# Pandora: An Efficient and Rapid Solution for Persistence-Based Tasks in High-Speed Data Streams

WEIHE LI, University of Edinburgh, United Kingdom

In data streams, persistence characterizes items that appear repeatedly across multiple non-overlapping time windows. Addressing persistence-based tasks, such as detecting highly persistent items and estimating persistence, is crucial for applications like recommendation systems and anomaly detection in high-velocity data streams. However, these tasks are challenging due to stringent requirements for rapid processing and limited memory resources. Existing methods often struggle with accuracy, especially given highly skewed data distributions and tight fastest memory budgets, where hash collisions are severe. In this paper, we introduce Pandora, a novel approximate data structure designed to tackle these challenges efficiently. Our approach incorporates the insight that items absent for extended periods are likely non-persistent, increasing their probability of eviction to accommodate potential persistent items more effectively. We validate this insight empirically and integrate it into our update strategy, providing better protection for persistent items. We formally analyze Pandora's error bounds to validate its theoretical soundness. Through extensive trace-driven tests, we demonstrate that Pandora achieves superior accuracy and processing speed compared to state-of-the-art methods across various persistence-based tasks. Additionally, we further accelerate Pandora's update speed using Single Instruction Multiple Data (SIMD) instructions, enhancing its efficiency in high-speed data stream environments. The code for our method is open-sourced.

CCS Concepts: • **Theory of computation → Sketching and sampling**.

Additional Key Words and Phrases: Data stream processing, approximate data structure, persistence, persistent items

**ACM Reference Format:**
Weihe Li. 2025. Pandora: An Efficient and Rapid Solution for Persistence-Based Tasks in High-Speed Data Streams. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 61 (February 2025), 26 pages. https://doi.org/10.1145/3709711

## 1 Introduction

Detecting items that exhibit specific patterns in high-velocity data streams poses a critical challenge in data mining applications. These streams, sourced from networks, system logs, and user behavior, often harbor valuable insights into system health [31, 55], security threats [2], user experience [52], business intelligence [37], and more. However, traditional deterministic analysis methods face difficulties in efficiently processing these data streams due to their high velocity and volume [6, 49]. Consequently, an emerging paradigm of probabilistic analysis has surfaced, emphasizing the utilization of approximate data structures (sketches) for efficient processing of massive data under constrained memory budgets [11, 17, 33, 48, 57, 60, 64].

Existing sketch-based work primarily focuses on data characteristics such as frequency [42, 49, 51, 58], cardinality [20, 30, 46, 50], and quantiles [15, 22, 24, 29]. Recently, another crucial data characteristic—persistence—has garnered significant attention [16, 18, 62]. Given a data stream

Author's Contact Information: Weihe Li, weihe.li@ed.ac.uk, University of Edinburgh, Edinburgh, Scotland, United Kingdom.


ACM 2836-6573/2025/2-ART61
https://doi.org/10.1145/3709711

divided into $W$ non-overlapping, consecutive time windows, the persistence of an item $e$ is the number of distinct windows in which it appears. Persistence is often crucial for various practical applications, including anomaly traffic detection [21, 54], user behavior analysis [23], and click fraud detection [45]. For instance, persistent user behaviors, like consistently used features or repeatedly visited sections, indicate users' strong preferences and can guide developers in optimizing user interface and experience. Another instance involves automated bots repeatedly clicking on advertisements to boost advertiser revenue in pay-per-click online advertising systems [27].

There are two main detection tasks associated with persistence: (i) *persistence estimation*, which provides an approximate estimate of persistence for each item in the data stream, and (ii) *persistent item lookup*, which identifies items whose persistence exceeds a predetermined threshold. In addition to these tasks, there is a challenge that has been rarely considered: (iii) *mining persistent items with high frequency*. This task has significant practical applications in various domains. For instance, in financial security, detecting accounts or credit cards with persistent and frequent transaction patterns can aid in identifying fraudulent activities or money laundering schemes. In cybersecurity, identifying IP addresses that consistently and frequently send requests can facilitate early detection and mitigation of distributed denial-of-service (DDoS) attacks.

### 1.1 Limitations of Prior Art

While some work exists on persistence, primarily focusing on persistent item lookup, these methods can generally be categorized into three main types: sample-based [13, 32], coding-based [18], and sketch-based [33, 62]. Each method aims to balance memory usage, accuracy, and update speed, yet they encounter significant limitations, particularly under constrained memory conditions. Below, we briefly outline the key conceptual differences and limitations of these approaches:

**(i) Sample-Based Approaches [13, 32]:** These methods rely on sampling to track potential persistent items. Although they offer reduced memory usage by focusing on a subset of items, they often include too many non-persistent items, leading to memory inefficiency. The reliance on sampling also means accuracy suffers when memory is constrained, as the reduced sample size introduces larger estimation errors.

**(ii) Coding-Based Approaches [18]:** These methods encode every item observed in each time window to identify persistent items. They come with high memory overhead, especially when encoding a large number of non-persistent items. As the number of time windows increases, memory usage scales linearly, making these methods impractical for environments with limited memory. Additionally, they require computationally intensive operations, such as matrix multiplication for encoding and decoding, which significantly slow down updates, making them unsuitable for high-speed data streams.

**(iii) Sketch-Based Approaches [33, 62]:** Sketches leverage probabilistic data structures to estimate persistence with lower memory requirements and faster update speed. However, the eviction strategies in most existing sketch-based methods often misclassify non-persistent items as persistent during hash collisions, reducing accuracy, especially in memory-constrained environments.

### 1.2 Challenges

In practice, accurately and swiftly detecting persistence in high-speed data streams presents a complex and demanding task. This complexity arises from two primary factors:

(i) In real-world applications, small memory sizes (e.g., less than 64KB) are crucial in resource-constrained environments like embedded systems, IoT devices, and edge computing [53]. These systems have limited memory and processing power but are responsible for critical tasks such as network monitoring, traffic analysis, and real-time anomaly detection. In such scenarios, where memory efficiency is essential, traditional methods often struggle to maintain detection accuracy.

Using smaller memory portions (e.g., 16KB or 32KB) not only conserves resources but also improves efficiency with faster query time for retrieving persistent items. Achieving high accuracy under stringent resource constraints is particularly challenging, and a method that excels in these conditions demonstrates its robustness and suitability for deployment across diverse platforms with varying resource limitations.

(ii) As an item arrives within a time window, its persistence only increases by 1, regardless of how many times it appears. Consequently, the persistence value is typically much smaller than the item frequency. Due to the highly-skewed data distribution in practical scenarios, where most items have small persistence, the eviction of persistent items from buckets by non-persistent ones is more likely, resulting in low detection accuracy.

### 1.3 Our Solution - Pandora

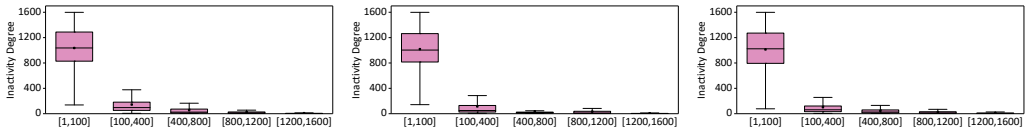
To overcome these challenges, we propose a novel approach, Pandora, for diverse persistence-based detection tasks. Our method achieves high detection accuracy and fast processing speed simultaneously, even with limited memory resources. Technically, our method is based on a key insight: *the longer a tracked item is missing, the less likely it is to be a persistent item.* We refer to the number of time windows a recorded item is missing as its *inactivity degree*. To the best of our knowledge, this is the first study to employ the concept of inactivity degree for various detection tasks in high-speed data streams. During the update processing, when an incoming item arrives, Pandora adjusts the replacement probability dynamically in response to hash collisions, taking into account the inactivity degree. If the inactivity degree of the tracked item is high, the probability of the incoming item successfully replacing the tracked item in the bucket is elevated; conversely, if the inactivity degree is low, the probability is reduced. This approach facilitates easy eviction of items exhibiting low persistence from buckets, making space for potentially persistent items, thereby ensuring high memory efficiency and accurate lookups. The update procedure is straightforward, requiring no additional data structures, which contributes to its high update speed. The elegant design also enables further acceleration through the utilization of SIMD instructions, enhancing its update speed even further.

We rigorously derive the error bound of Pandora to establish its theoretical accuracy. Extensive trace-driven evaluations demonstrate Pandora's significant improvements in detection accuracy across various tasks compared to state-of-the-art methods. For persistence estimation, Pandora reduces the error by up to 96.2% compared to On-Off Sketch. In persistent item lookup, Pandora achieves the highest F1 score among considered baselines, improving accuracy by 57.65% over On-Off Sketch. Moreover, Pandora maintains high performance when identifying persistent items with high frequency, achieving an F1 score of 0.8 under tight 16KB memory constraints. Additionally, our method achieves the highest update speed in persistent item lookup, being 41.65% faster than the competitive On-Off Sketch. Furthermore, our method adapts easily to other frequency-based tasks while maintaining superiority. With the aid of SIMD instructions, we further enhance the update speed by 19.08%. The code for Pandora can be found at [1].

## 2 Problem Definition

(i) Data Stream: Given a high-velocity data stream $\mathcal{S}$ consisting of various items $\mathcal{S} = \{e_1, e_2, \ldots, e_n\}$, where each item $e_i$ can be denoted as a key-value pair $(k_i, v_i)$. The key $k_i$ is the unique identifier of the item, and the value $v_i$ corresponds to a metric associated with the item. For instance, in the network domain, the key could represent a source-destination IP address pair, while the value could represent the item's persistence or frequency.

(ii) Persistence Estimation: For a data stream $\mathcal{S}$, we segment it into $W$ consecutive time windows based on a specified interval. For example, in a network trace, a window can be defined as 2000




(a) CHI15. (b) CHI16. (c) NY18.

Fig. 1. Relationship between item persistence and inactivity degree.

packets, but the division can also be based on timestamps, with window sizes either fixed or flexible. Let $f_i^t$ represent the frequency of item $e_i$ in window $t$. The persistence $v_i$ of an item $e_i$ is defined as $v_i = \sum_{t=1}^{W} \mathbb{I}(f_i^t > 0)$, where $\mathbb{I}(\cdot)$ is the indicator function. This means that for any given item $e_i$, its persistence $v_i$ reflects the number of windows in which it appears, regardless of how many times it occurs within each window. In essence, an item's persistence increases by 1 for every window in which it appears at least once. The persistence estimation problem focuses on accurately determining the persistence value $v_i$ for all unique items in the data stream.

(iii) Persistent Item Lookup: With the persistence values $v_i$ computed for all items $e_i \in \mathcal{S}$, an item $e_i$ is classified as a persistent item if its persistence $v_i \geq \gamma W$, where $\gamma \in (0, 1)$ is a user-defined persistence threshold parameter. The persistent item lookup task involves accurately identifying all items $e_i \in \mathcal{S}$ that satisfy the condition $v_i \geq \gamma W$.

(iv) Persistent Items with High Frequency: An item $e_i$ is classified as persistent with high frequency if it satisfies two conditions: (1) Persistence condition: $v_i \geq \gamma W$; and (2) Frequency condition: $f_i \geq \theta F$, where $f_i = \sum_{t=1}^{W} f_i^t$ is the total frequency of $e_i$ across all windows, $F$ is the sum of frequencies of all items in the stream, and $\theta \in (0, 1)$ is a user-defined frequency threshold.

## 3 Pandora's Design

### 3.1 Key Insights

Persistent items are those that appear in the majority of time windows within a data stream. Conversely, if an item is absent from a substantial number of recent time windows, it indicates a pattern of inactivity, suggesting that this item is more likely to be non-persistent. This observation forms the core of our design philosophy: *items exhibiting prolonged inactivity in recent time windows should have a higher probability of being classified as non-persistent, making them prime candidates for eviction from the sketch data structure.* This strategic eviction policy creates more room for potentially persistent items, thereby optimizing memory utilization and enhancing lookup accuracy.

To validate this attribute, we analyze the arrival patterns of items using the CHI15, CHI16, and NY18 network traces, sourced from the widely recognized CAIDA [7] and MAWI [41] datasets, which are commonly utilized in related research [25, 35, 49, 63]. The specific traces are randomly selected from these datasets, ensuring unbiased selection. Detailed information about these traces is provided in Section 5.1. Following the methodology in [62], we divide the traces into 1,600 time windows. We define the inactivity degree as the number of consecutive time windows an item does not appear. Figure 1 illustrates the relationship between an item's persistence and its maximum inactivity degree. As shown in Figure 1, the inactivity degree of an item decreases as its persistence increases. For example, items with persistence in the range of $[1, 100]$ have a significantly higher inactivity degree compared to items with higher persistence, such as those in the range of $[1200, 1600]$. Specifically, for the NY18 trace, the average maximum inactivity degree for items with persistence between $[1, 100]$ is 1012.53, while for items with persistence between $[1200, 1600]$, the average maximum inactivity degree drops to 12.18. Therefore, the inactivity degree of an item can be an effective metric for distinguishing whether an item is a potential persistent item. If an item does not appear for an extended period (indicating a high inactivity degree), it is less likely to

be a persistent item. Therefore, we can leverage this feature to quickly expel non-persistent items and provide additional priority for persistent items.

In some cases, items appear in waves, with intermittent periods of inactivity, frequently appearing in certain windows but remaining absent for extended stretches in others. However, we emphasize that such cases only account for a small portion of persistent items in practice. Compared to items that appear consistently over time, wave-based items tend to have a higher inactivity degree. For instance, in the NY18 trace, we calculate the inactivity degree for each persistent item and find that out of 747 real persistent items, over 619 have an inactivity degree of less than 20, indicating that most persistent items do not exhibit wave-like behavior. Only 16 items have an inactivity degree greater than 300, constituting just 2.1% of all persistent items, showing that such cases are rare in practice. Similar trends are observed across other traces and different window settings (e.g., 1200 and 1000), demonstrating that the inactivity degree effectively protects potential persistent items. Moreover, even in cases where a small fraction of items reappear frequently after long absences, Pandora remains highly effective at capturing these items. Its probability-based update strategy (described later) is specifically designed to protect items with higher persistence, minimizing the risk of incorrect eviction and achieving higher detection accuracy compared to existing methods like On-Off Sketch [62]. This is further validated in Section 5.8.4, where we present a detailed evaluation using wave-based traces.

### 3.2 Data Structure

Figure 2 illustrates Pandora's data structure, consisting of $r$ rows, each containing $b$ buckets. Each row is associated with a unique pairwise-independent hash function $h_1, h_2, \ldots, h_r$. We denote the bucket located in the $p$-th row and $q$-th column as $B(p, q)$, where $1 \leq p \leq r, 1 \leq q \leq b$. Every bucket comprises four fields: $B(p, q).K$ tracks the key of the recorded item; $B(p, q).V$ stores the persistence value of the tracked item; $B(p, q).I$ tracks the inactivity degree of the incumbent item (the calculation method is described in Section 3.3.1); $B(p, q).F$ is a flag field to avoid duplicates, with two possible states, $True$ or $False$. At the start of each time window, all flag bits in every bucket are reset to $True$. When the tracked item arrives in the current window, its flag bit is set to $False$, indicating that the bucket's value counter cannot be further incremented.

In contrast to hierarchical data structures [60], which attempt to separate persistent and non-persistent items, Pandora's flat data structure eliminates the need for complex update procedures, such as tracking relationships between counters across layers, ensuring higher update speed. Additionally, by carefully replacing non-persistent items, Pandora achieves high lookup accuracy, as further confirmed in Section 5. Hence, we opt for the flat data structure. Moreover, tracking item keys rather than fingerprints [56] maintains excellent invertibility [49], mitigating redundant hash operations during updates and queries.

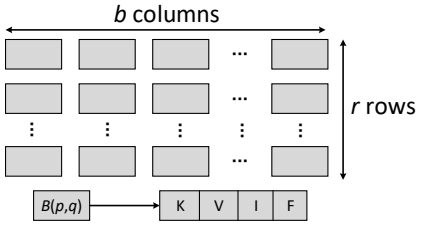



Fig. 2. Data structure of Pandora.

### 3.3 Pandora's Operations

*3.3.1 Update.* Algorithm 1 outlines Pandora's update process. Initially, all persistence counters ($B(p,q).V$), inactivity degrees ($B(p,q).I$), and flags ($B(p,q).F$) in the buckets are initialized to 0, 0, and $True$, respectively, while all key fields ($B(p,q).K$) are set to *null* (Line 1). At the end of each time window, all flag fields ($B(p,q).F$) are reset to $True$ for the next window (Line 29).

**Case 1: Finding an available bucket.** When an incoming item $e_i$ first arrives in a new time window, it utilizes the hash function $h_1$ associated with the first row to locate the bucket $B(1, h_1(e_i))$. If this bucket is empty or already contains $e_i$ (indicating a match), the status of the hashed bucket is updated as follows: the key field $B(1, h_1(e_i)).K$ is set to $e_i.key$, the persistence value $B(1, h_1(e_i)).V$ is incremented by 1, and the inactivity value $B(1, h_1(e_i)).I$ is decreased by 1 (no smaller than 0) since $e_i$ has just arrived. The flag field $B(1, h_1(e_i)).F$ is then changed to $False$ to avoid redundant increments of $e_i$'s persistence within the current time window. After this update, the hash operations cease. By recording $e_i$ in a single bucket, Pandora conserves memory space and avoids redundantly tracking the same item across multiple buckets in different rows [26]. However, if the hashed bucket $B(1, h_1(e_i))$ is occupied by a different item, Pandora recursively applies the hash functions $h_2, \ldots, h_r$ to locate an available bucket in the subsequent rows (Lines 2-9).

**Case 2: Hash collisions in all rows.** If the incoming item $e_i$ experiences hash collisions in all rows, indicating that the hashed buckets are occupied by other items, $e_i$ identifies the bucket $B(g,m)$ among the $r$ hashed buckets with the smallest persistence value $B(g,m).V$ (Line 10). If the flag field $B(g,m).F$ is $False$, indicating that the incumbent item has arrived in the current time window, the replacement operation for $e_i$ is abandoned (Lines 11-12). Otherwise, the replacement rate $R$ is computed as $\frac{1}{\max(1,(Z-B(g,m).I))\times B(g,m).V+1}$, where $Z$ is a predefined constant (e.g., 50), $B(g,m).I$ is the inactivity degree of the tracked item, and $B(g,m).V$ is its persistence value (Lines 13-14).

When a time window ends, Pandora checks the flag status in each bucket. If the flag is $True$, indicating that the tracked item did not arrive in this window, the corresponding inactivity degree $B(g,m).I$ is incremented by 1. Otherwise, $B(g,m).I$ is decremented by 1 (but not below 0) (Lines 22-28). Thus, for a potential persistent item that appears in most windows, its inactivity degree remains low. In the replacement probability formulation, we constrain the minimum value of $(Z - B(g,m).I)$ to be 1. Consequently, for a potential persistent item with a large persistence $B(g,m).V$ and a small inactivity degree $B(g,m).I$, its replacement probability is minimized, making it less likely to be evicted by other items.

If the replacement is successful, the incoming item $e_i$ will replace the tracked item in the bucket $B(g,m)$. The key field $B(g,m).K$ is set to $e_i.key$, the persistence value $B(g,m).V$ is reset to 1 (to avoid overestimation errors), the inactivity degree $B(g,m).I$ is set to 0, and the flag $B(g,m).F$ is set to $False$ (Lines 15-19). Otherwise, if the replacement is unsuccessful, the incoming item $e_i$ will be discarded directly (Lines 20-21).

Incorporating the inactivity degree during the replacement procedure offers two key advantages over methods that solely consider item persistence. First, by taking into account the arrival patterns of persistent items, the inactivity degree adds a layer of protection that mitigates the risk of potential persistent items being inadvertently evicted by a surge of non-persistent items. This safeguard is particularly crucial in scenarios with highly skewed data distributions, where numerous non-persistent items could otherwise displace persistent ones. Second, unlike conventional heavy hitter detection tasks that rely on item frequency, the persistence metric typically increments by only 1 within a time window, regardless of an item's frequency, making it much smaller in magnitude. Consequently, persistent items are more vulnerable to eviction. By incorporating

**Algorithm 1:** Pandora's Update Operation

**Input:** an item $e_i$, hash functions $h_1, h_2, ..., h_r$, number of rows $r$, number of buckets per row $b$, predefined constant $Z$

**Output:** Update the data structure for the incoming item $e_i$

**Initialization:** All persistence counters $B(p,q).V$, inactivity degrees $B(p,q).I$, and flags $B(p,q).F$ are initialized to 0, 0, and True, respectively. All key fields $B(p,q).K$ are set to NULL.

**for** $p = 1$ *to* $r$ **do**
  $pos \leftarrow h_p(e_i.key)$ `// Compute hash position for row p`
  **if** $B(p,pos).K = NULL$ ***or*** $B(p,pos).K = e_i.key$ ***and*** $B(p,pos).F = True$ **then**
    `// Empty bucket or bucket contains` $e_i$
    $B(p,pos).K \leftarrow e_i.key$;
    $B(p,pos).V \leftarrow B(p,pos).V + 1$; `// Increment persistence`
    $B(p,pos).I \leftarrow \max(B(p,pos).I - 1, 0)$; `// Decrement inactivity`
    $B(p,pos).F \leftarrow False$;
    **return**;
    `// Item inserted or updated, update complete`
`// All buckets occupied, attempt replacement`
$g, m \leftarrow \arg\min_{p,q} B(p,q).V$; `// Find bucket with smallest persistence`
**if** $B(g,m).F = False$ **then**
  Discard $e_i$;
**else**
  $R \leftarrow \frac{1}{\max(1,(Z - B(g,m).I)) \times B(g,m).V + 1}$ ;
  **if** *random(0, 1)* $< R$ **then**
    `// Replacement successful`
    $B(g,m).K \leftarrow e_i.key$ ;
    $B(g,m).V \leftarrow 1$ ;
    $B(g,m).I \leftarrow 0$ ;
    $B(g,m).F \leftarrow False$ ;
  **else**
    `// Replacement unsuccessful, discard` $e_i$
    Discard $e_i$;
**if** *End of a time window* **then**
  **for** $p = 1$ *to* $r$ **do**
    **for** $q = 1$ *to* $b$ **do**
      **if** $B(p,q).F = True$ **then**
        $B(p,q).I \leftarrow B(p,q).I + 1$;
      **else**
        $B(p,q).I \leftarrow \max(B(p,q).I - 1, 0)$;
      $B(p,q).F \leftarrow True$; `// Reset flag for next window`

the inactivity degree as an additional metric, Pandora enhances the resilience of persistent items against replacement, enabling more accurate tracking.

*3.3.2 Differences From LRU Caching.* Note that while Pandora draws on the principle of deteriorating relevance over time, similar to Least Recently Used (LRU) [44], it offers a significant advancement tailored to persistence-based tasks in high-speed data streams. Unlike LRU, which focuses solely on recency, Pandora adopts a multifaceted approach by considering both occurrence patterns and persistence value, providing a more nuanced understanding of item significance. Additionally, Pandora introduces a probabilistic replacement mechanism, moving beyond LRU's deterministic eviction strategy. This enables Pandora to better handle the skewed nature of real-world data streams, particularly in memory-constrained environments. By efficiently retaining persistent items, Pandora effectively addresses the challenges of persistence-based tasks in data stream processing, where traditional LRU methods fall short. We further validate this through empirical tests in Section 5.8.3.

*3.3.3 Running Examples.* To illustrate the update process clearly, we provide several running examples as shown in Figure 3. In this instance, the sketch is initialized with two rows, each containing three buckets, and the parameter $Z$ is set to 5. All events from $T_0$ to $T_3$ occur within a single time window.

**(1) At $T_0$ (top-left)**: When item $e_1$ arrives, it first uses the hash function $h_1$ to locate a bucket in the first row. Since the hashed bucket is empty, item $e_1$ is inserted. The bucket's status updates from $(Null, 0, 0, T)$ to $(e_1, 1, 0, F)$, and no further action is required.

**(2) At $T_1$ (top-right)**: When item $e_2$ arrives, it utilizes two hash functions to search for an available bucket. It finds a suitable bucket in the second row, where the flag is set to $True$, indicating that $e_2$ has not yet arrived in this time window. The bucket's status is then updated: the persistence value increments by 1, the inactivity degree decreases by 1, and the flag is set to $False$.

**(3) At $T_2$ (bottom-left)**: When item $e_8$ arrives, it encounters hash collisions in all rows. To resolve this, it identifies the bucket with the smallest persistence value, which is tracking $e_9$, and whose flag is set to $True$. Pandora attempts to replace the incumbent item $e_9$ with a replacement probability of $\frac{1}{(5-2)\times 3+1}$. If the replacement is successful, item $e_8$ replaces $e_9$, and the bucket's status changes from $(e_9, 3, 2, T)$ to $(e_8, 1, 0, F)$. Otherwise, item $e_8$ is discarded.

**(4) At $T_3$ (bottom-right)**: When item $e_4$ arrives, hash collisions occur across all rows. The bucket with the smallest value, tracking item $e_7$, is identified. However, the flag for this bucket is $False$, indicating that $e_7$ has already appeared in the current window. Therefore, item $e_4$ abandons the replacement and is discarded.

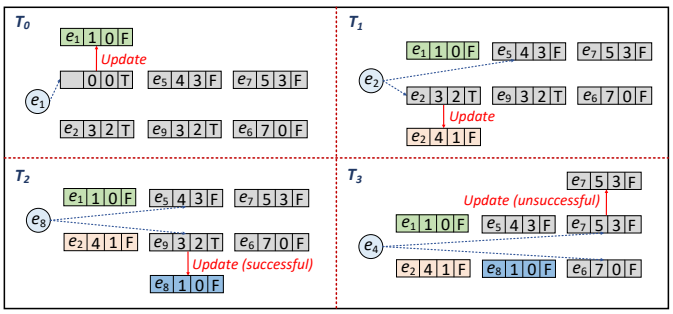



Fig. 3. Running examples.

*3.3.4 Query.* The query can be conducted either in real-time during measurement (online) or after the measurement period has concluded (offline) [25]. The detailed query process for each lookup task is described in the following subsection.

### 3.4 Deployment on Various Tasks

(i) Persistence Estimation: The update process follows the same steps as previously described. To query the persistence of all items, Pandora uses the following approach: for items tracked in the buckets, their persistence is directly obtained from the value stored in the corresponding bucket $B(p,q).V$. For items not present in any bucket, Pandora applies the $r$ hash functions to identify $r$ candidate buckets, one in each row. The smallest persistence value among these $r$ candidate buckets is used as the item's estimated persistence.

(ii) Persistent Item Lookup: The update process is identical to that in the previous subsection. For querying, Pandora performs a single scan of all buckets to retrieve the persistence values $B(p,q).V$ of tracked items. Any item $e_i$ whose retrieved persistence $B(p,q).V \geq \gamma W$ is classified as a persistent item.

(iii) Persistent Items with High Frequency: Pandora extends its data structure to identify persistent items with high frequency by adding a frequency field to each bucket. The update process maintains the same persistence logic as before, while incrementing the item frequency by 1 for each arrival. During hash collisions across all rows, Pandora selects the bucket with the smallest sum of persistence and frequency. The replacement probability becomes $\frac{1}{(Z-B(g,m).I)\times B(g,m).V\times B(g,m).F+1}$, where $B(g,m).F$ is the tracked item's frequency. Successful replacement resets both persistence and frequency to 1; otherwise, the incoming item is discarded. In the query process, Pandora identifies an item as persistent with high frequency if it meets two criteria: (1) persistence condition: $B(p,q).V \geq \gamma W$, and (2) frequency condition: frequency value exceeds $\theta F$, where $\theta$ is a predefined threshold and $F$ is the total frequency of all items.

## 4 Error Bound of Pandora

We provide a mathematical analysis of Pandora's estimation error bound, with a focus on persistent item lookup as the detection task.

Theorem 4.1. *Consider the set of persistent items under study, ordered by their persistence values in descending order, with $e_i$ denoting the item in the $i$-th position. For any persistent item $e_i$ and a small positive real number $\epsilon$, we define a probability bound on the estimation error. This bound quantifies the likelihood that the absolute difference between the true value $V_{e_i}$ and its estimate $\hat{V}_{e_i}$ exceeds $\epsilon V$, where $V = \sum_{e_i \in \mathcal{U}} V_{e_i}$ represents the total persistence of all items in the data stream. The probability bound is given by:*

$$\begin{aligned}
&\Pr\{V_{e_i} - \hat{V}_{e_i} \geq \epsilon V\} \\
&\leq \frac{V_{e_i} \cdot 2a - \left(\Delta + \sqrt{\Delta^2 - \frac{4(a^2+1)}{a} \cdot V_{e_i} \cdot Z_{eff} - \frac{4(a^2+1)}{a} \cdot Pr_w \cdot Pr_s}\right)}{2a \cdot \epsilon V}
\end{aligned}$$

*where:*

$$\begin{aligned}
&Pr_w = \binom{i-1}{r-1}\left(\frac{1}{b}\right)^{r-1}\left(1-\frac{1}{b}\right)^{i-r}, Pr_s = \left[1-\left(1-\frac{1}{b}\right)^{n-1}\right]^r, \\
&a = \frac{W}{V_{e_i}}, Z_{eff} = Z - \frac{W}{W - V_{e_i} + 1}, \Delta = Z_{eff} + Pr_w \cdot Pr_s + V_{e_i} \cdot a.
\end{aligned} \tag{1}$$

*Here, $W$ denotes the number of time windows, $b$ represents the number of buckets in each row, $n$ indicates the number of items, and $r$ is the number of rows.*

Proof. First, we calculate the probability $Pr_s$ that a newly arrived item $e_j$ encounters all $r$ rows with occupied buckets. This probability is given by:

$$Pr_s = \left[1 - \left(1 - \frac{1}{b}\right)^{n-1}\right]^r, \tag{2}$$

where $b$ is the number of buckets per row, and $n$ is the total number of distinct items in the stream. For any single bucket, the probability of an item being mapped to that bucket is $\frac{1}{b}$. Therefore, the probability that a bucket remains free from other items is $\left(1 - \frac{1}{b}\right)^{n-1}$. Thus, the probability of a hash collision occurring in the bucket for item $e_j$ is $1 - \left(1 - \frac{1}{b}\right)^{n-1}$. Since there are $r$ rows, the probability $Pr_s$ of item $e_j$ encountering collisions across all rows is represented as the product of these probabilities.

Next, we define $Pr_w$ as the likelihood that item $e_i$ retains its status as the item with the minimum persistence count across all rows where $e_j$ is mapped. This can be expressed as:

$$Pr_w = \binom{i-1}{r-1}\left(\frac{1}{b}\right)^{r-1}\left(1 - \frac{1}{b}\right)^{i-r}, \tag{3}$$

where $i$ represents the index of the current item.

Now, considering the total number of time windows $W$, we assume that for the persistent item $e_i$, with a real persistence of $V_{e_i}$, the probability of its appearance in each time window is $\frac{V_{e_i}}{W}$. Thus, the expected persistence of item $e_i$ at the $t$-th time window is:

$$E(V_{e_i}^t) = \frac{t}{W} \cdot V_{e_i}. \tag{4}$$

Next, we calculate the inactivity value of item $e_i$ at the $t$-th time window. The inactivity value is given by:

$$I_{e_i}^t = \left[t - E(V_{e_i}^t)\right] - E(V_{e_i}^t) + E(B_{e_i}^t), \tag{5}$$

where $E(B_{e_i}^t)$ represents the expected number of consecutive time windows in which the item appears at the beginning.

The term $E(B_{e_i}^t)$ follows a geometric distribution and can be derived as:

$$E(B_{e_i}^t) = \frac{1}{1 - \frac{V_{e_i}}{W}} \approx \frac{W}{W - V_{e_i} + 1}. \tag{6}$$

Thus, the inactivity value of item $e_i$ at the $t$-th time window becomes:

$$I_{e_i}^t = t - 2 \cdot \frac{t}{W} \cdot V_{e_i} + \frac{W}{W - V_{e_i} + 1}. \tag{7}$$

Let us model the persistence of item $e_i$ as a series of discrete states $k$, where $k \in \{1, 2, \ldots, V_{e_i}\}$. Based on the assumption for $E(V_{e_i}^t)$, $I_{e_i}^k$ can be approximately formulated as:

$$I_{e_i}^k \leq \frac{W}{V_{e_i}} \cdot k + \frac{W}{W - V_{e_i} + 1}. \tag{8}$$

Note that the inactivity value $I_{e_i}^k$ is bounded below by 0. If the calculated value is less than 0, the inactivity is set to 0.

We assume that once a persistent item $e_i$ is placed in a bucket, it can be replaced by other items at most once, and when it arrives, it can successfully re-enter the bucket at once. Given these considerations, we can formulate the expected number of times the persistence value of item $e_i$

decreases due to replacement. This expectation, denoted as $E(X_{e_i})$, is derived from the previously established probabilities and can be expressed as:

$$\begin{aligned} E(X_{e_i}) &= \sum_{k=1}^{E(\hat{V}_{e_i})} V_{e_i}^k \cdot \frac{1}{(Z - I_{e_i}^k) \cdot V_{e_i}^k + 1} \cdot Pr_w \cdot Pr_s \\ &< \sum_{k=1}^{E(\hat{V}_{e_i})} \frac{1}{(Z - I_{e_i}^k)} \cdot Pr_w \cdot Pr_s \\ &\leq Pr_w \cdot Pr_s \cdot \sum_{k=1}^{E(\hat{V}_{e_i})} \frac{1}{Z - \left(\frac{W}{V_{e_i}} \cdot k + \frac{W}{W - V_{e_i} + 1}\right)}. \end{aligned} \tag{9}$$

For the summation:

$$\sum_{k=1}^{E(\hat{V}_{e_i})} \frac{1}{Z - \left(\frac{W}{V_{e_i}} \cdot k + \frac{W}{W - V_{e_i} + 1}\right)}, \tag{10}$$

The term $\frac{W}{W - V_{e_i} + 1}$ can be treated as a constant since it does not depend on $k$. This constant can be combined with $Z$ to define a new effective constant $Z_{\text{eff}}$:

$$Z_{\text{eff}} = Z - \frac{W}{W - V_{e_i} + 1}, \tag{11}$$

where $Z_{\text{eff}}$ is constrained to no smaller than 0.

Thus, by setting $a = \frac{W}{V_{e_i}}$, the summation simplifies to:

$$\sum_{k=1}^{E(\hat{V}_{e_i})} \frac{1}{Z_{\text{eff}} - a \cdot k}, \tag{12}$$

This summation can be approximated via the integral approach:

$$\sum_{k=1}^{E(\hat{V}_{e_i})} \frac{1}{Z_{\text{eff}} - a \cdot k} \approx \int_1^{E(\hat{V}_{e_i})} \frac{dx}{Z_{\text{eff}} - a \cdot x} = \frac{1}{a} \ln\left(\frac{Z_{\text{eff}} - a}{Z_{\text{eff}} - a \cdot E(\hat{V}_{e_i})}\right). \tag{13}$$

Based on the above, we compute the expected persistence of $V_{e_i}$:

$$\begin{aligned} E(\hat{V}_{e_i}) = V_{e_i} - E(X_{e_i}) &\geq V_{e_i} - Pr_w \cdot Pr_s \cdot \frac{1}{a} \cdot \ln\left(\frac{Z_{\text{eff}} - a}{Z_{\text{eff}} - a \cdot E(\hat{V}_{e_i})}\right) \\ &\approx V_{e_i} - Pr_w \cdot Pr_s \cdot \frac{1}{a}\left(\frac{Z_{\text{eff}} - a}{Z_{\text{eff}} - a \cdot E(\hat{V}_{e_i})} - 1\right). \end{aligned} \tag{14}$$

By solving formula (14) and applying scaling, we obtain:

$$E(\hat{V}_{e_i}) \approx \frac{\Delta + \sqrt{\Delta^2 - \left[\frac{4(a^2+1)}{a} \cdot V_{e_i} \cdot Z_{\text{eff}} + \frac{4(a^2+1)}{a} \cdot Pr_w \cdot Pr_s\right]}}{2a}, \tag{15}$$

where:

$$\Delta = Z_{\text{eff}} + Pr_w \cdot Pr_s + V_{e_i} \cdot a.$$

Furthermore, by applying the Markov inequality, we deduce that:

$$Pr\{V_{e_i} - \hat{V}_{e_i} \geq \epsilon V\} \leq \frac{V_{e_i} - E(\hat{V}_{e_i})}{\epsilon V}$$
$$= \frac{V_{e_i} \cdot 2a - \left(\Delta + \sqrt{\Delta^2 - \frac{4(a^2+1)}{a} \cdot V_{e_i} \cdot Z_{\text{eff}} - \frac{4(a^2+1)}{a} \cdot Pr_w \cdot Pr_s}\right)}{2a \cdot \epsilon V}$$

□

To validate the accuracy of the derived error bound, we conduct empirical tests using the NY18 trace. We adjust $\epsilon$ to set $\epsilon V = 50$, with $Z = 50$, $W = 1600$, and $r = 4$. The parameter $b$ is computed based on the memory budget and $r$. Following the method in [56], we vary the memory size from 10KB to 50KB. As shown in Figure 4, the derived error bound consistently exceeds the empirical values across the tests, confirming the accuracy of the theoretical bound. We also test with memory sizes over 200KB and other $\epsilon V$ values (such as 30), finding that the bound remains effective (figure omitted due to space constraints).

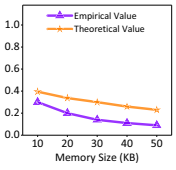



Fig. 4. Theoretical bound vs. actual value.

## 5 Evaluation

### 5.1 Setup

*5.1.1 Testbed.* We implement Pandora in C++ and evaluate its performance on a machine with an Intel(R) Core(TM) i5-1135G7 @ 2.40GHz processor and 16GB DRAM, running Ubuntu 20.04 LTS. The processor features a 48KB L1 data cache and 1,280KB L2 cache per core, with all cores sharing an 8,192KB L3 cache.

*5.1.2 Datasets.* For the evaluation, we use five real-world datasets from major backbone routers: four from the US (CHI15 [7], CHI16 [8], NY18 [9], NY19 [10], spanning 2015–2019), and one from Japan (JPN20 [41], 2020). The datasets vary in size and complexity: CHI15 (20.2M packets, 520K items), CHI16 (22.3M packets, 730K items), NY18 (22.3M packets, 760K items), NY19 (29.5M packets, 1.53M items), and JPN20 (44.5M packets, 2.75M items).

*5.1.3 Implementation.* We employ MurmurHash [3] to hash incoming items, using source-destination pairs as unique identifiers across all datasets. Each pair is represented using 8 bytes. For each trace, we adjust the window size based on the trace length, dividing the data into 1,600 time windows ($W = 1600$) [62], and set the default persistence threshold $\gamma$ to 0.5. In our implementation, we detected 682, 587, 747, 1424, and 28 real persistent items in the CHI15, CHI16, NY18, NY19, and JPN20 traces, respectively. To identify high-frequency persistent items, we set the frequency threshold $\theta$ to $5 \times 10^{-4}$.

The algorithms, including Pandora and the benchmarks, are implemented in C++. In Pandora, each bucket uses 8 bytes to store the item key, 2 bytes for the item value (since the maximum number of time windows is 1600), 2 bytes to track the inactivity degree, and 1 byte for a flag. In our setup, we use 4 rows [35, 49], with the number of buckets $b$ in each row determined by the available memory. For example, with 32KB of memory, each row contains approximately 630 buckets, calculated as $b = \frac{32 \times 1024}{4 \times 13} \approx 630$.

*5.1.4 Benchmarks.* For persistence estimation, we compare Pandora with the state-of-the-art On-Off Sketch [62] and Pyramid-based On-Off Sketch [60]. For persistent item lookup, we compare Pandora against On-Off Sketch [62], WavingSketch [33], and Small-Space (SS) [32]. Although PIE

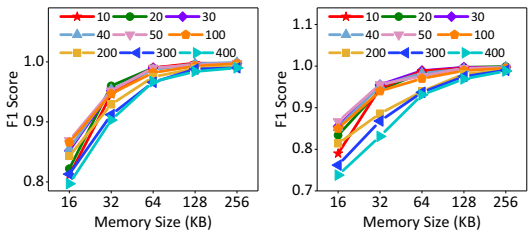
[18] is another solution for persistent measurements, it is not functional in small memory configurations, so we omit it from our comparisons. The details of these methods are introduced in Section 6. For persistence estimation, we configure the number of rows in both On-Off Sketch and Pyramid On-Off Sketch to 8. For persistent item lookup, we vary the number of slots in On-Off Sketch and WavingSketch between 4 and 16. The configuration of SS remains consistent with [33].

*5.1.5 Offline Query and Online Query.* Queries can be made either online (i.e., during the measurement) or offline (i.e., after a measurement period ends) to obtain the persistence of different items [25]. Pandora supports both types of queries flexibly. Unless otherwise specified, queries are assumed to be offline, as is typical in most existing evaluations [34, 35, 58, 62]. We thoroughly assess Pandora's performance in an online setting in Section 5.5.

*5.1.6 Metrics.* The performance of the algorithm is evaluated using six key metrics. Recall (R) measures how many truly persistent items are correctly identified, defined as $R = \frac{|\mathcal{P} \cap \mathcal{P}'|}{|\mathcal{P}|}$, where $\mathcal{P}$ is the set of true persistent items and $\mathcal{P}'$ the set reported by the algorithm. Precision (P) evaluates the accuracy of the reported items, calculated as $P = \frac{|\mathcal{P} \cap \mathcal{P}'|}{|\mathcal{P}'|}$. F1 Score, $F_1 = \frac{2RP}{R+P}$, balances recall and precision. Average Absolute Error (AAE) is the average difference between true and estimated values of persistent items, while Average Relative Error (ARE) normalizes this difference by the true value, $\text{ARE} = \frac{1}{|\mathcal{P}|} \sum_{e_i \in \mathcal{P}} \frac{|V(e_i) - \hat{V}(e_i)|}{V(e_i)}$. Lastly, Throughput rate quantifies processing speed in millions of operations per second (Mops), averaged over five trials to ensure consistency.

## 5.2 Parameter Settings

Simpler sketches with fewer parameters are preferable, as they are easier to configure and deploy in practice. Pandora's design adheres to this principle by involving only one key parameter: the predefined parameter $Z$ for the replacement probability. Consistent with prior sketch-based work [33, 40, 64, 65], we conduct empirical evaluation to determine appropriate parameter configurations for our method. In line with [26, 34, 35, 61], we vary the memory budget from 16KB to 256KB. We utilize CHI15 and CHI16 as test traces and pick persistent item lookup as the test detection task.




(a) CHI15. (b) CHI16.

Fig. 5. F1 score in persistent item lookup with different parameter $Z$, as a function of memory size.

As shown in Figure 5, we observe that as $Z$ increases from 10 to 50, the F1 score of Pandora shows an increasing trend on both traces. The reason is that with a larger $Z$, it provides better protection for persistent items, making it less likely for them to be wrongly expelled from buckets by non-persistent items. However, as $Z$ further increases beyond 50, the detection accuracy displays a decreasing trend. This is because for an excessively large $Z$, once a non-persistent item enters a bucket, it becomes difficult to evict due to the small replacement probability. Overall, based on the results, we select $Z = 50$ as it can guarantee good detection accuracy.

We further vary the value of $Z$ to thoroughly assess its impact on detection accuracy in an online scenario for persistent item lookup. Using the NY18 trace, we define a time window size of 2000 packets and evaluate Pandora's ability to detect persistent items at intervals of 2M, 4M, 6M, 8M, 10M, and 15M packets. The threshold is adjusted to maintain the number of persistent items as 900, with the memory size set to 16KB. Figure 6 shows the F1 scores for persistent item lookup across different values of $Z$ and trace lengths. As illustrated, when $Z$ is set to 50, the F1 score consistently remains high across different intervals, indicating that this setting is effective.

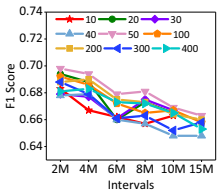
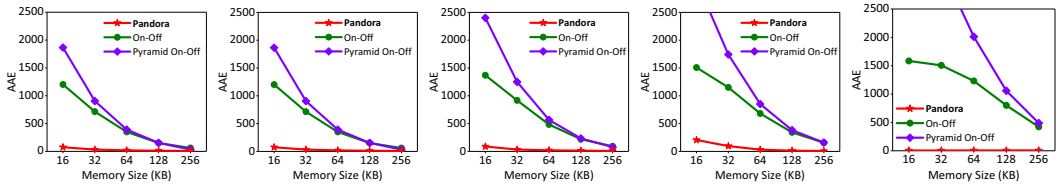



Fig. 6. F1 score for persistent item lookup with varying $Z$ at different lookup intervals.



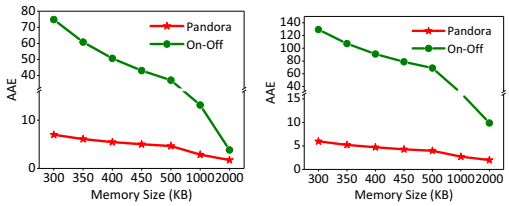

(a) CHI15. (b) CHI16. (c) NY18. (d) NY19. (e) JPN20

Fig. 7. Average Absolute Error (AAE) in persistence estimation with different algorithms, as a function of memory size.

### 5.3 Persistence Estimation

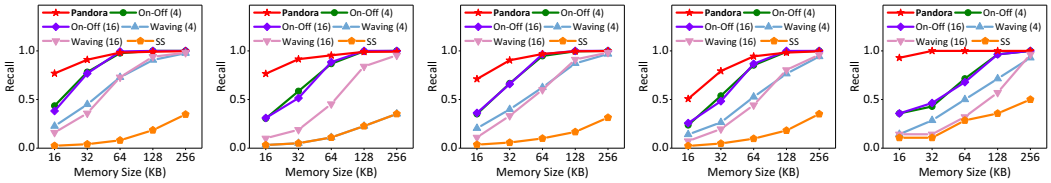
Figure 7 shows the AAE for persistence estimation across different methods and traces. Our method significantly reduces estimation error. On average, Pandora reduces AAE by 1610.39% and 2210.71%, 2531.99% and 4098.12%, 1792.49% and 2682.23%, 987.21% and 1654.82%, and 19555.86% and 39151.01% compared to On-Off Sketch and Pyramid On-Off Sketch on the CHI15, CHI16, NY18, NY19, and JPN20 traces, respectively. The substantial performance gain of Pandora comes from its cautious replacement strategy, reducing the likelihood of evicting persistent items and preventing overestimation errors. In contrast, On-Off Sketch and Pyramid On-Off Sketch use coarse swap replacement strategies, making persistent items more vulnerable to eviction, especially under tight memory constraints. Moreover, Pyramid On-Off Sketch tends to have higher AAE than On-Off Sketch due to overestimation when multiple counters share a parent and overflow.

**Performance with Larger Memory Settings:** We also evaluate Pandora with larger memory budgets to demonstrate its versatility and effectiveness beyond small memory scenarios. In this evaluation, we use persistence estimation as the detection task, varying memory sizes from 300KB to 2000KB, and employ the NY18 and NY19 traces. Pandora is compared with On-Off Sketch, configured with 16 slots. As shown in Figure 8, Pandora maintains high estimation accuracy even with larger memory. For example, with 2000KB of memory, Pandora's AAE is still 120.23% and 394.61% smaller than On-Off Sketch, highlighting its superior performance.



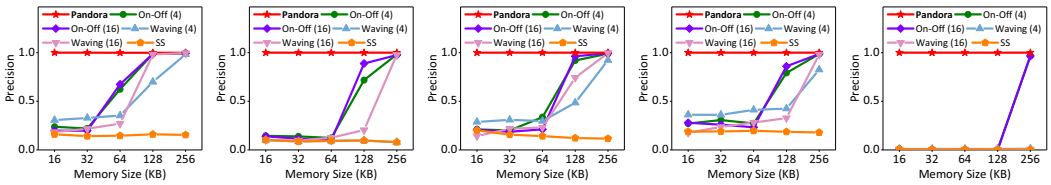

(a) AAE (NY18). (b) AAE (NY19).

Fig. 8. AAE for persistence estimation across different algorithms with larger memory budgets.



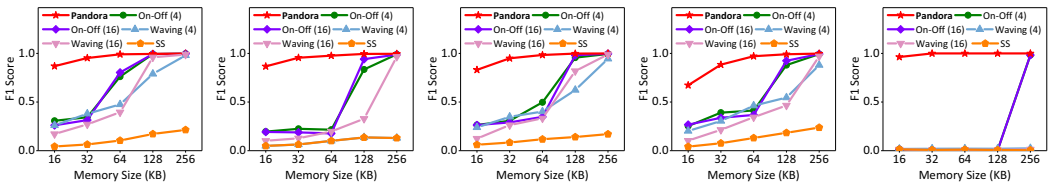

(a) CHI15. (b) CHI16. (c) NY18. (d) NY19. (e) JPN20.

Fig. 9. Persistent item lookup recall with different algorithms, as a function of memory size.



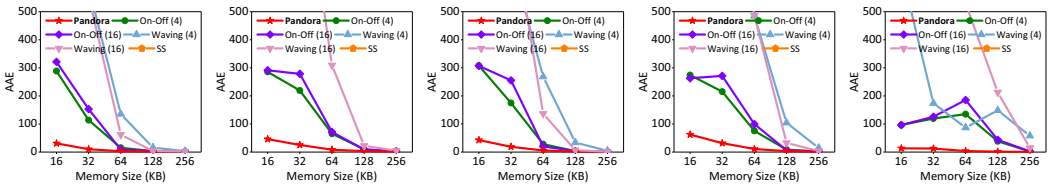

(a) CHI15. (b) CHI16. (c) NY18. (d) NY19. (e) JPN20.

Fig. 10. Persistent item lookup precision with different algorithms, as a function of memory size.



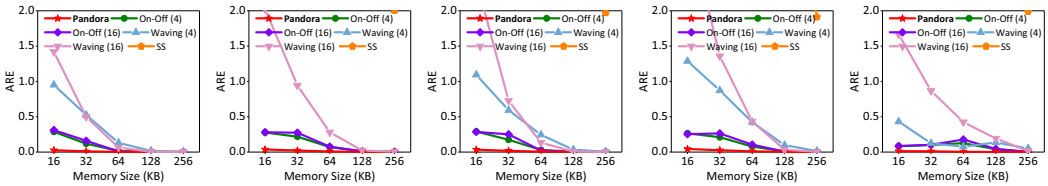

(a) CHI15. (b) CHI16. (c) NY18. (d) NY19. (e) JPN20.

Fig. 11. Persistent item lookup F1 score with different algorithms, as a function of memory size.



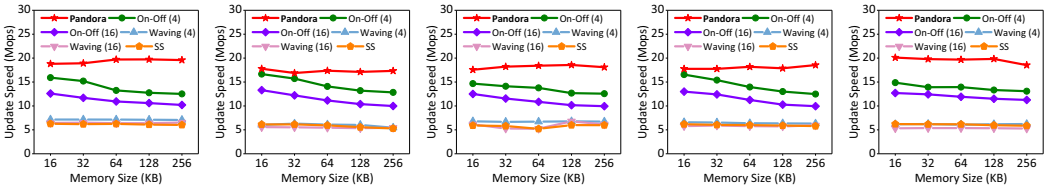

(a) CHI15. (b) CHI16. (c) NY18. (d) NY19. (e) MAWI.

Fig. 12. Average Absolute Error (AAE) in persistent item lookup with different algorithms, as a function of memory size.




(a) CHI15. (b) CHI16. (c) NY18. (d) NY19. (e) JPN20

Fig. 13. Average Relative Error (ARE) in persistent item lookup with different algorithms, as a function of memory size.




(a) CHI15. (b) CHI16. (c) NY18. (d) NY19. (e) JPN20

Fig. 14. Update Speed in persistent item lookup with different algorithms, as a function of memory size.

## 5.4 Persistent Item Lookup

*5.4.1 Detection Accuracy.* Figures 9-13 present the recall, precision, F1 score, AAE, and ARE for the persistent item lookup task across different traces. For On-Off Sketch and WavingSketch, the number in parentheses indicates the number of slots used.

As shown in Figure 9, Pandora achieves the highest recall rate. With a 16KB memory budget, Pandora's recall is 76.51%–160% higher than On-Off Sketch (4 slots) across the CHI15, CHI16, NY18, NY19, and JPN20 traces. Figure 10 shows Pandora's precision consistently at 1, significantly surpassing the benchmarks. On the JPN20 trace, the precision of other methods drops due to its more heavy-tailed distribution, but Pandora remains robust. Figure 11 highlights Pandora's F1 score as the highest across all approaches, outperforming On-Off Sketch, WavingSketch, and SS by up to 729.45% on the NY18 trace. In Figures 12 and 13, Pandora achieves the lowest AAE and ARE. On the CHI16 trace, Pandora reduces AAE and ARE by 589.56% and 717.9%, respectively, compared to On-Off Sketch (4).

The results emphasize Pandora's significant performance improvements over existing methods. On-Off Sketch uses a simple swap strategy to replace items, but in memory-constrained environments, hash collisions frequently result in the mistaken eviction of persistent items, reducing detection accuracy. WavingSketch employs a Bloom filter to remove duplicates, but this consumes memory, worsening hash collisions and increasing false positives. SS's sampling technique inefficiently tracks non-persistent items and struggles with low sampling rates in tight memory conditions, further increasing detection errors. In contrast, Pandora leverages inactivity degree and persistence value to handle replacements during hash collisions, preventing incorrect eviction of persistent items. Moreover, its probability-based replacement strategy introduces only small underestimation errors, ensuring robust detection accuracy.

*5.4.2 Processing Speed.* Figure 14 presents the update speed of various methods. Our scheme achieves the highest processing speed, outperforming others by 38.95%–213.66% on CHI15, 18.44%–214.08% on CHI16, 33.95%–214.36% on NY18, 25.65%–207.9% on NY19, and 41.65%–267.07% on JPN20. This performance boost is due to our streamlined update procedure, which avoids pointers or additional data structures. Once an item locates a bucket, the update process stops, reducing hash operations and enabling fast processing for high-speed data streams.

*5.4.3 Deep Dive.* We further evaluate our method against advanced methods, including Pyramid On-Off Sketch [60], P-Sketch [34], and Stable-Sketch [35], using the same settings for P-Sketch and Stable-Sketch as in [34]. Table 1 shows the F1 scores of these methods. As demonstrated, Pandora outperforms the other approaches, with increases of 29.32%, 55.73%, 14.95%, and 10.42% over Pyramid On-Off Sketch (4), Pyramid On-Off Sketch (16), P-Sketch, and Stable-Sketch, respectively, under the tight memory constraint of 16KB. While Pyramid On-Off Sketch employs finer-grained counters, it still inherits the limitations of On-Off Sketch. Additionally, under constrained memory, the overestimation errors caused by its counter designs further degrade detection accuracy.

P-Sketch, although utilizing a probability-based replacement strategy, suffers from both overestimation and underestimation errors, leading to lower detection accuracy compared to Pandora. Stable-Sketch, on the other hand, evicts items based on bucket status, which is affected by its eviction policy. Its probability decay method is slower at evicting high-persistence items that are not truly persistent, causing them to occupy space and preventing actual persistent items from entering the bucket during hash collisions, resulting in a lower F1 score.

Table 1. F1 score of different methods over the CHI15 trace.

| F1 Score | Pandora | Pyramid On-Off (4) | Pyramid On-Off (16) | P-Sketch | Stable-Sketch |
|---|---|---|---|---|---|
| 16KB | **0.869** | 0.672 | 0.558 | 0.756 | 0.787 |
| 32KB | **0.952** | 0.907 | 0.885 | 0.892 | 0.901 |
| 64KB | **0.991** | 0.963 | 0.971 | 0.965 | 0.961 |
| 128KB | **0.995** | 0.982 | 0.979 | 0.99 | 0.987 |
| 256KB | **0.998** | 0.991 | 0.99 | 0.992 | 0.992 |

### 5.5 Online Lookup

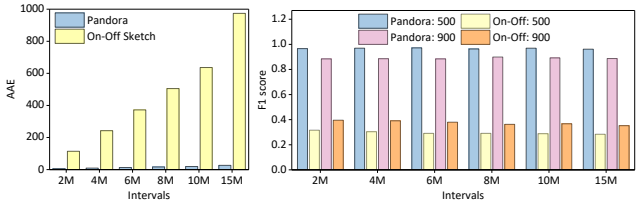
To assess Pandora's performance in live streaming applications, we test its real-time detection capabilities during data measurement [25, 59]. Without any modifications, Pandora can be directly employed for online lookup. Using the NY18 trace with a time window size of 2000 packets, we evaluate Pandora's performance on various persistence-based tasks at checkpoints of 2M, 4M, 6M, 8M, 10M, and 15M packets as the data stream progresses.

We first measure the accuracy of *persistence estimation* in the online setting, comparing Pandora with On-Off Sketch (also modified for online lookup) with a memory size of 32KB. As shown in Figure 15(a), Pandora achieves significantly lower estimation errors across various intervals compared to On-Off Sketch. For instance, at the 6M packet measurement point, Pandora's AAE is 13.07, whereas On-Off Sketch has an AAE of 372.15. For *persistent item lookup*, similar to [49], we adjust thresholds to maintain 500 and 900 detected persistent items at the end of each interval, ensuring real-time detection as the stream progressed. Figure 15(b) compares Pandora's detection accuracy with On-Off Sketch over time with 32KB of memory. Pandora consistently delivers higher detection accuracy in live scenarios. For instance, when detecting 500 persistent items, Pandora maintains an F1 score above 0.96, significantly higher than On-Off Sketch. Even when detecting 900 items, Pandora's F1 score remains around 0.9, demonstrating strong adaptability in real-time detection tasks. Further evaluation across other traces, including scenarios with 1500 persistent items, consistently shows Pandora outperforming existing baselines (figures omitted).

These real-time evaluations confirm that Pandora is well-suited for live streaming applications, demonstrating its ability to maintain high accuracy even in memory-constrained settings, and highlighting its robustness in real-time, high-speed data streams.

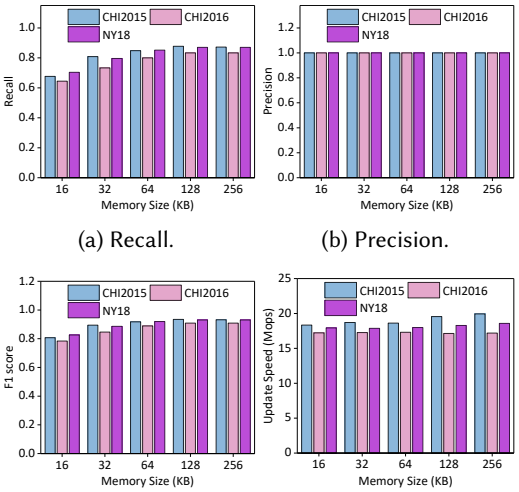
### 5.6 High-Frequency Persistent Item Lookup

Figure 16 illustrates Pandora's lookup performance for persistent items with high frequency across various memory budgets and traces. Our method demonstrates high effectiveness, with recall rates exceeding 0.8 for all considered traces at 64KB memory allocation (Figure 16(a)), and perfect precision (1.0) across all tested scenarios (Figure 16(b)). Notably, even with limited memory (16KB), Pandora's F1 score remains high, at or above 0.8 (Figure 16(c)). Additionally, Pandora maintains high efficiency with an average update speed exceeding 17.2Mops on various traces, showcasing its capability to perform well under constrained resources while maintaining both accuracy and speed.




(a) Persistence estimation. (b) Persistent item lookup.

Fig. 15. Accuracy of real-time persistent-based tasks at various measurement intervals.




(a) Recall. (b) Precision.

(c) F1 score. (d) Update Speed.

Fig. 16. Performance for high-frequency persistent item detection across different traces.

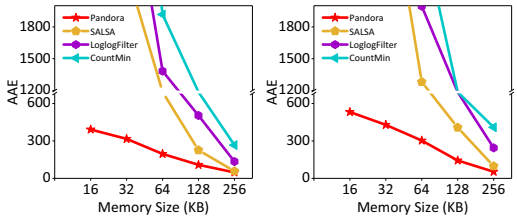
## 5.7 Performance on Frequency-Based Tasks

*5.7.1 Frequency Estimation.* When the number of time windows is set to 1, persistence estimation becomes frequency estimation, which provides the approximate frequency $f_i$ of each item. To do so, we have adapted Pandora by slightly modifying the update process to increment the counter by 1 when an item arrives, while removing the flag bit, keeping the rest of the update procedure unchanged. During the query process, if the item is already present in the sketch, its estimated value is reported directly. Otherwise, the item is hashed into all rows, and the bucket with the smallest value among the hashed positions is used for estimation. We compare Pandora with several frequency-based methods, including Count-Min [17], SALSA [4], and LogLogFilter [28]. Details about these methods are provided in Section 6. or Count-Min, we configure the sketch with 4 rows. For SALSA, we integrate it with Count-Min Sketch, also setting 4 rows and using 8-bit counters as the default configuration. For LogLogFilter, we employ the CU variant and allocate 75% of the memory to it. The evaluation is conducted using the NY18 and NY19 traces.

As shown in Figure 17, Pandora achieves the lowest estimation errors among the evaluated methods. Compared to the baselines, Pandora significantly reduces the estimation errors across various memory budgets. This superior performance is due to Pandora's enhanced protection for heavy items, ensuring high accuracy. For non-heavy items, Pandora selects the smallest value among the

hashed buckets, effectively reducing overestimation errors. In contrast, Count-Min Sketch does not track item keys, leading to substantial overestimation due to hash collisions, particularly in limited memory settings.

Additionally, Pandora demonstrates improved performance with larger memory settings (figures omitted due to the space limitation). For example, with 1000KB of memory, the AAE on the NY19 trace is lower than other baselines. While SALSA, LogLogFilter, and Count-Min record AAEs of 10.68, 9.14, and 62.87, respectively, Pandora achieves a much lower AAE of 7.49, highlighting its superior accuracy across different memory configurations.

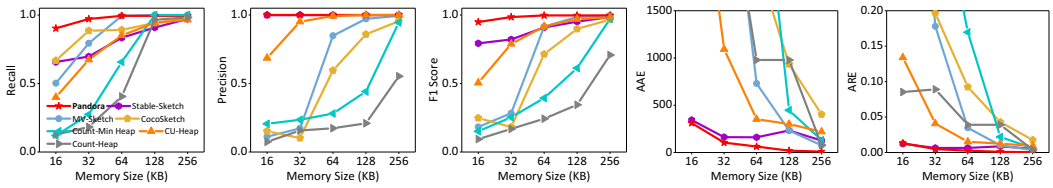



(a) AAE (NY18). (b) AAE (NY19).

Fig. 17. Performance of frequency estimation with different algorithms, as a function of memory size.

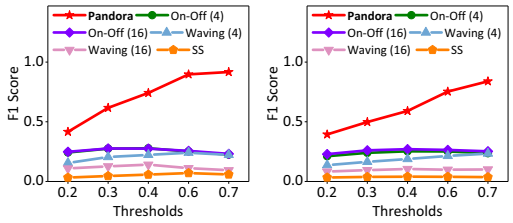
*5.7.2 Heavy Hitter Detection.* We further extend Pandora to heavy hitter detection. The update process remains the same as for frequency estimation. During query time, Pandora performs a scan to identify items whose frequencies exceed a predefined threshold. Here, we set the heavy hitter threshold as $\theta = 5 \times 10^{-4}$ [35]. We compare Pandora with several methods for heavy item lookup, including Stable-Sketch [35], MV-Sketch [49], CocoSketch [63], Count-Min Heap [17], CU-Heap [19], and Count-Heap [12]. For MV-Sketch and Stable-Sketch, the parameter settings are aligned with their original papers, while the remaining baselines follow the configurations from the literature [63].

Figure 18 displays the performance of different methods for heavy hitter detection. As shown, Pandora achieves the highest recall rate among all baselines, maintaining a recall rate of over 0.9 even with a limited memory budget of 16KB. Compared to the state-of-the-art Stable-Sketch, Pandora's average recall rate improves by 19.38%. Figure 18(b) shows that Pandora also consistently maintains a high precision rate across different memory budgets. Overall, Pandora delivers the highest F1 score, with improvements ranging from 10.4% to 216.04%. Additionally, Pandora achieves the lowest estimation errors. For AAE and ARE, Pandora reduces errors by 102.03% and 78.2% on average compared to the advanced Stable-Sketch. These results clearly demonstrate Pandora's effectiveness in the heavy hitter lookup task.



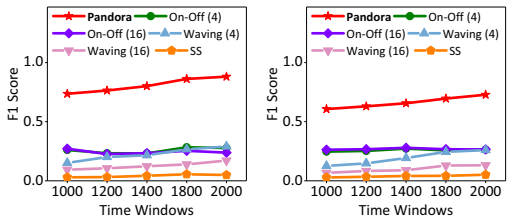

(a) Recall. (b) Precision. (c) F1 Score. (d) AAE. (e) ARE.

Fig. 18. Performance of heavy hitter detection with different algorithms, as a function of memory size.




(a) NY18. (b) NY19.

Fig. 19. F1 score for persistent item lookup with varying thresholds as a function of memory size.




(a) NY18. (b) NY19.

Fig. 20. F1 score for persistent item lookup with varying time windows as a function of memory size.

## 5.8 Multiple Cases

*5.8.1 Performance under Different Thresholds.* We employ persistent item lookup as the test task, varying the persistence threshold $\gamma$ from 0.2 to 0.7 to evaluate Pandora's robustness. The memory budget is set at 16KB. As depicted in Figure 19, our approach achieves the highest detection accuracy across different threshold settings. For instance, when the threshold is 0.3, Pandora's F1 score is 123.38% to 1458.9% higher than the considered benchmarks on the NY18 trace. Additionally, we observe that the performance gain of our method compared to the considered baselines increases as the threshold rises. This is because a higher threshold results in a decreased number of persistent items. In such scenarios, our method can still accurately capture persistent items under a tight memory budget, whereas existing approaches are prone to incorrectly evict persistent items by non-persistent ones.

*5.8.2 Performance under Different Number of Time Windows.* In the evaluation for persistent item lookup, each trace is initially partitioned into 1,600 time windows by default. To study the impact of varying the number of time windows on different lookup approaches, we adjust the number of time windows from 1,000 to 2,000 using a 16KB memory. As shown in Figure 20, Pandora consistently achieves the highest detection accuracy among the compared benchmarks. For instance, when using the NY19 trace with 1,000 windows, Pandora's F1 score is 130.85% higher than the competing On-Off Sketch (16). Additionally, we observe that the F1 score of our method increases as the number of time windows rises. This trend can be attributed to the decreasing number of persistent items in the data stream as the number of windows increases.

*5.8.3 Comparison with LRU-Based Approach.* The LRU-based method tracks the last access time for each item in the cache. When an item is accessed or inserted, its access time is updated. In cases of hash collisions across all rows, the least recently used item is identified and replaced to create space for the new entry. We evaluate this method on the NY18 trace and find that the F1 score reaches only 0.07 and 0.40 with memory sizes of 128KB and 256KB, respectively, which is

significantly lower than Pandora's performance (around 1). These results confirm that the LRU approach is inadequate for persistent item lookup in memory-constrained environments.

*5.8.4 Performance under Wave-Based Traces.* To further test the robustness of Pandora in scenarios involving wave-based arrival patterns, we generate synthetic traces where persistent items exhibit wave-like behavior. Specifically, we create a trace consisting of 1.5M items, with 1500 categorized as persistent and the remainder as non-persistent, distributed across 1600 time windows. An item is considered persistent if it appears in more than 800 of the 1600 time windows [62]. The persistent items are divided into two categories: a percentage ($Q\%$) of the persistent items follows a wave-based arrival pattern, where their arrival frequencies are controlled by a sine wave function. Each wave-based persistent item is assigned a distinct wave amplitude and frequency to regulate its arrival pattern. The wave amplitude controls the intensity of occurrences during peak periods, while the wave frequency governs the speed of transitions between peaks and valleys. The remaining $(100 - Q)\%$ of the persistent items follow a random arrival pattern, appearing in more than 800 windows but without the periodic wave behavior, simulating random intermittent activity. Non-persistent items are distributed across fewer than 800 time windows.

To evaluate the performance of existing methods on such traces, we vary the proportion of wave-based persistent items by adjusting $Q$ from 30 to 70, with a memory size of 32KB. As shown in Figure 21, Pandora's performance exhibits a slight decline as $Q$ increases. This is because a higher proportion of wave-based persistent items makes it more challenging for all methods to capture them, especially under tight memory constraints. Nevertheless, Pandora outperforms the existing On-Off Sketch across all configurations, showcasing its robustness.

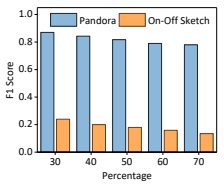



Fig. 21. F1 scores for persistent item lookup on wave-based trace.

## 5.9 Ablation Study

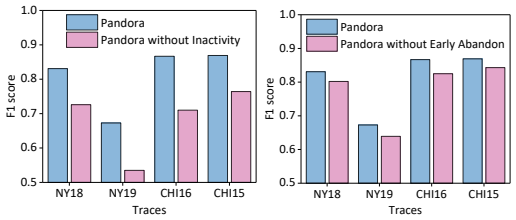
Pandora's design includes two key components: one accounts for item inactivity, and the other prevents replacement when an incoming item faces hash collisions across all rows and the flag of the bucket with the smallest persistence is set to *False*. To evaluate each component's effectiveness, we perform an ablation study using various traces with a 16KB memory budget.

*5.9.1 Considering Item Inactivity.* Figure 22(a) compares detection accuracy with and without considering item inactivity. While accounting for item inactivity increases memory usage per bucket, reducing the number of buckets, the performance gain outweighs this cost. Vanilla Pandora improves the F1 score by 14.46%, 25.77%, 22.08%, and 13.74% on the NY18, NY19, CHI16, and CHI15 traces, respectively. This improvement is due to the highly skewed item distribution, where item inactivity better protects persistent items from being prematurely replaced by non-persistent ones, resulting in higher accuracy.

*5.9.2 Early Abandon.* Figure 22(b) demonstrates the F1 score of Pandora with and without early abandon. As shown, the original Pandora enhances the F1 score by 3.61%, 5.3%, 5.06%, and 3.08% over the NY18, NY19, CHI16, and CHI15 traces, respectively. This showcases the effectiveness of employing the early abandon strategy.

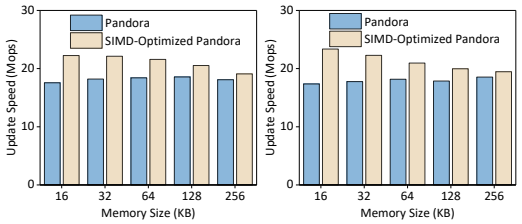
## 5.10 Boosting Speed with SIMD Instructions

Although Pandora already boasts the fastest update speed among the benchmarks, we can further enhance its performance using SIMD instructions. These instructions are widely supported across modern CPU architectures, including x86 (Intel and AMD) and ARM platforms, making SIMD




(a) Considering inactivity degree. (b) Early abandon.

Fig. 22. Ablation analysis of Pandora.




(a) NY18. (b) NY19.

Fig. 23. Update Speed with/without SIMD optimization.

optimizations applicable to a broad range of computing environments [5, 43, 47]. In our implementation, we utilized Intel's AVX2 instruction set, which is prevalent on most modern x86 processors [38].

This optimization primarily benefits the hash operations when an incoming item attempts to locate a bucket. In the standard version of Pandora, up to $r$ hash operations may be required in the worst case to locate a bucket. However, SIMD-optimized Pandora performs all $r$ hash operations in parallel, reducing the process to a single step and eliminating redundant operations. By simultaneously obtaining hash values for all $r$ buckets, Pandora can quickly track the incoming item in an empty or matching bucket if one is available, significantly enhancing efficiency. Figure 23 illustrates the update speed with and without SIMD optimization of Pandora for persistent item lookup. It shows that with SIMD instructions, the update speed increases by 16.22%, and 18.21% over the NY18, and NY19 traces, respectively, on average.

## 6 Related Work

**Persistence Estimation:** A basic approach to persistence estimation combines a conventional sketch, like Count-Min Sketch [17], with a Bloom filter [39]. Incoming items pass through the Bloom filter to check if they have appeared in the current window, and if not, proceed to the Count-Min Sketch. However, the Bloom filter's false positives become problematic under constrained memory (e.g., L1 Cache). On-Off Sketch [62] improves estimation by using an On/Off flag to remove duplicates, but it mixes persistent and non-persistent items, leading to significant overestimation errors, especially with limited memory. Pyramid On-Off Sketch [60] enhances memory efficiency with a hierarchical counter structure but inherits the same issues and its added complexity slows update speed.

**Persistent Item Lookup:** Small-Space (SS) [32] uses a sampling-based technique with a hash table to track item persistence and the most recent time window. However, once an item is stored,

it remains indefinitely, leading to inefficient memory usage. Additionally, the sampling rate decreases with constrained memory, reducing detection accuracy. PIE [18] encodes item keys to minimize memory, but this slows down updates, and storing Raptor codes for non-persistent items further wastes memory. On-Off Sketch [62] can be adapted for persistent item lookup, but its swap mechanism often displaces persistent items, leading to accuracy loss as hash collisions increase. WavingSketch [33], designed for frequency estimation, can be adapted for persistent item lookup by adding a Bloom filter, but this introduces Bloom filter-related issues and incorrect replacements during hash collisions, especially under tight memory. P-Sketch [34] uses a probability-based replacement strategy but suffers from both underestimation and overestimation errors, limiting accuracy in real-world scenarios. Stable-Sketch [35] relies on bucket status for eviction, but its probability decay method is slow at removing non-persistent high-persistence items, leading to inefficient memory use and reduced accuracy.

**Frequency Estimation:** When the number of time windows is set to 1, persistence-based tasks convert to frequency-based tasks. Count-Min Sketch [17] is a classic method for frequency estimation, where each incoming item is hashed into multiple rows, and the corresponding counters are incremented. The smallest counter value across all rows is used as the estimated frequency. However, this method suffers from significant overestimation errors, particularly in limited memory scenarios. Variants like CU-Sketch [19] and Count-Sketch [12] have been proposed to mitigate this, but similar issues persist. LogLog Filter [28] enhances accuracy through a two-stage update process that first filters out low-frequency items, while SALSA [4] starts with small counters and merges them into larger ones with more bits upon overflow. However, SALSA requires an additional bitmap to track counter status, leading to memory inefficiency and a slower decoding process.

**Heavy Hitter Lookup:** To identify items with high frequency, Count-Min, CU, and Count Sketch rely on an additional heap, causing memory inefficiency. Moreover, these methods are non-invertible, requiring a full scan of items in the data stream to retrieve all heavy items. MV-Sketch uses a majority vote algorithm, while CocoSketch [63] applies stochastic variance minimization for heavy item lookup. However, since these approaches replace items based only on their frequency, they fail to effectively protect heavy items, often leading to their replacement by non-heavy items and resulting in reduced accuracy.

## 7 Future Work

**(i) Dynamic Parameter $Z$:** While a fixed $Z$ has proven effective in many cases, and our current configuration aligns with established works [14, 36, 40, 64], we recognize the benefits of dynamically adjusting $Z$ based on traffic patterns. As a future direction, we propose determining optimal $Z$ values for varying traffic skewness levels offline and creating a mapping table to associate each skewness level with its optimal $Z$.

This approach involves generating synthetic datasets with different traffic skewness levels, performing a grid search to identify the optimal $Z$ values, and constructing the mapping table. In the online phase, a sliding window mechanism would monitor traffic skewness (e.g., using Pearson's coefficient). When significant changes in skewness are detected (e.g., exceeding a predefined threshold), the system would automatically adjust $Z$ by consulting the mapping table, enabling adaptive tuning to improve performance under varying traffic conditions.

**(ii) Balancing Memory Efficiency and Processing Speed:** For now, we allocate the size of each field in our data structure at the byte level. While exploring bit-level item tracking could optimize memory usage, this approach often compromises processing speed [4], highlighting the importance of balancing memory efficiency and speed as a key priority for future work.

## 8 Conclusion

In this paper, we present Pandora, an advanced sketch method designed for efficient and accurate handling of persistence-based tasks in high-speed data streams. Pandora tackles key challenges in data stream analysis by introducing innovative techniques such as item inactivity degree and adaptive replacement strategies. Our theoretical analysis establishes the error bound, and extensive experiments on real-world traces demonstrate Pandora's superior performance across tasks like persistence estimation, persistent item lookup, and identifying persistent items with high frequency. Additionally, we evaluate Pandora's adaptability on frequency-based tasks. Even under tight memory constraints, Pandora significantly outperforms state-of-the-art methods in both accuracy and processing speed. With SIMD optimization, Pandora is particularly effective in high-velocity data environments.

## Acknowledgments

The author sincerely thanks his supervisor, Prof. Paul Patras, for his unwavering support and guidance. Gratitude is also extended to Zukai Li for offering insightful comments, particularly on the formal analysis. Finally, the author appreciates the valuable suggestions provided by the anonymous reviewers. This work was partially supported by Cisco through the Cisco University Research Program Fund (Grant no. 2019-197006).

## References

[1] 2024. Pandora Code. https://github.com/PandoraSketch/Pandora.
[2] Cristina Alcaraz and Javier Lopez. 2022. Digital twin: A comprehensive survey of security threats. *IEEE Communications Surveys & Tutorials* 24, 3 (2022), 1475–1503.
[3] Austin Appleby. 2008. MurmurHash. *URL https://sites. google. com/site/murmurhash* (2008).
[4] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. 2021. Salsa: self-adjusting lean streaming analytics. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 864–875.
[5] Lawrence Benson, Richard Ebeling, and Tilmann Rabl. 2023. Evaluating SIMD Compiler-Intrinsics for Database Systems.. In *VLDB Workshops*.
[6] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 267–280.
[7] CAIDA. 2015. Anonymized Internet Traces 2015. https://catalog.caida.org/dataset/passive_2015_pcap.
[8] CAIDA. 2016. Anonymized Internet Traces 2016. https://catalog.caida.org/dataset/passive_2016_pcap.
[9] CAIDA. 2018. Anonymized Internet Traces 2018. https://catalog.caida.org/dataset/passive_2018_pcap.
[10] CAIDA. 2019. Anonymized Internet Traces 2019. https://catalog.caida.org/dataset/passive_2019_pcap.
[11] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*. Springer, 693–703.
[12] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2004. Finding frequent items in data streams. *Theoretical Computer Science* 312, 1 (2004), 3–15.
[13] Lin Chen, Raphael C-W Phan, Zhili Chen, and Dan Huang. 2022. Persistent items tracking in large data streams based on adaptive sampling. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 1948–1957.
[14] 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.
[15] Ziling Chen, Haoquan Guan, Shaoxu Song, Xiangdong Huang, Chen Wang, and Jianmin Wang. 2024. Determining Exact Quantiles with Randomized Summaries. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.
[16] Shiyu Cheng, Dongsheng Yang, Tong Yang, Haowei Zhang, and Bin Cui. 2020. LTC: a fast algorithm to accurately find significant items in data streams. *IEEE Transactions on Knowledge and Data Engineering* 34, 9 (2020), 4342–4356.
[17] 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.
[18] Haipeng Dai, Muhammad Shahzad, Alex X Liu, and Yuankun Zhong. 2016. Finding persistent items in data streams. *Proceedings of the VLDB Endowment* 10, 4 (2016), 289–300.
[19] Cristian Estan and George Varghese. 2003. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)* 21, 3 (2003), 270–313.

[20] Cristian Estan, George Varghese, and Mike Fisk. 2003. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. 153–166.
[21] Frederic Giroire, Jaideep Chandrashekar, Nina Taft, Eve Schooler, and Dina Papagiannaki. 2009. Exploiting temporal persistence to detect covert botnet channels. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 326–345.
[22] Elena Gribelyuk, Pachara Sawettamalya, Hongxun Wu, and Huacheng Yu. 2024. Simple & Optimal Quantile Sketch: Combining Greenwald-Khanna with Khanna-Greenwald. *Proceedings of the ACM on Management of Data* 2, 2 (2024), 1–25.
[23] S. Gündüz and M.T. Özsu. 2003. A Web Page Prediction Model Based on Click-Stream Tree Representation of User Behavior. In *Proceedings of ACM KDD*.
[24] Jiarui Guo, Yisen Hong, Yuhan Wu, Yunfei Liu, Tong Yang, and Bin Cui. 2023. Sketchpolymer: Estimate per-item tail quantile using one sketch. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 590–601.
[25] He Huang, Jiakun Yu, Yang Du, Jia Liu, Haipeng Dai, and Yu-E Sun. 2023. Memory-Efficient and Flexible Detection of Heavy Hitters in High-Speed Networks. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–24.
[26] Jiawei Huang, Wenlu Zhang, Yijun Li, Lin Li, Zhaoyi Li, Jin Ye, and Jianxin Wang. 2022. ChainSketch: An efficient and accurate sketch for heavy flow detection. *IEEE/ACM Transactions on Networking* (2022).
[27] Nicole Immorlica, Kamal Jain, Mohammad Mahdian, and Kunal Talwar. 2005. Click fraud resistant methods for learning click-through rates. In *Internet and Network Economics: First International Workshop, WINE 2005, Hong Kong, China, December 15-17, 2005. Proceedings 1*. Springer, 34–45.
[28] Peng Jia, Pinghui Wang, Junzhou Zhao, Ye Yuan, Jing Tao, and Xiaohong Guan. 2021. Loglog filter: Filtering cold items within a large range over high speed data streams. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 804–815.
[29] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal quantile approximation in streams. In *2016 ieee 57th annual symposium on foundations of computer science (focs)*. IEEE, 71–78.
[30] Matti Karppa and Rasmus Pagh. 2022. HyperLogLogLog: cardinality estimation with one log more. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 753–761.
[31] Samir Khan and Takehisa Yairi. 2018. A review on the application of deep learning in system health management. *Mechanical Systems and Signal Processing* 107 (2018), 241–265.
[32] Bibudh Lahiri, Jaideep Chandrashekar, and Srikanta Tirthapura. 2011. Space-efficient tracking of persistent items in a massive data stream. In *Proceedings of the 5th ACM international conference on Distributed event-based system*. 255–266.
[33] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. 2020. Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1574–1584.
[34] Weihe Li and Paul Patras. 2023. P-Sketch: A Fast and Accurate Sketch for Persistent Item Lookup. *IEEE/ACM Transactions on Networking* (2023).
[35] Weihe Li and Paul Patras. 2024. Stable-Sketch: A Versatile Sketch for Accurate, Fast, Web-Scale Data Stream Processing. In *Proceedings of the ACM on Web Conference 2024*. 4227–4238.
[36] Yuanpeng Li, Feiyu Wang, Xiang Yu, Yilong Yang, Kaicheng Yang, Tong Yang, Zhuo Ma, Bin Cui, and Steve Uhlig. 2023. Ladderfilter: Filtering infrequent items with small memory and time overhead. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–21.
[37] Ee-Peng Lim, Hsinchun Chen, and Guoqing Chen. 2013. Business intelligence and analytics: Research directions. *ACM Transactions on Management Information Systems (TMIS)* 3, 4 (2013), 1–10.
[38] Chris Lomont. 2011. Introduction to intel advanced vector extensions. *Intel white paper* 23 (2011), 1–21.
[39] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. 2018. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1912–1949.
[40] Tianyu Ma, Guoju Gao, He Huang, Yu-E Sun, and Yang Du. 2024. Scout Sketch: Finding promising items in data streams. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 1561–1570.
[41] MAWI. 2020. MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/
[42] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory*. Springer, 398–412.
[43] Gaurav Mitra, Beau Johnston, Alistair P Rendell, Eric McCreath, and Jun Zhou. 2013. Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 1107–1116.
[44] Vijay S Mookerjee and Yong Tan. 2002. Analysis of a least recently used cache management policy for web browsers. *Operations Research* 50, 2 (2002), 345–357.

[45] Shishir Nagaraja and Ryan Shah. 2019. Clicktok: Click fraud detection using traffic analysis. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*. 105–116.

[46] Nikos Ntarmos, Peter Triantafillou, and Gerhard Weikum. 2009. Distributed hash sketches: Scalable, efficient, and accurate cardinality estimation for distributed multisets. *ACM Transactions on Computer Systems (TOCS)* 27, 1 (2009), 1–53.

[47] Krishna Chaitanya Pabbuleti, Deepak Hanamant Mane, Avinash Desai, Curt Albert, and Patrick Schaumont. 2013. SIMD acceleration of modular arithmetic on contemporary embedded platforms. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[48] Siyuan Sheng, Qun Huang, Sa Wang, and Yungang Bao. 2021. PR-sketch: monitoring per-key aggregation of streaming data with nearly full accuracy. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1783–1796.

[49] 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. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2026–2034.

[50] Daniel Ting. 2016. Towards optimal cardinality estimation of unions and intersections with sketches. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1195–1204.

[51] Daniel Ting. 2018. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*. 1129–1140.

[52] Russ Unger and Carolyn Chandler. 2023. *A Project Guide to UX Design: For user experience designers in the field or in the making*. New Riders.

[53] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. 2022. Potential of webassembly for embedded systems. In *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 1–4.

[54] Qingjun Xiao, Yan Qiao, Mo Zhen, and Shigang Chen. 2014. Estimating the persistent spreads in high-speed networks. In *2014 IEEE 22nd International Conference on Network Protocols*. IEEE, 131–142.

[55] Hui Yang, Yamei Luo, Xiaolei Ren, Ming Wu, Xiaolin He, Bowen Peng, Kejun Deng, Dan Yan, Hua Tang, and Hao Lin. 2021. Risk prediction of diabetes: big data mining with fusion of multifarious physical examination indicators. *Information Fusion* 75 (2021), 140–149.

[56] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. 2018. Heavyguardian: Separate and guard hot items in data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2584–2593.

[57] 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 561–575.

[58] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: an accurate algorithm for finding Top-$k$ elephant flows. *IEEE/ACM Transactions on Networking* 27, 5 (2019), 1845–1858.

[59] Tong Yang, Haowei Zhang, Dongsheng Yang, Yucheng Huang, and Xiaoming Li. 2019. Finding significant items in data streams. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1394–1405.

[60] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. 2017. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1442–1453.

[61] Jin Ye, Lin Li, Wenlu Zhang, Guihao Chen, Yuanchao Shan, Yijun Li, Weihe Li, and Jiawei Huang. 2022. UA-Sketch: An Accurate Approach to Detect Heavy Flow based on Uninterrupted Arrival. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[62] Yinda Zhang, Jinyang Li, Yutian Lei, Tong Yang, Zhetao Li, Gong Zhang, and Bin Cui. 2020. On-off sketch: A fast and accurate sketch on persistence. *Proceedings of the VLDB Endowment* 14, 2 (2020), 128–140.

[63] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. 2021. CocoSketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 207–222.

[64] Zheng Zhong, Shen Yan, Zikun Li, Decheng Tan, Tong Yang, and Bin Cui. 2021. Burstsketch: Finding bursts in data streams. In *Proceedings of the 2021 International Conference on Management of Data*. 2375–2383.

[65] 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 *Proceedings of the 2018 International Conference on Management of Data*. 741–756.

Received July 2024; revised September 2024; accepted November 2024