The VLDB Journal (2019) 28:735–763
https://doi.org/10.1007/s00778-019-00560-1

REGULAR PAPER

Fast and accurate stream processing by filtering the cold

Tong Yang[1] · Jie Jiang[1] · Yang Zhou[1] · Long He[1] · Jinyang Li[1] · Bin Cui[1] · Steve Uhlig[2] · Xiaoming Li[1]

Received: 26 May 2018 / Revised: 22 June 2019 / Accepted: 29 July 2019 / Published online: 13 August 2019

Abstract

Approximate stream processing algorithms, such as Count-Min sketch, Space-Saving, support numerous applications across multiple areas such as databases, storage systems, and networking. However, the unbalanced distribution in real data streams are challenging to existing algorithms. To enhance these algorithms, we propose a meta-framework, called Cold Filter, that enables faster and more accurate stream processing. Different from existing filters that mainly focus on hot (frequent) items, our filter captures cold (infrequent) items in the first stage, and hot items in the second stage. Existing filters also require two-direction communication—with frequent exchanges between the two stages; our filter on the other hand is one-direction—each item enters one stage at most once. Our filter can accurately estimate both cold and hot items, providing a level of genericity that makes it applicable to many stream processing tasks. To illustrate the benefits of our filter, we deploy it on four typical stream processing tasks. Experimental results show speed improvements of up to 4.7 times, and accuracy improvements of up to 51 times.

Keywords Data streams · Sketch · Frequency estimation · Top-k hot items · Heavy changes · Persistent items

1 Introduction

In many big data scenarios, the data come as a high-speed stream [1–5], such as online social networks, videos, sensors data, network traffic, web clicks and crawls. Such data streams are often processed in a single pass [4,6–8]. In many applications, some statistical information in each time window of the data stream is needed, such as item frequencies [9], top-*k* hot items [10,11], heavy changes [12], quantiles [13], and algorithms regarding time series like SSH [14], Ada-Sketch [15]. However, it is often impractical to compute exact statistics (e.g., using hash tables), because the space and time cost for storing the whole data stream are too high. Therefore, probabilistic data structures [8,11,16–26] have become popular for approximate processing.

The speed at which data streams arrive and their sizes, together, make approximate stream processing challenging. First, the memory usage for the processing should be small enough to fit into the limited-size and expensive SRAM (Static RAM, such as CPU cache), so as to match the required processing speed. Second, having to process the data in a single pass also constrains the speed at which processing must take place. Finally, to guarantee the performance of applications, accuracy should be as high as possible.

1.1 Characteristics of real data streams

According to our tests on real datasets and the literature [4,8], in practice, the items present in real data streams often

✉ Bin Cui
bin.cui@pku.edu.cn

Tong Yang
yang.tong@pku.edu.cn

Jie Jiang
jie.jiang@pku.edu.cn

Yang Zhou
zhou.yang@pku.edu.cn

Long He
helong@pku.edu.cn

Jinyang Li
lijinyang@pku.edu.cn

Steve Uhlig
steve.uhlig@qmul.ac.uk

Xiaoming Li
lxm@pku.edu.cn

[1] Department of Computer Science and Technology & Key Laboratory of High Confidence Software, Technologies (MOE), Peking University, Beijing, China

[2] Queen Mary University of London, London, UK

obey unbalanced distribution, such as Zipf [27] or Power law [28]. This means that most of the items are unpopular (called cold items), while a few items are very popular (called hot items). We refer to such data streams as *skewed* data streams. Such characteristics bring significant challenges to stream processing tasks. There are two kinds of stream processing tasks: *frequency-based tasks* and *persistency-based tasks*. Frequency-based tasks focus on the frequencies of items in a stream. For example, we may want to know the frequency of an item, or the set of items whose frequencies are above a threshold. Persistency-based tasks on the other hand have to do with the persistency of items, which means that if we divide the whole stream into many sub-streams, we may want to know in how many sub-streams an item occurs. We discuss four examples of stream processing tasks and the challenges that come with them. The first three tasks are frequency-based tasks and the last one is persistency-based.

1.2 Estimating item frequency

Estimating the frequency of each item is one of the most classic tasks in data streams [8,9]. Two typical solutions are the Count-Min sketch [9] and the CM–CU sketch [29]. They both use a number of counters of fixed size to store the frequencies of items. If each counter is small, the frequencies of hot items that are beyond the maximum value of the counters cannot be recorded. This is not acceptable, as hot items are often regarded as more important in practice. If each counter is large enough to accommodate the largest frequency, the high bits of most counters will be wasted, as hot items are fewer than cold items in real data streams.

1.3 Finding top-*k* hot items

Finding the Top-k hot items is important in various fields, including in data streams [8–11,30]. As not all incoming items can be stored, and an item can only be processed once, the state-of-the-art solution, Space-Saving [10], approximately keeps top-k items in a data structure called *Stream-Summary*. Given an incoming item that is not in the Stream-Summary, Space-Saving assumes it is a little *larger* than the minimum one in the Stream-Summary, and exchanges them, so as to achieve fast processing speed. Most items are cold, and every cold item will enter the Stream-Summary, and could stay or be expelled. Frequent exchanges, incurred by cold items, should be avoided, as they degrade the accuracy of the results of top-k.

1.4 Detecting heavy changes

The frequencies of some items can significantly change in a short amount of time. Detecting such changes is important for search engines [31] and security [12,32] for instance. The state-of-the-art solution is FlowRadar [32] that relies on an Invertible Bloom Lookup Table (IBLT) [33]. It uses an IBLT to approximately monitor all incoming items and their frequencies in two adjacent time windows. Then, it compares their frequencies and draws conclusions. FlowRadar achieves high accuracy if there is enough memory to record every item, which is not always the case.

1.5 Finding persistent items

Some items may not have large frequencies, but they may occur frequently enough within sub-streams: If we split the whole data stream into many sub-streams, an item may appear in most of these sub-streams, and finding such items is important for stealthy DDoS detection, stealthy port scanning detection and click-fraud detection [34]. The state-of-the-art solution for this task is PIE [35], which is based on Raptor code [36]. For each item, PIE writes its Raptor code to several buckets in an array by hashing. PIE builds such an array for each sub-stream. A bucket will be invalid if it is mapped by two or more distinct items due to hash collisions. A persistent item may not be a heavy hitter, but it is definitely not a cold item. Since most items are cold, they are definitely not persistent items, but they may cause many buckets to become invalid in PIE. As a result, PIE need a large memory to guarantee it can record persistent items accurately.

In a nutshell, the characteristics of skewed data streams make the state-of-the-art algorithms perform poorly or require large amounts of resources in the above four tasks. To address this, several algorithms have been proposed to do filtering on data streams, such as the Augmented sketch [4], skimmed sketch [37]. They use a CPU-cache-like mechanism: all items are first processed in the first stage, and then cold items are swapped out to the second stage. The advantage is that hot items then require fewer memory accesses. However, it is difficult to catch hot items accurately, because all hot items are initially cold and stored in the second stage and then become hot. Therefore, existing algorithms need to be implemented using two-way communication, with frequent exchanges and communication between the two stages. Existing filters using two-direction communication have the following shortcomings: (1) they use a heap or a table in the first stage, and thus often need many memory accesses to process each item; (2) the first stage can capture only a few hot items (e.g., 32 hot items in the Augmented sketch), because more hot items means more memory accesses; (3) they make it hard to process items in parallel. Our design goal is to devise a filter that relies on one-direction communication, and targets accurate estimation of both hot and cold items at a high processing speed.

Our solution, Cold Filter (CF), as shown in Fig. 1, uses a two-layer sketch with small counters to accurately record the

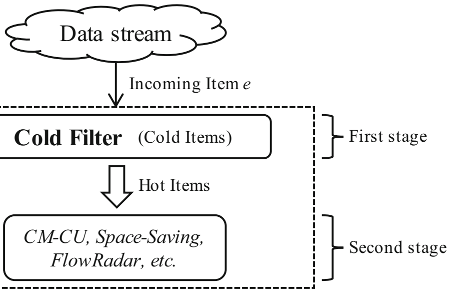


Fig. 1 Cold Filter captures unpopular (cold) items in the first stage, and forwards popular (hot) items to the second stage

frequencies of cold items.[1] If all the hashed counters overflow, CF will report the incoming item as hot (one-direction communication) and send it to some stream processing algorithm (e.g., the CM–CU sketch, Space-Saving, and FlowRadar). We can combine CF with different algorithms in different ways for maximize the benefits. Hence, we call CF a meta-framework. CF works with existing algorithms by requiring limited modifications, while significantly improving accuracy. The first stage only uses small counters to store the frequencies of cold items, making it memory efficient. By filtering out a large number of cold items, the second stage concentrates on hot items, therefore achieving high accuracy. To improve the processing speed, we leverage a series of techniques: *(1) aggregate-and-report (including SIMD parallelism), (2) one-memory-access, and (3) multi-core parallelism*. As our Cold Filter can accurately record the information of both cold items and hot items, it is applicable to most stream processing tasks. All source code is made publicly available on Github [38].

2 Related work

Sketches have been widely applied to estimating item frequency in data streams. The most widely used sketch is the Count-Min sketch [9]. It relies on d arrays, $A_1 \ldots A_d$, and each array consists of w counters. There are d hash functions, $h_1 \ldots h_d$, in the Count-Min sketch. When inserting an item e with frequency f, the Count-Min sketch increments all the d *hashed counters*, $A_1[h_1(e)] \ldots A_d[h_d(e)]$, by f. When querying an item e', it reports its estimated frequency as the minimum of the d hashed counters, i.e., $\min_{1 \leqslant i \leqslant d}\{A_i[h_i(e')]\}$. Another algorithm, the CM–CU sketch [29], achieves higher accuracy. The only difference is that CM–CU only increments the smallest one(s) among the d hashed counters. Both CM and CM–CU have no underestimation error.

Sketches have attracted the attention of many researchers, due to their ability to summarize data streams efficiently. We list some recent work in the following. Based on Space-Saving [10,39] proposed Unbiased Space-Saving which can give an unbiased result for a query of frequency estimation. [40] proposed the idea of persistent (multi-version) sketches, which can answer queries about the stream of specific time points, and persistent Bloom filters[41] extended this idea for Bloom filters. [42] proposed bias-aware sketches, which can be proved that have strictly better error guarantees. [43] studied how to create a sketch for a matrix on the sliding window model. Some approximate query processing systems [44–46] use sketches as basic operators to summarize the streaming data and transform queries of users to queries to the sketches. SketchML [47] uses sketches to compress gradient in machine learning systems to reduce the transmission volume in the network.

The closest work to our Cold Filter is the Augmented sketch [4]. It adds an additional filter (a queue with k items and counters) to an existing sketch Φ, to maintain the most frequent items within this filter. When inserting an item e, it scans the items stored in the filter one by one. If e has already been in the filter, it just increments its corresponding counter. Otherwise, it stores e with an initial count of one if there is available space in the filter. If there is no available space, i.e., the filter is full, it inserts this item into the sketch Φ. During insertions, if the frequency of this item reported by Φ is larger than the minimum value (associated with item e') in the filter, the Augmented sketch needs to expel the item e' to Φ, and insert e into the filter.

There are some possible applications where Cold Filter helps to enhance other algorithms. MISSION in [48] is a framework for large-scale feature selection using a Count-Sketch. MISSION uses information from the given dataset to compute a stochastic gradient update term, consider the gradient as a data stream where each non-zero entry is an item, and insert these "items" into the sketch. Time Adaptive Sketches (Ada-Sketches) [15] integrate pre-emphasis and de-emphasis, making sketching algorithms such as Count-Min Sketch temporally adaptive and thereby more time adaptive. For each item, Adaptive Count-Min Sketch (Ada-CMS) hashes both the item itself and its arriving time simultaneously to locate counters in the sketch, and multiplies the update weight with a monotonically increasing function of time, thus inflating the weight of more recent items. MACH in [49] which is used for training 100,000 classes on a single Titan X can also be seen precisely as a Count-Min Sketch for finding heavy hitters by setting the number and range of universal hashes to be particular values. We believe our Cold Filter can be used in such applications to improve their performance.

[1] An optional part can exist in the first stage in Fig. 1, in case more information is required about the cold items, i.e., not only their frequencies.

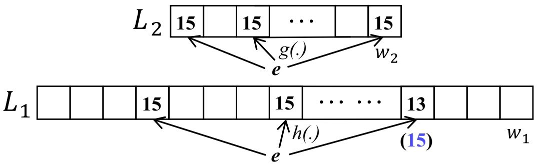


Fig. 2 Data structure of two-layer CF

3 The cold filter meta-framework

We employ the standard streaming model, namely the *cash register model* [50,51]. Given a *data stream* $\mathcal{S}$ with E items and N distinct items, where $N \leqslant E$. $\mathcal{S} = (e_1, e_2, \ldots, e_E)$, where each item $e_i \in U = \{e_{\beta_1}, e_{\beta_2}, \ldots, e_{\beta_N}\}$. Note that items in U are distinct, while items in $\mathcal{S}$ may not. Let t be the current time, e_t the current incoming item, and $\mathcal{S}_t = (e_1, e_2, \ldots, e_t)$ the *current sub-stream*. e_t occurs $f_{e_t}[1, t]$ times in the current sub-stream $\mathcal{S}_t$, and $f_{e_t}[1, E]$ times in the whole stream $\mathcal{S}$. For convenience, we use $f_{e_t}[t]$ and $f_{e_t}[E]$ for short.

Problem statement Given a data stream $\mathcal{S} = (e_1, e_2, \ldots, e_E)$ and a time t, the current sub-stream is $\mathcal{S}_t = (e_1, e_2, \ldots, e_t)$. Given the current item e_t, how do we accurately and quickly estimate whether its *current* frequency $f_{e_t}[t]$ exceeds the predefined threshold $\mathcal{T}$?

3.1 Naive solution

One naive solution is to use a sketch Φ (e.g., the Count-Min sketch, the CM–CU sketch, etc) as a CF. Specifically, we use Φ to record the frequency of each item starting from the first time point 1. For each incoming item, we first query Φ, and get its estimated frequency. Then we check whether this estimated frequency exceeds the threshold $\mathcal{T}$. However, this solution suffers from the drawback of *memory inefficiency* on real data streams. Suppose $\mathcal{T} = 1000$. For Φ, we set its counter size to 16, which can count the frequency of up to 65535. In real data streams however, most items have low frequencies and cannot "fill up" the counters that they are hashed to. As a result, many high-order bits in most counters of Φ are wasted, which means memory inefficiency and sub-optimal filtering performance. If instead we could automatically allocate small counters for cold items and large counters for hot items, the allocated memory would be much better utilized. This is what our proposed solution achieves.

3.2 Proposed solution

As shown in Fig. 2, our Cold Filter (CF)[2] consists of two layers: a low layer L_1, and a high layer L_2. These two layers are made of w_1 and w_2 counters, and associate with d_1 and d_2 hash functions ($h(.)$ and $g(.)$), respectively. The sizes of each counter at layer L_1 and layer L_2 are δ_1 and δ_2 bits, respectively. We split the threshold $\mathcal{T}$ into two parts: $\mathcal{T} = \mathcal{T}_1 + \mathcal{T}_2$ $(1 \leqslant \mathcal{T}_1 \leqslant 2^{\delta_1} - 1,\ 1 \leqslant \mathcal{T}_2 \leqslant 2^{\delta_2} - 1)$. The procedure for CF to process incoming item e_t is shown in Algorithm 1. There are two processes: update and report.

Algorithm 1: Stream processing algorithm for CF.

```
   Input: Incoming item $e_t$.
   Output: Update CF, and report whether $f_{e_t}[t] > \mathcal{T}$.
 1 $V_1 \leftarrow$ Estimate($L_1, e_t$);
 2 if $V_1 < \mathcal{T}_1$ then
 3     ConservativeUpdate($L_1, e_t, V_1 + 1$);
 4     return False;
 5 else
 6     /* concurrently overflow at layer $L_1$ */
 7     $V_2 \leftarrow$ Estimate($L_2, e_t$);
 8     if $V_2 < \mathcal{T}_2$ then
 9         ConservativeUpdate($L_2, e_t, V_2 + 1$);
10         return False;
11     else
12         return True;
```

Update process of CF As shown in Algorithm 1, we use V_1 and V_2 to denote the minimum value of the d_1 hashed counters at the low layer, and of the d_2 hashed counters at the high layer, respectively (see $Estimate(L_1, e_t)$ in line 1 and $Estimate(L_2, e_t)$ in line 7). If $V_1 < \mathcal{T}_1$, CF increments the smallest hashed counter(s) at the low layer by one, which is denoted by $ConservativeUpdate(L_1, e_t, V_1)$ in line 3. Note that if there are multiple counters with the same smallest value, all of them should be incremented. During the update process, the values of the d_1 hashed counters could differ. However, the increment operation of the smallest counters is always narrowing the differences of values of the d_1 hashed counters. If the value of one or more of these d_1 hashed counters reaches $\mathcal{T}_1$, then all the subsequent increments will be added to the other counters as well. Therefore, the ultimate state is that all d_1 hashed counters will reach $\mathcal{T}_1$ concurrently. We call this state the *concurrent overflow state*. When reaching this state (i.e., $V_1 = \mathcal{T}_1$), CF resorts to the high layer to record the information of this item. For the d_1 hashed counters in concurrent overflow state at the low layer, we propose a new strategy: keep them unchanged. This strategy makes it unnecessary to use additional flags to indicate the concurrent overflow state that is critical for subsequent query operations on CF. The update process at the high layer is similar to the one at the low layer: If $V_2 < \mathcal{T}_2$, CF increments the smallest hashed counter(s) by one (see line 9).

For the current item e_t, if its hashed counters concurrently overflow at the low layer, we must have $f_{e_t}[t-1] \leqslant$

[2] In the rest of this paper, "CF" refers to our two-layer Cold Filter.

V_1;[3] if its hashed counters concurrently overflow at the lower layer but not at the higher layer, $f_{e_t}[t-1] \leqslant V_1 + V_2$. This is because each past item e_t must increment the value of $V_1 + V_2$ by one, when $V_1 < \mathcal{T}_1$ or $V_1 = \mathcal{T}_1 \wedge V_2 < \mathcal{T}_2$ before updating. In fact, the potential gap between V_1 or $V_1 + V_2$ and $f_{e_t}[t-1]$ comes from the hash collisions between this item and other items at layer L_1 or L_2.

Report process of CF: Simply put, if the hashed counters concurrently overflow at both layers before updating, CF reports $f_{e_t}[t] > \mathcal{T}$; otherwise, CF reports $f_{e_t}[t] \leqslant \mathcal{T}$. Note that $f_{e_t}[t] = f_{e_t}[t-1] + 1$. We formally present the report process as follows:

1. If $V_1 < \mathcal{T}_1$ (line 2 in Algorithm 1), we have: $f_{e_t}[t-1] \leqslant V_1 < \mathcal{T}_1 < \mathcal{T}$. Thus, we report $f_{e_t}[t] \leqslant \mathcal{T}$.
2. If $V_1 = \mathcal{T}_1$ but $V_2 < \mathcal{T}_2$ (line 8), we have: $f_{e_t}[t-1] \leqslant V_1 + V_2 < \mathcal{T}_1 + \mathcal{T}_2 = \mathcal{T}$. Thus, we also report $f_{e_t}[t] \leqslant \mathcal{T}$.
3. If $V_1 = \mathcal{T}_1$ and $V_2 = \mathcal{T}_2$ (line 13), two cases are possible:
   (a) $f_{e_t}[t-1] \geqslant \mathcal{T}$, and $f_{e_t}[t]$ definitely exceeds $\mathcal{T}$. We should report $f_{e_t}[t] > \mathcal{T}$.
   (b) $f_{e_t}[t-1] < \mathcal{T}$, but the hash collisions lead to $V_1 = \mathcal{T}_1$ and $V_2 = \mathcal{T}_2$. We should report $f_{e_t}[t] \leqslant \mathcal{T}$.

   Unfortunately, it is not easy to differentiate these two cases. For the sake of space and time efficiency, we choose to report $f_{e_t}[t] > \mathcal{T}$ only.

Example As shown in Fig. 2, we set $d_1 = d_2 = 3, \delta_1 = \delta_2 = 4, \mathcal{T}_1 = \mathcal{T}_2 = 15$. For incoming item e_t: (1) If its three hashed counters at layer L_1 are 15, 15, 13. We get $V_1 = \min\{15, 15, 13\} = 13$. Then, we increment the third hashed counter at layer L_1 by one, and report $f_{e_t}[t] \leqslant \mathcal{T}$. (2) If its three hashed counters at layer L_1 are 15, 15, 15 (in blue color). We get $V_1 = \min\{15, 15, 15\} = 15 = \mathcal{T}_1$. Then, we need to access layer L_2. Assume its three hashed counters at layer L_2 are 15, 15, 15. We get $V_2 = \min\{15, 15, 15\} = 15 = \mathcal{T}_2$. Then, we need to report $f_{e_t}[t] > \mathcal{T}$.

This solution leads to no false negatives and only a small amount of false positives. If $f_{e_t}[t]$ exceeds threshold $\mathcal{T}$, CF will definitely identify this excess (no false negative). For a small portion of items whose frequencies $f_{e_t}[t]$ do not exceed threshold $\mathcal{T}$, CF may draw wrong conclusions (false positives).

Here we use a numerical example to further illustrate the advantages of our proposed two-layer CF over the naive solution. Suppose $\mathcal{T} = 1000$. For $\varPhi$ in the naive solution, we set its counter size to 16 bits. Recall that w denotes the number of counters in $\varPhi$. For our proposed two-layer CF, we set $\delta_1 = 4, \delta_2 = 16, \mathcal{T}_1 = 15, \mathcal{T}_2 = 985$. We allocate 50% memory to layer L_1, and 50% memory to layer L_2. Obviously, the w_1 of two-layer CF is twice the w of $\varPhi$. Therefore, at layer L_1, the two-layer CF can achieve lower hash collisions, and thus fewer cold items will be misreported. Since the average probability that one item accesses layer L_2 is very low (often less than $1/20$ in real data streams when $\delta_1 = 4$), layer L_2 still has low-level hash collisions. Further experiments about the selection of layer number are provided in Sect. 6.2.3.

3.3 Optimization 1: aggregate-and-report

In real data streams, some items will appear many times over consecutive points in time [52,53]. This is called a *stream burst*, which provides an opportunity to accelerate CF. For this situation, we propose a strategy called aggregate-and-report. The key idea of this strategy is to *add another small filter to aggregate the bursting items before CF, and then report the aggregated items and their frequencies (often much larger than one) to CF under certain conditions.* This small filter can be implemented in different ways. A typical one is what the Augmented sketch does: it scans the whole queue and expels the item with the minimum frequency. However, this method suffers from low speed when the queue is large. Worse, it needs two-direction processing—frequent exchanges between the filter and the sketch behind it, which is costly. In contrast, we implement a one-direction filter by using a modified lossy hash table [54]: each item is hashed into a bucket, and each bucket consists of several items and their corresponding frequencies. We use Single Instruction Multiple Data (SIMD) [55] to scan a specific bucket. The detailed procedure of bucket scan is shown in Section A in our conference paper [56].

Figure 3 shows the data structure of our implementation of aggregate-and-report. There are d_b buckets, each bucket consists of w_c cells, and each cell is used to store a Key-Value pair. The part of the data structure handling the key records the item ID, while one dealing with the value records the aggregated frequency that has been accumulated during a time window when the corresponding item resides in this bucket. For each incoming item, we use a hash function to locate a bucket. Within this hashed bucket, we perform the *bucket scan* operation:

1. If the key part of one cell matches the ID of the incoming item, we increment the corresponding value part;

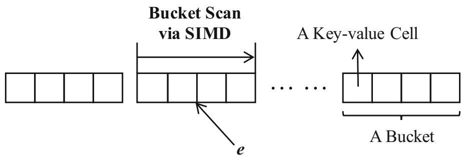


Fig. 3 SIMD-based bucket scan

[3] $f_{e_t}[t-1]$ is the frequency of item e_t before updating, and $f_{e_t}[t] = f_{e_t}[t-1] + 1$.

2. Otherwise, if there are available cells, we insert the current item with frequency of 1 into the new cell;
3. Otherwise, we expel one cell of this bucket in a global round-robin fashion (across the d_b buckets): replace the key part of this cell with the ID of the incoming item, and set the value part of this cell to 1. The expelled item with its aggregated frequency from the bucket will be inserted into CF.

Also, at the end of each time window, we need to flush all items of all buckets into CF. Let f_{agg} be the value of reported aggregated frequency of an arbitrary item. Since f_{agg} is often larger than one, we need to make some modifications to Algorithm 1. Recall that V_1 and V_2 denote the minimum value of the hashed counters at the two layers, respectively. Algorithm 2 shows the procedure for CF to process the reported item with its aggregated frequency from the aggregate-and-report phase. The principle of the modified algorithm is unchanged. The difference lies in how CF with aggregate-and-report strategy reaches the concurrent overflow state. Recall that we need to estimate whether an item's frequency exceeds the threshold $\mathcal{T}$. We split $\mathcal{T}$ to be $\mathcal{T}_1 + \mathcal{T}_2$, which denotes the thresholds of the two layers, respectively; and V_1, V_2 denote the minimum value of the hashed counters in the two layers, respectively. We need to check whether $f_{agg} > \mathcal{T}_1 - V_1$ or $f_{agg} - (\mathcal{T}_1 - V_1) > \mathcal{T}_2 - V_2$ to determine whether the hashed counters concurrently overflow at layer L_1 or L_2. Note that the item frequency reported to the specific stream processing algorithm (the CM–CU sketch, Space-Saving, FlowRadar, etc) by CF with aggregate-and-report strategy is $f_{agg} - (\mathcal{T}_1 - V_1) - (\mathcal{T}_2 - V_2)$, instead of the default of 1 in Algorithm 1.

Algorithm 2: Stream processing algorithm for CF with aggregate-and-report strategy.

```
Input: The reported item $e_t$ with its aggregated frequency $f_{agg}$.
Output: Update CF, and return whether $f_{e_t}[t] > \mathcal{T}$.
$V_1 \leftarrow$ Estimate($L_1, e_t$);
if $V_1 + f_{agg} \leqslant \mathcal{T}_1$ then
    ConservativeUpdate($L_1, e_t, V_1 + f_{agg}$);
    return False;
else
    /* concurrently overflow at layer $L_1$ */
    ConservativeUpdate($L_1, e_t, \mathcal{T}_1$);
    $\Delta_1 \leftarrow \mathcal{T}_1 - V_1$;
    $f_{agg} \leftarrow f_{agg} - \Delta_1$;
    $V_2 \leftarrow$ Estimate($L_2, e_t$);
    if $V_2 + f_{agg} \leqslant \mathcal{T}_2$ then
        ConservativeUpdate($L_2, e_t, V_2 + f_{agg}$);
        return False;
    else
        /* concurrently overflow at layer $L_2$ */
        ConservativeUpdate($L_2, e_t, \mathcal{T}_2$);
        return True;
```

3.4 Optimization 2: one-memory-access

Each incoming item needs to access layer L_1, and a few items need to access layer L_2. Accessing layer L_1 requires d_1 memory accesses and hash computations and is likely to become the bottleneck of the system. To handle this, we propose a *one-memory-access* strategy tailored for layer L_1 only. Before discussing this strategy, we introduce one critical fact. In our implementation, we set the size of each counter at the lower layer δ_1 to 4 (with $\mathcal{T}_1$ of 15) and adjust δ_2 at the higher layer to accommodate the threshold $\mathcal{T}$ required by the specific stream processing algorithm (e.g.CM–CU sketch, Space-Saving, FlowRadar). Therefore, for the lower layer, a machine word of 64 bits contains 16 counters. 16 is often three or more times d_1. Based on this, our one-memory-access strategy contains the following two parts: (1) we confine the d_1 hashed counters within a machine word of W bits to reduce memory accesses; (2) we use only one hash function to locate the d_1 hashed counters and thus reduce the hash computations. Specifically, we split the value produced by a hash function into multiple segments, and each segment is used to locate a machine word or a counter. For example, for layer L_1 with $w_1 = 2^{20}$, $\delta_1 = 4$ and $d_1 = 3$ (memory usage is 1 MB), we split a 32-bit hash value into four segments: one of 16 bits, and three of 4 bits (discarding the remaining 4 bits). We use the 16-bit value to locate one machine word at layer L_1, and the three 4-bit values to locate three counters within this machine word (containing $16 = 2^4$ counters). In practice, a 64-bit hash value is always enough.

In this way, OMA method can achieve the desired speed improvements. However, we still need to make a more careful discussion about collision. The probability of collision will be higher when we implement OMA method. For example, in OMA method, if we set $d_1 = 3$, $\delta_1 = 4$, and the hash function results for two items e_1 and e_2 is x_{11}, x_{12}, x_{13} and x_{21}, x_{22}, x_{23}, respectively. In this case, e.g., if x_{12} and x_{23} is the same location, then all of the six hash results are limited to 16 counters, the probability of more collisions is quite high. If we use $d_1 = 3$ independent hash functions, then even if $x_{12} = x_{23}$, the probability of more collisions will be low. However, according to our experiment, OMA has little influence on accuracy at our experiment settings. It is shown in Figs. 4 and 5.

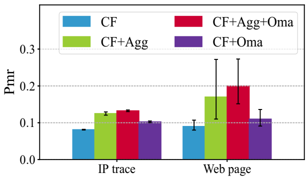
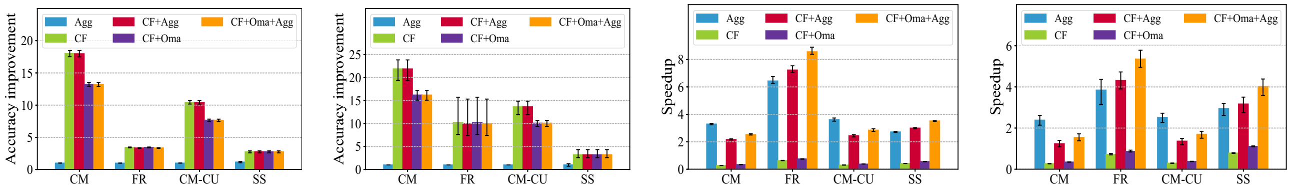


Fig. 4 Impact of different optimizations on P_{mr}



(a) Varying optimization (IP trace).

(b) Varying optimization (web page).

(c) Varying optimization (IP trace).

(d) Varying optimization (web page).

Fig. 5 Impact of different optimizations on the accuracy and speed of Count-Min, CM–CU, Space-Saving and FlowRadar

4 Cold filter implementation

In this section, we show how to implement the cold filter framework for two types of stream processing tasks. Specifically, we show how to process frequency estimation, finding heavy hitters and finding heavy changes of frequency-based tasks, and finding persistent items of persistency-based tasks.

4.1 Frequency-based tasks

4.1.1 Estimating item frequency

Key idea For frequency estimation, we use a CF to record the frequencies of cold items, and a sketch Φ (e.g., the Count-Min sketch, the CM–CU sketch) to record the frequencies of hot items.

Insertion When inserting an item, we first update the CF as described earlier. If the hashed counters concurrently overflow at both layers before this insertion, we employ sketch Φ to record the remaining frequency of this item.

Query Recall that V_1 and V_2 denote the minimum value of the hashed counters at the two layers, respectively. Let V_ϕ be the query result of sketch Φ. When querying an item, we have three cases: (1) if the hashed counters do not concurrently overflow at layer L_1 ($V_1 < \mathcal{T}_1$), we report V_1; (2) if the hashed counters concurrently overflow at layer L_1 ($V_1 = \mathcal{T}_1$), but not at layer L_2 ($V_2 < \mathcal{T}_2$), we report $V_1 + V_2$; (3) otherwise, we report $V_1 + V_2 + V_\phi$.

Discussion Here, we discuss why the sketch with CF can achieve higher accuracy than the standard sketch. Conventional sketches used for estimating item frequency do not differentiate cold from hot items. They use counters of a fixed size determined by the largest expected frequency. As hot items are much fewer than cold ones in real data streams, the high bits of most counters will be wasted (memory inefficiency). If we use CF to approximately differentiate cold from hot items, then we can exploit the skew in popularity in the counters. For hot items, we use another sketch with large counters to record its frequency. For cold items, CF with small counters provides more accurate estimation, as it leverages a similar updating strategy as the CM–CU sketch while containing many more counters. By employing the counters with different sizes to do the counting, we can guarantee the memory efficiency, and thus improve the accuracy.

Other sketches for frequency estimation, such as Counter Braids (CB) [57], also leverage the skewness to achieve high space efficiency. CB uses several counters for each flow, and it allocates fewer counters for “more significant bits”. To get the frequency of an item, CB uses an expensive decoding procedure to get all frequencies, and it needs all item IDs, which is not realistic in some scenarios. Compared with CB, our CF does not need all item IDs and supports queries for a single item.

4.1.2 Finding top-*k* hot items

Prior art There are two kinds of approaches to find top-k hot items: sketch-based and counter-based [7]. Sketch-based methods use a sketch (e.g., the Count-Min sketch [9], the CM–CU sketch [29], and others [11]) to count the frequency of each item in data streams, and a min-heap of size k to maintain the top-k hot items. The prominent counter-based methods include Lossy Counting [58], the Frequent algorithm [59,60], and Space-Saving [10]. In this paper, we focus on Space-Saving, as it is the most popular. Space-Saving maintains a data structure called *Stream-Summary* that consists of H ($H \geqslant k$) item-counter pairs. For each incoming item e, if e has already been monitored by the Stream-Summary, it increments its corresponding counter. Otherwise, it inserts e into the Stream-Summary if there is available space. If there is no space available, it creates new space by expelling the item with the minimum count ($C_{\min}$) from the Stream-Summary, and stores e with count of $C_{\min} + 1$ at its place. During queries, Space-Saving returns the top-k hot items from the Stream-Summary according to their recorded frequencies (i.e., the values in their counters).

Key idea To enhance the performance of Space-Saving, we use a CF to prevent the large number of cold items from accessing the Stream-Summary.

Insertion When inserting an item, we first update CF as described before. If the hashed counters concurrently overflow at both layers before this insertion, we feed this item to Space-Saving.

Report Below, we show how to report the top-k hot items. After processing all the items, we get the IDs and recorded frequencies of the top-k hot items from the Stream-Summary. Their estimated frequencies will be the corresponding recorded frequencies plus $\mathcal{T}$. For the above procedures, we need to guarantee that the frequency of the kth hottest item is larger than the threshold $\mathcal{T}$. In practice, to get the kth largest frequency, we use the kth largest frequencies from previous measurement periods to predict the kth largest frequency of the current measurement period, using *EWMA* [61], with some history weight. The larger $\mathcal{T}$ is, the better we expect the results to be. Therefore, we set $\mathcal{T} = \mathcal{F} \times \alpha$ $(\alpha \to 1)$, where $\mathcal{F}$ is the predicted frequency.

Discussion Now, we discuss why Space-Saving with CF can achieve higher accuracy than standard Space-Saving. Standard Space-Saving processes each item identically: each incoming item needs to be fed to the Stream-Summary. Unfortunately, the large number of cold items will lead to many unnecessary exchanges in the Stream-Summary, over-estimating the recorded frequencies, since each exchange leads to one increment operation in the counter associated with the expelled item. Over-estimation of frequencies further leads to many incorrect exchanges in the Stream-Summary. As a results, the accuracy of standard Space-Saving will degrade. If we use CF to filter out the large number of cold items, fewer incorrect exchanges will occur in the Stream-Summary, and the accuracy of both recorded frequencies and Space-Saving can be improved.

4.1.3 Detecting heavy changes

Prior art Heavy changes refer to the items that experience abrupt changes of frequencies between two consecutive time windows. We also call these items the *culprit items*. We assume the data stream during the first time window has frequency vector $\mathbf{f}_1 =< f_{1e_1}, f_{1e_2}, \ldots, f_{1e_L} >$ where f_{1e_i} denotes the frequency of item e_i (picked from the universe $U = \{e_1, e_2, \ldots, e_L\}$). Similarly, we have $\mathbf{f}_2 =< f_{2e_1}, f_{2e_2}, \ldots, f_{2e_L} >$ during the second time window. For item e_i, if $|f_{1e_i} - f_{2e_i}| \geqslant \phi \cdot D$, where ϕ is a predetermined threshold and $D = \sum_{j=1}^{L} |f_{1e_j} - f_{2e_j}|$, it is called a heavy change. Note that computing D, the L_1 difference of $\mathbf{f}_1$ and $\mathbf{f}_2$, is a well-studied problem [62]. The k-ary sketch [63] can efficiently capture the difference between $\mathbf{f}_1$ and $\mathbf{f}_2$, but requires a second pass to obtain the IDs of culprit items. The reversible sketch [64], based on the k-ary sketch, can infer the IDs of culprit items with time complexity of $O(L^{0.75})$, which depends on the ID space of items, and could be large in practice. Therefore, we will not focus on it in this paper. Recent work—FlowRadar [32], encodes each distinct item and its frequency in an extended IBLT (Invertible Bloom Lookup Table) [33] with the aid of a Bloom filter [16] and decodes them in time complexity of $O(n)$ where n is the number of distinct items. When the number of hash functions used in the extended IBLT is set to 3, FlowRadar can decode all the items with a very high probability, if $m_c > 1.24n$ where m_c is the number of cells in IBLT. Obviously, FlowRadar can be used for detecting heavy changes, by comparing the two decoded item sets.

Key idea To enhance the performance of FlowRadar, we use a CF to prevent the large number of cold items from accessing FlowRadar.

Insertion During the first time window, when inserting an item, we first update CF as described before. If the hashed counters concurrently overflow at both layers before this insertion, this item needs to be inserted into FlowRadar. At the end of this time window, we employ new instances of CF and FlowRadar. The insertion process during the second time window is the same as during the first time window.

Report Below, we show how to report heavy changes. At the end of the second time window, we decode the two IBLTs associated with each time window in FlowRadar. Let S_1 and $\mathbf{f}_1^I$ be the item set and frequency vector decoded from the first IBLT, respectively. For each item, $e \in S_1$, f_{1e}^I is its recorded frequency in IBLT. Similarly, we get S_2 and $\mathbf{f}_2^I$ from the second IBLT. Recall that V_1 and V_2 denote the minimum value of the hashed counters at the two layers in CF, respectively. For an arbitrary item $e \in S_1 \cup S_2$, we define the function $Q_1(.)$ for the first CF as follows: (1) if the hashed counters do not concurrently overflow at layer L_1 $(V_1 < \mathcal{T}_1)$, $Q_1(e) = V_1$; (2) otherwise, $Q_1(e) = V_1 + V_2$. Similarly, we define $Q_2(.)$ for the second CF. The procedure for detecting heavy changes is shown in Algorithm 3. Note that we need to guarantee $\mathcal{T} \leqslant \phi \times D$. Given this constraint, (1) items that do not access FlowRadar in either time window have their frequencies in both time windows definitely lower than or equal to $\mathcal{T}$, and thus their frequency changes will not exceed $\mathcal{T} \leqslant \phi \times D$; (2) the IDs and frequencies of the other items in the two time windows can be answered by IBLTs and CFs.

Discussion Now, we discuss why FlowRadar with CF requires less memory than the standard FlowRadar. According to the literature [32], the memory usage of the IBLT in FlowRadar should be proportional to the number of distinct items it records. Therefore, the large number of distinct cold items will incur a significant memory consumption for the standard FlowRadar. If we use CF to filter out the cold items, the number of distinct items that FlowRadar needs to record will be largely reduced, and much memory can be saved.

Algorithm 3: Detecting heavy changes.

```
Input: $S_1, \mathbf{f}_1^I, Q_1(.)$ and $S_2, \mathbf{f}_2^I, Q_2(.)$.
Output: The culprit item set $C$.
$C \leftarrow \emptyset$;
foreach $e \in S_1 \cup S_2$ do
    /* $f_{1e}$: e's frequency in the first time window */
    if $e \in S_1$ then $f_{1e} \leftarrow f_{1e}^I + \mathcal{T}$ else $f_{1e} \leftarrow Q_1(e)$ /* $f_{2e}$: e's frequency in the second time window */
    if $e \in S_2$ then $f_{2e} \leftarrow f_{2e}^I + \mathcal{T}$ else $f_{2e} \leftarrow Q_2(e)$ if $|f_{1e} - f_{2e}| \geqslant \phi \cdot D$ then $C \leftarrow C \cup \{e\}$
return $C$;
```

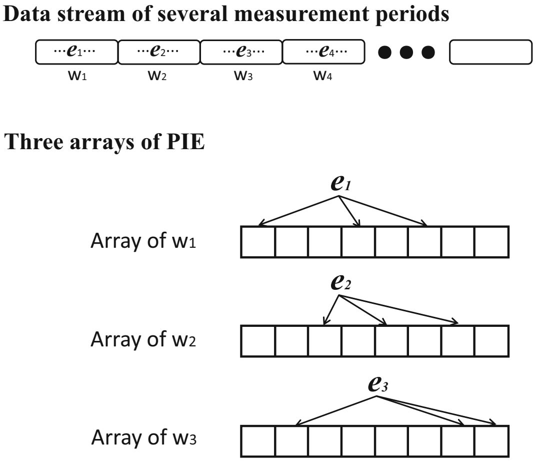


Fig. 6 PIE algorithm

4.2 Persistency-based tasks

4.2.1 Finding persistent items

Prior art In this paper we use the definition of persistent items from [35]. Given a data stream $\mathcal{S}$ consisting of $\mathcal{T}'$ continuous equally sized measurement periods (*periods for short*), if an item e occurs in x periods, then x is the occurrence of e. If an item appears many times but only in one period, its occurrence is 1. An item e is persistent if and only if its occurrence over time is no less than a predefined threshold $\mathcal{T}$, and when these conditions are met, we call e a *persistent item. Finding persistent items* is to report all items whose occurrence is no less than $\mathcal{T}$.

The state-of-the-art algorithm for finding persistent items is called PIE [35]. Below we present how PIE works (Fig. 6). Given a data stream $\mathcal{S}$ consisting of $\mathcal{T}'$ periods, PIE builds one array for each period; then there will be $\mathcal{T}'$ equally sized arrays in total. Each array consists of w cells, and each cell consists of three fields: flag, Raptor code [36] and a fingerprint. During the recording the stream, for an incoming item e, like Bloom filters [16], PIE computes k hash functions of e, and gets k cells in the array of the current period, and we call them ***the** k **mapped cells*** for convenience. If a mapped cell is empty, PIE computes its fingerprint v using a hash function and computes a Raptor code α. Then, PIE sets the cell to $< v, \alpha, true >$. If a mapped cell is not empty, PIE will compare e's fingerprint v with the fingerprint stored in the cell. If the two fingerprints are the same, PIE does nothing; otherwise, PIE sets the flag to false. A cell with a flag of false means that this cell has collisions and is useless. In the decoding phase, PIE searches all cells in the $\mathcal{T}'$ arrays. Given an item e, it will be decoded successfully if and only if there are enough number of fingerprints that are the same as e's fingerprint. Otherwise, e cannot be decoded successfully. For a non-persistent item, it can hardly be successfully decoded, because there are a very few Raptor codes recorded in the arrays. For example, if an item e appears many times but *in only one period*, all the $\mathcal{T}'$ arrays at most record only 1 Raptor code. In this case, e cannot be decoded. In contrast, for a persistent item, it could have Raptor codes stored in cells in many arrays and thus could be decoded successfully with a much higher probability. In real data streams, as mentioned above, the frequency distribution of items is very skewed. In other words, most items occur only a few times (e.g., less than 4). Such cold items will be hashed to many cells, leading to many invalid cells. As a consequence, the number of successful decoding will be sharply reduced if using the original PIE algorithm.

Key idea Our observation is that those cold items, which are very cold (e.g., with frequency less than 3 or 5), are definitely not persistent items. Non-persistent items should not be inserted into PIE, because inserting non-persistent items into PIE not only degrades the accuracy, but also degrades the encoding and decoding speed. Therefore, our approach uses cold filter to filter very cold items in advance, discarding these items directly.

Next, we show the data structure, insertion, report details of our implementation of CF to the PIE algorithm. As the operation details of PIE have been presented in the beginning of this subsection, we omit the insertion and decoding operations of PIE below.

Data structure (Fig. 7) Our data structure consists of two parts: a cold filter and a PIE structure. In this implementation, we only need an one-layer cold filter, as only very cold items have to be filtered. The counter in the cold filter is small, e.g., 2 or 4 bits. We do not employ the Aggregate-and-report technique in this application, because it may aggregate items across multiple time periods. In other words, the simplified CF framework is an array associated with k hash functions. For each insertion, the k mapped counters that are confined into one machine word are incremented: only the smallest counter(s) will be incremented by 1. If some counters reach the maximum value, it will stay unchanged when mapped again.

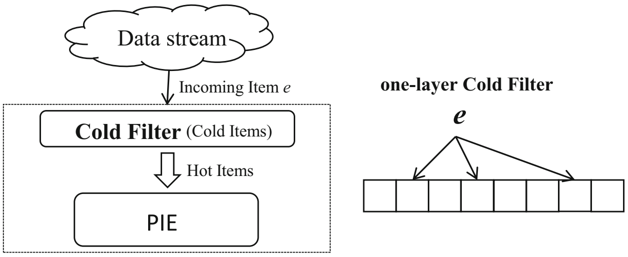


Fig. 7 Deployment Cold Filter on the PIE algorithm

Insertion Recall that the cold filter (CF) can report whether the frequency of an incoming item exceeds the predefined threshold $\mathcal{T}$. Therefore, when inserting an item e, we first update CF as described before. If CF reports that the frequency of e exceeds $\mathcal{T}$, we insert e into the PIE. Otherwise, we discard e.

Report After processing all the items in the data stream, we get a CF and a PIE containing many fingerprints and Raptor codes.We want to decode persistent items, not non-persistent ones. All the successfully decoded items will be reported as persistent.

Discussion Next, we discuss why PIE with CF can achieve higher accuracy than the standard PIE. The standard approach inserts all incoming items in one scheme indiscriminately. When a hash collision happens in a cell, the standard approach simply discards this cell. As discussed before, in practice, items in real data streams often obey skewed distributions. Therefore, although some cold items may be persistent, most cold items are definitely not persistent. As the majority are cold items and PIE inserts all cold items, it makes a significant number of cells invalid. Fortunately, our CF filters very cold items, significantly reducing the number of invalid cells. As a result, after using CF, PIE becomes much more accurate when using a given amount of memory. Besides, the insertion speed also improves because a number of items have been discarded by CF, whose inserting speed is much higher than PIE.

5 Formal analysis

5.1 Performance of CF

Given a time window $[1, E]$, the data stream $\mathcal{S} = (e_1, e_2, \ldots, e_E)$ contains E items with N unique items $e_{\beta_1}, e_{\beta_2}, \ldots, e_{\beta_N}$. Within this time window, we construct a CF with threshold $\mathcal{T}(= \mathcal{T}_1 + \mathcal{T}_2)$. Recall that we need to estimate whether an item's frequency exceeds the threshold $\mathcal{T}$ or not. Before we get into the formal analysis of the performance of CF, we first need to specify the frequency distribution of the items of this data stream.

Definition 1 For each time point $j \in [E]$,[4] let $I_k[j]$ be the subset of items whose current frequency is greater or equal to k. Formally, $I_k[j] = \{e_{\beta_i} | f_{e_{\beta_i}}[j] \geqslant k, i \in [N], k \in Z^+\}$.[5] Let $\Delta_k[j]$ be $I_k[j] - I_{k+1}[j]$. We only assume that the values of $|I_k[E]|, \ \forall k \in N^+$ are known.

Consider the time window $[1, E]$. The hot items whose frequencies $f_e[E]$ are larger than $\mathcal{T}$ will finally be identified as hot items. The only error CF makes is in letting some cold items whose frequencies $f_e[E]$ are smaller or equal to $\mathcal{T}$ "pass". We say that these items are *misreported* to the specific stream processing algorithms. Let I_{mr} be the subset of misreported items. To characterize the filtering performance of CF formally, we define the misreported rate P_{mr} as follows:

$$P_{\text{mr}} = |I_{\text{mr}}| / (|I_1[E]| - |I_{\mathcal{T}+1}[E]|). \tag{1}$$

For the P_{mr} of CF (without optimizations), we first focus on analyzing the CM–CU sketch and then use the analysis to handle the two-layer CF. Below, we first use the known properties of standard Bloom filters to derive the P_{mr} of CM–CU.

The reason we use P_{mr} here is that the misreport rate is the direct metric for the effectiveness of CF itself, and it is not related to the specific task. If the misreport rate is high, then a lot of cold items will be regarded as hot items, which means Cold Filter does not successfully filter cold items. And the accuracy for specific tasks is expected to be low.

5.2 Standard Bloom filter

A standard Bloom filter [16] can tell whether an item appears in a set. It is made of a w-bit array associated with d hash functions. When inserting an item, it uses the d hash functions to locate d *hashed bits*, and sets all these bits to one. When querying an item, if all d hashed bits are one, it reports true; otherwise, it reports false. We use n to denote the number of items in the given set. The standard Bloom filter only has false positives, no false negatives. It may report true for some items that are not in the set, but never reports false for an item that is in the set. Given w, d and n, the *false positive rate* P_{fp} of a standard Bloom filter is known to be:

$$P_{\text{fp}}(w, d, n) = \left[1 - \left(1 - \frac{1}{w}\right)^{nd}\right]^d \approx \left(1 - e^{-\frac{nd}{w}}\right)^d \tag{2}$$

We have the following lemma:

Lemma 1 $\overline{P}_{\text{fp}}(w, d, x) = \frac{1}{x} \sum_{i=0}^{x-1} P_{\text{fp}}(w, d, i), \ \forall x \in N^+$ *is a monotonic increasing function of* x.

[4] $[E] = \{1, 2, \ldots, E\}$.

[5] Z^+ is the set of non-negative integers.

The detailed proof is provided in Section B.1 in our conference paper [56].

5.3 Multi-layer Bloom filter

To bridge the Bloom filter with CM–CU, we introduce a new data structure called *multi-layer Bloom filter*, used to estimate item frequency. The multi-layer Bloom filter is an array of standard Bloom filters with the same w, d and hash functions. Each Bloom filter has its *level*, equal to its index in the array from 1 to λ. When inserting an item, we check whether the level-1 Bloom filter reports true: (1) if it reports false, we just set the d hashed bits in the level-1 Bloom filter to one, and the insertion ends; (2) if it reports true, we need to check whether the level-2 Bloom filter reports true, and rely on the result to determine whether we should end the insertion or continue to check the level-3 Bloom filter. Such operations will continue until the Bloom filer at level λ' reports false, and we set the d hashed bits at this level to one. When querying an item, we find the last (from low to high level) Bloom filter that reports true for this item. Then, we report the level of this Bloom filter as the estimated frequency of this item.

5.4 Equivalence between multi-layer Bloom filter and CM–CU

The multi-layer Bloom filter is equivalent to CM–CU if they have the same hash functions and $w = w_1$, $d = d_1$, $\lambda = 2^{\delta_1} - 1$. Therefore, if we want to analyze the misreporting rate of CM–CU, we can rely on the one of the multi-layer Bloom filter.

For each time point $j \in [E]$, let $(\hat{f}_{e_{\beta_1}}[j], \hat{f}_{e_{\beta_2}}[j], \ldots, \hat{f}_{e_{\beta_N}}[j])$ be the current estimated frequencies (of all distinct items) reported by the multi-layer Bloom filter.

Definition 2 For each time point $j \in [E]$, let $J_k[j]$ be the subset of items such that each item's current estimated frequency is larger than or equal to k. Formally, $J_k[j] = \{e_{\beta_i} | \hat{f}_{\beta_i}[j] \geqslant k, i \in [N], k \in Z^+\}$.

Lemma 2 *The item subsets* $I_k[j]$ *and* $J_k[j]$, *defined in Definition 1 and 2, have the following relation:*

$$\begin{aligned} |I_k[j]| &\leqslant |J_k[j]| \\ &\leqslant |I_k[j]| \\ &\quad + \sum_{i=1}^{k-1} \left[(|I_i[j]| - |I_{i+1}[j]|) \times \prod_{u=1}^{i} \overline{P}_{\rm fp}(w, d, |J_{l_u}[j]|) \right] \end{aligned} \tag{3}$$

where $l_1, \ldots, l_u, \ldots, l_{k-1}$ *is the permutation of* $1, 2, \ldots, k-1$ *that makes sequence* $\overline{P}_{\rm fp}(w, d, |J_{l_u}[j]|)$ *in descending order. Note that the upper bound is an expectation, but not deterministic.*

The detailed proof is provided in Section B.2 in our conference paper [56].

The detailed proof is provided as follow.

Proof The lower bound $|J_k[j]| \geqslant |I_k[j]|$ can be easily observed. Given an incoming item e, in most cases, the Bloom filter reports 0 (false) and 1 (true) before and after inserting e, which is equivalent to incrementing the estimated frequency of e by one. Therefore, we have: $|J_k[j]| \geqslant |I_k[j]|$.

The upper bound of $|J_k[j]|$ is related to the function $\overline{P}_{\rm fp}()$, which is called *past false positive rate*. This function solves the following problem: given an initial standard Bloom filter with w and d, and x distinct items, for each coming distinct item, we check whether the Bloom filter reports true for this item, and then insert it into the Bloom filter regardless. After processing all the x distinct items, how many distinct items are expected to be reported true by the Bloom filter? The answer is $\overline{P}_{\rm fp}(w, d, x) \times x$. We consider the false positive rate of every item. For the ith $(1 \leqslant i \leqslant x)$ incoming item, its false positive rate equals to $P_{\rm fp}(w, d, i-1)$, since these are total $i-1$ items inserted into the Bloom filter previously. Therefore, the total number of distinct items that are expected to be reported true is $\sum_{i=0}^{x-1} P_{\rm fp}(w, d, i) = \overline{P}_{\rm fp}(w, d, x) \times x$.

We consider the specific process of inserting all E items into the multi-layer Bloom filter. The contributions to $J_k[j]$ derive from the following k parts: (1) items in set $I_k[j]$; (2) items in set $I_{k-1}[j] - I_k[j]$ and experiencing one or more false positives from level 1 to level $k-1$; (3) items in set $I_{k-2}[j] - I_{k-1}[j]$ and experiencing two or more false positives from level 1 to level $k-1$; ...; k) items in set $I_1[j] - I_2[j]$ and experiencing $k-1$ false positives from level 1 to level $k-1$. For the first part, the set size is $|I_k[j]|$; for the second part, the set size is $|I_{k-1}[j]| - |I_k[j]|$, and the *maximum* probability of experiencing such false positives is $\overline{P}_{\rm fp}(w, d, |J_{l_1}[j]|)$; For the third part, the set size is $|I_{k-2}[j]| - |I_{k-1}[j]|$, and the *maximum* probability of experiencing such false positives is $\overline{P}_{\rm fp}(w, d, |J_{l_1}[j]|) \times \overline{P}_{\rm fp}(w, d, |J_{l_2}[j]|)$; ...; For the kth part, the set size is $|I_1[j]| - |I_2[j]|$, and the probability of experiencing such false positives is $\prod_{u=1}^{k-1} \overline{P}_{\rm fp}(w, d, |J_{l_u}[j]|)$; Considering all the above k parts, the lemma holds. □

Since $\overline{P}_{\rm fp}(w, d, x)$ is a monotonically increasing function of x, $|J_k[j]|$ can be bounded recursively by this lemma. Let $|J_k[j]|^L$ and $|J_k[j]|^U$ be the lower and upper bound of $|J_k[j]|$, respectively.

In addition, we derive an upper bound of $|J_k[j]| - |I_k[j]|$ in probabilistic version, in other words, a confidence interval of it.

$|J_k[j]| - |I_k[j]|$ increases only if there is a false positive error in the Bloom filter. Therefore, $|J_k[j]| - |I_k[j]|$ is bounded by the number of false positive error defined as $N_{\rm fp}$.

$N_{\rm fp}$ is a random variable with values from 0 to n, where n is the number of distinct items. $N_{\rm fp}$ is proved to obey the binomial distribution $\mathbf{B}(n, P_{\rm fp}(w, d, n))$, where $P_{\rm fp}(w, d, n)$

is the *false positive rate*. Given the confidence α, the confidence interval of $|J_k[j]|-|I_k[j]|$ is included in the confidence interval of N_{fp}. Thus, we can derive an α-confidence upper bound of $|J_k[j]|-|I_k[j]|$, given w, d, n.

5.5 Bound of P_{mr} of multi-layer Bloom filter ($\lambda = \mathcal{T}$)

Generally, its P_{mr} is associated with the distribution of the appearance order of each item in the whole data stream. We can use Gaussian, Poisson or other distributions to model it. We employ the *random order model* [65,66] defined as follows:

Definition 3 (*Random order model*) Let $\mathcal{P}$ be an arbitrary frequency distribution over distinct item set $U = \{e_{\beta_1}, e_{\beta_1}, \ldots, e_{\beta_N}\}$. At each time point, the incoming item in the stream is picked independently and uniformly at random from U according to $\mathcal{P}$.

Theorem 1 *Under the random order model, the misreporting rate of the multi-layer Bloom filter is bounded by:*

$$P_{\text{mr}} \geqslant \frac{\sum_{k=1}^{\lambda}\left\{\left[1-\prod_{u=1}^{k}\left(1-\prod_{i=u}^{\lambda} P_{\text{fp}}(|J_{l_i}[t_u]|^{L})\right)\right]\times|\Delta_k[E]|\right\}}{|I_1[E]|-|I_{\lambda+1}[E]|}$$
$$P_{\text{mr}} \leqslant \frac{\sum_{k=1}^{\lambda}\left\{\left[1-\prod_{u=1}^{k}\left(1-\prod_{i=1}^{\lambda-u+1} P_{\text{fp}}(|J_{l_i}[t_u]|^{U})\right)\right]\times|\Delta_k[E]|\right\}}{|I_1[E]|-|I_{\lambda+1}[E]|} \tag{4}$$

where $l_1, \ldots, l_i, \ldots, l_\lambda$ is the permutation of $1, 2, \ldots, \lambda$ that makes sequence $P_{\text{fp}}(w, d, |J_{l_i}[t_u]|)$ in descending order, and $|J_{l_i}[t_u]|^L$ and $|J_{l_i}[t_u]|^U$ can be calculated using Eq. 3 and

$$|I_{l_i}[t_u]| = \frac{(2u-1)}{2k}\times|I_{l_i}[E]|\ (1\leqslant i\leqslant\lambda, 1\leqslant u\leqslant k\leqslant\lambda) \tag{5}$$

The detailed proof is provided in Section B.3 in our conference paper [56].

5.6 Bound of P_{mr} of CF

For a two-layer CF, since it has two distinct layers with different parameter settings, we define a unified function for its false positive rates at different layers.

Definition 4 For each time point $j \in [E]$,

$$P_{pf}^{U}(|J_x[j]|) = \begin{cases} P_{\text{fp}}(w_1, d_1, |J_x[j]|) & (x\in[1,\mathcal{T}_1]) \\ P_{\text{fp}}(w_2, d_2, |J_x[j]|) & (x\in[\mathcal{T}_1+1,\mathcal{T}]) \end{cases}$$
$$\overline{P}_{pf}^{U}(|J_x[j]|) = \begin{cases} \dfrac{1}{|J_x[j]|}\displaystyle\sum_{i=0}^{|J_x[j]|-1} P_{\text{fp}}(w_1, d_1, i) & (x\in[1,\mathcal{T}_1]) \\ \dfrac{1}{|J_x[j]|}\displaystyle\sum_{i=0}^{|J_x[j]|-1} P_{\text{fp}}(w_2, d_2, i) & (x\in[\mathcal{T}_1+1,\mathcal{T}]) \end{cases} \tag{6}$$

where $|J_k[j]|$ is bounded in the following lemma:

Lemma 3 *The item subsets $I_k[j]$ and $J_k[j]$, defined in Definition 1 and 2, have the following relation:*

$$\begin{aligned}|I_k[j]| &\leqslant |J_k[j]| \\ &\leqslant |I_k[j]| + \sum_{i=1}^{k-1}\left[(|I_i[j]|-|I_{i+1}[j]|)\right. \\ &\quad \left.\times\prod_{u=1}^{i}\overline{P}_{\text{fp}}^{U}(|J_{l_u}[j]|)\right]\end{aligned} \tag{7}$$

where $l_1, \ldots, l_u, \ldots, l_{k-1}$ is the permutation of $1, 2, \ldots, k-1$ that makes sequence $\overline{P}_{\text{fp}}^{U}(|J_{l_u}[j]|)$ in descending order.

The detailed derivation process for this lemma is similar to the one of Lemma 2; hence, we omit it. Similarly, we can get $|J_k[j]|^L$ and $|J_k[j]|^U$ from this lemma.

Theorem 2 *Under the random order model, the misreporting rate of a two-layer CF is bounded by:*

$$P_{\text{mr}} \geqslant \frac{\sum_{k=1}^{\mathcal{T}}\left\{\left[1-\prod_{u=1}^{k}\left(1-\displaystyle\prod_{i=u}^{\lambda}\overline{P}_{\text{fp}}^{U}(|J_{l_i}[t_u]|^{L})\right)\right]\times|\Delta_k[E]|\right\}}{|I_1[E]|-|I_{\lambda+1}[E]|}$$
$$P_{\text{mr}} \leqslant \frac{\sum_{k=1}^{\mathcal{T}}\left\{\left[1-\displaystyle\prod_{u=1}^{k}\left(1-\prod_{i=1}^{\lambda-u+1}\overline{P}_{\text{fp}}^{U}(|J_{l_i}[t_u]|^{U})\right)\right]\times|\Delta_k[E]|\right\}}{|I_1[E]|-|I_{\lambda+1}[E]|} \tag{8}$$

where $l_1, \ldots, l_i, \ldots, l_\lambda$ is the permutation of $1, 2, \ldots, \lambda$ that makes the sequence $P_{\text{fp}}(w, d, |J_{l_i}[t_u]|)$ in descending order, and $|J_{l_i}[t_u]|^L$ and $|J_{l_i}[t_u]|^U$ can be calculated by Eq. 7 and

$$|I_{l_i}[t_u]| = \frac{(2u-1)}{2k}\times|I_{l_i}[E]|\ (1\leqslant i\leqslant\lambda, 1\leqslant u\leqslant k\leqslant\lambda) \tag{9}$$

The detailed derivation process for this theorem is similar to Theorem 1, and we omit it.

5.7 Analysis of CM–CU with CF

Now, we give a formal analysis of the CM–CU sketch with CF (with two optimizations) for estimating item frequency. We use the same notations and assumptions employed in Sect. 5.1. We use w_ϕ and d_ϕ to denote the number of counters per array and the number of arrays in the CM–CU sketch (Φ), respectively.

We first consider the error bound of the CM–CU sketch with CF plus one-memory-access strategy and then illustrate how this error bound can be generalized to the aggregate-and-report strategy. We consider the two layers of CF and the CM–CU sketch as three multi-layer Bloom filters with different parameter settings. The literature [19] shows that the Bloom filter with partitioning (e.g., d_ϕ segments in Φ) has the same asymptotic false positive rate as the standard one without partitioning. Here, we consider them as equivalent. Therefore, we define a unified function about its false positive rates of different layers as follows:

Definition 5 For each time point $j \in [E]$,

$$P_{pf}^{\phi\ U}(|J_x[j]|) = \begin{cases} P_{\text{fp}}^{\text{oma}}\left(w_1, d_1, |J_x[j]|, \frac{\text{W}}{\delta_1}\right) \ (1 \leqslant x \leqslant \mathcal{T}_1) \\ P_{\text{fp}}(w_2, d_2, |J_x[j]|) \ (\mathcal{T}_1 + 1 \leqslant x \leqslant \mathcal{T}) \\ P_{\text{fp}}(w_\phi, d_\phi, |J_x[j]|) \ (x \geqslant \mathcal{T} + 1) \end{cases}$$

$$\overline{P}_{pf}^{\phi\ U}(|J_x[j]|) = \begin{cases} \frac{1}{|J_x[j]|} \sum_{i=0}^{|J_x[j]|-1} P_{\text{fp}}^{\text{oma}}\left(w_1, d_1, i, \frac{\text{W}}{\delta_1}\right) \\ \qquad (1 \leqslant x \leqslant \mathcal{T}_1) \\ \frac{1}{|J_x[j]|} \sum_{i=0}^{|J_x[j]|-1} P_{\text{fp}}(w_2, d_2, i) \\ \qquad (\mathcal{T}_1 + 1 \leqslant x \leqslant \mathcal{T}) \\ \frac{1}{|J_x[j]|} \sum_{i=0}^{|J_x[j]|-1} P_{\text{fp}}(w_\phi, d_\phi, i) \\ \qquad (x \geqslant \mathcal{T} + 1) \end{cases} \quad (10)$$

Using the same approach as before, we get $|J_k[j]|^U$.

Theorem 3 *For an arbitrary item* e_{β_i} $(i \in [N])$, *let* $f_{e_{\beta_i}}$ *and* $\hat{f}_{e_{\beta_i}}$ *be its real and estimated frequency during time window* $[1, E]$, *respectively. Let* V *be* $\sum_{i=1}^{N} f_{e_{\beta_i}}$. *Let* $l_1, \ldots, l_u, \ldots$ *be the permutation of* $1, 2, \ldots$ *that makes sequence* $P_{\text{fp}}^{\phi\,U}(|J_{l_u}[E]|^U)$ *in descending order.*

We have the following accuracy guarantee about the CM–CU sketch with CF (with one-memory-access):

$$Pr[\hat{f}_{e_{\beta_i}} - f_{e_{\beta_i}} \leqslant \lceil \varepsilon V \rceil] \geqslant 1 - \prod_{u=1}^{\lceil \varepsilon V \rceil} P_{\text{fp}}^{\phi\,U}(|J_{l_u}[E]|^U) \quad (11)$$

The detailed proof is provided in Section D of [56].

Consider the aggregate-and-report strategy. Obviously, the aggregate-and-report strategy only changes the appearance order of some items. Since in Theorem 3 we pick the appearance order that results in the worst case false positive rate to derive the error bound, this error bound is also applicable to CF with the two optimizations.

5.8 Analysis of PIE with CF

Now we give the brief equations and proof of false negative rate and false positive rate of PIE with CF. Based on the equations and proof, we recommend the approximate optimal values for some parameters. Following is the notations used in the analysis later (Tables 1, 2, 3).

Table 1 Parameters of PIE

Parameters	Descriptions
$\mathcal{T}$	Threshold for item persistence, i.e., an item is persistent if and only if it occurs in t different measurement periods, where $t \geq \mathcal{T}$
k	Number of hash functions for PIE
m	Width of PIE
$\mathcal{T}'$	Number of arrays in PIE, also the number of measurement periods
p	Number of bits of each fingerprint in PIE
r	Number of bits of each Raptor code in PIE

Table 2 Parameters of Cold Filter

Parameters	Descriptions
N	Total number of occurrences of all items during $\mathcal{T}'$ measurement periods
$\mathcal{T}_{\text{CF}}$	For a PIE with CF, set the threshold of CF to be $\mathcal{T}_{\text{CF}}$, *i.e.*, items whose frequency is less than $\mathcal{T}_{\text{CF}}$ are very cold items and are supposed to be discarded by CF

Table 3 Notations In the Computation

Parameters	Descriptions
P_{nc}	Probability that a cell of PIE has no hash collision
P_{m}	Hash-mapping collision survival probability
$P_{\text{df}}(r; l)$	Decoding failure probability when using r bits to encode the ID using Raptor codes
P_{p}	Hash-print collision survival probability
ω_t	Percentage of items occurring in measurement period t
P_{sr}	Expected probability that an item is successfully recovered
P_{FN}	False negative rate of PIE
P_{FP}	False positive rate of PIE
α	Percentage of occurrences that pass CF

5.8.1 False negative rate estimation

First we discuss the representation of α. Recall that an item e whose frequency is equal or larger than threshold $\mathcal{T}_{\mathrm{CF}}$ will definitely pass CF, while an item whose frequency is less than $\mathcal{T}_{\mathrm{CF}}$ may pass CF with probability P_{mr}. So that α can be represented as

$$\alpha = \frac{P_{\mathrm{mr}} \sum_{t=1}^{\mathcal{T}_{\mathrm{CF}}-1} (\omega_t \cdot t) + \sum_{t=\mathcal{T}_{\mathrm{CF}}}^{\mathcal{T}'} (\omega_t \cdot t)}{\sum_{t=1}^{\mathcal{T}'} (\omega_t \cdot t)}$$

In each measurement period, the expected number of occurrences is $N\alpha/\mathcal{T}'$. Thus,

$$P_{\mathrm{nc}} = \left(1 - \frac{1}{m}\right)^{k(N\alpha/\mathcal{T}'-1)} \approx \left(1 - \frac{1}{m}\right)^{kN\alpha/\mathcal{T}'} \tag{12}$$

We use the similar approach from [35] to find a simple expression for describing the hash-print collision survival probability P_{p} that is feasible for parameter optimization. We state the optimization problem as:

$$\text{Minimize } P_{\mathrm{FN}} \text{ such that } m(r + p + 1) = M.$$

The latter equation represents the memory constraint, where M is the size of the entire STBF in bits. It is obvious that the difference between PIE and PIE with CF is the change of total occurrences N, i.e., from N to $N\alpha$, while the other parameters remain unaffected. Consequently, we are able to use the similar derivation to reformulate the optimization problem as:

$$\text{Minimize } (N\alpha \cdot 1/2^p \cdot 1/m \cdot P_{\mathrm{nc}})^k \text{ such that } m(r + p + 1) = M.$$

According to the proofs from [35], we are able to recommend the approximate optimal values for some parameters as follows:

$$r = \left\lceil \frac{\kappa l}{\mathcal{T}} \cdot \left(1 - \frac{1}{m}\right)^{-\frac{kN\alpha}{\mathcal{T}'}} \right\rceil \tag{13}$$

$$p = \left\lfloor \frac{M}{m} \right\rfloor - \left\lceil \frac{\kappa l}{\mathcal{T}} \cdot \left(1 - \frac{1}{m}\right)^{-\frac{kN\alpha}{\mathcal{T}'}} \right\rceil - 1 \tag{14}$$

where κ is a preset constant such as $\kappa = 1.01$ or $\kappa = 1.1$.

5.8.2 False positive rate estimation

There are two situations in which false positives happen: (1) The recovered ID is different from any of the items in PIE. In other words, the recovered ID belongs to a very cold item which is discarded by CF or does not belong to any item in the data stream; (2) The recovered ID is the same as a non-persistent item in PIE. We use P_{FP1} and P_{FP2} to denote the false positive rate of these two scenarios.

Scenario 1 Because we assume that any very cold item cannot be a persistent item, although a very cold item passes CF and enters PIE, its ID cannot be recovered the normal way. Any recovered ID in this situation must come from several different items' Raptor codes. There must be two different cells containing the same fingerprint, and the recovered ID comes from these cells, to which different items are mapped. The probability that fingerprints in such cells are the same is $(1/2^p)^{\mathcal{T}-1}$. To be reported as a persistent item, the Raptor codes and hash-prints from this recovered ID in other $k-1$ cells cannot be different. In other words, other $k-1$ cells are either invalid or filled with the same Raptor codes and hash-prints as those of the recovered ID. Recall that P_{nc} is the probability that no collision happens in a given cell, so the probability of this scenario is

$$\sum_{i=0}^{k-1} \binom{i}{k-1} P_{\mathrm{nc}}^{i} \left(\frac{1}{2^{r+p}}\right)^{i} (1 - P_{\mathrm{nc}})^{k-1-i} \tag{15}$$

The false negative rate for the first scenario is therefore:

$$\begin{aligned} P_{\mathrm{FP1}} &= \left(\frac{1}{2^p}\right)^{\mathcal{T}-1} \sum_{i=0}^{k-1} \binom{i}{k-1} P_{\mathrm{nc}}^{i} \left(\frac{1}{2^{r+p}}\right)^{i} (1 - P_{\mathrm{nc}})^{k-1-i} \\ &= \left(\frac{1}{2^p}\right)^{\mathcal{T}-1} \left(\frac{P_{\mathrm{nc}}}{2} + 1 - P_{\mathrm{nc}}\right)^{k-1} \end{aligned} \tag{16}$$

Scenario 2 In scenario 2, the recovered ID belongs to an item e in PIE which is not a very cold item, but is a non-persistent item which passes CF and enters PIE. Suppose e occurs in t different measurement periods ($1 \le t < \mathcal{T}$), then the group of items must be mixed due to other items mapped to some other cells. The worst case is that the group is mixed due to only one other item. Similarly, we can get the false positive rate for this scenario as follows ($s = \mathcal{T} - 1 - t$).

$$\begin{aligned} P_{\mathrm{FP2}} &= \left(\frac{1}{2^p}\right)^{s} \sum_{i=0}^{s} \binom{i}{s} P_{\mathrm{nc}}^{i} \left(\frac{1}{2^{r+p}}\right)^{i} (1 - P_{\mathrm{nc}})^{s-i} \\ &= \left(\frac{1}{2^p}\right)^{s} \left(\frac{P_{\mathrm{nc}}}{2^{r+p}} + 1 - P_{\mathrm{nc}}\right)^{s} \end{aligned} \tag{17}$$

And the false positive rate FPR for the worst case is shown as follows.

$$P_{\mathrm{FP}} = \max\{P_{\mathrm{FP1}}, P_{\mathrm{FP2}}\} \tag{18}$$

With the equation above, we can derive the bound of some parameters of PIE to achieve a desired false positive rate. Given the upper bound of false positive rate $\overline{P_{\mathrm{FP}}}$, we have

$$p \geq -\frac{\log_2 \overline{P_{\text{FP}}} - \log_2(1 + P_{\text{nc}}/2)^{k-1}}{\mathcal{T} - 1} \tag{19}$$

And under the constraint of $m(r+p+1) = M$, we can select r once p is determined

$$r = \left\lfloor \frac{M}{m} \right\rfloor - p - 1 \tag{20}$$

6 Performance evaluation

In this section, we provide experimental results using cold filter with different algorithms. After introducing the experimental setup in Sect. 6.1, we provide results of frequency-based tasks (Sect. 6.2) and persistency-based tasks (Sect. 6.3) separately since the data structure of cold filter is slightly different in these two types of tasks.

6.1 Experimental setup

6.1.1 Datasets

(1) *IP trace datasets* We use the anonymized IP trace streams collected in 2016 from CAIDA [67]. Each flow is identified by its source IP address (4 bytes). We use the first 256 M packets (items) from this trace, and uniformly divide them into 8 sub-datasets, each of which has around 0.4M distinct items.

(2) *Web page datasets* We downloaded the raw dataset from the website [68]. This dataset is built from a crawled collection of web pages. Each item (4 bytes) records the number of distinct terms in a web page. We use the first 256 M items from the raw dataset and uniformly divide them into 8 sub-datasets, each of which has around 0.9 M distinct items.

These two types of datasets have the same number of items, but different numbers of distinct items, due to their different item frequency distributions. After each of them is divided into 8 sub-datasets, the two types of sub-datasets have different numbers of distinct items, 0.4 versus 0.9 M. For all experiments using the above two types of datasets, we will plot their 5th and 95th percentile error bars across the 8 sub-datasets.

(3) *Synthetic datasets* We generated 11 datasets following the Zipf [27] distribution with various skewness (from 0.0 to 3.0 with a step of 0.3). Each dataset has 32M items and different numbers of distinct items depending on the skewness. The length of each item in each dataset is 4 bytes. The generator's code comes from an open source performance testing tool named Web Polygraph [69].

6.1.2 Implementation

We have implemented CM (Count-Min for short), CM–CU, min-heap, SS (Space-Saving), FR (FlowRadar), ASketch (Augmented sketch), PIE and our CF (including two speed optimizations) in C++. The hash functions are implemented from the 32-bit Bob Hash (obtained from the open source website [70]) with different initial seeds.

6.1.3 Computation platform

We conducted all the experiments on a machine with two 6-core processors (12 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62 GB DRAM memory. Each processor has three levels of cache: one 32 KB L1 data cache and one 32 KB L1 instruction cache for each core, one 256 KB L2 cache for each core, and one 15 MB L3 cache shared by all cores. Although we have 12 cores, the experiments in this section are done in only one core. More cores are used in the next section.

6.1.4 Queries

We define the query set for each task in our experiments, which is used in the metrics. In frequency estimation, the query set is the distinct items in the dataset. In top-k, heavy changes, and persistent items, the query set is the set of items reported by the specific algorithm.

6.1.5 Metrics

(1) *Average absolute error (AAE) in frequency estimation and top-k* $\frac{1}{|\Psi|}\sum_{e_i \in \Psi} |f_i - \widehat{f_i}|$, where f_i is the real frequency of item e_i, $\widehat{f_i}$ is its estimated frequency, and Ψ is the query set. Here, we query the whole dataset by querying each distinct item once.

(2) *Average absolute error (AAE) in persistent items* $\frac{1}{|\Psi|}\sum_{e_i \in \Psi} |f_i - \widehat{f_i}|$, where f_i is the real occurrence of item e_i which is reported as persistent when using PIE or PIE+CF, $\widehat{f_i}$ is its estimated occurrence, and Ψ is the set of items that are reported when using PIE or PIE + CF.

(3) *Average relative error (ARE) in persistent items* $\frac{1}{|\Psi|}\sum_{e_i \in \Psi} \frac{|f_i - \widehat{f_i}|}{f_i}$, where f_i is the actual occurrence of item e_i reported as persistent when using PIE or PIE + CF, $\widehat{f_i}$ is its estimated occurrence, and Ψ is the set of items that are reported when using PIE or PIE + CF.

(4) *Precision rate (PR) in top-k and heavy changes* ratio of the number of correctly reported instances to the number of reported instances.

Table 4 Parameter setting for CF

	$\mathcal{T}_2$	d_b	w_c	$M_1 : M_2$
CM/CM–CU [9,29]	241	1000	16	13:7
Space-Saving [10]	–	96	16	7:13
FlowRadar [32]	241	200	16	13:7

(5) *Recall rate (CR) in heavy changes and persistent items* ratio of the number of correctly reported instances to the number of correct instances.

(6) *Memory threshold* ($\mathcal{T}_m$) *in heavy changes* the least total memory usage of FlowRadar (with CF) when the F_1 score [71] (= $\frac{2\times PR\times CR}{PR+CR}$) reaches 99%.

(7) *Speed* million operations (insertions or queries) per second (Mops). All the experiments about speed are repeated 100 times to ensure statistical significance.

(8) *Decoding time in persistent items* time to decode all the persistent items.

6.2 Evaluation of frequency-based tasks

6.2.1 Parameter setting

Let M_{cf} be the memory of CF, M_{ar} the memory of the aggregate-and-report component, and M_1 (resp. M_2) the memory of layer L_1 (resp. L_2) in CF. We have $M_{cf} = M_{ar} + M_1 + M_2$. For each task, we set $d_1 = d_2 = 3$, $\delta_1 = 4$, $\delta_2 = 16$, $\mathcal{T}_1 = 15$ in CF. Table 4 lists the other parameter settings for each task. The remaining setting is as follows:

(1) *Estimating item frequency* We compare four approaches: CM, CM–CU, CM–CU with ASketch, and CM–CU with CF. For each of the four CM/CM–CU sketches, we allocate 2 MB of memory (M_t), use 3 hash functions,[6] and set the counter size to 32 bits. For CM–CU with ASketch, we set its filter size to 32 items as the original paper [4] recommends.

(2) *Finding top-k hot items* We compare four approaches: CM with heap, CM–CU with heap, SS, and SS with CF. We do not compare the above approaches with ASketch, since it can only capture a small number of the hottest items (around 32 items) while we will vary k between 32 and 1024. Recall that H denotes the number of item-counter pairs in SS. For SS with CF, we set $H = 2.5\ k$, $M_{cf} = 200$ KB. We use the actual frequency of the kth hottest item from one extra dataset (e.g., the 9th IP trace or web page dataset) as the prediction for the experimental datasets. Let $\mathcal{F}$ be the predicted frequency. We set $\mathcal{T} = \mathcal{F} \times 0.90$. SS uses a linked list and a hash table to implement the Stream-Summary data structure, and achieves $O(1)$ time complexity on average. Each item-counter pair needs around 100 bytes memory on average (including pointers and unoccupied space for handling hash collisions quickly). Therefore, for SS, we set $H = 2.5\ k + M_{cf}/(100 \text{ bytes}) = 2.5\ k + 2048$ for a fair comparison. CM/CM–CU with heap maintains k item-counter pairs in its heap and uses a hash table for item lookup. Each item-counter pair needs around 100 bytes of memory on average. Therefore, we allocate $M_{cf} + 2.5\ k * 100$ bytes $- k * 100$ bytes $= M_{cf} + 150\ k$ bytes memory to CM/CM–CU.

(3) *Detecting heavy changes* We compare two approaches: FR, and FR with CF. For both, we set the numbers of hash functions in the Bloom filter and IBLT to 3 (recommended by [33,73]). We set $M_{cf} = 200$ KB. Let M_{bf} and M_{ib} be the memory of the Bloom filter and IBLF, respectively. We set M_{bf}:M_{ib} = 1:9, as FR achieves the best performance according to our tests in such setting.

6.2.2 Evaluation on three key tasks

(1) Estimating item frequency

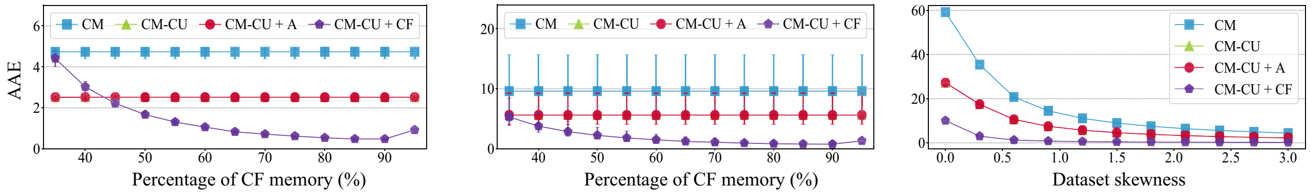
Accuracy (*Fig. 8a, b*) Our results show that the AAE of CM–CU with CF is about 9.8, 5.2 and 5.2 times, and 12.5, 7.3 and 7.3 times lower than CM, CM–CU and CM–CU with ASketch when the percentage of CF memory, M_{cf}/M_t, is set to 90% on two real-world datasets, respectively. ASketch improves the accuracy of CM–CU a little on both datasets. We further study how the skewness of synthetic dataset affects the accuracy, see Fig. 8c. Here, M_{cf}/M_t is fixed to 90%. We find that CM–CU with CF achieves higher accuracy than the other three approaches, irrespective of the skewness.

In Fig. 8a, all methods obtain an error of at most 5, which is not large and could be acceptable in many circumstances. However, in some circumstances, cold items are important. For a cold item with frequency of only 1 or 2, AAE of 6 could not be acceptable. Therefore, AAE should be as small as possible. And most items are cold; therefore, reducing the AAE will affect many cold items, and thus AAE should be as small as possible.

Accuracy (10× *long stream*) (*Fig. 9*) We use the same parameters setting as above, but load 10× packets, namely 320 M packets. In practical, We clear CF every 32 M packets in our experiment. Our results show that the AAE of CM–CU with CF is about 14.0, 7.9 and 7.9 times lower than CM, CM–CU and CM–CU with ASketch when the percentage of CF memory, M_{cf}/M_t, is set to 90%.

Insertion speed (*Fig. 10a, b*) Our results show that the insertion speed of CM–CU with CF is about 2.5, 2.9 and 3.4 times, and 1.6, 1.7 and 3.4 times faster than CM, CM–CU

[6] The literature [57,72] recommends using a small number of hash functions.



(a) Varying percentage of CF memory (IP trace).

(b) Varying percentage of CF memory (web page).

(c) Varying dataset skewness.

Fig. 8 AAE versus percentage of CF memory on two real-world datasets, and versus skewness of synthetic dataset

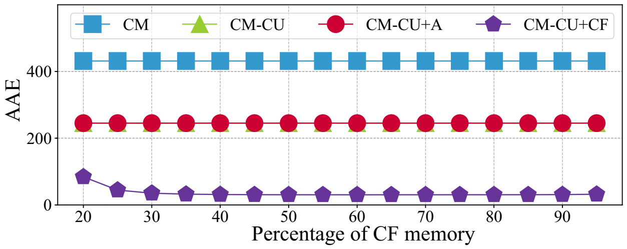


Fig. 9 Accuracy experiment on 10× long streams

and CM–CU with ASketch when M_{cf}/M_t is set to 90% on two real-world datasets, respectively. ASketch lowers the insertion speed of CM–CU, due to dynamically capturing hot items in its filter on both datasets. We further study how the skewness of synthetic dataset affects the insertion speed, see Fig. 10c. Here, again, M_{cf}/M_t is fixed to 90%. When $skewness > 1.35$,[7] CM–CU with CF achieves a higher insertion speed than the other three approaches. The reason why CM–CU with CF on synthetic datasets cannot achieve such high speedup as on the IP trace datasets is that the appearance order of items in synthetic datasets is fully randomized (while stream burst often happens in real data streams, see Sect. 3.3), which largely weakens the aggregating performance of the aggregate-and-report component and degrades the speed. Our CF with Agg can perform better than ASketch in terms of insertion speed mainly because of the following 4 reasons. (1) Each cell in ASketch has three fields: item ID, new_count and old_count, but our CF has only two fields: item ID and frequency. Thus, ASketch needs more time for memory access for one cell. (2) For each incoming item, ASketch needs to travel the whole filter (containing 32 items). But our Cold Filter only needs to travel one bucket (containing 16 items). (3) When the incoming item e isn't inserted into the filter of ASketch, ASketch needs to travel the filter again to track the smallest value in the filter. Different from ASketch, our Agg technique does not need to track the smallest value; thus, we will always only need to travel the bucket once. (4) Furthermore, when an item in the filter of ASketch needs to move into the underlying sketches, the data are two-way exchanged. However, our Cold Filter relies on one-direction communication, which helps to guarantee a high processing speed. In Fig. 10c, CM–CU+A has lower throughput than other method, which is different from the ASketch paper [4]. The authors of the ASketch paper use 8 hash functions in the original paper. Differently, we use 3 hash functions for all sketches according to the recommendation of the literature [57,72] in our experiments. The experiment results of 8 hash functions are shown in Figs. 11 and 12.

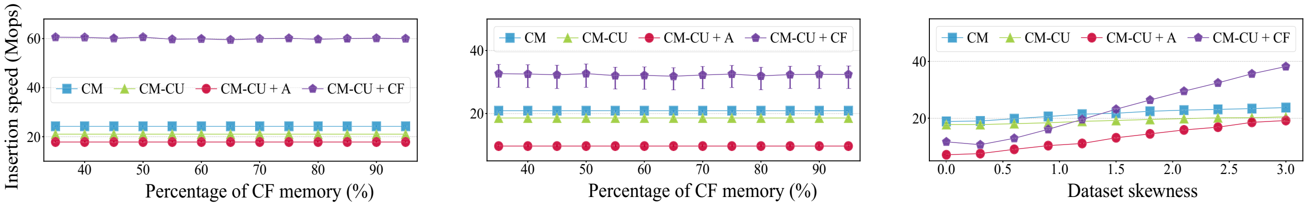
In Fig. 11, the insertion speed of ASketch with 8 hash functions can be faster than CM (also with 8 hash functions) when the dataset skewness is high, which is consistent with the ASketch paper. However, our experiment results (see Figs. 8c, 10c, 11 and 12) show that using 3 hash functions improves not only insertion speed, but also accuracy. Thus we think using 3 hash functions is more reasonable.

Besides, we find that after changing the number of hash functions to 8, the insertion speed of ASketch implemented by us is still much slower than their result when skewness is high (e.g., 3 versus 100 Mops when skewness = 3). To find out reasons of the gap, we have tested the insertion speed of the filter itself (i.e., if an item passes the filter, we will do nothing instead of inserting it into a CM), and we find that the insertion speed of the filter itself (consists of 32 items) barely achieves 20 Mops, which is far slower than 100 Mops, the experimental result of the complete ASketch when the skewness is 3 in their paper. Therefore, we believe that the insertion speed of ASketch cannot exceed 20 Mops in our computation platform.

Finally, we make our implementation of ASketch open source [38]. We also try to ask the authors of ASketch paper for their codes and implementation details, but we have not got a reply from them.

Query speed (*Fig.* 13*a, b*) Our results show that the query speed of CM–CU with CF is about 1.1, 1.1 and 1.3 times, and 1.3, 1.3 and 1.5 times faster than CM, CM–CU and CM–CU with ASketch when M_{cf}/M_t is set to 90% on two real-world datasets, respectively. On both datasets, the query speed of CM–CU with CF is higher than the other three approaches. The reason is that the one-memory-access strategy signif-

[7] The literature [2] reported skewness > 1.4 in real data streams.



(a) Varying percentage of CF memory (IP trace).

(b) Varying percentage of CF memory (web page).

(c) Varying dataset skewness.

Fig. 10 Insertion speed versus percentage of CF memory on two real-world datasets, and versus skewness of synthetic dataset

icantly speeds up the query process for the large number of cold items, improving the overall query speed. We further study how the skewness of synthetic dataset affects the query speed, see Fig. 13c. Here, M_{cf}/M_t is fixed to 90%. When skewness > 0.45, CM–CU with CF achieves a higher query speed than the other three approaches.

(2) Finding top-k hot items

The aforementioned four approaches have the same query process, and thus we skip query speed below.

Accuracy (*Fig.* 14*a, b*) Our results show that SS with CF achieves precision rate above 99.8% on two real-world datasets. SS with CF achieves higher and more stable accuracy than the other three approaches on both datasets. We further study how the skewness of synthetic dataset affects the precision rate, see Fig. 14c. Here, k is set to 256. When skewness is 0, all approaches have precision rates of 0. The reason is that on uniform datasets (skewness = 0), the frequencies of top-k hot items are very close to those of other items, leading to difficulties of differentiating them from others. When

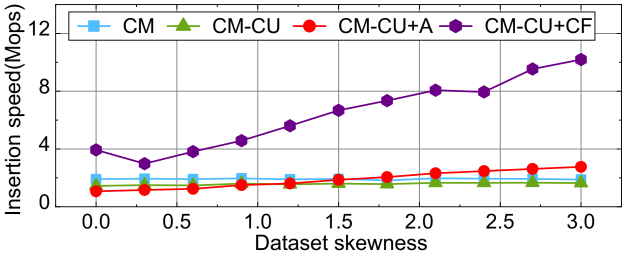


Fig. 11 Insertion speed versus skewness (8 hash functions)

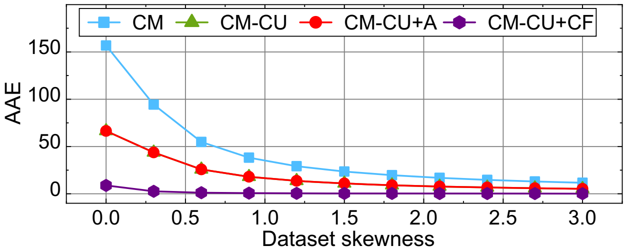
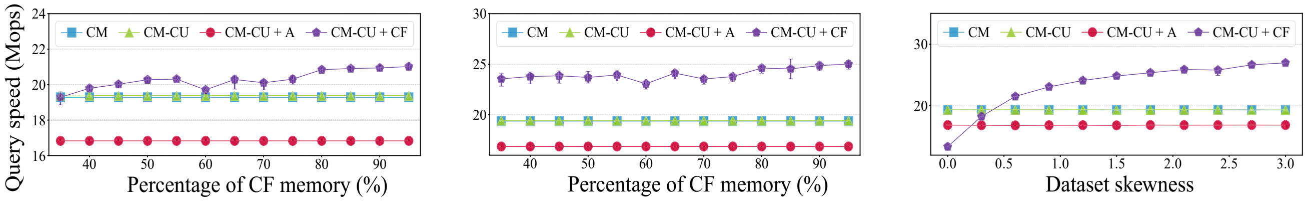


Fig. 12 Accuracy versus skewness (8 hash functions)

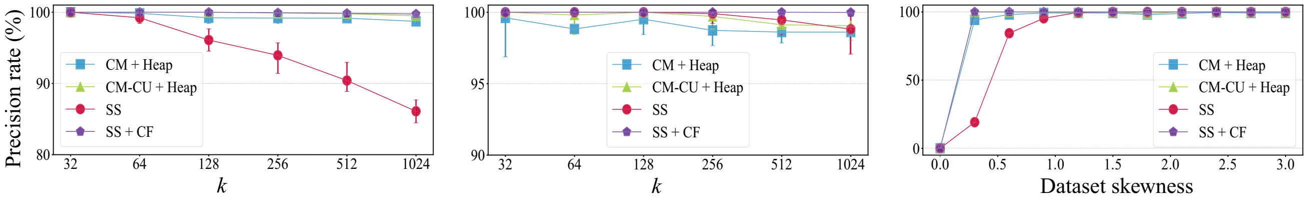


(a) Varying percentage of CF memory (IP trace).

(b) Varying percentage of CF memory (web page).

(c) Varying dataset skewness.

Fig. 13 Query speed versus percentage of CF memory on two real-world datasets, and versus skewness of synthetic dataset

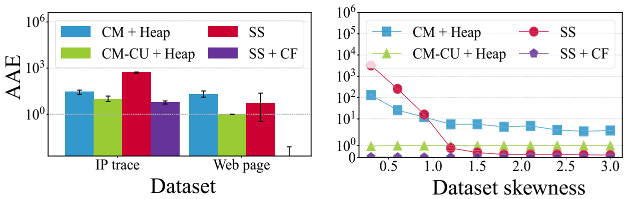


(a) Varying k (IP trace).

(b) Varying k (web page).

(c) Varying dataset skewness.

Fig. 14 Precision rate versus k on two real-world datasets, and versus skewness of synthetic dataset



(a) On real-world datasets. **(b)** On synthetic datasets.

Fig. 15 AAE on real-world and synthetic datasets

skewness ⩾ 0.3, SS with CF achieves precision rates above 99.9%. We finally test the AAE for frequencies of the correctly reported items on different datasets, see Fig. 15a, b. On all datasets, SS with CF achieves a much lower AAE than the other three approaches.

Insertion speed (*Fig.* 16*a, b*) Our results show that the insertion speed of SS with CF is about 3.7, 3.9 and 1.9 times, and 1.6, 1.7 and 1.2 times faster than CM with heap, CM–CU with heap and SS when k is set to 256 on two real-world datasets, respectively. We then study how the skewness of synthetic dataset affects the insertion speed of SS, see Fig. 16c. Here, k is set to 256. When skewness ⩾ 0.6, SS with CF achieves a higher insertion speed than the other three approaches. There are some differences in insertion speed between different input types. The reason for the differences is that the distribution of different datasets may vary. Different practical frequency distribution leads to different processing speed.

(3) Detecting Heavy Changes

We detect heavy changes using threshold ϕ of 0.04% between the first 16 M and the second 16 M items of the considered datasets. Since the value of ϕ does not affect the performance much in our experiments, we omit the figures when varying ϕ.

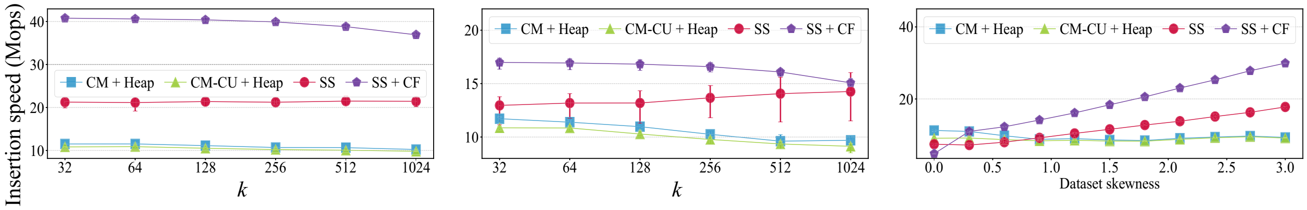
Memory consumption (*Fig.* 17*a, b*) Our results show that the memory threshold ($\mathcal{T}_m$) of FR with CF is about 12.6 times and 22.4 times lower than FR on two real-world datasets, respectively. The major reason for such memory reduction is that one hot item consumes the same memory in FR as one cold item; CF makes only hot items (which are much fewer than cold ones) and a small portion of cold items fed to FR (Sect. 4.1.3), and thus the memory usage for FR to record items is largely reduced. We can get the same accuracy, since CF accurately records the frequencies of cold items.

Insertion speed (*Fig.* 17*c, d*) Our results show that the insertion speed of FR with CF is about 4.7 times and 1.8 times faster than FR on two real-world datasets, respectively.

Query speed Our results show that the query speed of FR with CF is about 18.3 times and 39.2 times faster than FR on two real-world datasets, respectively. The average query time on the IP trace datasets for FR and FR with CF is 547 ms and 30 ms, respectively. On Web page datasets, the average query time is 1066 ms and 27 ms, respectively. Due to space limitations, we do not plot them.

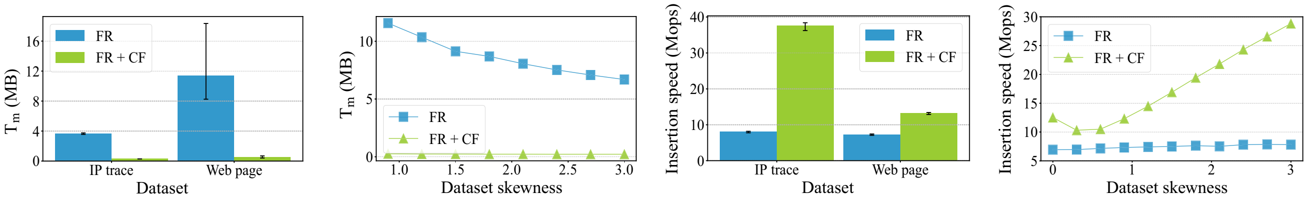
6.2.3 Sensitivity analysis

We use two metrics, namely *accuracy improvement* and *speedup* (for insertion), to uniformly depict the performance of all considered stream processing algorithms. For ease of presentation, we define them specially to make a larger value always mean higher performance. For CM and CM–CU, their accuracy improvements are both defined



(a) Varying k (IP trace). **(b)** Varying k (web page). **(c)** Varying dataset skewness.

Fig. 16 Insertion speed versus k on two real-world datasets, and versus skewness of synthetic dataset



(a) On real-world datasets. **(b)** On synthetic datasets. **(c)** On real-world datasets. **(d)** On synthetic datasets.

Fig. 17 T_{m} and insertion speed on real-world and synthetic datasets to be $\text{AAE}_{\text{pure}}/\text{AAE}_{\text{with CF}}$. For SS, its accuracy improvement is defined to be $\text{PR}_{\text{with CF}}/\text{PR}_{\text{pure}}$. For FR, its accuracy improvement is defined to be $\mathcal{T}_{m_{\text{pure}}}/\mathcal{T}_{m_{\text{with CF}}}$. For all the four algorithms, their speedups are defined to be $\text{speed}_{\text{with CF}}/\text{speed}_{\text{pure}}$.

(1) Impact of different optimizations

In this subsection, we focus on the impact of different optimizations on P_{mr}, and accuracy improvements and speedups of CM, CM–CU, SS, and FR. Recall that the misreported rate P_{mr} is defined as Eq. 1. We also evaluate how Agg (aggregate-and-report for short) solely influences the performance of these four algorithms. We set $M_1 + M_2 = 1\text{MB}$, $d_b = 1000$ for each CF, and the rest of parameters are the same as in Sect. 6.1. In this subsection, CF refers to the pure Cold Filter without any optimization.

Impact on P_{mr} (*Fig.* 4) Here, we set $\mathcal{T} = 256$. Recall that $\mathcal{T}$ is a threshold that we need to estimate whether an item's frequency exceeds this threshold. We observe that method combining Agg and Oma (one-memory-access) will elevate the P_{mr}, but Oma degrades the P_{mr}. The reason is that in Agg, the appearance order of items witnessed by CF is changed, which could influence the P_{mr}; in Oma, word constraint degrades the P_{mr}.

Impact on accuracy improvement (*Fig.* 5*a, b*) In average, on both datasets, for CM and CM–CU, the percentages of accuracy improvement contributed by CF, Agg, and Oma are around 140%, 0%, and −40%, respectively; for SS and FR, they are around 100%, 0%, and 0%. In other words, CF helps each algorithm achieve the maximum accuracy improvement, while Agg does not improve the accuracy of each algorithm; Oma degrades the accuracy of CM and CM–CU, and makes little impact on the accuracy of SS and FR.

Impact on speedup (*Fig.* 5*c, d*) In average, on both datasets, for CM, the percentages of speedup contributed by CF, Agg, and Oma are around −73%, 150%, and 23%, respectively; for CM–CU, they are around −64%, 142%, and 22%; for SS, they are around 11%, 68%, and 21%; for FR, they are around 11%, 72%, and 17%. In other words, CF degrades the speed of CM and CM–CU, while it improves the speed of SS and FR; Agg improves the speed largely; Oma improves the speed.

Impact on speedup (*synthetic dataset*) (*Fig.* 18) Our result shows that Agg plays the most significant role in speedup, and all the three optimizations containing Agg can achieve higher speedup with the increasing of skewness. As to Oma, it can help improve insertion speed, because when comparing two optimizations that only have difference on Oma, the one with Oma will always achieve a higher insertion speed than the other. Finally, CF has negative effect on insertion speed, because the overhead of CF is huge, comparing to the throughput of CM Sketch.

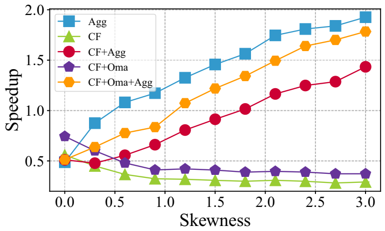


Fig. 18 Impact of different optimizations on the speed of Count-Min

In most cases, adding CF degrades the processing speed, such as adding CF to CM/CM–CU, CM/CM–CU + Agg, SS, or FR, because the overhead of processing every item in CF is larger than the benefit of only processing *hot* items in the existing algorithms. However, when adding CF to SS + Agg or FR+Agg, the processing speed increases. The reason behind is as follows. Whether CF+Agg is faster than Agg depends on whether Agg and CF cooperate well, while the latter depends on the two factors of existing algorithm: (1) the value of $\mathcal{T}$ required by it ($\mathcal{T}$ is the frequency threshold for items to pass CF), and (2) the processing speed of it. Specifically, SS has a high $\mathcal{T}$ (see Sect. 4.1.2). This makes CF cooperate well with Agg: CF filters out many more cold items while Agg works better for hot items. FR is much slower than CM/CM–CU (see Figs. 10a and 17c). This makes the cooperation of CF and Agg gain larger benefits: existing algorithms only process hot items with aggregated frequencies.

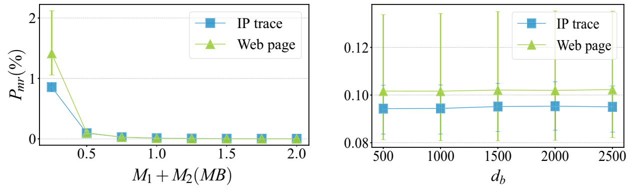
Summary (1) Pure CF plays the main role in improving the accuracy, while Agg is the primary factor in improving the speed; (2) CF+Agg+Oma achieves both high accuracy and high speed; (3) For the stream processing algorithms that require high $\mathcal{T}$ or are relatively slow, adding CF to Agg improves the speed.

(2) Impact of Memory Budget of CF

Next, we focus on the impact of memory budget of CF (i.e., $M_1 + M_2$ and d_b) on P_{mr}, and accuracy improvements and speedups of CM–CU, SS, and FR. We set $M_1 + M_2 = 1\text{MB}$ and $d_b = 1000$ by default, and the rest of parameters are the same as in Sect. 6.1.

Impact on P_{mr} (Fig. 19*a, b*) Here, we set $\mathcal{T} = 256$. When $M_1 + M_2 \geqslant 0.5\text{MB}$, P_{mr} decreases to around 0.1% on both datasets. Besides, d_b makes little impact on the P_{mr} on both datasets.

Impact on accuracy improvement (*Fig.* 20*a–d*) For CM–CU, on both datasets, larger $M_1 + M_2$ leads to higher accuracy, while the opposite is the case for FR. The reason for such opposite case is the following: larger $M_1 + M_2$ helps reduce more memory in FR (due to lower P_{mr}), but leads to the



(a) Varying $M_1 + M_2$. **(b)** Varying d_b.

Fig. 19 Impact of memory budget of CF on P_{mr}

increasing of T_{m} (recall that T_{m} contains M_1+M_2), while the latter is much larger than the former. The accuracy improvement of SS remains unchanged on both datasets, since it has reached 100%. Actually, larger $M_1 + M_2$ does influence SS by lowering its AAE (not covered in figures due to space limitation). Besides, d_{b} makes little impact on the accuracy improvement of each algorithm on both datasets.

Impact on speedup (*Fig.* 21*a–d*) M_1+M_2 makes little impact on the speedup of each algorithm on both datasets. Larger d_{b} leads to a higher speedup of each algorithm on both datasets, except that on web page datasets, the speedup begins to decrease a little when $d_{\mathrm{b}} > 2000$. The reason for such decreasing is twofold: (1) the cache performance declines as the memory of Agg increases; (2) the aggregating performance of Agg nearly reaches the maximum.

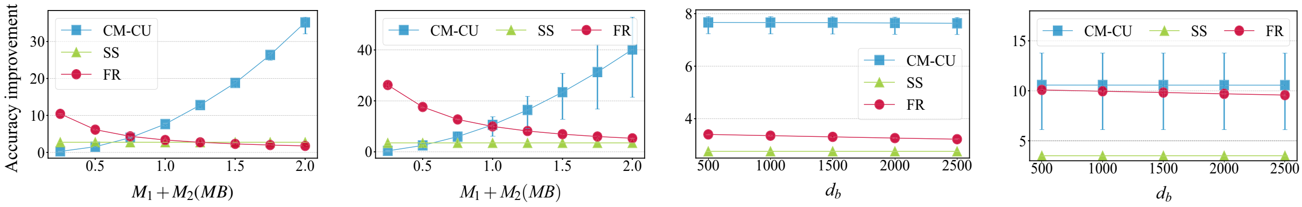
Summary (1) M_1+M_2 mainly influences accuracy, while d_{b} mainly influences speed; (2) For CM–CU, larger $M_1 + M_2$ leads to higher accuracy; for SS, M_1+M_2 makes little impact on its precision rate; for FR, relatively small M_1+M_2 brings lower T_{m}.

(3) Impact of parameter setting in CF

Below, we focus on the impact of parameter setting (including layer number, $M_1/(M_1 + M_2)$, δ_1:δ_2, hash number, and $\mathcal{T}$) in CF on P_{mr}, and accuracy improvements and speedups of CM–CU, SS, and FR. Since the accuracy improvement and speedup behave similarly on both datasets, we only show their figures on the IP trace datasets. We set $M_1 + M_2 = 1$ MB and $d_{\mathrm{b}} = 1000$, and the rest of parameters are the same as in Sect. 6.1 by default, unless otherwise specified. During varying layer number, we equally divide the memory across different layers; all the layers except for the highest one have 4-bit counters with Oma; all the layers have the same number of hash functions. During varying layer number and $\delta_1 : \delta_2$, the counter size in the highest layer is set according to the value of $\mathcal{T}$ required by all the three algorithms.

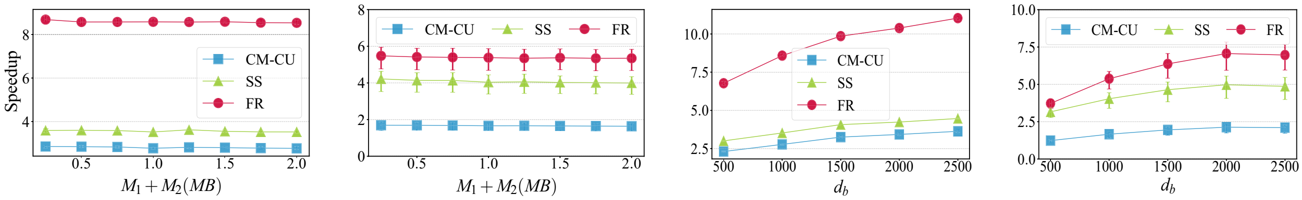
Impact on P_{mr} (*Fig.* 22*a–e*) Two layers and $\delta_1 : \delta_2 = 4 : 16$ lead to the minimum P_{mr}; Larger $M_1/M_1 + M_2$ leads to larger P_{mr}; Larger number of hash functions and larger $\mathcal{T}$ lead to smaller P_{mr}.

Impact on accuracy improvement (*Fig.* 23*a–e*) For CM–CU, two layers and $\delta_1 : \delta_2 =$ 4:16 lead to the maximum accuracy improvement; when $M_1/(M_1 + M_2)$ is around 55%, the accuracy improvement achieves the maximum (the corresponding ratio on web page datasets is around 70%); larger number of hash functions leads to higher accuracy, but such impact is not remarkable when hash number is larger than 4; when $\mathcal{T} \geqslant 256$, its accuracy improvement remains almost unchanged. For SS and FR, layer number, $M_1/(M_1 + M_2)$,



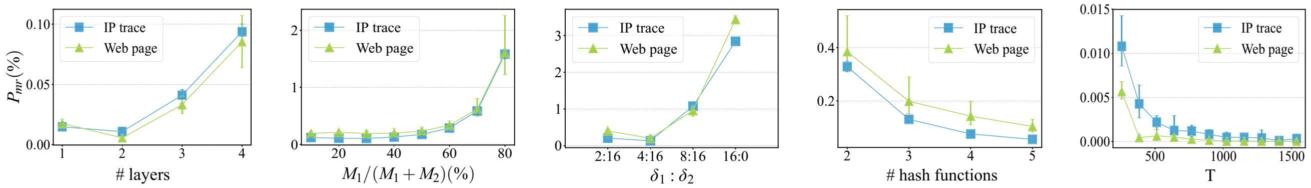
(a) Varying $M_1 + M_2$ (IP trace). **(b)** Varying $M_1 + M_2$ (web page). **(c)** Varying d_b (IP trace). **(d)** Varying d_b (web page).

Fig. 20 Impact of memory budget of CF on the accuracy of CM–CU, Space-Saving and FlowRadar



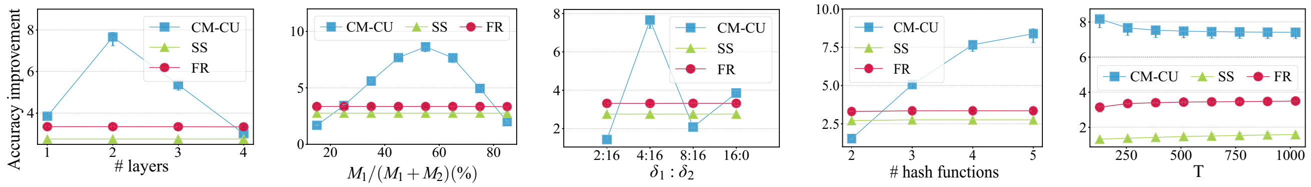
(a) Varying $M_1 + M_2$ (IP trace). **(b)** Varying $M_1 + M_2$ (web page). **(c)** Varying d_b (IP trace). **(d)** Varying d_b (web page).

Fig. 21 Impact of memory budget of CF on the speed of CM–CU, Space-Saving and FlowRadar



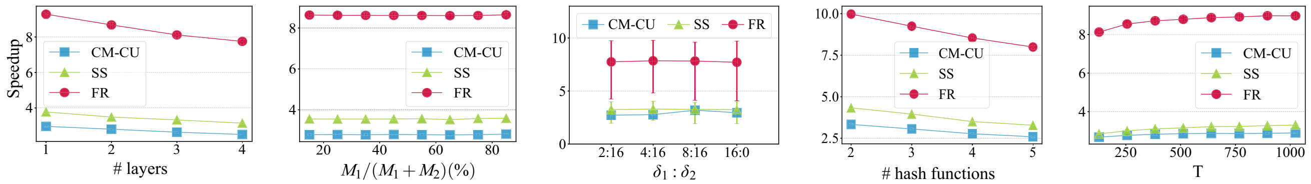
(a) Varying # layers. **(b)** Varying $M_1/(M_1 + M_2)$. **(c)** Varying $\delta_1 : \delta_2$. **(d)** Varying # hash functions. **(e)** Varying $\mathcal{T}$.

Fig. 22 Impact of parameter setting in CF on P_{mr}



(a) Varying # layers (IP trace). **(b)** Varying $M_1/(M_1 + M_2)$ (IP trace). **(c)** Varying $\delta_1 : \delta_2$ (IP trace). **(d)** Varying # hash functions (IP trace). **(e)** Varying $\mathcal{T}$ (IP trace).

Fig. 23 Impact of parameter setting in CF on the accuracy of CM–CU, Space-Saving and FlowRadar



(a) Varying # layers (IP trace). **(b)** Varying $M_1/(M_1 + M_2)$ (IP trace). **(c)** Varying $\delta_1 : \delta_2$ (IP trace). **(d)** Varying # hash functions (IP trace). **(e)** Varying $\mathcal{T}$ (IP trace).

Fig. 24 Impact of parameter setting in CF on the speed of CM–CU, Space-Saving and FlowRadar

δ_1:δ_2, and hash number make little impact on their accuracy (actually, when $M_1/(M_1 + M_2)$ is around 35%, the AAE of SS achieves the minimum); the accuracy improvement of SS increases gradually as $\mathcal{T}$ increases, and when $\mathcal{T}$ reaches near the frequency of the kth hottest item, the accuracy improvement achieves the maximum (not covered in figures); when $\mathcal{T} \geqslant 256$, the accuracy improvement of FR remains almost unchanged.

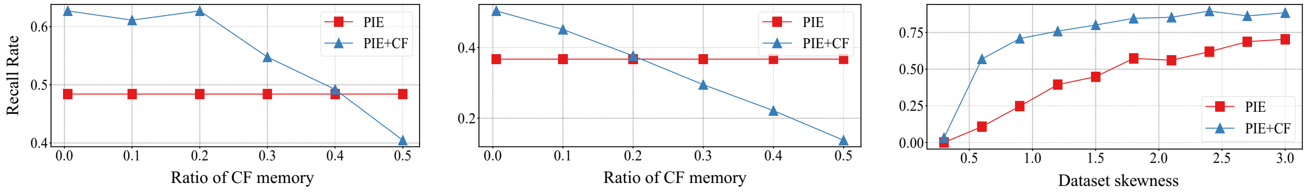
Impact on speedup (*Fig.* 24*a–e*) For each of three algorithms, larger numbers of layers and hash functions lead to lower speed; $M_1/(M_1 + M_2)$ and δ_1:δ_2 make little impact on the speed; larger $\mathcal{T}$ leads to a higher speed, but such impact is not remarkable.

Summary (1) Two layers, δ_1:δ_2 = 4:16 and 3 or 4 hash functions are recommended to achieve both high accuracy and high speed; (2) $M_1/(M_1 + M_2)$ for CM–CU should be in the range of $55\% - 70\%$; $M_1/(M_1 + M_2)$ for SS should be around 35%; $M_1/(M_1 + M_2)$ makes little impact on the performance of FR; (3) $\mathcal{T}$ makes little impact on the performance of CM–CU and FR; $\mathcal{T}$ for SS should be set according to the predicted frequency of the kth hottest item.

6.3 Evaluation of persistency-based tasks

In this section, we show the experimental results on finding persistent items. We compare PIE with CF and the origin PIE algorithm, in terms of accuracy, insertion speed, and decoding speed. We also show how parameters affect the performance in Sect. 6.3.2.

Parameter setting Let M_{cf} be the memory of CF, and M_{pie} be the memory of PIE, and $\mathcal{T}_1$ be the threshold of CF. M_{cf}, M_{pie}, and $\mathcal{T}_1$ can be changed or fixed in the experiment. We set $M_{\text{cf}} + M_{\text{pie}} = 4.8$ MB, $\mathcal{T}_1 = 3$ in our evaluation (Sect. 6.3.1). When varying dataset skewness, as shown in Figs. 25c, 26c and 27c, CF memory is set to 0.05 MB, $\mathcal{T}_1 = 3$. For each dataset, we read 600,000 items and separate them into 300 windows, so we get 2000 items in each window. The length for each item is 4 bytes. The threshold of persistent item $\mathcal{T}$

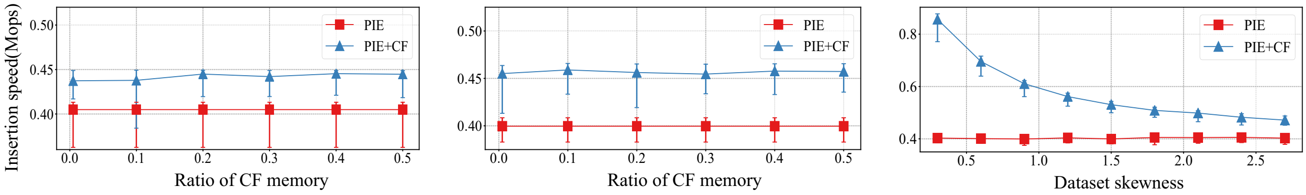


(a) Varying ratio of CF memory (IP trace).

(b) Varying ratio of CF memory (web page).

(c) Varying dataset skewness.

Fig. 25 Recall rate versus ratio of CF memory on two real-world datasets, and versus skewness of synthetic dataset

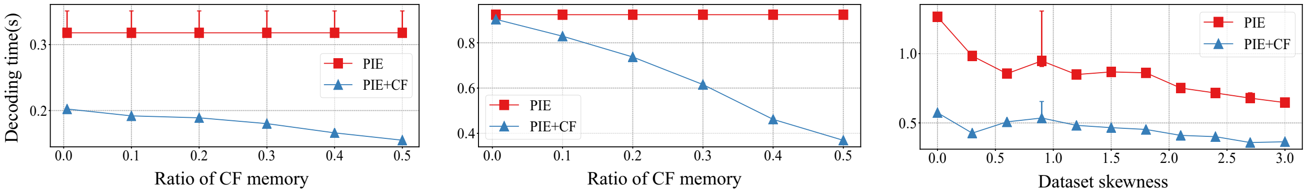


(a) Varying ratio of CF memory (IP trace).

(b) Varying ratio of CF memory (web page).

(c) Varying dataset skewness.

Fig. 26 Insertion speed versus ratio of CF memory on two real-world datasets, and versus skewness of synthetic dataset



(a) Varying ratio of CF memory (IP trace).

(b) Varying ratio of CF memory (web page).

(c) Varying dataset skewness.

Fig. 27 Decoding time versus ratio of CF memory on two real-world datasets, and versus skewness of synthetic dataset

is set to 110. In other words, if an item appears in more than 110 windows, the item is a persistent item.

6.3.1 Evaluation of accuracy and time

Recall rate (*Fig.* 25*a–c*) Our results show that the recall rate PIE with CF is 1.295 and 1.371 times better than PIE when CF memory ratio is set to 0.5% on two real-world datasets, respectively. We further study how the skewness of synthetic dataset affects the recall rate, see Fig. 25c. We find that PIE with CF achieves better accuracy than PIE, irrespective of the skewness. When skewness is 0, there is not any persistent items in the dataset, so the recall rate is 0. The number of different items in the dataset decreases when we increasing the skewness, and the decrease of the number of different items makes collision in PIE less likely to happen. When there is less collision in PIE, the recall rate will become higher. So the recall rate increases when increasing dataset skewness. In Fig. 28, we load 24,000,000 packets, and separate them into 300 windows. Let M_{cf} be the memory of CF, and M_{pie} be the memory of PIE, and $\mathcal{T}_1$ be the threshold of CF. We set $M_{\text{cf}} + M_{\text{pie}} = 36$ MB, $\mathcal{T}_1 = 15$. The result is similar with the experiment on the smaller datasets.

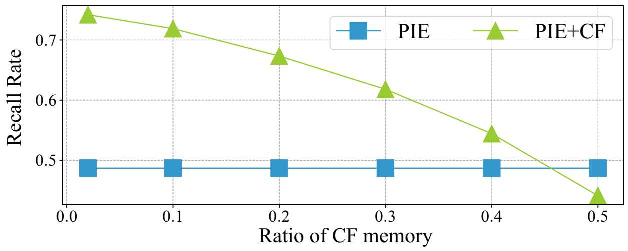
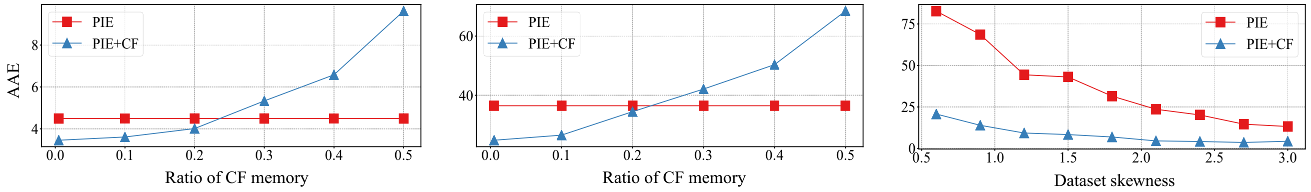


Fig. 28 Recall rate versus ratio of CF memory on larger dataset

AAE (*Fig.* 29*a–c*) Our results show that the AAE of PIE with CF is 1.300 and 1.471 times lower than PIE when CF memory ratio is set to 0.5% on two real-world datasets, respectively. The AAE increases when CF memory becomes larger, because the memory of PIE becomes less and it makes collision more likely to happen. When varying dataset skewness, see Fig. 29c, the improvement of accuracy is significant at any skewness.

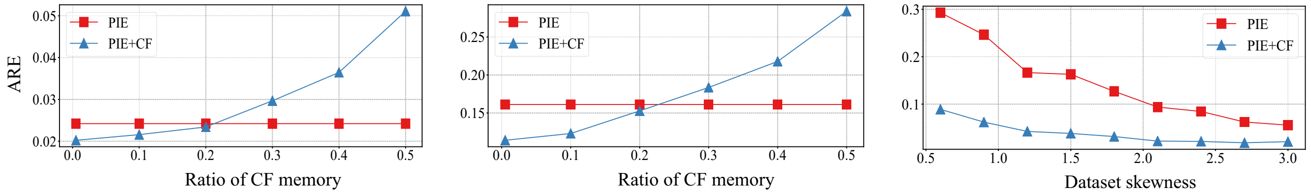


(a) Varying ratio of CF memory (IP trace).

(b) Varying ratio of CF memory (web page).

(c) Varying dataset skewness.

Fig. 29 AAE versus ratio of CF memory on two real-world datasets, and versus skewness of synthetic dataset



(a) Varying ratio of CF memory (IP trace).

(b) Varying ratio of CF memory (web page).

(c) Varying dataset skewness.

Fig. 30 ARE versus ratio of CF memory on two real-world datasets, and versus skewness of synthetic dataset

ARE (*Fig.* 30*a–c*) Our results show that the ARE of PIE with CF is 1.197 and 1.413 times lower than PIE when CF memory ratio is set to 0.5% on two real-world datasets, respectively. The ARE increases when CF memory becomes larger, because the memory of PIE becomes less and it makes collision more likely to happen. When varying dataset skewness, see Fig. 30c, the improvement of accuracy is significant at any skewness.

Insertion speed (*Fig.* 26*a–c*) Our results show that the insertion speed of PIE with CF is 1.080 and 1.139 times better than PIE when CF memory ratio is set to 0.5% on two real-world datasets, respectively. When varying dataset skewness, see Fig. 26c, the improvement of insertion speed decreases when increasing skewness. There are more items filtered by CF in the lower skewness dataset than in the higher skewness dataset, so there are more items needed to be inserted in PIE for PIE+CF algorithm when increasing skewness, which causes a lower insertion speed. Insertion speed in the PIE experiment is much slower than other experiments, because PIE needs to encode/decode, and the time complexity is $O(n)$, where n is the size of flow ID. However, the time complexity of others is $O(1)$.

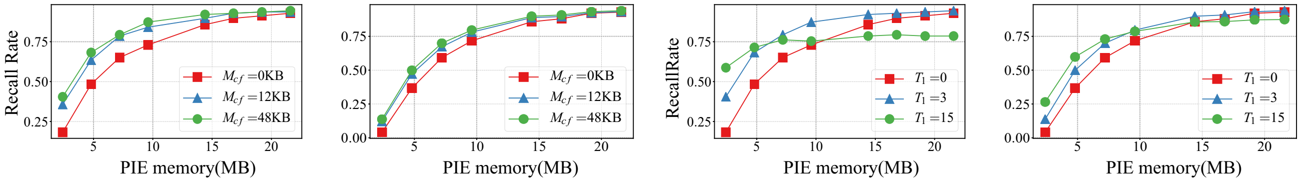
Decoding time (*Fig.* 27*a, b*) Our results show that the decoding speed of PIE with CF is 1.571 and 1.023 times better than PIE when CF memory ratio is set to 0.5% on two real-world datasets, respectively. The decoding time decreases when CF memory become larger, because the number of reported items becomes less. When varying dataset skewness, see Fig. 27c, the improvement of decoding speed is significant at any skewness.

6.3.2 Sensitivity analysis

In the experiment we observe the impact of PIE memory on recall rate, insertion speed, decoding time, AAE and ARE at different parameters' setting of CF. The parameters of CF include CF memory and threshold of CF. When CF memory M_{cf} is set to 0KB or the threshold of CF $\mathcal{T}_1$ is set to 0, we do not add CF to PIE.

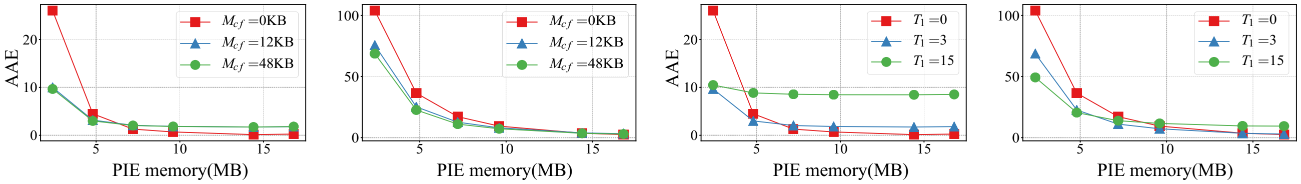
Impact of PIE memory and CF memory on recall rate (*Fig.* 31*a, b*) In this experiment we set CF threshold $\mathcal{T}_1$ to 3. We observe that the improvement of recall rate decreases when increasing PIE memory. And we observe that the improvement of recall rate increases when increasing CF memory, but such impact is not remarkable when CF memory is larger than 12 KB. The 12 KB CF can filter nearly all the cold items, which appears less than $\mathcal{T}_1 = 3$ times. The improvement of recall rate decreases when increasing PIE memory, because increasing PIE memory will decrease collision, and the effect of adding CF to alleviate collision will decrease.

Impact of PIE memory and CF threshold on recall rate (*Fig.* 31*c, d*) In this experiment we set CF memory M_{cf} to 96 KB. We observe that the improvement of recall rate decreases when increasing PIE memory. And we observe that the high CF threshold ($\mathcal{T}_1 = 15$) performs better than the low CF threshold ($\mathcal{T}_1 = 3$) when PIE memory is less than 5 MB. But when the PIE memory is more than 10 MB, the recall rate becomes lower for $\mathcal{T}_1 = 15$ than $\mathcal{T}_1 = 0$. This is because the high threshold CF filters too many items.



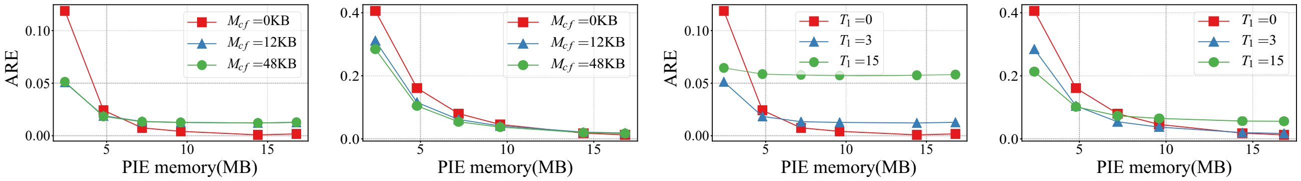
(a) Varying PIE memory and CF memory (IP trace). **(b)** Varying PIE memory and CF memory (web page). **(c)** Varying PIE memory and CF threshold (IP trace). **(d)** Varying PIE memory and CF threshold (web page).

Fig. 31 Impact of parameters on recall rate



(a) Varying PIE memory and CF memory (IP trace). **(b)** Varying PIE memory and CF memory (web page). **(c)** Varying PIE memory and CF threshold (IP trace). **(d)** Varying PIE memory and CF threshold (web page).

Fig. 32 Impact of parameters on AAE



(a) Varying PIE memory and CF memory (IP trace). **(b)** Varying PIE memory and CF memory (web page). **(c)** Varying PIE memory and CF threshold (IP trace). **(d)** Varying PIE memory and CF threshold (web page).

Fig. 33 Impact of parameters on ARE

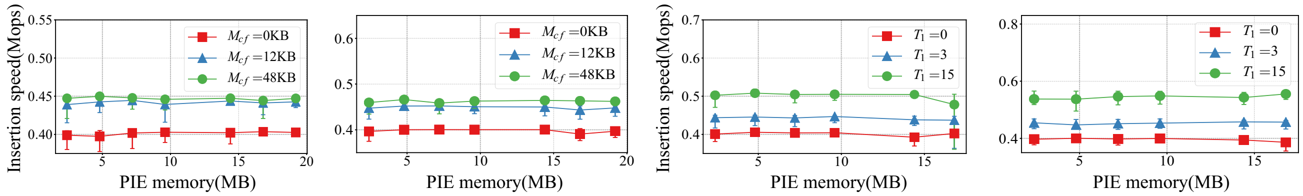
Impact of PIE memory and CF memory on AAE (*Fig.* 32*a, b*) In this experiment we set CF threshold $\mathcal{T}_1$ to 3. We observe that when the PIE memory is more than 7.2 MB, the AAE becomes higher for $M_{\text{cf}} = 12$ KB or $M_{\text{cf}} = 48$ KB than $M_{\text{cf}} = 0$ KB for IP trace dataset. And we observe that the impact of varying the CF memory is not remarkable when CF memory is larger than 12 KB. The 12 KB CF can filter nearly all the cold items, which appears less than $\mathcal{T}_1 = 3$ times.

Impact of PIE memory and CF threshold on AAE (*Fig.* 32*c, d*) In this experiment we set CF memory M_{cf} to 96 KB. We observe that the improvement of accuracy is better when $\mathcal{T}_1 = 3$ than when $\mathcal{T}_1 = 15$ for IP trace dataset. When PIE memory is more than 10 MB, the AAE is lower when $\mathcal{T}_1 = 0$ than when $\mathcal{T}_1 = 3$ or $\mathcal{T}_1 = 15$ for two real-world datasets.

Impact of PIE memory and CF memory on ARE (*Fig.* 33*a, b*) In this experiment we set CF threshold $\mathcal{T}_1$ to 3. We observe that when the PIE memory is more than 7.2 MB, the ARE becomes higher for $M_{\text{cf}} = 12$ KB or $M_{\text{cf}} = 48$ KB than $M_{\text{cf}} = 0$ KB for IP trace dataset. And we observe that the impact of varying the CF memory is not remarkable when CF memory is larger than 12 KB. The 12 KB CF can filter nearly all the cold items, which appears less than $\mathcal{T}_1 = 3$ times.

Impact of PIE memory and CF threshold on ARE (*Fig.* 33*c, d*) In this experiment we set CF memory M_{cf} to 96 KB. We observe that the improvement of accuracy is better when $\mathcal{T}_1 = 3$ than when $\mathcal{T}_1 = 15$ for IP trace dataset. When PIE memory is more than 10 MB, the ARE is lower when $\mathcal{T}_1 = 0$ than when $\mathcal{T}_1 = 3$ or $\mathcal{T}_1 = 15$ for two real-world datasets.

Impact of PIE memory and CF memory on insertion speed (*Fig.* 34*a, b*) In this experiment we set CF threshold $\mathcal{T}_1$ to 3. We observe that the improvement of insertion speed is significant for any PIE memory and CF memory, comparing with $M_{\text{cf}} = 0$, but such impact is not remarkable when CF memory is larger than 12 KB. The 12 KB CF can filter nearly all the cold items, which appears less than $\mathcal{T}_1 = 3$ times.



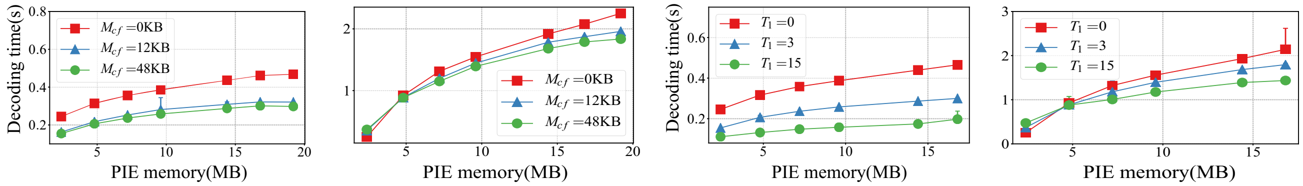
(a) Varying PIE memory and CF memory (IP trace).

(b) Varying PIE memory and CF memory (web page).

(c) Varying PIE memory and CF threshold (IP trace).

(d) Varying PIE memory and CF threshold (web page).

Fig. 34 Impact of parameters on insertion speed



(a) Varying PIE memory and CF memory (IP trace).

(b) Varying PIE memory and CF memory (web page).

(c) Varying PIE memory and CF threshold (IP trace).

(d) Varying PIE memory and CF threshold (web page).

Fig. 35 Impact of parameters on decoding time

Impact of PIE memory and CF threshold on insertion speed (*Fig.* 34*c, d*) In this experiment we set CF memory M_{cf} to 96 KB. We observe that the improvement of insertion speed is significant for any PIE memory and CF threshold, comparing with $\mathcal{T}_1 = 0$. And we observe that the higher CF threshold can provide higher insertion speed. This is because the CF with higher threshold can filter more items.

Impact of PIE memory and CF memory on decoding time (*Fig.* 35*a, b*) In this experiment we set CF threshold $\mathcal{T}_1$ to 3. We observe that the improvement of decoding speed is better for IP trace dataset than for web page dataset. And we observe that the impact of varying the CF memory is not remarkable when CF memory is larger than 12 KB. The 12 KB CF can filter nearly all the cold items, which appears less than $\mathcal{T}_1 =$ 3 times. The decoding time increases when increasing PIE memory, this is because PIE with larger memory can decode more persistent items, which increases decoding time.

Impact of PIE memory and CF threshold on decoding time (*Fig.* 35*c, d*) In this experiment we set CF memory M_{cf} to 96KB. We observe that the improvement of decoding speed is better for IP trace dataset than for web page dataset. The decoding time increases when increasing PIE memory, this is because PIE with larger memory can decode more persistent items, which increases decoding time.

Summary (1) Larger CF memory can usually lead to higher accuracy, insertion speed and decoding speed. But the improvement may be very little comparing with less CF memory. The ratio of CF memory is recommended to about 1%;

(2) The CF threshold $\mathcal{T}_1$ should be set according to the PIE memory. When total memory is very limited, which means the pure PIE algorithm can only report less than half persistent items, and a high CF threshold may much more efficiently improve recall rate, AAE and ARE. But when the total memory is adequate, which means recall rate can reach more than 0.8, the high CF threshold may decrease recall rate.

7 CF with parallel strategies

7.1 CF on multi-core platform (Pipeline)

Owing to the one-direction communication, the processing of items in CF can be seen as a form of pipeline, and this makes our CF fit in multi-core platform naturally. We divide the CF meta-framework into two parts: the CF (including aggregate-and-report and one-memory-access), and the specific stream processing algorithm (the CM–CU sketch, Space-Saving, FlowRadar, etc). These two parts can be placed on two cores: the CF on $\mathbf{C}_0$ and the specific algorithm on $\mathbf{C}_1$. Our CF naturally supports pipeline parallelism on multi-core platform. As shown in Fig. 36, We divide the CF meta-framework into two parts: the CF (including aggregate-and-report and one-memory-access) on core $\mathbf{C}_0$, and the specific stream processing algorithm (the CM–CU sketch, Space-Saving, FlowRadar, etc) on core $\mathbf{C}_1$. We employ the message passing interface to replace the shared memory accesses between these two cores. Specifically, when one item and its frequency

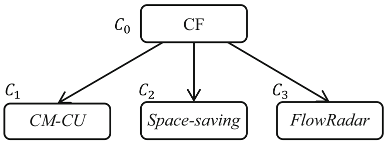


Fig. 36 CF in multi-task scenario

need to be reported to the specific stream processing algorithm, core $\mathbf{C}_0$ will forward them to core $\mathbf{C}_1$.

Then, the remaining job is executed by $\mathbf{C}_1$, and $\mathbf{C}_0$ can immediately start processing the next incoming item. Such pipeline operations improve the processing speed. Recall that only hot items (and a small portion of cold ones) need to be reported to the specific stream processing algorithm or the core $\mathbf{C}_1$, and the frequency of such report is low (e.g., less than 1 per 20 incoming items in real data streams when $\delta_1 = 4$). Therefore, in most cases the communication costs will not become the bottleneck, and we can also use a small batch to remove this potential bottleneck. As soon as core $\mathbf{C}_0$ forwards the message (key-value pair) to core $\mathbf{C}_1$, the remaining job can be performed by $\mathbf{C}_1$, and $\mathbf{C}_0$ can immediately start processing the next incoming item.

The above parallel architecture can process different measurement tasks of the same categories, but cannot process different tasks of different categories, because tasks of different categories should use different CF structures. In this section, we show how our parallel architecture works for the frequency-based measurement tasks.

7.2 CF in multi-task scenario

In practice, multiple tasks are often running simultaneously, and we call this the *multi-task scenario*. Fortunately, our CF fits well with the multi-task scenario: all tasks can share the same CF. We just need to set the threshold of CF to the maximum desired threshold among different tasks. We can also utilize the multi-core technique to further improve the processing speed. Specifically, core $\mathbf{C}_0$ (*master core*) performs CF (including aggregate-and-report and one-memory-access), and the other cores $\mathbf{C}_1, \mathbf{C}_2, \ldots$ (*slave cores*) perform specific stream processing algorithms. When one item and its frequency need to be reported to the specific stream processing algorithm (the specific threshold is reached in CF), master core will forward them to the corresponding slave core through the message passing interface. In our experiments, we find that each slave core stays idle in most time because it has much fewer items to process compared with the master core. To fully utilize the computational resources of slave cores, we manage to deploy multiple different stream processing algorithms on one slave core.

Another paper [74] also applies parallelism to frequency-based measurement tasks. The authors add a pre-filtering stage in front of a conventional frequent item algorithm (Space-Saving). And the parallelism means one CPU core runs the pre-filtering stage, while another CPU core runs the Space-Saving part. The parallel version of the algorithm is called *Parallel-Filter*. There are three main differences between "parallelism" approaches of Parallel-Filter and our Cold Filter. First, Parallel-Filter is a two-direction communication algorithm, while Cold Filter is an one-direction algorithm. In Parallel-Filter, the second stage needs to send messages to the first stage, and some important parameters in the first stage rely on such messages. However, as for Cold Filer, there are only messages from the first stage to the second, none in the opposite direction. Second, the kind of items hold in the two stages are different. In Parallel-Filter, items enter the pre-filtering stage first, which records heavy hitters. Only items that are not monitored by this stage will be forwarded to Space-Saving. In other words, the first stage records hot items while the second records cold items. However, our Cold Filter is to record cold items, while the second stage records hot items. Third, parallelism of Cold Filter also means multiple tasks running simultaneously on different cores on the second stage, while Parallel-Filter cannot deal with multi-task scenario. The Cold Filter offers to filter cold items, and the main task is done by the second stage, so different tasks running on different cores on the second stage can share one CF. However, as for Parallel-Filter, the first stage is used to directly pick out particularly frequent items and route them through a "shortcut" code path, which is hard to fit with multi-task scenario.

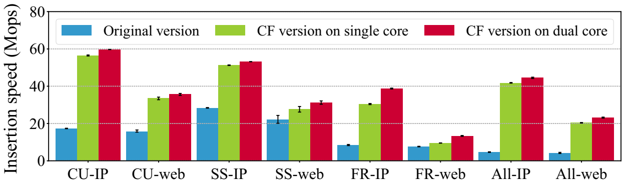


Fig. 37 Impact of multi-core on real-world datasets

7.3 Evaluation on CF with multi-core CPU

The results are shown in Fig. 37, where "All" means running these three algorithms simultaneously. *Our results show that the overall insertion speed of running three tasks with one shared CF on dual core is* 9.2 *times and* 5.5 *times faster than that without CF on a single core on the IP trace and web page datasets, respectively.* We observe that the multi-core technique helps improves the insertion speed of each task with CF compared to that on a single core. We find that the insertion speeds of two CF versions of "All" are higher than those of "FR". The reason is that in "All", all three tasks share the same CF that has a larger aggregate-and-report component than "FR" (see settings in Table 4).

8 Conclusion

In this paper, we propose a meta-framework named Cold Filter to enhance existing approximate stream processing algorithms. Our meta-framework is applicable to various stream processing tasks, and improves the accuracy and speed at the same time. We also present how to deploy it on four key stream processing tasks including estimating item frequency, finding top-*k* hot items, detecting heavy changes, and finding persistent items. Experimental results show that it significantly improves their processing speed and accuracy compared with the state-of-the-art solutions. Our Cold Filter meta-framework can be applied to many more approximate stream processing tasks, such as distribution of item frequencies, heavy hitters, information entropy, and improve their performance. All source code is released at Github [38].

Acknowledgements This work is supported by the National Key Research and Development Program of China (2018YFB1004403, 2016YFB1000304), NSFC (61672061, 61832001, and 61572039).

References

1. Cormode, G., Johnson, T., Korn, F., Muthukrishnan, S., Spatscheck, O., Srivastava, D.: Holistic UDAFs at streaming speeds. In: Proceedings of ACM SIGMOD, pp 35–46 (2004)
2. Manerikar, N., Palpanas, T.: Frequent items in streaming data: an experimental evaluation of the state-of-the-art. Data Knowl. Eng. **68**(4), 415–430 (2009)
3. Zhao, P., Aggarwal, C.C., Wang, M.: gSketch: on query estimation in graph streams. Proc. VLDB **5**, 193–204 (2011)
4. Roy, P., Khan, A., Alonso, G.: Augmented sketch: faster and more accurate stream processing. In: Proceedings of ACM SIGMOD, pp. 1449–1463 (2016)
5. Chen, B., Shrivastava, A.: Densified winner take all (WTA) hashing for sparse datasets. In: Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6–10, 2018, pp. 906–916 (2018)
6. Dobra, A., Garofalakis, M., Gehrke, J., Rastogi, R.: Processing complex aggregate queries over data streams. In: Proceedings of ACM SIGMOD, pp. 61–72. ACM (2002)
7. Cormode, G., Hadjieleftheriou, M.: Finding frequent items in data streams. Proc. VLDB **1**(2), 1530–1541 (2008)
8. Cormode, G., Garofalakis, M., Haas, P.J., Jermaine, C.: Synopses for massive data: samples, histograms, wavelets, sketches. Found. Trends Databases **4**(1–3), 1–294 (2012)
9. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Alg. **55**(1), 58–75 (2005)
10. Metwally, A., Agrawal, D., El Abbadi, A.: Efficient computation of frequent and top-k elements in data streams. In: International Conference on Database Theory, pp. 398–412. Springer (2005)
11. Charikar, M., Chen, K., Farach-Colton, M.: Finding frequent items in data streams. In: Widmayer, P., Eidenbenz, S., Triguero, F., Morales, R., Conejo, R., Hennessy, M. (eds.) Automata, Languages and Programming. Springer, Berlin (2002)
12. Schweller, R., Gupta, A., Parsons, E., Chen, Y.: Reversible sketches for efficient and accurate change detection over network data streams. In: Proceedings of ACM IMC, pp. 207–212. ACM (2004)
13. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.J.: How to summarize the universe: dynamic maintenance of quantiles. In: Proceedings of VLDB, pp. 454–465. VLDB Endowment (2002)
14. Luo, C., Shrivastava, A.: SSH (sketch, shingle, & hash) for indexing massive-scale time series. In: NIPS 2016 Time Series Workshop, pp. 38–58 (2017)
15. Shrivastava, A., Konig, A.C., Bilenko, M.: Time adaptive sketches (ada-sketches) for summarizing data streams. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1417–1432. ACM (2016)
16. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
17. Garofalakis, M., Gibbons, P.B.: Wavelet synopses with error guarantees. In: Proceedings of ACM SIGMOD, pp. 476–487. ACM (2002)
18. Guha, S., Koudas, N., Shim, K.: Data-streams and histograms. In: Proceedings of STOC, pp. 471–475. ACM (2001)
19. Kirsch, A., Mitzenmacher, M., Varghese, G.: Hash-based techniques for high-speed packet processing. In: Cormode, G., Thottan, M. (eds.) Algorithms for Next Generation Networks, pp. 181–218. Springer, London (2010)
20. Pandey, P., Bender, M.A., Johnson, R., Patro, R.: A general-purpose counting filter: Making every bit count. In: Proceedings of ACM SIGMOD, pp. 775–787
21. Thomas, D., Bordawekar, R., et al.: On efficient query processing of stream counts on the cell processor. In: Proceedings of IEEE ICDE (2009)
22. Yang, T., Liu, A.X., Shahzad, M., Zhong, Y., Fu, Q., Li, Z., Xie, G., Li, X.: A shifting bloom filter framework for set queries. Proc. VLDB **9**(5), 408–419 (2016)
23. Yang, T., Zhou, Y., Jin, H., Chen, S., Li, X.: Pyramid sketch: a sketch framework for frequency estimation of data streams. Proc. VLDB **10**(11), 1442–1453 (2017)
24. Zhou, Y., Liu, P., Jin, H., Yang, T., Dang, S., Li, X.: One memory access sketh: a more accurate and faster sketch for per-flow measurement. In: IEEE Globecom (2017)
25. Gong, J., Yang, T., Zhou, Y., Yang, D., Chen, S., Cui, B., Li, X.: Abc: a practicable sketch framework for non-uniform multisets. IEEE Bigdata (2017)
26. Wang, L., Cai, Z., Wang, H., Jiang, J., Yang, T., Cui, B., Li, X.: Fine-grained probability counting: Refined loglog algorithm. IEEE Bigcomp (2018)
27. Powers, D.M.: Applications and explanations of Zipf's law. In Proceedings on EMNLP-CoNLL. Association for Computational Linguistics (1998)
28. Adamic, L.A., Huberman, B.A.: Power-law distribution of the world wide web. Science **287**(5461), 2115–2115 (2000)
29. Goyal, A., Iii, Daume H., Cormode, G.: Sketch algorithms for estimating point queries in NLP. In: Proceedings of EMNLP (2012)
30. Mandal, A., Jiang, H., Shrivastava, A., Sarkar, V.: Topkapi: parallel and fast sketches for finding top-k frequent elements. In: Advances in Neural Information Processing Systems, pp. 10898–10908 (2018)
31. Henzinger, M.R.: Algorithmic challenges in web search engines. Internet Math. **1**(1), 115–123 (2004)
32. Li, Y., Miao, R., Kim, C., Yu, M.: Flowradar: a better netflow for data centers. In: Proceedings of USENIX NSDI, pp. 311–324 (2016)
33. Goodrich, M.T., Mitzenmacher, M.: Invertible bloom lookup tables. In: Proceedings of the 49th Annual Allerton Conference on Communication, Control, and Computing, pp. 792–799. IEEE (2011)
34. Xiao, Q., Qiao, Y., Zhen, M., Chen, S.: Estimating the persistent spreads in high-speed networks. In: 2014 IEEE 22nd International Conference on Network Protocols (ICNP), pp. 131–142. IEEE (2014)

35. Dai, H., Shahzad, M., Liu, A.X., Zhong, Y.: Finding persistent items in data streams. Proc. VLDB Endow. **10**(4), 289–300 (2016)
36. Shokrollahi, A.: Raptor codes. IEEE Trans. Inf. Theory **52**(6), 2551–2567 (2006)
37. Ganguly, S., Garofalakis, M., Rastogi, R.: Processing data-stream join aggregates using skimmed sketches. In: International Conference on Extending Database Technology, pp. 569–586. Springer (2004)
38. Source code related to cold filter meta-framework. https://github.com/zhouyangpkuer/ColdFilter. Accessed May 2018
39. Ting, D.: Data sketches for disaggregated subset sum and frequent item estimation. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1129–1140. ACM (2018)
40. Wei, Z., Luo, G., Yi, K., Du, X., Wen, J.-R.: Persistent data sketching. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 795–810. ACM (2015)
41. Peng, Y., Guo, J., Li, F., Qian, W., Zhou, A.: Persistent bloom filter: membership testing for the entire history. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1037–1052. ACM (2018)
42. Chen, J., Zhang, Q.: Bias-aware sketches. Proc. VLDB Endow. **10**(9), 961–972 (2017)
43. Wei, Z., Liu, X., Li, F., Shang, S., Du, X., Wen, J.-R.: Matrix sketching over sliding windows. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1465–1480. ACM (2016)
44. Agrawal N., Vulimiri, A.: Low-latency analytics on colossal data streams with summarystore. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 647–664. ACM (2017)
45. Cui, H., Keeton, K., Roy, I., Viswanathan, K., Ganger, G.R.: Using data transformations for low-latency time series analysis. In: Proceedings of the Sixth ACM Symposium on Cloud Computing, pp. 395–407. ACM (2015)
46. Rabkin, A., Arye, M., Sen, S., Pai, V.S., Freedman, M.J.: Aggregation and degradation in jetstream: streaming analytics in the wide area. NSDI **14**, 275–288 (2014)
47. Jiang, J., Fu, F., Yang, T., Cui, B.: SketchML: Accelerating distributed machine learning with data sketches. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1269–1284. ACM (2018)
48. Aghazadeh, A., Spring, R., LeJeune, D., Dasarathy, G., Shrivastava, A., Baraniuk, R.G.: MISSION: ultra large-scale feature selection using count-sketches. In: Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10–15, 2018, pp. 80–88 (2018)
49. Shrivastava, A.: Fast and accurate training of 100,000 classes on a single titan x. **(Preprint)**
50. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of ACM PODS, pp. 1–16. ACM (2002)
51. Muthukrishnan, S. et al.: Data streams: algorithms and applications. Found. Trends® Theor. Comput. Sci. **1**(2), 117–236 (2005)
52. Guo, C., Yuan, L., Xiang, D., et al.: Pingmesh: a large-scale system for data center network latency measurement and analysis. ACM SIGCOMM CCR **45**(4), 139–152 (2015)
53. Zhu, Y., Kang, N., Cao, J. et al.: Packet-level telemetry in large datacenter networks. In: ACM SIGCOMM CCR, vol. 45, pp. 479–491. ACM (2015)
54. Pagh, R., Rodler, F.: Lossy dictionaries. Algorithms—ESA 2001, pp. 300–311 (2001)
55. Intel SSE2 Documentation. https://software.intel.com/en-us/node/683883. Accessed May 2018
56. Zhou, Y., Yang, T., Jiang, J., Cui, B., Yu, M., Li, X., Uhlig, S.: Cold filter: a meta-framework for faster and more accurate stream processing. In: Proceedings of SIGMOD (2018)
57. Lu, Y., Montanari, A., Prabhakar, B., Dharmapurikar, S., Kabbani, A.: Counter braids: a novel counter architecture for per-flow measurement. ACM Sigmetrics Perform. Eval. Rev. **36**(1), 121–132 (2008)
58. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proceedings of VLDB, pp. 346–357. VLDB Endowment (2002)
59. Golab, L., DeHaan, D., Demaine, E.D., Lopez-Ortiz, A., Munro, J.I.: Identifying frequent items in sliding windows over on-line packet streams. In: Proceedings of ACM IMC, pp. 173–178. ACM (2003)
60. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst. (TODS) **28**(1), 51–55 (2003)
61. Roberts, S.: Control chart tests based on geometric moving averages. Technometrics **1**(3), 239–250 (1959)
62. Indyk, P.: Stable distributions, pseudorandom generators, embeddings and data stream computation. In: Proceedings of the 41st Annual Symposium on Foundations of Computer Science, pp. 189–197. IEEE (2000)
63. Krishnamurthy, B., Sen, S., Zhang, Y., Chen, Y.: Sketch-based change detection: methods, evaluation, and applications. In: Proceedings of ACM IMC, pp. 234–247. ACM (2003)
64. Schweller, R., Li, Z., Chen, Y., et al.: Reversible sketches: enabling monitoring and analysis over high-speed data streams. IEEE/ACM Trans. Netw. (ToN) **15**(5), 1059–1072 (2007)
65. Guha, S., McGregor, A.: Stream order and order statistics: quantile estimation in random-order streams. SIAM J. Comput. **38**(5), 2044–2059 (2009)
66. Wei, Z., Luo, G., Yi, K., Du, X., Wen, J.-R.: Persistent data sketching. In: Proceedings of ACM SIGMOD, pp. 795–810. ACM (2015)
67. The caida anonymized 2016 internet traces. http://www.caida.org/data/overview/. Accessed May 2018
68. Real-life transactional dataset. http://fimi.ua.ac.be/data/. Accessed May 2018
69. Rousskov, A., Wessels, D.: High-performance benchmarking with web polygraph. Softw.: Pract. Exp. **34**(2), 187–211 (2004)
70. Hash website. http://burtleburtle.net/bob/hash/evahash.html. Accessed May 2018
71. Ji, M., Yan, J., Gu, S., Han, J., He, X., Zhang, W.V., Chen, Z.: Learning search tasks in queries and web pages via graph regularization. In: Proceedings of ACM SIGIR, pp. 55–64. ACM (2011)
72. Goyal, A., Daume Iii, H., Cormode, G.: Sketch algorithms for estimating point queries in NLP. In: EMNLP-CoNLL, pp. 1093–1103 (2012)
73. Qiao, Y., Li, T., Chen, S.: One memory access bloom filters and their generalization. In: INFOCOM, 2011 Proceedings IEEE, pp. 1745–1753. IEEE (2011)
74. Roy, P., Teubner, J., Alonso, G.: Efficient frequent item counting in multi-core hardware. In: ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (2012)