Finding Significant Items in Data Streams

Tong Yang, Haowei Zhang, Dongsheng Yang, Yucheng Huang, Xiaoming Li
Department of Computer Science, Peking University, China

***Abstract*—Finding top-k frequent items has been a hot issue in databases. 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. It includes two key techniques: Long-tail Replacement and a modified CLOCK algorithm. We theoretically prove there is no overestimation error and derive the correct rate and error bound. We conduct extensive experiments on three real datasets. Our 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 the 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 get 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 improve the performance of the applications. Second, such queries are often intermediate results, serving for a 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 does 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, people only care about the most significant part of the dataset. In the typical use cases, it is a classic and important topic to find frequent items [10]–[15]. Other than that, some works focus on finding persistent items [16], [17]. However, it is often insufficient to merely involve one metric for significant items. In other words, people actually care about significant items which are both frequent and persistent. This has not been touched by existing works. In this paper, we will focus on how to accurately and quickly find significant items. We first 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. We show three use cases as follows.

Use case 1: DDoS attacks detection [18]. Accurate and timely detection of DDoS attacks has been a hot issue. The DDoS victim receives many packets from attackers, which can be detected by finding frequent items. However, only some frequent items attack traffic. It would be time consuming to carefully check all frequent items one by one. Actually, the attack traffic is often not only frequent but also persistent. Therefore, finding significant items can somehow separate attack traffic from normal traffic more accurately.

Use Case 2: Website evaluation. Evaluating the websites by popularity should be accurate and fast, and the rank should be updated in real time. There are two key metrics of popularity: frequency and persistency. Frequency refers to the amount of the user's accesses to the website, while persistency indicates whether this website is popular all the time. Therefore, both of the two metrics should be considered in ranking the popularity/significance of a website.

Use Case 3: Network congestion. Network congestion happens every second in data centers [19]–[21]. One effective solution for solving congestion is to change the forwarding path of some flows[1]. As there could be millions of flows in every

[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.

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 too many entries of the forwarding table, it is highly desirable 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. Therefore, changing the significant flows is a better solution. Indeed, this could cause a congestion at another place. That is why network congestion has been a challenging issue for many years. If persistent flow all over the data center can be efficiently identified, we can make a global solution to schedule the persistent flow, mitigating congestion.

B. Prior Art and Their Limitations

Existing solutions focus on only one dimension, such as finding top-k frequent items [10]–[15], or finding top-k persistent items [16], [17]. However, in practice, finding both frequent and persistent items is more practical and important. 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. Obviously, the time and space overhead is the sum of the overhead of two 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 with limited memory.

C. Our Solution

To find significant (frequent, persistent, or both) items, 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. This 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 $f_{min} + 1$. It will cause large overestimation error. 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* [22]–[26]. 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 expelled 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 overestimation of the initial value of e . However, it would be decremented and expelled soon from the k cells.

Based on the long-tail distribution, there is no other appropriate initial value, More details about Long-tail Replacement are provided in Section III-D.

Second, we modify the CLOCK algorithm to record persistencies. The most challenging issue is to increment the persistency *by one* for any item that appears more than once in one period. Our idea is to leverage the spirit of the well known CLOCK algorithm [27]–[29]. We propose two modified CLOCK algorithms: a basic version (see Section III-B) with overestimation error and an optimized version (see Section III-C) eliminating overestimation error.

Key Contributions:

In this paper, we make the following key contributions.

1) We abstract a problem named finding top-k significant items, which is encountered in many applications but not studied before. Furthermore, we propose a new algorithm, namely LTC, to accurately find top-k significant items.

2) We prove there is no overestimation error and further derive the theoretical bound of correct rate and error for LTC.

3) We conduct extensive experiments on 3 real datasets, and the results show that our LTC achieves high accuracy and high speed at the same time when using small size of memory.

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) [13], Lossy Counting(LC) [15], CSS [14], etc. These algorithms are similar to each other. Take Space-Saving as an example, it maintains several cells, and 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 it matches the j^{th} cell, SS increments f_j by 1. Otherwise it finds 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.

Sketch-based: Sketch-based algorithms include the Count sketch [5], the CM sketch (CM) [6], etc. They are similar to each other. Take CM 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. Cristian and George proposed the CU sketch (CU) [7]. It improves CM by incrementing only the minimum value(s) among the mapped cells by 1 instead of incrementing all the values of mapped cells when an item arrives. 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 [17] and PIE [16]. 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 [30][2] 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.

Besides, we can modify sketch-based algorithms to be adapted to find top-k persistent items. The thorniest problem is that some items might appear more than once in one period, which means we cannot update persistency directly. To deal with this problem, we maintain a standard Bloom filter(BF) [1] to record whether it has appeared in the current period. We also need to maintain a min-heap to assist in finding top-k persistent items.

III. The LTC Algorithm

In this section, we first describe the data structure and operations of LTC in detail. 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.

A. Data Structure

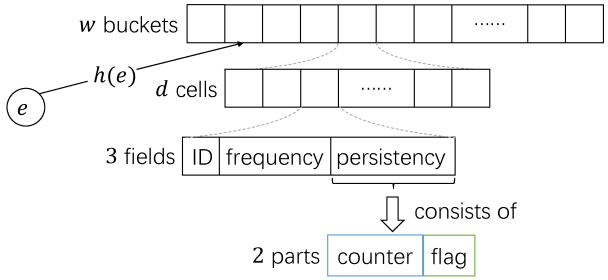


Fig. 1: The data structure of LTC.

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 stores 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 the cells in one bucket, we call the cell with the smallest significance `the smallest cell`.

B. Operations

We call a cell `empty` if and only if the ID field is NULL and the significance of this cell equals to 0. We initiate LTC by setting all the cells in the lossy table to empty and all the 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 the 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 empty. However, as frequency is always greater than persistency, the recorded persistency might be negative after several significance decrementing operations are performed on the cell. We can avoid such cases by keeping 0 if it is already 0.

Example: As shown in Figure 2, we set $w = 3, d = 3, \alpha = 1, \beta = 0$, where w and d are the number of buckets and the number of cells in each bucket, respectively. 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 a persistency of 3. The color of

[2]Raptor codes are the first known class of fountain codes with linear time encoding and decoding.

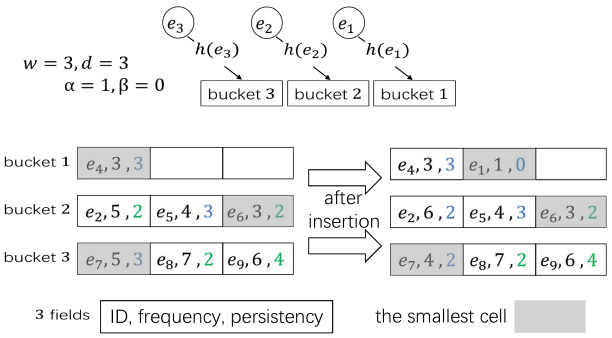


Fig. 2: An example for inserting an item.

3 is blue means that the flag is true. 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.

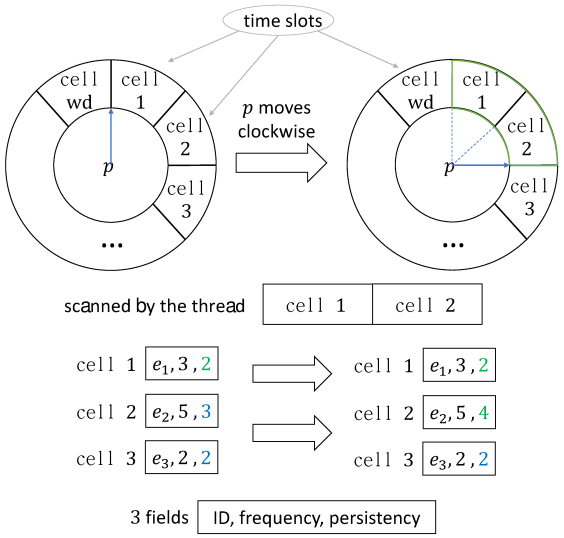


Fig. 3: The CLOCK and Persistency Incrementing.

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 [27]–[29]. Our key idea is as follows. We use a pointer to scan the lossy table, and check the flag. By the ending of a period, 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. 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 there are m cells/time slots in total, suppose the arriving speed of every item is the same, the step size 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. After each item arrives, we use a thread to scan m/n cells, which corresponds to those m/n time slots. LTC scans each cell as follows:

Case 1: The flag of this cell is false. We do nothing.

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

Our method can be easily extended when the period is defined by time. In practice, the arriving speed of items could vary a lot. To adapt to the arriving speed, we can dynamically adjust the scanning speed by modifying the step size of pointer p. Suppose our lossy table has m cells/time slots in total. Each period has t seconds. Within every period (t seconds), the lossy table is exactly scanned one time, from the start to the end. Given a current item arriving at x^{th} second and its previous item at y^{th} second, we can just let the pointer p pass $(x-y)/t * m$ time slots. In this way, although the step size of p for every item is different, p still passes all cells exactly once in each period.

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 wd cells (w buckets and d cells in each bucket) in total, the pointer p moves clockwise and passes both cell 1 and cell 2. LTC uses a thread to scan these two cells. For cell 1, the flag is false, so 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 achieves 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 effectively 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 this design 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 scanned exactly once in a period. Thus, the persistency will be incremented by at most 1 in one period even if the corresponding item appeared many times, which exactly matches the definition of persistency.

4) Limitations:

First, the period of each cell has a deviation from the real period, and we propose an optimization namely Deviation Eliminator, which is detailed 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.

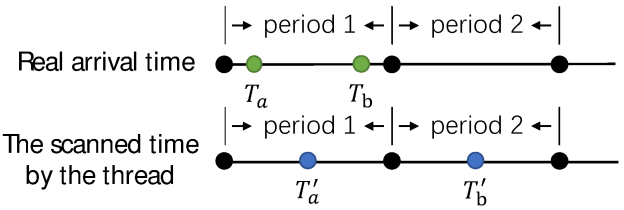


Fig. 4: An example of the deviation.

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 scanned by a thread 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, thus 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.

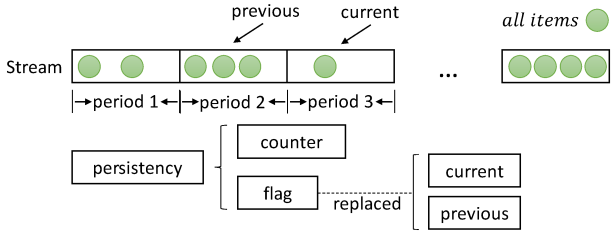


Fig. 5: Deviation Eliminator.

We use an example to explain our solution. As shown in Figure 5, we replace the flag of the persistency field with two flags: a `current` flag and a `previous` flag. Suppose the current period is period 3, we call the period 2 the previous period. The `current` flag indicates whether the recorded item has appeared in the current period (period 3), and the `previous` flag indicates whether the recorded item has appeared in the previous period (period 2).

The operations for the frequency field are the same as the basic version. Here we show the operations for the persistency field when an item e arrives. At first, e is mapped to one bucket. Same as the basic version, there are three cases.

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

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

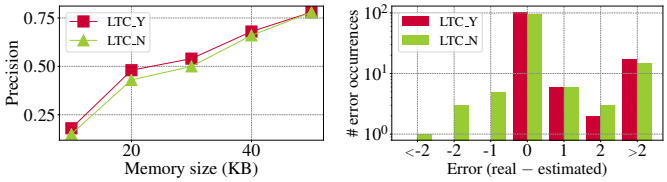
Case 3: e is not found and there is no empty cell in $A[h(e)]$. The strategy is the same as that of the basic version.

The Persistency Incrementing operation is determined by the value of the `previous` flag. Specifically, suppose the pointer p scans z cells after an item arrives. We use a thread to scan these z cells, and check their `previous` flags: for each flag, if it is true, the corresponding persistency is incremented by 1 and the `previous` flag is reset to false; otherwise, nothing is changed. The Significance Decrementing operation is the same as that of the basic version.

Flag refreshment: At the end of each period, we immediately scan all cells: for each cell, we first move the value of the `current` flag to the `previous` flag and then reset the `current` flag to false.

In this way, we achieve that the recorded information exactly corresponds to a real period, and thus eliminate the deviation.

Refreshment elimination: Fortunately, the above flag refreshment operation can be eliminated by using the following method. We split all the periods into two parts: odd-numbered periods and even-numbered periods, and replace the current and previous flags with two new flags: an `even` flag and an `odd` flag. If the current period is an even-numbered period,



(a) Precision vs. memory size. (b) # error occurrences vs. error.

Fig. 6: LTC with the deviation eliminator vs. the basic version.

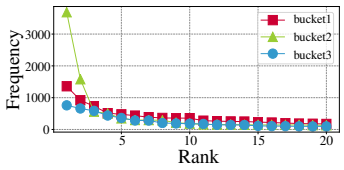


Fig. 7: Top-20 frequencies distribution of three buckets.

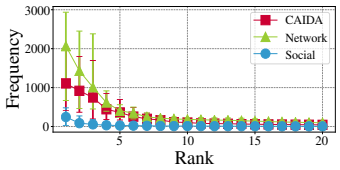


Fig. 8: Top-20 frequencies distribution on three datasets.

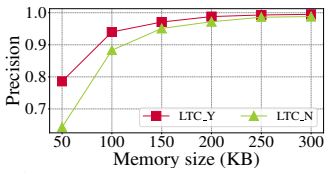


Fig. 9: Precision vs. memory size.

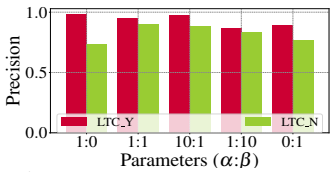


Fig. 10: Precision vs. parameters.

the `even` flag denotes the `current` flag, and the `odd` flag denotes the `previous` flag, or vice versa. After each period ends, the current flag is either changed from even to odd or from odd to even. In this way, the above flag refreshment operation is finished automatically.

We conduct experiments to compare the optimized version (with the deviation eliminator) with the basic version. Here we focus on finding persistent items, which means $\alpha = 0, \beta = 1$. The dataset we use is Network dataset, which is detailed in Section V-B. As shown in Figure 6(a) and Figure 6(b), *precision* means the ratio of the correct number of the reported top-k persistent items to the number of real top-k persistent items, *i.e.*, k, LTC_Y means the optimized version, and LTC_N means the basic version.

For the first experiment (Figure 6(a)), we set $k = 1000$, $d = 8$, and vary the memory size from 10KB to 50KB. The reason why we use the small memory size is to compare the performance of them more intuitively. The results show that the precision of LTC_Y is slightly larger than that of LTC_N.

The goal of the second experiment (Figure 6(b)) is to show that LTC_Y has no overestimation error, while LTC_N has errors of both overestimation and underestimation. In this experiment, we set the total number of cells to 128, $d = 8$, and report all the differences between the real persistencies and estimated persistencies recorded in the cells. The results fully verify that the eliminator deviation can totally eliminate the overestimation error.

D. Optimization II: Long-tail Replacement

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

Assumption: In practice, the real datasets often follow long-tail distribution. Furthermore, we assume that the frequencies of items that fall into the same bucket still follow long-tail distribution.

This Assumption is consistent with the following experiments. We set the number of buckets to 800 and use different datasets (detailed in Section V-B) to verify our assumption. Figure 7 shows the frequencies of top-20 frequent items in three arbitrary buckets on Network dataset, and Figure 8 shows the frequencies of top-20 frequent items on three datasets. From the figures above, we can clearly see that frequencies follow long-tail distribution no matter what dataset or specific bucket we choose.

Based on the above observations, we propose a novel optimization, namely Long-tail Replacement, to set the initial value (frequency or persistency) of the newly inserted item. For convenience we use *value* to denote persistency and frequency, which means this optimization can be adapted to both persistency and frequency field in the lossy table. Specifically, we set the initial *value* of the newly inserted item to the second smallest *value* minus 1.

Given a full bucket, when a new item e arrives, we decrement the *value* of the smallest cell by 1. 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 in 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 accurately 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 expelled soon, since it seldom appears later.

When the *value* means significance, we decrement item with the smallest significance: decrementing both frequency and persistency by 1. Same as the Significance Decrementing operation, LTC does not decrement the persistency when it is 0. When a new item is inserted into the bucket, we need to find the second smallest frequency and the 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. The dataset we use is Network dataset. As shown in Figure 9 and Figure 10, *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 9), 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 10), we set the memory size to 50KB, $d = 8$, $k = 1000$, and vary the parameters. The results show that the precision of LTC_Y is still always larger than that of LTC_N.

Shortcoming: Such an optimization is based on the assumption that the dataset follows long-tail distribution. Long-tail Replacement may not work well for other distribution, such as the uniform distribution, every item has the similar frequencies. Therefore, users should check whether the distribution is long-tail if they want to use Long-tail Replacement. The

method is as follows: users can sample the dataset, and plot a figure to show the frequency distribution to check whether there is a long tail in the figure. More methods can be found in the literature [22]–[26].

IV. Mathematical Proofs

We first claim the basic version with the Deviation Eliminator has no overestimation error on significance, and then derive the formula of error bound and correct rate. Due to space limitation, the proof of no overestimation is detailed in the appendix of technical report [31].

A. The Claim 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 Bound of 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) The corresponding cell is not the smallest cell at any point of time.*

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 $\alpha + \beta$, 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 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. For convenience, we assume that $f_1 \geqslant f_2 \geqslant \cdots \geqslant f_M$. According to the nature of Zipfian dataset, we have

$$f_i = \frac{N}{i^\gamma \zeta(\gamma)} \tag{3}$$

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 $\wp_i$ denote the probability that e_i is mapped to the same bucket as e and the number of appearances of e_i has ever been larger than that of e at some time during the whole process. If $f_i > f$, it is obvious that $\wp_i = \frac{1}{w}$, otherwise $\wp_i = \frac{1}{w} * \frac{f_i}{f+1}$[3], where w is the number of buckets. For convenience, we use *useful* to describe that an element e_i satisfies the above condition.

Let $dp_{j,x}$ denote the probability that for the first j frequent items, there are x useful items. We have

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

Let $\mathcal{P}$ denote the correct rate for e, since the Lemma IV.1 is a sufficient condition that the reported significance of e is correct, we have

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

where d is the number of cells in each bucket.

Proof. For the j^{th} item, it has the probability of $\wp_j$ to be useful. Since there are totally two different conditions, according to the principle of probability accumulation, we get the probability in Equation 4.

According to IV.1, if there are k useful items, as long as $k < d - 1$, the reported significance of e is sure to be correct. Since k ranges from 0 to $d - 2$, we get the probability in Equation 5. □

C. The Error Bound

The definitions of symbols used below are similar to the ones mentioned in IV-B .

Theorem IV.2. *Given a small positive number ϵ, for an item e_i recorded in the lossy table, let s_i and $\hat{s}_i$ be its real significance and estimated significance, respectively. Assume that the coefficients of frequency and persistency are respectively α and β, we have*

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

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

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

Proof. If no Significance Decrementing operation is performed on e_i, the estimated significance is exactly equal to the real significance. Let X_i be the number of times this operation is performed on e_i, we have

$$\hat{s}_i = s_i - X_i * (\alpha + \beta) \tag{9}$$

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

$$P_{small} = \binom{i-1}{d-1} \left(\frac{1}{w}\right)^{d-1} \left(\frac{w-1}{w}\right)^{i-d} \tag{10}$$

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 significant than e_i. The expectation of V is

$$\mathbb{E}(V) = \frac{1}{w} \sum_{j=i+1}^{M} f_j \tag{11}$$

Then we can drive the formula for $\mathbb{E}(X_i)$:

$$\mathbb{E}(X_i) = P_{small} * \mathbb{E}(V) \tag{12}$$

According to the equation 9, we can get the equality for $\mathbb{E}(\hat{s}_i)$:

$$\mathbb{E}(\hat{s}_i) = s_i - P_{small} * \mathbb{E}(V) * (\alpha + \beta) \tag{13}$$

[3]It is similar to the non-descending path number problem.

Based on Markov inequality, we have

$$Pr\{s_i - \hat{s_i} \geqslant \epsilon N\} \leqslant \frac{\mathbb{E}(s_i - \hat{s_i})}{\epsilon N} \leqslant \frac{P_{small} * \mathbb{E}(V) * (\alpha + \beta)}{\epsilon N} \quad (14)$$

Therefore, the theorem holds. □

V. Experimental Results

In this section, we conduct experiments to show the performance of LTC 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[4]: *ARE*(average relative error) and *Precision*. Suppose 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.

B. Datasets

1) Social: This dataset comes from a real social network [32], 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. [33] 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* [34], consisting of IP packets. 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.

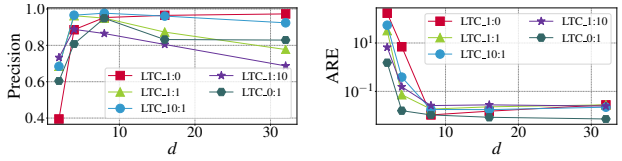
We also generate synthetic datasets and conduct experiments on them. Due to space limitation, we present results of these experiments in the appendix of technical report [31].

Implementation: We implement our algorithm and other related algorithms in C++. Here we use Bob Hash [35], [36] as our hash function.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 LTC and other related algorithms are available on Github [31].

C. Experiment Setup

For finding frequent items, we compare the performance of LTC with SS, CSS, LC, CM+heap (CM) and CU+heap (CU). For finding persistent items, we compare the performance of LTC with PIE, modified CM+heap (CM) and modified CU+heap (CU). For finding significant items, we compare the performance of LTC with modified CM+heap (CM) and modified CU+heap (CU). To achieve a head-to-head comparison, we use the same memory for every algorithm. For the PIE algorithm, since it needs to maintain a Space Time Bloom Filter for each period, it cannot decode any item when the memory is tight. Thus, we use T times of the default memory size for PIE, where T is the number of periods, to make its performance comparable. For SS, CSS, LC, PIE and LTC, the number of cells is determined by the memory size. For sketch-based algorithms, we set the number of arrays to 3. When they are used to find frequent items, the size of the heap is k, and we allocate the rest memory to the sketch. When they are used to find persistent items, we need to maintain a standard BF to record whether an item has appeared in the current period, and therefore we allocate half of the memory to the standard BF and the rest memory to the sketch+heap. When they are used to find significant items, for each algorithm we maintain two sketches: one for finding frequent items, and the other for finding persistent items, and we allocate the whole memory to them evenly. For every experiment, we query top-k items once at the end.

D. Effects of d



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

Fig. 11: Varying d.

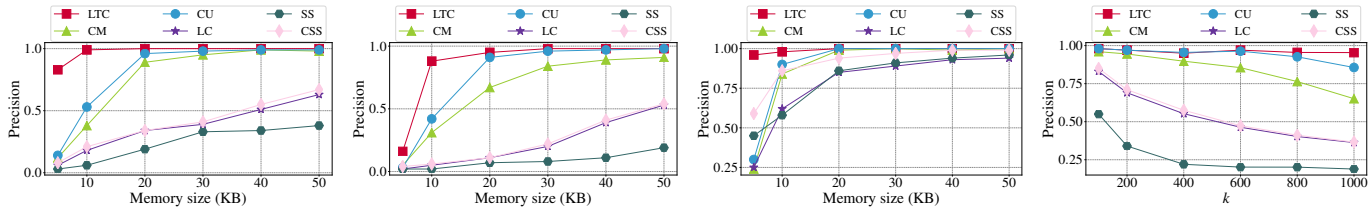
In this experiment, we evaluate the effects of the parameter d, where d is the number of cells in each bucket. *When d increases, the performance first goes up and then goes down.* To make the experiments more comprehensive, we set five pairs of parameters: 1) $\alpha = 0, \beta = 1$, 2) $\alpha = 1, \beta = 0$, 3) $\alpha = 1, \beta = 1$, 4) $\alpha = 10, \beta = 1$, and 5) $\alpha = 1, \beta = 10$. We also set $k = 1000$ and the memory size to 100KB. 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 11(a) shows how precision changes when d varies from 2 to 32. Figure 11(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.

E. 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, CM and CU.

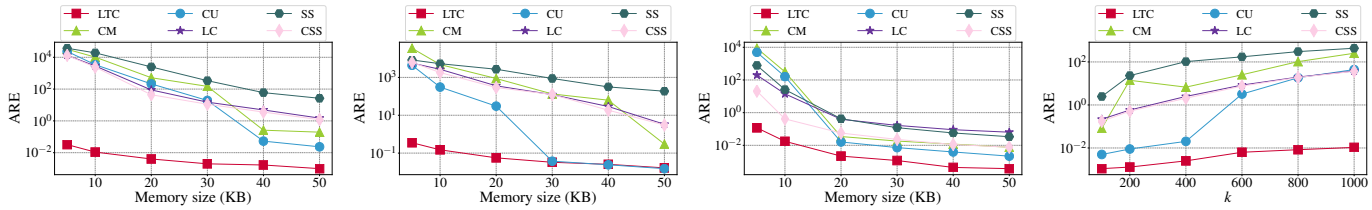
1) Precision vs. memory size (Figure 12(a)-(c)). *LTC has the highest precision among all data structures no matter how the memory size changes.* In this experiment, we set $k = 100$, and vary the memory size from 5KB to 50KB. As shown in Figure 12(a), for the CAIDA dataset, when the memory size is 10KB, the precision of LC, SS, CSS, CM and CU is respectively

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



(a) Varying memory size (CAIDA). (b) Varying memory size (Network). (c) Varying memory size (Social). (d) Varying k (Network).

Fig. 12: Measuring precision on finding frequent items.



(a) Varying memory size (CAIDA). (b) Varying memory size (Network). (c) Varying memory size (Social). (d) Varying k (Network).

Fig. 13: Measuring ARE on finding frequent items.

18%, 6%, 21%, 38% and 52%, while the one of LTC reaches 99%. As the memory size increases, the precision of LTC is always 100%, while the precision of LC, SS, CSS, CM and CU is respectively 63%, 38%, 67%, 98% and 99% at most. As shown in Figure 12(b), for the Network dataset, when the memory size is 10KB, the precision of LC, SS, CSS, CM and CU is respectively 5%, 2%, 6%, 31% and 41%, while the one of LTC reaches 88%. As shown in Figure 12(c), for the Social dataset, when the memory size is 10KB, the precision for LC, SS, CSS, CM, CU and LTC is respectively 62%, 58%, 86%, 84%, 87% and 98%. All these experiments show that LTC has much better precision than the other algorithms.

2) ARE vs. memory size (Figure 13(a)-(c)). *LTC has the lowest ARE among all data structures no matter how the memory size changes.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 13(a), for the CAIDA dataset, the ARE of LTC is respectively between 1512 and 260869 times, 26657 and 1748590 times, 1199 and 220837 times, 155 and 940484 times, 105 and 872303 times smaller than LC, SS, CSS, CM and CU. As shown in Figure 13(b), for the Network dataset, the ARE of LTC is respectively between 202 and 17960 times, 11243 and 48184 times, 183 and 12075 times, 18 and 32087 times, 1.01 and 27620 times smaller than LC, SS, CSS, CM and CU. As shown in Figure 13(c), for the Social dataset, the ARE of LTC is respectively between 141 and 805 times, 90 and 1465 times, 20 and 26 times, 16 and 17879 times, 11 and 17765 times smaller than LC, SS, CSS, CM and CU.

3) Precision vs. k (Figure 12(d)). *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 100 to 1000. Due to space limitation, we just show the results on Network dataset. As shown in Figure 12(d), as k increases, the precision of LC, SS, CSS, CM and CU is respectively changed from 78% to 36%, 53% to 19%, 79% to 36%, 97% to 65% and 99% to 88%, while the one of LTC is always larger than 95%.

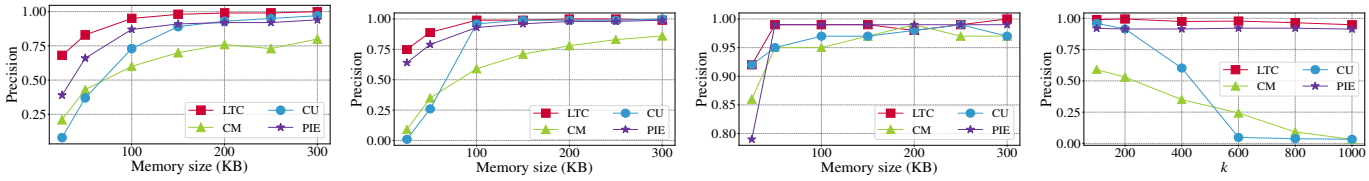
4) ARE vs. k (Figure 13(d)). *LTC has the lowest ARE among all data structures no matter how the standard of 'top' changes.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 13(d), for the Network dataset, the ARE of LTC is respectively between 448 and 3565 times, 17869 and 41172 times, 398 and 3459 times, 2677 and 23786 times, 132 and 3575 times smaller than LC, SS, CSS, CM and CU.

Analysis. Sketch-based algorithms usually record frequencies of all infrequent items, which is space-consuming, leading to poor accuracy. As for counter-based algorithms, let us take Space-Saving as an example. When the data structure (Stream-Summary) is full of items, every item will replace the currently smallest item in Stream-summary, and increment the frequency by 1. Since most items are infrequent, such an increment would lead to huge overestimation error. In contrast, we propose Long-Tail Replacement strategy: every infrequent item only decrements (rather than replace) the current smallest item by 1, and replacement only happens when the frequency of the smallest item is decremented to 0, and then a new item will be inserted. The frequency of the new item can be accurately restored thanks to the long-tail distribution. Therefore, LTC achieves much better performance than other related algorithms, and the above experimental results fully verify this conclusion.

F. Comparisons on Finding Persistent Items

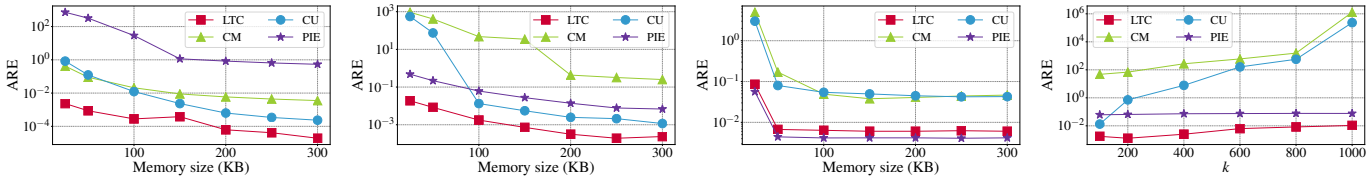
In this set of experiments, we set $\alpha = 0, \beta = 1$, which means the significance of an item is only related to its persistency. We compare the performance of LTC with PIE, CM and CU. Note we use T times of the default memory size for PIE, where T is the number of periods. In other words, when other algorithms use hKB, PIE will use 200*hKB, 1000*hKB, and 500*hKB on Social dataset, Network dataset, and CAIDA dataset respectively.

1) Precision vs. memory size (Figure 14(a)-(c)). *LTC has the highest precision for all memory settings.* In this experiment, we set $k = 100$, and vary the memory size from 25KB to 300KB for LTC, CM and CU. As shown in Figure 14(a), for



(a) Varying memory size (CAIDA). (b) Varying memory size (Network). (c) Varying memory size (Social). (d) Varying k (Network).

Fig. 14: Measuring precision on finding persistent items.



(a) Varying memory size (CAIDA). (b) Varying memory size (Network). (c) Varying memory size (Social). (d) Varying k (Network).

Fig. 15: Measuring ARE on finding persistent items.

the CAIDA dataset, the precision of CM, CU, PIE, and LTC is respectively changed from 6% to 80%, 2% to 95%, 66% to 94%, and 70% to 100%. As shown in Figure 14(b), for the Network dataset, the precision of CM, CU, PIE and LTC is respectively changed from 4% to 86%, 2% to 99%, 64% to 99%, and 75% to 99%. As shown in Figure 14(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 three algorithms.

2) ARE vs. memory size (Figure 15(a)-(c)). *LTC has the lowest ARE for all memory settings.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 15(a), for the CAIDA dataset, the ARE of LTC is respectively between 3020 and 370384 times, 145 and 301294 times, 23 and 184 times smaller than CM, CU and PIE. As shown in Figure 15(b), for the Network dataset, the ARE of LTC is respectively between 106 and 49224 times, 3 and 47230 times, 26 and 44 times smaller than CM, CU and PIE. As shown in Figure 15(c), for the Social dataset, the ARE of LTC is respectively between 6 and 25 times, 5 and 23 times smaller than CM and CU, and approximately the same as PIE.

3) Precision vs. k (Figure 14(d)). *LTC has the highest precision for all k settings.* In this experiment, we set the memory size to 100KB for LTC, CM and CU, and vary k from 100 to 1000. Due to space limitation, we just show the results on Network dataset. As shown in Figure 14(d), when $k = 100$, the precision of CM, CU and PIE is respectively 60%, 93% and 91.5%, while the one of LTC reaches 99%. Furthermore, as k increases, the precision of LTC is always larger than 95%.

4) ARE vs. k (Figure 15(d)). *LTC has the lowest ARE for all k settings.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 15(d), the ARE of LTC is respectively between 52942 and 10^8 times, 32 and 10^8 times, 7 and 50 times smaller than CM, CU and PIE.

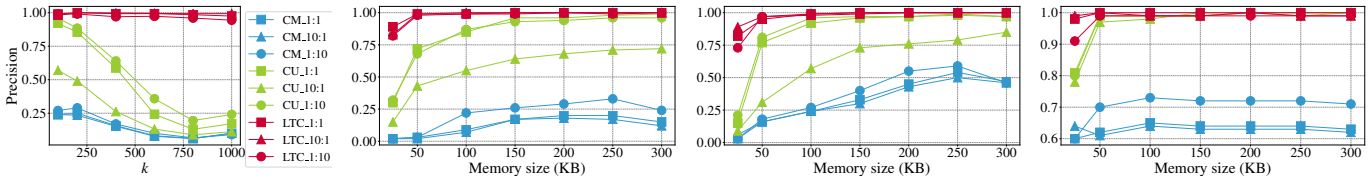
Varying the number of periods. We also conduct experiments of varying the number of periods. Due to space limitation, we present the results in the appendix of technical report. The results show that *LTC has the highest precision and lowest ARE for all settings of the number of periods.*

Analysis. Sketch-based algorithms still have their shortcomings. They need to record persistencies of all items and allocate about half of the size of memory to record whether an item appears in the current period, leading to a waste of memory. As for PIE, it uses Raptor codes [30] to record and identify item IDs. However, PIE needs to maintain a data structure for every period to record persistencies of all items, which also fails to achieve high memory efficiency. In contrast, LTC only records persistent items, and thus achieves higher memory efficiency. Besides, LTC leverages the spirit of the well known CLOCK [27], [28] algorithm to increment the persistentcy by one for any item that appears once or more than once in one period. It saves memory efficiently, since we just use two flags (two bits) for every cell.

G. Comparisons on Finding Significant Items

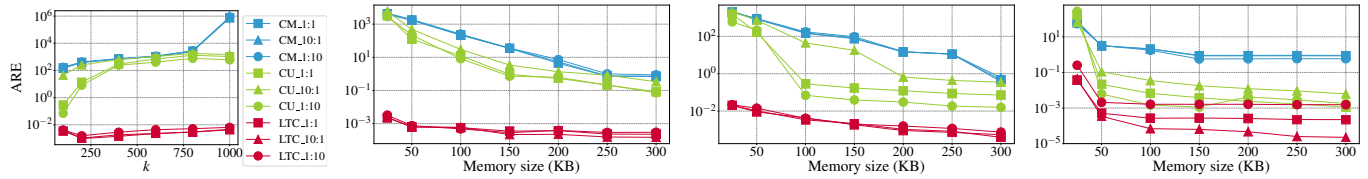
There is no prior work on finding significant items, thus we combine the best algorithm on finding frequent items with the best algorithm on finding persistent items to find significant items. Notice that sketch-based algorithms are the best algorithms except LTC on both finding frequent items and finding persistent items, therefore we modify them 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 LTC_1:10, LTC_1:1, and LTC_10:1 to denote the performance of LTC on the corresponding parameters, where the first value is α and the second value is β. The denotations of CM and CU on corresponding parameters are similar. *We observe that the performance of CU is much better than that of CM. Therefore, below we just illustrate the details about LTC and CU.*

1) Precision vs. memory size (Figure 16(b)-(d)). *The precision of LTC is higher than that of CM and CU.* In this experiment, we set $k = 100$, and vary the memory size from 25KB to 300KB. As shown in Figure 16(b), 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 CU is respectively 68%, 41%, and 71% on each pair of parameters. As shown in Figure 16(c), for the Network dataset, when the memory size is 25KB, the precision of LTC reaches 70% no matter on which pair of parameters, while



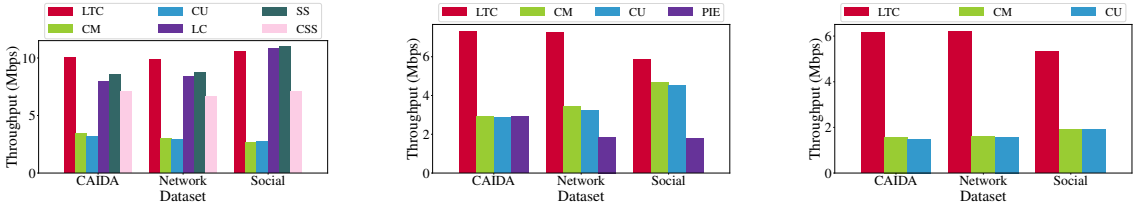
(a) Varying k (Network). (b) Varying memory size (CAIDA). (c) Varying memory size (Network). (d) Varying memory size (Social).

Fig. 16: Measuring precision on finding significant items.



(a) Varying k (Network). (b) Varying memory size (CAIDA). (c) Varying memory size (Network). (d) Varying memory size (Social).

Fig. 17: Measuring ARE on finding significant items.



(a) Finding frequent items. (b) Finding persistent items. (c) Finding significant items.

Fig. 18: Throughput vs. dataset.

the one of CU is respectively 27%, 12%, and 28% on each pair of parameters. As shown in Figure 16(d), for the Social dataset, when the memory size is 25KB, the precision of CU is respectively 82%, 78%, and 73% on each pair of parameters, while the one of LTC is always larger than 90% no matter on each pair of parameters.

2) ARE vs. memory size (Figure 17(b)-(d)). *The ARE of LTC is lower than that of CM and CU.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 17(b), for the CAIDA dataset, the ARE of LTC is respectively between 1683 and 128520 times, 850 and 37120 times, 1650 and 1920182 times smaller than CU on each pair of parameters. As shown in Figure 17(c), for the Network dataset, the ARE of LTC is respectively between 15 and 53928 times, 1958 and 1952018 times, 32 and 29185 times smaller than CU on each pair of parameters. As shown in Figure 17(d), for the Social dataset, the ARE of LTC is respectively between 1.01 and 857 times, 192 and 3219 times, 15 and 7293 times smaller than CU on each pair of parameters.

3) Precision vs. k (Figure 16(a)). *The precision of LTC is higher than that of CM and CU.* In this experiment, we set the memory size to 100KB, and vary k from 100 to 1000. Due to space limitation, we just show the results on Network dataset. As shown in Figure 16(a), as k increases, the precision of CU is respectively changed from 93% to 23%, 53% to 17%, and 94% to 21% on each pair of parameters, while the one of LTC is always larger than 94% on any k or pair of parameters.

4) ARE vs. k (Figure 17(a)). *The ARE of LTC is lower than that of CM and CU.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 17(a), the ARE of LTC is respectively between 128592 and 10^8 times, 182059 and 10^8 times, 92730 and 10^8 times smaller than CU on each pair of parameters.

Varying the number of periods. We also conduct experiments of varying the number of periods. Due to space limitation, we present the results in the appendix of technical report. The results show that *the precision of LTC is higher than that of CM and CU, and the ARE of LTC is lower than that of CM and CU.*

Analysis. Sketch-based algorithms need to record frequencies and persistencies of all items, while LTC only records significant items. Therefore, our LTC is much more memory efficient. When we combine these two metrics into significance, the gap of performance between existing algorithms and LTC becomes larger.

H. Experiments on Throughput

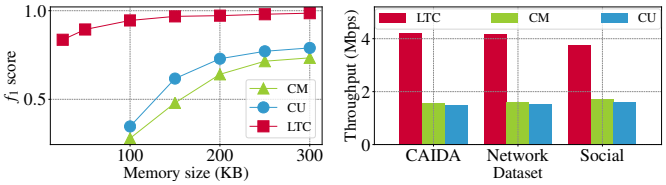
In this section, we conduct experiments to compare the throughput of all the above algorithms.

1) Finding frequent items (Figure 18(a). We set memory size to 100KB and $k = 100$. As shown in Figure 18(a), the result shows that LTC is not only more accurate, but also achieves higher throughput. The reason why sketch-based algorithms are slow is that they need to maintain a min-heap whose time complexity of update is $O(log(k))$ for each insertion, where $log(k)$ is the depth of the min-heap. In contrast, LTC only needs about 1 memory access for each insertion.

2) Finding persistent items (Figure 18(b)). We set the memory size to 100KB and $k = 100$. As shown in Figure 18(b), the throughput of LTC is still much higher than the other algorithms.

3) Finding significant items (Figure 18(c)). We set the memory size to 100KB, $k = 100$, and $\alpha = 1, \beta = 1$. As shown in Figure 18(c), LTC still achieves high throughput.

I. Application1



(a) f_1 score vs. memory size. (b) Throughput vs. dataset.

Fig. 19: Comparisons between LTC and sketch-based algorithms.

In practice, user may tend to use such a definition of significance: the items with frequency at least x and persistency at least y. Fortunately, our LTC can also work well. We can simply use two LTCs, one for finding items with frequency at least x by setting β to 0; and the other for finding items with persistency at least y by setting α to 0.

We also conduct experiments to compare LTC with CM and CU. Suppose that the correct significant set is ϕ, and the reported set is ψ. Let $P = \frac{|\phi \cap \psi|}{|\phi|}$ and $Q = \frac{|\phi \cap \psi|}{|\psi|}$. f_1 score is defined as $\frac{2*P*Q}{P+Q}$. For the first experiment, the dataset we use is Network dataset. We set $x = 1000, y = 300$, and vary the memory size from 25KB to 300KB. As shown in Figure 19(a), the results show that the f_1 score of LTC is much higher than CM and CU. For the second experiment, we set the memory size to 100KB, $x = 1000, y = 300$. As shown in Figure 19(b), the results show that the throughput of LTC is still much higher than CM and CU.

J. Application2

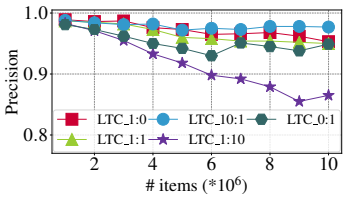


Fig. 20: Real-time queries

We can apply LTC to real-time queries. For every query, we report the top-k significant items recorded in the lossy table. In this experiment, we set $k = 1000$ and the memory size to 100KB. The dataset we use is Network dataset. For every 10^6 items, we query top-k significant items. As shown in Figure 20, the results show that our LTC still achieves a high precision on real-time queries.

VI. Conclusion

In this paper, we abstract a problem named finding top-k significant items, which is encountered in many applications but not 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 show that our LTC achieves high accuracy and high speed with small memory. The source codes of LTC and other related algorithms are available at Github [31].

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

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *Acm Sigcomm Computer Communication Review*, 34(2):39–53, 2004.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

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

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

[30] Qingchun Meng and Xiaojing Wang. Research on precoding method in raptor code. *Computer Engineering*, 33(1):1–3, 2007.

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

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

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

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

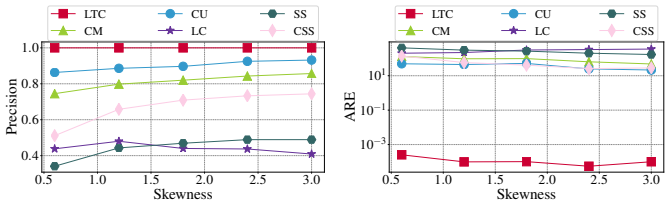
[35] The source code of Bob Hash. http://burtleburtle.net/bob/hash/evahash.html.

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

[37] 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.

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

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



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

Fig. 21: Varying skewness.

VII. Appendix

A. The 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*

$$\hat{s}_i \leqslant s_i \tag{15}$$

Proof. Apparently, the significance of an item can be decomposed into two metrics (*i.e.*, frequency and persistency). We will respectively prove that there is no overestimation error for those two metrics below.

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

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

Case 1: e_i is not found in the bucket $A[h(e_i)]$, but there is an empty cell. The estimated frequency of this item is equal to 1, and the real frequency is obviously no less than 1, thus the inequality holds.

Case 2: e_i is not found in the bucket $A[h(e_i)]$ and there is no empty cell, which means it can not be inserted into the lossy table. The estimated frequency of this item is equal to 0, thus the inequality holds.

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

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

To make the proof clearer, let p_i^* be the real persistency of e_i other than the current period. Specifically, if e_i appears in the period t, p_i^* will be incremented at time of period $t+1$. Apparently, $p_i^* \leqslant p_i$ holds. We will prove the inequality $\hat{p}_i \leqslant p_i^*$ holds below. Similar to the proof of the inequality of frequency, at first all cells in the lossy table are empty, thus the inequality holds. For the item e_i, if it appeared in the previous period, $\hat{p}_i$ is incremented by 1 at most and p_i^* is surely incremented by 1, thus the inequality still holds. Otherwise, $\hat{p}_i^*$ and p_i are both not changed. Therefore, the inequality $\hat{p}_i \leqslant p_i^* \leqslant p_i$ holds at any time of period.

Since s_i is a combination of f_i and p_i, we have $\hat{s}_i \leqslant s_i$. □

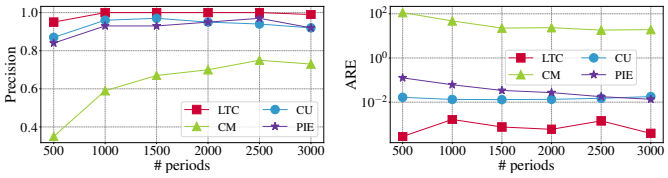
B. Remaining Experimental Results

1) Experiments on Synthetic Datasets:

Synthetic datasets generator: We generate 5 different datasets according to Zipfian [37] distribution by Web Polygraph [38], with different skewness [39] 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.

Measuring precision, varying skewness (Figure 21(a)). *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 21(a), the precision of LC, SS, CSS, CM and CU is respectively between 41% and 48%, 34% and 49%, 51% and 74%, 74% and 85%, 82% and 91%, while the one of LTC achieves 100% no matter on which skewness.

Measuring ARE, varying skewness (Figure 21(b)). *LTC has the lowest ARE among all data structures no matter how the skewness of dataset changes.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 21(b), the ARE of LTC is respectively between 803309 and 6034705 times, 1647522 and 3709083 times, 283537 and 592141 times, 482892 and 1173926 times, 230129 and 481020 times smaller than LC, SS, CSS, CM and CU.



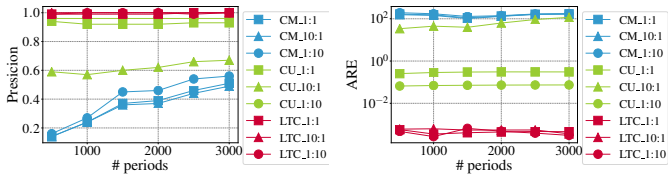
(a) Measuring precision (Network). (b) Measuring ARE (Network).

Fig. 22: Varying the number of periods on finding persistent items.

2) Varying the Number of Periods on Finding Persistent Items:

Measuring precision (Figure 22(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, CM, and CU, $k = 100$, and vary T from 500 to 3000, where T is the number of periods. As the experimental method mentioned above, when T is changed, the memory size of PIE changes but that of the other algorithms does not change. We just show the results on Network dataset. As shown in Figure 22(a), when $T = 500$, the precision of CM, CU and PIE is respectively 35%, 86% 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%.

Measuring ARE (Figure 22(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 22(b), the ARE of LTC is respectively between 12834 and 225914 times, 12 and 87 times, 12 and 449 times smaller than CM, CU and PIE.



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

Fig. 23: Varying the number of periods on finding significant items.

3) Varying the Number of Periods on Finding Significant Items:

Measuring precision (Figure 23(a)). *The precision of LTC is higher than that of CM and CU.* 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. We just show the results on Network dataset. As shown in Figure 23(a), the precision of CU is approximately 91%, 61%, 93% on each pair of parameters, respectively, while the one of LTC is always larger than 99% on any T or pair of parameters.

Measuring ARE (Figure 23(b)). *The ARE of LTC is lower than that of CM and CU.* The configuration of this experiment is the same as the previous experiment. We just show the results on Network dataset. As shown in Figure 23(b), the ARE of LTC is respectively between 117 and 328 times, 2 and 7 times, 539 and 829 times smaller than CU on each pair of parameters.