Fair Election: A Generic Framework for Data Stream Processing

Paper #579, 9+2 pages

ABSTRACT

Data stream processing has become an important area over the recent years. Storing data streams exactly as they come is often too memory-consuming, as well as unnecessary. Although a data stream can be large in size, very often one only cares about a very small part of it, e.g., its items that are largest in terms of a certain property. Existing works propose solutions for a specific property, such as finding frequent items or caching recent items. This paper targets a more generic goal: finding the top-k items for a given property. The state-of-the-art solutions use a data structure named Stream-Summary, which evicts the globally smallest item in the Stream-Summary when new items arrive, using different eviction strategies. Stream-Summary is similar to a min-heap: it can locate the globally smallest item in O(1) time, but needs much more memory than the min-heap. Therefore, existing solutions are not memory efficient. To address this issue, we propose a generic framework, namely Fair Election, in two versions. The basic version uses the *divide and conquer* to find and evicts the locally smallest item, so as to achieve time and space efficiency. However, unfairness happens in the worst case. Our second and optimized version minimizes unfairness. We applied our framework to three algorithms for finding frequent items. We also use the same framework to complete the task of performing approximate LRU eviction algorithm. Theoretical analysis and experimental results show that after using our framework, the accuracy improves up to 800 times, and the speed improves up to 1.3 times. The source code used in this paper is released on Github [1].

1 INTRODUCTION

1.1 Background and Motivation

Big data is often organized as a high-speed data stream, such as phone calls, network traffic, video/audio streams, web clicks and website requests [2–5]. A data stream is a series of items arriving over time, often at high speed, and each item can appear multiple times. Each item has a countable property $\mathcal{P}$, such as the number of appearances, or its arriving time sequence. For convenience, given a countable property, we call item with a large or small value `large item` or `small item`, respectively.

Despite the large volume of data of some streams, users are often interested in a very small part of it. Typically, the most fundamental query is to report the top-k largest items: the k items with the largest value of some property. For example, finding the top−k frequent items is to report the k items whose frequencies are the largest [6]. For another example, managing a memory cache consists in keeping top-k `recent items` while evicting `old items` [7].

Because the number of items in a data stream is often very large, it is challenging to keep track of the top-k largest items. For example, in a graph, to elect the top-k edges with largest weight, checking every single edge of the graph is time consuming. Therefore, in practice, grouping or clustering is a fast and practical method to achieve time and space efficiency, at the cost of unfairness. Unfairness happens because large items (*e.g.*, edges with large weight) could be assigned to the same group, and not all can be kept in the memory assigned to the limited-size group. In this paper, we aim to achieve a fair election to overcome unfairness, and to improve space and time efficiency at the same time.

1.2 Prior Art and Their Limitations

Existing works focus only on one specific property, such as finding frequent items or caching recent items. We briefly present the related work for each of the approach.

There are mainly two types of solutions for finding frequent items. First, the classic method is to use one sketch (*e.g.*, sketches of CM [8], CU [9], or Count [10]) together with a min-heap. We call these solutions sketch+heap for convenience. More advanced sketches such as the Pyramid sketch [11] and the Biased sketch [12] did not claim to find frequent items, and therefore we do not discuss them in this paper. A sketch records the frequencies of all items, chooses items with large frequencies, and inserts them into the min-heap. The second type of solution, also the most widely used algorithm for finding frequent items, is Space-Saving [13] and its various variants [14–16]. The original Space-Saving has comparable accuracy to sketch+heap. However, because Space-Saving does not record the information of small items, it has the potential to achieve much higher accuracy. Therefore, many works have tried to improve Space-Saving. Among existing works, [15] is the most recent work that achieves much higher accuracy than Space-Saving. However, to keep the top-k frequent items, all Space-Saving based algorithms use two data structures, a Stream-Summary to store the largest items, and a hash table to check whether the incoming item is already stored. Both of these two data structures achieve O(1) time complexity at the cost of large memory usage.

Caching recent items is another classic problem, and LRU is often the best or the second-best choice in various caching scenarios [17–19]. Unfortunately, when the data stream is large, it is time- or space-consuming to locate the least recently used item. Therefore, approximate LRU algorithms [20–23] have been proposed. The most widely used approximate LRU algorithm is the CLOCK algorithm [24]. CLOCK is time efficient in average, but not in the worst case. Furthermore, it is not very accurate, because it seldom evicts the least recently used item.

We argue that existing algorithms for keeping global top-k items are inevitably either time-consuming or space-consuming. The goal of this paper is to achieve time- and space-efficiency, as well as high accuracy at the same time.

1.3 Our Solution

In this paper, we propose a generic framework, called Fair Election, to find the top-k largest items in terms of a given property. Our solution has two versions, a basic version crafted to do well in the average case, and an optimized version that improves accuracy in the worst case.

In the basic version, we use a common strategy: *divide and conquer*. Specifically, we divide the data stream into w small groups using a hash function, and for each group we keep track of the top k' ($w * k' \geqslant k$) largest items, which we call *locally largest items*. We have three other related terms: *locally smallest*, *globally largest*, and *globally smallest*. This basic version is simple but effective, because for each incoming item, we only need to locate the locally smallest item in a group, which is much easier than locating the globally smallest in the whole data stream. As k' is usually very small (*e.g.*, 4), we do not need hash tables. Instead, we just check the incoming items against all items of the group. It is easy to prove that this basic version is both memory and time efficient. The only shortcoming of this basic version is that it suffers the problem of unfairness. Specifically, in the worst case, when more than k' globally large items are occasionally hashed into the same group, some globally large items will be discarded, which is highly undesirable.

To overcome the shortcoming of the basic version, we propose an optimized version of Fair Election which improves the performance in the worst case. Our key idea is to use a small list, named *coordinator*, to minimize the unfairness among different groups. When a group is hashed by too many globally large items, we will coordinate this situation by moving some of these items to the coordinator. When the coordinator is full, the smallest item in the coordinator will be evicted back to its original group. In general, unfairness does not happen, and the optimized version has a similar performance as the basic version. In the worst case, unfairness happens, the optimized version achieves much higher accuracy than the basic version.

We insist that our solution is a generic framework. Specifically, as long as users define the property, our framework can accurately find top-k globally largest or smallest items with small memory usage and fast speed. To demonstrate this, we present two case studies: finding frequent items and caching recent items. Theoretical analysis and experimental results show that when using our framework, the time, speed and accuracy improve a lot. Typically, when applying our framework to find frequent items, for a given amount of memory, the speed is improved by up to 1.3 times, while the accuracy is improved by up to 800 times. When applying to cache recent items, for a given memory size, the speed is improved by up to 12 times, while the miss rate is lowered by up to 38% compared to exact LRU algorithm.

1.4 Key Contributions

In summary, the key contributions of this paper are the following.

- We propose a generic framework to find top-k items, for a given definition of the item property. Our framework is suitable to most existing solutions.
- To demonstrate how our framework works, we apply it to two kinds of queries: finding top-k frequent items and caching recent items.
- To present theoretical analysis and experiments. The results show that when using our framework, existing algorithms achieve much higher accuracy and speed.

2 BACKGROUND AND RELATED WORK

In this section, we first give the definition of the problem. Then, we briefly introduce existing algorithms which are commonly used for solving the problem.

2.1 Problem Definition

Definition 2.1. **Data Stream:** A data stream $\mathcal{S}$ is a series of incoming items, each of them can appear multiple times.

Definition 2.2. **Item Property:** Each item has a countable property $\mathcal{P}$, such as the number of appearances, or the arriving time point. For convenience, if an item has large value of the property, we call it a large item.

Definition 2.3. **Top-k query:** The most fundamental query in a stream is to report the top-k largest items: k items with the largest property.

2.2 Prior Works for Finding Frequent Items

Finding frequent items in data streams is a relevant task [25] in multiple applications, such as anomaly and DDoS detection [26–28], and network traffic engineering [29]. There are two types of solutions to find frequent items.

The first type of solutions record only the information of frequent items. This includes Space-Saving [13] and its variants. Space-Saving uses a data structure called Stream-Summary to record the top-k frequent items as shown in Figure 1. The recording process of Space-Saving is rather simple: for each incoming item, if it exists in the Stream-Summary, its frequency stored in the Stream-Summary is incremented. Otherwise, the incoming item will replace the smallest item in the summary, and its frequency is set to $\hat{f}_{min}$+1, where $\hat{f}_{min}$ is the frequency of the smallest item in the summary. Using Stream-Summary, Space-Saving can achieve insertions in O(1) time complexity. However, the Stream-Summary data structure is not space efficient, because of too many pointers and a hash table.

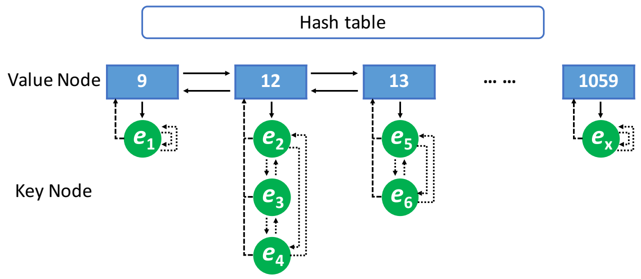


Figure 1: Stream-Summary used in Space-Saving. It contains a hash table and a value nodes list. Each value node has a list of key nodes.

The two contributions of Space-Saving are the Stream-Summary data structure and its update mode, *i.e.*, using $\hat{f}_{min}$+1 as the initial frequency of a new incoming item. For convenience, we call this update mode the *increment strategy*. Many other update modes can be used in the Stream-Summary structure. For example, in Unbiased Space-Saving [15], when a new item arrives, the smallest item is replaced by the new item with probability $\frac{1}{\hat{f}_{min}+1}$. If it is replaced, its frequency is incremented by 1. Otherwise, the frequency does not change. Therefore, as the number of inserted items increases, the smallest item is almost directly replaced by some new items. Another update mode is, when a new item arrives, decreasing the frequency of the smallest item by 1 with probability $c^{-\hat{f}_{min}}$, where c is a constant. If the frequency reaches zero, the smallest item is replaced with the new item and its frequency is set to 1. Therefore, we define two update modes: 1) *probabilistic replacement strategy:* replace the smallest item with the new item with probability $\frac{1}{\hat{f}_{min}+1}$. If it is replaced, increase the frequency by 1. 2) *decrement strategy:* decrease the frequency of the smallest item by 1, and if the frequency reaches zero, replace the smallest item with the new item and set the frequency to 1.

The second kind of solutions for finding frequent items records the information of both frequent and infrequent items. Typical algorithms include sketch+heap, and ASketch [30]. Sketch+heap consists in using one classic sketch (*e.g.*, the CM sketch [8], the CU sketch [9], or the Count sketch [10]) plus a min-heap. When inserting an item e into the sketch, its frequency is also queried. If its frequency is larger than the smallest one in the min-heap, e will be inserted into the min-heap, and the original smallest item is evicted. In addition to the min-heap, it also needs a hash table to decide whether e is already in the min-heap. ASketch [30] is similar to sketch+heap. The main difference is that ASketch replaces the min-heap and the hash table with a small array. This can reduce memory usage, but the array is searched linearly, so it cannot be large. The authors recommend a length of 32, and claim that ASketch mainly focuses on recording all items' frequencies. Therefore, in our experiments, we compare with the sketch+heap, but not with ASketch. In addition, based on these algorithms for finding frequent items, Cold-Filter [16] can improve their accuracy by separating frequent and infrequent items beforehand.

2.3 Prior Works for Caching Recent Items

In many caching scenarios, some items must be evicted from the cache to make room for the new incoming items. LRU is a well-known strategy to decide which item should be evicted when the cache is full, *i.e.*, the least recently used item is evicted and replaced by a newly inserted item. Based on the idea of evicting the least recently used items, some LRU-based algorithms have been proposed. For example, [21] uses a tree structure to achieve approximate LRU cache replacement strategy, *i.e.*, it evicts the approximately least recently used item when a new item arrives. CLOCK is another approximate LRU cache replacement strategy. It is more time efficient than normal LRU on average, but not in the worst case. However, these algorithms are not memory efficient. In LRU scenarios, a hash table is used to store the location of each item, to make the lookup fast, and a queue is used to record the order of the items. These two structures help achieve lookup times of $O(1)$, but they need extensive additional memory. To improve the space efficiency of LRU, MemC3 [31] combines the CLOCK scheme and cuckoo hashing [32], a hashing scheme with high space efficiency and O(1) processing time. MemC3 improves space efficiency by 40% over the default LRU strategy, which is still arguably insufficient.

3 THE FAIR ELECTION FRAMEWORK

In this section, we first define the problem that we want to solve. Then, we propose the basic version of our framework, Fair Election. We finish this section by proposing the final version of our framework.

3.1 Problem Definition

Finding the Top−k Items: Given a data stream $\mathcal{S}$, each item could appear more than once. Each item has a countable property $\mathcal{P}$, such as the number of appearances, or the arrival time. Many applications care about only the top-k items: k items with the largest property. There are two typical problems: 1) finding frequent items, 2) caching recent items. Prior works consider them as two independent issues, and design different data structures for them. In this paper, we devise a generic framework, *Fair Election*, to find efficiently the top-k items, or any definition of the property.

3.2 Basic Version: Divide and Conquer

Rationale: With the classic algorithms, the data structures can catch only a small portion of items. When a new item arrives and the data structure is full, we need to find a globally smallest item to update. However, implementing this finding operation in $O(1)$ time can be space inefficient, because an ordered data structure and a hash table are needed. Inspired by the idea of *divide and conquer*, given a data stream, we divide it into sub-streams, each processed separately. Specifically, we use a hash function to uniformly generate sub-streams. For each sub-stream, we use a bucket to keep several items (the locally largest) and their properties. When a new item arrives and the corresponding bucket is full, we find the smallest item in this bucket and update it. This sub-stream grouping is simple and fast. Moreover, it doesn't need additional data structure such as a hash table which limits the space overhead.

Data Structure: As shown in Figure 2, our framework consists of an array of buckets: $B[0], B[1], \cdots, B[m-1]$, where m is the number of buckets. We use a hash function $h(.)$ to uniformly divide a data stream into m sub-streams, each of which is recorded into one unique bucket. A bucket has c cells, and each cell has two fields: item ID and property. The j-th cell in $B[i]$ is denoted as $B[i, j-1]$. For example, the data structure in Figure 2 contains 8 buckets, each of which has 4 cells.

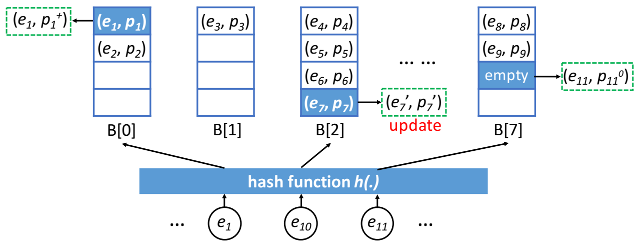


Figure 2: Basic version of Fair Election.

Insertion: Given an incoming item e, we hash it into bucket $B[h(e)\%m]$. We call this bucket the *hashed bucket*. Then, we check all cells in the hashed bucket one by one, and there are three cases:

Case 1: Item e matches one cell in the bucket. We update the cell by increasing its property. For example, in Figure 2, to insert item e_1, we find that e_1 matches cell $B[0, 0]$. Therefore, we update property p_1 to p_1^+.

Case 2: Item e doesn't match any cell, but the hashed bucket is not full. We insert e with an initial property of p^0 into an empty cell of the bucket. For example, as shown in Figure 2, to insert item e_{11}, e_{11} doesn't exist in $B[7]$ but the third cell is empty. Therefore, $B[7, 2]$ is updated to (e_{11}, p_{11}^0) where p_{11}^0 is an initial property.

Case 3: Item e doesn't match any cell and the hashed bucket is full. We find the smallest item in this bucket and update it. For example, to insert item e_{10} in Figure 2, e_{10} doesn't exist in $B[3]$

and the bucket is full. We find the locally smallest item, (e_7, p_7), and update it. Both the item and the property may be changed depending on the different update strategies.

Query: Every bucket in the basic Election framework is designed to hold d locally largest items of a sub-stream. To report the top-k globally largest items in the whole data stream, we simply sort the items in all the buckets.

Large items overflow: In the worst case, many globally large items are hashed to one bucket, and we call this *large items overflow*. In this case, our basic version does not work well, since some large items have to be discarded. To compute the probability that such overflow happens, we first need the following theorem.

THEOREM 3.1. *Let $\mathcal{S}$ be a data stream, m be the number of buckets, c be the number of cells in each bucket, and k be the number of large items that we need to report. Given a bucket, there will be i $(0 \leqslant i \leqslant k)$ large items hashed to this bucket. The probability $Pr(i)$ that in total, i large items are hashed to this bucket, can be computed by the following formula.*

$$Pr(i) = e^{-k/m}\frac{(k/m)^i}{i!} \tag{1}$$

The proof of the above theorem can be found in Section 7.2.

According to the above theorem, we can get the probability of large items overflow, $P_{overflow}$:

$$P_{overflow} = 1 - \sum_{i=0}^{c} Pr(i) \tag{2}$$

In this basic version of Fair Election, when $m = 256$, $c = 8$ and $k = 500$, *i.e.*, the number of buckets is 256 and the number of cells per bucket is 8, to find the largest 500 items, the overflow rate is around 4.99%, which means that overflow happens in around 12 buckets and thus more than 12 large items are discarded. To minimize the probability of discarding large items, we propose an optimized version of Fair Election in the following section.

3.3 Fair Election: Optimized Version

Rationale: In the basic Fair Election, the numbers of large items hashed to each bucket might vary drastically, while the numbers of cells in each bucket are fixed. In the worst case, one bucket could be mapped by too many globally large items, and some large items will be discarded. To improve the performance in the worst case, larger buckets should be used for sub-streams who have more large items. However, such design is difficult to realize, because predicting the number of large items in a sub-stream is challenging, if possible at all. To achieve fairness, we can use an auxiliary *"coordinator"*. The coordinator is an array that can store several items. When a bucket overflows, we move some locally largest/smallest items from the bucket to this coordinator. In this way, the large items that would otherwise be missed will be kept in our optimized Fair Election data structure.

Data Structure: As shown in Figure 3, in the optimized Fair Election framework, in addition to the hash function $h(.)$ and buckets $B[0], B[1], \cdots, B[m-1]$, we use a "coordinator" with l cells, to store those items that would otherwise be discarded. Recall that we use e_j to denote the j-th item in a data stream and p_j to denote the property of e_j recorded in our Fair Election framework. For instance, as shown in Figure 3, $l = 4$, which means that the coordinator can hold 4 items.

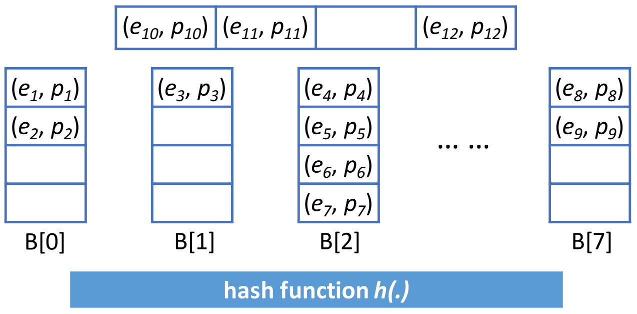


Figure 3: Fair Election with coordinator.

Insertion: To insert an item e, there are four cases.

Case 1: Item e exists in the coordinator or the hashed bucket $B[h(e)\%m]$. We increase the property. For example, as shown in Figure 4(a), to insert item e_2 and e_4, we find e_2 in the coordinator and e_4 in $B[0]$. Then, we increase their properties to p_2^+ and p_4^+, respectively. We use $^+$ to denote a larger property.

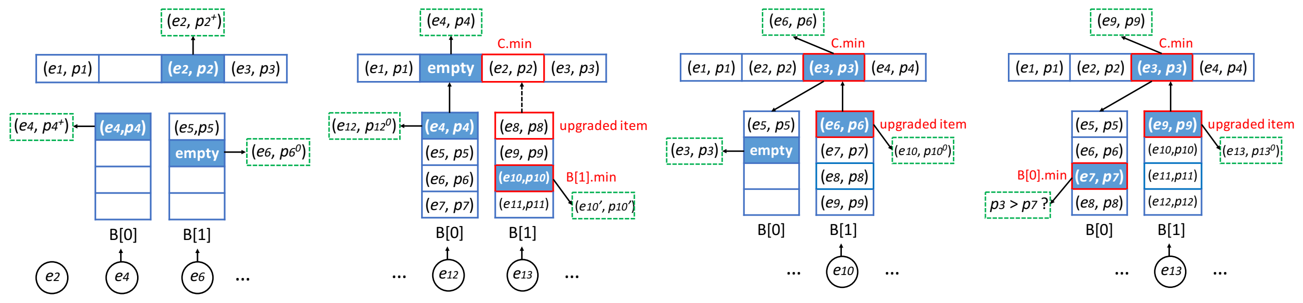
Case 2: Item e doesn't exist in the bucket array or the coordinator, but the hashed bucket $B[h(e)\%m]$ has an empty cell. We insert e with an initial property into the empty cell. For example, as shown in Figure 4(a), to insert item e_6, we insert (e_6, p_6^0) into an empty cell in $B[1]$, where p_6^0 is an initial property. We use ^{0} to denote an initial property.

Case 3: Item e doesn't exist in the framework and the hashed bucket $B[h(e)\%m]$ is full, but we find an empty cell in the coordinator. We move the largest/smallest item in $B[h(e)\%m]$ to an empty cell in the coordinator. For convenience, we call this item *the upgraded item* and the corresponding cell becomes empty. We call this cell the `new-empty cell`. Then, we insert e and an initial property into the new-empty cell. For example, as shown in Figure 4(b), to insert item e_{12}, we move item e_4 to an empty cell in the coordinator and insert (e_{12}, p_{12}^0) into $B[0, 0]$, where p_4^0 is an initial property.

Case 4: Item e doesn't exist in the bucket array or the coordinator, while the hashed bucket $B[h(e)\%m]$ and the coordinator are full. Similar to *case 3*, we choose the largest/smallest item in the hashed bucket as an upgraded item and compare it with the smallest item in the coordinator, which is denoted as $C.min$. Then, there are three sub-cases:

Case 4.1: $C.min$ is larger than the upgraded item. We update the smallest item in the hashed bucket. Both the item and the property may be changed. For example, as shown in Figure 4(b), when inserting an item e_{13}, the hashed bucket $B[1]$ and the coordinator is full. The upgraded item in $B[1]$ is e_8 and the smallest item in the coordinator is e_2. Since $p_8 < p_2$, we update the smallest item in $B[1]$, *i.e.*, e_{10}, according to the specific strategy.

Case 4.2: $C.min$ is smaller than the upgraded item, and the bucket $B[h(C.min)\%m]$ is not full. We insert e with an initial property into the new-empty cell and insert the upgraded item into the cell of $C.min$. Then, we insert $C.min$ into an empty cell in bucket $B[h(C.min)\%m]$. For example, as shown in Figure 4(c), to insert e_{10}, the upgraded item e_6 is larger than $C.min$ (*i.e.*, e_3), and the hashed bucket of e_3, *i.e.*, $B[0]$, is not full. Therefore, we insert e_{10} and an



(a) Cases 1 and 2 (b) Cases 3 and 4.1 (c) Case 4.2 (d) Case 4.3

Figure 4: Insertion of Fair Election. For convenience, we show only two buckets in each graph. In figure (d), if p_3 is larger, we replace (e_7, p_7) with (e_3, p_3).

initial property p_{10}^0 into $B[1]$ and insert (e_6, p_6) into the coordinator. Finally, item e_3 is inserted into an empty cell in $B[0]$.

Case 4.3: $C.min$ is smaller than the upgraded item, and bucket $B[h(C.min)\%m]$ is full. We also insert e into the new-empty cell and insert the upgraded item into the cell of $C.min$. Then, we compare $C.min$ and the smallest item in $B[h(C.min)\%m]$. If $C.min$ is larger, we replace this smallest item in $B[h(C.min)\%m]$ with $C.min$. For example, as shown in Figure 4(d), if $p_3 > e_3$, we replace the smallest item (e_7, p_7) in $B[0]$ with $C.min$ (*i.e.*, (e_3, p_3)).

Note that the operations in the coordinator can be very fast when using SIMD acceleration, the details are provided in Section 7.3.

Query: In the optimized framework, top-k items could also be stored in the bucket array. To report the top-k globally largest items in the data stream, we sort the items in the buckets and the coordinator, and then report a list of large items.

Overflow Rate: In the optimized Fair Election framework, the probability of overflow is exponentially reduced. In the following part, we analyze the reduction of the overflow rate.

Assuming each bucket has c cells, for an arbitrary bucket, the probability that j large items overflow in the bucket is:

$$P_{overflow}(j) = Pr(j + c) \tag{3}$$

Let l be the number of items stored in the coordinator. We can accommodate l large items in the bucket array. Therefore, if and only if over l large items were supposed to be discarded in the bucket array, the Fair Election framework has to discard large items, *i.e.*, it overflows.

Theorem 3.2. *Let $p_{i,j}$ be the probability that in total i large items were supposed to be discarded in the first j buckets, i.e., $B[0], B[1], \cdots, B[j-1]$. We have the following equation.*

$$p_{i,j} = \sum_{k'=0}^{i} p_{i-k',j-1} * P_{overflow}(k') \tag{4}$$

where $i \geqslant 0$ and $j \geqslant 1$.

Proof. For the j^{th} bucket, we use k' to denote the number of large items hashed to this bucket. Assuming that $i - k'$ large items have been hashed to the first $j-1$ buckets, the probability that i large items are hashed to the first j buckets is exactly the probability that k' large items are hashed to the j^{th} bucket. Since k' ranges from 0 to i, we get the probability in Equation 4. □

Therefore, given the number of buckets m, the number of cells in each bucket c, and the number of large items in a data stream k, we can calculate the overflow rate using formula 4. For example, when the number of buckets is 256 and each bucket has 8 cells, we show the effect of the coordinator size (number of cells in the coordinator) on the overflow rate in Table 1.

Coordinator Size	Overflow Rate
0	4.989e-02
1	1.057e-02
2	2.074e-03
4	6.744e-05
8	4.914e-08
12	2.948e-11
16	2.853e-14

Table 1: Overflow rate as a function of the coordinator size.

When the coordinator size is 0, the optimized version of our framework is the same as the basic version. In this case, the overflow rate is 4.989e-02. When the coordinator stores only 4 items, the overflow rate can be reduced by 740 times. When the coordinator stores 16 items, the overflow rate can be reduced by $1.7 * 10^{12}$ times.

4 APPLICATIONS

In this section, we apply our Fair Election to two typical top-k tasks: 1) finding frequent items and 2) caching recent items. Next, we describe in detail these two applications.

4.1 Finding the Top-k Frequent Items

To find the top-k frequent items, we use frequency as the property in our Fair Election framework. We use `heavy items` to denote items with large frequencies and `light items` to denote the items with small frequencies. Figure 5 shows some examples of the insertion process.

In this task, to increase the property of an existing item, we increment its frequency by 1. As shown in Figure 5(a), when inserting item e_2 and e_4, we find they exist in the framework. Therefore, their frequencies are incremented by 1.

In this task, the initial frequency of an item is 1. As shown in Figure 5(a), when inserting item e_6, we find that it is a new item. Therefore, we insert $(e_6, 1)$ into $B[1]$.

In this task, when we need to update the locally lightest item for a new incoming item, there are three strategies: 1) *increment*

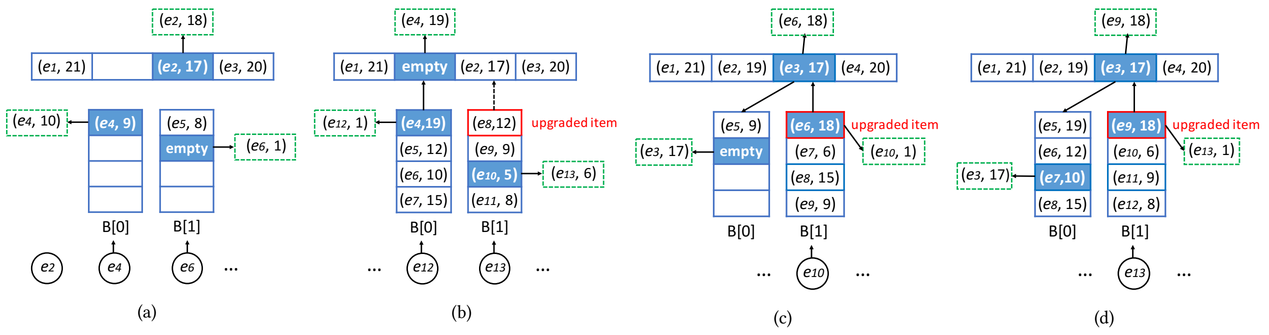


Figure 5: Examples of finding frequent items using Fair Election framework.

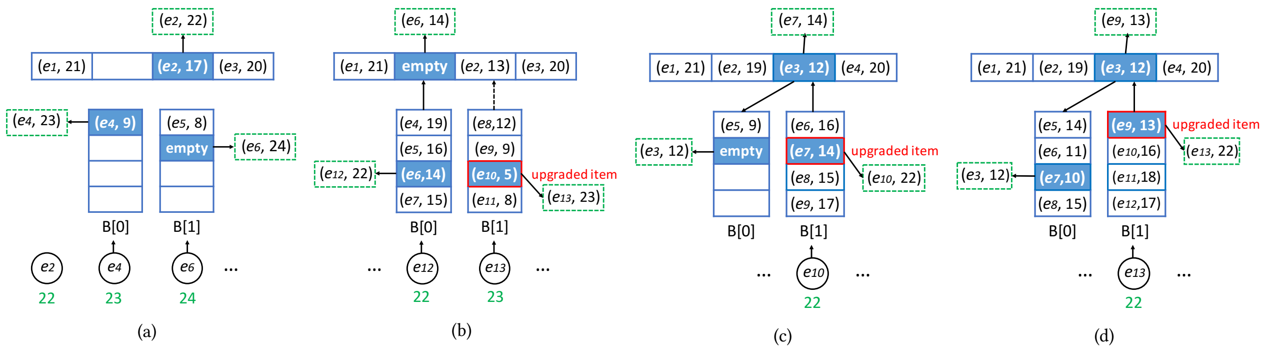


Figure 6: Examples of caching recent items using our Fair Election framework. The numbers in green are the arrival time of each item.

[13]: increment its frequency by 1 and replace the locally lightest item with the new item; 2) *probabilistic replacement*[15]: replace the locally lightest item with probability $\frac{1}{\hat{f}_{min}+1}$, where $\hat{f}_{min}$ is the frequency of the locally lightest item. If it is replaced, its frequency is incremented by 1. Otherwise, the frequency does not change. 3) *decrement:* reduce the frequency by 1. If the updated frequency is 0, insert the new incoming item into the cell of the locally lightest item and set the frequency to 1. As shown in Figure 5(b), when inserting item e_{13}, we find that the hashed bucket $B[1]$ is full and the heaviest item in $B[1]$ is smaller than the lightest item in the coordinator. Therefore, we need to update the lightest item e_{10} in $B[1]$. Using the increment strategy, e_{10} is replaced by e_{13} and the frequency is updated to 6.

In this task, we choose the locally heaviest item as the upgraded item in a bucket. As shown in Figure 5(c), when inserting item e_{10}, we find that $B[1]$ is full and try to move e_6 whose frequency is largest in $B[1]$ to the coordinator.

4.2 Caching Recent Items

In caching, LRU is a well-known replacement strategy. It keeps recently used items by evicting the least recently used ones. To find the top-k recently used items, we use the arrival time as the property in our Fair Election framework. For convenience, we use `recent items` to denote the items that arrived recently and `old items` to denote the items that arrived early. We analyze the performance of our approximate LRU algorithm in Section 7.1. Figure 6 shows some examples of the insertion process.

To increase the property of an existing item, we use the new arrival time to replace the old arrival time. As shown in Figure 6(a), when inserting item e_2 and e_4, we find they exist in the bucket array or the coordinator. Therefore, their arrival time is updated to 22 and 23, respectively.

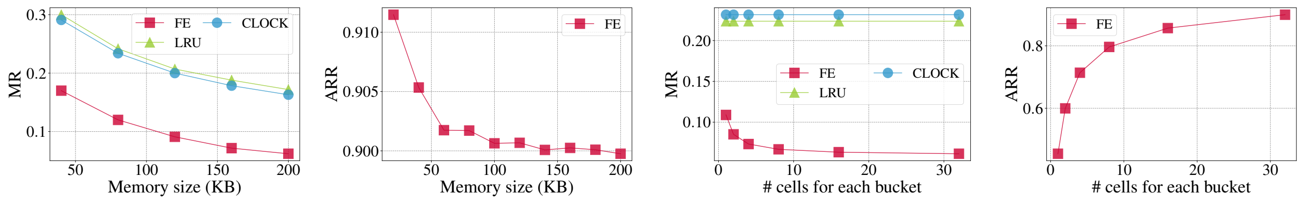
In this task, the initial property of an item is its arrival time. As shown in Figure 6(a), when inserting item e_6, we find that it is a new item arriving at time 24. Therefore, we insert $(e_6, 24)$ into an empty cell of $B[1]$.

When we need to update the locally oldest item for a new incoming item, we replace the locally oldest item with the new item and set the time to the arrival time of the new item. As shown in Figure 6(b), when inserting item e_{13}, we find that the hashed bucket $B[1]$ is full and the oldest item in $B[1]$ is older than all the items in the coordinator. Therefore, the oldest item $(e_{10}, 5)$ in $B[1]$ is replaced by $(e_{13}, 23)$.

We choose the locally oldest item as the upgraded item in a bucket. As shown in Figure 6(c), when inserting item e_{10}, we find that $B[1]$ is full and try to move e_7, who is the oldest in $B[1]$, to the coordinator.

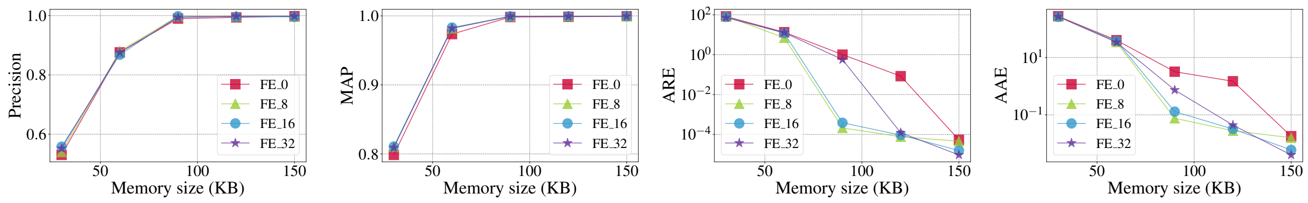
5 EVALUATION

In this section, we conduct experiments to evaluate the performance of our framework on two applications: 1) finding the top-k frequent items, 2) caching recent items. For the first two applications, we show the performance of the original Space-Saving with three strategies (increment, decrement and probabilistic replacement) and Space-Saving using our framework. For convenience, we use *+1, -1, Pro* to denote Space-Saving with the corresponding strategies and *+1_EF, -1_EF, Pro_EF* to denote those using our framework, respectively. For caching, we show the performances of the



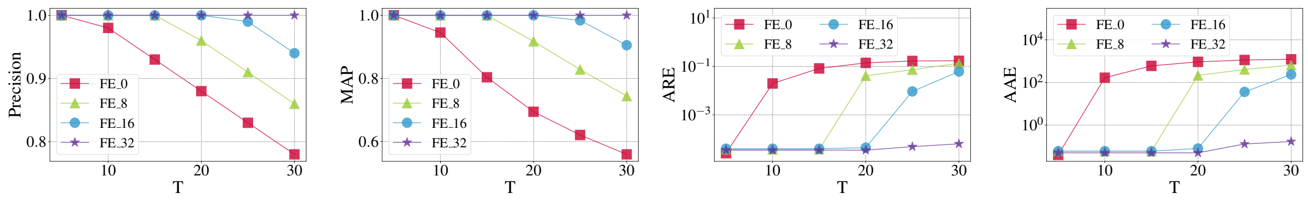
(a) MR *vs.* Memory size (b) ARR *vs.* Memory size (c) MR *vs.* c (d) ARR *vs.* c

Figure 7: Varying memory size and c.



(a) Precision *vs.* Memory size (b) MAP *vs.* Memory size (c) ARE *vs.* Memory size (d) AAE *vs.* Memory size

Figure 8: Varying memory size for finding top-k frequent items (average case).



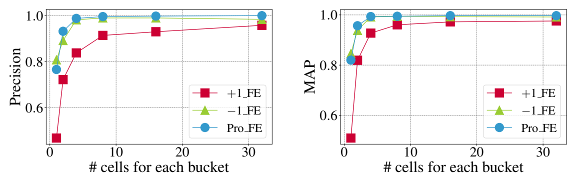
(a) Precision *vs.* T (b) MAP *vs.* T (c) ARE *vs.* T (d) AAE *vs.* T

Figure 9: Varying T for finding top-k frequent items (worst case).

exact LRU, CLOCK and approximate LRU using our Fair Election which is denoted as *FE* for convenience. To achieve a head-to-head comparison, we use the same memory for each algorithm in the experiments. The datasets, implementation, and metrics can be found in section -7.6 in appendix.

5.1 Effect of c

In this set of experiments, we focus on the effect of parameter c, *i.e.*, the number of cells in each bucket.



(a) Precision *vs.* c (top-k) (b) MAP *vs.* c (top-k)

Figure 10: Varying # cells for each bucket.

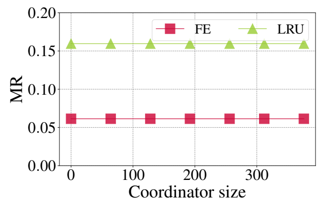
Finding top-k frequent items (Figure 10(a), 10(b)): We set $k = 500$, the memory size to 30KB, and vary c from 1 to 32. The experimental results show that *as c increases, the precision and MAP increase. When c is larger than 8, the performance improves slowly.* The results show that the precision of Space-Saving using our framework with three strategies are improved from 47% to 96%, 81% to 98%, and 77% to 100%, respectively. The MAP is improved from 51% to 97%, 84% to 99%, and 82% to 99%, respectively.

Caching recent items (Figure 7(c),7(d)): We set the memory size to 200KB, the coordinator size to 32, and vary c from 1 to 32. The experimental results show that *the miss rate of FE decreases from 0.11 to 0.06, and the ARR is improved from 0.45 to 0.90.* However, as c increases, the speed of our framework decreases. To obtain a better trade-off between accuracy and speed, we let $c = 8$ for finding top-k frequent items, and let $c = 32$ for caching recent items in the following experiments.

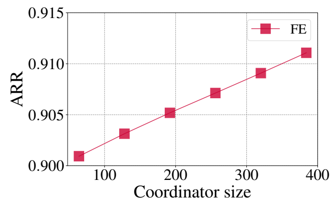
5.2 Effect of Coordinator Size

In this set of experiments, we focus on the effect of the coordinator size, denoted by l. We use *FE_l* to denote our framework with a coordinator of size l. We compare the performance of our framework with different values of l (0, 8, 16, and 32) in the average case. When $l = 0$, it is equivalent to the basic version of the Fair Election framework. We also generate datasets to test the performance of our framework in the worst case.

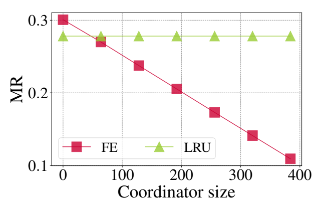
Worst case dataset simulation for finding top-k frequent items: Given a dataset, we generate the worst case dataset as following steps: 1) measure the set of large items in the dataset, 2) choose T instances in the large item set, where T is predefined, and 3) change the hash function $h(.)$ to make all the T instances



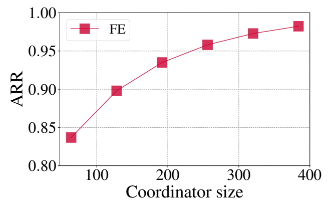
(a) MR *vs.* l (average case)



(b) ARR *vs.* l (average case)

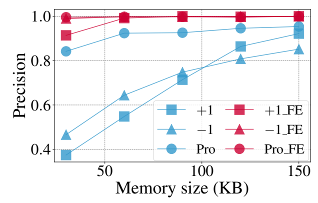


(c) MR *vs.* l (worst case)

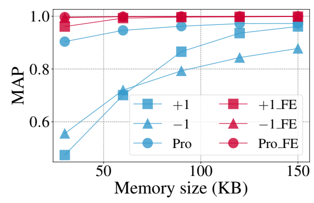


(d) ARR *vs.* l (worst case)

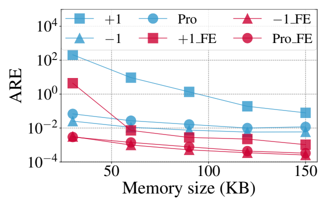
Figure 11: Varying coordinator size.



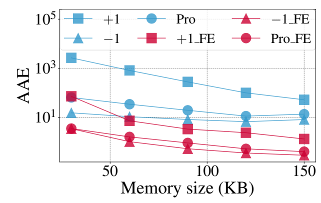
(a) Precision *vs.* Memory size



(b) MAP *vs.* Memory size

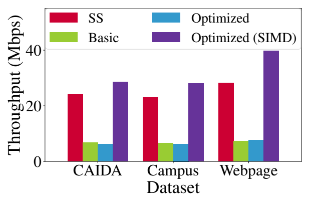


(c) ARE *vs.* Memory size

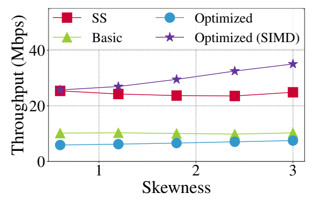


(d) AAE *vs.* Memory size

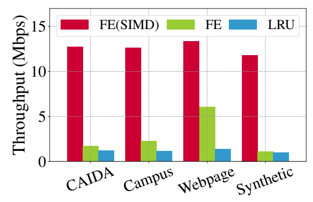
Figure 12: Varying memory size for finding top-k frequent items.



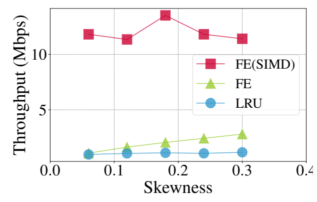
(a) Throughput *vs.* Datasets (top-k)



(b) Throughput *vs.* Skewness (top-k)



(c) Throughput *vs.* Datasets (caching)



(d) Throughput *vs.* Skewness (caching)

Figure 13: Varying datasets.

hashed to the same bucket. In this way, the hashed bucket will easily overflow when T is large.

Worst case dataset simulation for caching recent items: The generated worst case dataset is made as follows: one item appears about half of the total number of times, and the remaining items appear only once.

Finding top-k frequent items (Figure 8(a)-(d), 9(a)-(d)): For the experiments of the average case, we set $k = 500$, and vary the memory size from 30KB to 150KB. For the experiments in the worst case, we set $k = 500$, the memory size to 30KB, and vary T from 5 to 30. The experimental results show that *the performance using different l are similar in the average case. However, in the worst case, the performance significantly improves as the value of l increases.*

As shown in Figure 8(a)-(d), the performances for different values of l are similar in terms of precision, MAP, AAE and ARE. The precisions (Figure 9(a)) for the four values of l increase from 100% to 78%, from 100% to 86%, from 100% to 94%, and from 100% to 100%, respectively. The MAPs (Figure 9(b)) decrease from 100% to 56%, 100% to 74%, 100% to 90%, and 100% to 100%, respectively. The AREs (Figure 9(c)) increase from $2 * 10^{-5}$ to 0.17, $3 * 10^{-5}$ to 0.13, $3 * 10^-5$ to 0.07, and $3 * 10^{-5}$ to $6 * 10^{-5}$, respectively. Finally, the AAEs (Figure 9(d)) increase from 0.04 to 1223, 0.06 to 652, 0.06 to 236, and 0.05 to 0.17, respectively.

Caching recent items (Figure 11(a)-(d)): For the experiments in the average case, we set the memory size to 200KB, $c = 32$ and vary l from 64 to 384. For the experiments in the worst case, we use a smaller cache with a memory size of 4KB, $c = 4$, and vary l from 64 to 384. The experimental results show that *in the average case, the coordinator size has limited effect on the performance of our algorithm. In the worst case on the other hand, the coordinator size has a significant effect.* As shown in Figure 11(a) and 11(b), *the miss rate is 0.06 and the ARR is 0.91.* The miss rate of FE (Figure 11(c)) decreases from 0.30 to 0.11, and the miss rate of exact LRU with the same memory is 0.27. The ARR of FE (Figure 11(d)) increases from 0.84 to 0.98.

5.3 Effect of Memory Size

In this experiment, we compare the algorithms using our framework with the original algorithms under different memory sizes.

Finding top-k frequent items (Figure 12(a)-(d)): We set $k = 500$, and vary the memory size from 30KB to 150KB. Our results show that, *the precision and MAP of the original Space-Saving using our framework increase up to 53% and 43% respectively, and the ARE and AAE decrease by 80 and 100 times, respectively.* As shown in Figure 12(a), the precisions of the original Space-Saving with three update strategies increase from 37% to 92%, 46% to 85%, and 84% to 95%, respectively. When increasing the memory size to 60KB, our framework can achieve 100% precision. The MAPs of the original algorithms (Figure 12(b)) increase from 47$ to 96%, 55% to 87%, 90% to 97%, respectively, while the MAPs of those using our framework are larger than 96% even if the memory size is 30KB. As shown in Figure 12(c), the AREs of the original algorithms are always around 50 times larger than those using our framework. Finally, the AAEs of the original algorithms (Figure 12(d)) are about $5 \sim 100$ times larger than those using our framework.

5.4 Throughput

Finding top-k frequent items (Figure 13(a)-(b)): Since the insertion processes of Space-Saving using the three strategies are similar, we conduct experiments on only one of them. Specifically, we test the Space-Saving using the increment strategy. For convenience, we use *SS* to denote the original algorithm, *Basic* for our basic Fair Election, *Optimized* for our optimized Fair Election, and *Optimized (SIMD)* for our optimized framework with SIMD. As shown in Figure 13(a), the experimental results show that *Optimized (SIMD)* is much faster than the original algorithm on all the datasets, including the synthetic datasets with different skewness (Figure 13(b)).

Caching recent items (Figure 13(c)-(d)): For our approximate LRU algorithm, the experimental results shows that Fair Election using SIMD can achieve a throughput of over 10Mbps on four kinds of datasets. Fair Election without SIMD (Figure 13(c)) and the LRU algorithm have a similar performance, with a throughput around 1Mbps.

6 CONCLUSION

In data stream processing, many top-k tasks, such as finding top-k frequent items and caching recently used items, are useful for various applications. Unfortunately, existing algorithms are either time-consuming or space-consuming. To address this issue, we propose two versions of our Fair Election, a generic framework that works with any property of the items. The basic version improves the memory efficiency, and the optimized version improves the accuracy in the worst case. We apply our framework to find top-k frequent items and cache recently used items. To the best of our knowledge, no prior work can handle these two issues at the same time. Extensive results show that Fair Election achieves much higher accuracy and faster speed. We are confident that our Fair Election can be applied to more algorithms and more top-k tasks.

REFERENCES

[1] Related source codes. https://github.com/FairElection/FairElection/.

[2] Md Shahadat Iqbal, Charisma F Choudhury, Pu Wang, and Marta C González. Development of origin–destination matrices using mobile phone call data. *Transportation Research Part C: Emerging Technologies*, 40:63–74, 2014.

[3] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms*, pages 348–360. Springer, 2002.

[4] Rune Hjelsvold and Roger Midtstraum. Modelling and querying video data. In *VLDB*, volume 94, pages 686–694. Citeseer, 1994.

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

[6] Graham Cormode and Marios Hadjieleftheriou. Methods for finding frequent items in data streams. *The VLDB Journal*, 19(1):3–20, 2010.

[7] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003.

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

[9] Cristian Estan and George Varghese. *New directions in traffic measurement and accounting*, volume 32. ACM, 2002.

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

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

[12] Jiecao Chen and Qin Zhang. Bias-aware sketches. *Proceedings of the VLDB Endowment*, 10(9):961–972, 2017.

[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] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1129–1140. ACM, 2018.

[16] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 741–756. ACM, 2018.

[17] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[18] Surajit Chaudhuri, Gerhard Weikum, et al. Foundations of automated database tuning. In *ICDE*, page 104, 2006.

[19] Nimrod Megiddo and Dharmendra S Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[20] Hassan Ghasemzadeh, Sepideh Sepideh Mazrouee, and Mohammad Reza Kakoee. Modified pseudo lru replacement algorithm. In *null*, pages 368–376. IEEE, 2006.

[21] John T Robinson. Generalized tree-lru replacement. Technical report, Technical Report RC23332, IBM Research Division, 2004.

[22] Grant Wallace and Philip Shilane. Contention-free approximate lru for multi-threaded access, December 27 2016. US Patent 9,529,731.

[23] Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for lru cache performance. In *Teletraffic Congress (ITC 24), 2012 24th International*, pages 1–8. IEEE, 2012.

[24] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

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

[26] Vyas Sekar, Nick G Duffield, Oliver Spatscheck, Jacobus E van der Merwe, and Hui Zhang. Lads: Large-scale automated ddos detection system. In *USENIX Annual Technical Conference, General Track*, pages 171–184, 2006.

[27] Pedro Garcia-Teodoro, J Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1-2):18–28, 2009.

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

[29] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 8. ACM, 2011.

[30] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1449–1463. ACM, 2016.

[31] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 371–384, 2013.

[32] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

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

[34] 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.

[35] Frequent items mining dataset repository. http://fimi.ua.ac.be/data/.

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

7 APPENDIX

7.1 Mathematical Analysis of Approximate LRU using Our Framework

Suppose there are m buckets $(B[0], B[1], ..., B[m-1])$ and each bucket has c cells, where $m * c \gg 1$. Let $B[i, j-1]$ be the j-th cell in $B[i]$. We use the ARR (average relative ranking) defined in Section 7.6 to evaluate the performance. Specifically, we define the *RR* (relative ranking) of an item as follows: for a cache which can store C items, we index items in the cache in the order of their most recent access time: the most recent item got the index *1* and the least recent item got the index C. Then we define $RR(e) = \frac{index}{C}$, where e is an item evicted from a cell in a bucket. Therefore, the item with $r = 1$ is the least recently used item. The ARR shows the similarity between our approximate LRU algorithm and the exact LRU algorithm in the average case. Since the coordinator does not effect the average performance of our algorithm, we analyze the basic version for simplicity.

When the cache is full, items start to be evicted. The new items are randomly distributed to $B[0], B[1], ..., B[m-1]$. Moreover, since $m * c \gg 1$, each $RR(e)$ follows a uniform distribution within the range of $(0, 1]$. Suppose an item is hashed to $B[i]$, our algorithm removes the item with the largest RR in $B[i]$. As a result, we can define a random variable $Y_i = \max\{RR(B[i, j]) \mid 0 \leqslant j < c\}$, representing the relative ranking for the removed item in bucket $B[i]$.

Next, we derive the expectation value and variance of Y_i. When the removed item has a relative ranking smaller than a certain value y, it means that all other items in the same bucket have relative rankings no larger than y. Thus, we have $Pr(Y \leq y) = y^c$. Therefore, we can get the expectation value of Y_i:

$$E[Y_i] = \int_0^1 c * y^c \, dy = \frac{c}{c+1} \tag{5}$$

The result shows that, the more cells a bucket has, the larger relative ranking the removed item has, and the better our algorithm approximates LRU. In order to give an error bound for our approximate LRU, we have to derive the variance of Y_i

$$Var[Y_i] = \int_0^1 (y - \frac{c}{c+1})^2 c * y^{c-1} \, dy = \frac{c}{(c+2)(c+1)^2} \tag{6}$$

Using Chebyshev theorem, we have:

$$Pr(|Y_i - \frac{c}{c+1}| \geq k) \leq \frac{Var[Y_i]}{k^2} \tag{7}$$

For $M \gg 1$, the above equation becomes:

$$Pr(Y_i < 1 - k) < \frac{c}{k^2(c+1)^2(c+2)} \tag{8}$$

The result shows that the error bound is proportional with $\frac{1}{k^2}$ where k is an arbitrary positive number in $(0, 1)$, and the error bound decreases with the increase of the number of cells per bucket in a speed of $O(\frac{1}{c^2})$. Theoretically, when $c = 32$, we have $Pr(Y_i < 1 - k) < \frac{8.64e-04}{k^2}$.

7.2 Mathematical Analysis of Theorem 3.1

Proof. Given a data stream, the hash function randomly maps the items to m buckets. Assume there are k large items in the data stream. For an arbitrary large item and an arbitrary bucket, the probability that this item is hashed to this bucket is $\frac{1}{m}$. Therefore, for any bucket, the number of large items hashed to this bucket, which is denoted by N_l, follows a Binomial distribution $B(k, \frac{1}{m})$. When k is large (*e.g.*, $k > 100$), and $\frac{1}{m}$ is small, N_l follows approximately a Poisson distribution $\pi\left(\frac{k}{m}\right)$. Therefore, the probability that i large items are hashed into an arbitrary bucket is:

$$\begin{aligned} Pr(i) &= Pr\{N_l = i\} \\ &= e^{-k/m} \frac{(k/m)^i}{i!} \end{aligned} \tag{9}$$

□

7.3 Acceleration using SIMD

In this section, we describe how we use SIMD (Single Instruction and Multiple Data) to accelerate the insertion process of Fair Election. There are three processes we can accelerate during insertions. 1). the match processes for items in the coordinator. 2). the match processes for items in the bucket array. 3). the lookup for the smallest item in a bucket. Note that an empty cell in Fair Election has a property of 0. When we want to find an empty cell, we can find the smallest item and check that whether it is 0.

For the first two match processes, instead of matching each item one by one, we do match processes in groups of eight. Specifically, for each group, we use the instruction `_mm256_cmpeq_epi32` for matching. If we find a matched item in a group, we use the instructions `_mm256_movemask_ps` and `_tzcnt_u32` to locate the matched item. For the third process, in each group, we divide the eight items into two sub-groups (G_1 and G_2), each of which has four items. Then, we use the instruction `_mm_min_epi32` to compare each 32-bit value in G_1 and G_2. We use the instruction `_mm_shuffle_epi32` to repeat the process iteratively and find the smallest one in the eight items. The detailed source code of Fair Election using SIMD can be found on Github [1].

7.4 Datasets

CAIDA: This dataset is from the *CAIDA Anonymized Internet Trace 2016* [33], and consists of 5-tuples, *i.e.*, source IP address, destination address, source port, destination port, and protocol type. We use the source IP addresses to identify items and choose the first 10M packets as our CAIDA dataset.

Campus: This dataset is from our campus. Similar to the CAIDA dataset, it consists of 5-tuples. We still use the source IP addresses to identify items and choose the first 10M packets as our Campus dataset.

Synthetic: According to the Zipf distribution [34], we generate 5 different datasets with different skewness (from 0.6 to 3.0). Each dataset has 10M items. The default skewness of the Synthetic dataset is 0.6.

Webpage: This dataset is obtained from the website [35], which is crawled from a collection of web pages. The items in the dataset record the number of distinct terms in a web page. Again, we use the source IP addresses to identify items and choose the first 10M items as our Webpage dataset.

7.5 Implementation

We implement the original Space-Saving with different strategies, Space-Saving using our framework, the exact LRU, the CLOCK and the approximate LRU using our framework in C++. Here we use the Bobhash recommended in [36] as the default hash function. All the programs run on a server with dual 6-core CPUs (24 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62GB total system memory. The source code of the mentioned algorithms is available on Github [1].

7.6 Metrics for Caching Recent Items

Given a data stream $\mathcal{S}$, let χ be the set of items that trigger a cache miss. Let ω be the set of items which are evicted by new incoming items. For each item in the cache, we index them in the order of their most recently arrival time. The most recently accessed one gets the index of 1 and the least recently accessed one gets the index of s_e, where s_e is the total number of items in the cache. Let r_e be the index of e when it is evicted. We define the following metrics:

Miss Rate (MR): It is defined as $\frac{|\chi|}{|\mathcal{S}|}$, the ratio of items that triggered a cache miss.

Average Relative Ranking (ARR): It is defined as $\frac{1}{|\omega|} * \sum_{e \in |\omega|} \frac{r_e}{s_e}$, which reflects the extent to which the algorithm approximates the exact LRU. If the ARR of an algorithm equals 1, it is an exact LRU.