# Hypersistent Sketch: Enhanced Persistence Estimation via Fast Item Separation

Lu Cao*
*Harbin Institute of Technology (Shenzhen)*

Qilong Shi*
*Tsinghua University*

Weiqiang Xiao
*Harbin Institute of Technology (Shenzhen)*

Nianfu Wang
*Harbin Institute of Technology (Shenzhen)*

Wenjun Li
*Peng Cheng Laboratory*

Zhijun Li
*Harbin Institute of Technology*

Weizhe Zhang
*Harbin Institute of Technology (Shenzhen)*

Mingwei Xu
*Tsinghua University*

***Abstract*—Efficient data stream processing, particularly for persistence estimation, is crucial in handling high-velocity data streams characterized by skewed distributions of item frequencies. Unlike more straightforward frequency metrics, persistence captures items' recurrence across multiple time windows, requiring nuanced processing approaches. In response, we introduce the Hypersistent Sketch, an algorithm that significantly enhances persistence estimation through innovative filtering techniques. Our design incorporates a Cold Filter to address the skewed nature of data streams where a few high-frequency (hot) items dominate. This filter allows for differential treatment by using smaller counters for most low-frequency (cold) items, thus conservatively allocating memory resources that would otherwise be sized uniformly based on hot items. However, the Cold Filter can reduce throughput due to its segregative processing. To mitigate this, we implement a Burst Filter, which optimizes the processing of hot items. The Burst Filter significantly improves throughput by preventing repeated insertions within a single window—where persistence increases by at most one—and deferring the insertion until the window's end. Comparative evaluations demonstrate that the Hypersistent Sketch outperforms existing solutions like the On-Off Sketch, offering up to 3 times improved throughput while maintaining competitive accuracy and substantially reducing memory usage in handling large-scale data streams.**

***Index Terms*—Data stream, Persistence estimation, Approximate algorithm, Sketch.**

## I. Introduction

In the realm of data stream processing, an important characteristic – persistence, has received growing attention. Given an item $e$ and a data stream with $T$ non-overlapping and contiguous time windows, the persistence of $e$ is defined as the number of time windows where $e$ appears. The persistent items—those that appear consistently over time—are paramount. Their identification and quantification are critical across various applications, from security-related tasks like intrusion detection and fraud prevention to operational enhancements in sectors such as transportation analytics [3], [4], [5]. For example, persistent network threats may be strategically deployed at low frequencies to evade traditional detection systems, posing serious security risks [6], [7]. Similarly, in digital advertising, sophisticated schemes utilize automated scripts for click fraud, repeatedly engaging with ads to illegitimately boost revenue. Beyond malicious activities [8], [9], understanding item's persistence is also invaluable in identifying regular patterns [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], optimizing operations [20], [21], and improving decision-making processes in dynamic environments like traffic flow management [22], [23], [24].

To bolster these applications in persistence estimation, one-pass Sketch algorithms are seen as particularly promising due to their efficiency and effectiveness [25], [26]. Recent advancements have focused on refining Sketch algorithms in various domains, including memory usage [27], [28], accuracy [29], [30], and processing speed [31], [32]. Yet, these enhancements often struggle to balance the trade-offs between memory usage and accuracy, a critical aspect given the increasing volume of data and the constraints of limited computational and memory resources [33], [34]. Taking the well-known On-Off Sketch [35] as an example, its primary data structure is similar to the traditional frequency estimation algorithm, the Count-Min Sketch, consisting of multiple columns of fixed-size counter arrays. In real-world scenarios, data streams exhibit high skewness, meaning the persistence of most items is relatively low (e.g., $\leq 5$, termed as cold items), while a minority of items show high persistence (referred to as hot items) [36], [37], [38], [39], [40], [41], [42], [43]. To accurately capture the persistence of hot items, the On-Off Sketch must allocate larger memory blocks (e.g., 32-bit counters) for each counter. This approach, however, leads to substantial memory wastage as many counters (those capturing cold items) do not utilize their higher bits.

We posed the question: Is it possible to create a persistence estimation algorithm that surpasses previous algorithms in both accuracy and speed? In response, we developed the Hypersistent Sketch, which outperforms the well-known algorithm, the On-Off Sketch, across all metrics. The design of the Hypersistent Sketch includes:

*Lu Cao and Qilong Shi are co-first authors, and they conducted this work under the guidance of corresponding authors Wenjun Li and Weizhe Zhang. Lu Cao, Nianfu Wang, Zhijun Li and Weizhe Zhang are also with Peng Cheng Laboratory. This work was supported in part by the Major Key Project of Peng Cheng Lab (PCL2023A06), the Joint Funds of the National Natural Science Foundation of China (U22A2036), the National Natural Science Foundation of China (62221003, 62402141, 62102203), the Science and Technology Innovation Project of Guangdong (2023TQ07X362), the NSFC/RGC Collaborative Research Scheme (62461160332 & CRS_HKUST602/24), Shenzhen Colleges and Universities Stable Support Program (GXWD20220817124251002, GXWD20231129102636001), Shenzhen Stable Supporting Program (General Project) (GXWD20231130110352002), Guangdong Basic and Applied Basic Research Foundation (2023A1515110271, 2025A1515011785). The source code of this paper can be downloaded from the Website [1] and the GitHub [2].

1) **Triple-Stage Structure:** It consists of Burst Filter, Cold Filter, and Hot Part. The Hot Part, akin to the reversible On-Off Sketch, stores complete IDs and persistence levels to facilitate finding persistent items directly.
2) **Cold Filter for Memory Efficiency:** It preemptively identifies cold items, allowing only hot items to proceed to the Hot Part. By allocating different counter sizes to these two sections, we effectively increase the number of counters under the same total memory, thereby significantly enhancing accuracy.
3) **Burst Filter for Speed Enhancement:** Created to tackle speed issues introduced by the Cold Filter, which, while increasing accuracy, reduces overall throughput. We observed that an item's persistence could only be incremented once within a single time window. Thus, items appearing multiple times within a window are retained in the Burst Filter to avoid unnecessary updates to subsequent structures. At the end of each window, all items from the Burst Filter are uniformly updated to the Cold Filter and Hot Part. This strategy dramatically increases throughput, potentially even surpassing that of the original On-Off Sketch.

To evaluate real-world performance, we implemented the Hypersistent Sketch on CPU platforms, conducting tasks that include persistence estimation and identifying persistent items. The Hypersistent Sketch achieves significantly better results than On-Off Sketch across all datasets, with error rates improved by up to an order of magnitude and throughput twice as fast. These experimental results demonstrate the Hypersistent Sketch's real-world feasibility and scalability.

## II. Background

### A. Problem Statement

• **Data Stream Model:** Consider a time-sequenced data stream $S$, represented as a set of tuples $\{(e_1, t_1), (e_2, t_2), \ldots, (e_N, t_N)\}$, where each tuple $(e_i, t_i)$ consists of a data item identifier $e_i$ (such as a quintuple or source IP in network packets) and its arrival time $t_i$ (the $t_i$ is monotonically increasing). The set of data items $E$ is defined as $E = \{e \mid \exists t \text{ such that } (e, t) \in S\}$.

• **Window:** The time range $[t_1, t_N]$ of the data stream $S$ is evenly divided into $w$ time windows, each of size $R = (t_N - t_1)/w$. These windows are represented as a set $W = \{[t_1, t_1 + R], [t_1 + R, t_1 + 2R], \ldots, [t_1 + (w-1)R, t_N]\}$.

• **Persistence:** For a data stream $S$, the persistence of an item $e_i$ is defined as the number of windows in which it appears, denoted $p_{e_i}$. It is formally defined as:

$$p_{e_i} = |\{[l, r] \mid (e_i, t) \in S \text{ and } t \in [l, r]\}|.$$

• **Persistence Estimation:** For all items in $E$, the task is to estimate their persistence.

• **Finding Persistent Item:** We define a threshold $\alpha$, where an item $e_i$ is considered $\alpha$-persistent if its persistence $p_{e_i} \geq \alpha \cdot w$, that is, it appears in at least $\alpha$ times the total number of windows (the maximum persistence). The goal is to find all such $\alpha$-persistent items.

### B. Related Work

This subsection reviews work related to estimating persistence in data streams, a concept closely related to frequency estimation. The primary distinction between these two lies in their counting approach: while frequency estimation aggregates every occurrence of an item, persistence estimation counts an item only once per window, regardless of how many times it appears within that window.

This difference has led to innovative adaptations of existing frequency estimation algorithms [44], [26], [45], [46], [47]. One straightforward approach is to modify a frequency estimation Sketch by incorporating a deduplication step within each window, ensuring each item updates the Sketch's counter at most once per window. The PIE (Persistence via Incremental Estimation) algorithm [36] exemplifies this approach by combining a Count-Min (CM) Sketch [48] for frequency estimation with a Bloom Filter (BF) [49] for window-based deduplication. Here's how it works:

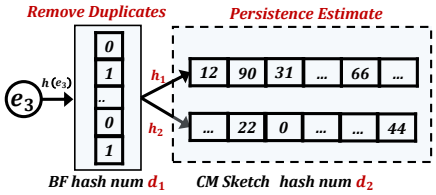



Fig. 1: Insertion of the strawman solution.

**PIE Algorithm**: As shown in Figure 1, at the start of each window, when an item $e$ arrives, it is hashed by $d_1$ hash functions to $d_1$ bits in a Bloom Filter. If any of these $d_1$ bits is 0, indicating $e$ has not yet appeared in the current window, the bits are set to 1, and $e$ is added to the CM Sketch on the right. The CM Sketch consists of $d_2$ rows of arrays, each containing $w$ counters. The item $e$ is hashed by $d_2$ different hash functions, each mapping to one counter per row, and these counters are incremented. If all $d_1$ bits in the Bloom Filter are 1, it indicates that $e$ has already been accounted for in the current window, and no further action is taken.

When querying the persistence of an item $e_2$, the algorithm checks the CM Sketch and returns the minimum value among the counters $e_2$ is mapped to by its $d_2$ hash functions.

**Limitations**: The PIE method has its shortcomings. First, the Bloom Filter is prone to false positives, which may report that an item has appeared when it has not. This can prevent a new item from being added to the CM Sketch, leading to underestimating its persistence. Additionally, hash collisions in the CM Sketch can cause counters to be incremented multiple times within a single window, even though an item's persistence should increase by only one per window. This results in a significant overestimation of persistence.

The well-known algorithm, the On-Off Sketch [35], made several improvements to address the inherent problems observed in previous approaches. There are two versions of the On-Off Sketch: one for persistence estimation (Version 1), which does not store item IDs, and another for finding persistent items (Version 2), which does store item IDs. Below, we discuss each version.

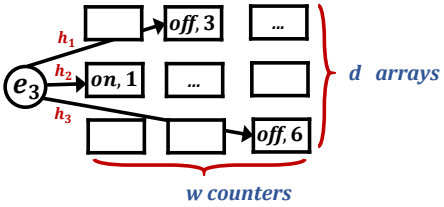



Fig. 2: An illustration of the On-Off Sketch framework.

**On-Off Sketch Version 1:** As shown in Figure 2, this version resembles the structure of a CM Sketch but adds a one-bit flag to each counter to indicate its state as either "on" or "off". This flag determines whether the counter can be updated. When an item $e$ arrives, it is mapped by $d$ hash functions to $d$ counters. Only those counters that are "on" are updated, after which their flags are set to "off". At the end of each window and the start of a new one, all flags are reset to "on". This mechanism ensures that within a window, a counter is incremented at most once, aligning with the definition of persistence and significantly reducing the risk of overestimation.

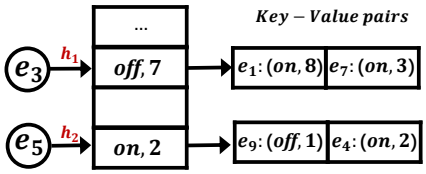
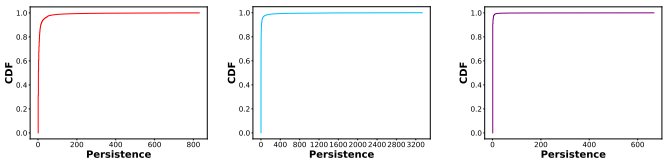



Fig. 3: On-Off Sketch for finding persistent items.

**On-Off Sketch Version 2:** This version consists of an array of buckets, each containing multiple cells, as shown in Figure 3. Each cell has the structure $\langle ID, flag, counter \rangle$, and each bucket has a global cell $\langle flag, counter \rangle$. When an item $e$ arrives:

1) If $e$ is already in the bucket, its counter is updated based on the on/off flag.
2) If $e$ is not in the bucket and there is an empty cell, $e$ is added as $\langle e, off, 1 \rangle$.
3) If $e$ is not in the bucket and the bucket is full, the global cell is updated. If the global cell's counter exceeds the minimum counter in the bucket's cells, $e$ is swapped into a bucket cell.

**Limitations:** Both versions do not account for the high skewness typical of natural data streams, where only a few items (hot items) have high persistence, while the majority (cold items) have persistence less than 5 (As shown in Figure 4, the CDF plots of persistence for the three datasets validate this point). Without prior knowledge of which items are hot, all counters must be large enough (e.g., 32-bit) to accommodate potential hot items without risk of overflow, leading to low memory utilization. Additionally, in Version 2, many cold items and resulting hash collisions can cause frequent swaps between the global counter and bucket cells, severely overestimating persistence. This highlights the motivation behind our design: to separate cold and hot items within the data stream and handle them differently, which could greatly improve memory efficiency and reduce errors without sacrificing speed.




(a) CDF-CAIDA (b) CDF-Campus (c) CDF-MAWI

Fig. 4: Persistence variation across datasets.

## III. Hypersistent Sketch Design

In this section, we first show the details of our designed Hypersistent Sketch, including algorithm framework, data structures, and basic operations. After that, we give a specific running example of Hypersistent Sketch. Finally, we discuss the optimization of Hypersistent Sketch.

### A. Algorithm Framework Overview

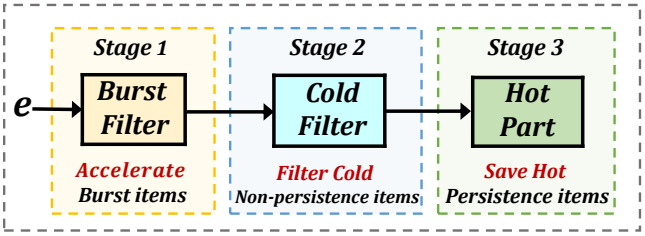



Fig. 5: Algorithm framework of Hypersistent Sketch.

Figure 5 illustrates the overall structure of the Hypersistent Sketch, which is divided into three stages: the Burst Filter for acceleration, the Cold Filter for storing cold items, and the Hot Part for storing hot items. In the following sections, we will introduce these components from right to left, aligning with the design sequence of our approach.

### B. Hot Part (Stage 3)

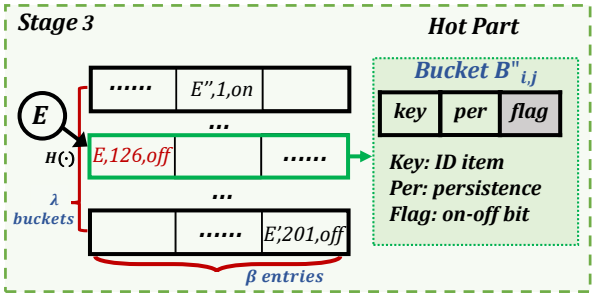



Fig. 6: Data structure of the Hot Part.

Figure 6 illustrates the data structure of the Hot Part, which consists of $\lambda$ buckets, each containing $\beta$ entries. Each entry stores an item's ID, a persistence value, and an on-off flag indicating whether the entry has been updated in the current time window (as each entry can be updated at most once within a single window). When an item $E$ arrives, it is hashed into the $H(E)$-th bucket $B[H(E)]$ via the hash function $H(\cdot)$. The entries in the bucket are then sequentially checked under the following three cases:

**Algorithm 1:** Insert_Stage3(e) ▷ Hot Part

**Input:** An item $e$, entries within buckets containing IDs $keys$, persistence $per$, and status $flag$

$replace \leftarrow null$;
$bucket \leftarrow L_3[H(e)]$;
**for** $entry \in bucket$ **do**
  **if** $entry.key = null \ || \ (entry.key = e \ \&\& \ entry.flag = on)$ **then**
    $entry.key \leftarrow e$;
    $entry.flag \leftarrow off$;
    $entry.per \leftarrow entry.per + 1$;
    **return**; ▷ If $e$ is found
  **end**
  **if** $replace = null \ || \ entry.per < replace.per$ **then**
    $replace \leftarrow entry$; ▷ If $e$ is not found
  **end**
**end**
**if** $H(e) \ \% \ (replace.per + 1) = 0$ **then**
  $replace.key \leftarrow e$;
  $replace.per \leftarrow replace.per + 1$;
  $replace.flag \leftarrow off$;
**end**
▷ Replace with probability $\frac{1}{per+1}$

1) If $B[H(E)]$ contains $E$: If the flag is on, the persistence value is incremented by 1, and the flag is set to off. If flag is off, no operation is performed.
2) If $B[H(E)]$ does not contain $E$ but has an empty entry: The item $\langle E, 1, \text{off} \rangle$ is inserted into the empty entry.
3) If $B[H(E)]$ neither contains $E$ nor has any empty entries: The entry with the smallest persistence value is probabilistically replaced, following the replacement strategy outlined in Algorithm 1.

• **Limitations**: In real-world data streams, the frequency distribution is typically highly skewed, which also applies to the persistence distribution (as shown in Figure 4). Specifically, most items exhibit low persistence (cold items), while a small subset of items demonstrate high persistence (hot items). This skewness introduces a key challenge for the data structure illustrated in Figure 6: in the aforementioned case 3, some potential hot items (which are critical in many tasks) might be replaced by a large number of cold items, leading to a decline in the accuracy of tasks aimed at identifying persistent items.

A straightforward and intuitive solution to address these challenges is to separate hot and cold items and store them independently. This is precisely the approach implemented by the Cold Filter in Stage 2.

### C. Cold Filter (Stage 2)

Figure 7 illustrates the structure of the Cold Filter. It consists of several arrays (two in this paper), $L_1$ and $L_2$, where each cell in the arrays comprises a persistence counter and an on-off flag. When an item $e'$ arrives, it is first hashed into multiple cells in $L_1$ (two in the example shown). We adopt an update strategy similar to the CU Sketch [50]: among the hashed cells

**Algorithm 2:** Insert_Stage2(e) ▷ Cold Filter

**Input:** An item $e$, buckets in $L_1, L_2$ with persistence counters $per$ and status flags $flag$
**Output:** Insertion in Stage 2

$v1 = min_{1 \le i \le d_1} L_1[i][h_i(e)].per$;
**if** $v1 < \Delta_1$ **then**
  **for** $i$ *from* $1 \rightarrow d_1$ **do**
    $b_1 \leftarrow L_1[i][h_i(e)]$;
    **if** $b_1.per = v1 \ \&\& \ b_1.flag = on$ **then**
      $b_1.per \leftarrow b_1.per + 1$;
      $b_1.flag \leftarrow off$;
    **end**
  **end**
  **return** $true$; ▷ Successful insertion in $L_1$
**end**
$v2 = min_{1 \le i \le d_2} L_2[i][g_i(e)].per$;
**if** $v2 < \Delta_2$ **then**
  **for** $i$ *from* $1 \rightarrow d_2$ **do**
    $b_2 \leftarrow L_2[i][g_i(e)]$;
    **if** $b_2.per = v2 \ \&\& \ b_2.flag = on$ **then**
      $b_2.per \leftarrow b_2.per + 1$;
      $b_2.flag \leftarrow off$;
    **end**
  **end**
  **return** $true$; ▷ Successful insertion in $L_2$
**end**
**return** $false$;

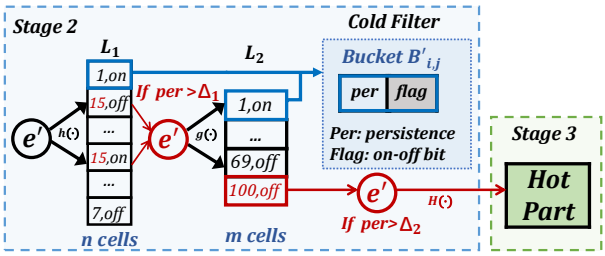



Fig. 7: Data structure of the Cold Filter.

with the flag set to on, the one with the smallest persistence value is selected. The processing follows two cases:

1) If the persistence value does not exceed a threshold $\Delta_1$, the persistence value is incremented by 1, and the flag is set to off.
2) If the persistence value exceeds $\Delta_1$, $e'$ is inserted into $L_2$ using a similar strategy as for $L_1$.

When inserting into $L_2$, if the minimum persistence value of the cells mapped by $e'$ in $L_2$ exceeds another threshold $\Delta_2$, $e'$ is considered a hot item with high persistence. It is then inserted into the subsequent Hot Part, as described in the previous subsection. This design retains cold items within $L_1$ and $L_2$, while only hot items proceed to the Hot Part. This separation effectively addresses the limitations mentioned in the previous subsection by clearly distinguishing between cold and hot items. The pseudocode for this process is presented in Algorithm 2.

**Algorithm 3:** Insert_Stage1(e) ▷ Burst Filter

**Input:** An item $e$, cells in bucket array $B$ containing IDs $keys$

**Output:** Insertion success in Stage 1

```
for cell ∈ B_{f(e)} do
    if cell.id = null || cell.id = e.id then
        cell.id = e.id;
        return true;
    end
end
return false;
```

**Algorithm 4:** Insert(e)

**Input:** Item $e$

```
if Insert_Stage1(e) then
    return;
end
if Insert_Stage2(e) then
    return;
end
Insert_Stage3(e);
return;
```

• **Limitations**: Although the proposed design addresses the separation of hot and cold items, and improves the accuracy of persistence estimation, it introduces a new challenge: additional hash computations are required to map items into the corresponding cells of $L_1$ and $L_2$. Experiments (as shown in Section V) indicate that using two hash functions for each layer yields optimal results. In contrast, the Hot Part only requires a single hash computation. Consequently, introducing the Cold Filter greatly reduces the algorithm's throughput.

To address this issue, we leverage a unique property of persistence estimation: **each cell in each array ($L_n$) can be updated at most once within a single window.** This means that for a given item $e$, only its first occurrence in a window is meaningful (as it updates the cells it maps to), while subsequent occurrences have no effect. Thus, we can skip hash computations for $e$'s repeated appearances.

To implement this optimization, we introduce an auxiliary data structure to record the set of items observed within the current window. At the end of the window, we iterate through this set and insert each item into the Cold Filter. This mechanism is implemented by the Stage 1 Burst Filter, which effectively mitigates the throughput reduction caused by the Cold Filter.

• **False Positive Control**: The Cold Filter behaves like a Bloom Filter, producing false positives but no false negatives. To minimize FPR (False Positive Rate) while maintaining system performance, memory is allocated between the Cold Filter and Hot Part in a balanced ratio (e.g., 3:2), reducing collisions and trying to avoid cold items are misclassified as hot. Experimental results in Section V.C confirm that the 3:2 memory ratio maintains an FPR below 0.1%, balancing throughput and accuracy across diverse data distributions.

### *D. Burst Filter (Stage 1)*

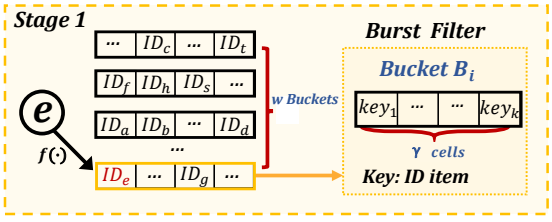



Fig. 8: Data structure of the Burst Filter.

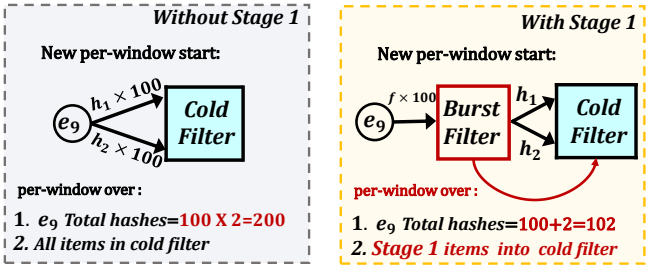
Figure 8 illustrates the structure of the Burst Filter. It consists of $w$ buckets, each containing $\gamma$ cells, where each cell stores the ID of an item. When an item $e$ arrives, it is hashed into the $f(e)$-th bucket $B[f(e)]$ using the hash function $f(\cdot)$. The bucket is then traversed, and the following cases are handled:

1) If $e$ is already in the bucket, no action is taken.
2) If $e$ is not in the bucket and there is an empty cell, $e$ is inserted into the empty cell.
3) If $e$ is not in the bucket and there are no empty cells, $e$ is inserted into the Cold Filter in Stage 2.

At the end of each time window, all items in the buckets are iterated over and inserted into the Stage 2 Cold Filter. The Burst Filter is then cleared by emptying all buckets.

• **Example Explanation**: To demonstrate why adding an additional Burst Filter layer can reduce the number of hash computations to increase the throughput, we present an illustrative example. As shown on the left side of Figure 9, consider only the $L_1$ layer of the Cold Filter, which uses two hash functions, $h_1$ and $h_2$. Suppose an item $e_9$ arrives 100 times within the current window. This would require $100 \times 2 = 200$ hash computations.

With the Burst Filter added, as shown on the right side of Figure 9, we first compute the hash function $f(\cdot)$ 100 times to insert $e_9$ into the Burst Filter. At the end of the window, when traversing the items in the Burst Filter, $e_9$ is inserted into the Cold Filter, requiring only 2 additional hash computations (for $h_1$ and $h_2$). This results in a total of 102 hash computations, saving $200 - 102 = 98$ hash computations. The difference becomes even more significant when the Cold Filter employs more hash functions.




(a) Without Stage 1 (b) With Stage 1

Fig. 9: Burst Filter for hash reduction.

### E. Complete Insertion of Hypersistent Sketch

The complete insertion workflow of the Hypersistent Sketch spans its three stages, as described in Figure 5 and the preceding sections. The process proceeds from Stage 1 to Stage 3 as follows: When an item $e$ arrives, it is first inserted into the Burst Filter. At the end of each time window, all items stored in the Burst Filter are retrieved and sequentially inserted into the Cold Filter. If an item's persistence estimate exceeds the threshold $\Delta_1 + \Delta_2$ in the Cold Filter (indicating an overflow), it is inserted into the Hot Part. The pseudocode for this process is presented in Algorithm 4.

### F. Persistence Query

After the data stream has ended, the persistence of an item $e$ in the Hypersistent Sketch is queried through the Cold Filter and Hot Part, following these steps:

1) Query $L_1$ of the Cold Filter: Hash $e$ into multiple cells of $L_1$ using the corresponding hash functions, and record the minimum persistence value $v_1$ among these cells. If $v_1 < \Delta_1$, return $v_1$.
2) Otherwise, if $v_1 \geq \Delta_1$, proceed to $L_2$: Hash $e$ into multiple cells of $L_2$ using the corresponding hash functions, and record the minimum persistence value $v_2$ among these cells. If $v_2 < \Delta_2$, return $\Delta_1 + v_2$.
3) Otherwise, if $v_2 \geq \Delta_2$, query the Hot Part: Retrieve the value $v_3$ of $e$ in the Hot Part (via collision-free ID matching), and return $\Delta_1 + \Delta_2 + v_3$.

This staged approach achieves $O(1)$ average query latency by: (1) filtering 98.7% cold items at $L_1$ (per our analysis in Section IV.B), and (2) It can be expected that if $e$ is a cold item, its persistence is determined within $L_1$. If its persistence is slightly larger, it will be handled in $L_2$. Only when $e$ is a hot item will it be processed in the Hot Part, (3) The Hot Part resolves hash collisions deterministically for items with $v_2 \geq \Delta_2$ through full ID storage, which stores the full IDs of items. This avoids hash collisions for hot items and supports tasks such as persistent item detection, which require complete item IDs. We support two query modes: **After-window querying** returns fully updated persistence values, reflecting all insertions within the window; **In-window querying** requires an additional hash check for items still being processed in the Burst Filter (Stage 1). Since the vast majority of flows are handled at the L1 layer, only a small fraction of flows enter the L2 layer, and even fewer hot flows enter the Hot Part. As a result, both query modes maintain $O(1)$ average query latency. The pseudocode is presented in Algorithm 5.

### G. A Running Example

Figure 10 illustrates a running example of the Hypersistent Sketch. The parameters for each layer are as follows:

- Burst Filter: Uses a single hash function to locate a bucket, with each bucket containing 4 entries.
- Cold Filter: For $L_1$ layer, we use 2 hash functions, with a threshold $\Delta_1 = 15$. For $L_2$ layer, we use 2 hash functions, with a threshold $\Delta_2 = 100$.
- Hot Part: Uses a single hash function to locate a bucket, with each bucket containing 3 cells.

**Algorithm 5:** Query(e)

**Input:** Item $e$, Item number $packet\_seq$, Window length $window\_len$
**Output:** The persistence of $e$

```
ret ← 0;
if packet_seq % window_len ≠ 0 then
    for cell ∈ B_{f(e)} do
        if cell.id = e.id then
            ret ← 1;        ▷ find out whether the current item exists in the Brust Filter.
        end
    endfor
end
v1 ← min_{1≤i≤d1}(L1[i][h_i(e)]);
if v1 < Δ1 then
    return ret + v1;                    ▷ L1 of Cold Filter
end
ret ← ret + v1;
v2 = min_{1≤i≤d2}(L2[i][g_i(e)]);
if v2 < Δ2 then
    return ret + v2;                    ▷ L2 of Cold Filter
end
ret ← ret + v2;
bucket ← L3[H(e)];
for entry ∈ bucket do
    if entry.key = e then
        ret ← ret + entry.per;          ▷ Hot Part
    end
endfor
return ret;
```

Examples of insertion into each layer are shown below.

*1) Inserting into Burst Filter:* Note that every item will first enter the Burst Filter.

- **Case 1**: Insert $e_1$. We find that the corresponding bucket does not contain $e_1$ but has an empty cell, so $e_1$ is placed into the bucket.
- **Case 2**: Insert $e_2$. We find that $e_2$ already exists in the corresponding bucket, so no action is taken.
- **Case 3**: Insert $e_3$. We find that the corresponding bucket does not contain $e_3$, and the bucket is full, so $e_3$ is inserted into the Cold Filter instead.

*2) Inserting into the Cold Filter:* Note that items enter the Cold Filter only in two cases: (1) when the Burst Filter's bucket is full within the window and cannot accommodate more items, and (2) after the window ends, when items from the Burst Filter are transferred to the Cold Filter.

- **Case 4**: $e_3$ is mapped to two cells in $L_1$. The smaller of the two cells is found, and since its flag is "on," the counter is incremented, and the flag is set to "off."
- **Case 5**: $e_7$ is mapped to two cells in $L_1$. Since both flags are "off," no action is taken.

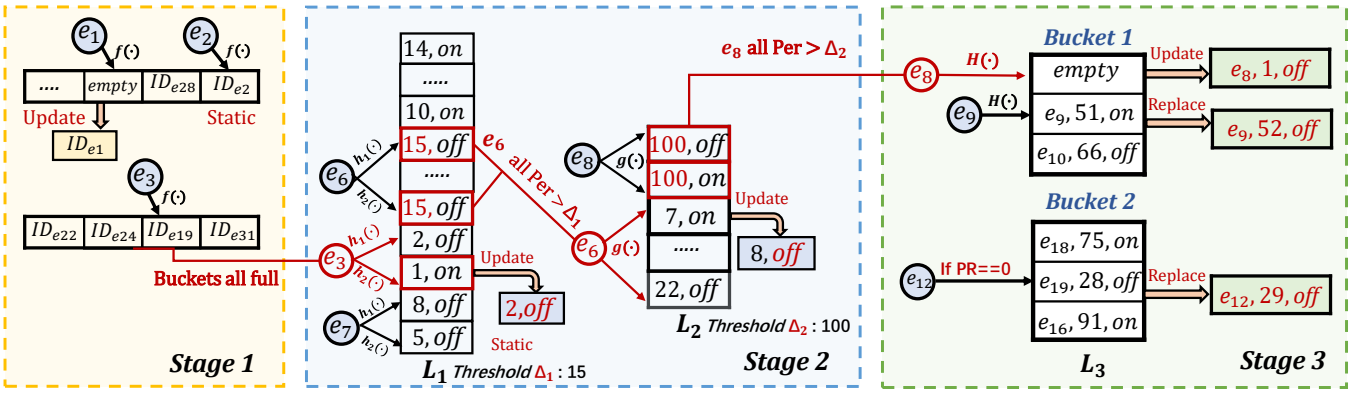



Fig. 10: An example of Hypersistent Sketch.

- **Case 6**: $e_6$ is mapped to two cells in $L_1$. Since the minimum value of these cells reaches the threshold $\Delta_1 = 15$, $e_6$ is moved to $L_2$. Similarly, the two cells in $L_2$ are found, and the smallest is updated.
- **Case 7**: $e_8$ (assuming $\Delta_1$ is reached in $L_1$) is mapped to two cells in $L_2$. Since the threshold $\Delta_2 = 100$ is met, it suggests that $e_8$'s persistence may exceed $\Delta_1 + \Delta_2 = 115$. Thus, $e_8$ is considered a hot item and is inserted into the Hot Part.

*3) Inserting into the Hot Part:* Note that only items with persistence greater than the Cold Filter's combined threshold $\Delta_1 + \Delta_2$ are inserted into the Hot Part.

- **Case 8**: Insert $e_8$. We find that $e_8$ is not present in the bucket, but there is an empty slot, so $e_8$ is inserted into the empty slot.
- **Case 9**: Insert $e_9$. We find that $e_9$ already exists in its corresponding bucket, so we update its counter and flag.
- **Case 10**: Insert $e_{12}$. We find that $e_{12}$ is not in its bucket, and the bucket has no available slots. We locate the item with the smallest persistence in the bucket, $e_{19}$, and replace it with probability $\frac{1}{28+1}$. If the replacement is successful, we insert $e_{12}$ and continue updating its counter and flag. If the replacement fails, we return.

### H. SIMD Optimization

In this section, we optimize the Burst Filter (Stage 1) for inserting items by exploiting Single Instruction Multiple Data (SIMD) instructions, as most insertions within a window are processed in the Burst Filter. SIMD instructions enable data parallelism through vectorization, which effectively accelerates sequential access operations. Our key optimization targets the item lookup process in the Burst Filter. Traditionally, inserting an item $e_i$ requires checking whether it exists in bucket $B[f(e_i)]$ by sequentially comparing it with each stored entry, which becomes inefficient when multiple items are present. SIMD-based parallel comparisons address this inefficiency. By utilizing these 128-bit SIMD operations, we reduce the number of comparison operations required, accelerating the bucket scanning process by approximately four times compared to traditional sequential scanning. We also demonstrate how SIMD instructions enhance the performance of the Burst Filter. As described in Section III-D, to insert an item $e_i$, the Burst Filter first checks whether it is stored in $B[f(e_i)]$. Specifically, this involves sequentially comparing the ID of $e_i$ with the IDs stored in $B[f(e_i)]$. By leveraging SIMD instructions, the Burst Filter can match item IDs in parallel. For example, assume the ID length is 4 bytes, and each bucket stores 4 items. The pseudo-code for the insertion process, implemented using AVX2 SIMD instructions, is presented in Algorithm 6.

**Algorithm 6:** Scanning a bucket with SIMD

**Input:** Item $e$. Pointer $p$ is the start of bucket entries
**Output:** Index of the matched entry or -1 (no found)

```
__m128i item = _mm_set1_epi32(e);
__m128i * keys = (__m128i *) p;
        ▷ Step 1: Duplicate item e into a 128-bit register
__m128i a_comp = _mm_cmpeq_epi32(item, keys[0]);
__m128i b_comp = _mm_cmpeq_epi32(item, keys[1]);
__m128i c_comp = _mm_cmpeq_epi32(item, keys[2]);
__m128i d_comp = _mm_cmpeq_epi32(item, keys[3]);
    ▷ Step 2: Compare item with four bucket entries in parallel
a_comp = _mm_packs_epi32(a_comp, b_comp);
c_comp = _mm_packs_epi32(c_comp, d_comp);
a_comp = _mm_packs_epi32(a_comp, c_comp);
                ▷ Step 3: Pack comparison results together
matched = _mm_movemask_epi8(a_comp);
  ▷ Step 4: Convert SIMD comparison results to scalar mask
return matched ≠ 0 ? TZCNT(matched) : -1;
          ▷ Return index of first match or -1 if no match
```

## IV. Mathematical Analysis

Due to space limitations, we place the detailed proofs of the following theorems on the Website [1] and the GitHub [2].

### A. Burst Filter Speedup

**Theorem IV.1.** *Assume that in multiple time windows $T$, there are $n_{BF}$ distinct items, and the total number of items is $E_{BF}$. The Burst Filter contains $w$ buckets, with each bucket having*

*$\gamma$ cells. Let $P_{Bur}$ denote the probability of the Burst Filter capturing the data stream. We have $P_{Bur} \to 1$.*

**We conclude that in practice, there is a substantial volume of data in data streams, allowing the Burst Filter to capture nearly all data effectively.**

### *B. Persistence Estimation*

*1) Error Bound:*

**Theorem IV.2.** *Let $\hat{p}_i$ be the estimated persistence of our method. We have*

$$p_i \leq \hat{p}_i \leq T. \tag{1}$$

**In summary, we have $p_i \leq \hat{p}_i \leq T$. This result indicates that the estimated persistence provided is within a reasonable range and will not introduce significant deviation when utilized.**

**Theorem IV.3.** *Let $n$, $m$ and $L$ be the number of buckets in layer $L_1$, $L_2$ and $L_3$, $l = \frac{e}{\epsilon}$ and $d = \ln\left(\frac{1}{\delta}\right)$. For small data streams, we obtain*

$$\mathbb{P}\left(\hat{p}_i \leqslant p_i + \epsilon \|p\|_1\right) \geqslant 1 - \delta. \tag{2}$$

*For medium data streams, where $\varepsilon = \frac{e}{n \times m}$ and $\delta = e^{-d_1 - d_2}$, it follows that*

$$\mathbb{P}(\hat{p}_i \leq p_i + \varepsilon \left\|p\right\|_1 \times \left\|p\right\|_1^1) \geq 1 - \delta. \tag{3}$$

*For large data streams, $\varepsilon_1 = \frac{e}{n \times m \times L}$ and $\delta = e^{-d_1 - d_2 - d_3}$, we conclude that*

$$\mathbb{P}(\hat{p}_i \leq p_i + \varepsilon_1 \left\|p\right\|_1 \times \left\|p\right\|_1^1 \times \left\|p\right\|_1^2) \geq 1 - \delta. \tag{4}$$

**It is evident that we can obtain the estimated persistence in a finite number of iterations and calculations. The time complexity of our estimation method is $O(\ln(\frac{1}{\delta}))$, while the space complexity is $O\left(\frac{1}{\varepsilon}\ln(\frac{1}{\delta})\right)$.**

*2) Comparison with Related Work:*

**Theorem IV.4.** *Let $\hat{p}_i^{OO}$ be the estimated persistence of the On-Off Sketch. Under the same memory conditions, we filter the entire data stream by assigning a different number of counters and $d_i$ for small, medium, and large data streams. For simplicity, we use the same names for the hash functions in the On-Off Sketch as in our method, even though they have different numbers of counters. Let $\Delta_j^{OO} p_i = C_j[h_j(e_i)] - p_i$, where $p_i = \min_{1 \leq j \leq d}\left(C_j[h_j(e_i)]\right)$, and $d$ represents the number of hash functions in the On-Off Sketch. There is:*

$$E\left(\Delta_j^{OO} p_i\right) > E\left(\Delta_j p_i\right).$$

**The above inequalities indicate that our method is superior to the On-Off Sketch.**

### *C. Finding Persistent Items*

**Theorem IV.5.** *Let $\hat{p}_i = B[h_1(e_i)][e_i]$ and denote the On-Off Sketch estimate as $\hat{p}_i^{OO}$, where $B[i]$ is the $i$th bucket. We have the following inequality:*

$$p_i \leq \hat{p}_i \leq \hat{p}_i^{OO} \leq T. \tag{5}$$

**Therefore, our method outperforms the On-Off Sketch in finding persistent items.**

### *D. Skewness-Aware and Threshold Sensitivity Analysis*

**Theorem IV.6: Skewness-Aware Error Bound.** Assume the persistence follows a Zipf distribution with parameter $s$, i.e., the persistence of the $i$-th most frequent item is:

$$p_i = \frac{1}{i^s H_N^{(s)}}, \quad \text{where } H_N^{(s)} = \sum_{k=1}^{N} \frac{1}{k^s}\ .$$

The expected error upper bound of the Hypersistent Sketch satisfies:

$$\mathbb{E}[\hat{p}_i - p_i] \leq \frac{H_N^{(s)}}{n} + \frac{H_N^{(s-1)}}{m},$$

where $n$ and $m$ are the number of counters in $L_1$ and $L_2$ layers of the Cold Filter, respectively.

This indicates that our method is adaptable to data with different skews. **When the skewness is large, our method performs better; when the skewness is not large, the data distribution is relatively uniform, and our method is not inferior to the comparison method.**

**Theorem IV.7: Threshold Sensitivity and Pareto Optimality.** Let the Cold Filter thresholds be parameterized as:

$$\Delta_1 = k_1 \cdot \frac{\log n}{\log\log n}, \quad \Delta_2 = k_2 \cdot \Delta_1 = k_1 k_2 \cdot \frac{\log n}{\log\log n}.$$

where $k_1, k_2$ are tunable constants. The memory-error trade-off satisfies:

$$\text{Memory Efficiency} \propto \frac{1}{k_1 k_2}, \ \text{Relative Error} \propto \frac{\sqrt{k_1}}{n^{1/2}} + \frac{\sqrt[3]{k_2}}{m^{1/3}}.$$

Pareto optimality is achieved when:

$$k_1 = \Theta\left(\sqrt{\frac{n}{\log n}}\right), \quad k_2 = \Theta\left(\sqrt[3]{\frac{m}{\log m}}\right).$$

This ratio automatically adapts to data scale while maintaining near-optimal performance. The theorem shows that **our method behaves differently for different thresholds and gives the memory allocation and the relationship between error and threshold parameters. In addition, the Pareto optimality of the theoretical threshold is given at the end, which can guide us in setting the corresponding threshold parameters in the experiment.**

### *E. Time and Space Complexity Analysis*

The overall insertion and query time complexity of Hypersistent Sketch is determined by the operations performed in each stage. **Insertion complexities of Burst Filter, Cold Filter (L1, L2) and Hot Part are $O(1)$, $O(d_1)$ and $O(1)$ respectively. Query complexities of Burst Filter, Cold Filter (L1, L2) and Hot Part are $O(\gamma)$, $O(d_1 + d_2)$ and $O(1)$ respectively.**

The space complexity of Hypersistent Sketch is determined by memory allocation across the three stages. **Space complexities of Burst Filter, Cold Filter (L1, L2) and Hot Part are $O(w\gamma)$, $O(d_1 w + d_2 w)$ and $O(\lambda\beta)$ respectively.**

Overall, the time complexity of the Burst Filter, Cold Filter, and Hot Filter in our method is $O(d_1 + d_2) + O(\gamma)$. Hypersistent Sketch divides the available memory into three

parts ($M_1$ for Burst Filter, $M_2$ for Cold Filter, $M_3$ for Hot Filter), so the total space complexity is:

$O(M_1+M_2+M_3)$ **(which can be combined into** $O(M)$**).**

**In stream processing applications,** $M$ **is usually a fixed amount set in advance based on accuracy requirements or hardware constraints, and it does not grow linearly with the data stream size** $N$**. Therefore, it is often represented as** $O(M)$**.**

**Theorem IV.8 Comparison of computational efficiency** Consider a stream of data items, with the total number of items denoted as $M$. When only the Cold Filter and Hot Part are available, data is processed by the hash function $2m$ times upon reaching the Cold Filter. Next, data with persistence exceeding the threshold $D_1+D_2$ is reprocessed in the Hot Part. With the introduction of the Burst Filter, data undergoes $m$ hash function calculations through the Burst Filter. Subsequently, data whose persistence exceeds $D_1$ is processed twice into the Cold Filter. If the persistence further exceeds $D_1+D_2$, one hash is performed for the Hot Part. We will compare the two methods sequentially under different data distributions.

Under various data distributions, such as uniform distribution, exponential distribution and Zipf distribution, incorporating a Burst Filter significantly increases computing efficiency by $2\times$.

## V. Experimental Results

This section presents experimental results, including experimental setup (V-A), performance on persistence estimation (V-B), and finding persistent items (V-C). Finally, we explore throughput enhancement using SIMD instructions (V-D).

### A. Experimental Setup

*1) Implementation:*

Experiments were conducted on a six-core Intel i5-8400 (2.80GHz, 16GB DRAM) with 32KB L1, 256KB L2, and 9MB shared L3 cache, ensuring sufficient computational resources.

*2) Metrics:*

**Insert Throughput:** $N/T$ million operations per second (Mops), where $N$ is the total number of insertions and $T$ is the total time taken. Median values are reported over five experimental runs.

**Query Throughput:** $Q/T$ million queries per second (Mqps), where $Q$ is the total number of queries and $T$ is the total time taken. Median values are reported over five experimental runs.

**AAE (Average Absolute Error):** $\frac{1}{|\Phi|}\sum_{e_i\in\Phi}|P(e_i)-\hat{P}(e_i)|$, where $P(e_i)$ is the real persistence of item $e_i$, $\hat{P}(e_i)$ is the estimated persistence, and $\Phi$ is the query set.

**ARE (Average Relative Error):** $\frac{1}{|\Phi|}\sum_{e_i\in\Phi}\frac{|P(e_i)-\hat{P}(e_i)|}{P(e_i)}$, evaluating the error rate of estimated persistence.

**F1-Score:** The F1-Score is a balanced metric for detecting persistent items, combining precision and recall to minimize both false positives and false negatives. F1-Score $=\frac{2\cdot\text{TP}}{2\cdot\text{TP}+\text{FP}+\text{FN}}$, where TP (True Positives) are correctly identified persistent items, FP (False Positives) are non-persistent items misclassified as persistent, and FN (False Negatives) are persistent items missed by the model.

**FNR (False Negative Rate):** The FNR quantifies the proportion of persistent items that are not identified: FNR $=\frac{\text{FN}}{\text{FN}+\text{TP}}$, where FN (False Negatives) refers to persistent items that are not reported, and TP (True Positives) represents persistent items correctly identified.

**FPR (False Positive Rate):** The FPR measures the proportion of non-persistent items that are incorrectly identified as persistent: FPR $=\frac{\text{FP}}{\text{FP}+\text{TN}}$, where FP (False Positives) refers to non-persistent items incorrectly identified as persistent, and TN (True Negatives) represents non-persistent items correctly identified.

*3) Dataset:*

**CAIDA Dataset:**(1) The CAIDA trace from the Equinix-Chicago monitor includes 109,534 items (13 bytes per packet). The primary trace lasts for 5 seconds (162K items, 2.49M packets, max item: 17K), while a 2-minute trace is used for large-scale evaluation (1.71M items, 53.72M packets, max item: 0.93M). Most items have a persistence below 50, and each item ID is 4 bytes. (2) **Big CAIDA Dataset:** This dataset consists of anonymous IP communication data collected by CAIDA, covering DDoS attack traffic, TCP/UDP probe traffic, and BGP monitoring traffic. These traffic types show varying traffic size distributions, with TCP traffic accounting for 44% to 65%. The dataset contains 30 million items and 543,996 distinct items, ensuring that the throughput evaluation reflects real-world use cases.

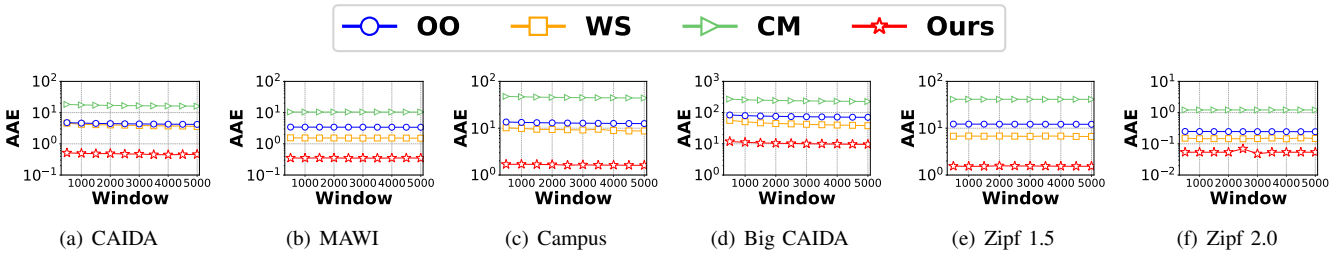
**MAWI Dataset:** It is collected by the MAWI Working Group [51], includes a 15-minute trace from January 1, 2022, containing 8.35M items. It comprises 2M flows with 200,471 distinct types (13 bytes per packet), with 266 hits and 125 flows exceeding the persistence threshold; most flows have persistence below 50.

**Campus Dataset:** It consists of 10 IP tracking traces from campus gateways, was used. A 1 million item subset includes 10M flows with 259,948 distinct types (13 bytes per packet), with 1,333 hits and 677 flows exceeding the persistence threshold. Most flows exhibit persistence below 50.

**Synthetic datasets:** Generated using the Web Polygraph tool, these datasets follow Zipf distributions with skewness values ranging from 1.5 to 2.5. Comprising approximately 9.8 million network packets, the number of distinct packet types varies with skewness: around 307,795 types for Zipf 1.5, 29,412 types for Zipf 2.0, and 6,552 types for Zipf 2.5. They enable performance evaluation across diverse distribution scenarios.

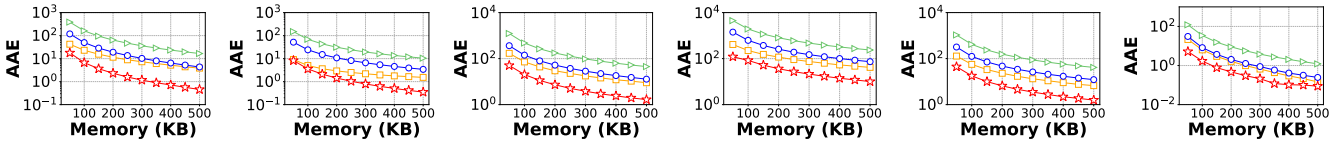
*4) Methodology:*

Hypersistent Sketch (HS) was implemented in C++ using BOBHash, with distinct random seeds per hash function, and regenerated each time window for probabilistic randomness. For persistence estimation, HS was compared with On-Off Sketch (OO) [52], WavingSketch (WS) [53], CM Sketch (CM) [48], focusing on AAE, ARE. Tight Sketch (TS) [54] and P-Sketch (PS) [55], while recent advancements in the field.



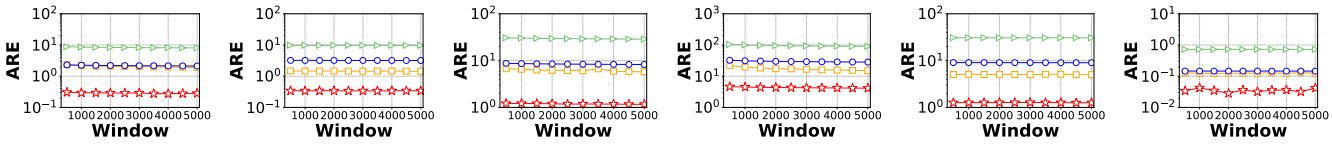

(a) CAIDA (b) MAWI (c) Campus (d) Big CAIDA (e) Zipf 1.5 (f) Zipf 2.0

Fig. 11: AAE on persistence estimation vs. window.



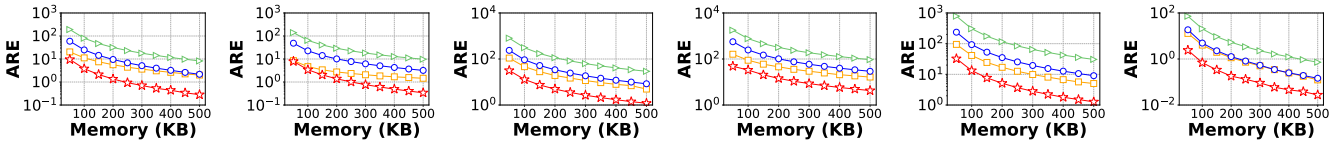

(a) CAIDA (b) MAWI (c) Campus (d) Big CAIDA (e) Zipf 1.5 (f) Zipf 2.0

Fig. 12: AAE on persistence estimation vs. memory.




(a) CAIDA (b) MAWI (c) Campus (d) Big CAIDA (e) Zipf 1.5 (f) Zipf 2.0

Fig. 13: ARE on persistence estimation vs. window.




(a) CAIDA (b) MAWI (c) Campus (d) Big CAIDA (e) Zipf 1.5 (f) Zipf 2.0

Fig. 14: ARE on persistence estimation vs. memory.

However, they were not included in the comparison because they do not maintain persistence records in the sketch, which makes it impossible for them to estimate the persistence of entire items. Evaluations were conducted under varying memory sizes (50-500 KB) with a persistence window of 3000, and with a fixed memory size of 500 KB while varying window sizes from 500 to 5000. For CM and WS, half of the memory was allocated to the Bloom Filter [49] to reduce false positives. OO used a three-layer structure as per the original implementation. In HS, 30% of memory was assigned to the Hot Part, the cold part had a 17:3 memory ratio, and the Burst Filter size was set to window size/100 KB.

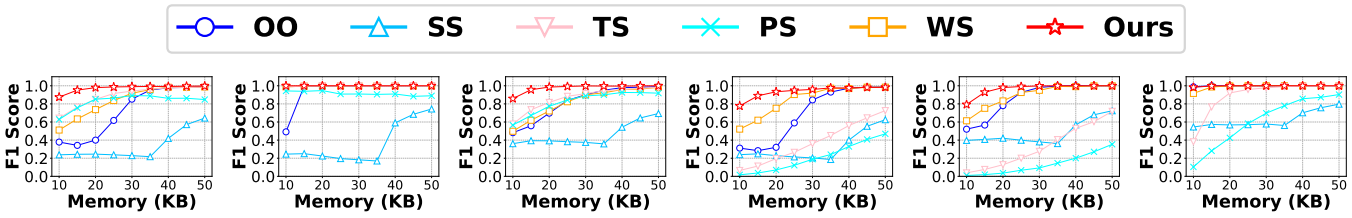
For persistent item detection, HS was compared with Small-Space (SS) [56], WavingSketch (WS) [53], On-Off Sketch (OO) [52], and state-of-the-art works based on On-Off Sketch, including Tight Sketch (TS) [54] and P-Sketch (PS) [55], using F1-Score, ARE, FNR, and FPR. Evaluations were performed with a persistence window of 1500 and memory ranging from 10 KB to 50 KB. CM and WS allocated half of the memory to the Bloom Filter to ensure a low FPR, while OO, PS and TS used a three-layer structure. In HS, 40% of memory was used for the Hot Part, the Burst Filter was set to 1 KB, and the cold part maintained a 17:3 memory ratio.

We use their open-source codes to compare the algorithms. The source code of Hypersistent Sketch can be downloaded from the Website [1], as well as the GitHub [2].

### *B. Persistence Estimation*

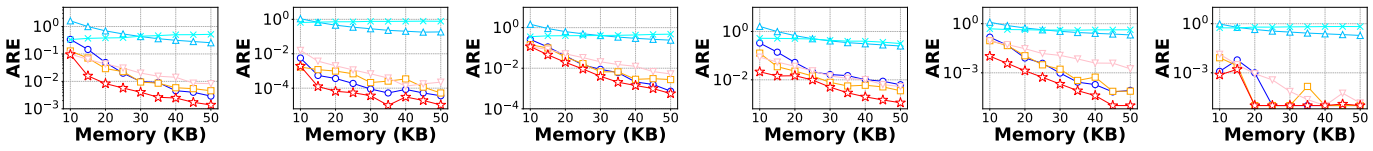
**AAE:** Figures 11, 12 illustrate the changes in AAE with varying window sizes and memory capacities. From Figure 11, we observe that the AAE of all algorithms remains relatively unaffected by window size, demonstrating a unique property of persistence estimation. Among the four algorithms, our Hypersistent Sketch (HS) consistently achieves the lowest AAE across all datasets, window sizes memory capacities and Zipf distributions. On average, HS outperforms WS by 0.7 orders of magnitude, OO by 1 order of magnitude, and CM by 1.5 orders of magnitude. In Figure 12, the AAE of all algorithms decreases as memory size increases. HS maintains its lead, achieving an average improvement of 0.8, 1, and 1.5 orders of magnitude over WS, OO, and CM, respectively, at a memory size of 500KB. Particularly in Zipf 1.5 and Zipf 2.0 distributions, AAE is reduced by 1.5 to 2 orders of magnitude compared to OO, WS, CM.

**ARE:** Figures 14 and 13 present the changes in ARE with varying window sizes and memory capacities. HS achieves



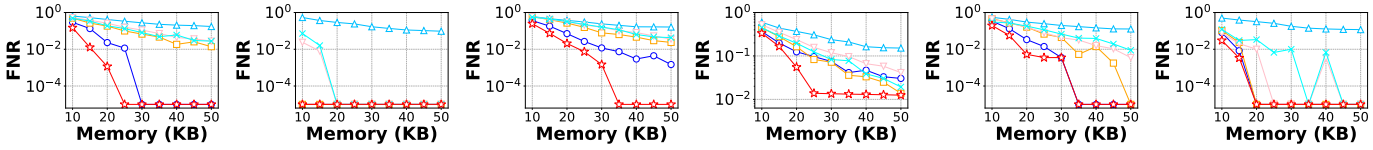

(a) CAIDA (b) MAWI (c) Campus (d) Big CAIDA (e) Zipf 1.5 (f) Zipf 2.0

Fig. 15: F1-score on finding persistent items.



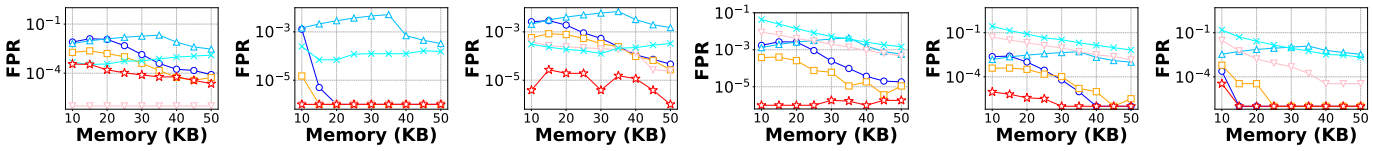

(a) CAIDA (b) MAWI (c) Campus (d) Big CAIDA (e) Zipf 1.5 (f) Zipf 2.0

Fig. 16: ARE on finding persistent items.




(a) CAIDA (b) MAWI (c) Campus (d) Big CAIDA (e) Zipf 1.5 (f) Zipf 2.0

Fig. 17: FNR on finding persistent items.




(a) CAIDA (b) MAWI (c) Campus (d) Big CAIDA (e) Zipf 1.5 (f) Zipf 2.0

Fig. 18: FPR on finding persistent items.

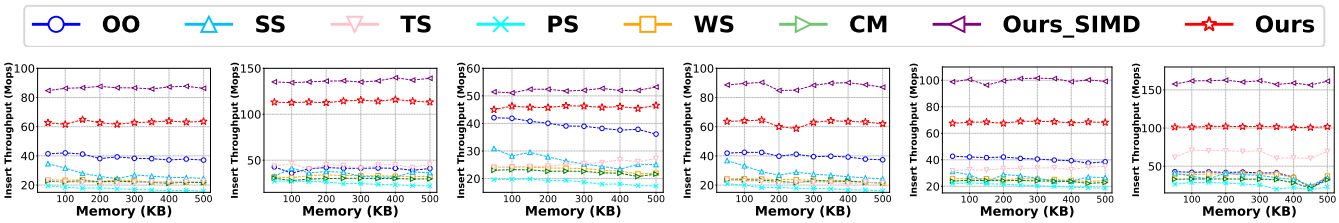
the lowest ARE on the Big CAIDA dataset, consistent with the trends observed for AAE. and consistently outperforms OO, WS, and CM across all Zipf distributions. In Zipf 1.5 and Zipf 2.0, HS achieves ARE reductions by 1.5 to 2 orders of magnitude compared to all other algorithms. These error analysis experiments demonstrate the effectiveness of Hypersistent Sketch in improving accuracy by introducing the Cold Filter, which efficiently separates cold and hot items, thereby enhancing persistence estimation accuracy.

### C. Finding Persistent Items

**F1-score:** Figure 15 presents the F1-score performance in finding persistent items under varying memory capacities. As memory increases from 10 KB to 50 KB, HS consistently achieves an F1-score close to 1.0 across all datasets. With only 10 KB of memory, HS's F1-score surpasses those of WS by 1.9×, TS by 1.5×, PS by 1.5×, OO by 2.1×, and SS by 3.3× on average. In both Zipf 1.5 and Zipf 2.0 distributions, HS consistently achieves the highest F1-score, approaching 1.0 as memory increases from 10 KB to 50 KB. At 10KB, HS's F1-score surpasses that of TS by 1.3×, PS by 1.2×, OO by 1.5×, and SS by 2.0×. This improvement is primarily due to HS's Cold Filter, which effectively handles a large number of cold items, while TS and PS lag behind HS in efficiently handling cold items, thereby resulting in lower performance in finding persistent items.

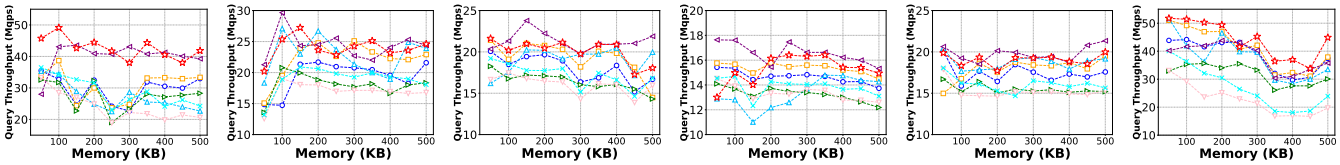
**ARE:** Figure 16 illustrates the changes in ARE with varying memory capacities in finding persistent items. The figure shows that while the ARE of all algorithms decreases as memory increases, HS consistently achieves the lowest ARE. In the 10 KB to 50 KB memory range, HS outperforms WS by 0.66, OO by 0.82, SS by 2.83, TS by 1.5, and PS by 1.8, orders of magnitude, respectively. Particularly under Zipf 1.5 and Zipf 2.0, at 10 KB to 15 KB memory range, HS shows ARE reduced from around $10^{-2}$ to nearly $10^{-3}$, significantly outperforming PS, TS, and WS. HS achieves ARE reductions by 1.5 to 2 orders of magnitude compared to TS, PS, OO, WS, and SS.

**FNR:** Figure 17 presents the FNR in finding persistent items under varying memory capacities. As memory capacity increases, Figure 17 shows that HS consistently achieves the lowest FNR. Specifically, when using 20 KB of memory, HS reduces the FNR to $10^{-5}$, while WS and SS remain at $10^{-1}$




(a) CAIDA (b) MAWI (c) Campus (d) Big CAIDA (e) Zipf 1.5 (f) Zipf 2.0

Fig. 19: Insert Throughput of Hypersistent Sketch with/without SIMD.




(a) CAIDA (b) MAWI (c) Campus (d) Big CAIDA (e) Zipf 1.5 (f) Zipf 2.0

Fig. 20: Query Throughput of Hypersistent Sketch with/without SIMD.

and $10^0$, respectively. Even with only 10 KB of memory, as shown in Figure 17, HS maintains the FNR at $10^{-5}$, whereas SS's,TS's, and PS's FNR all exceeds $10^{-1}$. Across both Zipf 1.5 and Zipf 2.0 distributions, at 20 KB to 25 KB memory, HS improves FNR by 1.8 to 2 orders of magnitude compared to PS and TS, reducing it from around $10^{-2}$ to approximately $10^{-3}$. OO shows higher FNR than HS, exceeding $10^{-2}$.

**FPR:** Figure 18 shows the FPR in finding persistent items under varying memory capacities. Figure 18 shows that as memory capacity increases, the FPR of all algorithms decreases, following a trend similar to the FNR, with HS consistently achieving the lowest FPR. With 10 KB of memory, HS achieves average improvements of 1.67, 2.61, 2.61, 3.11 and 3.93 orders of magnitude over WS, OO, SS, TS, and PS respectively. In both Zipf 1.5 and Zipf 2.0 distributions, HS's FPR follows a similar trend, reducing it from approximately $10^{-2}$ to $10^{-4}$ across the 10 KB to 25 KB memory range.

### *D. SIMD Acceleration*

SIMD [57] acceleration significantly enhances the performance of HS. Without SIMD, inserting an item into a bucket with 16 entries requires up to 16 comparisons. With SIMD, entries are grouped into four sets of four, and comparisons are performed in parallel using a 128-bit vector, reducing comparisons to just four per insertion.

**Insert Throughput**: Figure 19 shows that without SIMD, as memory capacity increases, the throughput of all algorithms improves, with HS maintaining the highest performance. This performance is mainly attributed to the Burst Filter preventing repeated insertions of the same item within a single window by deferring insertions until the window ends, further optimizing processing speed. With SIMD acceleration, the insert throughput of HS increases significantly. For the CAIDA dataset, throughput increases by 50% (from 60 Mops to 90 Mops). In the MAWI dataset, it improves by 27% (from 110 Mops to 140 Mops), and in the Campus dataset, by 22% (from 45 Mops to 55 Mops), and for the Big CAIDA dataset, throughput increases by 30% (from 60 Mops to 80 Mops). In Zipf 1.5 and 2.0 distributions, HS leads WS, OO, CM, and PS by approximately 1 order of magnitude, and TS by 2 orders of magnitude across all memory sizes. SIMD further improves HS by 0.2 orders of magnitude. In Zipf 1.5, this results in a 25% performance increase (from 70 Mops to 100 Mops), while in Zipf 2.0, it leads to a 20% increase (from 100 Mops to 150 Mops).

**Query Throughput**: Figure 20 presents that when the data exhibits a skewed distribution, figures 20(e) and 20(f) illustrate that the majority of items are processed at the L1 stage in the Cold Filter (i.e., Stage 2), with only a small fraction entering the L2 stage, and an even smaller proportion reaching Hot Part (i.e., Stage 3). Although SIMD acceleration significantly improves insert throughput, its impact on query throughput is relatively limited. This is due to the inherently low parallelism of query operations and fewer items queried, which limits the effectiveness of SIMD acceleration in queries. These results demonstrate the effectiveness of SIMD in boosting efficiency, particularly in high-traffic scenarios, by minimizing redundant computations.

## VI. Conclusion

This paper presents Hypersistent Sketch, a novel algorithm that improves persistence estimation in data streams by optimizing memory efficiency, accuracy, and processing speed at the same time. Its three-stage design—Burst Filter, Cold Filter, and Hot Part—efficiently separates cold and hot items, enabling accurate persistence estimates with optimized memory use. The Cold Filter removes low-persistence items to enhance memory efficiency, while the Burst Filter reduces redundant updates, significantly boosting throughput. Experimental results confirm that Hypersistent Sketch surpasses the well-known On-Off Sketch in both accuracy and speed, making it an effective solution for persistence estimation tasks.

## REFERENCES


[1] Our Website, https://wenjunli.com/HypersistentSketch.

[2] Our GitHub, https://github.com/wenjunpaper/HypersistentSketch.

[3] T. Yang, H. Zhang, D. Yang, Y. Huang, and X. Li, "Finding significant items in data streams," in *IEEE ICDE*, 2019.

[4] S. Guha, J. Chandrashekar, and N. Taft, "How healthy are today's enterprise networks?" in *ACM SIGCOMM*, 2008.

[5] M. Karppa and R. Pagh, "Hyperlogloglog: Cardinality estimation with one log more," in *ACM SIGKDD*, 2022.

[6] A. Alshamrani, S. Myneni, A. Chowdhary, and D. Huang, "A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1851–1877, 2019.

[7] L. Shang, D. Guo, Y. Ji, and Q. Li, "Discovering unknown advanced persistent threat using shared features mined by neural networks," *Computer Networks*, vol. 189, p. 107937, 2021.

[8] I. Ghafir *et al.*, "Advanced persistent threat attack detection: an overview," *Int J Adv Comput Netw Secur*, vol. 4, no. 4, p. 5054, 2014.

[9] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *ACM SIGMOD*, 2016.

[10] M. Li, H. Dai *et al.*, "Seesaw counting filter: A dynamic filtering framework for vulnerable negative keys," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 12 987–13 001, 2023.

[11] A. Sharma, B. B. Gupta, A. K. Singh, and V. Saraswat, "Advanced persistent threats (apt): evolution, anatomy, attribution and countermeasures," *Journal of Ambient Intelligence and Humanized Computing*, vol. 14, no. 7, pp. 9355–9381, 2023.

[12] Z. Fan, Z. Hu, Y. Wu, J. Guo, W. Liu, and T. Yang, "Pisketch: finding persistent and infrequent flows," in *ACM SIGCOMM*, 2022.

[13] L. Chen, H. Dai, L. Meng, and J. Yu, "Finding needles in a hay stream: On persistent item lookup in data streams," *Computer Networks*, vol. 181, p. 107518, 2020.

[14] H. Dai, M. Li, A. X. Liu, J. Zheng, and G. Chen, "Finding persistent items in distributed datasets," *IEEE/ACM Transactions on Networking*, vol. 28, no. 1, pp. 1–14, 2019.

[15] H. Dai, M. Shahzad, A. X. Liu, M. Li, Y. Zhong, and G. Chen, "Identifying and estimating persistent items in data streams," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2429–2442, 2018.

[16] H. Huang, Y.-E. Sun, C. Ma, S. Chen, Y. Zhou, W. Yang, S. Tang, H. Xu, and Y. Qiao, "An efficient k-persistent spread estimator for traffic measurement in high-speed networks," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1463–1476, 2020.

[17] Y. Sun, H. Huang, S. Chen, Y. Zhou *et al.*, "Privacy-preserving estimation of $k$-persistent traffic in vehicular cyber-physical systems," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8296–8309, 2019.

[18] H. Huang, Y.-E. Sun, S. Chen, S. Tang, K. Han, J. Yuan, and W. Yang, "You can drop but you can't hide: $k$-persistent spread estimation in high-speed networks," in *IEEE INFOCOM*, 2018.

[19] I. A. Elgendy, W. Zhang, Y.-C. Tian, and K. Li, "Resource allocation and computation offloading with data security for mobile edge computing," *Future Generation Computer Systems*, vol. 100, pp. 531–541, 2019.

[20] W. Zhang, I. A. Elgendy, M. Hammad, A. M. Iliyasu, X. Du, M. Guizani, and A. A. Abd El-Latif, "Secure and optimized load balancing for multitier iot and edge-cloud computing systems," *IEEE Internet of Things Journal*, vol. 8, no. 10, pp. 8119–8132, 2020.

[21] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," *ACM SIGMOD*, vol. 30, no. 2, pp. 58–66, 2001.

[22] D. M. Powers, "Applications and explanations of zipf's law," in *EMNLP/CoNLL*, 1998.

[23] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *ACM SIGCOMM*, 2010.

[24] H. Yang *et al.*, "Fedsteg: A federated transfer learning framework for secure image steganalysis," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1084–1094, 2020.

[25] Z. Fan and R. Wang, "Onesketch: A generic and accurate sketch for data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 12 887–12 901, 2023.

[26] H. Li, Q. Chen *et al.*, "Stingy sketch: a sketch framework for accurate and fast frequency estimation," in *ACM VLDB*, 2022.

[27] H. Huang, J. Yu *et al.*, "Memory-efficient and flexible detection of heavy hitters in high-speed networks," in *ACM SIGMOD*, 2024.

[28] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *ACM IMC*, 2003.

[29] L. Wang, G. Luo, K. Yi, and G. Cormode, "Quantiles over data streams: An experimental study," in *ACM SIGMOD*, 2013.

[30] R. Yadav, W. Zhang *et al.*, "Energy-latency tradeoff for dynamic computation offloading in vehicular fog computing," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 12, pp. 14 198–14 211, 2020.

[31] Q. Zhou, Y.-E. Sun *et al.*, "Cba sketch: A sketching algorithm mining persistent batches in data streams," in *International Conference on Algorithms and Architectures for Parallel Processing*, 2023.

[32] T. Yang, S. Gao, Z. Sun *et al.*, "Diamond sketch: Accurate per-flow measurement for big streaming data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 12, pp. 2650–2662, 2019.

[33] C. Guo, L. Yuan, D. Xiang *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *ACM SIGMCOMM*, 2015.

[34] Y. Zhu, N. Kang, J. Cao *et al.*, "Packet-level telemetry in large datacenter networks," in *ACM SIGMCOMM*, 2015.

[35] Y. Zhang, J. Li, and Lei, "On-off sketch: a fast and accurate sketch on persistence," *Proceedings of the VLDB Endowment*, vol. 13, no. 11, pp. 2372–2385, 2020.

[36] H. Dai, M. Shahzad, and A. X. Liu, "Finding persistent items in data streams," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 289–300, 2016.

[37] Y. Zhou, T. Yang *et al.*, "Cold filter: A meta-framework for faster and more accurate stream processing," in *ACM SIGMOD*, 2018.

[38] J. Tan and J. Wang, "Detecting advanced persistent threats based on entropy and support vector machine," in *IEEE ICA3PP*, 2018.

[39] Q. Xiao, Y. Qiao, M. Zhen, and S. Chen, "Estimating the persistent spreads in high-speed networks," in *IEEE ICNP*, 2014.

[40] C. Tankard, "Advanced persistent threats and how to monitor and deter them," *Network security*, vol. 2011, no. 8, pp. 16–19, 2011.

[41] S. A. Singh and S. Tirthapura, "Monitoring persistent items in the union of distributed streams," *Journal of Parallel and Distributed Computing*, vol. 74, no. 11, pp. 3115–3127, 2014.

[42] Y. Peng, J. Guo, F. Li, W. Qian, and A. Zhou, "Persistent bloom filter: Membership testing for the entire history," in *ACM SIGMOD*, 2018.

[43] Y. Sun, H. Huang, and S. Chen, "Persistent traffic measurement through vehicle-to-infrastructure communications in cyber-physical road systems," *IEEE Transactions on Mobile Computing*, vol. 18, no. 7, pp. 1616–1630, 2018.

[44] Q. Shi *et al.*, "Cuckoo counter: Adaptive structure of counters for accurate frequency and top-k estimation," *IEEE/ACM Transactions on Networking*, vol. 31, no. 4, pp. 1854–1869, 2023.

[45] Q. Shi, C. Jia, W. Li, Z. Liu, T. Yang, G. Xie *et al.*, "Bitmatcher: Bit-level counter adjustment for sketches," in *IEEE ICDE*, 2024.

[46] L. Cao, Q. Shi, Y. Liu, H. Zheng, Y. Xin, W. Li, T. Yang *et al.*, "Bubble sketch: A high-performance and memory-efficient sketch for finding top-k items in data streams," in *ACM CIKM*, 2024.

[47] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *ACM SIGCOMM*, 2018.

[48] D. Ting, "Count-min: Optimal estimation and tight error bounds using empirical error distributions," in *ACM SIGKDD*, 2018.

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

[50] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *ACM SIGCOMM*, 2002.

[51] MAWI Working Group, "MAWI Working Group Traffic Archive," http://mawi.wide.ad.jp/mawi/, 2024, accessed: 2024-09-30.

[52] Y. Zhang, J. Li, Y. Lei, T. Yang, Z. Li, G. Zhang, and B. Cui, "On-off sketch: A fast and accurate sketch on persistence," *Proceedings of the VLDB Endowment*, vol. 14, no. 2, pp. 128–140, 2020.

[53] J. Li, Z. Li, Y. Xu, S. Jiang, T. Yang, B. Cui, Y. Dai, and G. Zhang, "Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams," in *ACM SIGKDD*, 2020.

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

[55] W. Li *et al.*, "P-sketch: A fast and accurate sketch for persistent item lookup," *IEEE/ACM Transactions on Networking*, vol. 32, no. 2, pp. 987–1002, 2024.

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

[57] I. S. Documentation, https://software.intel.com/enus/node/683883.