HeavyLocker: Lock Heavy Hitters in Distributed Data Streams

Qilong Shi
Tsinghua University
Beijing, China
sql23@mails.tsinghua.edu.cn

Xirui Li
Tsinghua University
Beijing, China
li-xr24@mails.tsinghua.edu.cn

Hanyue Zheng
Peking University
Beijing, China
2100012943@stu.pku.edu.cn

Tong Yang
Peking University
Beijing, China
yangtong@pku.edu.cn

Yangyang Wang
Tsinghua University
Beijing, China
wangyy-13@tsinghua.edu.cn

Mingwei Xu
Tsinghua University
Beijing, China
xumw@tsinghua.edu.cn

Abstract

In recent years, sketching has emerged as a pivotal technique for identifying heavy hitters (items with high frequency) in large-scale data streams. Despite this progress, the majority of existing sketch algorithms are tailored primarily for detecting local heavy hitters within a single data stream, with only a few capable of extending their application to global heavy hitters across distributed data streams. A common challenge encountered by these algorithms is balancing performance with accuracy. To address this challenge, we introduce HeavyLocker, a novel sketch algorithm that takes advantage of a distinct feature of real data streams: the *separability* of heavy hitters. By leveraging this attribute, HeavyLocker precisely locks and protects potential heavy hitters during the data stream processing, ensuring accuracy in local heavy hitter detection without compromising on speed. This unique capability also facilitates its application to global detection tasks. Through theoretical analysis, we validate the efficacy of HeavyLocker's locking mechanism. Our extensive experiments show that HeavyLocker outperforms five benchmarked algorithms in accuracy and maintains fast speed for both local and global heavy hitter detection, significantly reducing errors by up to an order of magnitude compared to the renowned Double-Anonymous Sketch.

CCS Concepts

• **Networks** → **Network measurement**; • **Information systems** → **Data stream mining**.

Keywords

Distributed data streams; Sketch; Local/Global heavy hitter;

∗ Qilong Shi, Xirui Li, and Hanyue Zheng are co-first authors of this paper, and they conducted this work under the guidance of corresponding authors Mingwei Xu and Yangyang Wang. This work was supported in part by the National Natural Science Foundation of China (No. 62221003, 62132004). The source code is available on GitHub [3].

KDD '25, Toronto, ON, Canada.

ACM ISBN 979-8-4007-1245-6/25/08
https://doi.org/10.1145/3690624.3709167

ACM Reference Format:
Qilong Shi, Xirui Li, Hanyue Zheng, Tong Yang, Yangyang Wang, and Mingwei Xu. 2025. HeavyLocker: Lock Heavy Hitters in Distributed Data Streams. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1 (KDD '25), August 3–7, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3690624.3709167

KDD Availability Link:
The source code of this paper has been made publicly available at https://doi.org/10.5281/zenodo.14599301.

1 Introduction

Recent years have seen a surge in interest in detecting high-frequency items within data stream processing, a vital area with diverse applications spanning data mining [17, 37, 38, 44, 45], databases [13, 43, 46, 54, 55], data traffic measurement [39, 47, 48, 56], data security [21, 22, 24–26, 49], and more [6, 9, 11, 16, 18, 19, 27, 33]. This process involves finding items whose frequency exceeds a predefined threshold (we call them *heavy hitters*) and providing accurate frequency estimations. For instance, consider the need for load balancing in the data center [4, 8, 12, 20, 28, 31, 36, 42, 51]. Here, administrators are tasked with detecting the most substantial data traffic items, termed heavy hitters, to ensure efficient and balanced data performance. Similarly, on social platforms, service providers aim to discern users' closest connections by detecting those individuals who interact frequently with each user [5, 14, 23, 30, 32, 50, 52]. Consequently, the primary objective of this paper is to perform heavy hitter detection in single or multiple data streams.

We introduce the concept of a data stream, a sequence of $\langle key, value \rangle$ pairs, where the *key* identifies each item. For instance, in network data streams, a five-tuple can represent a network packet: source IP address, destination IP address, source port, destination port, and protocol. The *value* typically represents the count of an item, generally one. An example data stream is: $DataStream = \{\langle a, 1\rangle, \langle b, 1\rangle, \langle a, 1\rangle, \langle c, 1\rangle, \langle b, 1\rangle, \cdots\}$. Summing *values* for identical *keys* provides item frequency. Our goal is to identify items whose frequency exceeds a threshold, calculated as the product of the total number of items ($|DataStream|$) and a threshold θ (usually $\leq 0.1\%$). This definition suits local heavy hitter detection within a single stream, but real-world applications often involve multiple data streams requiring distributed processing. As data volume increases, detecting heavy hitters across distributed and multiple data streams becomes crucial. By considering the sum of all data stream sizes as N, we extend local detection to a global view without altering θ.

Distributed data streams can be either disjoint or intersecting. Disjoint data streams ensure a *key* appears exclusively in one stream, while intersecting data streams allow the same *key* to appear across multiple streams. This distinction is vital as both configurations occur frequently in real-world applications. *Disjoint data streams* are found in autonomous systems (AS) within wide-area networks (WAN), where external traffic is directed through various border routers. The WAN routing protocol mandates that all packets from the same source IP traverse a single border router. Thus, considering the source IP as the key, streams across distinct border routers are disjoint. Network operators must track primary traffic sources, identifying source IPs contributing the highest packet volume over time. Each border router tallies and reports the most frequent source IPs, enabling operators to pinpoint global heavy hitters. *Intersecting data streams* are prevalent in peer-to-peer (P2P) networks, where resources like files are shared directly between peers without a centralized server. Each peer can act as both client and server. Considering the file identifier (e.g., a hash of the file) as the key, streams representing file transfers of the same file across different peers are intersecting data streams. This intersection is significant for monitoring file distribution and popularity within the P2P network and for optimizing network resources.

In the era of big data, the rapid growth of data volume and speed makes capturing each item's information challenging. Approximate solutions like sketches have become crucial. Sketches such as CM Sketch+heap [17] and Elastic Sketch [46] detect local heavy hitters and extend to identify global heavy hitters across distributed data streams. They leverage the mergeability of CM Sketch and store hot item keys for global detection. However, CM Sketch's large error leaves room for improvement. USS [41] refines the SpaceSaving algorithm to be unbiased in multi-stream merging, but it increases frequency estimate variance in skewed data streams. DA Sketch [53] reduces aggregation errors with double anonymity principles but still needs enhancements for better accuracy.

We observed that existing algorithms primarily address the skewness of data streams, overlooking other potential features. Motivated by this, we ask ***whether incorporating additional features of heavy hitters could yield a more effective sketch algorithm suitable for both local and global heavy hitter detection.***

In response, we introduce **HeavyLocker**, a new sketch algorithm that accurately **locks** and safeguards **heavy hitters** in both single and distributed data streams by leveraging the separability of heavy hitters. The innovations of HeavyLocker are highlighted below:

- **First to Utilize separability:** HeavyLocker leverages the separability of heavy hitters and non-heavy hitters, enhancing prediction reliability.
- **Lockable Buckets:** It consists of rows of lockable buckets with multiple cells for item storage, using separability to set dynamic thresholds. Once a bucket's item frequency surpasses this threshold, it locks, preventing further operations and increasing accuracy.
- **Further Optimization:** Two optional schemes are proposed to improve accuracy further.
- **Scalable Across Distributed Streams:** Separability holds across multiple data streams, allowing HeavyLocker to extend naturally without compromising accuracy.

To evaluate real-world performance, we implemented HeavyLocker on software platforms (e.g., CPU) and conducted various heavy hitter detection tasks, including *local heavy hitters*, *global heavy hitters*, and *adaptability to distributed data streams*. Extensive experiments show that by considering the data stream's separability and dynamically locking buckets, HeavyLocker consistently outperforms state-of-the-art algorithms like DA Sketch by an order of magnitude while maintaining high processing speeds. These results underscore HeavyLocker's practical applicability and scalability.

The rest of the paper is organized as follows: Section 2 provides background on heavy hitter detection and reviews recent advancements. Section 3 describes HeavyLocker's data structure and algorithm. Section 4 offers a mathematical analysis, and Section 5 details the experimental setup and results. Finally, Section 6 summarizes our findings and implications.

2 Background and Motivation

2.1 Problem Statement

First, we define two different kinds of distributed data streams and local/global heavy hitter detection tasks. Next, we introduce the related work and algorithms in detail. Finally, we give a more intuitive motivation for our algorithm.

• **Data stream:** A data stream $\mathcal{D} = \{e_1, e_2, \cdots, e_N\}(e_i \in E$, where E is the item set) contains N items. Each item is a $\langle key, value\rangle$ pair. The key serves as an ID, while the value represents the frequency of this item. For example, in the field of computer networks, we often use the 5-tuple headers to identify a TCP flow. In this paper, we can set $value = 1$ (assume items arrive individually). The frequency of an item $e_i (e_i \in E)$ is defined as $f_i \triangleq \sum_j I\{e_i.key = e_j.key\}$ where I is the indicator variable (either 0 or 1). An example of a data stream is shown in Fig. 1: $DataStream = \{\langle e_1, 1\rangle, \langle e_3, 1\rangle, \langle e_2, 1\rangle, \langle e_1, 1\rangle, \langle e_4, 1\rangle, \langle e_3, 1\rangle, \langle e_4, 1\rangle, \langle e_4, 1\rangle, \cdots\}$. $f_1 \sim f_4$ can be calculated by counting the number of times they appear in the data stream.

• **Disjoint data stream:** Given n data streams $\mathcal{D}_1, \cdots, \mathcal{D}_n$, where $\mathcal{D}_i = \{e_{(i,1)}, \cdots, e_{(i,m_i)}\}$ contains m_i items. And each item $e_{(i,j)}$ belongs to set $\mathcal{U}_i = \{u_{(i,1)}, \cdots, u_{(i,n_i)}\}$, where n_i is the number of distinct items in $\mathcal{D}_i$. n data streams are disjoint if $\mathcal{U}_i \cap \mathcal{U}_j = \emptyset$ for any two different data stream $\mathcal{D}_i$ and $\mathcal{D}_j$.

• **Intersecting data stream:** if $\exists \mathcal{D}_i, \mathcal{D}_j$ such that $\mathcal{U}_i \cap \mathcal{U}_j \neq \emptyset$, then these n data streams are intersecting.

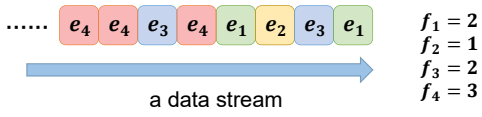
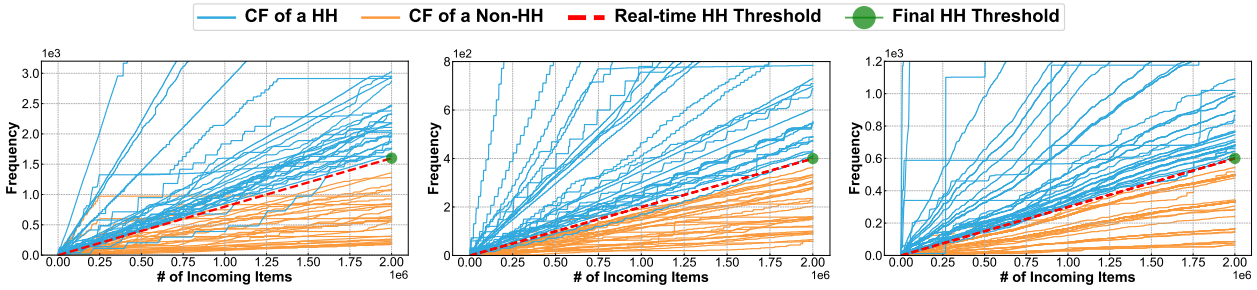


Figure 1: A data stream.

• **Local Heavy Hitter Detection [39]:** In a single data stream $\mathcal{D}$, we aim to find an item set E_{hh} and their frequencies satisfying that $\forall e_i \in E_{hh}, f_i \geq \theta \times N$, where θ is a predefined threshold and the $N = |\mathcal{D}|$ (the number of items in $\mathcal{D}$). Local heavy hitter detection is important in data-intensive applications like recommendation systems [10].

• **Global Heavy Hitter Detection:** In n data streams $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_n$, assume that the frequency of item e in $\mathcal{D}_j$ is defined as $f_i^{(j)}$. We aim to find an item set E_{hh} and their frequencies satisfying that



(a) CAIDA dataset (b) Campus dataset (c) IMC dataset

Figure 2: The separability of heavy hitters in real-world data streams.

$\forall e_i \in E_{hh}, \sum_{j=1}^{n} f_i^{(j)} \geq \theta \times N$, where θ is a predefined threshold and the $N = \sum_{j=1}^{n} |\mathcal{D}_j|$. It can be found that in the global scenario, we should merge all data streams first and then consider the heavy hitter on the merged data stream.

2.2 Related Work

This section explores established heavy hitter detection algorithms, focusing on those adaptable to distributed data streams. The **Count-Min sketch** (CM) [15] uses d arrays of counters, incremented by hash functions for each item. It reports the minimum counter value, ensuring no underestimation. CM Sketch includes a min-heap for hot items, allowing easy merging in distributed scenarios, though its accuracy limitations hinder performance.

Elastic Sketch (ES) [46] partitions into heavy and light parts, managed by an ostracism algorithm. Items exceeding a vote ratio threshold λ move to the light part (typically a CM Sketch), which can expel potential heavy hitters, reducing accuracy. ES merges heavy and light parts separately in multi-stream applications.

MV Sketch (MV) [39] enhances CM Sketch with rows of buckets, each recording total frequency, item key, and item frequency. Majority voting updates candidate heavy hitters, but extra storage per bucket affects memory efficiency.

Unbiased SpaceSaving (USS) [41] adapts the SpaceSaving algorithm for multi-streams by adjusting its item replacement strategy. While straightforward to merge, its method of overestimating hot items and underestimating cold ones increases absolute errors.

DA Sketch (DAS) [53], the current state-of-the-art, redefines unbiasedness with "fairness," treating heavy hitter selection and frequency estimation independently, achieving reliable results even in skewed multi-stream distributions.

2.3 Insights about "Lock"

We analyzed three real-world datasets to identify heavy-hitter patterns. As shown in Fig. 2, we selected two million items and plotted their cumulative frequency curves. These monotonically increasing curves represent item frequencies against the number of arriving items. Each line represents an item type, with heavy hitter thresholds indicated by green dots. Blue lines represent heavy hitters (above the threshold), and orange lines represent non-heavy hitters (below the threshold).

2.3.1 Star with Separability: We observed that most blue and orange lines are straight and separated by a red line representing the real-time heavy hitter threshold ($\theta\times$ number of arrived items). This consistent pattern, particularly among medium to highly frequent items, suggests "separability"—heavy and non-heavy hitters are separable.

2.3.2 ***Lock*** *to Protect Heavy Hitters:* From separability, we infer that final heavy hitters (blue lines) are likely real-time heavy hitters (above the red line). Conversely, items consistently qualifying as real-time heavy hitters are likely final heavy hitters. Therefore, retaining such items until the data stream ends is crucial.

Our algorithm maps items to buckets, each with a lock bit. When the frequency of all items in a bucket exceeds the real-time heavy hitter threshold, the lock bit is set to true, preventing decay/replacement operations. This strategy preserves real-time heavy hitters in the bucket, increasing the accuracy of final heavy hitter detection.

3 HeavyLocker Algorithm

This section outlines the data structure and functionality of the HeavyLocker algorithm, designed to support heavy hitter detection in distributed data streams. HeavyLocker is equipped with three fundamental interfaces: *Insert()*, *Query()*, and *Merge()*. The *Insert()* and *Query()* functions manage operations within a single sketch, while *Merge()* facilitates the integration of multiple sketches from different data streams.

3.1 Basic Framework

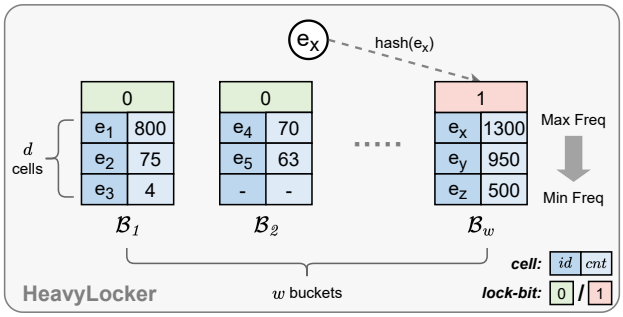


Figure 3: The data structure of HeavyLocker.

Fig. 3 depicts the structure of HeavyLocker, which consists of a row of w buckets, labeled $\{\mathcal{B}_1, \mathcal{B}_2, ..., \mathcal{B}_w\}$. Each bucket contains d cells, and we denote the j-th cell in the i-th bucket (from left to right, from bottom to top) as $B[i][j]$. The storage space in

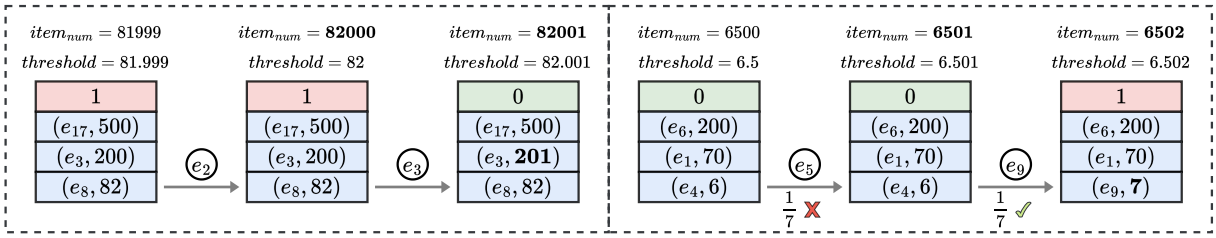


Figure 4: A running example of HeavyLocker (with $\theta = 0.1\%$).

each cell, $B[i][j]$, is partitioned into two sections: $B[i][j].id$ for the item's key and $B[i][j].cnt$ for its frequency count. Furthermore, each bucket is equipped with a lock-bit, $B[i].lock$, which controls whether replacements are permitted within that bucket. Cells within a bucket are sorted by frequency in descending order; thus, $B[i][0].cnt \leq B[i][1].cnt \leq \ldots \leq B[i][d-1].cnt$. When a bucket is full, and the value in $B[i][0].cnt$ exceeds a predefined threshold, the bucket is locked. This prevents any further replacement in the bucket, assuming it likely contains only heavy hitters. The pseudo-code for this section is in Appendix A.

3.1.1 ***Insert(e):*** Algorithm 1 shows the insert operation. Initially, all entries are set to 0. Providing that the heavy hitter threshold is θ. When inserting an item e, we first compute its corresponding bucket $B[i]$ via a hash function. We also increment $item_{num}$ by 1 to indicate how many items have been processed. Next, we check whether the frequency of the smallest item in $B[i]$ is greater than $item_{num} \times \theta$ and change the lock-bit $B[i].lock$ accordingly. For example, assuming that 100,000 items have been processed, $\theta = 0.05\%$, then when $B[i][0].cnt \geq 100,000 \times 0.05\% = 50$, the bucket will be locked; otherwise, it will be unlocked. As shown in Fig. 3, $B[1][0].cnt = 4 < 50$, and $B[2][0].cnt = 0 < 50$, so B_1 and B_2 not locked. Meanwhile, B_w is locked because its minimum cell frequency is $500 > 50$.

Next, we traverse all the cells $B[i][j] (0 \leq j < d)$ in the bucket to determine whether the item is stored. If so, we can add the *cnt* field. If an empty cell is in the bucket, insert it directly.

Finally, when the bucket is full, and e is not found, we try to do a replacement operation on the smallest item. The algorithm ends here if $B[i]$ is locked. If $B[i]$ is not locked, we will use the RAP_Replacement strategy [7] to the smallest item. If the frequency is reduced to 0, replace it; otherwise, we end here.

3.1.2 ***Query():*** Algorithm 2 shows the operation of querying all heavy hitters. We only need to traverse all $d \times w$ cells in HeavyLocker and return the items whose frequency exceeds the given threshold. Note that our algorithm is invertible, i.e., the item can be recovered directly from the data structure because we preserve the full key of the item. Those algorithms that do not record the item's full key are not invertible. They must traverse the item set again to query all heavy hitters, which is time-consuming.

3.1.3 ***Merge(V):*** Algorithm 3 outlines merging n data streams by combining n HeavyLockers, each with the same size and hash function configuration. We define $B[i]^{(v)}$ as the i-th bucket in the v-th HeavyLocker, where $1 \leq i \leq w$ and $1 \leq v \leq n$. To aggregate these, we use the Stream Summary [29] data structure to combine the largest items at the same location in each HeavyLocker. Algorithm 3 traverses the buckets at the same location, pushing data from each cell into the Stream Summary if the key is absent. Otherwise, it updates the item's counter and order in Stream Summary. The i-th bucket of the merged HeavyLocker is built by querying the top-d elements of Stream Summary.

Merging operates on buckets rather than the entire HeavyLocker, keeping the overhead manageable and avoiding the need for a large Stream Summary. This approach provides a precise network measurement as if detected by a single large detector [25, 39]. Additionally, we can divide a data stream into groups, deploy multiple HeavyLockers, and aggregate them before reporting to enhance detection and reduce reporting overhead, as demonstrated in the experiment in 5.5.

3.1.4 Selection of dynamic lock threshold in x data streams: It is crucial to appropriately adjust the θ threshold across various data stream configurations. In disjoint data streams, the threshold for each HeavyLocker is adjusted to $\theta \cdot x$ to account for separation, while in intersecting data streams, the threshold remains θ. This differentiation ensures optimal threshold settings to handle specific dynamics and overlaps of the data streams. For a detailed explanation, see the theoretical analysis in Section 4.3.

3.2 Example of Insertion

Here we show the insertion example of HeavyLocker, as shown in Fig. 4, assuming the heavy hitter threshold is 0.1%. In the example on the left, To insert e_2, the HeavyLocker calculates one hash function, maps e_2 to a bucket, and updates $item_{num}$. Because e_2 is not stored in it, and the threshold for locking the bucket at this time is $82000 \times 0.1\% = 82 \leq$ the smallest counter (82), so nothing is changed. To insert e_3, we also update $item_{num}$ and then find that e_3 is stored in the bucket, so we update this cell to $(e_3, 201)$. Currently, the bucket lock threshold is $82.001 > 82$, so the bucket is set to unlock. In the example on the right, to insert e_5, replacement is triggered because e_3 is not stored in this unlocked bucket. We replace the smallest $(e_4, 6)$ with probability $\frac{1}{6+1} = \frac{1}{7}$. Assuming the replacement fails, then nothing is changed. To insert e_9, similarly, e_9 does not exist in this bucket, so we try to replace $(e_4, 6)$. Assuming the replacement success, we update this cell to $(e_9, 7)$. At this time, since $7 > 6.502$, the bucket is set to lock.

3.3 Optimization 1: Tuning Lock Threshold

As mentioned in Section 2.3, the blue and orange lines in Fig. 2 are almost separated by the real-time heavy hitter red line. However, some blue lines temporarily fall below the red line, caused by heavy hitters arriving slowly initially and accelerating later. Using the real-time heavy hitter threshold for locking buckets can lead to errors if

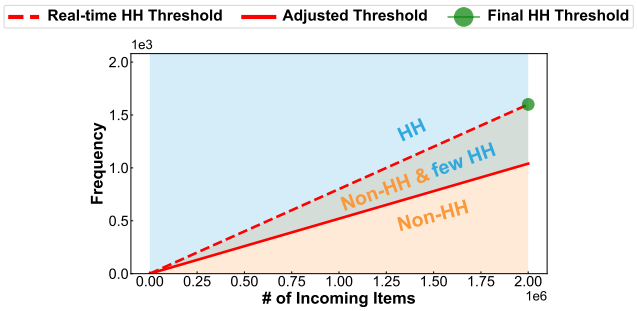


Figure 5: Insight of adjusting heavy hitter threshold.

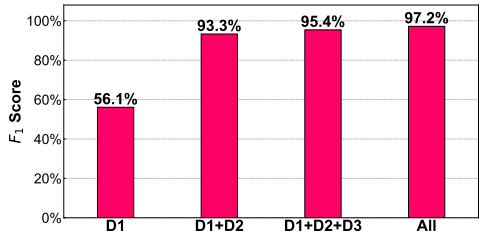


Figure 6: The maximum F_1 *score* when consider different design and optimization. $D1$: Multiple cells in each bucket. $D2$: Dynamically lock with real-time heavy hitter threshold. $D3$: Tuning lock threshold. $D4$: Multi-hashing.

these items are replaced by cold items. To protect them, we lower the original threshold's slope, creating a new threshold. As shown in Fig. 5, we use the solid red line as the new lock threshold, which is the red dotted line multiplied by a constant L (the *lock tuning value*). Although this may mistakenly protect some non-heavy hitters with medium frequency, Figs. 6 and 7(b) demonstrate its effectiveness. To implement this, we change $\underline{item_{num} \times \theta}$ to $\underline{item_{num} \times \theta \times L}$ in line 3 of Algorithm 1.

3.4 Optimization 2: Multi-hashing

In Algorithm 1, each item is mapped to a single candidate bucket using one hash function. This can lead to a situation where $d+1$ heavy hitters are mapped to the same bucket, causing one item to be lost since only d cells are available, reducing accuracy. To mitigate this, we use multiple hash functions to select more candidate buckets, reducing hash conflicts. This optimization is especially effective when memory is tight and collision rates are high, as detailed in Section 4.1. However, more hash functions increase overhead and decrease processing speed. Fig. 7(c) illustrates the trade-off between accuracy and speed. To implement this, we calculate multiple hash functions in line 2 of Algorithm 1 and traverse them in line 7.

3.5 Put Them All Together

Fig. 6 illustrates the impact of each design and optimization. Using only a multi-cell structure in a bucket (D1), the F_1 score for detecting heavy hitters is 56.1%. Adding the core of our algorithm, the dynamic threshold lock (D2), increases the F_1 score to 93.3%, demonstrating its effectiveness. Incorporating threshold tuning (D3) further improves the score to 95.4%. Finally, adding multi-hashing (D4) raises the F_1 score to 97.2%.

4 Mathematical Analysis

In this section, we provided the hash collision probability of HeavyLocker (Section 4.1), as well as its error bounds (Section 4.2), and from this, we derived its theoretical precision (Section 4.4) and recall rate (Section 4.5). In Section 4.3, we discussed several relevant facts, such as the exponential level optimization of error by the lock operation. In the argument of this section, we assume n as the total number of items, w as the number of buckets, and d as the number of cells in a bucket. We use e_i to represent an item, f_i to represent its size, and $\hat{f}_i$ to represent the estimated size of it. We use θ to denote our lock threshold, i.e., the bucket is locked when the smallest item reaches the current $\theta \times item_{num}$. We use ϕ to represent the threshold for heavy hitters, i.e., an item is considered a heavy hitter when it exceeds ϕn. We assume that the number of data streams is ds_{num}. For detailed proof, please refer to [3].

4.1 Hash Collision Probability

In this section, we present the collision probabilities of inserting an item under single-hash and multi-hash scenarios, respectively, demonstrating that the collision probability with multi-hash is significantly lower than that with single-hash.

Theorem 4.1. *Assuming we use a single hash h_1, and the sketch has inserted s distinct items, let p_1 be the probability of a collision when we insert a new item, we have:*

$$p_1 \approx \frac{1}{\sqrt{2\pi}\beta} e^{-\frac{\beta^2}{2}}$$

where $\beta = \frac{d - \frac{s}{w}}{\sqrt{\frac{s}{w}(1-\frac{1}{w})}}$.

Theorem 4.2. *Assume we use k independent hash functions $h_1, \cdots, h_k$, and the sketch already contains s distinct items. Let p_k be the probability of a collision when we insert a new item, we have:*

$$p_k \approx p_1^k$$

So, we can find that the hash collision probability decreases exponentially with the number of hash functions, proving the effectiveness of multi-hashing in Section 3.4.

4.2 Error Bound

In this section, we present the upper and lower bounds of overestimation and underestimation for HeavyLocker.

4.2.1 *Overestimation Bound.*

If a heavy hitter is overestimated, it implies that the last time it was stored in a bucket, it replaced an item larger than itself. However, due to the presence of a lock mechanism, it is unlikely that there exists an item larger than itself that is not locked. Below, we rigorously explain this point:

Theorem 4.3. *Assuming a heavy hitter e_1 is overestimated, its actual size is f_1, and HeavyLocker provides an estimated size $\hat{f}_1$. We have:*

$$\Pr[\hat{f}_1 \le f_1 + \epsilon] \ge 1 - \sqrt{\frac{p(1-p)}{8\pi(p-\theta)\epsilon}} e^{-2\frac{(p-\theta)\epsilon}{p(1-p)}}$$

where $p = \frac{f_1}{n}$.

4.2.2 Underestimation Bound. In this section, we will present the lower bounds for underestimation in HeavyLocker. Underestimation occurs because the underestimated item consistently fails to replace the smallest value in its bucket. However, due to the presence of the lock mechanism, if the bucket is not locked, the smallest value in the bucket should be below θn. This makes it unlikely that a heavy hitter cannot replace the smallest value in the bucket over a long period. We will explain this in detail below.

Theorem 4.4. *Assuming a heavy hitter e_1 is underestimated, its actual size is f_1, and HeavyLocker provides an estimated size of $\hat{f_1}$. Then, we have:*

$$\Pr[\hat{f_1} \geq f_1 - \epsilon] \geq 1 - (\frac{p-\theta}{\theta\epsilon})^{\frac{p}{\theta}}$$

where $p = \frac{f_1}{n}$.

4.3 Optimization of Error

In this section, we explain there facts: (1) The Lock mechanism reduces errors, thereby increasing accuracy; (2) The error in Global HeavyLocker is less than that in Local HeavyLocker; (3) If the bucket threshold at a single data stream is θ, it will change in the case of multiple data streams.

Theorem 4.5. *The lock mechanism reduces errors, thereby increasing accuracy.*

Theorem 4.6. *The error in Global HeavyLocker is less than that in Local HeavyLocker.*

Theorem 4.7. *Assuming that all streams arrive uniformly, and the bucket threshold at a single point is θ. Then, in the case of multiple data streams:*

- *if they are intersecting, the threshold for each Local HeavyLocker is θ.*
- *if they are disjoint, the threshold for each Local HeavyLocker is $\theta \cdot ds_{num}$.*

where ds_{num} represents the number of data streams.

4.4 Precision Rate

In this section, we estimate the precision of the HeavyLocker algorithm. Precision is defined as the ratio of the number of items that truly exceed ϕn to the number of items that HeavyLocker reports as exceeding ϕn.

Theorem 4.8. *Let the precision be denoted by p_0, the target threshold by ϕ, and the lock threshold by θ, the number of heavy hitter is s. We propose the following estimation for p_0:*

$$E(p_0) \geq 1 - 4\frac{w}{s}\sqrt{\frac{\phi(1-\phi)}{\pi(\phi-\theta)\epsilon}}e^{\frac{(\phi-\theta)\epsilon}{64\phi(1-\phi)}} - \epsilon_0$$

where $\epsilon_0 < 0.01$.

4.5 Recall Rate

In this section, we provide an estimation of the recall for HeavyLocker. Here, recall is defined as the proportion of items that truly exceed ϕn and are reported by HeavyLocker as exceeding ϕn.

Theorem 4.9. *Let the recall rate be denoted by r_0, the target threshold by ϕ. Then, we have:*

$$E(r_0) \geq 1 - \frac{1}{2\sqrt{2\pi}\beta}e^{-\frac{\beta^2}{2}} - \epsilon_0$$

where $\beta = \frac{d-\frac{1}{w\phi}}{\sqrt{\frac{1}{w\phi}(1-\frac{1}{w})}}$,$\epsilon_0 < 0.01$.

5 Experimental Results

In this section, we present a series of experiments conducted to evaluate HeavyLocker's performance. First, we describe the experimental setup and the metrics used, and then we tune the parameters for optimal results. Next, we analyze the effect of HeavyLocker on local and global heavy hitter detection, considering different memory capacities, varying heavy hitter thresholds, diverse data skewness, and changes in the number of data streams. Finally, we provide a thorough analysis of the experimental results.

To ensure that our experiments are fair and transparent, we have used either commonly adopted algorithms or open-sourced ones provided by their authors. We have made our related source codes and datasets available on GitHub [3], allowing for easy access and reproducibility of our experiments.

5.1 Experiment Setup

Here, we only show a brief experimental setup; please refer to Appendix B for a detailed version.

5.1.1 Test Platform. We conducted our experiments using a machine equipped with an Intel $i7-9700CPU@3.0GHz$ and 16GB DRAM, running Ubuntu 20.04. To mitigate CPU jitter errors, we computed average results based on 10 runs for each evaluation.

5.1.2 Datasets. We use 2 kinds of datasets in experiments: CAIDA [1] and Zipf [34] dataset.

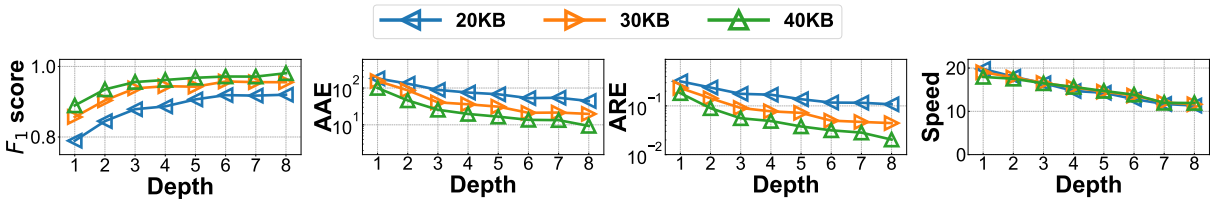
5.1.3 Comparing Algorithms. We implement our HeavyLocker (HL) in C++, and compare our results with CM sketch+heap (CM) [15], Elastic sketch (ES) [46], MV Sketch (MV) [40], Unbiased Space-Saving (USS) [41], and Double-anonymous Sketch (DAS) [53].

5.1.4 Tasks and Metrics. We perform heavy hitter detection (reporting items whose sizes are larger than a predefined threshold), and the following metrics are considered: Speed, AAE, ARE, Precision, Recall, and F_1 score.

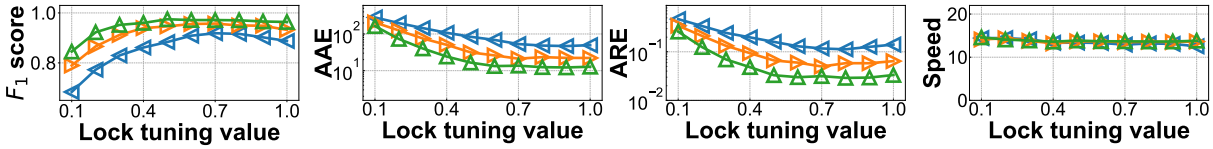
In the following sections, we illustrate the performance of HeavyLocker (HL) by processing speed, AAE, ARE, and F_1 score, against memory, heavy hitter threshold, skewness, and number of data streams. Due to space constraints, precision and recall figures are in the supplementary file [3]. Abbreviations are: HL for HeavyLocker, CM for CM Sketch+heap, ES for Elastic Sketch, MV for MV Sketch, USS for Unbiased SpaceSaving, and DAS for Double-anonymous Sketch. We consider USS and DAS as the state-of-the-art (SOTA) algorithms due to their superior accuracy.

5.2 Parameter Tuning

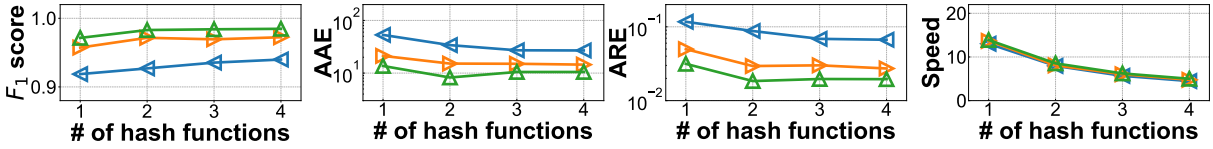
5.2.1 Number of cells per bucket (depth). The parameter of the number of cells (d) per bucket plays a crucial role in determining the optimal performance of HeavyLocker. The value of d greatly



(a) Accuracy and speed vs. depth.



(b) Accuracy and speed vs. lock tuning value.



(c) Accuracy and speed vs. # of hash.

Figure 7: Parameter Tuning.

influences the trade-off between accuracy and speed. If d is small, the number of items per bucket is limited, so the time required to traverse the entire bucket is less, but the accuracy is generally low at this time. Conversely, if d is large, each bucket can hold more items, so the time required to traverse the entire bucket is relatively large, but the accuracy is generally high at this time.

Fig. 7(a) illustrates our experimentation with varying the number of cells per bucket (i.e., depth) from 1 to 8 and measuring the corresponding AAE, ARE, F_1 score, and speed. Each figure comprises three lines representing the outcomes of using memory sizes of 20KB, 30KB, and 40KB, respectively. Our observation reveals that the accuracy for different memories improves considerably as d increases from 1 to 6, while it remains stable when it goes up to 8. Therefore, *we decided to set* $d = 6$ in the following experiments.

5.2.2 Lock tuning value (L). As mentioned in Section 3.3, in Algorithm 1, it is reasonable to lock a bucket only if the minimum frequency in the bucket exceeds the heavy hitter threshold (i.e., $item_{num} \times \theta$) to ensure that all locked buckets contain heavy hitters. However, some heavy hitters may come slowly in certain periods. If it is mistakenly replaced before reaching the heavy hitter threshold, it will cause a large error. To address this issue, we need to adjust the threshold of locking a bucket accordingly. This explains why the algorithm included the lock tuning value (L). By adjusting the lock tuning value appropriately, we can balance capturing all heavy hitters and ensuring that no heavy items are lost.

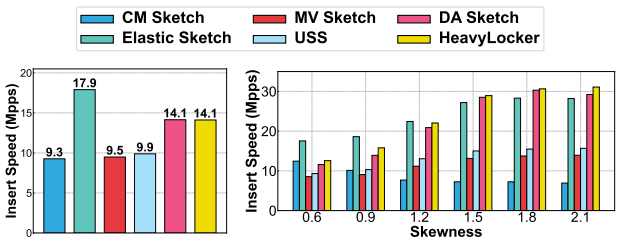
Based on the results displayed in Fig. 7(b), we experimented with three different memory sizes, each with a lock threshold ranging from 0.1 to 1.0 in increments of 0.1. The accuracy levels showed an initial increase and subsequent decrease, reaching an optimal value of 0.7. Thus, *we set the lock tuning value to* $L = 0.7$, which delivered the best accuracy performance.

5.2.3 Number of hash functions. As mentioned in Section 3.4, Algorithm 1 assigns each item to a candidate bucket based on only one hash function. One possible approach to enhance the performance of heavy hitter detection is to expand the number of hash functions utilized. By doing so, we can increase the loading rate of the overall data structure, which in turn may improve the accuracy of identifying heavy hitters.

Based on Fig. 7(c), we altered the identifier from 1 to 4. Our results reveal that increasing the number of hash functions enhances the accuracy of detecting heavy hitters while reducing the throughput rate due to the time-consuming nature of hash function calculation. For scenarios where speed requirements are not strict, we recommend using 2 hash functions to reduce errors, but in the experimental part, we choose to pay more attention to speed. Consequently, *we decided to maintain the number of hash functions used in HeavyLocker at 1 during the subsequent experiment.*

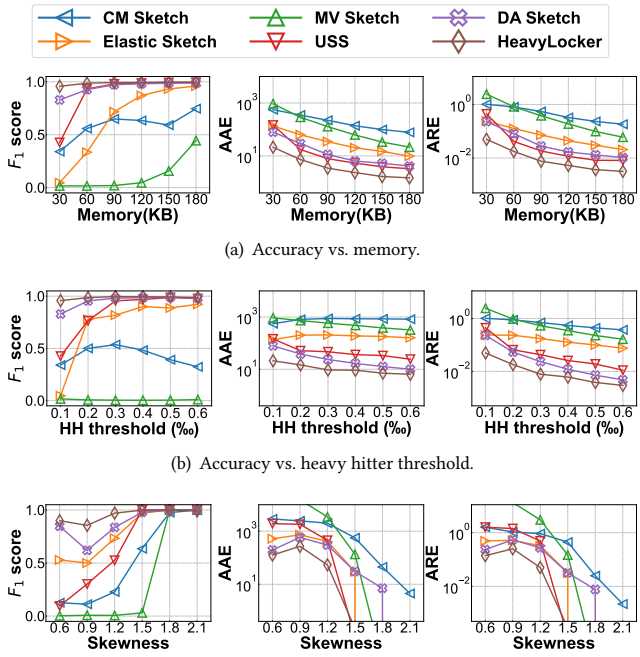
5.3 Processing Speed

Fig. 8(a) compares the processing speed of HeavyLocker (HL) with other methods on normal datasets. HL is consistently faster



(a) Insert speed in CAIDA. (b) Insert speed vs. data skewness.

Figure 8: Insert speed.

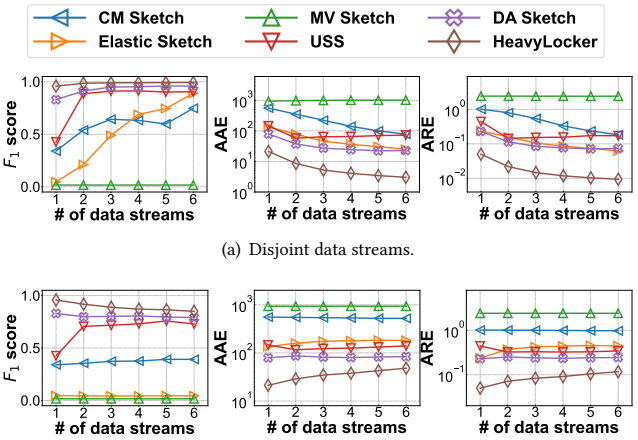


(a) Accuracy vs. memory.

(b) Accuracy vs. heavy hitter threshold.

(c) Accuracy vs. dataset skewness.

Figure 9: Local Heavy Hitter Detection.



(a) Disjoint data streams.

(b) Intersecting data streams.

Figure 10: Accuracy vs. # of Data Streams.

than CM, MV, and USS, but slower than ES. Specifically, HL matches DAS in speed, is 40% faster than USS, and 20% slower than ES. The processing speed is influenced mainly by the number of calls to costly operations like the hash and rand functions (used in probability replacement) as discussed in 3.1.1. Unlike methods with hierarchical hash functions, HL uses a single layer of buckets, avoiding multiple hash computations. However, HL incurs a slightly higher insertion cost than ES due to traversing six cells in a bucket and calculating random values.

As shown in Fig. 8(b), HL and DAS processing speeds increase with dataset skewness, surpassing ES. This is because HL and DAS use the rand function for conflicts. In the Zipf dataset, higher skewness leads to fewer item types and increased hot item frequencies, reducing competition. Thus, new items are more likely to hit or insert into empty cells, minimizing rand function calls and improving insertion efficiency.

5.4 Local Heavy Hitter Detection

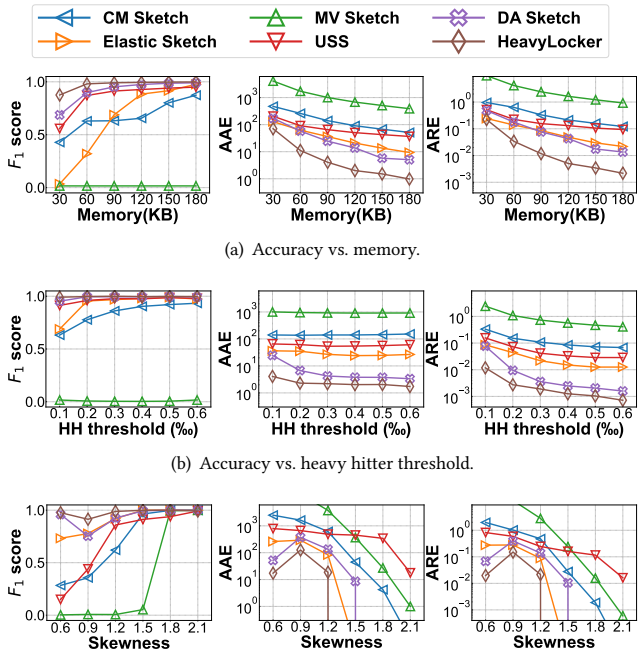
5.4.1 ***Change memory of data structure:*** In this experiment, we change the memory size from 30KB to 180KB to observe the influence of different memory sizes on each algorithm. From Fig. 9(a), we find that HL is much more accurate than the other five algorithms for most memory sizes. When the memory size is sufficient, HL's AAE/ARE is about 3.3× and 2.6× lower than that of DAS and USS and can reach F_1 score of 95% even with only 30KB. The results get even better when the memory is small, which means that compared to the most advanced methods, HL can better handle serious hash conflicts caused by small memory sizes. Although MV and ES can achieve a certain recall advantage with small memory, this is due to the overestimation of the size of their streams, which also makes their F_1 score much lower than HL. Besides, it should be noted that we implemented the invertible version of CM, which restricts the quantity of information it provides to a limited range and ensures that its recall always remains at a moderate level in the experiments. Thus, the performance of CM in our experiments may be different from the traditional cases.

5.4.2 ***Change heavy hitter threshold:*** In this experiment, we change the heavy hitter threshold from 0.01% to 0.06%. Fig. 9(b) shows how adjusting the heavy hitter threshold affects the methods' accuracy. We find that HL consistently outperformed other methods. Specifically, HL's AAE and ARE are up to 3.7× and 5.5× lower than DAS, respectively, and can reach F_1 score of 95% when the heavy hitter threshold = 0.01%. We observe that the AAE/ARE of most methods decreases and the F_1 score increases as the heavy hitter threshold increases. This is because a higher threshold reduces the number of heavy hitters that need identification, and hotter items tend to experience fewer hash collisions, thus diminishing errors.

5.4.3 ***Change skewness of dataset:*** In this experiment, we studied the influence of different dataset skewnesses on the methods by measuring the accuracy of each method under the Zipf datasets mentioned in 5.1.2. According to Fig. 9(c), HL achieves the highest F_1 score and the lowest AAE/ARE in datasets with any skewness, which proves that HL has good adaptability to datasets with varying skewnesses. Specifically, HL's AAE and ARE are up to 5.6× and 5.4× lower than DAS, respectively, and can reach F_1 score above 84% with any skewness.

5.5 Scalability of Heavy Hitter Detection

To evaluate HL's scalability, we split the CAIDA dataset to generate disjoint and intersecting data streams. For disjoint streams, the dataset is partitioned by hashing the item's key and taking modulo the number of partitions. For intersecting streams, the dataset is randomly split by computing a random value for each item. We vary the number of detectors and use merge functions to explore the impact of data stream quantity on the accuracy of aggregated frameworks while keeping the total flow rate constant.

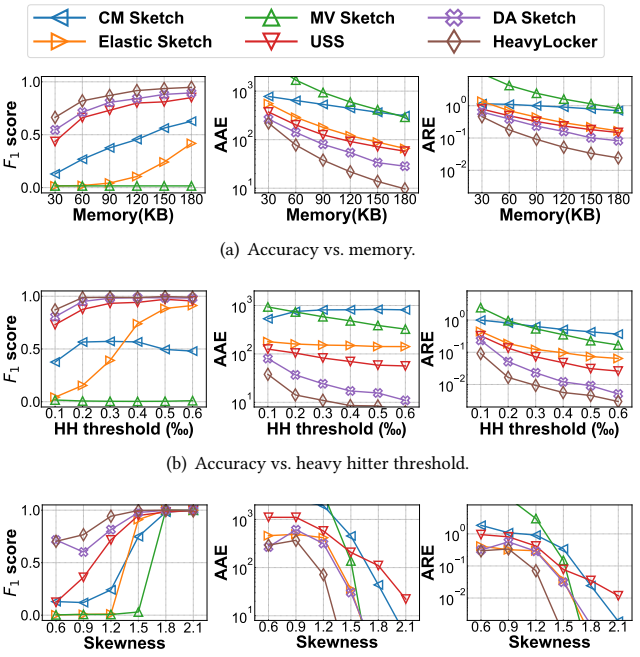


(a) Accuracy vs. memory.

(b) Accuracy vs. heavy hitter threshold.

(c) Accuracy vs. dataset skewness.

Figure 11: Global Heavy Hitter Detection (*Disjoint*).



(a) Accuracy vs. memory.

(b) Accuracy vs. heavy hitter threshold.

(c) Accuracy vs. dataset skewness.

Figure 12: Global Heavy Hitter Detection (*Intersecting*).

Fig. 10(a) shows that in the disjoint case, aggregation accuracy increases with the number of data streams. More detectors mean each tracks fewer heavy hitters, reducing sketch error at each detector. The merging process further reduces error (see Theorem 4.6), resulting in HL's AAE and ARE being up to 7.6× and 7.9× lower than DAS, respectively.

Fig. 10(b) shows that in the intersecting case, aggregation accuracy decreases with more data streams. This is because the number of heavy hitters each detector tracks remains unchanged, increasing inaccuracy during aggregation. Thus, more detectors lead to larger errors. The special performance of MV is likely due to its unique estimation method in the merge function.

In summary, it is recommended to divide a device's data stream into many disjoint streams and measure them individually with HeavyLockers. Aggregating these results at the device level before transmission to the control plane improves measurement accuracy without increasing reporting overhead. Notably, HL's F_1 score and AAE/ARE are consistently better than other methods, demonstrating its advantages in data aggregation.

5.6 Global Heavy Hitter Detection

In this experiment, we simulated the effect of memory size, heavy hitter threshold, and dataset skewness on global heavy hitter detection. To realize this experiment, we divided the dataset into six parts using different methods and measured them using six detectors, respectively. Then, we simulated the aggregation process of results in the collector to observe the effectiveness of various methods in finding global heavy hitters.

5.6.1 Disjoint data stream. Fig. 11 shows the result when data streams are disjoint. We observe that the overall trend is basically consistent with the local heavy hitter detection in 5.4, and the performance of all methods except MV is better than local heavy hitter detection. This means that for most methods, splitting the data stream into disjoint parts while measuring can benefit the final aggregation result. Specifically, when varying memory, heavy hitter threshold, and data skewness, HL consistently achieved the highest F_1 score. The AAE and ARE with HL were significantly lower than those observed with DAS, by factors of up to 5.2×, 6.1×, and 7.2× for AAE and 6.5×, 6.9×, and 6.2× for ARE, respectively.

5.6.2 Intersecting data stream. Fig. 12 shows the result when data streams intersect. We observe similar results as the experiment in 5.6.1. Note that the accuracy of CM, ES, USS, DAS, and HL in the intersecting case is lower than the experiment in the disjoint case, which is in line with the conclusion in 5.5. Specifically, when changing memory, heavy hitter threshold, and dataset skewness, HL always has the highest F_1 score. The AAE and ARE were significantly reduced compared to DAS, by factors of up to 2.9×, 2.1×, and 50.5× for AAE and 3.4×, 2.6×, and 41.3× for ARE, respectively.

6 Conclusion

Detecting global heavy hitters in distributed data streams is crucial for various applications. Existing algorithms designed for local detection often perform poorly when scaled to global scenarios. This paper introduces HeavyLocker, which leverages the "separability" feature of data streams to lock and protect heavy hitters precisely, facilitating scalability to global applications. We also present two optimizations and a solid theoretical analysis. Extensive experiments show that HeavyLocker achieves excellent results in local and global heavy hitter detection compared with the state-of-the-art, demonstrating its real-world feasibility.

References

[1] [n. d.]. The CAIDA traces. http://www.caida.org/data/overview/.
[2] [n. d.]. Hash website. http://burtleburtle.net/bob/hash/evahash.html.
[3] [n. d.]. Our open source Github. https://github.com/HeavyLocker/HeavyLockerSketch.
[4] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. 2022. {HeteroSketch}: Coordinating network-wide monitoring in heterogeneous and dynamic networks. *in USENIX NSDI* (2022).
[5] Omid Alipourfard, Masoud Moshref, Yang Zhou, Tong Yang, and Minlan Yu. 2018. A comparison of performance and accuracy of measurement algorithms in software. *in ACM SOSR* (2018).
[6] Ran Ben Basat, Gil Einziger, Shir Landau Feibish, Jalil Moraney, and Danny Raz. 2018. Network-wide routing-oblivious heavy hitters. *in ACM ANCS* (2018).
[7] Ran Ben Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2017. Randomized admission policy for efficient top-k and frequency estimation. *in IEEE INFOCOM* (2017).
[8] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. 2021. SALSA: Self-Adjusting Lean Streaming Analytics. *in IEEE ICDE* (2021).
[9] Ran Ben-Basat, Gil Einziger, Shir Landau Feibish, Jalil Moraney, Bilal Tayh, and Danny Raz. 2021. Routing-oblivious network-wide measurements. *IEEE/ACM Transactions on Networking* 29, 6 (2021), 2386–2398.
[10] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2016. Heavy hitters in streams and sliding windows. *in IEEE INFOCOM* (2016).
[11] Vladimir Braverman, Stephen R Chestnut, Nikita Ivkin, Jelani Nelson, Zhengyu Wang, and David P Woodruff. 2017. Bptree: An l-2 heavy hitters algorithm using constant memory. *in ACM SIGMOD* (2017).
[12] Valerio Bruschi, Ran Ben Basat, Zaoxing Liu, Gianni Antichi, Giuseppe Bianchi, and Michael Mitzenmacher. 2020. DISCOvering the heavy hitters with disaggregated sketches. *in ACM CoNEXT* (2020).
[13] Peiqing Chen, Dong Chen, Lingxiao Zheng, Jizhou Li, and Tong Yang. 2021. Out of many we are one: Measuring item batch with clock-sketch. In *Proceedings of the 2021 International Conference on Management of Data*. 261–273.
[14] Zhuo Cheng, Maria Apostolaki, Zaoxing Liu, and Vyas Sekar. 2024. TRUSTSKETCH: Trustworthy Sketch-based Telemetry on Cloud Hosts. *in NDSS* (2024).
[15] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.
[16] Damu Ding, Marco Savi, Gianni Antichi, and Domenico Siracusa. 2020. An incrementally-deployable P4-enabled architecture for network-wide heavy-hitter detection. *IEEE Transactions on Network and Service Management* 17, 1 (2020), 75–88.
[17] Cormode Graham, Korn Flip, Muthukrishnan Shanmugavelayutham, and Srivastava Divesh. 2003. Finding hierarchical heavy hitters in data streams. *in ACM VLDB* (2003).
[18] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. 2018. Network-wide heavy hitter detection with commodity switches. *in ACM SOSR* (2018).
[19] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S Muthukrishnan, and Jennifer Rexford. 2020. Carpe elephants: Seize the global heavy hitters. *in ACM SPIN* (2020).
[20] Eric R Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A self-designing range filter. *in ACM SIGMOD* (2022).
[21] Haoyu Li, Liuhui Wang, Qizhi Chen, Jianan Ji, Yuhan Wu, Yikai Zhao, Tong Yang, and Aditya Akella. 2023. Chainedfilter: Combining membership filters by chain rule. *in ACM SIGMOD* (2023).
[22] Weihe Li and Paul Patras. 2023. Tight-sketch: A high-performance sketch for heavy item-oriented data stream mining with limited memory size. *in ACM CIKM* (2023).
[23] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. Flowradar: A better netflow for data centers. *in USENIX NSDI* (2016).
[24] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and general sketch-based monitoring in software switches. *in ACM SIGCOMM* (2019).
[25] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. *in ACM SIGCOMM* (2016).
[26] Zirui Liu, Yixin Zhang, Yifan Zhu, Ruwen Zhang, Tong Yang, Kun Xie, Sha Wang, Tao Li, and Bin Cui. 2023. TreeSensing: Linearly Compressing Sketches with Flexibility. *in ACM SIGMOD* (2023).
[27] Lailong Luo, Pengtao Fu, Shangsen Li, Deke Guo, Qianzhen Zhang, and Huaimin Wang. 2023. Ark filter: A general and space-efficient sketch for network flow analysis. *IEEE/ACM Transactions on Networking* (2023).
[28] Antonis Manousis, Zhuo Cheng, Ran Ben Basat, Zaoxing Liu, and Vyas Sekar. 2022. Enabling efficient and general subpopulation analytics in multidimensional data streams. *in ACM VLDB* (2022).
[29] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. *in Springer ICDT* (2005).
[30] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2016. Trumpet: Timely and precise triggers in data centers. *in ACM SIGCOMM* (2016).
[31] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. 2022. {SketchLib}: Enabling efficient sketch-based monitoring on programmable switches. *in USENIX NSDI* (2022).
[32] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. 2023. Sketchovsky: Enabling ensembles of sketches on programmable switches. *in USENIX NSDI* (2023).
[33] Prashant Pandey, Shikha Singh, Michael A Bender, Jonathan W Berry, Martín Farach-Colton, Rob Johnson, Thomas M Kroeger, and Cynthia A Phillips. 2020. Timely reporting of heavy hitters using external memory. *in ACM SIGMOD* (2020).
[34] David MW Powers. 1998. Applications and explanations of Zipf's law. *in EMNLP-CoNLL* (1998).
[35] Alex Rousskov and Duane Wessels. 2004. High-performance benchmarking with Web Polygraph. *Software: Practice and Experience* 34, 2 (2004), 187–211.
[36] Rana Shahout, Roy Friedman, and Ran Ben Basat. 2023. Together is Better: Heavy Hitters Quantile Estimation. *in ACM SIGMOD* (2023).
[37] Qilong Shi, Chengjun Jia, Wenjun Li, Zaoxing Liu, Tong Yang, Jianan Ji, Gaogang Xie, Weizhe Zhang, and Minlan Yu. 2024. BitMatcher: Bit-level Counter Adjustment for Sketches. *in IEEE ICDE* (2024).
[38] Qilong Shi, Yuchen Xu, Jiuhua Qi, Wenjun Li, Tong Yang, Yang Xu, and Yi Wang. 2023. Cuckoo Counter: Adaptive Structure of Counters for Accurate Frequency and Top-k Estimation. *IEEE/ACM Transactions on Networking* (2023).
[39] Lu Tang, Qun Huang, and Patrick PC Lee. 2019. Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams. *IEEE INFOCOM* (2019).
[40] Lu Tang, Qun Huang, and Patrick PC Lee. 2020. A fast and compact invertible sketch for network-wide heavy flow detection. *IEEE/ACM Transactions on Networking* 28, 5 (2020), 2350–2363.
[41] Daniel Ting. 2018. Data sketches for disaggregated subset sum and frequent item estimation. *in ACM SIGMOD* (2018).
[42] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: a learning-enhanced range filter. *in ACM VLDB* (2022).
[43] Feiyu Wang, Qizhi Chen, Yuanpeng Li, Tong Yang, Yaofeng Tu, Lian Yu, and Bin Cui. 2023. JoinSketch: A Sketch Algorithm for Accurate and Unbiased Inner-Product Estimation. *in ACM SIGMOD* (2023).
[44] Yuhan Wu, Zhuochen Fan, Qilong Shi, Yixin Zhang, Tong Yang, Cheng Chen, Zheng Zhong, Junnan Li, Ariel Shtul, and Yaofeng Tu. 2022. She: A generic framework for data stream mining over sliding windows. *in ICPP* (2022).
[45] Kaicheng Yang, Sheng Long, Qilong Shi, Yuanpeng Li, Zirui Liu, Yuhan Wu, Tong Yang, and Zhengyi Jia. 2023. Sketchint: Empowering int with towersketch for per-flow per-switch measurement. *IEEE Transactions on Parallel and Distributed Systems* (2023).
[46] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. *in ACM SIGCOMM* (2018).
[47] Minlan Yu. 2019. Network telemetry: towards a top-down approach. *in ACM SIGCOMM Computer Communication Review* 49, 1 (2019), 11–17.
[48] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. *in USENIX NSDI* (2013).
[49] Hailin Zhang, Zirui Liu, Boxuan Chen, Yikai Zhao, Tong Zhao, Tong Yang, and Bin Cui. 2024. CAFE: Towards Compact, Adaptive, and Fast Embedding for Large-scale Recommendation Models. *in ACM SIGMOD* (2024).
[50] Yinda Zhang, Peiqing Chen, and Zaoxing Liu. 2024. {OctoSketch}: Enabling {Real-Time}, Continuous Network Monitoring over Multiple Cores. *in USENIX NSDI* (2024).
[51] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. 2021. DHS: Adaptive Memory Layout Organization of Sketch Slots for Fast and Accurate Data Stream Processing. *in ACM SIGKDD* (2021).
[52] Fuheng Zhao, Punnal Ismail Khan, Divyakant Agrawal, Amr El Abbadi, Arpit Gupta, and Zaoxing Liu. 2023. Panakos: Chasing the Tails for Multidimensional Data Streams. *in ACM VLDB* (2023).
[53] Yikai Zhao, Wenchen Han, Zheng Zhong, Yinda Zhang, Tong Yang, and Bin Cui. 2023. Double-anonymous sketch: Achieving fairness for finding global top-k frequent items. *in ACM SIGMOD* (2023).
[54] Zheng Zhong, Shen Yan, Zikun Li, Decheng Tan, Tong Yang, and Bin Cui. 2021. Burstsketch: Finding bursts in data streams. *in ACM SIGMOD* (2021).
[55] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. 2018. Cold filter: A meta-framework for faster and more accurate stream processing. *in ACM SIGMOD* (2018).
[56] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. 2020. Newton: Intent-driven network traffic monitoring. *in ACM CoNEXT* (2020).

Appendix

A Pseudo-Code

Algorithm 1: Insert(e)

Input: an item e

θ is the heavy hitter threshold (usually $\leq 0.1\%$, also the dynamic lock threshold);
$i = hash(e)$; $item_{num}++$;
if $B[i][0].cnt \geq item_{num} \times \theta$ **then**
  $B[i].lock = 1$;
end
else
  $B[i].lock = 0$;
end
for $j = d-1$ *to* 0 **do**
  if $B[i][j].id == e.key$ **then**
    $B[i][j].cnt++$;
    Swap $B[i][j]$ and $B[i][j+1]$ when necessary to maintain order in the bucket;
    return;
  end
  else if $B[i][j]$ *is empty* **then**
    $B[i][j].id = e.key$;
    $B[i][j].cnt = 1$;
    return;
  end
end
if $B[i].lock == 0$ **then**
  $RAP\_Replacement(B[i][0], e)$;
end
Function `RAP_Replacement`$(B[i][j], e)$**:**
  if $(rand()\ \%\ (B[i][j].cnt + 1)) == 0$ **then**
    $B[i][j].id = e.key$;
    $B[i][j].cnt++$;
  end

Algorithm 2: Query()

Output: A vector containing all heavy hitters

θ is the heavy hitter threshold;
N is the sum of frequencies of all items in the data stream;
$vector\langle key, value\rangle\ ans$;
for $i = 1$ *to* w **do**
  for $j = 0$ *to* $d-1$ **do**
    if $B[i][j].cnt \geq \theta \times N$ **then**
      $ans.push\_back(\ \langle B[i][j].id, B[i][j].cnt\rangle\ )$;
    end
  end
end
return ans;

Algorithm 3: Merge(V)

Input: A vector V containing n HeavyLockers

Output: One merged HeavyLocker

n is the number of HeavyLocker or data stream;
$HeavyLocker\ result$;
$StreamSummary\ q$;
for $v = 1$ *to* n **do**
  for $i = 1$ *to* w **do**
    for $j = 0$ *to* $d-1$ **do**
      $bool\ p = q.find(\ \langle B[i]^{(v)}[j].e\rangle$;
      if p **then**
        $q.relink(\ p,\ \langle B[i]^{(v)}[j].e, B[i]^{(v)}[j].cnt\rangle\ )$;
      end
      else
        $q.push(\ \langle B[i]^{(v)}[j].e, B[i]^{(v)}[j].cnt\rangle\ )$;
      end
    end
    for $k = 0$ *to* $d-1$ **do**
      $result[i]^{(v)}[j].e = q.top().e$;
      $result[i]^{(v)}[j].cnt = q.top().cnt$;
      $q.pop()$;
    end
  end
  $q.clear()$;
end
return $result$;

B Experiment Setup

B.1 Test Platform

We conducted our experiments using a machine equipped with an Intel $i7-9700CPU@3.0GHz$ and 16GB DRAM, running Ubuntu 20.04. To mitigate CPU jitter errors, we computed average results based on 10 runs for each evaluation.

B.2 Datasets

We use 2 kinds of datasets in experiments.

• **CAIDA Datasets:** We use the CAIDA trace collected in Equinix-Chicago monitor from CAIDA [1]. This trace is identical to the one used in the Elastic Sketch paper. The monitoring period for this trace was 5 seconds, during which it captured 165K kinds of items and 2.49M items in total. The largest item observed in this trace was 17K in size.

• **Zipf Datasets (synthetic):** We generate a series of synthetic traces that follow the Zipf [34] distribution using Web Polygraph [35]. The skewness of the traces ranges from 0.6 to 2.1. Each trace contains 32.0M items in total. The number of items decreases as the skewness increases. When the skewness = 0.6, there are 1M kinds of items; when the skewness = 2.1, there are 10K kinds of items. The maximum item size ranges from 62 to 2.22M.

B.3 Comparing Algorithms

We implement our HeavyLocker (HL) in C++, and compare our results with CM sketch+heap (CM) [15], Elastic sketch (ES) [46], MV Sketch (MV) [40], Unbiased SpaceSaving (USS) [41], and Double-anonymous Sketch (DAS) [53]. For CM, ES, MV, and DAS, we used the open-source code in their original paper; For USS, we implemented it by ourselves.

As for the configuration of these methods, we set their data structures to have a $depth = 4$ and use four 32-bit Bob hash functions [2] for item mapping. In addition, except for exceptional needs (for example, the counter of the light part of ES is set to 8 bits), we uniformly set the item ID and counter fields to 32 bits to ensure fairness. The optimal parameters of our HeavyLocker will be introduced later in the following subsection.

When it comes to network-wide heavy hitter detection, we implemented the aggregation functions of the five works mentioned above and compared them with HeavyLocker's. For ES and MV, we implement the aggregation functions mentioned in their source codes. For DAS and USS, we implement the aggregation function by merging the buckets rather than the counters, since their codes lack the discussion of the aggregation algorithm. For CM, we use sum merging and maximum merging [46] in different scenarios, just like what ES did to its light part.

B.4 Tasks and Metrics

We perform heavy hitter detection (reporting items whose sizes are larger than a predefined threshold), and the following metrics are considered.

Speed: Speed is used to measure the processing speed of the insertion and is estimated by the algorithm's running time. It is estimated by the formula $\frac{N}{T}$, where N is the number of items, and T is the running time. We use millions of packets per second (Mpps) to represent throughput. All the experiments are repeated 10 times to minimize accidental deviations.

AAE: AAE is defined as $\frac{1}{|E|}\sum_{(e_i \in E)} |f_i - \tilde{f_i}|$, where f_i is the real frequency of item e_i, $\tilde{f_i}$ is the estimated frequency, and the E is the query set. In heavy hitter detection, the query set is the reporting set, that is, all heavy hitters reported by each algorithm.

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

Precision: Fraction of true heavy hitters reported over all reported items.

Recall: Fraction of true heavy hitters reported over all true heavy hitters;

F_1 **score:** $\frac{2 \times Precision \times Recall}{Precision + Recall}$. $F1$ score is an indicator to measure the comprehensive accuracy of heavy hitter detection.