Cold Filter: A Meta-Framework for Faster and More Accurate Stream Processing

Anonymous Author(s)

ABSTRACT

Approximate stream processing algorithms, such as Count-Min sketch, Space-Saving, *etc.*, support numerous applications in databases, storage systems, networking, and other domains. Unfortunately, because of the unbalanced distribution in real data streams, existing algorithms can hardly achieve small memory usage, fast processing speed, and high accuracy at the same time. To address this gap, we propose a meta-framework, called **C**old **F**ilter (**CF**), that enables faster and more accurate stream processing.

Different from existing filters that mainly focus on hot items, our filter captures cold items in the first stage, and hot items in the second stage. Also, existing filters 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, giving it a genericity that makes it applicable to many stream processing tasks. To illustrate the benefits of our filter, we deploy it on three typical stream processing tasks and experimental results show speed improvements of up to 4.7 times, and accuracy improvements of up to 51 times. All source code is made publicly available anonymously at Github [1].

1 INTRODUCTION

In many big data scenarios, the data comes as a high-speed stream [2–5], such as phone calls, videos, sensor data, network traffic, web clicks and crawls. Such data streams are often processed in a single-pass [5–8]. 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], and quantiles [13]. 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 is too high. Therefore, probabilistic data structures [11, 14–20] have become more popular for approximate processing.

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

Characteristics of Real Data Streams: According to our tests on real datasets and the literature [5, 15], in practice, the items present in real data streams often obey unbalanced distribution, such as Zipf [21] or Power-law [22]. 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 pose great challenges on stream processing tasks. Stream processing tasks can be divided into two kinds: the first needs to accurately record both hot and cold items, such as estimating item frequencies, and item frequency distribution. The second needs to accurately record only hot items, such as top-k and heavy changes. Next, we show examples of three key stream processing tasks.

Estimating Item Frequency: Estimating the frequency of each item is one of the most classic tasks in data streams [9, 15]. Two typical solutions are the Count-Min sketch [9] and the CM-CU sketch [23]. 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 will be hardly 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 much fewer than cold items in real data streams.

Finding Top-k Hot Items: Top-k issue is important in various fields, including in data streams [9–11, 15]. As we cannot store all incoming items and can only process each item 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, which are incurred by cold items and should be avoided, degrade the accuracy of the results of top-k.

Detecting Heavy Changes: The frequencies of some items could significantly change in a short time. Detecting such items is important for search engines [24] and security [12, 25]. The state-of-the-art solution is FlowRadar [25] that relies on an Invertible Bloom Lookup Table (IBLT) [26]. It uses an IBLT to approximately monitor all incoming items and their frequencies in two adjacent time windows, then compares their frequencies and draws conclusions. FlowRadar achieves high accuracy if there is enough memory to record every item, which might be impractical in many scenarios. Actually, in each time window, there are a large number of cold items, which are unnecessary to be recorded, and cost more memory than hot items.

In a nutshell, the characteristics of skewed data streams make the state-of-the-art algorithms hardly work well or require large amounts of resources. To address this challenge, there are several proposed algorithms to do filtering on data streams, such as the Augmented sketch [5], skimmed-sketch [27], *etc.* 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 could have fewer memory accesses. In deed, it is difficult to catch hot items accurately, because all hot items are initially cold and stored in the second stage, and then

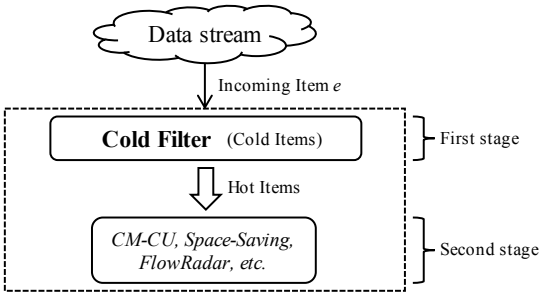


Figure 1: The Cold Filter captures unpopular (cold) items in the first stage, and forwards popular (hot) items to the second stage.

become hot. Therefore, existing algorithms need to be implemented using two-direction 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 need more memory accesses; 3) they make it hard to perform pipeline parallel. Our design goal is to *devise a filter that relies on one-direction communication, and targets at accurately estimating both hot items and cold items with a higher processing speed.*

Our **C**old **F**ilter (**CF**), as shown in Figure 1, uses a two-layer sketch with small counters to accurately record the frequencies of cold items[1]. If all the hashed counters overflow, CF will report the incoming item as a hot item (one-direction communication), and send it to the existing stream processing algorithms (*e.g.*, the CM-CU sketch, Space-Saving, and FlowRadar). We can combine CF with different existing algorithms in different ways and gain large benefits, and thus we call it a meta-framework. CF works with existing algorithms with limited modifications, but significantly improves accuracy. The first stage only uses small counters to store the frequencies of cold items, and thus is memory efficient. By filtering out a large number of cold items, the second stage concentrates on hot items, and thus can achieve high accuracy. To enhance 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*, to enable the three key tasks to achieve a higher processing speed. As our Cold Filter can accurately record the information of both cold items and hot items, it is applicable to most stream processing tasks.

After using our Cold Filter, on the IP trace datasets, the speed of CM-CU, Space-Saving, and FlowRadar is improved by 2.8~2.9, 1.7~1.9, and 4.7 times (Figure 6(a), 10(a), and 11(c)), respectively, the average absolute error (AAE) is reduced by up to 5 times for CM-CU (Figure 5(a)) and in average 51 times for Space-Saving (Figure 9(a)), and the memory usage of FlowRadar is reduced by in average 13 times (Figure 11(a)). The speedup and AAE reduction or memory usage reduction for each task can be achieved at the same time on the IP trace datasets.

[1] In deed, an optional part can exist in the first stage in Figure 1 in case more information is required about the cold items, *i.e.*, not only their frequencies, although the three key tasks in this paper do not need this part.

The contributions of this paper are the following:

- We propose a meta-framework named Cold Filter. It filters out cold items to enhance existing stream processing algorithms (*i.e.* , improving the accuracy and speed).
- We apply our meta-framework to three different tasks. Our experiments show significant performance improvements.

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, $A_1 ... A_d$, and each array consists of w counters. There are d hash functions, $h_1 ... h_d$, in the Count-Min sketch. When inserting an item e with frequency f, the Count-Min sketch increments all the d *hashed counters*, $A_1[h_1(e)] ... A_d[h_d(e)]$, by f. When querying an item e', it reports as the estimated frequency of this item 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 [23], 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 under-estimation error. More sketches are detailed in the survey [15].

The most relevant work about our Cold Filter is the Augmented sketch [5]. 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 the item e') in the filter, the Augmented sketch needs to expel the item e' to Φ, and insert e into the filter.

3 THE COLD FILTER META-FRAMEWORK

We employ the standard streaming model, namely the *cash register model* [28, 29]. Given a *whole data stream* $\mathcal{S}$ with E items and N distinct items, where $N \leqslant E$. $\mathcal{S} = (e_1, e_2, ..., e_E)$, where each item $e_i \in U = \{e_{\beta_1}, e_{\beta_2}, ..., e_{\beta_N}\}$. Note that items in U are distinct, while items in $\mathcal{S}$ may not. Let t be the current time point, e_t be the current incoming item, and $\mathcal{S}_t = (e_1, e_2, ..., e_t)$ be 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 $\mathcal{S} = (e_1, e_2, ..., e_E)$ and a current time point t, the current sub-stream is $\mathcal{S}_t = (e_1, e_2, ..., e_t)$. For the current item e_t, how to accurately and quickly estimate whether its *current* frequency $f_{e_t}[t]$ exceeds the predefined threshold $\mathcal{T}$?

3.1 A 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 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* in 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. But in real data streams, 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 suboptimal filtering performance. If instead we could automatically allocate small counters for cold items and large counters for hot items, then the allocated memory can be fully utilized. This is what our proposed solution achieves.

3.2 The Proposed Solution

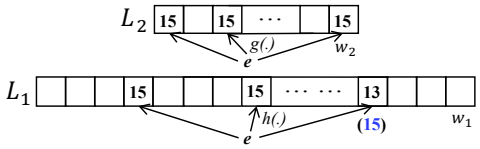


Figure 2: Data structure of two-layer CF.

As shown in Figure 2, our Cold Filter (CF[2]) consists of two layers: a low layer L_1, and a high layer L_2. These two layers consist 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. Specifically, there are two processes: update and report.

Algorithm 1: Stream processing algorithm for CF.

Input: The incoming item e_t.
Output: Update CF, and report whether $f_{e_t}[t] > \mathcal{T}$.

```
1  V1 ← min_{1≤i≤d1}(L1[hi(et)]);
2  if V1 < T1 then
3      foreach L1[hi(et)] (1 ≤ i ≤ d1) do
4          L1[hi(et)] ← max(V1 + 1, L1[hi(et)]);
5      return f_et[t] ⩽ T;
6  else
7      /* concurrently overflow at layer L1 */
8      V2 ← min_{1≤i≤d2}(L2[gi(et)]);
9      if V2 < T2 then
10         foreach L2[gi(et)] (1 ≤ i ≤ d2) do
11             L2[gi(et)] ← max(V2 + 1, L2[gi(et)]);
12         return f_et[t] ⩽ T;
13     else
14         /* concurrently overflow at layer L2 */
15         return f_et[t] > T;
```

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 that of the d_2 hashed counters at the high layer, respectively. If $V_1 < \mathcal{T}_1$, CF increments the smallest hashed counter(s) at the low layer by one (see line 4). 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 operation of only incrementing the smallest counters is always narrowing the differences of values of the d_1 hashed counters. If the values of one or more of these d_1 hashed counters reach $\mathcal{T}_1$, then all the subsequent increments will be added to the other counters. Therefore, the ultimate state is that all the 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 analogous to the one at the low layer: If $V_2 < \mathcal{T}_2$, CF increments the smallest hashed counter(s) by one (see line 11).

For the current item e_t, if its hashed counters concurrently overflow at the low layer, we must have $f_{e_t}[t-1] \leqslant V_1$[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 thus $f_{e_t}[t]$ 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 benefit of space and time efficiency, we choose to report $f_{e_t}[t] > \mathcal{T}$ only.

Example: As shown in Figure 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 negative and only a small portion of false positives. If $f_{e_t}[t]$ does exceed the 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 the threshold $\mathcal{T}$, CF may draw wrong conclusions (false positives).

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

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

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 Φ in naive solution, we set its counter size to 16 bits. Recall that w denotes the number of counters in Φ. 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 Φ. 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 collision. Further experiments about the selection of layer number are provided in §6.4.3.

3.3 Optimization 1: Aggregate-and-report

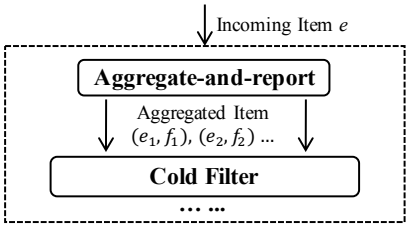


Figure 3: Aggregate-and-report strategy.

In real data streams, some items often appear many times across multiple continuous time points [30, 31]. This is called *stream burst*, which provides an opportunity to accelerate CF. We propose a strategy called aggregate-and-report, as shown in Figure 3. The key idea of this strategy is to *add an another small filter to aggregate the bursting items before CF, and then report the aggregated items and their frequencies (often much larger than one) under certain conditions*. This small filter can be implemented in different ways. A typical one is what the Augmented sketch does: scanning the whole queue and expelling the item with the minimum frequency. However, this method will suffer from low speed if the queue is large. What is 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 [32]: each item is hashed into a bucket, and each bucket consists of several items and their corresponding frequencies. We use SIMD (Single Instruction Multiple Data) [33] to scan a specific bucket.

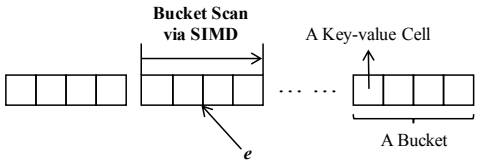


Figure 4: SIMD-based bucket scan.

Figure 4 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 key part records the item ID, while the value part 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 with the ID of the incoming item, we increment the corresponding value part;
(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 one. 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 in all buckets into CF. Since the value of reported aggregated frequency is often larger than one, we need to make some modifications to Algorithm 1. The principle of the modified algorithm is unchanged. The difference lies in how CF with aggregate-and-report strategy reaches the concurrent overflow state. The detailed procedure is shown in Algorithm 3 in the appendix. 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 $f - (\mathcal{T}_1 - V_1) - (\mathcal{T}_2 - V_2)$ (see Algorithm 3), instead of the default of 1 in Algorithm 1. Due to space limitations, we omit further explanations about this algorithm.

The bucket scan operation can be efficiently implemented with the SIMD instructions. The detailed procedure is shown in Algorithm 4 in the appendix. The *TZCNT* [34] in Algorithm 4 is an instruction added to the x86 instruction set with the Haswell microarchitecture. It returns the number of trailing 0-bits in the argument.

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 highly likely to become the bottleneck of the system. To handle this bottleneck, we propose the *one-memory-access* strategy tailored for only layer L_1. Before discussing this strategy, we need to 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 (the CM-CU sketch, Space-Saving, FlowRadar, *etc.*). 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 the 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 16-bit segment, and three 4-bit segments (discarding the remaining 4 bits). We use the 16-bit value to locate one machine word at layer L_1, and 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.

4 COLD FILTER DEPLOYMENT

4.1 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, *etc.*) to record the frequencies of hot items.

Insertion: When inserting an item, we first update 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: Next, 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 items from hot items. They use counters of a fixed size determined by the largest frequency to do the counting. 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 big 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.

4.2 Finding Top-k Hot Items

Prior art: There are two kinds of approaches to find top-k hot items: sketch-based and counter-based [35]. Sketch-based methods use a sketch (*e.g.*, the Count-Min sketch [9], the CM-CU sketch [23], and more [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 [36], Frequent algorithm [37, 38] and Space-Saving [10]. In this paper, we focus on Space-Saving, as it gains the most wide acceptances. 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 just increments its corresponding counter. Otherwise, it inserts e into the Stream-Summary if there is available space. If there is no available space, it creates new space by expelling the item with the minimum count (C_{min}) from the Stream-Summary, and stores e with count of $C_{min} + 1$ in this space. 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 will feed this item to Space-Saving.

Report: Below, we show how to report top-k hot items. After processing all the items in data streams, 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 k^{th} hottest item is larger than the threshold $\mathcal{T}$. In practice, to get the k^{th} largest frequency, we use the k^{th} largest frequencies from previous measurement periods to predict the k^{th} largest frequency of the current measurement period using *EWMA* [39], with some history weight. Obviously, the larger $\mathcal{T}$ is, the more accurate the results will be. Therefore, we set $\mathcal{T} = \mathcal{F} \times \alpha$ ($\alpha \to 1$), where $\mathcal{F}$ is the predicted frequency.

Discussion: Next, 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, making the recorded frequencies highly over-estimated, since each exchange leads to one increment operation in the counter associated with the expelled item. High 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.

4.3 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*. Formally, assume the data stream during the first time window has the frequency vector $\mathbf{f}_1 =< f_{1e_1}, f_{1e_2}, ..., f_{1e_L} >$ where f_{1e_i} denotes the frequency of item e_i (picked from the universe $U = \{e_1, e_2, ..., e_L\}$). Similarly, we have $\mathbf{f}_2 =< f_{2e_1}, f_{2e_2}, ..., f_{2e_L} >$ 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 [40]. The k-ary sketch [41] 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 [42], based on the k-ary sketch, can infer the IDs of culprit items in 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 [25], encodes fast each distinct item and its frequency in an extended IBLT (Invertible Bloom Lookup Table) [26] with the aid of a Bloom filter [43], 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 that 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 2. 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.

Algorithm 2: 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 ← ∅;
foreach e ∈ S1 ∪ S2 do
    /* f1e: e's frequency in the first time window */
    if e ∈ S1 then f1e ← f1e^I + T ;
    else f1e ← Q1(e) ;
    /* f2e: e's frequency in the second time window */
    if e ∈ S2 then f2e ← f2e^I + T ;
    else f2e ← Q2(e) ;
    if |f1e − f2e| ⩾ φ · D then C ← C ∪ {e} ;
return C;
```

Discussion: Next, we discuss why FlowRadar with CF requires less memory than the standard FlowRadar. According to the literature [25], 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 large 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.

5 FORMAL ANALYSIS OF CF

Given a time window $[1, E]$, the data stream $\mathcal{S} = (e_1, e_2, ..., e_E)$ contains E items with N unique items $e_{\beta_1}, e_{\beta_2}, ..., e_{\beta_N}$. Within this time window, we construct a CF with threshold $\mathcal{T}$ ($= \mathcal{T}_1 + \mathcal{T}_2$) for the data stream. Before we get into the formal analysis of the performance of CF, we first need to depict the frequency distribution of this data stream.

Definition 5.1. *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 whole 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 consisting of these misreported items. To depict the filtering performance of CF formally, we define the misreport rate P_{mr} as follows:

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

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 use the theories of standard Bloom filter to derive the P_{mr} of CM-CU.

The Standard Bloom Filter: A standard Bloom filter [43] 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 the d hashed bits are one, it reports true; otherwise, it reports false. The standard Bloom filter only has false positive errors, no false negative errors. 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:

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

We have the following lemma:

Lemma 5.1. *Function $\overline{P}_{fp}(w, d, x) = \frac{1}{x}\sum_{i=0}^{x-1} P_{fp}(w, d, i)$, $\forall x \in N^+$ is a monotonic increasing function of x.*

The detailed proof is provided in Appendix B.1.

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

[4] $[E] = \{1, 2, ..., E\}$.
[5] Z^+ is the set of non-negative integers.

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 (from the low level to the high level) the last Bloom filter that reports true for this item. Then we report the level of this Bloom filter as the estimated frequency of this item.

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 misreport rate of CM-CU, we can rely on the one of the multi-layer Bloom filter.

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

Definition 5.2. *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 5.2. *The item subsets $I_k[j]$ and $J_k[j]$, defined in Definition 5.1 and 5.2, have the following relation:*

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

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

The detailed proof is provided in Appendix B.2. Since $\overline{P}_{fp}(w, d, x)$ is a monotonic 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 bound and upper bound of $|J_k[j]|$, respectively.

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 streams. We can use Gaussian, Poisson or other distributions to model it. Without loss of generality, we employ the *random order model* [44, 45] defined as follows:

Definition 5.3 (Random order model). *Let $\mathcal{P}$ be an arbitrary frequency distribution over distinct item set $U = \{e_{\beta_1}, e_{\beta_1}, ..., e_{\beta_N}\}$. At each time point, the incoming item in the stream is picked independently and uniformly at random from U according to $\mathcal{P}$.*

Theorem 5.3. *Under the random order model, the misreport rate of the multi-layer Bloom filter is bounded by:*

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

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

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

We provide the detailed proof in the Appendix B.3.

The Bound of P_{mr} of CF: For (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 5.4. *For each time point $j \in [E]$,*

$$\begin{aligned} P^U_{pf}(|J_x[j]|) &= \begin{cases} P_{fp}(w_1, d_1, |J_x[j]|) \ (1 \leqslant x \leqslant \mathcal{T}_1) \\ P_{fp}(w_2, d_2, |J_x[j]|) \ (\mathcal{T}_1 + 1 \leqslant x \leqslant \mathcal{T}) \end{cases} \\ \overline{P}^U_{pf}(|J_x[j]|) &= \begin{cases} \frac{1}{|J_x[j]|} \sum_{i=0}^{|J_x[j]|-1} P_{fp}(w_1, d_1, i) \ (1 \leqslant x \leqslant \mathcal{T}_1) \\ \frac{1}{|J_x[j]|} \sum_{i=0}^{|J_x[j]|-1} P_{fp}(w_2, d_2, i) \ (\mathcal{T}_1 + 1 \leqslant x \leqslant \mathcal{T}) \end{cases} \end{aligned} \tag{6}$$

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

Lemma 5.4. *The item subsets $I_k[j]$ and $J_k[j]$, defined in Definition 5.1 and 5.2, have the following relation:*

$$|I_k[j]| \leqslant |J_k[j]| \leqslant |I_k[j]| + \sum_{i=1}^{k-1} \left[(|I_i[j]| - |I_{i+1}[j]|) \times \prod_{u=1}^{i} \overline{P}^U_{fp}(|J_{l_u}[j]|) \right] \tag{7}$$

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

The detailed derivation process for this lemma is similar to Lemma 5.2, hence we omit it. Similarly, we can get $|J_k[j]|^L$ and $|J_k[j]|^U$ from this lemma.

Theorem 5.5. *Under the random order model, the misreport rate of (two-layer) CF is bounded by:*

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

where $l_1, ..., l_i, ..., l_\lambda$ is the permutation of $1, 2, ..., \lambda$ that makes the sequence $P_{fp}(w, d, |J_{l_i}[t_u]|)$ 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 5.3, and we omit it.

6 PERFORMANCE EVALUATION

6.1 Experimental Setup

Datasets:

1) IP Trace Datasets: We use the anonymized IP trace streams collected in 2016 from CAIDA [46]. Each flow is identified by its source IP address (4 bytes). We use the first 256M 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 [47]. This dataset is built from a crawled collection of web pages. Each item (4 bytes) records the number of distinct terms of one web page. We use the first 256M items from the raw dataset, and uniformly divide them into 8 sub-datasets, each of which has around 0.9M distinct items.

These two types of datasets have the same number of items, but have different numbers of distinct items, because they have different

Table 1: Parameter setting for CF.

	$\mathcal{T}_2$	d_b	w_c	$M_1 : M_2$
CM/CM-CU [9, 23]	241	1000	16	13 : 7
Space-Saving [10]	–	96	16	7 : 13
FlowRadar [25]	241	200	16	13 : 7

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, 0.4M vs. 0.9M. For all the experiments conducted on the above two types of datasets, we will plot their 5^{th} and 95^{th} percentile error bars across the corresponding 8 sub-datasets.

3) Synthetic Datasets: We generated 11 datasets following the Zipf [21] 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 [48].

Implementation: We have implemented CM (Count-Min for short), CM-CU, min-heap, SS (Space-Saving), FR (FlowRadar), ASketch (Augmented sketch) 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 [49]) with different initial seeds.

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 1 lists the other (part of) parameter setting 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 2MB 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 [5] 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, because it can only capture a small number of hottest items (around 32 items) while we will vary k starting from 32 to 1024. Recall that H denotes the number of item-counter pairs in SS. For SS with CF, we set $H = 2.5k, M_{cf} = 200$KB. We use the actual frequency of the k^{th} hottest item from one extra dataset (*e.g.*, the 9^{th} IP trace or web page dataset) as the prediction of that of 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 $H = 2.5k + M_{cf}/(100\text{bytes}) = 2.5k + 2048$ 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 memory on average. Therefore, we allocate $M_{cf} + 2.5k * 100\text{bytes} - k * 100\text{bytes} = M_{cf} + 150k$ bytes memory to CM/CM-CU.

[6] The authors in literature [50, 51] recommend using small number of hash functions.

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 [26, 52]). 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.

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 total DRAM memory. Each processor has three levels of cache memory: one 32KB L1 data caches and one 32KB L1 instruction cache for each core, one 256KB L2 cache for each core, and one 15MB L3 cache shared by all cores.

6.2 Metrics

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.

Precision Rate (PR) in Top-k and Heavy Changes: ratio of the number of correctly reported instances to the number of reported instances.

Recall Rate (CR) in Heavy Changes: ratio of the number of correctly reported instances to the number of correct instances.

Memory Threshold ($\mathcal{T}_m$) in Heavy Changes: the least total memory usage of FlowRadar (with CF) when the F_1 score [53] ($= \frac{2 \times PR \times CR}{PR + CR}$) reaches 99%.

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

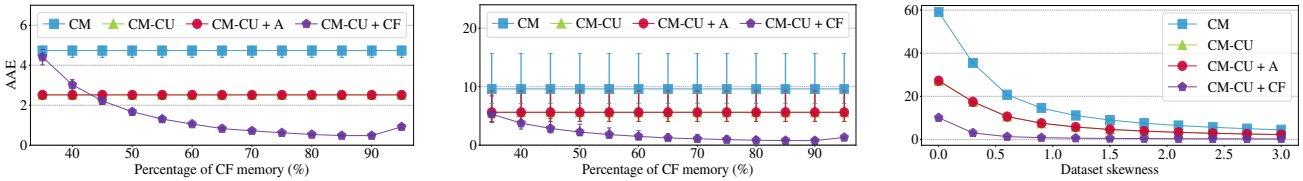
6.3 Evaluation on Three Key Tasks

6.3.1 *Estimating Item Frequency.*

Accuracy (Figure 5(a)-(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 Figure 5(c). 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.

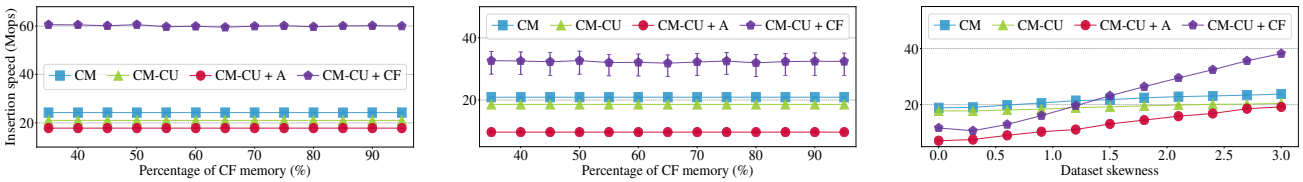
Insertion Speed (Figure 6(a)-(b)): Our results show that the insertion speed of CM-CU with CF is about 2.5, 2.9 and 3.4 times, and 1.6, 1.7 and 3.4 times faster than CM, CM-CU and CM-CU with ASketch when M_{cf}/M_t is set to 90% on two real-world datasets, respectively. ASketch lowers the insertion speed 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 Figure 6(c). 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

[7] The literature [54] reported $skewness > 1.4$ in real data streams.



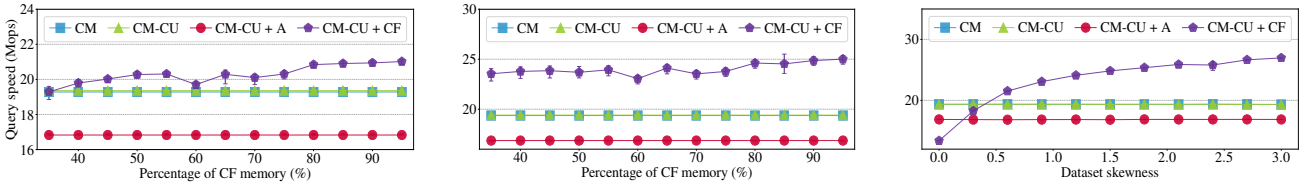
(a) Varying percentage of CF memory (IP trace). (b) Varying percentage of CF memory (web page). (c) Varying dataset skewness.

Figure 5: AAE vs. percentage of CF memory on two real-world datasets, and vs. skewness of synthetic dataset.



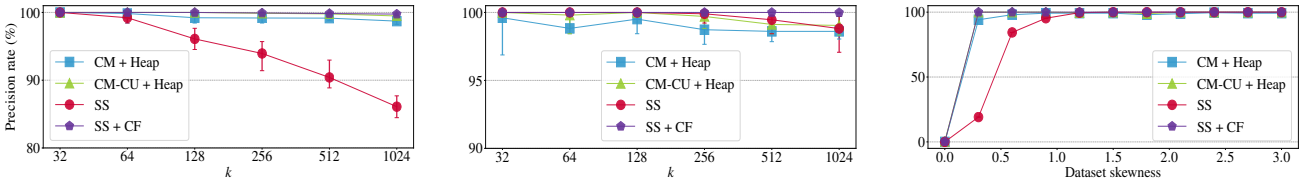
(a) Varying percentage of CF memory (IP trace). (b) Varying percentage of CF memory (web page). (c) Varying dataset skewness.

Figure 6: Insertion speed vs. percentage of CF memory on two real-world datasets, and vs. skewness of synthetic dataset.



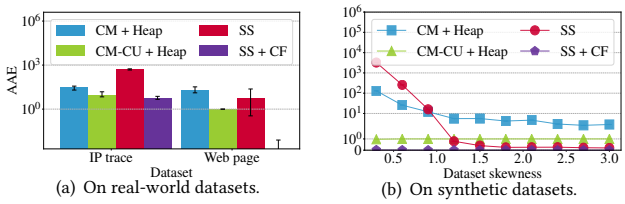
(a) Varying percentage of CF memory (IP trace). (b) Varying percentage of CF memory (web page). (c) Varying dataset skewness.

Figure 7: Query speed vs. percentage of CF memory on two real-world datasets, and vs. skewness of synthetic dataset.



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

Figure 8: Precision rate vs. k on two real-world datasets, and vs. skewness of synthetic dataset.



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

Figure 9: AAE on real-world and synthetic datasets.

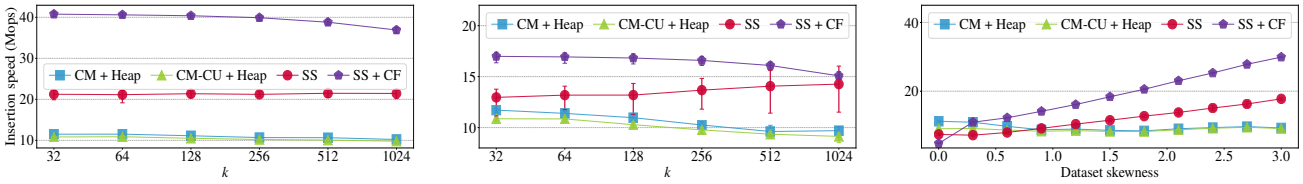
of items in synthetic datasets is fully randomized (while stream burst often happens in real data streams, see §3.3), which largely weakens the aggregating performance of the aggregate-and-report component and degrades the speed.

Query Speed (Figure 7(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 significantly 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 Figure 7(c). 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.

6.3.2 *Finding Top-k Hot Items.*

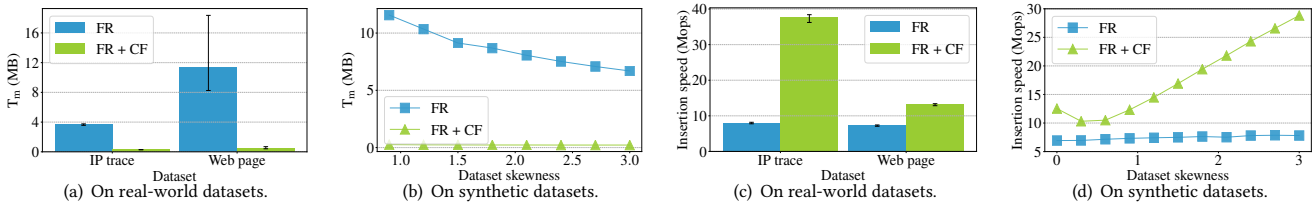
The aforementioned four approaches have the same query process, and thus we skip query speed below.

Accuracy (Figure 8(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 Figure 8(c). 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 $skewness \geqslant 0.3$, 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 Figure 9(a)-(b). On all datasets, SS with CF achieves a much lower AAE than the other three approaches.



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

Figure 10: Insertion speed vs. k on two real-world datasets, and vs. skewness of synthetic dataset.



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

Figure 11: T_m and insertion speed on real-world and synthetic datasets.

Insertion Speed (Figure 10(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 Figure 10(c). Here, k is set to 256. When $skewness \geqslant 0.6$, SS with CF achieves a higher insertion speed than the other three approaches.

6.3.3 *Detecting Heavy Changes.*

We detect heavy changes using threshold ϕ of 0.04% between the first 16M and the second 16M 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 (Figure 11(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 (§4.3), 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 (Figure 11(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 547ms and 30ms, respectively. On Web page datasets, the average query time is 1066ms and 27ms, respectively. Due to space limitations, we do not plot them.

6.4 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 to be $AAE_{pure}/AAE_{with\ CF}$. For SS, its accuracy improvement is defined to be $PR_{with\ CF}/PR_{pure}$. For FR, its accuracy improvement is defined to be $\mathcal{T}_{m_{pure}}/\mathcal{T}_{m_{wtih\ CF}}$. For all the four algorithms, their speedups are defined to be $speed_{with\ CF}/speed_{pure}$.

6.4.1 *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. We also evaluate how Agg (aggregate-and-report for short) solely influences the performance of these four algorithms. We set $M_1 + M_2 = 1$MB, $d_b = 1000$ for each CF, and the rest of parameters are the same as in §6.1. In this subsection, CF refers to the pure Cold Filter without any optimization.

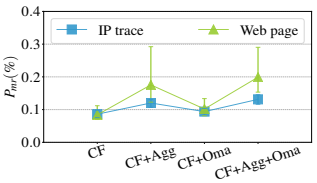
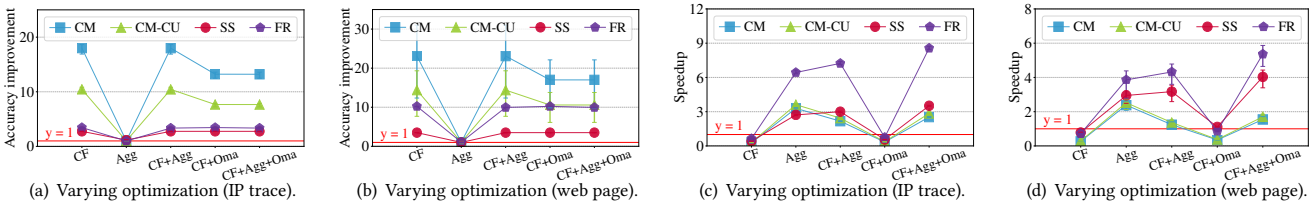


Figure 12: Impact of different optimizations on P_{mr}.

Impact on P_{mr} (Figure 12): Here, we set $\mathcal{T} = 256$. We observe that both Agg and Oma (one-memory-access) elevate 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}.

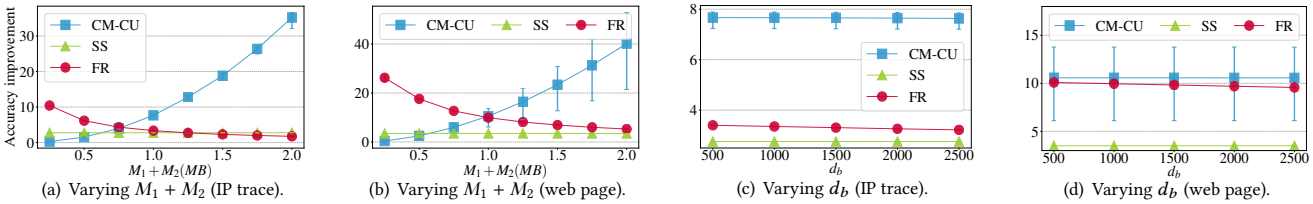
Impact on Accuracy Improvement (Figure 13(a)-(b)): In average, on both datasets, for CM and CM-CU, the percentages of accuracy improvement contributed by CF, Agg, and Oma are around 140%, 0%, and −40%, respectively; for SS and FR, they are around 100%, 0%, and 0%. In other words, CF helps each algorithm 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 (Figure 13(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.



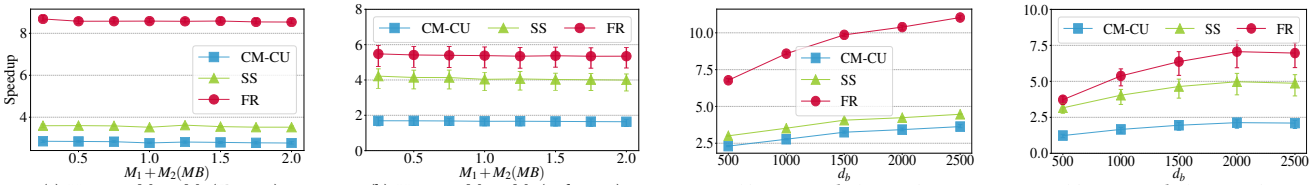
(a) Varying optimization (IP trace). (b) Varying optimization (web page). (c) Varying optimization (IP trace). (d) Varying optimization (web page).

Figure 13: Impact of different optimizations on the accuracy and speed of Count-Min, CM-CU, Space-Saving and FlowRadar.



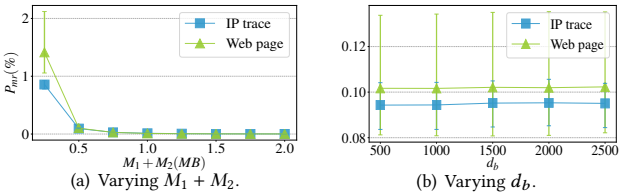
(a) Varying $M_1 + M_2$ (IP trace). (b) Varying $M_1 + M_2$ (web page). (c) Varying d_b (IP trace). (d) Varying d_b (web page).

Figure 14: 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).

Figure 15: Impact of memory budget of CF on the speed of CM-CU, Space-Saving and FlowRadar.



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

Figure 16: Impact of memory budget of CF on P_{mr}.

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 §4.2). 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 Figure 6(a) and 11(c)). 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.

6.4.2 *Impact of Memory Budget of CF.*

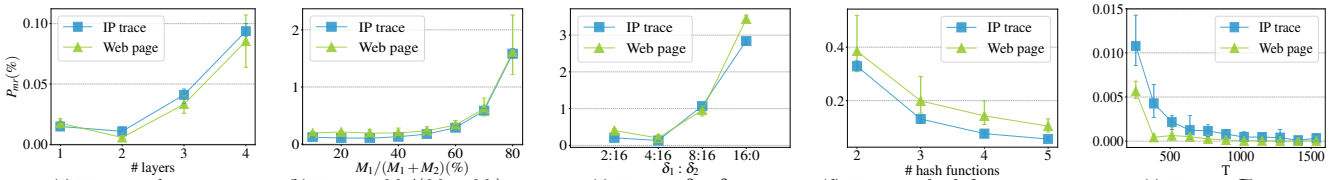
Next, we focus on the impact of memory budget of CF (*i.e.*, M_1+M_2 and d_b) on P_{mr}, and accuracy improvements and speedups of CM-CU, SS, and FR. We set $M_1 + M_2 = 1$MB and $d_b = 1000$ by default, and the rest of parameters are the same as in §6.1.

Impact on P_{mr} (Figure 16(a)-(b)): Here, we set $\mathcal{T} = 256$. When $M_1 + M_2 \geqslant 0.5$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 (Figure 14(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 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.

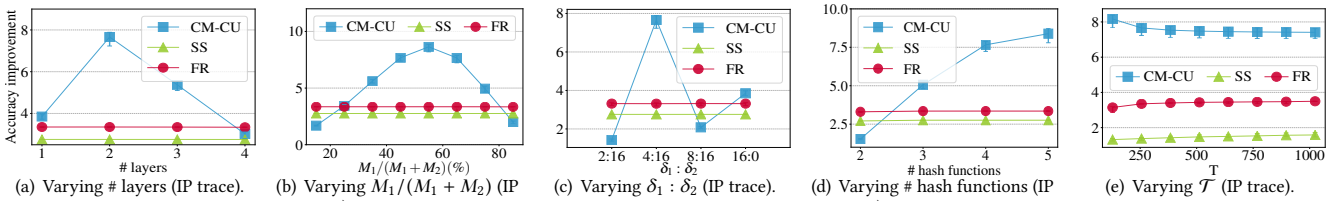
Impact on Speedup (Figure 15(a)-(d)): $M_1 + M_2$ makes little impact on the speedup of each algorithm on both datasets. Larger d_b leads to a higher speedup of each algorithm on both datasets, except that on web page datasets, the speedup begins to decrease a little when $d_b > 2000$. The reason for 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.



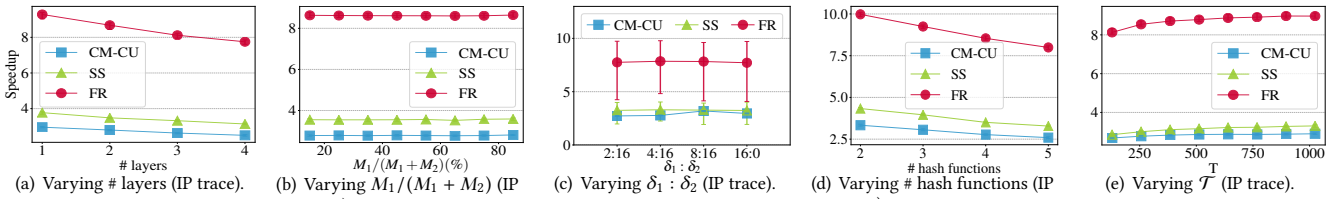
(a) Varying # layers. (b) Varying $M_1/(M_1 + M_2)$. (c) Varying $\delta_1 : \delta_2$. (d) Varying # hash functions. (e) Varying $\mathcal{T}$.

Figure 17: 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).

Figure 18: 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).

Figure 19: Impact of parameter setting in CF on the speed of CM-CU, Space-Saving and FlowRadar.

6.4.3 *Impact of Parameter Setting in CF.*

Below, we focus on the impact of parameter setting (including layer number, $M_1/(M_1+M_2)$, $\delta_1 : \delta_2$, 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 $d_b = 1000$, and the rest of parameters are the same as in §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} (Figure 17(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 (Figure 18(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, $M_1/(M_1 + M_2)$, $\delta_1 : \delta_2$, 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 k^{th} 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 (Figure 19(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 $\delta_1 : \delta_2$ make little impact on the speed; larger $\mathcal{T}$ leads to a higher speed, but such impact is not remarkable.

Summary: 1) Two layers, $\delta_1 : \delta_2 = 4 : 16$ 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 k^{th} hottest item.

7 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 three key stream processing tasks including estimating item frequency, finding top-k hot items, and detecting heavy changes. 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, *etc.*, and improve their performance. All source code is released anonymously at Github [1].

REFERENCES

[1] Source code related to cold filter meta-framework. https://github.com/streamclassifier/ColdFilter.

[2] Graham Cormode, Theodore Johnson, Flip Korn, Shan Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Holistic udafs at streaming speeds. In *Proc. ACM SIGMOD*, pages 35–46, 2004.

[3] Nishad Manerikar and Themis Palpanas. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data & Knowledge Engineering*, 68(4):415–430, 2009.

[4] Peixiang Zhao, Charu C Aggarwal, and Min Wang. gsketch: on query estimation in graph streams. *Proc. VLDB*, 2011.

[5] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proc. ACM SIGMOD*, pages 1449–1463, 2016.

[6] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proc. ACM SIGMOD*, pages 61–72. ACM, 2002.

[7] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB*, 1(2):1530–1541, 2008.

[8] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[9] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[10] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005.

[11] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*. Springer, 2002.

[12] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proc. ACM IMC*, pages 207–212. ACM, 2004.

[13] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin J Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *Proc. VLDB*, pages 454–465. VLDB Endowment, 2002.

[14] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[15] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

[16] Minos Garofalakis and Phillip B Gibbons. Wavelet synopses with error guarantees. In *Proc. ACM SIGMOD*, pages 476–487. ACM, 2002.

[17] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-streams and histograms. In *Proc. STOC*, pages 471–475. ACM, 2001.

[18] Adam Kirsch, Michael Mitzenmacher, and George Varghese. Hash-based techniques for high-speed packet processing. In *Algorithms for Next Generation Networks*, pages 181–218. Springer, 2010.

[19] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proc. ACM SIGMOD*, pages 775–787.

[20] Dina Thomas, Rajesh Bordawekar, and et al. On efficient query processing of stream counts on the cell processor. In *Proc. IEEE ICDE*, 2009.

[21] David MW Powers. Applications and explanations of Zipf's law. In *Proc. EMNLP-CoNLL*. Association for Computational Linguistics, 1998.

[22] Lada A Adamic and Bernardo A Huberman. Power-law distribution of the world wide web. *science*, 287(5461):2115–2115, 2000.

[23] Amit Goyal, Daume, Hal Iii, and Graham Cormode. Sketch algorithms for estimating point queries in nlp. In *Proc. EMNLP*, 2012.

[24] Monika R Henzinger. Algorithmic challenges in web search engines. *Internet Mathematics*, 1(1):115–123, 2004.

[25] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: a better netflow for data centers. In *Proc. USENIX NSDI*, pages 311–324, 2016.

[26] Michael T Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *Proceedings of the 49th Annual Allerton Conference on Communication, Control, and Computing*, pages 792–799. IEEE, 2011.

[27] Sumit Ganguly, Minos Garofalakis, and Rajeev Rastogi. Processing data-stream join aggregates using skimmed sketches. In *International Conference on Extending Database Technology*, pages 569–586. Springer, 2004.

[28] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proc. ACM PODS*, pages 1–16. ACM, 2002.

[29] Shanmugavelayutham Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005.

[30] Chuanxiong Guo, Lihua Yuan, Dong Xiang, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. *ACM SIGMCOMM CCR*, 45(4):139–152, 2015.

[31] Yibo Zhu, Nanxi Kang, Jiaxin Cao, et al. Packet-level telemetry in large datacenter networks. In *ACM SIGMCOMM CCR*, volume 45, pages 479–491. ACM, 2015.

[32] Rasmus Pagh and Flemming Rodler. Lossy dictionaries. *AlgorithmsâĂŤESA 2001*, pages 300–311, 2001.

[33] Intel SSE2 Documentation. https://software.intel.com/en-us/node/683883.

[34] Intel Advanced Vector Extensions Programming Reference. https://software.intel.com/file/36945. (Cited on page 498).

[35] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB*, 1(2):1530–1541, 2008.

[36] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. VLDB*, pages 346–357. VLDB Endowment, 2002.

[37] Lukasz Golab, David DeHaan, Erik D Demaine, Alejandro Lopez-Ortiz, and J Ian Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proc. ACM IMC*, pages 173–178. ACM, 2003.

[38] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)*, 28(1):51–55, 2003.

[39] SW Roberts. Control chart tests based on geometric moving averages. *Technometrics*, 1(3):239–250, 1959.

[40] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 189–197. IEEE, 2000.

[41] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proc. ACM IMC*, pages 234–247. ACM, 2003.

[42] Robert Schweller, Zhichun Li, Yan Chen, et al. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Transactions on Networking (ToN)*, 15(5):1059–1072, 2007.

[43] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[44] Sudipto Guha and Andrew McGregor. Stream order and order statistics: Quantile estimation in random-order streams. *SIAM Journal on Computing*, 38(5):2044–2059, 2009.

[45] Zhewei Wei, Ge Luo, Ke Yi, Xiaoyong Du, and Ji-Rong Wen. Persistent data sketching. In *Proc. ACM SIGMOD*, pages 795–810. ACM, 2015.

[46] The caida anonymized 2016 internet traces. http://www.caida.org/data/overview/.

[47] Real-life transactional dataset. http://fimi.ua.ac.be/data/.

[48] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 34(2):187–211, 2004.

[49] Hash website. http://burtleburtle.net/bob/hash/evahash.html.

[50] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter braids: a novel counter architecture for per-flow measurement. *Acm Sigmetrics Performance Evaluation Review*, 36(1):121–132, 2008.

[51] Amit Goyal, Hal DaumÃĺ Iii, and Graham Cormode. Sketch algorithms for estimating point queries in nlp. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1093–1103, 2012.

[52] Yan Qiao, Tao Li, and Shigang Chen. One memory access bloom filters and their generalization. In *INFOCOM, 2011 Proceedings IEEE*, pages 1745–1753. IEEE, 2011.

[53] Ming Ji, Jun Yan, Siyu Gu, Jiawei Han, Xiaofei He, Wei Vivian Zhang, and Zheng Chen. Learning search tasks in queries and web pages via graph regularization. In *Proc. ACM SIGIR*, pages 55–64. ACM, 2011.

[54] Nishad Manerikar and Themis Palpanas. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data & Knowledge Engineering*, 68(4):415–430, 2009.

A ALGORITHM

B PROOF

B.1 Proof of Lemma 5.1

Proof. Obviously, $P_{fp}(w, d, n)$ given by Eq. 2 is a monotonic increasing function of the non-negative variable n. And the outputs of this function locate in the window $[0, 1]$. For convenience, we use sequence $\{a_i\}$ to donate $P_{fp}(w, d, i)$. To prove Lemma 5.1, we need to prove:

$$\forall k \in N^+, \frac{1}{k}\sum_{i=0}^{k-1} a_i < \frac{1}{k+1}\sum_{i=0}^{k} a_i \Leftrightarrow (k+1)\sum_{i=0}^{k-1} a_i < k\sum_{i=0}^{k} a_i \Leftrightarrow \sum_{i=0}^{k-1} a_i < k a_k \tag{10}$$

Since a_i is a monotonic increasing sequence, we have $a_0 < a_k, a_1 < a_k, ..., a_{k-1} < a_k,\ \forall k \in N^+$. Adding up these k inequalities, we get $\sum_{i=0}^{k-1} a_i < k a_k,\ \forall k \in N^+$. Therefore, the lemma holds. □

Algorithm 3: Stream processing algorithm for CF with gather-and-report strategy.

Input: The reported item e_t with its aggregated frequency f.
Output: Update CF, and return whether $f_{e_t}[t] > \mathcal{T}$.

$V_1 \leftarrow \min_{1 \leqslant i \leqslant d_1}(L_1[h_i(e_t)])$;
if $V_1 + f \leqslant \mathcal{T}_1$ **then**
  foreach $L_1[h_i(e_t)]$ $(1 \leqslant i \leqslant d_1)$ **do**
    $L_1[h_i(e_t)] \leftarrow \max(V_1 + f, L_1[h_i(e_t)])$;
  return $f_{e_t}[t] \leqslant \mathcal{T}$;
else
  / concurrently overflow at layer L_1 */*
  foreach $L_1[h_i(e_t)]$ $(1 \leqslant i \leqslant d_1)$ **do**
    $L_1[h_i(e_t)] \leftarrow \mathcal{T}_1$;
  $\Delta_1 \leftarrow \mathcal{T}_1 - V_1$;
  $f \leftarrow f - \Delta_1$;
  $V_2 \leftarrow \min_{1 \leqslant i \leqslant d_2}(L_2[g_i(e_t)])$;
  if $V_2 + f \leqslant \mathcal{T}_2$ **then**
    foreach $L_2[g_i(e_t)]$ $(1 \leqslant i \leqslant d_2)$ **do**
      $L_2[g_i(e_t)] \leftarrow \max(V_2 + f, L_2[g_i(e_t)])$;
    return $f_{e_t}[t] \leqslant \mathcal{T}$;
  else
    / concurrently overflow at layer L_2 */*
    foreach $L_2[g_i(e_t)]$ $(1 \leqslant i \leqslant d_2)$ **do**
      $L_2[g_i(e_t)] \leftarrow \mathcal{T}_2$;
    $\Delta_2 \leftarrow \mathcal{T}_2 - V_2$;
    $f \leftarrow f - \Delta_2$;
    return $f_{e_t}[t] > \mathcal{T}$;

Algorithm 4: Scanning a bucket with 16 cells through the SSE2 SIMD instructions.

Input: The incoming item e, and the start address p of the key part array in the hashed bucket.
Output: Return the index of the matched key or −1.

```
/* load the item ID into a SSE2 register */
const __m128i item = _mm_set1_epi32 (e);
/* convert address type */
__m128i * keys_p = (__m128i *) p;
/* compare the item ID with the 16 keys */
__m128i a_comp = _mm_cmpeq_epi32(item, keys_p[0]);
__m128i b_comp = _mm_cmpeq_epi32(item, keys_p[1]);
__m128i c_comp = _mm_cmpeq_epi32(item, keys_p[2]);
__m128i d_comp = _mm_cmpeq_epi32(item, keys_p[3]);
/* get the final matching results */
a_comp = _mm_packs_epi32(a_comp, b_comp);
c_comp = _mm_packs_epi32(c_comp, d_comp);
a_comp = _mm_packs_epi32(a_comp, c_comp);
int matched = _mm_movemask_epi8(a_comp);
/* return index or -1 according to matched */
if matched ≠ 0 then return TZCNT(matched) ;
else return -1;
```

B.2 Proof of Lemma 5.2

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 $\overline{P}_{fp}()$, 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 $\overline{P}_{fp}(w, d, x) \times x$. We consider the false positive rate of every item. For the i^{th} $(1 \leqslant i \leqslant x)$ incoming item, its false positive rate equals to $P_{fp}(w, d, i-1)$, 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 $\sum_{i=0}^{x-1} P_{fp}(w, d, i) = \overline{P}_{fp}(w, d, x) \times x$.

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 $\overline{P}_{fp}(w, d, |J_{l_1}[j]|)$; 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 $\overline{P}_{fp}(w, d, |J_{l_1}[j]|) \times \overline{P}_{fp}(w, d, |J_{l_2}[j]|)$; ...; For the k^{th} part, the set size is $|I_1[j]| - |I_2[j]|$, and the probability of experiencing such false positives is $\prod_{u=1}^{k-1} \overline{P}_{fp}(w, d, |J_{l_u}[j]|)$; Considering all the above k parts, the lemma holds. □

B.3 Proof of Theorem 5.3

Proof. According to the frequencies of misreported items, we divide the P_{mr} into λ parts: $P^1_{mr}, P^2_{mr}, ..., P^\lambda_{mr}$, where P^k_{mr} $(1 \leqslant k \leqslant \lambda)$ denotes the misreport rate of the item set $\Delta_k[E] = \{e_i | f_i[E] = k\}$. Obviously, we have $\Delta_k[E] = I_k[E] - I_{k+1}[E]$ and $|\Delta_k[E]| = |I_k[E]| - |I_{k+1}[E]|$. P_{mr} is given by:

$$P_{mr} = \frac{\sum_{k=1}^{\lambda} \left(P^k_{mr} \times |\Delta_k[E]|\right)}{\sum_{k=1}^{\lambda} |\Delta_k[E]|} = \frac{\sum_{k=1}^{\lambda} \left(P^k_{mr} \times |\Delta_k[E]|\right)}{|I_1[E]| - |I_{\lambda+1}[E]|} \tag{11}$$

In order to derive P^k_{mr} $(1 \leqslant k \leqslant \lambda)$, we consider one arbitrary item $e \in \Delta_k[E]$. The item e appears in data streams k times during the time window $[1, E]$. Let $t_1, t_2, ..., t_k$ be the k appearance time points. Let $P^{k,u}_{mr}$ $(1 \leqslant u \leqslant k \leqslant \lambda)$ be the misreport rate of the item set $J_k[E]$ in the time point t_u. Obviously, if misreports do not happen in all the k time points, then the item e will not be misreported during the time window $[1, E]$. Therefore, P^k_{mr} is given by:

$$P^k_{mr} = 1 - \prod_{u=1}^{k} \left(1 - P^{k,u}_{mr}\right) \ (1 \leqslant k \leqslant \lambda) \tag{12}$$

Next, we focus on the $P^{k,u}_{mr}$ $(1 \leqslant u \leqslant k \leqslant \lambda)$. For the item e, just before the time point t_u, it has appeared in data streams $u-1$ times. If the misreport about e happens exactly in the time point t_u, this item must experience $\lambda - u + 1$ false positives from level 1 to level λ in the multi-layer Bloom filter. Therefore, $P^{k,u}_{mr}$ can be bounded by:

$$P^{k,u}_{mr}{}^L \leqslant P^{k,u}_{mr} \leqslant P^{k,u}_{mr}{}^U \ (1 \leqslant u \leqslant k \leqslant \lambda) \tag{13}$$

where

$$P^{k,u}_{mr}{}^L = \prod_{i=u}^{\lambda} P_{fp}(w, d, |J_{l_i}[t_u]|), \ P^{k,u}_{mr}{}^U = \prod_{i=1}^{\lambda-u+1} P_{fp}(w, d, |J_{l_i}[t_u]|) \tag{14}$$

Considering that $P_{fp}(w, d, x)$ is a monotonic increasing function about the variable x, we can use Lemma 6.1 to further lower bound ${P_{mr}^{k,u}}^L$ and upper bound ${P_{mr}^{k,u}}^U$ as following:

$${P_{mr}^{k,u}}^L \geqslant \prod_{i=u}^{\lambda} P_{fp}(w, d, |J_{l_i}[t_u]|^L),\ {P_{mr}^{k,u}}^U \leqslant \prod_{i=1}^{\lambda-u+1} P_{fp}(w, d, |J_{l_i}[t_u]|^U) \quad (15)$$

Then we need to know this sequence: $|J_{l_1}[t_u]|, |J_{l_2}[t_u]|, ..., |J_{l_\lambda}[t_u]|$ in each time point t_u ($1 \leqslant u \leqslant k \leqslant \lambda$). Recall that $|J_{l_i}[t_u]|$ ($1 \leqslant i \leqslant \lambda$) can be calculated recursively by the $|I_{l_i}[t_u]|$ (see Eq. 3). Therefore, we only need to get $|I_{l_i}[t_u]|$ ($1 \leqslant i \leqslant \lambda$). Under the random order model, for item e, the data streams can be uniformly partitioned into k small data streams, each of which contains total E/k items with N/k distinct items. Correspondingly, the time window $[1, E]$ is uniformly partitioned into k small time windows: $[1, \frac{E}{k}], [\frac{E}{k} + 1, \frac{2E}{k}], ..., [\frac{(k-1)E}{k} + 1, E]$. The item e appears in the middle of each time windows. In other words, $t_1 = \frac{E}{2k}, t_2 = \frac{3E}{2k}, ..., t_k = \frac{(2k-1)E}{2k}$. The item subset $I_{l_i}[t_u]$ is also uniformly distributed across and within these k time windows. Therefore, we have:

$$|I_{l_i}[t_u]| = \frac{(2u-1)}{2k} \times |I_{l_i}[E]|\ (1 \leqslant i \leqslant \lambda, 1 \leqslant u \leqslant k \leqslant \lambda) \quad (16)$$

Considering Eq. 11, 12, 13, 14, 15 and 16, the theorem holds. □

C ANALYSIS OF ONE-MEMORY-ACCESS

Still employing the random order model, we analyze the misreport rate of CF with our one-memory-access technique. One-memory-access constrains the d_1 hashed counters to a size of one word, which makes Eq. 2 (false positive rate of standard Bloom filter) not applicable to layer L_1 if we use the same derivation method (see § 5). Therefore, we need to recompute the false positive rate of the so-called one-memory-access Bloom filter. The one-memory-access Bloom filter (*omaBF*) consists of a w-bit array. The picking scheme of d hashed bits is: 1) choose a word with size of W_{bf}, 2) choose d hashed bits uniformly from this word. In particular, we have $\mathrm{W}_{bf} = \mathrm{W}/\delta_1$. Let l be the number of words in this w-bit array. We have $l = w/\mathrm{W}_{bf}$. Assume the number of distinct items inserted into this Bloom filter is n. Given w, d, n and W_{bf}, according to literature [52], the false positive rate of one-memory-access Bloom filter is:

$$P_{fp}^{oma}(w, d, n, \mathrm{W}_{bf}) = \sum_{x=0}^{n} \left[\binom{n}{x} \left(\frac{1}{l}\right)^x \left(1 - \frac{1}{l}\right)^{n-x} \left(1 - \left(1 - \frac{1}{\mathrm{W}_{bf}}\right)^{xd}\right)^d \right] \quad (17)$$

We redefine $P_{pf}^{U}(x)$ and $\overline{P}_{pf}^{U}(x)$ as follows:

Definition C.1. *For each time point $j \in [E]$,*

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

$$\overline{P}_{pf}^{U}(|J_x[j]|) = \begin{cases} \frac{1}{|J_x[j]|} \sum_{i=0}^{|J_x[j]|-1} P_{fp}^{oma}(w_1, d_1, i, \frac{\mathrm{W}}{\delta_1})\ (1 \leqslant x \leqslant \mathcal{T}_1) \\ \frac{1}{|J_x[j]|} \sum_{i=0}^{|J_x[j]|-1} P_{fp}(w_2, d_2, i)\ (\mathcal{T}_1 + 1 \leqslant x \leqslant \mathcal{T}) \end{cases} \quad (18)$$

The rest of the derivation is the same as Lemma 5.4 and Theorem 5.5 (getting $|J_k[j]|^L$ and $|J_k[j]|^U$, and then obtaining the bound of the misreport rate of CF with one-memory-access). We also omit it.

D ANALYSIS OF CM-CU WITH CF

Now, we give the theoretical analysis of the CM-CU sketch with CF (with two optimizations) for estimating item frequency. We use the same notations and assumptions employed in §5. 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.

D.1 Proof of No Under-estimation Error

When processing a data stream, our CF records the frequency or part of the frequency of each item. At the end of each measurement period, the frequency of each item is divided into up to three parts that locate in the two layers of CF and the CM-CU sketch, respectively. These three parts are all recorded and queried by the same approach as the CM-CU sketch. Since the CM-CU sketch has no under-estimation error [18], the estimated frequency assembled by these three parts is also free of under-estimation. Therefore, the CM-CU sketch with CF has no under-estimation error. Note that this property does not rely on any assumptions about the distribution of item frequency or appearance order.

D.2 Error Bound

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 be applicable 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 [18] shows that the Bloom filter with partitioning (*e.g.*, d_ϕ segments in Φ) has the same asymptotic false positive rate with standard one without partitioning. Here, we consider them as the same. Therefore, we define a unified function about its false positive rates of different layers as follows:

Definition D.1. *For each time point $j \in [E]$,*

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

$$\overline{P}_{pf}^{\phi\ U}(|J_x[j]|) = \begin{cases} \frac{1}{|J_x[j]|} \sum_{i=0}^{|J_x[j]|-1} P_{fp}^{oma}(w_1, d_1, i, \frac{\mathrm{W}}{\delta_1})\ (1 \leqslant x \leqslant \mathcal{T}_1) \\ \frac{1}{|J_x[j]|} \sum_{i=0}^{|J_x[j]|-1} P_{fp}(w_2, d_2, i)\ (\mathcal{T}_1 + 1 \leqslant x \leqslant \mathcal{T}) \\ \frac{1}{|J_x[j]|} \sum_{i=0}^{|J_x[j]|-1} P_{fp}(w_\phi, d_\phi, i)\ (x \geqslant \mathcal{T} + 1) \end{cases} \quad (19)$$

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

Theorem D.1. *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 the time window $[1, E]$, respectively. Let V be $\sum_{i=1}^{N} f_{e_{\beta_i}}$. Let $l_1, ..., l_u, ...$ be the permutation of $1, 2, ...$ that makes the sequence ${P_{fp}^{\phi}}^U(|J_{l_u}[E]|^U)$ in descending order. We have the following accuracy guarantee about the CM-CU sketch with CF (with one-memory-access):*

$$Pr[\hat{f}_{e_{\beta_i}} - f_{e_{\beta_i}} \leqslant \lceil \varepsilon V \rceil] \geqslant 1 - \prod_{u=1}^{\lceil \varepsilon V \rceil} {P_{fp}^{\phi}}^U(|J_{l_u}[E]|^U) \quad (20)$$

Proof. We employ the equivalence between CF and multi-layer Bloom filter to complete the proof. For convenience, let Δ be $\lceil \varepsilon V \rceil$. For an arbitrary item e_{β_i} ($i \in [N]$), we consider the upper bound of $Pr[\hat{f}_{e_{\beta_i}} - f_{e_{\beta_i}} > \Delta]$. $\hat{f}_{e_{\beta_i}} - f_{e_{\beta_i}} > \Delta$ means the item e_i experiences false positives at least $\Delta + 1$ times (at different layers in the multi-layer Bloom filter). This probability is upper-bounded by the multiple multiplications of the $\Delta + 1$ maximum false positive rates among all the layers in the multi-layer Bloom filter. Besides, we also need to find an appearance order for item e_{β_i} with the frequency of $f_{e_{\beta_i}}$ to achieve the worst case (maximum) false positive rate. Intuitively, the worst case happens, when the $f_{e_{\beta_i}}$ instances of item e_{β_i} all appear at the end of the measurement period (or near the time point E), since CF is most heavily loaded at this time point. Therefore, we can use the false positive rate instead of the past false positive rate to depict the upper bound of $Pr[\hat{f}_{e_{\beta_i}} - f_{e_{\beta_i}} > \Delta]$. Formally, we have:

$$\begin{aligned} Pr[\hat{f}_{e_{\beta_i}} - f_{e_{\beta_i}} > \Delta] &\leqslant \prod_{u=1}^{\Delta+1} {P_{fp}^{\phi}}^{U}(|J_{l_u}[E]|) \\ &< \prod_{u=1}^{\Delta} {P_{fp}^{\phi}}^{U}(|J_{l_u}[E]|) \leqslant \prod_{u=1}^{\Delta} {P_{fp}^{\phi}}^{U}(|J_{l_u}[E]|^U) \end{aligned} \tag{21}$$

Converting the less-than and greater-than signs, the theorem holds. □

Consider the aggregate-and-report strategy. Obviously, the aggregate-and-report strategy only changes the appearance order of some items. Since in Theorem D.1 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.

E CF WITH PARALLEL STRATEGIES

E.1 CF on Multi-core Platform (Pipeline)

Our CF naturally supports pipeline parallelism on multi-core platform. We divide the CF meta-framework into two parts: the CF (including aggregate-and-report and one-memory-access) on core C_0, and the specific stream processing algorithm (the CM-CU sketch, Space-Saving, FlowRadar, *etc.*) on core 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 need to be reported to the specific stream processing algorithm, core C_0 will forward them to core C_1. Then, the remaining job is executed by C_1, and 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 C_1, and the frequency of such report is low (*e.g.*, less than 1 per 20 incoming items in real data streams when $\delta_1 = 4$).

E.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 C_0 (*master core*) performs CF (including aggregate-and-report and one-memory-access), and the other cores $C_1, C_2, \ldots$ (*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.

E.3 Evaluation on CF with Multi-core CPU

The results are shown in Figure 20, 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 1).

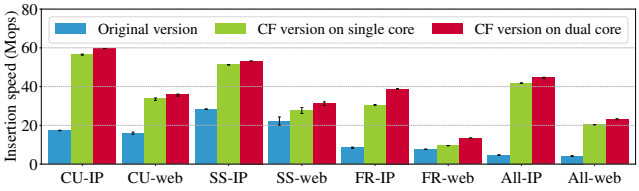
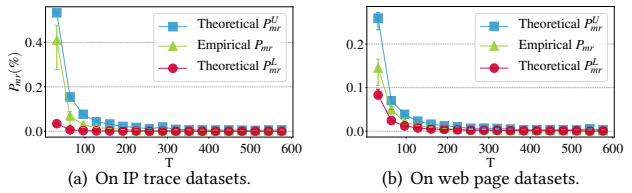


Figure 20: Effects of multi-core on real-world datasets.

F THEORETICAL VS. EMPIRICAL P_{mr} OF CF

We compare the theoretical and empirical results about the P_{mr} of CF (with one-memory-access strategy), see Figure 21(a)-(b). We employ the same parameter setting as in §6.4.1. Here, we use P_{mr}^U and P_{mr}^L to denote the upper and lower bound of P_{mr}, respectively. Our experimental results show that our theoretical results are nicely consistent with the empirical results especially when the threshold $\mathcal{T}$ becomes larger.



(a) On IP trace datasets. (b) On web page datasets.

Figure 21: Theoretical vs. Empirical results of CF on real-world datasets.