[33] Y. Zhang et al., "CocoSketch: High-performance sketch-based measurement over arbitrary partial key query," in *Proc. ACM SIGCOMM*, 2021, pp. 207–222.

[34] L. Chen, R. C.-W. Phan, Z. Chen, and D. Huang, "Persistent items tracking in large data streams based on adaptive sampling," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1948–1957.

[35] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[36] J. Huang et al., "Chainsketch: An efficient and accurate sketch for heavy flow detection," *IEEE/ACM Trans. Netw.*, vol. 31, no. 2, pp. 738–753, Apr. 2023.

[37] "The CAIDA anonymized internet traces." CAIDA. Accessed: Feb. 6, 2024. [Online]. Available: http://www.caida.org/data/overview/

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

[39] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "HeavyGuardian: Separate and guard hot items in data streams," in *Proc. ACM KDD*, 2018, pp. 2584–2593.

[40] B. Zhao, X. Li, B. Tian, Z. Mei, and W. Wu, "DHS: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing," in *Proc. ACM KDD*, 2021, pp. 2285–2293.

[41] "Mawi working group traffic archive." Mawi. Accessed: Feb. 6, 2024. [Online]. Available: http://mawi.wide.ad.jp/mawi/

[42] M. Singh, M. Singh, and S. Kaur. "10 days DNS network traffic from April-May, 2016." Mendeley Data. Accessed: Feb. 8, 2024. [Online]. Available: https://data.mendeley.com/datasets/zh3wnddzxy/2

[43] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proc. ACM SIGCOMM*, 2016, pp. 101–114.

[44] R. B. Basat et al., "Designing heavy-hitter detection algorithms for programmable switches," *IEEE/ACM Trans. Netw.*, vol. 28, no. 3, pp. 1172–1185, Jun. 2020.

[45] W. Li and P. Patras, "P-sketch: A fast and accurate sketch for persistent item lookup," *IEEE/ACM Trans. Netw.*, vol. 32, no. 2, pp. 987–1002, Apr. 2023.

Authorized licensed use limited to: University of Edinburgh. Downloaded on October 17,2025 at 09:19:16 UTC from IEEE Xplore. Restrictions apply.