Finding Significant Items in Data Streams

Tong Yang[1], Haowei Zhang[1], Dongsheng Yang[1], Yucheng Huang[1], Xiaoming Li[1]
[1]Peking University, China

***Abstract*—Finding top-k frequent items has been a hot issue in data bases. Finding top-k persistent items is a new issue, and has attracted increasing attention in recent years. In practice, users often want to know which items are significant, *i.e.*, not only frequent but also persistent. No prior art can address both of the above two issues at the same time. Also, for high-speed data streams, they cannot achieve high accuracy when the memory is tight. In this paper, we define a new issue, named finding top-k significant items, and propose a novel algorithm namely LTC to address this issue. LTC can accurately report top-k significant items with small memory size. It includes two key techniques: Long-tail Replacement and a modified CLOCK algorithm. To prove the effectiveness of LTC, we theoretically prove there is no overestimation error and derive the correct rate and error bound. We further conduct extensive experiments on real datasets and synthetic datasets. The experimental results show that LTC achieves $300 \sim 10^8$ and in average 10^5 times higher accuracy than other related algorithms.**

I. Introduction

A. Background and Motivations

Nowadays, the volume of data becomes larger and larger, posing great challenges on fast queries. On the one hand, the exploding data provides more opportunities for users to better understand the world. On the other hand, it becomes harder and harder for users or administrators to find information that they care about most in time.

In many scenarios, users only care about the most significant part of the dataset, and want to know the answers immediately. For example, people want to know the most influential tweets under a certain topic. For another example, customers want to know which products are sold the best in a certain category.

In the above examples, it is often unnecessary to report an exactly correct answer because of the following two reasons. First, the data often contains noise, which means getting the exact correct answer cannot help to improve the performance of the applications. Second, such queries are often intermediate results, serving for the comprehensive queries. For example, when we find feature values in machine learning, we do not care whether the feature values are all useful, but we hope that the number of useful feature values to be as many as possible. In other words, as long as the error is small enough, it almost do not influence the final results. Therefore, approximate queries has gained rising attention, and a series of approximate data structures have been proposed and played important roles in the last several decades, including Bloom filter [1] and its variances [2]–[4], and sketches [5]–[9]. There are two advantages of approximate queries. First, maintaining an approximate data structure requires less memory and less computing resources. Second, approximate data structures can support fast insertion and query, so they can catch up with the fast speed of data streams. In summary, approximate query is an effective method for big data applications.

In the aforementioned scenarios, though people only care about the most significant part of the dataset, general purposed approximate query methods such as Bloom filters [1] and sketches [6] need to record the whole dataset. As a result, most memory is wasted, occupied by a large amount of insignificant items. Indeed, there are some works on finding top-k frequent items in datasets, which only keep a small part of the whole dataset. However, they only consider the frequency of items, which is not a sufficient condition for significant items. Next, we present the definition of significant items.

Definition of Significant Items: Given a data stream or a dataset, we divide it into T equal-sized `periods`. Each item could appear more than once in the data stream or in each period. The *Significance* s of an item is a function of two metrics: frequency f and persistency p.

$$s = \alpha f + \beta p \tag{1}$$

where α, β are parameters defined by users. Frequency refers to the number of appearances of items. Persistency refers to the number of periods where the items have appeared.

Finding top-k significant items is to report k items with the largest significance. Finding top-k significant items is a key component in many applications, and below we show three use cases.

Use Case 1: Paper selection. In top conferences, the most honorable award is the test of time award paper. The key metric for this award is the citations. First, in the last ten years, papers with the largest number of citations should be candidates. Second, papers with persistent citations have persistent impacts, and thus also should be candidates. In a nutshell, the award should consider the above two metrics of each paper to value the significance of the paper.

Use Case 2: Social networks. Take Facebook as an example. One user could have a great number of friends, but only a few of them are significant to the user. Therefore, it is important for companies to find out the intimate friends of each user in order to build a model for predicting or promoting information dissemination. We argue that if two users 1) send many messages to each other, and 2) persistently send messages in a long period (*e.g.*, 2 years). These two users are very likely to be intimate friends. In a nutshell, finding intimate friends is to find the significant items among all friends of the user.

Use Case 3: Network congestion. Network congestion happens every second in data centers [10]–[12]. One effective solution for resolving congestion is to change the forwarding path of some flows[1]. As there could be millions of flows in every second, a straightforward solution is to change the forwarding paths of many flows, and then many entries of the forwarding table in the switch will be modified. Some algorithms for organizing the forwarding table do not support fast update. To avoid modifying the forwarding table frequently, it is desired to change as few flow paths as possible. A classic method is to only change the forwarding path of large flows. This method does not always work well, because the current large flows could be a burst, and there could be very few packets later. Therefore, changing the forwarding entry of such large flows is in vain. A better choice is to detect the significant flows. They are not only frequent, but also persistent, and thus with high probability they will be large flows in a long period later. Obviously, changing the significant flows is a better solution.

B. Prior Art and Their Limitations

Existing solutions focus on only one dimension, such as finding top-k frequent items [13]–[18], or finding top-k persistent items [19], [20]. However, in practice, finding both frequent and persistent items is often a more practical and important issue. One straightforward solution is to combine two kinds of algorithms. Specifically, for each incoming item, we build two data structures, one for recording the frequent items, and the other for recording the persistent items. When we need to report top-k significant items, we should check the items as long as they are in one of the two data structures. Obviously, the time and space overhead is the sum of the overhead of the two used algorithms. Therefore, existing one-dimension solution is inefficient in term of time and space. The *design goal* of this paper is to design one data structure to accurately and quickly find significant items using small memory.

C. Our Solution

In this paper, we propose a novel algorithm, namely Long-Tail Clock (LTC), which can accurately find top-k significant items with small memory and high speed. LTC includes two key techniques: Long-tail Replacement and a modified CLOCK algorithm.

First, we show how Long-tail Replacement works for finding frequent items. The process is similar for finding significant items. Suppose we want to use k cells to keep track of top-k frequent items. Each cell stores an item ID and its frequency $\langle e_i, f_i \rangle$. As k is usually small, the k cells will be full soon. The key problem is how to handle the new incoming item when the k cells are full. The most well known algorithm Space-Saving just lets the incoming item replace the smallest item $\langle e_{min}, f_{min} \rangle$ among all k items in those cells, and sets the initial value of the new incoming item to $f_{min} + 1$. This can guarantee no underestimation error but at the cost of large overestimation errors. In contrast, our *Long-tail Replacement* works as follows. When a new item arrives, we decrement f_{min} by 1. And when the f_{min} becomes 0, e_{min} will be replaced by the new item. *The key technique* is how to choose an appropriate initial value for the new item. Our algorithm is based on the observation that item frequencies in real datasets follow *long-tail distribution* [21]–[25]. Therefore, we set the initial value of the new item to the *second smallest frequency minus 1*. Next we use two cases to explain why such an initial value is reasonable.

Case I: The new incoming item e appeared continuously for f_{min} times and thus expels e_{min}. The initial value of e should be set to f_{min}. However, we do not know the value of f_{min}. Fortunately, according to the long-tail distribution, f_{min} is approximately the *second smallest frequency minus 1*.

Case II: The new incoming item e seldom appeared before, this strategy might cause the initial value of e is overestimated. However, it will be decremented and expelled soon from the k cells.

Based on the long-tail distribution, there is no other appropriate initial value, such as 0, 1, the second smallest frequency or any other value. More details about Long-tail Replacement are provided in Section III-D.

Second, we show how the modified CLOCK algorithm accurately records the item persistencies: the numbers of periods that each item appears in. The most challenging issue is that we need to increment the persistency *by one* for any item that appears more than once in one period. To address this issue, we leverage the spirit of the well known CLOCK algorithm [26]–[28], and propose a modified CLOCK algorithm as follows. Suppose we have k cells, and each cell has a flag. When a cell is updated, its flag is set to true. We use the modified CLOCK algorithm to scan all cells with a thread. If the flag is true, we just increment the corresponding persistency by 1 and reset the flag to false. When a period ends, all cells will be scanned exactly once and the scanning restarts from the beginning of the next period. Such a design can perfectly overcome the mentioned challenge. More details are provided in Section III-B. We further improve this basic version in Section III-C.

II. Related Work

To the best of our knowledge, there is no prior work to deal with finding top-k significant items. A straightforward solution is to combine two kinds of algorithms: one for finding frequent items and the other for finding persistent items. Here we briefly describe the prior works for these two issues.

A. Finding Top-k Frequent Items

For finding top-k frequent items, existing algorithms can be divided into two categories: counter-based and sketch-based. **Counter-based:** Counter-based algorithms include Space-Saving(SS) [16], Lossy Counting(LC) [18], CSS [17], etc. These algorithms are similar to each other. Take Space-Saving as an example, it maintains several cells, each cell records a pair $\langle e_i, f_i \rangle$, where e_i is an item ID and f_i is the estimated frequency of e_i. When an item arrives, it first judges whether this item matches one of the cells. If there is a match, denoted by the j^{th} cell, SS increments f_j by 1. Otherwise it finds

[1]A flow is usually defined as a part of the five tuples: source IP address, destination IP address, source port, destination port, and protocol.

the item whose estimated frequency is the smallest, denoted by $\langle e_{min}, f_{min}\rangle$. Then it replaces e_{min} by the new item ID, and increments f_{min} by 1. It uses a structure named Stream-Summary to accelerate the speed for the above operations. SS has no underestimation error, which means the estimated frequency is always no less than the real frequency. However, with the insertion of flows, the estimated frequencies are always incremented, leading to serious overestimation errors.

Sketch-based: Sketch-based algorithms include Count Sketch [5], CM Sketch [6], etc. They are similar to each other. Take CM sketch as an example, it uses multiple equal-sized buckets associated with different hash functions h_i to record frequencies. Each bucket is comprised of several cells. When an item e arrives, each bucket first computes $h_i(e)$ to map e to the cell $A[i][h_i(e)]$, then increments the value of that cell by 1. For each item, the estimated frequency is the smallest value of all the mapped cells. Similar to Counter-based algorithms, it has no underestimation error, but with the insertion of flows, the values of the cells are always incremented. Moreover, if an infrequent item is collided with frequent items in all the mapped cells, it will be wrongly reported as a frequent item. Therefore, the number of cells in each bucket needs to be large enough to avoid hash collisions as much as possible. To report top-k frequent items, it needs to maintain a min-heap to record and update top-k frequent items.

B. Finding Top-k Persistent Items

For finding top-k persistent items, there are several existing algorithms, such as coordinated 1-sampling [20] and PIE [19]. Because coordinated 1-sampling focuses on improving the performance of distributed data streams, we do not introduce it in detail. The state-of-the-art algorithm is PIE. The key idea of PIE is to use Raptor codes to record and identify item IDs, and to find the persistent items. It executes each period one by one. During each period, it maintains a data structure called Space-Time Bloom Filter and uses Raptor codes to encode the ID of items appeared in this period. Finally, it gathers all Space-Time Bloom Filters and decodes each item by the recorded raptor codes. The property of the raptor codes is that the more persistent the item, the higher the success rate of decoding. Therefore, if the persistence of an item is large enough, it will be successfully decoded and reported as a persistent item.

Besides, we can modify CM sketch to be adapted to find top-k persistent items. The thorniest problem is that some item might appear more than once in one period, which means we cannot update persistency directly. To deal with this problem, we maintain a conventional Bloom filter(BF) [1] to record whether it has appeared in the current period. Specifically, when an item comes, it first judges if it has appeared in the current period by querying the conventional BF, and it is inserted into CM sketch if and only if the conventional BF reports false. When the process ends, we insert this item into conventional BF to indicate that it has appeared. We also need to maintain a min-heap to assist finding top-k persistent items.

III. The LTC Algorithm

In this section, we first describe the data structure and operations of LTC in details. The key idea of LTC is to only keep track of the items with high potential to be significant. Second, we propose two novel optimizations: 1) Deviation Eliminator and 2) Long-tail Replacement. Table I lists all symbols and abbreviations used in this paper.

TABLE I: Symbols & abbreviations used in the paper.

Symbol	Description
e_i	The i^{th} item
A	The lossy table
$A[i]$	The i^{th} bucket in the lossy table
$A[i][j]$	The j^{th} cell in $A[i]$
$A[i][j].ID$	The ID field in $A[i][j]$
$A[i][j].f$	The frequency field in $A[i][j]$
$A[i][j].counter$	The counter of the persistency field in $A[i][j]$
$A[i][j].flag$	The flag of the persistency field in $A[i][j]$
$h(.)$	The hash function associated with A
N, M, T, w	# items, distinct items, periods, buckets
d	# cells in each bucket
α, β	The coefficients of frequency, persistency

A. Data Structure

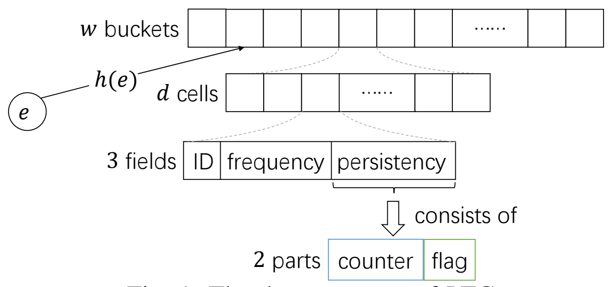


Fig. 1: The data structure of LTC.

As shown in Figure 1, the data structure of our LTC is a lossy table comprised of w buckets. Each bucket is comprised of d cells. Each cell is used to store the 3-tuples: $\langle key, frequency, persistency\rangle$. The key field is the item ID; The frequency field stores the estimated number of appearances of the item; The persistency field consists of two parts: a **counter** to store the estimated persistency and a **flag** bit to indicate whether it has appeared in the current period. Let $A[i][j]$ be the j^{th} cell of the i^{th} bucket in the lossy table, and let $A[i][j].ID$, $A[i][j].f$, $A[i][j].flag$ and $A[i][j].counter$ be the ID field, frequency field, flag and counter in the cell $A[i][j]$, respectively. Let $\alpha * A[i][j].f + \beta * A[i][j].counter$ be the significance of $A[i][j]$, where α and β are parameters defined by users. Among all cells in one bucket, we call the cell with the smallest significance `the smallest cell`.

B. Operations

We call a cell is `empty` if and only if the ID field is NULL and the significance of this cell is equal to 0. The initialization of LTC is to set all the cells in the lossy table to empty and set all flags to false.

1) Insertion:

For each incoming item e at time t, it first computes the hash function $h(e)$ to map e to bucket $A[h(e)]$. According to the cells of $A[h(e)]$, there are 3 cases as follows:

Case 1: e is found in a cell of $A[h(e)]$, denoted by $A[h(e)][j]$. In this case, LTC sets $A[h(e)][j].flag$ to true, and *increments* $A[h(e)][j].f$ by 1. It means that e has appeared in the current time period, and its total frequency is increased. The persistency will be incremented in the next period, which will be detailed later.

Case 2: e is not found in any cell of $A[h(e)]$, but there is an empty cell $A[h(e)][j]$. In this case, LTC inserts e into $A[h(e)][j]$. Note that in this basic version, inserting e into $A[h(e)][j]$ is to set $A[h(e)][j].ID$ to e, $A[h(e)][j].f$ to 1, $A[h(e)][j].flag$ to true, and $A[h(e)][j].counter$ to 0, which means it appears for the first time.

Case 3: e is not found in any cell of $A[h(e)]$, and there is no empty cell in $A[h(e)]$. In this case, LTC first finds `the smallest cell` in this bucket. And then it performs the `Significance Decrementing` operation on this cell. After that, if there is an empty cell, LTC inserts e into the empty cell.

Significance Decrementing: Suppose that it is performed on a cell $A[i][j]$. LTC first decrements $A[i][j].counter$ by 1, and then decrements $A[i][j].f$ by 1. After that, if the significance of $A[i][j]$ is 0, the item in $A[i][j]$ is expelled (*i.e.*, deleted) and $A[i][j]$ is made to be empty.

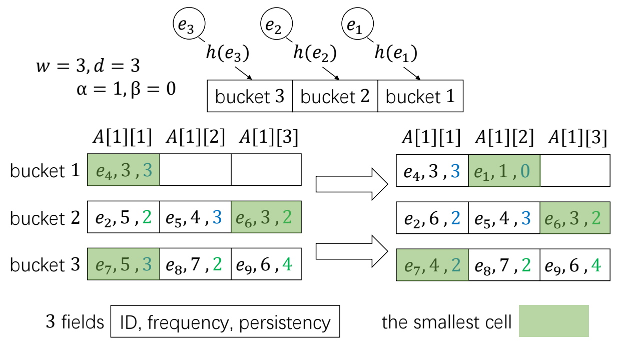


Fig. 2: An example for inserting an item.

Example: As shown in Figure 2, we set $w = 3, d = 3, \alpha = 1, \beta = 0$, where w and d are respectively the number of buckets and the number of cells in each bucket. For the persistency field in the figure, let the value of the number denote the counter, and let the color denote the flag: blue for true and green for false. For example, $< e_7, 5, 3 >$ means item e_7 has a frequency of 5 and persistency of 3. The color of 3 is blue means that the flag is true. Another example, $< e_8, 7, 2 >$ means e_8's persistency is 2, and the flag is false. When e_1 arrives, LTC first computes the hash function to map e_1 to $A[1]$. e_1 is not found in $A[1]$ but there is an empty cell $A[1][2]$. Therefore, e_1 is inserted into $A[1][2]$ which is an empty cell. When e_2 arrives, it is mapped to $A[2]$ and $A[2][1].ID$ is equal to e_2. Therefore, LTC sets $A[2][1].flag$ to true (the color is changed from green to blue in the figure), and increments $A[2][1].f$ by 1 (from 5 to 6). When e_3 arrives, it is mapped to $A[3]$, but e_3 is not found in this bucket and there is no empty cell in $A[3]$. Therefore, LTC finds the smallest cell $A[3][1]$, and performs Significance Decrementing operation on this cell: LTC decrements $A[3][1].counter$ by 1 (from 3 to 2), and decrements $A[3][1].f$ by 1 (from 5 to 4). After that, there is still no empty cell, e_3 is not inserted into the lossy table.

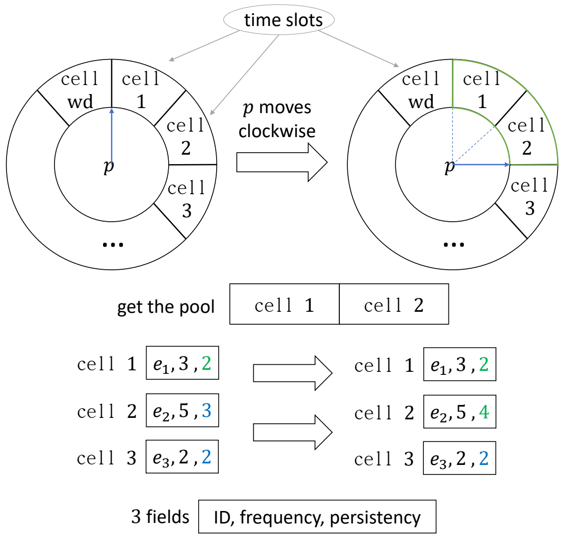


Fig. 3: The CLOCK and Persistency Incrementing.

Persistency Incrementing: To accurately record the persistency, we should increment the persistency only by 1 for items that appear one or more times in one period. To achieve this, we leverage the spirit of the well-known CLOCK algorithm [26]–[28]. Our key idea is as follows. We use a pointer to scan the lossy table, and check the flag. After a period ends, the whole lossy table is just scanned once, and will be scanned from the beginning again. Given a bucket with an item e, if the corresponding flag is true, it means that item e appeared at least once. In this case, we increment its persistency by 1, and reset the flag to false. Otherwise, we do nothing. Our method is essentially a lazy update strategy. Such a strategy can avoid additional incrementing of persistency.

To be clearer, we use a figure to show how our technique works. As shown in Figure 3, every cell corresponds to a *time slot* in the CLOCK. The pointer p points to the current time. At the beginning of each period, p points to the first slot of the CLOCK, and p moves clockwise and passes slots. For convenience, we suppose the arriving speed of every item is the same. In order to guarantee that p passes all cells exactly once, the time of scanning the whole lossy table needs to be equal to the length of each period. To achieve this, suppose a period contains n arriving items and suppose there are m cells/time slots in total, the step length is m/n. In other words, every time when each incoming item is processed, p passes m/n time slots. In this way, at the beginning of each new period, p will exactly move to the starting position again, and thus p passes all cells exactly once. For each incoming item, p passes m/n time slots, LTC inserts the corresponding m/n cells into a `pool`. For each cell in the pool, LTC processes it as follows:

Case 1: The flag of this cell is false. In this case, we just delete this cell from the pool.

Case 2: The flag of this cell is true. In this case, we increment the counter by 1, reset the flag to false, and delete this cell from the pool.

When all cells in the pool are processed, the pool will be empty to serve for the next incoming item.

Example: Similar to the previous example, the color of the persistency field indicates the value of the flag: blue for true and green for false. As shown in Figure 3, there are totally wd cells (w buckets and d cells in each bucket), the pointer p moves clockwise and passes both cell 1 and cell 2. LTC inserts these two cells into the pool. For cell 1, the flag is false, nothing is changed. For cell 2, the flag is true, thus its persistency is changed from 3 to 4, and the flag is reset to false (the color is changed from blue to green in the figure).

2) Query:

To query an item e, LTC checks the bucket $A[h(e)]$. If e matches a cell in this bucket, it reports the corresponding significance of this item, otherwise it reports this item did not appear. To find top-k significant items, LTC reports the largest k significant items recorded in the lossy table.

3) Advantages:

LTC can achieve that the items stored in LTC are those with high probability to be significant. With such a design, LTC discards most of the insignificant items and saves much memory. On the one hand, the item with high significance has a high probability to be significant. On the other hand, the newly inserted item still has a chance to be significant. Thus, when a new item arrives, we choose to hurt the most insignificant items for all considerations. The Significance Decrementing operation perfectly achieves such a design, it always decrements the significance of the smallest cells. When an item arrives continuously and turns the significance of the smallest cell to 0, it will be inserted into the lossy table. Our experimental results show that the accuracy of it is much better than other related algorithms.

LTC can accurately record the persistency of items stored in our lossy table. The challenge is that how to increment the persistency *by one* for any item that appears more than once in a period. To address this challenge, we use the modified CLOCK algorithm to deal with the update of persistency. As mentioned above, each cell will be inserted into the pool exactly once in a period. The persistency in the cell will be incremented only when the corresponding cell is deleted from the pool. That is to say, the persistency will be incremented by at most 1 in one period even though the corresponding item appeared many times, which exactly matches the definition of persistency.

4) Limitations:

First, the time of deleting the cell from the pool is different, so the period of each cell has a deviation from the real period. To address this problem, we propose an optimization namely Deviation Eliminator in Section III-C.

Second, the frequency and persistency of a newly inserted item are both initialized to 1. However, its real frequency and persistency tend to be much larger than 1, because it may have come many times to offset the original smallest item in the bucket. To address this problem, we propose an optimization namely Long-tail Replacement in Section III-D.

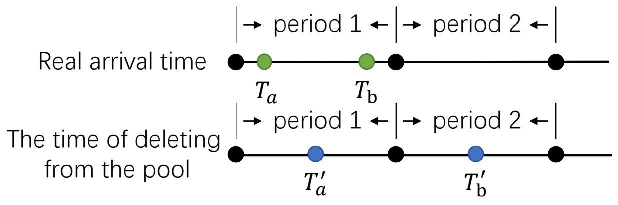


Fig. 4: An example for period deviation.

C. Optimization I: Deviation Eliminator

We use Figure 4 as an example to show how deviation happens. Given an item arriving at time T_a and T_b, the corresponding cell is deleted from the pool at time T'_a and T'_b. At time T_a, the flag is set to true. At time T'_a, LTC finds the flag is true, thus the persistency is incremented by 1, and the flag is reset to false. At time T_b, the flag is set to true. At time T'_b, the persistency is incremented by 1, and the flag is reset to false. In this way, the persistency is incremented twice. However, the real persistency is actually 1. The reason behind is that although the persistency is incremented by at most 1 in each period, the period of that cell has a deviation from the real period. Therefore, the recorded persistency might not be equal to the real persistency.

Our solution is based on this observation: using only one flag cannot differentiate the current period and the previous period, but the deviation is always less than one period. Therefore, as long as we use another bit to differentiate the current period and the previous period, the deviation can be totally eliminated. In our method, we use two flags instead of one flag in the basic version.

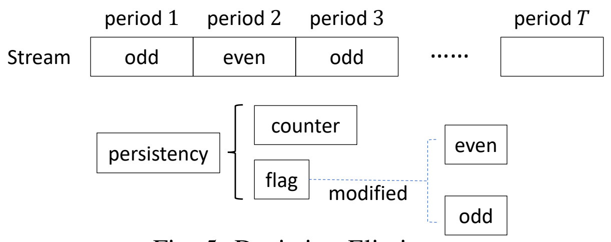


Fig. 5: Deviation Eliminator.

We use Figure 5 to explain how deviation eliminator works. We split all the periods into two parts: odd-numbered periods and even-numbered periods. Furthermore, we modify the flag of the persistency field into two flags: an `even` flag and an `odd` flag, indicating whether it has appeared in the current and the previous period.

The operations for frequency field are the same as the basic version. Here we show the operations for persistency field when an item e arrives in the even-numbered period. Under this assumption, *LTC always updates the even flag*. At first e is mapped to one bucket. Same as the basic version, there are three cases .

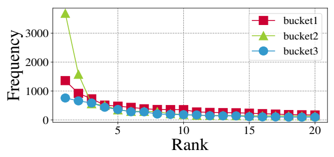


Fig. 6: Top-20 frequencies distribution of three buckets.

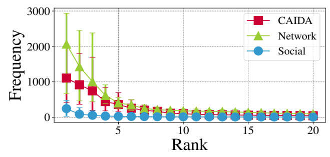


Fig. 7: Top-20 frequencies distribution on three datasets.

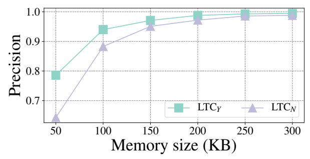


Fig. 8: Measuring precision, varying memory size.

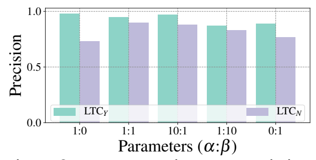


Fig. 9: Measuring precision, varying parameters.

In Case 1, e is found in a cell of $A[h(e)]$, denoted by $A[h(e)][j]$, LTC sets $A[h(e)][j].even$ to true.

In Case 2, e is not found in any cell of $A[h(e)]$, but there is an empty cell $A[h(e)][j]$. When e is inserted into the bucket, LTC sets $A[h(e)][j].even$ to true.

In Case 3, e is not found and there is no empty cell in $A[h(e)]$, the strategy is the same as that of the basic version.

The Significance Decrementing operation is the same as that of the basic version. The Persistency Incrementing operation is determined by the value of the odd flag. Specifically, for each cell in the pool, LTC checks the odd flag: If it is true, the corresponding persistency is incremented by 1 and the odd flag is reset to false; Otherwise, nothing is changed. Similarly, once a cell is processed, LTC deletes it from the pool.

In this way, we achieve that the recorded and utilized information corresponds to a real period, and thus the estimated persistency is exactly correct.

D. *Optimization II: Long-tail Replacement*

The key novelty of this paper is this optimization: Long-tail Replacement, as it significantly improves the accuracy of our algorithm. Next, we show the details of this optimization.

In practice, the real datasets often follow long-tail distribution. In our data structure of buckets, the frequencies of items mapped into one bucket are likely to follow long-tail distribution. This conclusion is consistent with our experimental results. As shown in Figure 6 and 7, we can observe for different datasets and different buckets, the frequencies still follow long-tail distribution.

Based on the above observations, we propose a novel optimization, namely Long-tail Replacement, to set the initial value of the newly inserted item. Specifically, we set the initial value of the newly inserted item to the second smallest value minus 1. The value can be only persistency or frequency.

Given a full bucket, when a new item e arrives, we decrement the value of the smallest cell by 1. And when the value becomes 0, e will be inserted into this bucket. As mentioned in Section III-B4, we set the initial value of e to 1 is not a good strategy. The reason behind is that with high probability it has arrived many times. The ideally initial value is the original smallest value. As mentioned above, item frequencies in real dataset often follow long-tail distribution, and it is similar to persistencies. Therefore, our strategy is to set the initial value of the new item to the second smallest value minus 1. In this way, the inserted cell is still the smallest cell, and we restore the value of the new item as accurate as possible. In the worst case, an infrequent or inpersistent item is inserted into the lossy table, the initial value might be large. However, with high probability it will be deleted soon as it seldom appears later.

When the value means significance, we decrement item with the smallest significance: decrementing both frequency and persistency by 1. When a new item is inserted into the bucket, we need to find the second smallest frequency and second smallest persistency to set the initial value of the newly inserted item.

We conduct experiments to compare the optimized version with the basic version. As shown in Figure 8 and Figure 9, *precision* means the ratio of the correct number of the reported top-k significant items to the number of real top-k significant items, *i.e.*, k, LTC$_Y$ means the optimized version, and LTC$_N$ means the basic version.

For the first experiment (Figure 8), we set $\alpha = 1, \beta = 1, d = 8, k = 1000$, and vary the memory size from 50KB to 300KB, the results show that the precision of LTC$_Y$ is always larger than that of LTC$_N$.

For the second experiment (Figure 9), we set the memory size to 50KB, $d = 8$, $k = 100$, and vary the parameters. The results show that the precision of LTC$_Y$ is still always larger than that of LTC$_N$.

IV. Mathematical Proofs

We first prove the basic version with Deviation Eliminator has no overestimation error on significance, and then derive the formula of the correct rate and the error bound. Due to Space limitation, we present the proofs in the Appendix of technical report [29].

A. *Proof of no Overestimation*

Theorem IV.1. *For an item e_i, let s_i and $\hat{s_i}$ be the real significance and estimated significance, respectively. We have*

$$\hat{s_i} \leqslant s_i \tag{2}$$

B. *The Correct Rate*

Lemma IV.1. *The reported significance of an item is correct if it satisfies the following conditions: 1) When the item arrives for the first time, the bucket is not full, and 2) Each time the corresponding cell is not the smallest cell.*

Given a data stream $\mathcal{S}$. Suppose that $\mathcal{S}$ has N items, consisting of M distinct items: $e_1, e_2, \cdots, e_M$. Let f_i be the real frequency of e_i.

For an item e_i, let

$$c_j = \begin{cases} 1 & f_i < f_j \\ \frac{f_j}{f_i+1} & otherwise \end{cases} \tag{3}$$

We use the dynamic programming algorithm to calculate the correct rate of an item e_i. Let $dp_{j,x}$ denote the probability that for the first j items, there are x items' frequencies larger than e_i at some time, we have

$$dp_{j,x} = dp_{j-1,x} * (1 - c_j) + dp_{j-1,x-1} * c_j \tag{4}$$

The correct rate of i^{th} item $\mathcal{P}_i$ can be derived as follows:

$$\mathcal{P}_i = \sum_{k=0}^{d-2} dp_{M,k} \tag{5}$$

C. The Error Bound

Given a data stream $\mathcal{S}$. Suppose that $\mathcal{S}$ has N items, consisting of M distinct items: $e_1, e_2, \cdots, e_M$. Let s_i and $\hat{s_i}$ be the real significance and estimated significance, respectively.

Theorem IV.2. *Given a small positive number ϵ, assume that the coefficients of frequency and persistency are respectively α and β, we have:*

$$Pr\{s_i - \hat{s_i} \geqslant \epsilon N\} \leqslant \frac{P_{small} * E(V) * (\alpha + \beta)}{\epsilon N} \tag{6}$$

where

$$P_{small} = \binom{i-1}{d-1} \left(\frac{1}{w}\right)^{d-1} \left(\frac{w-1}{w}\right)^{i-d} \tag{7}$$

and

$$E(V) = \frac{1}{w} \sum_{j=i+1}^{M} f_j \tag{8}$$

V. Experimental Results

In this section, we conduct experiments to show the performance of our algorithm with both two optimizations and other related algorithms. For convenience, we use *finding frequent/persistent/significant items* to denote finding top-k frequent/persisntent/significant items.

A. Metrics

The assessment is made by two metrics[2]: *ARE*(average relative error) and *Precision*. Suppose that the correct top-k significant set is ϕ, the reported set is ψ, consisting of $e_1, e_2, \ldots, e_k$, and the reported significance is respectively $\hat{s_1}, \hat{s_2}, \ldots, \hat{s_k}$. Precision is defined as $\frac{|\phi \cap \psi|}{k}$ and ARE is defined as $\frac{1}{k} * \sum_{i=1}^{k} \frac{|s_i - \hat{s_i}|}{s_i}$, where s_i is the real significance of e_i. Precision represents the ratio of the correct number of the reported top-k significant items to the number of real significant items, *i.e.*, k, and ARE represents the error rate of the estimated significance of the reported items. Both of them can objectively measure the errors of an algorithm.

[2]We ignore the metric AAE (average absolute error), because it will be significantly affected by the parameters (*e.g.* α, β).

B. Datasets

1) Social: This dataset comes from a real social network [30], which includes users' message and the sending time. We regard the username of the sender of a message as an item ID and the sending time of a message as the timestamp. This dataset contains 1.5M messages, and we divide it into 200 periods with a fixed time interval.

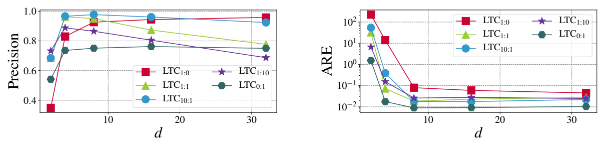
2) Network: This is a temporal network of interactions on the stack exchange web site. [31] Each item consists of three values u, v, t, which means user u answered user v's question at time t. We regard u as an item ID and t as the timestamp. This dataset contains 10M items, and we divide it into 1000 periods with a fixed time interval.

3) CAIDA: This dataset is from *CAIDA Anonymized Internet Trace 2016* [32], consisting of IP packets (*i.e.*, source IP address, destination IP address, source port, destination port, and protocol type). We regard the source IP address of a packet as an item ID and the index as the timestamp. This dataset contains 10M packets, and we divide it into 500 periods with a fixed time interval.

4) Synthetic: We generated 5 different datasets according to Zipfian [33] distribution by Web Polygraph [34], with different skewness [35] from 0.6 to 3.0. We regard the index as the timestamp. Each dataset contains 10M items, and we divide it into 1000 periods at a fixed time interval. We do not conduct experiments on finding persistent items and finding significant items on Synthetic datasets, because every item is distributed randomly, thus the persistent items are always the same as the frequent items on Synthetic datasets.

Implementation: We implement our algorithm and other related algorithms in C++. Here we use Bob Hash [36], [37], because it could generate many independent hash functions, which can be adapted to all algorithms. All the programs run on a server with duel 6-core CPUs(24 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62GB total system memory. The source codes of our LTC and other related algorithms are available on Github [29].

C. Effects of d



(a) Measuring precision. (b) Measuring ARE.

Fig. 10: Varying d.

In this experiment, we evaluate the effects of the parameter d, where d is the number of cells in each bucket. *When d increases, the performance first goes up and then goes down.* To make the experiments more comprehensive, we set five pairs of parameters: 1) $\alpha = 0, \beta = 1$, 2) $\alpha = 1, \beta = 0$, 3) $\alpha = 1, \beta = 1$, 4) $\alpha = 10, \beta = 1$, and 5) $\alpha = 1, \beta = 10$. We also set $k = 100$ and the memory size to 100KB. The number of cells is determined by the memory size. For convenience, we use LTC$_{0:1}$, LTC$_{1:0}$, LTC$_{1:1}$, LTC$_{10:1}$, and LTC$_{1:10}$ to denote

LTC on the corresponding parameters, where the first value is α and the second value is β. Due to space limitation, we just show the results on Network dataset. Figure 10(a) shows how precision changes when d varies from 2 to 32. Figure 10(b) shows how ARE changes when d varies from 2 to 32. We can observe that when $d = 8$, LTC has a perfect performance. Besides, taking the speed into consideration, we set $d = 8$ in the following experiments.

D. Comparisons on Finding Frequent Items

In this set of experiments, we set $\alpha = 1, \beta = 0$, which means the significance of an item is only related to its frequency. We compare the performance of LTC with SS, CSS, LC, and CM sketch plus a min-heap. For convenience, we use CM to denote CM sketch plus a min-heap. For all algorithms, the number of cells is determined by the memory size.

1) Measuring precision, varying memory size (Figure 11(a)-(c)). *LTC has the highest precision among all data structures no matter how the memory size changes.* In this experiment, we set $k = 100$, and vary the memory size from 10KB to 50KB. As shown in Figure 11(a), for the CAIDA dataset, when the memory size is 10KB, the precision of LC, SS, CSS, and CM is respectively 18%, 6%, 21%, 38%, while the one of LTC reaches 99%. As the memory size increases, the precision of LTC is always 100%, while the corresponding precision of LC, SS CSS, and CM is 63%, 38%, 67%, 98% at most. As shown in Figure 11(b), for the Network dataset, when the memory size is 10KB, the precision of LC, SS, CSS, and CM is respectively 5%, 2%, 6%, 31%, while the one of LTC reaches 88%. As shown in Figure 11(c), for the Social dataset, when the memory size is 10KB, the precision for LC, SS, CSS, CM and LTC is respectively 62%, 58%, 86%, 84%, and 98%. All these experiments show that LTC has much better precision than the other algorithms.

2) Measuring ARE, varying memory size (Figure 12(a)-(c)). *LTC has the lowest ARE among all data structures no matter how the memory size changes.* In this experiment, we set $k = 100$, and vary the memory size from 10KB to 50KB. As shown in Figure 12(a), for the CAIDA dataset, the ARE of LTC is between 1512 and 260869 times smaller than LC, between 26657 and 1748590 times smaller than SS, between 1199 and 220837 times smaller than CSS, and between 155 and 940484 times smaller than CM. As shown in Figure 12(b), for the Network dataset, the ARE of LTC is between 202 and 17960 times smaller than LC, between 11243 and 48184 times smaller than SS, between 183 and 12075 times smaller than CSS, and between 18 and 32087 times smaller than CM. As shown in Figure 12(c), for the Social dataset, the ARE of LTC is between 141 and 805 times smaller than LC, between 90 and 1465 times smaller than SS, between 20 and 26 times smaller than CSS, and between 16 and 17879 times smaller than CM.

3) Measuring precision, varying k (Figure 13(a)-(c)). *LTC has the highest precision among all data structures no matter how the standard of 'top' changes.* In this experiment, we set the memory size to 100KB, and vary k from 200 to 1000. As shown in Figure 13(a), for the CAIDA dataset, as k increases, the precision of LC, SS, CSS, and CM is changed from 78% to 42%, from 51% to 25%, from 81% to 44%, and from 98% to 70%, respectively, while the one of LTC is always larger than 99%. As shown in Figure 13(b), for the Network dataset, as k increases, the precision of LC, SS, CSS, and CM is changed from 69% to 36%, from 34% to 19%, from 71% to 36%, and from 95% to 65%, respectively, while the one of LTC is always larger than 95%. As shown in Figure 13(c), for the Social dataset, when $k = 1000$, the precision of LTC is 99%, while the one of the other algorithms is all less than 95%.

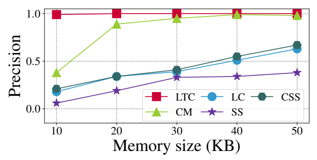
4) Measuring ARE, varying k (Figure 14(a)-(c)). *LTC has the lowest ARE among all data structures no matter how the standard of 'top' changes.* In this experiment, we set the memory size to 100KB, and vary k from 200 to 1000. As shown in Figure 14(a), for the CAIDA dataset, the ARE of LTC is between 386 and 41468 times smaller than LC, between 9075 and 304923 times smaller than SS, between 276 and 35769 times smaller than CSS, and between 132 and 110745 times smaller than CM. As shown in Figure 14(b), for the Network dataset, the ARE of LTC is between 448 and 3565 times smaller than LC, between 17869 and 41172 times smaller than SS, between 398 and 3459 times smaller than CSS, and between 2677 and 23786 times smaller than CM. As shown in Figure 14(c), for the Social dataset, the ARE of LTC is between 767 and 2535 times smaller than LC, between 245 and 647 times smaller than SS, between 61 and 76 times smaller than CSS, and between 22 and 1166 times smaller than CM.

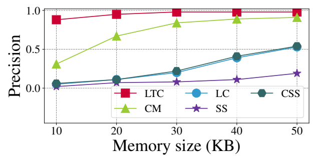
5) Measuring precision, varying skewness (Figure 11(d)). *LTC has the highest precision among all data structures no matter how the skewness of dataset changes.* In this experiment, we set the memory size to 100KB, $k = 1000$, and vary skewness from 0.6 to 3.0. As shown in Figure 11(d), the precision of LC, SS, CSS, and CM is respectively between 41% and 48%, between 34% and 49%, between 51% and 74%, and between 74% and 85%, while the one of LTC achieves 100% no matter on which skewness.

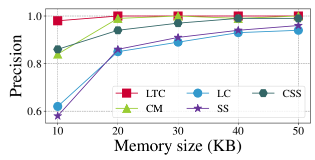
6) Measuring ARE, varying skewness (Figure 12(d)). *LTC has the lowest ARE among all data structures no matter how the skewness of dataset changes.* In this experiment, we set the memory size to 100KB, $k = 1000$, and vary skewness from 0.6 to 3.0. As shown in Figure 12(d), the ARE of LTC is between 803309 and 6034705 times smaller than LC, between 1647522 and 3709083 times smaller than SS, between 283537 and 592141 times smaller than CSS, and between 482892 and 1173926 times smaller than CM.

E. Comparisons on Finding Persistent Items

In this set of experiments, we set $\alpha = 0, \beta = 1$, which means the significance of an item is only related to its persistency. We compare the performance of LTC with PIE and a modified CM sketch plus a min-heap. The modified CM sketch needs to maintain a conventional BF to record whether an item has appeared in the current period. We allocate half of the memory to the conventional BF. For convenience, we use


(a) Varying memory size on CAIDA dataset.


(b) Varying memory size on Network dataset.


(c) Varying memory size on Social dataset.

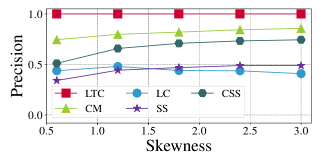
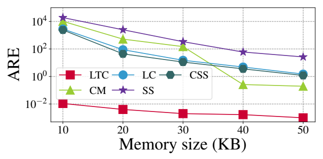
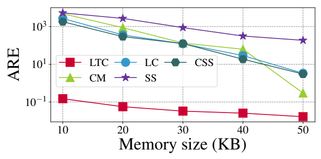
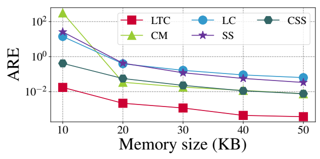

(d) Varying skewness on Synthetic datasets.

Fig. 11: Measuring precision on finding frequent items.


(a) Varying memory size on CAIDA dataset.


(b) Varying memory size on Network dataset.


(c) Varying memory size on Social dataset.

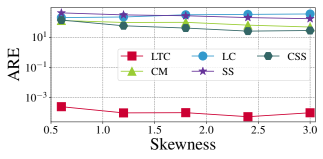
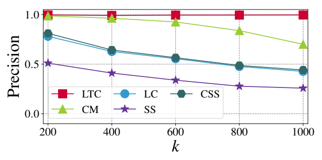
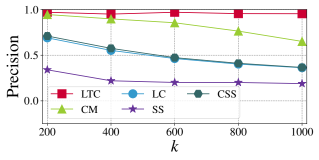

(d) Varying skewness on Synthetic datasets.

Fig. 12: Measuring ARE on finding frequent items.


(a) Varying k on CAIDA dataset.


(b) Varying k on Network dataset.

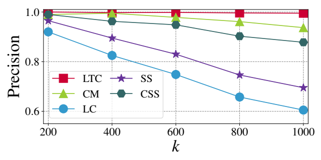
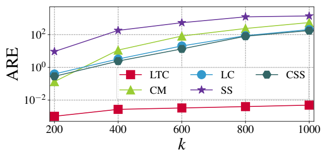
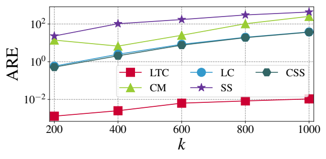

(c) Varying k on Social dataset.

Fig. 13: Measuring precision on finding frequent items.


(a) Varying k on CAIDA dataset.


(b) Varying k on Network dataset.

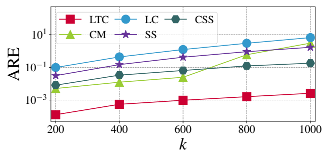
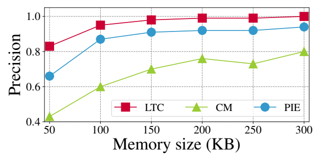
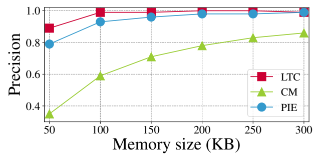
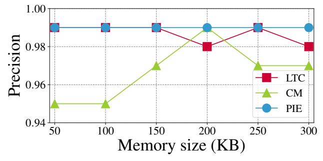

(c) Varying k on Social dataset.

Fig. 14: Measuring ARE on finding persistent items.


(a) Varying memory size on CAIDA dataset.


(b) Varying memory size on Network dataset.


(c) Varying memory size on Social dataset.

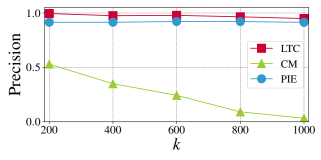
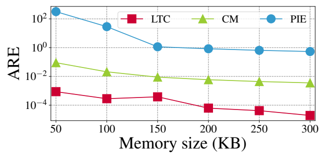
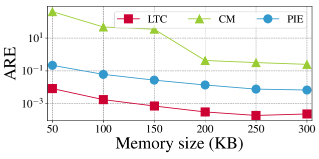
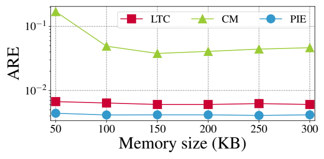

(d) Varying k on Network dataset.

Fig. 15: Measuring precision on finding persistent items.


(a) Varying memory size on CAIDA dataset.


(b) Varying memory size on Network dataset.


(c) Varying memory size on Social dataset.

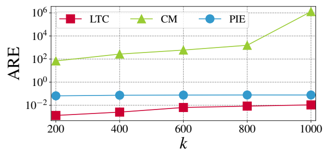
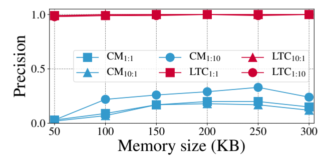
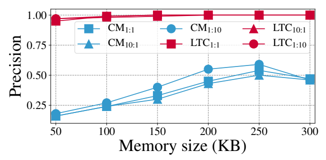

(d) Varying k on Network dataset.

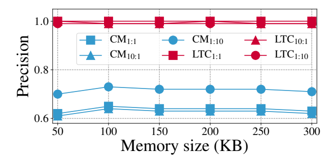
Fig. 16: Measuring ARE on finding persistent items.



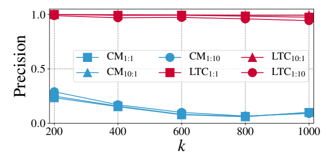
(a) Varying memory size on CAIDA dataset.



(b) Varying memory size on Network dataset.

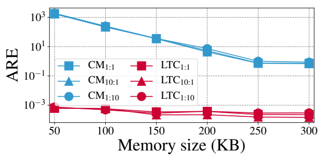


(c) Varying memory size on Social dataset.

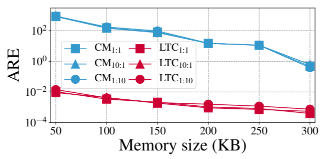


(d) Varying k on Network dataset.

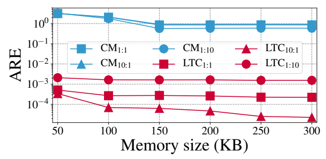
Fig. 17: Measuring precision on finding significant items.



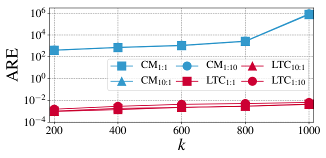
(a) Varying memory size on CAIDA dataset.



(b) Varying memory size on Network dataset.

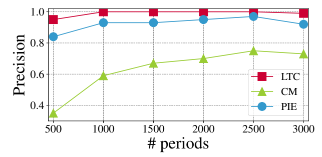


(c) Varying memory size on Social dataset.

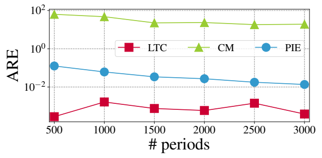


(d) Varying k on Network dataset.

Fig. 18: Measuring ARE on finding significant items.

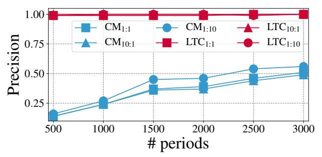


(a) Measuring precision on Network dataset.

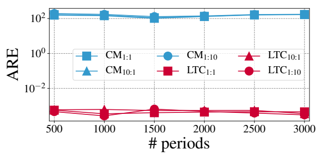


(b) Measuring ARE on Network dataset.

Fig. 19: Varying # periods on finding persistent items.



(a) Measuring precision on Network dataset.



(b) Measuring ARE on Network dataset.

Fig. 20: Varying # periods on finding significant items.

CM to denote this algorithm. For PIE, it needs to maintain a structure for each period, thus it cannot decode any item when the memory is tight. Therefore, we allocate T times memory size of other algorithms to it, where T is the number of periods, to make its performance comparable. In other words, when we allocate hKB to other algorithms, we will allocate 200*hKB, 1000*hKB, and 500*hKB to PIE in Social dataset, Network dataset, and CAIDA dataset respectively. For all algorithms, the number of cells is determined by the memory size.

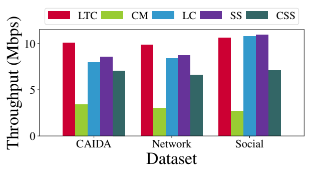
1) Measuring precision, varying memory size (Figure 15(a)-(c)). *LTC has the highest precision for all memory settings.* In this experiment, we set $k = 100$, and vary the memory size from 50KB to 300KB for LTC and CM. As shown in Figure 15(a), for the CAIDA dataset, the precision of CM, PIE, and LTC is changed from 43% to 80%, from 66% to 94%, and from 83% to 100%, respectively. As shown in Figure 15(b), for the Network dataset, the precision of CM, PIE and LTC is changed from 35% to 86%, from 79% to 99%, and from 89% to 99%, respectively. As shown in Figure 15(c), for the Social dataset, all the algorithms have a high precision. The reason for the perfect performance of PIE is that the memory size is T times that of the other two algorithms. As we can see, the performance of LTC is much better than the other algorithms.

2) Measuring ARE, varying memory size (Figure 16(a)-(c)). *LTC has the lowest ARE for all memory settings.* In this experiment, we set $k = 100$, and vary the memory size from 50KB to 300KB for LTC and CM. As shown in Figure 16(a), for the CAIDA dataset, the ARE of LTC is between 3020 and 370384 times smaller than CM and between 23 and 184 times smaller than PIE. As shown in Figure 16(b), for the Network dataset, the ARE of LTC is between 106 and 49224 times smaller than CM and between 26 and 44 times smaller than PIE. As shown in Figure 16(c), for the Social dataset, the ARE of LTC is between 6 and 25 times smaller than CM and approximately the same as PIE.

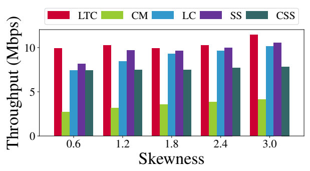
3) Measuring precision, varying k Figure 15(d). *LTC has the highest precision for all k settings.* In this experiment, we set the memory size to 100KB for LTC and CM, and vary k from 200 to 1000. Due to space limitation, we just show the results on Network dataset. As shown in Figure 15(d), when $k = 200$, the precision of CM and PIE is respectively 53% and 91.5%, while the one of LTC reaches 99%. Furthermore, as k increases, the precision of LTC is always larger than 95%.

4) Measuring ARE, varying k (Figure 16(d)). *LTC has the lowest ARE for all k settings.* In this experiment, we set the memory size to 100KB for LTC and CM, and vary k from 200 to 1000. Due to space limitation, we just show the results on Network dataset. As shown in Figure 16(d), the ARE of LTC is between 52942 and 10^8 times smaller than CM and between 7 and 50 times smaller than PIE.

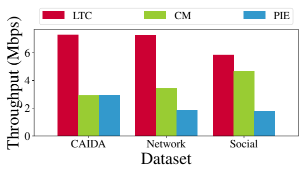
5) Measuring precision, varying the number of periods (Figure 19(a)). *LTC has the highest precision for all settings of the number of periods.* In this experiment, we set the memory size to 100KB for LTC and CM, $k = 100$, and vary T from 500 to 3000, where T is the number of periods. As the experimental method mentioned above, when T is



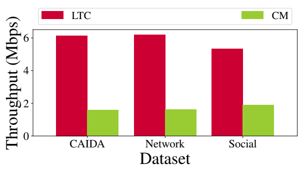
(a) Finding frequent items.



(b) Finding frequent items.



(c) Finding persistent items.



(d) Finding significant items.

Fig. 21: Measuring throughput, varying dataset and skewness.

changed, the memory size of PIE changes but that of the other algorithms does not change. Due to space limitation, we just show the results on Network dataset. As shown in Figure 19(a), when $T = 500$, the precision of CM and PIE is respectively 35% and 84%, while the one of LTC reaches 95%. Furthermore, when T is larger than 1000, the precision of LTC is always larger than 99%.

6) Measuring ARE, varying the number of periods (Figure 19(b)). *LTC has the lowest ARE for all settings of the number of periods.* The configuration of this experiment is the same as the previous experiment. We just show the results on Network dataset. As shown in Figure 19(b), the ARE of LTC is between 12834 and 225914 times smaller than CM and between 12 and 449 times smaller than PIE.

F. Comparisons on Finding Significant Items

There is no prior work on finding significant items, thus we combine the best algorithm on finding frequent items with the best algorithm on finding persistent items, to find significant items. Notice that CM sketch plus a min-heap is the best algorithm except LTC on both finding frequent items and finding persistent items, therefore we modify it to find significant items. To make the experiments more comprehensive, we set three pairs of parameters: 1) $\alpha = 1, \beta = 10$, 2) $\alpha = 1, \beta = 1$ and 3) $\alpha = 10, \beta = 1$. For convenience, we use $\text{CM}_{1:10}$, $\text{CM}_{1:1}$, and $\text{CM}_{10:1}$ to denote the performance of CM sketch plus a min-heap on the corresponding parameters, where the first value is α and the second value is β. Similarly, we use $\text{LTC}_{1:10}$, $\text{LTC}_{1:1}$, and $\text{LTC}_{10:1}$ to denote the performance of LTC on the corresponding parameters.

1) Measuring precision, varying memory size (Figure 17(a)-(c)). *The precision of LTC is higher than that of CM.* In this experiment, we set $k = 100$, and vary the memory size from 50KB to 300KB. As shown in Figure 17(a), for the CAIDA dataset, when the memory size is 50KB, the precision of LTC reaches 99% no matter on which pair of parameters. And the precision of CM is respectively 15%, 12%, and 24% on each pair of parameters even if the memory size is 300KB. As shown in Figure 17(b), for the Network dataset, when the memory size is 50KB, the precision of LTC reaches 99% no matter on which pair of parameters, while the one of CM is respectively 16%, 16%, and 18% on each pair of parameters. As shown in Figure 17(c), for the Social dataset, the precision of CM is respectively 62%, 63%, and 71% on each pair of parameters. And the precision of LTC is always larger than 99% no matter on which pair of parameters.

2) Measuring ARE, varying memory size (Figure 18(a)-(c)). *The ARE of LTC is lower than that of CM.* In this experiment, we set $k = 100$, and vary the memory size from 50KB to 300KB. As shown in Figure 18(a), for the CAIDA dataset, the ARE of LTC is respectively between 2913 and 2517407 times, between 1188 and 81589 times, and between 4632 and 2738419 times smaller than CM on each pair of parameters. As shown in Figure 18(b), for the Network dataset, the ARE of LTC is respectively between 513 and 62873 times, between 3180 and 2795956 times, and between 1141 and 91060 times smaller than CM on each pair of parameters. As shown in Figure 18(c), for the Social dataset, the ARE of LTC is respectively between 352 and 1557 times, between 3116 and 7698 times, and between 9119 and 40361 times smaller than CM on each pair of parameters.

3) Measuring precision, varying k (Figure 17(d)). *The precision of LTC is higher than that of CM.* In this experiment, we set the memory size to 100KB, and vary k from 200 to 1000. Due to space limitation, we just show the results on Network dataset. As shown in Figure 17(d), as k increases, the precision of CM is changed from 25% to 10%, from 23% to 10%, and from 29% to 9% on each pair of parameters, respectively, while the one of LTC is always larger than 94% on any k or pair of parameters.

4) Measuring ARE, varying k (Figure 18(d)). *The ARE of LTC is lower than that of CM.* In this experiment, we set the memory size to 100KB, and vary k from 200 to 1000. Due to space limitation, we just show the results on Network dataset. As shown in Figure 18(d), the ARE of LTC is respectively between 262056 and 10^8 times, between 383976 and 10^8 times, and between 403629 and 10^8 times smaller than CM on each pair of parameters.

5) Measuring precision, varying the number of periods (Figure 20(a)). *The precision of LTC is higher than that of CM.* In this experiment, we set the memory size to 100KB, $k = 100$, and vary T from 500 to 3000, where T is the number of periods. Due to space limitation, we just show the results on Network dataset. As shown in Figure 20(a), as T increases, the precision of CM is changed from 14% to 51%, from 14%, to 49%, and from 16% to 56% on each pair of parameters, respectively, while the one of LTC is always larger than 99% on any T or pair of parameters.

6) Measuring ARE, varying the number of periods (Figure 20(b)). *The ARE of LTC is lower than that of CM.* In this experiment, we set the memory size to 100KB, $k = 100$, and vary T from 500 to 3000, where T is the number of periods. Due to space limitation, we just show the results on Network dataset. As shown in Figure 20(b), the ARE of LTC is respectively between 196922 and 645944 times, between 273032 and 435427 times, and between 188973 and 464208 times smaller than CM on each pair of parameters.

G. Experiments on Throughput

In this section, we conduct experiments to compare the throughput of all the above algorithms.

1) Finding frequent items (Figure 21(a) and 21(b)). We set the memory size to 100KB and $k = 100$. As shown in Figure 21(a) and Figure 21(b), the results show that LTC not only is more accurate than the other algorithms, but also achieves higher throughput. The reason for the slow speed of CM sketch is it needs to maintain a min-heap.

2) Finding persistent items (Figure 21(c)). We set the memory size to 100KB and $k = 100$. As shown in Figure 21(c), the throughput of LTC is still much higher than the other algorithms.

3) Finding significant items (Figure 21(d)). We set the memory size to 100KB, $k = 100$, and $\alpha = 1, \beta = 1$. As shown in Figure 21(d), LTC still achieves high throughput.

VI. Conclusion

In this paper, we abstract a problem named finding top-k significant items, which is encountered in many applications but not fully studied before. To accurately find top-k significant items with small memory size, we propose a new algorithm, namely LTC. It has two key techniques, Long-tail Replacement and a modified CLOCK algorithm. We derive the theoretical correct rate and error bound for LTC. Extensive experiments on 3 real datasets and synthetic datasets show that our LTC achieves high accuracy and high speed at the same time when using small size of memory. The source codes of LTC and other related algorithms are available at Github [29].

References

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

[2] Saar Cohen and Yossi Matias. Spectral bloom filters. In *ACM SIGMOD International Conference on Management of Data*, 2003.

[3] Tong Yang, Alex X. Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. A shifting bloom filter framework for set queries. *Proceedings of the Vldb Endowment*, 9(5), 2016.

[4] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM transactions on networking*, 10(5), 2002.

[5] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Automata, languages and programming*, 2002.

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

[7] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3), 2003.

[8] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proc. SIGMOD*, 2016.

[9] 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), 2017.

[10] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM, 2009.

[11] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*. ACM, 2015.

[12] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. Technical report, 2009.

[13] Katsiaryna Mirylenka, Graham Cormode, Themis Palpanas, and Divesh Srivastava. Conditional heavy hitters: detecting interesting correlations in data streams. *The VLDB Journal*, 24(3), 2015.

[14] Joong Hyuk Chang and Won Suk Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003.

[15] Yin-Ling Cheung and Ada Wai-Chee Fu. Mining frequent itemsets without support threshold: with and without item constraints. *IEEE Transactions on Knowledge and Data Engineering*, 16(9), 2004.

[16] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. Springer ICDT*, 2005.

[17] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *Proc. IEEE INFOCOM*, 2016.

[18] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. VLDB*, 2002.

[19] Haipeng Dai, Muhammad Shahzad, Alex X. Liu, and Yuankun Zhong. Finding persistent items in data streams. *Proceedings of the Vldb Endowment*, 10(4):289–300, 2016.

[20] Sneha Aman Singh and Srikanta Tirthapura. Monitoring persistent items in the union of distributed streams. *Journal of Parallel and Distributed Computing*, 74(11):3115–3127, 2014.

[21] Anja Feldmann and Ward Whitt. Fitting mixtures of exponentials to long-tail distributions to analyze network performance models . *Performance Evaluation*, 31(34):1096–1104 vol.3, 1997.

[22] Joseph Abate, Gagan L. Choudhury, and Ward Whitt. Waiting-time tail probabilities in queues with long-tail service-time distributions. *Queueing Systems*, 16(3-4):311–338, 1994.

[23] Gagan L. Choudhury and Ward Whitt. Long-tail buffer-content distributions in broadband networks. *Performance Evaluation*, 30(3):177–190, 1997.

[24] Allen B. Downey. Evidence for long-tailed distributions in the internet. In *ACM SIGCOMM Internet Measurement Workshop*, pages 229–241, 2001.

[25] N. G. Duffield and W. Whitt. Network design and control using on-off and multi-level source traffic models with long-tailed distributions. *At T Labs*, pages 421–445, 1997.

[26] Gunner Danneels. System for synchronizing data stream transferred from server to client by initializing clock when first packet is received and comparing packet time information with clock, 1997.

[27] Nicholas A. J Millington. System and method for synchronizing operations among a plurality of independently clocked digital data processing devices, 2015.

[28] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, 2013.

[29] The source codes of our and other related algorithms. https://github.com/Finding-Significant-Items/Finding-Significant-Items.

[30] The Social dataset Internet Traces. http://open.weibo.com/wiki/2/friendships/friends/.

[31] The Network dataset Internet Traces. http://snap.stanford.edu/data/.

[32] The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.

[33] David MW Powers. Applications and explanations of zipf's law. In *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*. Association for Computational Linguistics, 1998.

[34] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 2004.

[35] Graham Cormode, Balachander Krishnamurthy, and Walter Willinger. A manifesto for modeling and measurement in social media. *First Monday*, 15(9), 2010.

[36] The source code of Bob Hash. http://burtleburtle.net/bob/hash/evahash.html.

[37] Christian Henke, Carsten Schmoll, and Tanja Zseby. Empirical evaluation of hash functions for multipoint measurements. *SIGCOMM Comput. Commun. Rev.*, 38(3), July 2008.

VII. Appendix

A. Proof of no Overestimation

Theorem VII.1. *For an item e_i, let s_i and $\hat{s_i}$ be the real significance and estimated significance, respectively. We have*

$$\hat{s_i} \leqslant s_i \tag{9}$$

Proof. Apparently, the significance of an item can be decomposed into two metrics ($e.g.$ frequency, persistency). We will respectively prove that there is no overestimated error for the above two metrics below.

Firstly, we prove the inequality $\hat{f_e} \leqslant f_e$ holds at any point of time, where f_e is the real frequency and $\hat{f_e}$ is the estimated frequency.

Initially, all cells in the lossy table are empty, the inequality holds. When an item e arrives, it is first mapped to a bucket $A[h(e)]$, and then there are 3 cases as follows:

Case 1: e is not found in the $A[h(e)]$ but it can be inserted into the lossy table. The estimated frequency of this item is equal to 1, and the real frequency is obviously no less than 1, the inequality holds.

Case 2: e is not found in the $A[h(e)]$ and it cannot be inserted into the lossy table. The estimated frequency of this item is equal to 0, the inequality holds.

Case 3: e is found in the $A[h(e)]$. In this situation, the estimated frequency and the real frequency of this item are both incremented by 1, the inequality holds.

Secondly, we prove the inequality $\hat{p_e} \leqslant p_e$ holds at any point of time, where p_e is the real persistency and $\hat{p_e}$ is the estimated persistency.

For convenience, let p_e^* be the real persistency of e other than the current period. Specifically, if e appears in the period t, p_e^* will be incremented at time of period $t+1$. Apparently, $p_e^* \leqslant p_e$ holds. We will prove the inequality $\hat{p_e} \leqslant p_e^*$ holds below. Same as the proof of the inequality about frequency, at first all cells in the lossy table are empty, thus the inequality holds. For each item e, if it appeared in the previous period, $\hat{p_e}$ is incremented by 1 at most and p_e^* is exactly incremented by 1, the inequality holds. Otherwise, $\hat{p_e^*}$ and p_e are both not changed. Therefore, the inequality $\hat{p_e} \leqslant p_e^* \leqslant p_e$ holds at any time of period.

In summary, the theorem holds. □

B. The Bound of Correct Rate

We will present how to derive the correct rate about significance.

Lemma VII.1. *The reported significance of an item is correct if it satisfies the following conditions: 1) When the item arrives for the first time, the bucket is not full, and 2) Each time the corresponding cell is not the smallest cell.*

Proof. When an item e arrives for the first time, if there is an empty cell in the bucket mapped by e, the estimated significance of e, which is 1, is equal to the real significance. When another item arrives later, if it is already in the bucket, the estimated significance of e is not influenced. Otherwise, the Significance Decrementing operation will be performed on the smallest cell. Since e is not in the smallest cell, the estimated significance of e is not influenced, either. In summary, the reported significance of e is correct. □

Assume that there is a stream $\mathcal{S}$ which has M distinct items: $e_1, e_2, \cdots, e_M$. Let f_i be the frequency of e_i in the stream and N be the total number of items. According to the nature of Zipfian dataset, we have

$$f_i = \frac{N}{i^{\gamma}\zeta(\gamma)} \tag{10}$$

where $\zeta(\gamma) = \sum_{i=1}^{M} \frac{1}{i^{\gamma}}$ and γ is the skewness of the stream.

For an item e, let f denote the number of appearances of e. Let p_i denote the probability that e_i is mapped to the same bucket as e and the number of appearances of e_i is larger than that of e at some point of time. If $f_i > f$, it is obvious that $p_i = \frac{1}{w}$, otherwise $p_i = \frac{1}{w} * \frac{f_i}{f+1}$, where w is the number of buckets. For convenience, we use *useful* to describe that an element e_i satisfying the above condition.

According to VII.1, if there is less than $d-1$ useful items at any point of time, the reported significance of e is sure to be correct. Let $\mathcal{P}_i$ denote the correct rate of i^{th} item.

We use the dynamic programming algorithm to calculate $\mathcal{P}_i$. Let $dp_{j,x}$ denote the probability that for the first j items, there are x useful items. We have

$$dp_{j,x} = dp_{j-1,x} * (1-p_j) + dp_{j-1,x-1} * p_j \tag{11}$$

and

$$\mathcal{P}_i = \sum_{k=0}^{d-2} dp_{N,k} \tag{12}$$

C. The Error Bound

We will present how to derive the error bound about significance.

Theorem VII.2. *Given a small positive number ϵ, for each item e_i, let s_i and $\hat{s_i}$ be the real significance and estimated significance, respectively. Assume that the coefficients of frequency and persistency are respectively α and β, we have:*

$$Pr\{s_i - \hat{s_i} \geqslant \epsilon N\} \leqslant \frac{P_{small} * E(V) * (\alpha+\beta)}{\epsilon N} \tag{13}$$

where

$$P_{small} = \binom{i-1}{d-1}\left(\frac{1}{w}\right)^{d-1}\left(\frac{w-1}{w}\right)^{i-d} \tag{14}$$

and

$$E(V) = \frac{1}{w}\sum_{j=i+1}^{M} f_j \tag{15}$$

Proof. For an item e_i, if there is no Significance Decrementing operation performed on it, the estimated significance is exactly equal to the real significance. Let X_i be the number of times the operation is performed on e_i. We have:

$$\hat{s_i} = s_i - X_i * (\alpha + \beta) \tag{16}$$

e_i will be performed that operation if and only if it is the smallest cell in the bucket. Let P_{small} be the probability of it, we have

$$P_{small} = \binom{i-1}{d-1} \left(\frac{1}{w}\right)^{d-1} \left(\frac{w-1}{w}\right)^{i-d} \tag{17}$$

where w is the number of buckets and d is the number of cells in each bucket.

Let V be the number of items which could perform Significance Decrementing operation on e_i. In other words, these items need to be mapped to the same bucket as e_i and less frequent than e_i. The expectation of V is

$$E(V) = \frac{1}{w} \sum_{j=i+1}^{M} f_j \tag{18}$$

Then we drive the formula for $E(X_i)$:

$$E(X_i) = P_{small} * E(V) \tag{19}$$

According to the equation 16, we can get the equality for $E(\hat{s_i})$:

$$E(\hat{s_i}) = s_i - P_{small} * E(V) * (\alpha + \beta) \tag{20}$$

Based on Markov inequality, we have:

$$\begin{aligned} Pr\{s_i - \hat{s_i} \geqslant \epsilon N\} &\leqslant \frac{E(s_i - \hat{s_i})}{\epsilon N} \\ &\leqslant \frac{P_{small} * E(V) * (\alpha + \beta)}{\epsilon N} \end{aligned} \tag{21}$$

Therefore, the theorem holds. $\square$