Computer Networks xxx (xxxx) xxx

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Finding needles in a hay stream: On persistent item lookup in data streams

Lin Chen [a,*], Haipeng Dai [b], Lei Meng [b], Jihong Yu [c]

[a] School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China
[b] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[c] School of Information and Electronics, Beijing Institute of Technology, Beijing, China

ARTICLE INFO

Keywords:
Persistent item lookup
Data stream mining

ABSTRACT

In a data stream composed of an ordered sequence of data items, *persistent items* refer to those persisting to occur over a long timespan. Compared with ordinary items, persistent ones, though not necessarily occurring more frequently, typically convey more valuable information. *Persistent item lookup*, the functionality to identify all persistent items, emerges as a pivotal building block in many computing and network systems. In this paper, we devise a generic persistent item lookup algorithm supporting high-speed, high-accuracy lookup with limited memory cost. The key technicalities we propose in our design are two-fold. First, our algorithm attempts to record only persistent items seen so far based on the currently available information about the stream, thus significantly reducing memory overhead, especially for real-life highly skewed data streams. Second, our algorithm balances the recording load in both time and space domains: in the time domain, we partition persistent items into approximately equal-size subsets and record only one subset in each epoch; in the space domain, we apply the state-of-the-art load balancing technique to evenly distribute recorded items across the on-die memory. By holistically integrating these components, we iron out a persistent item lookup algorithm outperforming existing solutions in a wide range of practical settings.

1. Introduction

1.1. Background and motivation

In a data stream composed of an ordered sequence of data items, *persistent items* refer to those persisting to occur in the stream over a long timespan. Compared with ordinary items, persistent ones, though not necessarily occurring more frequently, typically convey more valuable information on the intrinsic data pattern about the stream. Examples can be found in a variety of computing and network applications, including but not limited to:

- **Traffic mining** [1], where persistent traffic, e.g., vehicles passing the same locations periodically, can help understand the traffic pattern in a given region with the vehicles being the monitored items at each location;
- **Click fraud detection** [2], where automatic robots are configured to persistently click on advertisements to increase the advertiser's revenue in pay-per-click online advertising systems; therefore, persistent cliques need to be detected;
- **Stealthy distributed denial of service (DDoS) attack detection** [3], where attackers may inject malicious packets at a limited rate but over a long period of time, rather than to overwhelm the victim, to avoid being detected quickly by the intrusion detection system (IDS); hence, packets persistently sent from the same source needs to be monitored.

To enable further fine-grained persistent item mining and analysis, a bootstrapping functionality is to quickly and accurately identify and record all the persistent items in the stream. This is by no means a trivial task. On the one hand, in the emerging large-scale data stream mining applications such as internet traffic analysis [4], graph stream mining [5], sensor data mining [6], web crawling [7], and natural language processing [8], the massive data to be processed are often organized as high-speed streams, requiring monitoring points[1] to record, process and extract stream information in real time. On the other hand, the limited memory at the monitoring point, especially the on-die static random-access memory (SRAM), urges us to implement this functionality as compact as possible by keeping the memory usage at

* Corresponding author.
E-mail addresses: chenlin69@mail.sysu.edu.cn (L. Chen), haipengdai@nju.edu.cn (H. Dai), menglei5633@163.com (L. Meng), jihong-yu@hotmail.com (J. Yu).

[1] Throughout the paper, we use the generic term *monitoring point*, or observation point, to denote any computing device that processes and records information about items in the data stream. Examples include end hosts, firewalls and proxies, network middleboxes such as routers and switches.

https://doi.org/10.1016/j.comnet.2020.107518
Received 20 April 2020; Received in revised form 3 July 2020; Accepted 26 August 2020
Available online 3 September 2020

the lowest level, while still guaranteeing the required lookup accuracy. These two quests – high-speed, high-accuracy operation and limited memory usage – are often at odds with each other, since optimizing for one may come at the expense of the other.

1.2. Problem formulation

Motivated by above observation, we embark in this paper on a generic study of persistent item lookup. By *generic* we mean that no specific problem or context is assumed for the analysis; both the probability distribution of data items and their order of occurrence in the stream can be arbitrary and not known *a priori*. The only important assumption is that each item is characterized by a globally unique item ID, whose hashprint is uniformly distributed on the corresponding domain.

Formally, consider a time interval during which we need to analyze persistent items in a data stream. We divide the whole time interval into T epochs, index from 0 to $T-1$. A data item e is called persistent if it occurs in all the T epochs, where *occurrence* refers to an event that an item appears at least once within the considered time interval.[2] We seek a persistent item lookup algorithm that meets the following requirements:

Lookup accuracy. Both the *false negative rate*, the probability of a persistent item not being identified and recorded, and the *false positive rate*, the probability of a non-persistent item being marked as persistent, should be strictly upper-bounded by the target rates specified by the user.

Lookup efficiency. Given the ID of any item e, the algorithm should respond in time sub-linear or constant to the number of persistent items whether e is persistent or not, and if yes, return the information related to e; the algorithm should be able to reconstruct the entire list of persistent items in time linear to the number of persistent items.

Memory efficiency. The algorithm should incur limited memory overhead at the monitoring point, especially in terms of the on-die memory; ideally, the memory overhead should asymptotically scale to the number of persistent items rather than the stream cardinality.

We emphasize that these requirements should be addressed holistically in the persistent item lookup algorithm.

1.3. Main contributions and technicalities

We devise a persistent item lookup algorithm meeting the above requirements. The key technicalities we propose in our design are twofold. First, our algorithm attempts to record only persistent items seen so far based on the currently available information about the stream. Compared to the common wisdom adopted in the literature where all the items need to be recorded, our algorithm significantly reduces memory overhead, especially noticing that real-life data streams exhibit high item skewness such that persistent items are in practice only a small portion of the entire stream. Secondly, we balance the recording load in both time and space domains: in the time domain, we partition persistent items into approximately equal-size subsets and record only one subset each epoch; in the space domain, we apply the state-of-the-art load balancing technique to evenly distribute the recorded items across the on-die memory. By holistically integrating these components, we iron out a persistent item lookup algorithm that outperforms existing solutions in a wide range of practical settings, as demonstrated in the numerical and empirical analysis.

[2] Mathematically, a stream can be regarded as a *multiset* of items, each of which may appear multiple times.

1.4. Paper organization

The rest of the paper is structured as follows. Section 2 summarizes related work and analyzes their limitation. Section 3 provides an overview of our algorithm by presenting the key technicalities used in our design. Section 4 describes our algorithm in detail and Section 5 gives a mathematical analysis on our algorithm by establishing the false negative and the false positive rates, deriving the time complexity and discussing parameter tuning and optimization. Section 6 investigates a number of variants and extensions related to the persistent item lookup problem. Section 7 presents numerical and empirical study to evaluate our algorithm. Finally, Section 8 concludes the paper with a discussion of our methodology and results.

2. Related work and limitation

While significant research efforts have been devoted to data stream mining, persistent item mining has surprisingly attracted far less attention, despite its fundamental and practical importance. For example, in the stealthy DDoS attack mentioned in the Introduction, the attacker deliberately spread the malicious packets over a long period of time to avoid being detected by the standard volume-based detectors. In persistent traffic monitoring, persistent vehicles demonstrate the volume of minimal stable traffic at a location, as the transient traffic varies over time. In these applications, algorithms that are capable of detecting persistent items are called for, which cannot draw upon the standard algorithms for frequent item detection. To the best of our knowledge, the algorithm proposed in [9], termed as Persistent items Identification schemE (PIE), is the state-of-the-art proposition in the literature that is able to record, identify persistent items and reconstruct the list of persistent items. The key innovation in PIE consists of a compact hash-based data structure and a Raptor code-based approach to efficiently record item IDs. Specifically, PIE uses Raptor codes to encode the ID of each item and to stores only part of the encoded ID during each epoch, thus boosting the memory efficiency. Despite its elegant design, PIE still follows the common wisdom to record all the data items, from which persistent ones are extracted for further processing. Motivated by this limitation, we demonstrate in this paper that it is not necessary to record all the items in the stream. In this regard, we develop an algorithm that records only persistent items seen so far based on the currently available information about the stream. Our algorithm is particularly adapted for streams exhibiting high item skewness, a phenomena largely observed in real-life datasets today.

Another persistent item mining and tracking algorithm is proposed in [10]. The idea is to configure a filter, through which each item passes and gets selected by the filter with a certain probability. If an item is selected, its persistence is tracked. The filter is configured such that the probability of an item being selected increases if it reappears in different time periods, while remains the same if it reappears in the same period. The selected items are recorded, among which persistent ones are identified. Despite the efforts in recording only potential persistent items, the algorithm may still record too many non-persistent ones as once selected, an item remains in the filter even if it does not reappear in later periods.

There are a handful of propositions addressing the problem of persistent item counting in various applications, which is to count the number of persistent items [1,3,11]. Despite its focus on persistent items, the persistent item counting problem is by nature different from the persistent item lookup problem. In persistent item counting, the objective is to produce an estimate of the number of persistent items without the need to either record their information or reconstruct the whole list of persistent items. Therefore, highly compact counters, such as the HyperLogLog (HLL) counters [12], are usually employed to produce sufficiently accurate estimates with low memory overhead. In contrast, persistent item lookup not only reveals the number of persistent items, but should also be able to record information concerning

Table 1
Main notations.

N	Number of distinct items arrived in each epoch
N_p	Number of persistent items
T	Number of epochs
t	Current epoch
$\overline{\mathcal{N}}(t)$	Set of non-persistent items misidentified as persistent items at epoch t
$N^*(t)$	Number of items recorded in epoch t
e	ID of the item to be processed
$\mathcal{U}$	Set of ID universe
B	Bloom filter tracing persistent items
b	Bloom filter tracing items occurred in epoch t
L	Length of B and b
w	Number of item subsets
$g(\cdot)$	Hash function dividing IDs into w subsets
m	Number of buckets in the SRAM
c	Number of cells in each bucket
R	Memory space to stock an item
d	Number of candidate buckets for an item
$h_j(\cdot)$	k hash functions casting an item ID to k bits in B and b
$h'_i(\cdot)$	d hash functions providing d candidate buckets
$f(\cdot)$	Item spread distribution
β_l	Fraction of bins with $\geq l$ balls in the bin-and-ball problem
ϕ_d	Exponent of generalized d-order Fibonacci sequence
P_{FN}	False negative rate
P_{FP}	False positive rate

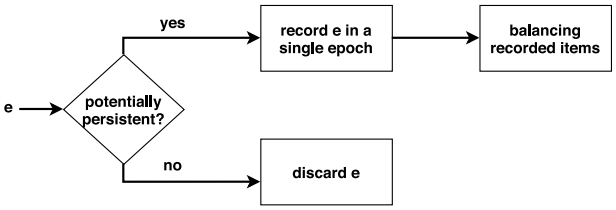

Fig. 1. Illustration of techniques used in our algorithm.

each persistent item, support item search, and return the entire list of the persistent items.

In a broader context, the problems of item frequency estimation and frequent item counting and heavy hitter detection in data streams are classical problems in streaming algorithms [13–15]. Their solutions can be divided into two categories by employing sketches and counters. We refer readers to [13] for a comprehensive survey. Any persistent item is an item with frequency at least $1/T$. In this sense, persistent items can be regarded as a particular subset of frequent items with an additional property of occurring at least once per epoch. However, algorithms mining frequent items cannot be directly used for mining persistent items because frequent item mining schemes count the frequencies of items without taking the temporal dimension into account. Consequently, the problem of persistent item lookup requires an original analysis and design that cannot draw on existing approaches.

3. Design rationale and overview

In this section, we illustrate the three key techniques of our algorithm, as shown in Fig. 1. Though they can be applied independently one to another, their holistic integration can further enable synergy bringing significant performance gain. A formal description of our algorithm is presented in Section 4. Table 1 lists the major notations used throughout the paper.

3.1. Technique 1: recording only persistent items

As analyzed in Introduction, a common approach used in the literature consists of compactly recording the information of all the items occurred in the current epoch in the SRAM. From the perspective of mining persistent items, recording all items is clearly highly inefficient. Therefore, we propose to record only persistent items. To illustrate our idea, consider a data stream composed of 6 items e_i $(1 \leq i \leq 6)$; the time horizon is divided to 4 epochs; for each epoch we have 4 arriving items enumerated as below:

- Items occurred in epoch 0: e_1, e_2, e_3, e_4,
- Items occurred in epoch 1: e_1, e_2, e_3, e_5,
- Items occurred in epoch 2: e_1, e_2, e_4, e_6,
- Items occurred in epoch 3: e_1, e_2, e_5, e_6.

In the state-of-the-art approach, we need to record 4 items in each epoch in the on-die memory and in total 16 items in the off-die memory, if we dump the recorded items off-die after each epoch. However, only e_1 and e_2 are persistent. In our algorithm, we use a Bloom filter to trace the persistent items (cf. Section 4 for details) and record only persistent ones (e_1 and e_2) in the last epoch.

3.2. Technique 2: distributing recording load

Our second technique to further reduce the on-die memory cost is to distribute the recording task across multiple epochs. The core idea is to only record a subset of persistent items in a single epoch. Consider again the example above. We balance the recording task by recording only e_1 in epoch 2 and e_2 in epoch 3, thus further halving the memory cost in this regard.

Technically, we partition the entire item set into w approximately equal-size subsets by applying a hash-based partition scheme on the item ID, which is particularly memory and time-efficient. We then record only the persistent items belonging to a single subset in each of the last w epochs. The recording task is performed during the last w epochs so as to avoid recording too many non-persistent items. For a given epoch t $(T-w \leq t < T)$, the recorded items may still contain non-persistent ones because some items may not reappear in a later epoch. To mitigate such false positive, we maintain a Bloom filter recording all the persistent items from the beginning to the current epoch. In the lookup phase, an item is identified as persistent if it passes the Bloom filter membership test.

Tuning w allows to trade off between limiting the number of recorded non-persistent items, on the one hand, and expanding the recording task across more epochs, on the other. A smaller w reduces the number of recorded non-persistent items with an extreme example $w = 1$ minimizing the false positive rate, while increases the number of items recorded in a single epoch. Section 5.4 studies the parameterization of w by quantifying this trade-off.

3.3. Technique 3: giving an item more choices

Our third technique seeks to maximize the recording efficiency within a single epoch by balancing recorded items to achieve more compact memory usage.

Consider a standard setting that we need to record n items in the SRAM organized as n buckets; each bucket is further divided into c cells, each able to hold one item. In the baseline approach, each item is hashed into a random bucket based on its ID and can be successfully recorded if the bucket has an empty cell. This approach leads to unbalanced item distribution across buckets. Mathematically, the maximum loaded bucket has $\Theta\left(\frac{\log n}{\log \log n}\right)$ items hashed into it w.h.p. [16].[3]

To mitigate this unbalance, we apply the load-balancing technique by hashing an item to d buckets and placing it into the one with minimum load. This mathematically reduces the maximum load to

[3] Throughout the paper, we use the following asymptotic notations:
- $g_1(x) = O(g_2(x)) \iff \exists c > 0,\ \exists x_0,\ \forall x > x_0,\ |g_1(x)| \leq c|g_2(x)|$;
- $g_1(x) = \Omega(g_2(x)) \iff \exists c > 0,\ \exists x_0,\ \forall x > x_0,\ |g_2(x)| \leq c|g_1(x)|$;
- $g_1(x) = \Theta(g_2(x)) \iff g_1(x) = O(g_2(x))$ and $g_1(x) = \Omega(g_2(x))$;
- $g_1(x) = o(g_2(x)) \iff \lim_{x\to\infty} \frac{g_1(x)}{g_2(x)} = 0$.

$\Theta\left(\frac{\log\log n}{\log d}\right)$ [17]. We further apply the *always-go-left policy* [18] by breaking the tie by prioritizing the leftmost bucket, which reduces the maximum load to $\Theta\left(\frac{\log\log n}{d\log\phi_d}\right)$, with ϕ_d corresponding to the exponent of growth for a generalized Fibonacci sequence.[4] This reduces the memory size to $\Theta\left(\frac{n\log\log n}{d\log\phi_d}\right)$ for achieving negligible overflow probability, as proved in Section 5.1.

To further mitigate recording failure, we record the overflowed items in a ternary content addressable memory (TCAM, cf. Section 4.1 for a detailed specification). We demonstrate that a small TCAM is sufficient to hold all overflowed items.

We note that some of the building blocks in our design are inspired by or readily follow from the existing literature, to which we do not acclaim credit. Specifically, the always-go-left policy, or more generically the 1-out-of-d policy [18,19] is a well-known load-balancing policy that has been applied in many diverse engineering problems and data structures including the well-known Cuckoo hashing [20]. The idea of using a stash to record overflowed items is proposed in [21] for Cuckoo hashing. Our technical contribution consists of building a holistic persistent item lookup framework over these building blocks that can achieve significant performance gain, as demonstrated by both theoretical and empirical analysis.

4. Algorithm

Our algorithm consists of two subroutines, Recording and Lookup, which we expose and analyze in the sequel. For notation conciseness, we denote $[a] := [0, a-1] \cap \mathbb{Z}_{\geq 0}$, i.e., the set of non-negative integers smaller than a; we denote $\mathbf{0}$ ($\mathbf{1}$, respectively) the L-bit vector with each bit being 0 (1).

4.1. Memory and data structure

Our algorithm uses 3 types of memories.

Static Random-Access Memory (SRAM) [22]. The SRAM is volatile semi-conductor memory that does not need to be periodically refreshed. Usually implemented on-die with up to 72 Mb, it can transfer up to 4 words per access with 9–36 bit word size. Given its high access facility but limited size, the SRAM is used in our algorithm to support online recording of persistent items in the current epoch.

Dynamic Random-Access Memory (DRAM) [22]. The DRAM is volatile memory requiring to be refreshed periodically. Typically implemented off-die with up to 4 Gb, it can read up to 8 words per access with 4–16 bit word size. Given its low access facility but larger size compared to SRAM, DRAM is used to hold the contents dumped from the SRAM at the end of each epoch.

Ternary Content Addressable Memory (TCAM) [23]. The TCAM is small on-die memory supporting very fast access. Typically sized a few Mbits, it locates an entry by comparing it against all entries in a single clock cycle. Given its low access delay but very limited size, the TCAM is used in our algorithm to record the few overflowed items that cannot be recorded in either the DRAM or SRAM.

In our work, the on-die SRAM is organized into m buckets. Each bucket is further divided into c cells, each able to hold a single item. This is the standard SRAM cache organization today, thus making our algorithm hardware-friendly. We denote R the memory space to stock an item, e.g., in network flow measurement, the minimum information to be recorded for a single flow typically includes its flow ID, the IP addresses and port number of source and destination nodes. The total on-die memory cost sums up to mcR.

The data structures used in our algorithm are summarized below.

Bloom filters. We maintain 2 Bloom filters B and b of equal length L. B records the presence of all the items appearing in every epoch from epoch 0 to the epoch preceding the current one. b records the presence of the items appearing in the current epoch. B is initialized to $\mathbf{1}$ at the beginning of the whole measurement process. b is reset to $\mathbf{0}$ at the beginning of each epoch.

Hash functions. We use k hash functions associated with the Bloom filters, denoted by $\{h_j(\cdot)\}_k^{j=1}$, where $h_j(\cdot) : \mathcal{U} \to [L]$ with $\mathcal{U}$ denoting the item ID universe. We use another hash function $g(\cdot) : \mathcal{U} \to [w]$ to partition the items into w approximately equal-size subsets. Thirdly, we uses d hash functions $\{h'_l(\cdot)\}_d^{l=1}$ to choose d candidate buckets for each item to be recorded. We split the SRAM into d groups of buckets, each containing $\frac{m}{d}$ buckets. Each hash function $h'_l(\cdot)$ $(1 \leq l \leq d)$ returns a random bucket in group l, thus giving in total d candidate buckets to record each item.

To illustrate the above data structures, consider an item e to be recorded. $\{h_j(\cdot)\}_k^{j=1}$ are used to check in B whether e is potentially persistent. In case of yes, $g(e)$ is used to determine the epoch to record e, and $\{h'_l(\cdot)\}_d^{l=1}$ is used to determine the bucket to record e.

We note that as the hash functions used in our algorithm all use item ID as their seed, we can apply the *one hashing technique* proposed in [24] to reduce the computation overhead related to hash computations. The core idea is to split the output of a hash function into several segments to emulate multiple hash functions, thus reducing the hash computation.

4.2. Recording

The Recording subroutine, invoked upon arrival of each item e, is composed of three steps.

Step 1: pre-processing (updating b)

We first check if e appears in each epoch before the current epoch t by checking if there exists j such that $B(h_j(e)) == 0$.[5] If yes, we drop e because it is not a persistent item by definition. Otherwise, we further check if e has already been processed by checking if $b(h_j(e)) == 1, \forall 1 \leq j \leq k$ and drop it if already processed.

If e is a persistent item and has not been processed, we set $b(h_j(e)) = 1,\ 1 \leq j \leq k$. We then check if e should be recorded in the current epoch by checking whether $g(e) == T - t$. Recall that we record a subset of items in each of the last w epochs. This test checks whether e should be recorded in the current epoch t. If yes, we invoke Step 2 to record it, otherwise we drop it.

The pseudo-code of our algorithm is shown in Algorithm 1. The procedure Pre-processing returns 1 if e needs to be recorded and the procedure Insert is further invoked.

Step 2: recording (inserting e in SRAM)

When Step 2 is invoked, we record e in the SRAM. To make the recording space-efficient, we balance the load across buckets by offering e multiple choices and putting it into the least loaded bucket. Technically, we choose d candidate buckets indexed by $h'_l(e), 1 \leq l \leq d$ and put e in the bucket with minimum load. In case of tie, we choose the left-most least loaded bucket to record e.

If all the d chosen buckets are full, we offer a last chance by recording e in the TCAM.

Step 3: post-processing (updating B)

At the end of each epoch, the SRAM containing the recorded item IDs including the stash is dumped to the off-die DRAM for future lookup. We update the Bloom filter maintaining the current persistent item list $B \leftarrow B \wedge b$ where $\wedge$ denotes the bitwise And operation. Recall that B traces the persistent items up to the previous epoch and b traces the items appeared in the current epoch. It follows straightforwardly that $B \wedge b$ traces the persistent items appeared in all epochs including the current one. By tracing we mean that we can check if an item is present by checking the corresponding bits in the Bloom filter.

[4] Algebraically, it holds that $\phi_d > 1.61$ and $\phi_d^d = \sum_{i=0}^{d-1} \phi_d^i = \frac{\phi_d^d - 1}{\phi_d - 1}$.

[5] By slightly abusing notation, we use e to denote the ID of item e.

At the end of the whole measurement, i.e., T epochs, the Bloom filter B is dumped to the DRAM for future lookup. The RECORDING subroutine is thus completed.

Algorithm 1 Recording subroutine

```
1: Initialization: B ← 1, b ← 0
2: procedure RECORDING(e)                          ▷ invoked on arrival of e
3:     if PRE-PROCESSING(e) == 1 then
4:         INSERT(e)
5:     end if
6:     POST-PROCESSING
7: end procedure
8: function PRE-PROCESSING(e)
9:     if there exists j such that B(h_j(e)) == 0 then
10:        return 0                                ▷ e is not a persistent item
11:    else if b(h_j(e)) == 1, ∀1 ≤ j ≤ k then
12:        return 0                                ▷ e has already been processed
13:    else if g(e) == T − t then
14:        b(h_j(e)) ← 1, 1 ≤ j ≤ k
15:        return 1   ▷ e is a persistent item and has not been processed
   and it belongs to the set of items to be recorded in the current epoch
16:    else
17:        b(h_j(e)) ← 1, 1 ≤ j ≤ k
18:        return 0   ▷ e is a persistent item and has not been processed, but it does not belong to the set of items to be recorded in the current epoch
19:    end if
20: end function
21: procedure INSERT(i)
22:    Choose d buckets indexed h'_l(e), 1 ≤ l ≤ d
23:    if there is at least one empty cell then
24:        Insert e to the leftmost least loaded bucket
25:    else if there is empty cell in TCAM then
26:        Insert e in TCAM
27:    else
28:        return insertion failure
29:    end if
30: end procedure
31: procedure POST-PROCESSING                      ▷ after each epoch
32:    Dump SRAM and TCAM off-die to DRAM
33:    Empty SRAM and TCAM
34:    Update B: B ← B ∧ b
35:    Set b ← 0 for the next epoch
36:    if t == T − 1 then                          ▷ at the end of measurement
37:        Dump B off-die to DRAM
38:    end if
39: end procedure
```

4.3. Lookup

The LOOKUP subroutine supports two operations, persistent item lookup and persistent item list reconstruction.

Persistent item search takes the item ID as input and returns the recorded item information if it is a persistent item. To this end, given an item ID e, we first check the Bloom filter B whether e is a persistent item. If yes, we find the epoch in which e is supposed to be recorded, i.e., epoch $g(e)$. We then search in the d candidate buckets corresponding and returns the information of e. If no information of e is found and the d buckets are all full, the algorithm then searches the TCAM. We return lookup failure if e cannot be found in either one of the d buckets or the TCAM.

Persistent item list reconstruction checks the items recorded in the DRAM and returns the list of every item e with related information if it passes the Bloom filter test, i.e., $B(h_j(e)) = 1, \forall 1 \le j \le k$.

The pseudo-code of LOOKUP is straightforward and thus omitted here.

5. Analysis

As mentioned in Introduction, the lookup accuracy of any persistent item lookup algorithm is quantified by the *false negative rate*, the probability of a persistent item not being recorded, and the *false positive rate*, the probability of a non-persistent item being marked as persistent. In this section, we first derive these metrics for our algorithm, then derive its complexity and study how to tune various parameters to optimize the performance and balance among lookup accuracy, memory efficiency, and computation overhead.

5.1. False negative rate

We give an overview on how to compute the false negative rate. Note that a persistent item fails to be recorded if its d candidate buckets and the TCAM are full. We call the event that an item fails to be recorded in the SRAM (as also the DRAM accordingly) as an *overflow*. We calculate the overflow probability by three steps.

Step 1: deriving the number of non-persistent items marked as persistent

Let $\overline{\mathcal{N}}(t)$ denote the set of non-persistent items misidentified as persistent items at epoch t whose cardinality is denoted as $\overline{N}(t)$. To compute $\overline{N}(t)$ $(T - w \le t < T)$, recall Algorithm 1, we notice that if an item is marked as persistent in epoch t, it must also be regarded as persistent for all epochs preceding t; this leads to $\overline{\mathcal{N}}(T-1) \subseteq \cdots \subseteq \overline{\mathcal{N}}(T-w)$. It then holds that

$$\overline{N}(t) \le \overline{N}(T-w), \quad T - w \le t < T. \tag{1}$$

We next derive $\overline{N}(T-w)$ as an upper-bound of $\overline{N}(t)$. Given any item e, we define p as the probability that the k bits $h_j(e)$ $(1 \le j \le k)$ in the Bloom filter b are set to 1 at the end of any epoch. Following standard Bloom filter analysis, by setting $k = \frac{L}{N} \ln 2$, we have:

$$p = \left(1 - \left(1 - \frac{1}{L}\right)^{kN}\right)^k = \left(1 - e^{-\frac{kN}{L}}\right)^k = 2^{-k}. \tag{2}$$

We define the *spread* as the number of epochs in which an item appears. Persistent items have spread T. Let $f(\cdot)$ denote the item spread distribution with $f(s)$ denoting the probability of an item having spread s $(1 \le s \le T)$. For any non-persistent item e with spread s to be marked as persistent in the epoch $T - w$, e needs to be marked as present in at least $T - w - s$ epochs if $s < T - w$. Generically we have:

$$\begin{aligned}\mathbf{Pr}(e \in \overline{\mathcal{N}}(T-w)) &\le p^{T-w-\min(s,T-w)} \\ &= 2^{-k(T-w-\min(s,T-w))}.\end{aligned} \tag{3}$$

Recall (1), we can derive the expected number of non-persistent items marked as persistent in epoch $T - w \le t \le T - 1$ as

$$\begin{aligned}&\mathbb{E}[\overline{N}(t)] \le \mathbb{E}[\overline{N}(T-w)] \\ &= \sum_e \mathbf{Pr}(e \in \overline{\mathcal{N}}(T-w)) \le N \left[\sum_{s=1}^{T-w-1} 2^{-k(T-w-s)} \cdot f(s) + \sum_{s=T-w}^{T-1} f(s)\right].\end{aligned} \tag{4}$$

Note that in practice $f(s)$ is non-zero only for small s, by choosing reasonable w, we can make $\mathbb{E}[\overline{N}(t)]$ vanish quickly.

Step 2: computing the number of recorded items

Denote $N^*(t)$ the number of items recorded in epoch t $(T - w \le t < T)$. These items include persistent items and non-persistent ones marked as persistent. Recall that our algorithm balances the recording task among w epochs. We have

$$\mathbb{E}[N^*(t)] = \frac{\mathbb{E}[\overline{N}(t)] + N_p}{w}. \tag{5}$$

By the multiplicative Chernoff bound [25], $\forall \delta > 0$ we have,

$$\mathbf{Pr}\left(N^*(t) \le (1+\delta)\mathbb{E}[N^*(t)]\right) \ge 1 - \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mathbb{E}[N^*(t)]}.$$

Injecting $\delta = e - 1$ into the above inequality, we have that $N^*(t)$ is upper-bounded by $e\mathbb{E}[N^*(t)]$ w.h.p. asymptotically:

$$\mathbf{Pr}\left(N^*(t) \le e\mathbb{E}[N^*(t)]\right) \ge 1 - e^{-\mathbb{E}[N^*(t)]}. \tag{6}$$

Step 3: deriving the overflow rate

Consider a standard balls-and-bins setting with n balls randomly thrown into n bins. Under the *always-go-left* load balancing policy applied in our algorithm, it is shown in [18] that β_l, the fraction of bins with at least l balls, is upper-bounded by $a^{\phi_d^{(l-5)d+3}}$ w.h.p.,[6] where $a \in (0,1)$ is a constant.

Applying the above results by setting the number of buckets $m = \mathbb{E}[N^*(t)]$, we can upper-bound the number of overflowed items at epoch t, denoted by $\Delta N(t)$, w.h.p. as below:

$$\frac{\Delta N(t)}{N^*(t)} < \sum_{l=c+1}^{+\infty} l\cdot(\beta_l - \beta_{l+1}) = \sum_{l=c+1}^{+\infty} l\beta_l - \sum_{l=c+2}^{+\infty}(l-1)\beta_l$$
$$= c\beta_{c+1} + \sum_{l=c+1}^{+\infty}\beta_l.$$

We further bound $\sum_{l=c+1}^{+\infty}\beta_l$ as

$$\sum_{l=c+1}^{+\infty}\beta_l = \sum_{l=c+1}^{+\infty} a^{\phi_d^{(l-5)d+3}} < \sum_{l=c+1}^{+\infty} a^{\phi_d^{(l-7)d}} = \sum_{l=c-6}^{+\infty} a^{\phi_d^{ld}}$$
$$< \int_{c-7}^{+\infty} a^{\phi_d^{d\cdot x}}\,dx = \int_{d(c-7)}^{+\infty} a^{\phi_d^{x}}\,dx.$$

The first inequality follows from $d \ge 2$; the second inequality follows from $a^{\phi_d^{ld}} < \int_{l-1}^{l} a^{\phi_d^{d\cdot x}}\,dx$. To compute $\int_{d(c-7)}^{+\infty} a^{\phi_d^{x}}\,dx$, we substitute $u = \phi_d^x$ and get

$$\int_{d(c-7)}^{+\infty} a^{\phi_d^{x}}\,dx = \frac{1}{\ln\phi_d}\int_{\phi_d^{d(c-7)}}^{+\infty}\frac{a^u}{u}\,du < \frac{\int_{\phi_d^{d(c-7)}}^{+\infty} a^u\,du}{\phi_d^{d(c-7)}\ln\phi_d}$$
$$= \frac{a^{\phi_d^{d(c-7)}}}{\phi_d^{d(c-7)}\ln a\ln\phi_d}.$$

It then holds that if $a^{\phi_d^{cd}} = \Theta\left(\mathbb{E}[N^*(t)]^{-(1+\epsilon)}\right)$ for any $\epsilon > 0$, i.e., $c = \Theta\left(\frac{\ln\ln\mathbb{E}[N^*(t)]+\ln(1+\epsilon)}{d\ln\phi_d}\right)$, it follows from (6) that the false negative rate, denoted by P_{FN}, is upper-bounded by the overflow rate as below:

$$P_{FN} \le \frac{\Delta N(t)}{N^*(t)} = O\left(\frac{\ln\ln\mathbb{E}[N^*(t)] + \ln(1+\epsilon)}{\mathbb{E}[N^*(t)]^{\epsilon}}\right) \tag{7}$$

with probability at least $\ge 1 - O\left(e^{-\mathbb{E}[N^*(t)]}\right)$. Note that a persistent item fails to be recorded if the d candidate buckets and the TCAM are full. $\frac{\Delta N(t)}{N^*(t)}$ thus gives the upper-bound of the false negative rate without TCAM. If we further deploy a TCAM of size $\Theta\left(\frac{\ln\ln\mathbb{E}[N^*(t)]+\ln(1+\epsilon)}{\mathbb{E}[N^*(t)]^{\epsilon-1}}\right)$, we can further ensure that no false negative occurs w.h.p., which implies that our algorithm has virtually only one-sided error, a desirable property because false negatives are usually much more critical than false positives.

Case study

Our previous analysis establishes P_{FN} for any item spread distribution f. It suffices to apply (4) to derive $\mathbb{E}[\overline{N}(t)]$ then (5) to derive $\mathbb{E}[N^*(t)]$ and finally (7) to compute P_{FN}.

In real-life data streams, item spread typically exhibits high skewness such that a predominately large portion of items appear in a few epochs. We next derive the overflow rate by focusing on two typical spread distributions.

[6] In our context w.h.p. means with prob. at least $1 - n^{-\kappa}$ for any $\kappa > 0$.

Case 1. $f(s)$ is non-zero only for small s: mathematically $f(s) > 0$ when $s \le S$ with $S \ll T$. Algebraically we have:

$$\mathbb{E}[N^*(t)] = O\left(\frac{2^{-k(T-w-S)}N + N_p}{w}\right).$$

Case 2. $f(s)$ is a Zipf function with parameter a and S: mathematically $f(s) = \frac{1}{s^a\sum_{i=1}^{S}(1/i)^a}$ for $s \le S$ and $f(s) = 0$ for $s > S$. Usually S is significantly small compared to T. If $S = o(T)$, this subcase degenerates to Case 1. We consider the subcase where $S = \mu T$ with $\mu < 1$. To bound $\mathbb{E}[\overline{N}(t)]$, noticing (4) and $w \ll T$, we have

$$\sum_{s=1}^{T-w-1} 2^{-k(T-w-s)}f(s) < \sum_{s=1}^{S}\frac{2^{-k(T-w-s)}}{s^a}$$
$$< 2^{-k(T-w)}\sum_{s=1}^{S}\frac{2^{ks}}{s^a}.$$

Algebraically, when $s \ge 1$, we have $\frac{2^{ks}}{s^a} < \frac{2^{k(s+1)}}{s}$. We get

$$\sum_{s=1}^{S}\frac{2^{ks}}{s^a} < \sum_{s=1}^{S+1}\frac{2^{ks}}{s} < \int_2^{S+1}\frac{2^{kx}}{x}dx < \mathrm{Ei}(k(S+1)\ln 2),$$

where Ei denotes the exponential integral function. It follows from the property $\mathrm{Ei}(x) < e^x\ln(1-\frac{1}{x})$ that

$$\sum_{s=1}^{S}\frac{2^{ks}}{s^a} < 2^{k(S+1)}\ln\left(1 - \frac{1}{k(S+1)\ln 2}\right) < \frac{2^{k(S+1)}}{k(S+1)}.$$

We thus have

$$\sum_{s=1}^{S} 2^{-k(T-w-s)}f(s) < \frac{2^{-k(T-w-S-1)}}{k(S+1)} = O\left(\frac{2^{-k(1-\mu)T}}{k\mu T}\right). \tag{8}$$

It then follows from (4) that

$$\mathbb{E}[N^*(t)] = O\left(\frac{2^{-k(1-\mu)T}}{wk\mu T} + \frac{N_p}{w}\right).$$

We observe two properties: (1) When N_p is very small, both the required SRAM space $mcR + 2L$ (L is the length of B and b) and DRAM space $TmcR + L$ decreases in T, making our approach scalable. This is not surprising as a larger T decreases the number of recorded non-persistent items. (2) A small TCAM is sufficient to hold almost all the overflowed items, limiting both memory cost and looking up complexity.

5.2. False positive rate

It follows from (2) that the false positive rate for any epoch is $p = 2^{-k}$. For any non-persistent item e with spread s, the false positive rate, denoted as P_{FP}^s, can be upper-bounded as

$$P_{FP}^s \le p^{T-w-\min(s,T-w)} = 2^{-k(T-w-\min(s,T-w))}. \tag{9}$$

We can derive the false positive rate, denoted by P_{FP}, as

$$P_{FP} = \sum_{s=1}^{T-1} P_{FP}^s f(s). \tag{10}$$

We also consider the two cases in Section 5.1.

- **Case 1**. We can easily upper-bound P_{FP} by $2^{-k(T-w-S)}$.
- **Case 2**. Under Zipf distribution, it follows from (8) that

$$P_{FP} \le \sum_{s=1}^{S} 2^{-k(T-w-s)}f(s) = O\left(\frac{2^{-k(1-\mu)T}}{wk\mu T}\right).$$

5.3. Complexity

We now analyze the complexity of our algorithm.

Recording. Upon arrival of an item e, our algorithm first check the Bloom filters B and b whether e is a persistent item and whether it has already been processed. These operations have complexity $O(k)$. If e has not been processed and needs to be recorded, our algorithm then invokes the procedure Insert. In the worst case, the algorithm needs to check d buckets and the TCAM, leading to $O(d)$ complexity. The overall complexity sums up to $O(d+k)$ in the worst case if the TCAM size is $\Theta(1)$.

Lookup. Given the ID of an item e, our algorithm first checks the Bloom filter B if e is persistent, resulting $O(k)$ complexity. If e is a persistent item, the algorithm searches the d buckets (cd cells) and if necessary the TCAM for e, leading to the worst-case complexity $O(k+cd)$.

Reconstruction. To reconstruct the persistent item list, our algorithm parses all the recorded items and checks the Bloom filter for each of them. Recall (5), we can derive the expected number of recorded items, denoted by N_R as:

$$\mathbb{E}[N_R] = \sum_{t=T-w}^{T-1} \mathbb{E}[N^*(t)].$$

By the same reasoning as (6), we obtain that $N_R \le e\mathbb{E}[N_R]$ w.h.p. Consider the two practical cases in Section 5.1. For Case 1, the complexity is upper-bounded w.h.p. by $O\left(\frac{k}{w}(N\cdot 2^{-k(T-w-S)}+N_p)\right)$. For Case 2, the corresponding bound is $O\left(\frac{2^{-k(1-\mu)T}}{w\mu T}+\frac{kN_p}{w}\right)$. In both cases, if persistent item mining is executed with a long time horizon T, the complexity is largely dominated by the size of persistent items N_p, which is a significant performance gain over existing approaches requiring parsing all the items.

5.4. Parameter tuning and optimization

We conclude this section by discussing how the parameters are tuned to achieve desired performance trade-off.

Bloom filter parameters. Our algorithm uses two Bloom filters B and b of length L with k bits for each item. Following standard Bloom filter analysis, k and L are set such that $k=\frac{L}{N}\ln 2$ so as to minimize false positive rate. In practice, given the target false positive rate, we can choose the minimum k based on (10) and then derive the corresponding L.

Parameters in our algorithm. The main parameters used in our algorithm are d and w. From our analysis in this section, we can see that selection of d involves a trade-off between memory efficiency and algorithm complexity, both increases in d. A possible way to set d, given the performance requirement on the item recording and lookup time, is to choose the maximum d satisfying the time constraint to optimize memory efficiency. On the other hand, w impacts both false negative and false positive rates. A natural choice is to minimize the weighted sum of them, which can be numerically done by using the upper-bound derived in Section 5.1.

SRAM parameters. The last set of parameters, m and c, are related to the SRAM. The configuration is detailed in Section 5.1.

We conclude this section by noting that in the case where T is unknown, we can set the parameters based on a lower-bound of T. We can observe from our analysis that the estimation accuracy can still be satisfied. However, the price to pay is that we cannot achieve optimal memory efficiency.

6. Variants and extensions

In this section, we investigate a number of variants and extensions related to the persistent item lookup problem.

6.1. Generalized persistent item lookup

The persistent item lookup problem can be generalized with the following definitions of persistent items:

- An item is persistent if occurring in at least $\hat{T}\le T$ epochs;
- An item is persistent if occurring in at least $\hat{T}$ consecutive epochs.

Note that if $\hat{T}=T$, both generalizations degenerate to the baseline problem instance addressed previously.

We now describe a unified framework by extending our algorithm to solve both generalizations.

Data structure. We replace the Bloom filter B by a counting Bloom filter whose counter range is $[0,\hat{T}]$. The other Bloom filter b is maintained as in the baseline formulation. At the end of each epoch, for each bit 1 in b, we increment the counter of the corresponding bit in B by 1 if the value of the counter is less than $\hat{T}$. In this way B can trace the number of occurrence of each item. For the second formulation, for each bit 0 in b, we clear the corresponding counter in B to 0, which indicates that the items casting to any of these bits are absent in the current epoch. We then build another Bloom filter B' of the same length as B, in which $B'(i)$ is set to 1 if $B(i)=\hat{T}$ and 0 otherwise. In this way, we can check whether an item e appears in at least $\hat{T}$ consecutive epochs by checking if $B(h_i(e))=\hat{T}$, $1\le i\le k$.

Recording. The insertion procedure works in the same way as in the baseline formulation in the last w epochs. To check whether an item e should be recorded in epoch t, we proceed as below:

- For the first formulation, we check whether e has appeared in at least $\hat{T}-(T-t)$ epochs. If not, noticing that there are $(T-t)$ epochs left until the end of the measurement, e cannot be a persistent item and thus should not be recorded.
- For the second formulation, we check whether e has appeared in at least the last $\hat{T}-(T-t)$ epochs by checking whether the minimum counter value of $B(h_i(e))$ $(1\le i\le k)$ is at least $\hat{T}-(T-t)$. At the end of each epoch t $(\max\{\hat{T},T-w\}\le t<T)$, we dump B' off-die. We denote B' dumped at epoch t as B'_t.

Lookup. Given an item e, for the first formulation, we examine the k counters at positions $h_j(e)$ $(1\le j\le k)$ in B and mark e as persistent if none of the counters is smaller than $\hat{T}$. For the second formulation, we examine the k bits $h_j(e)$ $(1\le j\le k)$ in B'_t and mark e as persistent if there exists t such that all the k bits are 1. Similar operations are performed in both formulations to reconstruct the persistent item list.

For presentation conciseness, we mainly focus on the design intuition and idea without detailing the performance analysis. The analysis we present for the baseline algorithm in Section 5 can be adapted in the generic case by taking into account the counting error in counting Bloom filters.

6.2. The case of distributed streams

The second variant is the distributed situation where there are z substreams $f_1, f_2, \cdots, f_z$, each monitored at a different node F_i $(1\le i\le z)$ in a distributed environment. The goal is to identify the persistent items, where an item is persistent if it occurs in at least one substream at each epoch.

To apply our algorithm in this context, we set up a master node F whose role is to coordinate the whole monitoring process. The master node can be one of the monitoring node F_i or a dedicated node. Each monitor node F_i runs Algorithm 1 locally. At the end of each epoch, F_i uploads its Bloom filter B to F. We denote the local Bloom filter B at F_i as B_i. F then computes $\overline{B}=\bigvee_{i=1}^{z}B_i$ as the Bloom filter that records all the persistent items and broadcasts $\overline{B}$ to each monitor node F_i such that F_i further sets B_i to $\overline{B}$. At the end of the measurement each monitoring node F_i uploads its recorded items to F. The persistent item search and reconstruction operations in the single stream scenario can be easily extended in this new context.

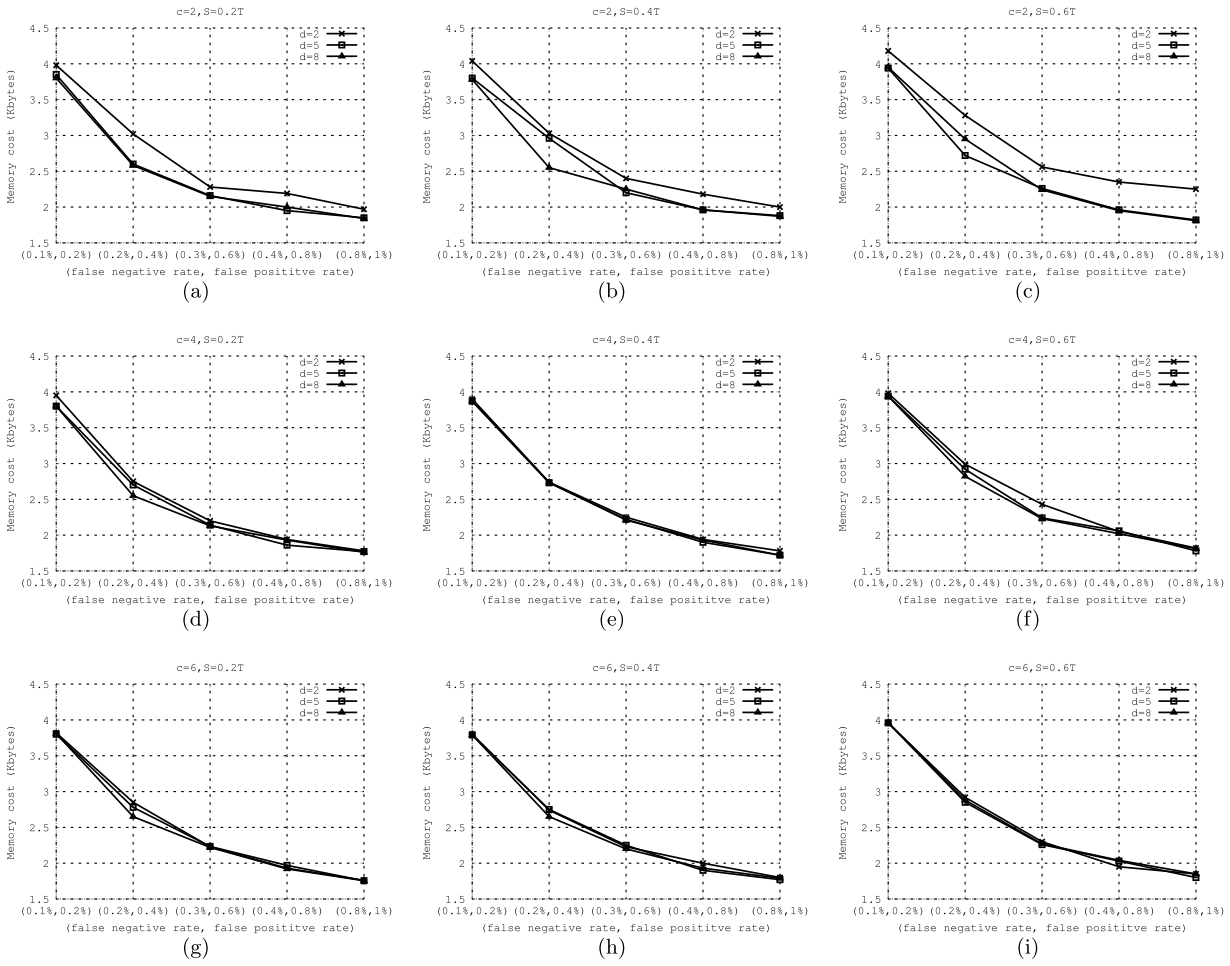

Fig. 2. On-die memory cost of our algorithm under different parameter settings.

One drawback of the above natural extension is that an item may be recorded at several monitoring nodes. To limit duplicated recording, we propose to *desynchronize* the recording procedure at each monitoring node. Specifically, for each item e to be recorded at F_i, the epoch in which e is effectively recorded is now set to $[g(e)+i]\%w$. By this way e is recorded in different epochs at different monitoring nodes. Moreover, F_i maintains a Bloom filter b'_i tracing all the items recorded locally. All b'_i are collected by F who further computes $b' = \bigvee_{i=1}^{z} b'_i$, which traces all the recorded items. F then broadcasts b' to each F_i, who checks b' before recording any item to avoid duplicated recording.

6.3. Further improving memory efficiency

In our algorithm, a recorded item is not necessarily a persistent one as a recorded item may not occur in a later epoch; in this case the memory space recording its information is *wasted*. Motivated by this observation, to further improve the memory efficiency, we split the information of each item into multiple parts and record them in multiple epochs. In this way if an item does not occur in a later epoch, we do not need to record the remaining parts, thus reducing memory waste.

To further illustrate our idea, we take an example where the information of each item e is divided into two parts, each containing half of the item ID denoted by e_1 and e_2. We then regard e_1 and e_2 as independent items and record them in the corresponding epochs as specified in our algorithm only if e appears in all the preceding epochs. To search and retrieve a persistent item e, the algorithm first verifies the Bloom filter B whether e is persistent and proceeds to retrieve the two parts by using e_1 and e_2.

To enable persistent item list reconstruction, we need to attach with one of the two parts the entire item ID e so as to locate the other part. Suppose that $g(e_1) < g(e_2)$, i.e., the first part is supposed to be recorded in an earlier epoch. We can attach the entire ID e in either part, but it is preferable to attach it with the second part. The subtlety is that by doing so, if e does not occur in some epoch between $g(e_1)$ and $g(e_2)$, it is not a persistent item and hence we do not need to record e_2, thus avoiding recording the entire ID.

7. Numerical and empirical study

In this section, we evaluate our algorithm on both synthetic and real data streams for a variety of application scenarios.

7.1. Experiment on synthetic data

We simulate a data stream of 10^6 items among which 0.1% of them, i.e., 1000 items, are persistent. The length of item IDs is 32 bits, corresponding to e.g. an IP address in networking applications. The time horizon T is set to 100 periods. The items follow a Zipf distribution with $\alpha = 2$ for the non-persistent items with the maximum spread S varying from $0.2T$ to $0.6T$ to simulate different skewness. We first evaluate the performance of our algorithm in various parameter settings and then compare our algorithm with the state-of-the-art approach PIE.

To evaluate our algorithm, we trace the minimum memory space required to satisfy the lookup accuracy in terms of false positive and false negative rates. To this end, we vary the target false positive and false negative rates from 0.001 to 0.01 and trace the minimum memory

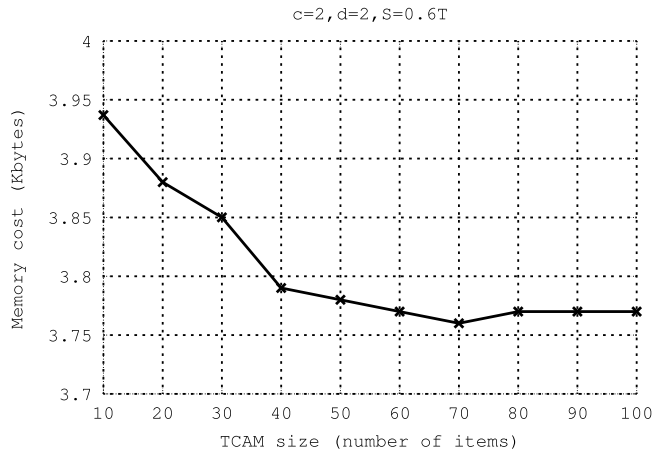

Fig. 3. On-die memory cost under various TCAM size.

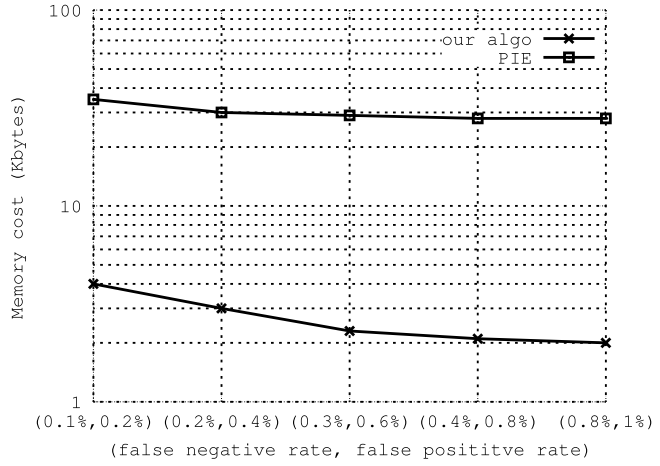

Fig. 4. Performance comparison between our algorithm and PIE (y-axis in log scale).

space such that the false positive and false negative rates are both met. To this end, we set $k = 3$, w to $T - S$, and increase the Bloom filter length L until the target false positive rate is met. We then increase m until the false positive rate is met. We trace the resulting on-die memory cost under different values of d and c.

From the simulation results, illustrated in Fig. 2, we can draw the following observation.

Among the different scenarios we simulate, we observe that the memory cost decreases significantly when we relax the lookup accuracy requirement in terms of the false positive and false negative rates. This observation is coherent to our analytic results.

We also observe that the performance of our algorithm is not very sensitive to d and c, which, from the engineering perspective, is a desirable property that makes the implementation simple and robust. Nevertheless, we observe a slight performance gain with large d, a logic result as larger d offers more candidates to record items and thus leads to more balanced and compact outcome.

We also simulate the impact of TCAM size on performance. To this end, we vary the TCAM size from 5 to 100 items under the setting $c = 2$, $d = 2$, $S = 0.6T$. The results in Fig. 3 shows that the memory cost further decreases when a small global TCAM is used and the performance gain stabilizes when the TCAM scales to around 50 items, i.e., 200 bytes. We observe similar results for other settings of c, d and S.

We then compare our algorithm with the state-of-the-art approach PIE [9] under the optimum parameter setting. The core idea of PIE is the use of a compact data structure called the space–time Bloom filter (STBF). In our simulations, we increase the size of the STBF in PIE and trace the minimum memory space such that the false positive and false negative rates are met. We vary the values of c, d, and S. Fig. 4 illustrates the comparison results for the setting $c = 2$, $d = 2$ and $S = 0.2T$. We observe similar results for other settings, which are omitted here for conciseness. From the results illustrated in Fig. 4 (in logarithmic scale), we observe that our algorithm incurs only 5 – 20% memory cost compared to PIE, given that PIE is shown in [9] to outperform other classical approaches based on Invertible Bloom Filter (IBF) [26] and Count-Min (CM) sketch [27]. By further examining the simulation traces, we observe that despite the efforts in PIE, it still records a large number of non-persistent items; in contrast, the number of non-persistent items recorded in our algorithm is significantly smaller, resulting in significant memory saving.

Table 2
Summary of traces used in our experiment.

Trace	Duration	#pkts	# flows
CHIC	6 min	25.3M	101 374
ICSI	1 h	1.49M	8797
DC	1 h	10 289	10 289

7.2. Experiment on real data traces

We proceed to evaluate our algorithm on real data traces by using 3 network traces CHIC [28], ICSI [29] and DC [30], summarized as below.[7]

- CHIC is a backbone header trace published by CAIDA and collected in 2015. It traces the arrival times of packets at a 10GigE link interface with the flow IDs associated with those packets. In our experiment, we capture HTTP flows for 6 min.
- ICSI is an enterprise network traffic trace collected at a medium-sized enterprise network. In our experiment, we use TCP traces from 22 different ports in one hour.
- DC is a data center traffic trace collected at a university data center for more than an hour. In our experiment, we use TCP traces generated in one hour.

We note that these traces are also used to evaluate PIE in [9]. The same PIE parameters are taken in our experiment. Table 2 summarizes the traces used in our experiment.

We follow the same procedure as in our simulation by first evaluating the performance of our algorithm in various parameter settings and then comparing our algorithm with the state-of-the-art approach PIE.

Fig. 5 illustrates the experiment results of our algorithm in terms of memory overhead. For presentation conciseness, we illustrate the result in the setting $c = 4$ and $w = 25$ for the three traces and similar result trends are observed in other settings, too. From our experiment results, we can draw similar observations as the simulations we perform on synthetic data.

We then compare our algorithm to PIE over the three traces. Fig. 6 illustrate representative results of our comparison, one for each trace, noticing that we observe similar results for other settings. The results are globally coherent to the simulation results, i.e., our algorithm incurs less memory cost under the same lookup accuracy. Nevertheless, we observe a larger performance gap between our algorithm and PIE over the real network traces. This larger gap can be explained by the fact that the real network traces are more skewed than the data streams generated in our simulations and our algorithm is particularly tailored for high skewness by recording only persistent items, while PIE still

[7] For more detailed explanation of the traces, readers are referred to the respective references.

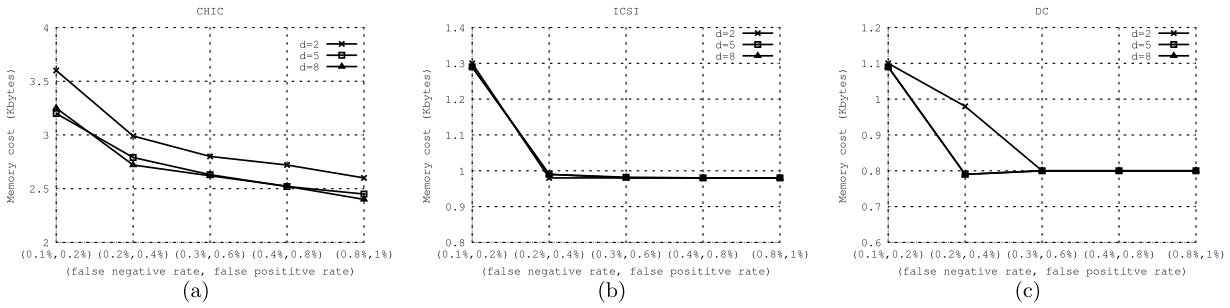

Fig. 5. Performance of our algorithm on real data traces.

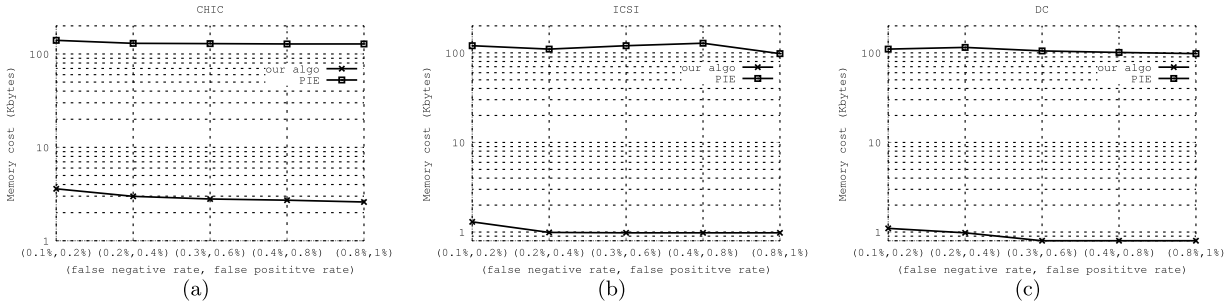

Fig. 6. Performance comparison between our algorithm and PIE on real data traces (y-axis in log scale).

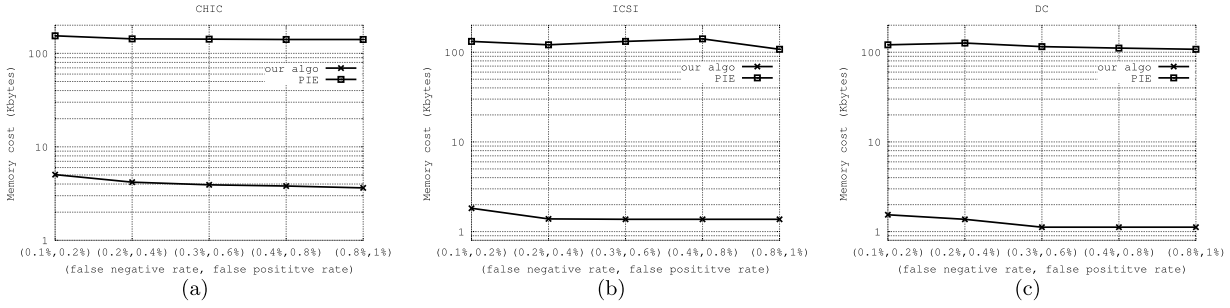

Fig. 7. Performance comparison between our generalized algorithm and PIE on real data traces (y-axis in log scale).

needs to record all the items. To complete our experimentation, we simulate the generalized variant of our algorithm where an item is persistent if occurring in at least $0.8T$ epochs. Fig. 7 compares the performance of our algorithm against PIE in such generalized scenario. The memory cost is almost the same as the baseline case with PIE as the data structure is the same in the generalized case. For our algorithm, the memory cost increases due to the use of counting Bloom filter. However, this overhead is limited as the main data structure remains the same.

8. Conclusion

We have investigated the problem of persistent item lookup, a pivotal functionality in many computing and networking paradigms. Our main contribution is the design and analysis of an algorithm and the related data structure that are compact and amendable for hardware implementation, while guaranteeing user-tunable lookup accuracy and supporting interactive query processing. Despite our focus on persistent item lookup, we believe that some of our techniques may well extend beyond this problem to other data stream mining problems where memory efficiency is a central concern.

CRediT authorship contribution statement

Lin Chen: Algorithm design and analysis. **Haipeng Dai:** Algorithm analysis and proof. **Lei Meng:** Algorithm evaluation. **Jihong Yu:** Algorithm analysis and evaluation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] H. Huang, Y.E. Sun, S. Chen, H. Xu, Y. Zhou, Persistent traffic measurement through vehicle-to-infrastructure communications, in: Proc. IEEE ICDCS, 2017.
[2] N. Immorlica, K. Jain, M. Mahdian, K. Talwar, Click fraud resistant methods for learning click-through rates, in: Proc. ACM WINE, 2005.
[3] Q. Xiao, Y. Qiao, M. Zhen, S. Chen, Estimating the persistent spreads in high-speed networks, in: Proc. IEEE ICNP, 2014.
[4] T. Li, S. Chen, Y. Ling, Per-flow traffic measurement through randomized counter sharing, IEEE/ACM Trans. Netw. 20 (5) (2012) 1622–1634.
[5] P. Zhao, C.C. Aggarwal, M. Wang, gSketch: On query estimation in graph streams, in: Proc. VLDB, 2011.
[6] S. Nath, P.B. Gibbons, S. Seshan, Z.R. Anderson, Synopsis diffusion for robust aggregation in sensor networks, in: Proc. ACM SenSys, 2004.

[7] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, D. Srivastava, Holistic UDAFs at streaming speeds, in: Proc. ACM SIGMOD, 2004.
[8] A. Goyal, H. Daumé III, G. Cormode, Sketch algorithms for estimating point queries in NLP, in: Proc. Joint Conf. Empirical Methods in Natural Language Processing and Computational Natural Language Learning, 2012.
[9] H. Dai, M. Shahzad, A.X. Liu, Y. Zhong, Finding persistent items in data streams, Proc. VLDB 10 (4) (2016) 289–300.
[10] B. Lahiri, S. Tirthapura, J. Chandrashekar, Space-efficient tracking of persistent items in a massive data stream, Stat. Anal. Data Min. 7 (1) (2014) 70–92.
[11] Y. Zhou, Y. Zhou, M. Chen, S. Chen, Persistent spread measurement for big network data based on register intersection, Proc. ACM Meas. Anal. Comput. Syst. 1 (1) (2017) 1–29.
[12] P. Flajolet, E. Fusy, O. Gandouet, F. Meunier, Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm, in: Proc. Conf. Anal. Algo., 2007.
[13] G. Cormode, M. Hadjieleftheriou, Finding the frequent items in streams of data, Commun. ACM 52 (10) (2009) 97–105.
[14] G. Cormode, S. Muthukrishnan, An improved data stream summary: The count-min sketch and its applications, J. Algorithms 55 (1) (2005) 58–75.
[15] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, Proc. VLDB 5 (12) (2012) 1699.
[16] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge University Press, 1995.
[17] M. Mitzenmacher, A.W. Richa, R. Sitaraman, The power of two random choices: A survey of techniques and results, in: Handbook of Randomized Computing, Kluwer, 2000, pp. 255–312.
[18] B. Vöcking, How asymmetry helps load balancing, J. ACM 50 (4) (2003) 568–589.
[19] M. Mitzenmacher, The power of two choices in randomized load balancing, IEEE Trans. Parallel Distrib. Syst. 12 (10) (2001) 1094–1104.
[20] R. Pagh, F.F. Rodler, Cuckoo hashing, J. Algorithms 51 (2) (2004) 122–144.
[21] A. Kirsch, M. Mitzenmacher, U. Wieder, More robust hashing: Cuckoo hashing with a stash, SIAM J. Comput. 39 (4) (2009) 1543–1561.
[22] A.S. Tanenbaum, Structured Computer Organization, Prentice-Hall, 2005.
[23] K. Pagiamtzis, A. Sheikholeslami, Content-addressable memory circuits and architectures: a tutorial and survey, IEEE J. Solid-State Circuits 41 (3) (2006) 712–727.
[24] T. Yang, Y. Zhou, H. Jin, S. Chen, X. Li, Pyramid sketch: A sketch framework for frequency estimation of data streams, Proc. VLDB (2017).
[25] M. Mitzenmacher, E. Upfal, Probability and Computing: Randomized Algorithms and Probabilistic Analysis, Cambridge University Press, 2005.
[26] D. Eppstein, M.T. Goodrich, F. Uyeda, G. Varghese, What's the difference?: Efficient set reconciliation without prior context, in: Proc. ACM SIGCOMM, 2011.
[27] G. Cormode, S. Muthukrishnan, An improved data stream summary: The count-min sketch and its applications, J. Algorithms 55 (1) (2005) 58–75.
[28] The CAIDA UCSD anonymized internet traces, www.caida.org/data/overview.
[29] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, B. Tierney, A first look at modern enterprise traffic, in: Proc. ACM IMC, 2005.
[30] T. Benson, A. Akella, D.A. Maltz, Network traffic characteristics of data centers in the wild, in: Proc. ACM IMC, 2010.

Lin Chen received the B.E. degree in radio engineering from Southeast University, China, in 2002, the Engineer Diploma and Ph.D. degrees from Telecom ParisTech, Paris, in 2005 and 2008, respectively, and the M.S. degree in networking from the University of Paris 6. From 2009 to 2019, he was an associated professor with the department of Computer Science, University of Paris-Sud. He is currently a professor with the School of Data Computer Science, Sun Yat-sen University. His main research interests include distributed algorithm design and analysis in networked systems, security and privacy in cyber-physical systems.

Haipeng Dai received the B.S. degree in the Department of Electronic Engineering from Shanghai Jiao Tong University, Shanghai, China, in 2010, and the Ph.D. degree in the Department of Computer Science and Technology in Nanjing University, Nanjing, China, in 2014. He is an associate professor in the Department of Computer Science and Technology in Nanjing University.

He is an IEEE and ACM member.

He received Best Paper Award from IEEE ICNP'15, Best Paper Award Runner-up from IEEE SECON'18, and Best Paper Award Candidate from IEEE INFOCOM'17.

Lei Meng received the BS degree in software engineering from Jilin University in 2018. He is currently working toward the MS degree in the computer science and technology at Nanjing University, focusing on knowledge graph and data mining.

Jihong Yu received the B.E degree in communication engineering and M.E degree in communication and information systems from Chongqing University of Posts and Telecommunications, Chongqing, China, in 2010 and 2013, respectively, and the Ph.D. degree in computer science at the University of Paris-Sud, Orsay, France, in 2016. He was a postdoc fellow in the School of Computing Science, Simon Fraser University, Canada. He is currently a professor in the School of Information and Electronics at Beijing Institute of Technology. His research interests include RFID, backscatter networks, and Internet of things.