LadderFilter: Filtering Infrequent Items with Small Memory and Time Overhead

Yuanpeng Li
Peking University

Feiyu Wang
Peking University

Xiang Yu
Peking University

Yilong Yang
Xidian University

Kaicheng Yang
Peking University

Tong Yang
Peking University

Zhuo Ma
Xidian University

Bin Cui
Peking University

Steve Uhlig
Queen Mary
University of London

ABSTRACT

Data stream processing is critical in streaming databases. Existing works pay a lot of attention to frequent items. To improve the accuracy for frequent items, existing solutions focus on accurately filtering infrequent items. While these solutions are effective, they keep track of all infrequent items and require multiple hash computations and memory accesses. This increases memory and time overhead. To reduce this overhead, we propose LadderFilter, which can *discard* infrequent items efficiently in terms of both memory and time. To achieve memory efficiency, LadderFilter discards (approximately) infrequent items using multiple LRU queues. To achieve time efficiency, we leverage SIMD instructions to implement LRU policy without timestamps. We apply LadderFilter to four types of sketches. Our experimental results show that LadderFilter improves the accuracy by up to 60.6×, and the throughput by up to 1.37×, and can maintain high accuracy with small memory usage. All related code is provided open-source at Github.

1 INTRODUCTION

Data stream processing is very important in a variety of areas in data science, such as intrusion detection [1, 2], recommendation systems [3, 4], *etc.* [5–7]. Data streams are usually highly skewed [8–10], *i.e.*, a few items are very popular (called frequent items), while the vast majority of items are unpopular (called infrequent items). The research community so far has paid more attention to the frequent items in the data stream. Many important measurement tasks focus on frequent items, including finding top-k items [11, 12], finding heavy changes [13, 14], finding super-spreaders [15, 16], *etc.* [17–19]. In these tasks, the numerous infrequent items consume too much memory, which degrades the accuracy. Sketch, as a kind of compact data structure with small error, is promising in data stream processing [8–10, 20, 21]. Its speed is constant: each insertion needs several hash computations and memory accesses. To satisfy the tasks favoring frequent items, a widely-acknowledged approach is to filter infrequent items [10, 22].

Ideally, one would want to filter all infrequent items without error. However, initially, every item is infrequent, and could become frequent after a long enough period of time. The large volume and high item arrival rate of data streams make it impractical to keep the frequency of all items without error, with limited memory and time. Therefore, our goal is to approximately filter infrequent items while satisfying the following two requirements.

- **Memory efficiency:** A method consists in keeping approximate frequency of all items. However, this method is still memory inefficient because of the numerous infrequent items. In this paper, we manage to discard infrequent items with small error.
- **Time Efficiency:** Achieving time efficiency is possible through two types of methods. 1) We can reduce the number of hash computations and memory accesses. These are the two bottlenecks for the processing speed [23, 24]. 2) To further accelerate the processing, we can leverage the full use of the new features of CPU instructions or rely on hardware acceleration.

The most directly related works in filtering infrequent items are ColdFilter [10] and LogLogFilter [22]. ColdFilter [10] uses a 2-layer CU sketch [1] [8] to record the frequency of each item, and sets a threshold to separate frequent items from infrequent items. When inserting an item, ColdFilter first inserts it into the CU sketch and queries its frequency. The items whose queried frequency exceeds a pre-defined threshold will be reported as frequent items. To enlarge the filter range of ColdFilter, LogLogFilter [22] replaces the CU sketch with a LogLog structure [25].

ColdFilter keeps the frequency of all infrequent items, which comes with unnecessary memory overhead. ColdFilter also requires multiple hash computations and memory accesses (*e.g.*, 6 or more), resulting in considerable time overhead. LogLogFilter inherits the previously mentioned limitations of ColdFilter. To the best of our knowledge, no existing solution simultaneously meets the above two requirements.

To accurately filter infrequent items with small memory and time overhead, we design a new probabilistic algorithm, LadderFilter. The key technique of LadderFilter is to discard "unpromising" items in time, based on our observation about which items are not likely to become popular. We also propose a SIMD-based method to optimize LadderFilter.

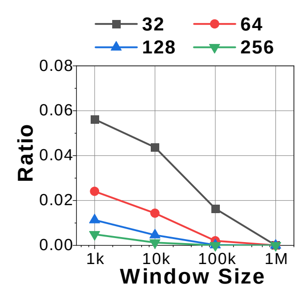


Figure 1: Ratio of unpromising items becoming frequent. Different lines represent different thresholds of frequent items.

To better illustrate our observation, we first define active, inactive, promising, and unpromising items. If an item does not appear in the recent time window, we call it an *inactive item*; otherwise, we call it an *active item*. When an active item becomes inactive, 1) if its frequency is small (*e.g.*, < 5), we call it an *unpromising item*; 2) if its frequency is moderate (*e.g.*, $5 \sim 30$), we call it a *promising item*. Note that we judge whether an item is promising/unpromising according to its

[1] A CU sketch is a classic counter-based sketch (see more details in § 3.1).

frequency only when it becomes inactive. [2] We study a number of real datasets, and observe that an item that is unpromising for a long time rarely becomes frequent afterwards. Figure 1 shows the results on IP trace dataset (see § 4.1) [26]. Less than 6% of the unpromising items become frequent items. In detail, we consider an inactive item with frequency less than 5 as an unpromising item. When the threshold of frequent item is 256, there are about 10.1k frequent items. When the sliding window size is 10k items, there are about 200k items that are unpromising till the end of the stream. Among them, only 262 (0.1%) unpromising items grow into frequent items. When the sliding window size exceeds 100k, the number decreases to 0.

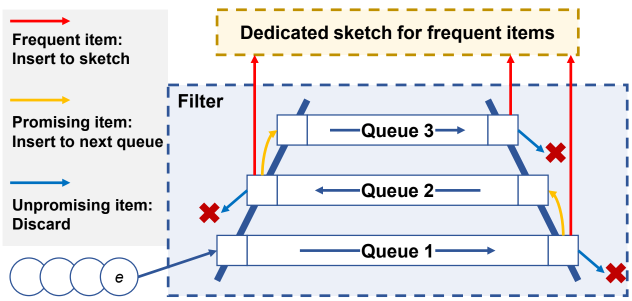


Figure 2: LadderFilter workflow.

Based on the above observation, we design LadderFilter, whose key technique is to *discard the unpromising items in LadderFilter as early as possible*. The data structure of LadderFilter is similar to a ladder consisting of multiple LRU queues [3] (see Figure 2). For queue i, it is associated with a low threshold T_i^{low} and a common high threshold T^{high}. When an item is dequeued, if its frequency exceeds T^{high}, we consider it as a frequent item; if its frequency is lower than T_i^{low}, we consider it as an unpromising item; otherwise, we consider it as a promising item. Frequent items are sent to a dedicated sketch designed to record frequent items; unpromising items are simply discarded; promising items are inserted to the next queue. In this way, we give the promising items another chance to become frequent items. If the item grows fast in the next queue and exceeds T^{high}, it will become a frequent item; if it grows too slowly (less than T_{i+1}^{low}), it will be considered as an unpromising item and discarded; otherwise, it is still a promising item and will enter queue $i+2$.

To achieve time efficiency, we propose an optimized version of LadderFilter, using two methods. The first method is to approximate LRU queues with bucket arrays. Each LRU queue is replaced by an LRU bucket array associated with a hash function, which maps each item to one bucket (see § 2.2). The second method is *SIMD Acceleration*. We leverage SIMD instructions in two ways. First, we accelerate the ID match, similarly to previous work [9, 27]. Second, we use SIMD instructions to sort the items. Through only two SIMD instructions, we keep items in time order, and we implement the LRU policy without recording any timestamp. To the best of our knowledge, we are the **first work** to sort items in the context of sketching algorithms.

[2] *Example:* Suppose that item e arrives 30 times continuously, and then stops for a relatively long time, which means it becomes inactive. Because 30 is moderate, we recognize e as a promising item.

[3] The reason for using the LRU policy rather than LFU is that LFU is time-agnostic (see more details in § 2.1).

We apply LadderFilter to four kinds of widely used sketches: the CU sketch [8], SpaceSaving [11], FlowRadar [13], and WavingSketch [16]. Our experimental results show that LadderFilter improves the accuracy by up to 60.6×, and the throughput by up to 1.37×. Also, LadderFilter can maintain high accuracy even with extremely limited memory, while the accuracy of prior works degrades significantly as memory shrinks. All related code is open-sourced at Github[4].

Key Contributions:

- We propose a basic version of LadderFilter to discard infrequent items with small memory overhead, based on the observation that unpromising items rarely grow into frequent items.
- We propose an optimized version to accelerate LadderFilter. We leverage SIMD instructions to implement the LRU policy.
- We implement LadderFilter and apply it to four kinds of frequently used sketches on four typical data stream tasks. The experimental results show that LadderFilter improves the accuracy and throughput by up to 60.6× and 1.37×, respectively, and can maintain high accuracy even with limited memory.

2 LADDER FILTER

In this section, we present the data structure and operation of LadderFilter. We first present the basic version of LadderFilter which achieves memory efficiency. We then present an optimized version of LadderFilter to enhance its time efficiency. After that, we present a SIMD-based method to accelerate LadderFilter.

2.1 Basic Version

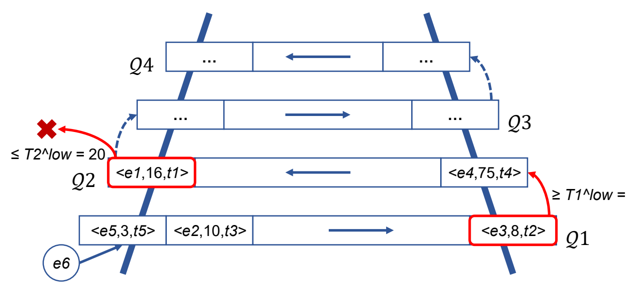


Figure 3: An example of the basic version of LadderFilter.

Data structure: As shown in Figure 3, LadderFilter consists of λ LRU queues. The i^{th} queue Q_i consists of l_i cells. Each cell records a distinct item with three fields: ID, $freq$, and $timestamp$, representing the ID, frequency, and the last arrival timestamp of the item, respectively. Each queue is associated with a low threshold T_i^{low} and a common high threshold T^{high}. All high thresholds are equal, and the low thresholds are increasing, *i.e.*, $T_1^{low} < T_2^{low} < \cdots < T_{\lambda-1}^{low} < T_\lambda^{low} = T^{high}$.

Insertion: There are two cases when inserting an item e.

Case 1: If e has already been recorded in one of the queues, LadderFilter increments its frequency by 1, and updates its last arrival timestamp to the current timestamp. If its frequency exceeds the high threshold T^{high}, LadderFilter reports it as a frequent item.

Case 2: If e is not recorded in LadderFilter, we enqueue it to the first LRU queue Q_1. If Q_1 is not full, LadderFilter enqueues e to Q_1 with frequency 1 and the current timestamp. Otherwise, LadderFilter

[4] https://github.com/LadderFilterCode/LadderFilter

dequeues the least recent item e_{LRU} from Q_1, and enqueues e to Q_1. If the frequency of e_{LRU} exceeds the low threshold T_i^{low}, we consider e_{LRU} as a promising item, and enqueue it to the next queue Q_2. The enqueueing process is the same as for Q_1, except that the last queue Q_λ will discard the least recent item instead of trying to enqueue it to another queue.

Algorithm 1: Insertion of LadderFilter.

Input: Item e
Function Enqueue(Q_i, e, $freq$, $timestamp$):
  if Q_i *is full* **then**
    $e_{LRU} \leftarrow$ the least recent item in Q_i
    if $i < \lambda$ *and* $Q_i[e_{LRU}].freq \geqslant T_i^{low}$ **then**
      Enqueue(Q_{i+1}, e_{LRU}, $Q_i[e_{LRU}].freq$, $timestamp$)
    dequeue item e_{LRU} from Q_i
  enqueue item $< e, freq, timestamp >$ to Q_i

for $i \in [1, k]$ **do**
  if $e \in Q_i$ **then**
    $Q_i[e].freq \leftarrow Q_i[e].freq + 1$
    $Q_i[e].timestamp \leftarrow current\_timestamp$
    if $Q_i[e].freq \geqslant T^{high}$ **then**
      report e as a frequent item
    return
Enqueue(Q_1, e, 1, $current\_timestamp$)

Example 1: Figure 3 shows an example of the basic version of LadderFilter. The LadderFilter consists of 4 LRU queues Q_1, Q_2, Q_3, and Q_4. Q_1 is associated with a low threshold $T_1^{low} = 8$, Q_2 is associated with a low threshold $T_2^{low} = 20$, and Q_3 is associated with a low threshold T_3^{low}. All queues are associated with a high threshold T^{high}. Suppose we insert e_6 at time t_7. We find that e_6 is not recorded in LadderFilter, and we enqueue it to the first queue Q_1. Q_1 is full, so we dequeue the least recent item e_3 from Q_1, and record $< e_6, 1, t_7 >$ in the cell. Then we compare the frequency of e_3 and Q_1's low threshold T_1^{low}. The frequency 8 exceeds the threshold 8. Therefore, we enqueue e_3 to Q_2 with frequency 8 and timestamp t_7. Q_2 is also full, so we dequeue the least recent item e_1, and record $< e_3, 8, t_7 >$ in the cell. e_1's frequency 16 does not exceed T_2^{low}, so we discard e_1.

Discussions on replacement policies: We choose to use the LRU policy. By using the LRU policy, we can distinguish between active and inactive items. By recording frequency, we can further distinguish between promising and unpromising items, and discard the unpromising items. We do not use the LFU policy, because LFU is time-agnostic, and thus we cannot distinguish promising items and unpromising items without time information. Another possible policy is LRFU. LRFU takes into account both arrival time and frequency. However, LRFU requires more parameters and different optimization strategies. We leave LRFU for future work.

2.2 Optimized Version

Rationale: There are mainly two methods to implement LRU queues.

- **Memory-oriented method:** Using no additional data structure. When looking for an item, we scan the whole queue. However, the time complexity is O(queue len).
- **Time-oriented method:** Using a hash table to locate the incoming items and a bidirectional linked list to maintain the arrival order of items. However, this consumes a lot of extra memory.

In summary, the above two methods are either time consuming or memory consuming. In contrast, our design goal is to implement LRU queues in a method that optimizes both memory and time. Our methodology is to achieve this design goal by approximately implementing LRU. Fortunately, accurate LRU and approximate LRU has little performance difference for LadderFilter. Therefore, we choose to implement LRU queues in an approximate manner and propose an optimized version of LadderFilter.

Data structure: The LRU queue Q_i is replaced by an LRU bucket array with w_i buckets. Let $Q_i[j]$ denote the j^{th} bucket. Each bucket contains c cells ($w_i \times c = l_i$), where c is usually small (*e.g.*, 8). Q_i is also associated with a hash function $h_i(.) (0 \leqslant h_i(.) < w_i)$, which maps each item to one of the buckets.

Operations: Each bucket obeys LRU policy independently. When enqueueing an item e to Q_i, LadderFilter first computes hash function $h_i(e)$ to locate one LRU bucket $Q_i[h_i(e)]$. Then LadderFilter enqueues e to the bucket in a process similar to the basic version. If the bucket is full, LadderFilter dequeues the least recent item from the bucket. The dequeueing operation works as follows: LadderFilter scans the bucket, finds the least recent item, and dequeues it. To sum up, both the enqueueing and dequeueing operations are applied to only one hashed LRU bucket instead of the whole queue in the basic version.

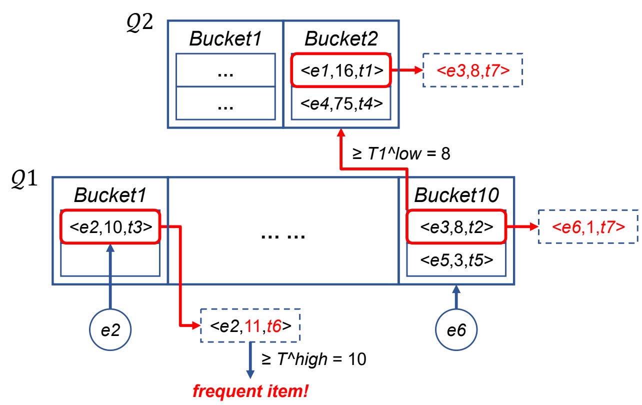


Figure 4: An example of the optimized version.

Example 2: Figure 4 shows an example of the optimized version of LadderFilter. The LadderFilter consists of 2 LRU queues Q_1 and Q_2. Q_1 consists of 10 LRU buckets, and Q_2 consists of 2 LRU buckets. When inserting e_2 at time t_6, we first calculate the two hash functions $h_1(e_2) = 1$ and $h_2(e_2) = 1$ to locate the corresponding bucket in each queue. We find that e_2 has already been recorded in Q_1. Therefore, we increment its frequency by 1 to 11, and update its timestamp to t_6. Then we compare the frequency of e_2 and the high threshold T^{high}. The frequency exceeds the threshold, and LadderFilter reports e_2 as a frequent item.

Example 3: When inserting e_6 at time t_7, we first calculate the two hash functions $h_1(e_6) = 10$ and $h_2(e_2) = 2$ to locate the corresponding bucket in each queue. We find that e_6 is not recorded in any corresponding bucket. Therefore, we enqueue e_6 to bucket 10 in Q_1. We find that the bucket is full. Therefore, we dequeue the least recent item e_3, and record $< e_6, 1, t_7 >$ in the cell. Then we compare the frequency of e_3 and Q_1's low threshold T_1^{low}. The frequency exceeds the threshold, and we enqueue it to Q_2 with frequency 8 and timestamp t_7. We find that bucket 2 in Q_2 is also full. Therefore, we dequeue the least recent item e_1, and record $< e_3, 8, t_7 >$ in the cell. Note that Q_2 is the last queue in LadderFilter, therefore, e_1 is discarded.

Next, we show that the optimized version is similar to the basic version in terms of dequeueing items.

Theorem 1. *In both versions, the expectation of the dequeueing interval*[5] *of an item e is the same.*

Proof. Let E^{basic} and E^{opt} be the expectation of the dequeueing interval. Let w be the number of buckets, and c be the number of cells in each bucket in the optimized version. The number of cells in the LRU queue in the basic version is $w \cdot c$. Suppose distinct items arriving at a constant rate v. In the basic version, the expectation of the dequeueing interval

$$E^{basic} = \frac{w \cdot c}{v}.$$

In the optimized version, according to the randomness of the hash computation, an item is inserted to every bucket with equal probability, *i.e.*, $\frac{1}{w}$. Therefore, the expectation of the time that a distinct item inserted to a specific bucket b

$$E^{opt}\left\{1\ distinct\ item\ inserted\right\} = \frac{w}{v}.$$

The expectation of the dequeueing interval

$$E^{opt} = c \cdot E^{opt}\left\{1\ distinct\ item\ inserted\right\} = \frac{w \cdot c}{v} = E^{basic}.$$

□

Analyses on worst cases: There are mainly two worst cases in the optimized version.

- **Hash collision:** All items are hashed to the same bucket. This will lead to low accuracy as many frequent items are discarded since they are classified as unpromising. If this occurs, error is large, and we can address this by replacing the hash function.
- **Hash starving:** Some buckets have no item hashed into. This means the bucket array has a low loading rate, and it is memory wasting.

Next, we derive the probability that the worst cases occur. Suppose there are w buckets. Considering the randomness/uniformity of hashing, for an arbitrary bucket $Q[i]$, the probability that an arbitrary item e is located to $Q[i]$ is $\frac{1}{w}$. Suppose the number of distinct items is N. Let N_i be the number of distinct items located to $Q[i]$. The expectation of N_i is $E(N_i) = \frac{N}{w}$. The variance $D(N_i) = \frac{N(w-1)}{w^2}$. Therefore, for each arbitrary ϵ, by Chebyshev inequality,

$$P\{|N_i - E(N_i)| \geqslant \epsilon\} \leqslant \frac{N(w-1)/w^2}{\epsilon^2}.$$

Hash collision means $N_i \gg E(N_i)$. Suppose a is a constant that satisfies $1 \leqslant a \ll w$. Therefore,

$$\begin{aligned} P\{N_i \geqslant \frac{N}{a}\} &\leqslant P\{|N_i - E(N_i)| \geqslant \frac{N}{a} - \frac{N}{w}\} \\ &\leqslant \frac{a^2(w-1)}{N(w-a)^2} \approx \frac{a^2}{Nw}. \end{aligned}$$

Hash starving means $0 \approx N_i \ll E(N_i)$. Suppose b is a constant that satisfies $1 \leqslant b \ll E(N_i) = \frac{N}{w}$. Therefore,

$$\begin{aligned} P\{0 \leqslant N_i \leqslant b\} &\leqslant P\{|N_i - E(N_i)| \geqslant \frac{N}{w} - b\} \\ &\leqslant \frac{N(w-1)}{(N-bw)^2} \approx \frac{w}{N}. \end{aligned}$$

Note that, N and w are large in data stream and deployment, and w is usually several orders of magnitude smaller than N (see § 4). Therefore, the probability of the two worst cases occurring is very low.

Optimization – using fingerprints. As many existing works [28, 29], LadderFilter also supports using fingerprints to replace the IDs when the length of item ID is long (*e.g.*, 104 bits in TCP packet streams). Although using fingerprints may result in hash collision of two distinct items, it can significantly reduce the memory usage. In other words, it can achieve higher accuracy with the same memory. Next, we show the probability of hash collision, and the expectation of overestimation.

Lemma 2. *In the optimized version, the probability of an item e suffering from hash collisions*

$$Pr\left\{hash\ collision\right\} = 1 - \left(1 - 2^{-l}\right)^n,$$

where l is the length of the fingerprint, and n is the number of distinct items inserted to the bucket when e is in the bucket.

Lemma 3. *The expectation of the overestimation of an item e caused by hash collisions*

$$E\left\{overestimation\right\} = n \cdot 2^{-l}.$$

Table 1: The expectation of overestimation caused by hash collisions.

Probability	$n = 10$	$n = 100$	$n = 1000$
$l = 8$	3.906×10^{-2}	3.906×10^{-1}	3.906×10^{0}
$l = 16$	1.526×10^{-4}	1.526×10^{-3}	1.526×10^{-2}
$l = 32$	2.328×10^{-9}	2.328×10^{-8}	2.328×10^{-7}

The expectation of the overestimation caused by hash collisions is shown in Table 1. For an infrequent item, $n \leqslant \mathcal{T}^{high} \cdot c$. Suppose $\mathcal{T}^{high} = 100$ and $c = 8$. $n \leqslant 800$. $E\left\{overestimation\right\} \leqslant 1.526 \times 10^{-2}$. We recommend using 16-bit fingerprints.

2.3 SIMD Acceleration

The optimized version meets our requirement in terms of memory and time efficiency. However, it still requires storing and comparing timestamps, which still incurs a large memory and time overhead. Motivated by this, we propose to accelerate the insertion of LRU buckets with SIMD instructions. For each bucket, we maintain the ID and frequency of each item, while removing the last arrival timestamp. To locate the LRU item, we keep the items in time

[5]The interval between the last arrival time of an item and the time it is dequeued from the bucket/queue.

order. Unlike the basic version, when inserting an item e to bucket $Q_i[h_i(e)]$, after inserting/updating a cell, we further sort the items in the bucket according to time. Suppose e is the j^{th} item in the bucket. We move the $(j+1)^{th}$ items to the j^{th} cell, the $(j+2)^{th}$ items to the $(j+1)^{th}$ cell, ..., the c^{th} items to the $(c-1)^{th}$ cell, and e /the j^{th} item to the c^{th} cell. The 1^{st}, ..., $(j-1)^{th}$ items remain in their original cells.

Algorithm 2: SIMD acceleration.

```
Input: The sequence of the arriving item i
1 uint16_t id[8], freq[8];
2 __m128i index[4] =
    _mm_setr_epi8(8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15);
3 __m128i *p_id = (__m128i*)&id;
4 p_id[0] = _mm_shuffle_epi8(p_id[0], index[i]);
```

This version seems to require a lot of operations and thus be slow. However, it is ideal for SIMD acceleration. For a better demonstration, we show the detailed implementation under the following parameter settings: each bucket consists of 8 cells, and each cell consists of a 16-bit ID/fingerprint and a 16-bit frequency. Algorithm 2 shows the C++ code for the sorting of IDs. The operation on frequencies is the same as IDs. For lookup and update operations, please refer to [9, 10, 30]. The idea is to use function `_mm_shuffle_epi8` to rearrange each byte in IDs into proper order. To ensure memory continuity, we record IDs and frequencies in two arrays separately (see Line 1). We pre-set the order of each byte in rearrangement operations. Line 2 gives a example of the pre-set order when the arriving item is the 4^{th} item in the bucket. Line 3 transposes the ID array into a `_m128i` pointer. The compiler will load all IDs into a 128-bit SIMD register. Line 4 uses SIMD instruction `_mm_shuffle_epi8` to rearrange bytes in the register in proper sequence. The IDs will then be stored to the proper cells (1 CPU cycle [31]). In summary, we sort the items within 2 SIMD instructions (1 for IDs, and 1 for frequencies), *i.e.*, 2 CPU cycles. We can also implement the operation over larger scale with `_mm256_shuffle_epi8` and `_mm512_shuffle_epi8`. The sort can be done within 2 SIMD instructions but requires more swap operations on integers.

Time complexity: Using multiple LRU buckets can accelerate the operations without additional data structures. Each bucket contains much fewer items than the whole queue, hence we scan much fewer items during each operation. The optimized version reduces the time complexity from O(queue len) to O(bucket size). Most importantly, we use SIMD instructions to optimize enqueue/dequeue. SIMD instructions can quickly rearrange cells in time order with only 2 instructions, *i.e.*, 2 CPU cycles.

Discussions on filter algorithm: The reviewer proposes to enqueue the promising item again with an updated priority (*i.e.*, mark it as recently used) to the same queue. The idea is novel and interesting but is incompatible with our SIMD acceleration. We will study it in the future work.

3 LADDERFILTER DEPLOYMENT

In this section, we describe how to deploy LadderFilter on four important tasks in data stream processing: estimating item frequency, finding top-k items, finding heavy changes, and finding super-spreaders. For each task, we first present the problem definition. Then we introduce popular prior solutions for the task. Finally, we describe how to apply LadderFilter to these solutions.

3.1 Estimating Item Frequency

Problem definition: Given a data stream, reporting the frequency of every item ID.

Prior solutions: The CU sketch [8] is an extension of the well-known CM sketch [20] for estimating item frequency. A CU sketch consists of d counter arrays, and each array is associated with a hash function. When inserting item e, the CU sketch first computes the d hash functions to locate the d mapped counters in each counter array. Then, the CU sketch increments the minimum mapped counters by one, which is called the conservative update strategy. When querying the frequency of item e, the CU sketch computes the d hash functions and locates the d mapped counters. Then, the CU sketch reports the minimum value among the mapped counters as the frequency of item e.

Applying LadderFilter: We build a LadderFilter to cooperate with the CM sketch. LadderFilter will be used to prevent infrequent items from being inserted into the CM sketch, since we consider the accuracy of frequent items to be more important.

Insertion: When inserting item e, we first insert e into LadderFilter as mentioned in § 2.2. If LadderFilter reports e as a frequent item, we further insert e into the CM sketch. The insertion frequency depends on whether e is reported for the first time. If e is reported as a frequent item for the first time, we insert it with frequency (T^{high}) to the CM sketch; otherwise, we insert e with frequency (one) to the CM sketch.

Query: There are two steps for a query. 1) We first query CM for the frequency of item e. If its frequency is not 0, it must exceed the high threshold T^{high}. Therefore, we consider it as a frequent item and report the frequency from CM. 2) Otherwise, e is an infrequent item. We then check whether e is in LadderFilter. If it is recorded in LadderFilter, we report the frequency from LadderFilter; otherwise, we report its frequency as 0.

3.2 Finding Top-k Items

Problem definition: Given a data stream and k, reporting the k items with the highest frequency.

Prior work: SpaceSaving [11] is the most well-known solution for finding top-k items. SpaceSaving uses a data structure called Stream-Summary to maintain frequent items. Stream-Summary achieves updating and querying in linear time, while maintaining the order of the items. When inserting item e, if e is already recorded in Stream-Summary, or it is not full, SpaceSaving inserts e into Stream-Summary. Otherwise, SpaceSaving replaces the item with the minimum frequency in Stream-Summary with item e, and increments its frequency by 1. When querying top-k items, SpaceSaving reports the k items with the highest frequency in Stream-Summary.

Applying LadderFilter: We build a LadderFilter to cooperate with SpaceSaving. LadderFilter will be used to prevent infrequent items from being inserted into SpaceSaving. We do this because all top-k items must be frequent items, therefore, inserting infrequent items to SpaceSaving will degrade accuracy.

Insertion: When inserting item e, we first check whether e is already recorded in SpaceSaving. If so, we insert it into SpaceSaving.

Otherwise, we insert item e into LadderFilter. If LadderFilter reports item e as a frequent item, we further insert it into SpaceSaving. Note that similarly to estimating item frequency, we update SpaceSaving with frequency depending on whether item e is reported as a frequent item for the first time.

Query: When querying top-k frequent items, we report the k items reported by SpaceSaving.

3.3 Finding Heavy Changes

Problem definition: Given a data stream, reporting all items that experience a frequency change exceeding a threshold $\mathcal{T}_\Delta$ between two consecutive time windows.

Prior work: FlowRadar [13] is a promising solution for finding heavy changes. To find heavy changes, one FlowRadar is built for each time window. The FlowRadar consists of a Bloom filter [32] and a counting table. The bloom filter is used to identify whether an inserting item is a new distinct item. The counting table is an extended Invertible Bloom Lookup Table (IBLT) [33] used to encode item IDs and their frequency. When inserting item e, FlowRadar first checks the bloom filter to identify whether item e is a new item. If so, FlowRadar increments its frequency; otherwise, FlowRadar further encodes the ID. When querying heavy changes, FlowRadar first decodes its counting table to get an $< item, frequency >$ set. Then, FlowRadar compares the two sets in the two consecutive time windows, and reports the heavy changes.

Applying LadderFilter: We build a LadderFilter to cooperate with FlowRadar. LadderFilter will be used to prevent infrequent items from being inserted into the Flowradar. The reason behind it is as follows: if an item is a heavy change, it must be a frequent item in at least one of the time window. Because LadderFilter can automatically discard unpromising items, we build a single LadderFilter and use it to filter infrequent items in all time windows.

Insertion: Similar to finding top-k items, when inserting an item, we first check the Bloom filter to find whether the item is already recorded in FlowRadar. If so, we insert the item to it. Otherwise, we insert the item into LadderFilter. If LadderFilter reports the item as a frequent item, we further insert it into FlowRadar.

Query: When querying the heavy changes, we first decode the two corresponding FlowRadar for two consecutive time windows, and get two $< item, frequency >$ sets S_1 and S_2. Then we insert the infrequent items recorded by LadderFilter into S_2. We consider the items in the two sets as potential heavy changes. After getting the two sets, we calculate the frequency difference between the two sets. Note that if an item is not recorded in one set, we consider its frequency in the corresponding time window as zero. We report all items whose frequency difference exceeds $\mathcal{T}_\Delta$.

3.4 Finding Super-Spreaders

Problem definition: Given a data stream with $< src, dst >$ (source, destination) pair, report all sources whose number of destinations connected exceeds a threshold $\mathcal{T}$.

Prior work: WavingSketch [16] is a recent solution for finding top-k items, and can be extended to find super-spreaders. WavingSketch is made of multiple buckets. Each bucket consists of a Waving counter and a Heavy part. The Heavy part contains several cells, recording ID, frequency, and error flags. During an insertion, if the item is recorded in the Heavy Part with no error, or the Heavy Part is not full, WavingSketch inserts it to the Heavy Part; otherwise, WavingSketch additionally updates the Waving Counter with an equal probability of $+1/-1$. To find super-spreaders, WavingSketch cooperates with a Bloom filter (BF) [32] to remove duplicates. Given an item $< src, dst >$, WavingSketch first checks the Bloom filter, to find whether the item is a duplicate. If not, WavingSketch insert $< src, dst >$ to the Bloom filter, and insert src to the sketch.

Applying LadderFilter: We build a LadderFilter between the Bloom filter and WavingSketch. LadderFilter will be used to prevent infrequent items from being inserted into the WavingSketch **after removing duplicates**.

Insertion: When inserting an item $< src, dst >$, we first check the Bloom filter, to find out whether the item is a duplicate. We discard the duplicate. We then check whether src is already recorded in the Heavy Part of WavingSketch. If so, we insert the item to it. Otherwise, we insert src into LadderFilter. If LadderFilter reports the item as a frequent item, we further insert it into WavingSketch.

Query: When querying super-spreaders, we report the frequent items reported by WavingSketch.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

Computation platform: We conduct all experiments on a CPU server (Intel i9-10980XE). The CPU has three levels of caches: 64KB L1 cache and 1MB L2 cache for each core, and 24.75MB L3 cache shared by all cores. We set the CPU frequency to 4.2GHZ and the memory frequency to 3200MHZ.

Implementation: We implement LadderFilter (Ours), ColdFilter (CF) [10], and LogLogFilter (LLF) [22] in C++, and apply them to the CU sketch [8], SpaceSaving (SS) [11], FlowRadar (FR) [13], and WavingSketch (WS) [16].

Datasets: The datasets used for the evaluation are listed below.

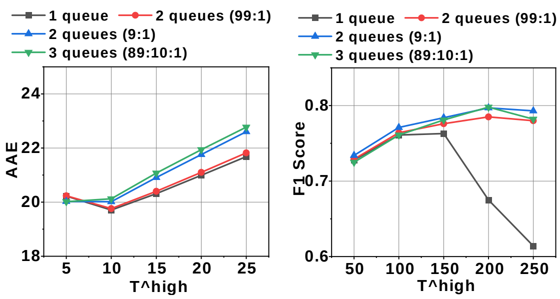
- *IP trace dataset:* The IP trace dataset is an anonymized IP trace streams collected from [26]. We use *srcIP* as the item ID in the former three tasks. The dataset contains 27M items, with 250k distinct items. We use a 10× longer dataset for finding super-spreader, and use $< srcIP, dstIP >$ as the item ID.
- *WebDocs dataset:* The WebDocs dataset is a transactional dataset built from a collection of web documents [34]. The dataset contains 32M items, with 950k distinct items.
- *Synthetic datasets:* The two synthetic datasets are generated following the Zipf distribution [35]. The skewness of the two datasets are 0.5 and 1.0, respectively. Each dataset contains 32M items, with 1.0M distinct items.

Metrics: Metrics used for evaluation are listed below.

- *Average Absolute Error (AAE):* $\frac{1}{N}\sum_{i=1}^{N}|f_i - \hat{f_i}|$, where N is the number of distinct items, f_i and $\hat{f_i}$ are the actual and estimated frequency of the items respectively.
- *F1 Score:* $\frac{2 \cdot PR \cdot RR}{PR+RR}$, where PR (Precision Rate) is the ratio of the number of the correct items reported to the number of all items reported, and RR (Recall Rate) is the ratio of the number of the correct items reported to the number of all correct items.
- *Throughput:* The number of operations per second, in million operation per second (Mops).

4.2 Parameter Settings

In this section, we first propose the parameter adjusting method. Then, we show experiments on some important parameters.



(a) Estimating frequency. (b) Finding top-k items.

Figure 5: Impact of queue number and size.

4.2.1 Parameter Adjusting Method.

Methodology: When LadderFilter is cooperating with a sketch, there are two sources of error: 1) the under-estimation error caused by LadderFilter's discarding some unpromising items; 2) the inherent error caused by the sketch, which could be under-estimation, over-estimation, or bidirectional. The parameters can affect both of them at the same time. *Our parameter setting methodology is to balance the under-estimation error and the inherent error.* Take T^{high} as an example. If T^{high} is too large, it will lead LadderFilter to discard too many items, and thus result in large under-estimation error. If T^{high} is too small, too many items will be inserted into the sketch, resulting in large inherent error. Therefore, our method for adjusting the parameters is to 1) analyze the nature of the two sources of error, and 2) find a variable $\mathcal{V}$ that can reflect the overall error and minimize it.

LadderFilter+CU: LadderFilter only leads to under-estimation error, while CU only leads to over-estimation error. Therefore, the variable $\mathcal{V}$ that we choose to reflect the overall error is the difference between the total under-estimation and the total over-estimation of all items. When adjusting the parameters, for each round, we build a LadderFilter+CU. After each round of insertion, we calculate the total under-estimation and the total over-estimation of all items, respectively. If the under-estimation and the over-estimation are almost equal, we consider that we have obtained an optimal parameter. If the under-estimation is smaller/larger than the over-estimation, we adjust the threshold to a larger/smaller value, respectively, and then proceed to the next round of parameter adjustment.

LadderFilter+SS: Similar to CU, SS only leads to over-estimation error. The parameter adjustment process of LadderFilter+SS is similar to that of LadderFilter+CU, except that we only calculate the under-/over-estimation of the items in SS.

LadderFilter+FR: FR can be considered as a zero-error hash table when its loading rate (number of distinct item [6] : number of buckets) is lower than a theoretical maximum value of around 80% [13]. When its loading rate exceeds the theoretical maximum value, it can hardly be decoded, and thus all the items inserted into it become error. Therefore, the only error is the under-estimation error caused by LadderFilter when the loading rate of FR is lower than the theoretical maximum value. We find that the fewer items are filtered, the smaller the under-estimation error caused by LadderFilter, meanwhile the higher the loading rate of FR. Therefore, the variable $\mathcal{V}$ that we choose to reflect the overall error is the loading rate of FR. To minimize the error, the loading rate of FR should be as higher as possible while lower than the theoretical maximum value. Therefore, when adjusting the parameters, for each round, we compute the loading rate. If the loading rate is too small/large, we adjust the threshold to a smaller/larger value, respectively.

LadderFilter+WS: We still choose the difference between the total under-estimation and the total over-estimation of all items. Unlike CU and SS, WS leads to bidirectional error. According to our many experimental tests, we observe that when the under-estimation is slightly larger than the over-estimation, the accuracy reaches the optimal value.

[6]The number of distinct items can be estimated quickly by linear counting [36].

4.2.2 Experiments on Parameter Settings.

Impact of queue number and size (Figure 5): We find that when using multiple queues to find top-k items, the accuracy is insensitive to different parameter settings. As shown in Figure 5(b), under the near-optimal threshold (50 in the figure, the best observed value of T^{high} in our experiment), both single queue and multiple queues achieve high accuracy; while under other thresholds (> 150 in the figure), the accuracy of using multiple queues, however, is significantly higher than that of using a single queue. As shown in Figure 5(a), when estimating item frequency, the trend is opposite. We find that when using 2 queues and setting $M_{Q_1} : M_{Q_2}$ to 99 : 1, LadderFilter achieves near optimal accuracy. Therefore, we recommend using these parameters. Note that using 3 or more queues may help to find frequent items in other datasets and scenarios, and we remain this design.

Impact of # cells per bucket (Figure 6): We find that when # cells per bucket exceeds 8, the accuracy stops increasing. The F1-score of 8 cells per bucket is on average 1.35% lower than more cells per bucket. Therefore, we recommend setting # cells per bucket to 8 to balance the accuracy and ease of deployment.

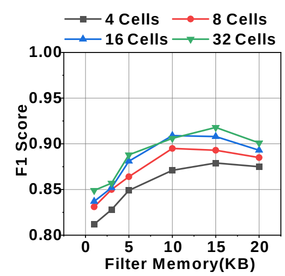
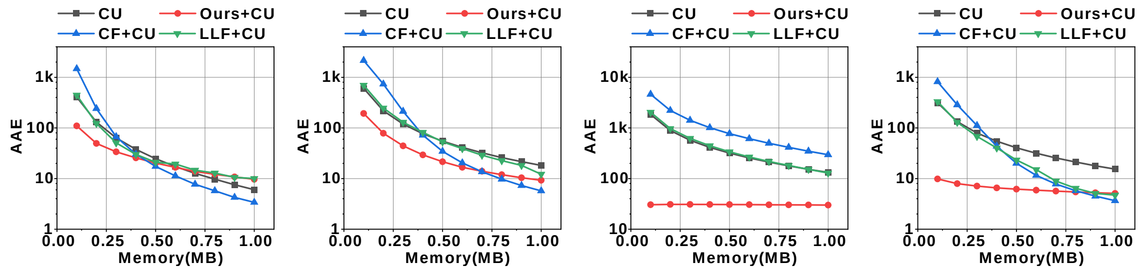


Figure 6: Impact of # cells per bucket.

4.3 Experiments on Estimating Item Frequency

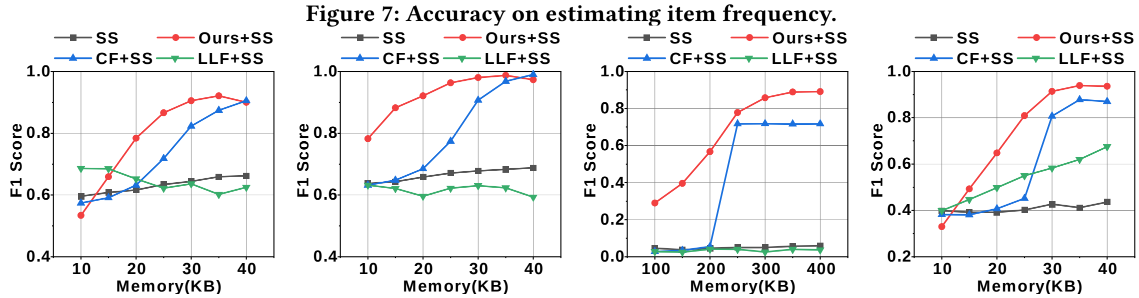
In this section, we compare LadderFilter+CU with CU, CF+CU, and LLF+CU. For Ours+CU, we set the memory of filter and sketch $M_{Ours} : M_{CU} = 1 : 9$. We set parameters of the compared algorithms to the recommended values referred to their respective papers.

Accuracy (Figure 7): We find that LadderFilter reduces the error of CU by up to 28.8 times. As shown in Figure 7, the AAE of LadderFilter is on average 7.43, 15.2, and 7.29 times lower than that of CU, CU+CF, and CU+LLF, respectively. Note that LadderFilter achieves high accuracy under limited memory. For example, when the memory is 100KB, the AAE of LadderFilter is on average 7.08, 5.95, 93.0, and 49.5 times lower than the compared algorithms on each datasets, respectively. The reason is that LadderFilter approximately discards infrequent items from the filter, while CF and LLF keep all infrequent items. Therefore, LadderFilter consumes less memory, and can use it more efficiently.



(a) IP trace. (b) WebDocs. (c) Zipf 0.5. (d) Zipf 1.0.

Figure 7: Accuracy on estimating item frequency.



(a) IP trace. (b) WebDocs. (c) Zipf 0.5. (d) Zipf 1.0.

Figure 8: Accuracy on Finding top-k items.

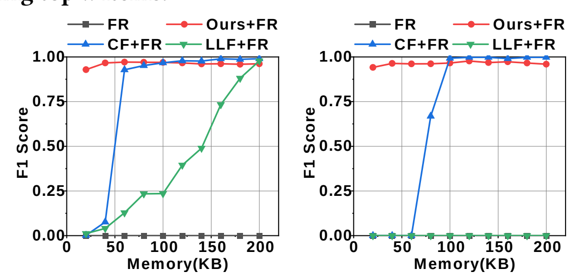
Throughput (Figure 11): We find that LadderFilter achieves higher throughput compared to CF [7] and LLF. As shown in Figure 11, the throughput of LadderFilter is 1.17 and 1.09 times higher than that of CU+CF and CU+LLF, respectively.

Discussions on different datasets: The accuracy of LadderFilter varies among different datasets. There are mainly two reasons. First, the skewness varies among different datasets. The accuracy of LadderFilter is related to the accuracy of the dedicated data structure it cooperated with, and the accuracy of the dedicated data structures is usually highly correlated with the skewness of the data stream [9, 23, 37]. Therefore, LadderFilter has different accuracy on datasets with different skewness. This feature is more evident in synthetic datasets (see Figure 7(c)-(d) & 8(c)-(d)). Second, the arrival pattern of items varies among different datasets. For example, in synthetic datasets, items arrive in random order. In the IP trace dataset, items arrive in the pattern of burst [17]. Therefore, the time it takes an infrequent item to become a frequent item varies among datasets. Our experiments have demonstrated that LadderFilter can work well for different skewnesses and arrival patterns.

4.4 Experiments on Finding Top-k Items

In this section, we compare LadderFilter+SS with SS, CF+SS, and LLF+SS. We set k to 1000. For filter+SS, we set the number of items in SS to $1.5 \times k$. For the original SS, we additionally record $\frac{M_{filter}}{100B}$ items for comparison fairness [8].

Accuracy (Figure 8): We find that LadderFilter improves the accuracy of SS by up to 17.2 times. As shown in Figure 8, the F1 Score of LadderFilter is on average 0.330, 0.130, and 0.310 higher than the one of SS, SS+CF, and SS+LLF, respectively. Note that LadderFilter achieves high accuracy even with very little memory. With only 30KB, 20KB, 350KB, and 30KB of memory, the F1 Score of LadderFilter exceeds 0.9 on each dataset, respectively.



(a) IP trace. (b) WebDocs.

Figure 9: Accuracy on finding heavy changes.

Throughput (Figure 11): We find that LadderFilter improves the throughput of SS. As shown in Figure 11, the throughput of LadderFilter is 1.29, 1.67, and 2.73 times higher than the one of SS, SS+CF, and SS+LLF, respectively.

4.5 Experiments on Finding Heavy Changes

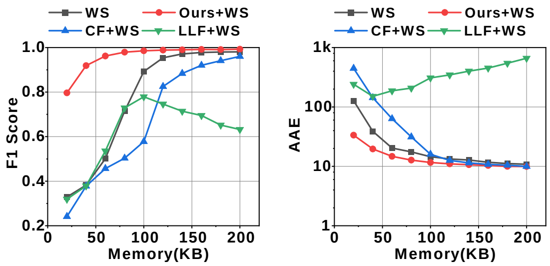
In this section, we compare LadderFilter+FR with FR, CF+FR, and LLF+FR. We set the threshold of heavy changes $\mathcal{T}_\Delta$ to 0.01% of total item number. For filter+FR, we allocate 1MB memory for FR.

Accuracy (Figure 9): We find that LadderFilter with limited memory can filter the infrequent items inserted into FR, so that FR can be decoded successfully. As shown in Figure 9, with only 20KB of memory, the F1 Score of LadderFilter exceeds 0.9 on both datasets. The required memory of filter is on average 4.0 and 14.5 times lower than that of CF and LLF, respectively. Note that to successfully decode, FR requires more than 2.7MB and 9.6MB of memory, respectively; FR+LLF requires more than 400KB of filter memory on the Web page dataset.

Throughput (Figure 11): We find that LadderFilter improves the throughput of FR. As shown in Figure 11, the insertion throughput of LadderFilter is 1.37, 1.61, and 1.78 times higher than the one of FR, FR+CF, and FR+LLF, respectively.

[7]The results do not include aggregate-and-report, because this optimization is orthogonal to the filter, and can be applied to any compared algorithm.

[8]Existing works show that the memory usage of each item in SS is around 100B [10, 22].



(a) F1 Score (b) AAE

Figure 10: Accuracy on finding super-spreaders.

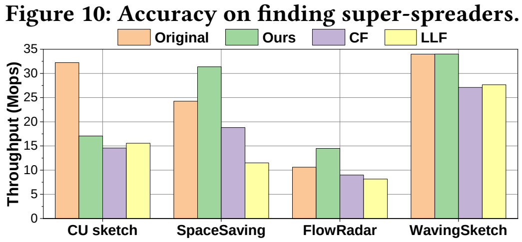


Figure 11: Throughput on four data stream tasks.

4.6 Experiments on Finding Super-Spreaders

In this section, we compare LadderFilter+WS with WS, CF+WS, LLF+WS. We set the threshold of super-spreaders $\mathcal{T}$ to the number of destinations connected to the 1500^{th} super-spreader. To remove duplicates, we allocate 5MB for BF. For Ours+WS, we set the memory of filter and sketch $M_{Ours} : M_{WS} = 3 : 7$.

Accuracy (Figure 10): We find that LadderFilter improves the accuracy of WS by up to 2.42 times. As shown in Figure 10, the F1 Score of LadderFilter is on average 0.191, 0.291, and 0.341 higher than the one of WS, CF+WS, and LLF+WS, respectively. The AAE of LadderFilter is on average 1.55, 3.41, and 30.4 times lower than the one of WS, CF+WS, and LLF+WS, respectively.

Throughput (Figure 11): We find that LadderFilter achieves a comparable throughput with WS, and higher throughput compared with CF and LLF. As shown in Figure 11, the throughput of LadderFilter is 1.26 and 1.23 times higher than the one of CF+WS and LLF+WS, respectively.

5 RELATED WORK

In this section, we first introduce existing solutions for filtering infrequent items. Then we introduce existing solutions for four typical tasks in data stream processing.

5.1 Filtering Infrequent Items

In skewed data streams, filtering infrequent items is an important strategy to improve the accuracy of tasks favoring frequent items. The most relevant works to LadderFilter are ColdFilter (CF) [10] and LogLogFilter (LLF) [22]. ColdFilter uses an additional sketch to filter infrequent items, and only inserts frequent items to the dedicated sketch. ColdFilter relies on a 2-layer CU sketch [8] with different-sized counters. The counter size of the first layer is small (*e.g.*, 4 bits), and the counter size of the second layer is large (*e.g.*, 16 bits). For every incoming item, ColdFilter first inserts it to the first layer. If all mapped counters in the first layer overflow, ColdFilter then inserts it to the second layer. ColdFilter is also associated with a threshold for identifying frequent items. If the frequency of an item exceeds the threshold, ColdFilter reports the item as a frequent item. By filtering the infrequent items, ColdFilter improves the accuracy of frequent items. However, ColdFilter falls short in terms of memory efficiency as it records the approximate frequency of all items. Further, it requires multiple hash computations and memory accesses, and thus is less time efficient.

LogLogFilter [22] replaces the CU sketch in ColdFilter by a LogLog structure [25], so as to enlarge the filter range. LogLogFilter is a register array associated with multiple hash functions and a random generator. For every incoming item, LogLogFilter first computes hash functions to locate the corresponding registers, and decides whether the item is an infrequent item. If so, LogLogFilter generates random numbers that follow a geometric distribution and updates the corresponding registers. LogLogFilter inherits the advantages and limitation of ColdFilter, and thus also falls short in terms of both memory and time efficiency.

On top of the previous two works, many sketches record frequent and infrequent items separately. Typical sketches include ASketch [9], HeavyGuardian [28], ElasticSketch [27], NitroSketch [24], SeqSketch [38], *etc.* [17, 23, 29, 39].

5.2 Data Stream Processing Tasks

Estimating item frequency: Classic solutions in estimating item frequency include the CM (Count-Min) sketch [20], the CU (Conservative Update) sketch [8], and the Count sketch [21]. A CM sketch consists of multiple counter arrays and hash functions for mapping items to counters in counter arrays. The CM sketch increments the mapped counters by 1 during insertion, and reports the minimum value of the mapped counters during query. The CU sketch applies a conservative update strategy to the CM sketch, and thus improves the accuracy. The Count sketch also consists of multiple counter arrays and hash functions. It updates each counter with an equal probability of +1/-1, and thus achieves unbiased estimation.

Finding top-k items: Typical solutions in finding top-k items include SpaceSaving [11], Unbiased SpaceSaving [40], *etc.* [12, 16, 28, 41]. SpaceSaving maintains top-k items and their frequency using a data structure called Stream-Summary, and guarantees no underestimated error. Unbiased SpaceSaving applies a probabilistic replacement strategy to SpaceSaving for unbiased estimation.

Finding heavy changes: A kind of solution in finding heavy changes is to record all items in each time window, and then compare the two consecutive time windows and report heavy changes. Typical solutions include FlowRadar [13], k-ary [14], and the reversible sketch [42].

Finding super-spreaders: A kind of solution in finding super-spreaders is to combine an existing sketch with a bitmap/Bloom filter to remove duplicates. Typical solutions include OpenSketch [43] and WavingSketch [16].

6 CONCLUSION

In this paper, we proposed LadderFilter, which filters infrequent items with limited memory and time overhead. To achieve memory efficiency, LadderFilter relies on multiple LRU queues to discard unpromising items, instead of keeping all frequent and infrequent items. To achieve time efficiency, we leverage SIMD instructions to implement a LRU policy. LadderFilter can be applied to various sketches, and can significantly improve their accuracy and throughput. All related code is provided open-source at Github.

REFERENCES

[1] Sang-Hyun Oh, Jin-Suk Kang, Yung-Cheol Byun, Taikyeong T Jeong, and Won-Suk Lee. Anomaly intrusion detection based on clustering a data stream. In *International Conference on Information Security*, pages 415–426. Springer, 2006.

[2] Mustafa Amir Faisal, Zeyar Aung, John R Williams, and Abel Sanchez. Securing advanced metering infrastructure using intrusion detection system with data stream mining. In *Pacific-Asia Workshop on Intelligence and Security Informatics*, pages 96–111. Springer, 2012.

[3] Şule Gündüz and M Tamer Özsu. A web page prediction model based on click-stream tree representation of user behavior. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–540, 2003.

[4] Vincent S Tseng and Kawuu W Lin. Efficient mining and prediction of user behavior patterns in mobile web systems. *Information and software technology*, 48(6):357–369, 2006.

[5] Bryan Ball, Mark Flood, Hosagrahar Visvesvaraya Jagadish, Joe Langsam, Louiqa Raschid, and Peratham Wiriyathammabhum. A flexible and extensible contract aggregation framework (caf) for financial data stream analytics. In *Proceedings of the International Workshop on Data Science for Macro-Modeling*, pages 1–6, 2014.

[6] Andr LL De Aquino, Carlos MS Figueiredo, Eduardo F Nakamura, Luciana S Buriol, Antonio AF Loureiro, Antnio Otvio Fernandes, and JN Claudionor Jr. Data stream based algorithms for wireless sensor network applications. In *21st International Conference on Advanced Information Networking and Applications (AINA'07)*, pages 869–876. IEEE, 2007.

[7] Dhivya Eswaran, Christos Faloutsos, Sudipto Guha, and Nina Mishra. Spotlight: Detecting anomalies in streaming graphs. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1378–1386, 2018.

[8] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. *ACM SIGMCOMM CCR*, 32(4), 2002.

[9] R. Pratanu, K. Arijit, and A. Gustavo. Augmented sketch: Faster and more accurate stream processing. In *Proc. ACM SIGMOD*, 2016.

[10] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *SIGMOD Conference*, 2018.

[11] 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*. Springer, 2005.

[12] Junzhi Gong, Tong Yang, Haowei Zhang, Hao Li, Steve Uhlig, Shigang Chen, Lorna Uden, and Xiaoming Li. Heavykeeper: An accurate algorithm for finding top-k elephant flows. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 909–921, 2018.

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

[14] K. Balachander, S. Subhabrata, Z. Yin, and C. Yan. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 234–247. ACM, 2003.

[15] S. Venkataraman, D. Xiaodong Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*, 2005.

[16] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1574–1584, 2020.

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

[18] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[19] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C Luizelli, and Erez Waisbard. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 127–140, 2017.

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

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

[22] Peng Jia, Pinghui Wang, Junzhou Zhao, Ye Yuan, Jing Tao, and Xiaohong Guan. Loglog filter: Filtering cold items within a large range over high speed data streams. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 804–815. IEEE, 2021.

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

[24] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. 2019.

[25] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *European Symposium on Algorithms*, pages 605–617. Springer, 2003.

[26] The CAIDA UCSD Anonymized Internet Traces 2018. https://www.caida.org/catalog/datasets/passive_dataset/.

[27] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM 2018*, pages 561–575, 2018.

[28] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. Heavyguardian: Separate and guard hot items in data streams. In *SIGKDD*, 2018.

[29] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2285–2293, 2021.

[30] Yang Zhou, Omid Alipourfard, Minlan Yu, and Tong Yang. Accelerating network measurement in software. *ACM SIGCOMM Computer Communication Review*, 48(3):2–12, 2018.

[31] Intel Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

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

[33] 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.

[34] Webdocs: a real-life huge transactional dataset. http://fimi.ua.ac.be/data/.

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

[36] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[37] Peiqing Chen, Yuhan Wu, Tong Yang, Junchen Jiang, and Zaoxing Liu. Precise error estimation for sketch-based flow measurement. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 113–121, 2021.

[38] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. Toward nearly-zero-error sketching via compressive sensing. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 1027–1044, 2021.

[39] Kaicheng Yang, Yuanpeng Li, Zirui Liu, Tong Yang, Yu Zhou, Jintao He, Tong Zhao, Zhengyi Jia, Yongqiang Yang, et al. Sketchint: Empowering int with towersketch for per-flow per-switch measurement. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2021.

[40] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *SIGMOD Conference*, 2018.

[41] M. Gurmeet Singh and M. Rajeev. Approximate frequency counts over data streams. In *Proc. VLDB*, pages 346–357, 2002.

[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] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI 2013*, 2013.