

# Finding Frequent Items in Time Decayed Data Streams

Shanshan Wu<sup>1</sup>, Huaizhong Lin<sup>1(✉)</sup>, Leong Hou U<sup>2</sup>,  
Yunjun Gao<sup>1</sup>, and Dongming Lu<sup>1</sup>

<sup>1</sup> College of Computer Science and Technology,  
Zhejiang University, Hangzhou, China  
{wuss,linhz,gaoyj,ldm}@zju.edu.cn

<sup>2</sup> Department of Computer and Information Science,  
University of Macau, Macau, China  
ryanlhu@umac.mo

**Abstract.** Identifying frequently occurring items is a basic building block in many data stream applications. A great deal of work for efficiently identifying frequent items has been studied on the landmark and sliding window models. In this work, we revisit this problem on a new streaming model based on time decay, where the importance of every arrival item is decreased over the time. To address the importance changes over the time, we propose a new heap structure, named Quasi-heap, which maintains the item order using a lazy update mechanism. Two approximation algorithms, Space Saving with Quasi-heap (SSQ) and Filtered Space Saving with Quasi-heap (FSSQ), are proposed to find the frequently occurring items based on the Quasi-heap structure. Extensive experiments demonstrate the superiority of proposed algorithms in terms of both efficiency (i.e., response time) and effectiveness (i.e., accuracy).

## 1 Introduction

A data stream is a massive unbounded sequence of items continuously received at a rapid rate and it appears in a variety of applications, such as network monitoring, financial monitoring, web logging, etc. Substantial analytical studies have been devoted to the data streams, such as clustering [6], classification [17], and mining frequent patterns [3]. Finding frequent items [8, 11, 14, 16, 18] has received considerable attentions in the data stream analytical tasks. This problem has been served as an important building block for different data stream mining problems, such as mining frequent itemsets [3] and computing the entropy of a data stream [2].

In typical data stream scenarios, the item arrival rate is very high so that not every received item can be kept in the main memory. Thereby, the solutions normally scan every arrival item once (i.e., sequential access) and drop unpromising items (e.g., less frequently occurring items) when the main memory becomes full. Complying with these constraints, the prior studies mostly focus on how to answer the data stream problem approximately with an error bound.

Early solutions [8, 10, 11, 14, 16] of this problem are developed based on two traditional streaming models, the landmark (i.e., the frequent items can be any item in the entire stream) and the sliding window (i.e., the frequent items can only be the items of the current window). While the landmark model preserves better data completeness, it ignores the importance of newly arrival items. On the other hand, the sliding window model partially addresses the item freshness but the items not occurring in the current window are completely ignored.

To address these, finding the frequent items in *time decayed* data streams has received substantial attention over the past decade [4, 5, 9, 12, 15, 19]. Under the *time decay* model, the weight of a received item is decreased over the time and the frequent items are then computed based on the time decayed weights. This model preserves better completeness (i.e., every item is considered) and item freshness (i.e., the recent items are more important) than the prior streaming models.

In the landmark and the sliding window models, the count of an item always increases by one when the item comes from the stream. Hence, given an ordered list of the frequent items (maintained by a linked list or a heap structure), we can easily maintain the order consistency by swapping the affected item with its neighbor items. Under the time decay model, the weight of every item is updated over the time subject to the decay function. The order maintenance may become costly since the updated item may swap with multiple items in the ordered list.

To address this challenge, we propose a new heap structure, named Quasi-heap, which maintains the order of the items in a heap by a lazy manner. Based on the Quasi-heap, two approximation algorithms are studied to solve the frequent item problem on time decayed data streams. We briefly list our main contributions as follows.

- We propose a new heap structure, named Quasi-heap, to maintain the frequent items based on their time decayed counts.
- We invent two approximation algorithms, Space Saving with Quasi-heap (based on Space Saving [16]) and Filtered Space Saving with Quasi-heap (based on Filtered Space Saving [10]). Our improved algorithms answer the frequent item problem with reasonable memory overhead and a guaranteed error bound.
- Extensive experiments are conducted to demonstrate the superiority of our algorithms in terms of the running time and the estimation accuracy.

The remainder of this paper is organized as follows. A survey of related work is presented in Sect. 2. Section 3 formulates the frequent item problem in time decayed streams. Section 4 discusses and analyzes the proposed Quasi-heap. Sections 5 and 6 depict two improved algorithms SSQ and FSSQ, respectively. Section 7 evaluates the proposed algorithms, and we conclude this paper in Sect. 8.

## 2 Related Work

### 2.1 Landmark Model and Sliding Window Model

In the landmark and the sliding window models, there are a great deal of work proposed to find the frequent items from a data stream. These work can be classified into two main categories [7], counter-based and sketch-based. The counter-based algorithms are deterministic, which only monitor a subset of the items from a data stream. These algorithms maintain a set of counters to track the frequent items over the subset. Space Saving (SS) [16], Lossy Counting [14], and Frequent [11] are the representative ones in this category. The sketch-based algorithms which use a set of array counters to estimate the frequencies of the items. Different from the counter-based algorithms, each item is projected into a set of corresponding sketches by some hash functions. The frequency of an item is estimated from the counters of its corresponding sketches. To minimize the collision probability of the hash functions, we can increase the granularity of the sketch (i.e., more counters are used). However, this will lead to huge memory consumption. CountSketch [2], Count-Min Sketch [8], and Filtered Space Saving (FSS) [10] are the representative algorithms in this category.

However, these work either treat the stale and the fresh data the same (i.e., the landmark model) or remove the stale item by a subjective window length (i.e., the sliding window model). In real world applications, it is more desirable if the frequent item problem not only considers every arrival item but also treats the fresh items more important than the stale items.

### 2.2 Time Decay Model

Finding frequent items in a time decayed data stream has received substantial attention from the community recently [4, 5, 9, 12, 15, 19]. Zhang et al. [19] proposed two  $\epsilon$ -approximation algorithms called Frequent-Estimating (FE) and FE with Heap (FEH). FE updates the frequent item result for an item arrival in  $O(\epsilon^{-1})$  time by a linked list and FEH updates the result in  $O(\log \epsilon^{-1})$  by a heap structure. Chen et al. [5] proposed another  $\epsilon$ -approximation algorithm, called Frequent-item Counting (FC), which takes  $O(1)$  time to maintain the answer for each arrival item by a hash function. Mei and Chen [15] proposed to estimate the frequencies of items by multiple hash functions. However, their work did not give the analysis of the memory consumption and the estimation accuracy.

Recent developments attempted to improve the estimation accuracy by either exploiting the decay function or employing a new data structure. Lim et al. [12] proposed a new  $\epsilon$ -approximation algorithm, TwMinSwap, which takes  $O(\epsilon^{-1})$  time to process each arrival item. The basic idea is to drop the minimum item (with the smallest counter) when the memory becomes full, where the counters are updated over the time by multiplying the decay rate.  $\lambda$ -HCount algorithm [4] employs a double linked list to record the frequent items and improves the frequency estimation accuracy by multiple hash functions. The items monitored in the double linked list are arranged in the descending order of their recently

updated time. Since the items are organized in a double linked queue structure, the algorithm can reallocate an item entry to the end of the list in  $O(1)$  time.

All the above algorithms are based on a backward decay function where the item importance is decreased over the time. The main challenge under the backward decay function is that the weights of the existing items are constantly changed. To address this, Cormode et al. [9] studied an alternative decay function that is a monotone increasing function to the *age* of an item (i.e., the subtraction of the arrival time and the origin time). In this model, the item weight is fixed when the *age* of an item is decided. In other words, the weights of the existing items become stable and the problem of finding the frequent item becomes easier. However, the *forward* weight of an item will become very large (due to the *age*) if the system has been running for a long time. One possible solution is to reset the origin time periodically but it needs extra effort to recompute the frequent items. The effectiveness of this model on the frequent item problem is unknown.

For clarity, in this work we focus on finding the frequent items based on the backward decay function as it is widely adopted in the prior studies.

### 3 Definitions and Preliminaries

Table 1 summarizes the notation to be used in this paper. We use a standard stream model with discrete timestamps and only one item arrives at every timestamp. The current data stream  $D_n = \langle I_1, I_2, \dots, I_n \rangle$ , where  $I_t \in \{a_1, \dots, a_D\}$ .

**Table 1.** Summary of notation

Notation	Description	Notation	Description
$D_n$	The data stream up to time $n$	$I_t$	The received item at time $t$
$D$	The number of distinct items	$\{a_1, \dots, a_D\}$	The set of distinct items
$\phi$	Frequency threshold	$\epsilon$	Error tolerance parameter
$m$	The length of the monitored list	$\tau$	Time decay rate
$C_t(a_i)$	The decayed count of $a_i$ at time $t$	$c_t(a_i)$	The estimate value of $C_t(a_i)$

While processing a long stream, it is reasonable to treat a recent item more important than an old item. In this work we adopt a backward time decay model that is used to gradually decrease the effect of the obsolete items.

**Definition 1 (Decayed count of an item,  $C_n(a_i)$  [3]).** Given a time decay rate  $\tau$  ( $0 < \tau \leq 1$ ),  $C_n(a_i)$  is the decayed count of an item  $a_i$  at time  $n$ , i.e.,

$$C_n(a_i) = C_{n-1}(a_i) \times \tau + W_n(a_i) \quad (1)$$

where  $C_1(a_i) = W_1(a_i)$  and  $W_k(a_i)$  is a function that indicates the arrival of an item  $a_i$  at the timestamp  $k$ . Specifically, if  $I_k = a_i$ ,  $W_k(a_i) = 1$ ; otherwise,  $W_k(a_i) = 0$ .

Definition 2 defines the frequent item in this work subject to a frequency threshold  $\phi$ . Specifically, an item  $a_i$  is in the result set if and only if its *normalized* decayed count is higher than  $\phi$  and the normalization factor is the sum of all items  $|D_n|$  ( $= \frac{1-\tau^n}{1-\tau}$ , which approaches to  $\frac{1}{1-\tau}$  when  $n \rightarrow \infty$ ).

**Definition 2 ( $\phi$ -frequent item).** Given a stream  $D_n$ , and a frequency threshold  $\phi$ ,  $0 < \phi \leq 1$ . If the decayed count  $C_n(a_i)$  is higher than  $\phi \times |D_n|$  ( $|D_n|$  is the sum of decayed counts of all items in  $D_n$ ), then  $a_i$  is a  $\phi$ -frequent item.

We need huge memory to maintain the  $\phi$ -frequent items exactly in a long running data stream, where the space complexity is  $\Omega(D)$  when the number of distinct items in the stream is  $D$  [7]. Thereby, the typical solutions focus on answering this problem approximately subject to an error bound  $\epsilon$ . Formally, the approximation version of the problem is defined as follows.

*Problem 1. ( $\epsilon$ -Approximate Decayed Frequent items).* Given a data stream  $D_n$ , the  $\epsilon$ -approximate frequent item set contains all items where their decayed counts are higher than  $(\phi - \epsilon) \times |D_n|$ .

## 4 Quasi-heap

To find the frequent items in a data stream, a group of counters is used to record the candidate items. We assume that there are  $m$  counters available (subject to the memory budget) and these counters are organized into a linked list or a heap structure. To address the memory budget, the prior studies [16] replace the smallest item by the newly arrival item when the memory becomes full.

According to Definition 1, the decayed count of an item  $a_i$  is increased when  $a_i$  is received from the stream. Thereby, the position of  $a_i$  in the counters should be changed in order to keep the order correct. The complexity of each update takes  $O(m)$  (for the linked list) and  $O(\log m)$  (for the heap structure), which is definitely too time consuming in data stream environments. Hence, we propose a new data structure called Quasi-heap which aims at postponing their sorting operations when the decayed count of an existing item is increased. In other words, we allow certain order inconsistency in the Quasi-heap structure. An example of the Quasi-heap is given in the following Example 1.

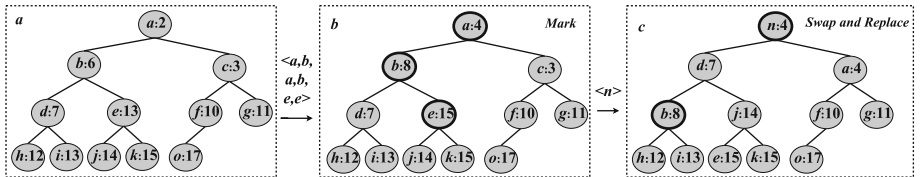


Fig. 1. A running example of Quasi-heap

*Example 1. (An example of Quasi-heap).* In Fig. 1, each node consists of the item name and its decayed count. We adopt  $\tau = 1$  for ease of presentation. Figure 1a shows a Quasi-heap that contains 12 items where the order is identical to that of the ordinary heap. Upon receiving the next sequence  $\langle a, b, a, b, e, e \rangle$ , the corresponding counts of  $a, b, e$  are increased. Instead of running heapify to maintain the heap structure, we only mark these items as *delayed* (e.g., these items marked by thick lines in Fig. 1b) since they are the old items in the Quasi-heap. While receiving a new item  $n$ , we start to run the heapify partially to those *delayed* nodes starting from the root. After the heapify process,  $c$  becomes the root node as it is the smallest item in the Quasi-heap. And then,  $c$  is replaced by  $n$  (cf. Fig. 1c) where the *estimated count* of  $n$  is set to 4 (i.e., the count of  $c + 1$ ) as followed the suggestion of other counter-based algorithms [16].

According to the discussion in Example 1, the main operation, *delayedSorting*, is to execute heapify partially on the Quasi-heap so that the minimum node can be properly identified and removed.

---

**Algorithm 1.** *delayedSorting*


---

```

1:  $c.error \leftarrow c.error \times \tau^{(t-c.ut)}$ ;  $c.cnt \leftarrow c.cnt \times \tau^{(t-c.ut)}$ ;  $c.ut \leftarrow t$ ;
2: if  $c.delay = 0$  or  $c$  is a leaf node then return;
3: delayedSorting (each child counter of  $c$ );
4: if  $c.cnt > sml.cnt$  then  $\triangleright sml$  is the smaller of the two child counters
5:   swap  $c$  and  $sml$  and  $sml.delay \leftarrow 1$ ;
6: if  $c.cnt = sml.cnt$  then
7:   if  $sml.error > c.error$  then
8:     swap  $c$  and  $sml$  and  $sml.delay \leftarrow 1$ ;
9:  $c.delay = 0$ ;

```

---

Algorithm 1 describes the *delayedSorting* operation in detail. The information of an item is updated in line 1. If the delayed flag of an item is not marked, in line 2, then its count must be the minimum count in its subtree according to Lemma 1 (being discussed shortly). If the delayed flag of an item is marked, the order of this item may be inaccurate so that we need to execute the *delayedSorting* operation on its each child (line 3). After the recursive calls, if the count of the root is larger than that of its children, we swap the root with its child in lines 4–5. If the counts are identical, we swap the root with its child only when the child has larger estimated error than  $c$  (i.e., the estimated error is decided when the node is inserted into the Quasi-heap, cf. line 7 of Algorithm 2 and Example 1).

**Lemma 1.** *Let  $p$  and  $q$  be two counters in a Quasi-heap and  $p$  is an ancestor of  $q$ . If  $p.delay = 0$ , then the decayed count of  $p$  is no larger than  $q$ .*

*Proof.* When the Quasi-heap is not full, the Quasi-heap is identical to an ordinary heap so that  $p.cnt \leq q.cnt$  due to the heapify.

When the Quasi-heap becomes full, there are two cases. If  $p$  is not received from the data stream again, then  $p.cnt \leq q.cnt$  is still held no matter whether  $q$

has been updated or not due to the monotonicity of the decayed count function (cf. Eq. 1). If  $p$  has been received from the data stream again, the *delay* flag of  $p$  must be set to 1. The *delay* flag is reset to 0 only when the subtree of  $p$  is refined by the heapify (cf. Algorithm 1) so that  $p.cnt \leq q.cnt$  is still held.  $\square$

**Time Complexity Analysis:** When the newly arrival item is in the Quasi-heap, we only update the decayed count of this item and mark the delay flag. Hence, processing an existing item takes  $O(1)$  time. When the newly arrival item is not in the Quasi-heap, we replace the minimum item of the Quasi-heap by this new item. To find the minimum item in the Quasi-heap, we execute Algorithm 1 to ensure the correctness of the order. The cost of Algorithm 1 is  $O(m)$  as it may traverse the entire tree in the worst case. However, this case is very rare to happen in real datasets. In addition, a frequent item is likely kept in the Quasi-heap (as their counts are high) and it is more frequently received from the data stream than other items. The response time of processing the existing items is dramatically reduced from  $O(\log m)$  to  $O(1)$ . In our experiments, the Quasi-heap can reduce response time up to 80 % as compared with the ordinary heap.

## 5 Space Saving Algorithm with Quasi-heap

We study a counter-based algorithm, SSQ (that is based on the SS algorithm [16]), to find the  $\epsilon$ -approximate decayed frequent items. Algorithm 2 depicts the SSQ in detail. If the new arrival item  $c$  is already in the Quasi-heap (lines 2–3), we update its statistics and mark the *delay* flag as 1. Otherwise, we first check whether the Quasi-heap is full or not. If the Quasi-heap is not full (lines 9–11), we simply execute heapify to maintain the consistence of the Quasi-heap. Otherwise, we run the *delayedSorting* from the root of the Quasi-heap and replace the *refined* root by the new item  $c$ . Similar to the SS algorithm, the estimated count of a new item  $c$  is derived from the count of the removal item  $r$ .

---

### Algorithm 2. SSQ Algorithm

---

```

1: for each coming item  $c$  at timestamp  $t$  do
2:   if  $c$  is tracked in Quasi-heap then
3:      $c.error \leftarrow c.error \times \tau^{t-c.ut}$ ;  $c.cnt \leftarrow c.cnt \times \tau^{t-c.ut} + 1$ ;  $c.ut \leftarrow t$ ;  $c.dealy \leftarrow 1$ 
4:   if  $c$  is not tracked in Quasi-heap then
5:     if Quasi-heap is full then
6:        $delayedSorting(r)$   $\triangleright r$  is the root of Quasi-heap
7:        $c.error \leftarrow r.cnt \times \tau^{t-r.ut}$ ;  $c.cnt \leftarrow r.cnt \times \tau^{t-r.ut} + 1$ ;  $c.ut \leftarrow t$ ;
8:        $c.dealy \leftarrow 1$ ; and replace  $r$  by  $c$ 
9:     if Quasi-heap is not full then
10:      create a new counter  $c$ ;  $c.error \leftarrow 0$ ;  $c.cnt \leftarrow 1$ ;  $c.ut \leftarrow t$ ;  $c.dealy \leftarrow 0$ 
11:      insert and maintain  $c$  in the Quasi-heap

```

---

We present two properties of SSQ algorithm, which are based on the properties proposed in SS [16] and FE [19] with some minor modifications.

**Lemma 2.** *Among all  $m$  counters, the minimum counter value  $\mu \leq \frac{1-\tau^n}{m(1-\tau)}$ .*

*Proof.* The sum of estimated counts of all  $m$  items in the monitored list is equal to the sum of decayed counts of all  $n$  items in the data stream, i.e.,  $\sum_i c_n(a_i) = \frac{1-\tau^n}{1-\tau}$ . It is following that  $m\mu \leq \sum_i c_n(a_i)$ , so  $\mu \leq \frac{1-\tau^n}{m(1-\tau)}$ .  $\square$

Based on the Lemma 2, SSQ can use confined space (i.e.,  $m = \lceil 1/\epsilon \rceil$ ) to find  $\epsilon$ -approximate frequent items by securing the error ratio at most  $\epsilon \times |D_n|$  [16].

**Theorem 1 (No False Negative).** *For any item  $a_i$  with the decayed count  $C_n(a_i)$  greater than  $\mu$  is present in the Quasi-heap.*

*Proof.* We prove the theorem by contradiction. Assume at the current time  $t$ , an item  $a_i$  with decayed count  $C_t(a_i) > \mu$  is not in the Quasi-heap. Then,  $a_i$  must be evicted sometime in the past. Suppose  $a_i$  was last evicted at time unit  $t'$ , then its decayed count  $C_{t'}(a_i) = C_t(a_i)/\tau^{t-t'}$ , which is larger than  $\mu/\tau^{t-t'}$  (based on the condition  $C_t(a_i) > \mu$ ). Let  $\mu_{t'}$  be the minimal count at  $t'$ , then  $\mu_{t'} \leq \mu/\tau^{t-t'}$ . We can get  $C_{t'}(a_i) = C_t(a_i)/\tau^{t-t'} > \mu/\tau^{t-t'} \geq \mu_{t'}$ . So, clearly,  $C_{t'}(a_i) \geq \mu_{t'}$ . This fact means that the estimated count of the item  $a_i$  was greater than the minimum counter value when it was evicted at time unit  $t'$ . This contradicts the fact that the SSQ algorithm evicts the item with the minimum counter value.  $\square$

## 6 Filtered Space Saving Algorithm with Quasi-Heap

In this section, we propose a sketch-based algorithm, FSS with Quasi-Heap (FSSQ) (that is based on the FSS algorithm [10]), to find the frequent items. The FSSQ algorithm employs two data structures, (1) the Quasi-heap and (2) a sketch (i.e., a two-dimensional array with width  $w$  and depth  $d$ ). The sketch used in this work is similar to that of the Count-Min algorithm [7]. Each entry of the sketch is composed of an estimated count and the time of last update, denoted as  $(cnt, ut)$ . To update the value of the sketch entries, we need  $d$  pairwise-independent hash functions:  $h_1, \dots, h_d : \{1, 2, \dots, D\} \rightarrow \{1, 2, \dots, w\}$ . FSSQ improves the estimation accuracy since the count of a new item is estimated by  $d$  sketch entries instead of the minimum item in the Quasi-heap (cf. SSQ).

Algorithm 3 depicts the FSSQ, whose idea is similar to that of Algorithm 2 except the situation that a new item is not tracked in Quasi-heap and Quasi-heap becomes full, hence we omit the similar parts. We first run *delayedSorting* from the root (line 4) in order to find the minimum item. Next we update the corresponding sketch entries by  $c$  (lines 6–7), and estimate its minimum value among  $d$  corresponding sketch entries (line 8). If the estimated minimum count is larger than the root of Quasi-heap, then we replace the root by the new item  $c$  (lines 12–13) and update the corresponding sketch entries by the evicted item  $r$  (lines 10–11).

The properties of the FSSQ are given in Lemmas 3–5 and Theorem 2.



**Algorithm 3.** *FSSQ* Algorithm

---

```

1: for each coming item  $c$  at timestamp  $t$  do
2:   if  $c$  is not tracked in the Quasi-heap then
3:     if Quasi-heap is full then
4:        $delayedSorting(r)$   $\triangleright r$  is the root of Quasi-heap
5:       Let  $s[x, y]$  be the counter in the sketch entry  $(x, y)$ ;
6:       for  $j = 1, \dots, d$  do  $\triangleright d$  is the depth of the sketch
7:          $s[j, h_j(c)].cnt \leftarrow s[j, h_j(c)].cnt \times \tau^{t-s[j, h_j(c)].ut} + 1$ ;  $s[j, h_j(c)].ut \leftarrow t$ ;
8:        $est \leftarrow \min_{1 \leq j \leq d} s[j, h_j(c)].cnt$ ;
9:       if  $est > r.cnt$  then
10:        for  $j = 1, \dots, d$  do
11:           $s[j, h_j(r)].cnt \leftarrow r.cnt \times \tau^{t-r.ut}$ ;  $s[j, h_j(r)].ut \leftarrow t$ ;
12:           $c.error \leftarrow r.cnt \times \tau^{t-r.ut}$ ;  $c.cnt \leftarrow r.cnt \times \tau^{t-r.ut} + 1$ ;  $c.ut \leftarrow t$ ;
13:           $c.dealy \leftarrow 1$ ; and replace  $r$  by  $c$ 

```

---

**Lemma 3.** *At any moment, for each item  $a_i$ , the minimum count  $\mu$  in the Quasi-heap is no less than the minimum entry that the hash function values of  $a_i$  associates, i.e.,  $\mu \geq \min_{1 \leq j \leq d} s[j, h_j(a_i)]$ .*

*Proof.* In the initialization phase, all the hits are reflected in Quasi-heap, and all the entries in sketch have value of 0. Hence, the conclusion is trivially true.

When the Quasi-heap has been full. For a new coming item, if it is being monitored in the Quasi-heap, its counter in the Quasi-heap increases and all the entries in the sketch remain unchanged. If a new coming item is not being monitored in the Quasi-heap, the increment of the entry in the sketch may lead to the situation that the minimum entries become larger than  $\mu$ . However, at this case, a replacement is taken place. Hence, the conclusion is still true.  $\square$

**Lemma 4.** *For any item in the Quasi-heap, its overestimated error is no greater than  $\mu$ .*

*Proof.* For any item  $a_i$  in the Quasi-heap, its maximum overestimated error is always assigned the minimum decayed count in the entries that  $a_i$  associates, when a replacement takes place. This value is no greater than  $\mu$  according to Lemma 3, so the maximum overestimated error is no greater than  $\mu$ .  $\square$

**Lemma 5.** *Assume  $w = \lceil e/\epsilon \rceil$  and  $d = \lceil \ln(1/\delta) \rceil$ , in which  $e$  is the Euler's constant, i.e., the base of natural logarithms. For any item in the Quasi-heap, its count error is no greater than  $\epsilon/(1 - \tau)$  with probability at least  $1 - \delta$ .*

*Proof.* We omit the proof due to the space limit.

**Theorem 2.** *For any item with  $C_n(a_i) > \mu$  is present in the Quasi-heap.*

*Proof.* We first prove that  $a_i$  is present in the Quasi-heap. If the last arrival time of  $a_i$  is  $t_1$  time unit ago, then the estimated count of  $a_i$  in the sketch  $t_1$  time units ago is no less than  $C_n(a_i)/\tau^{t_1}$ . Thereby,  $\min_{1 \leq j \leq d} s[j, h_j(a_i)]/\tau^{t_1} \geq C_n(a_i)\tau^{t_1} > \mu/\tau^{t_1}$ . This means that  $a_i$  was inserted in the monitored counters  $t_1$  time units ago. We also need to show there is no false negative result. The proof is identical to that of Theorem 1 if  $a_i$  is present in the Quasi-heap.  $\square$

## 7 Experimental Study

In this section, we empirically evaluated the efficiency and effectiveness of SSQ and FSSQ using both real and synthetic datasets. We compared proposed solutions with the state-of-the-art solutions, TwMinSwap [12] (counter-based) and  $\lambda$ -HCount [4] (sketch-based). All methods were implemented in C++ and compiled using the Microsoft Visual Studio 2012 compiler. All experiments were conducted on a 3.20 GHz Pentium PC machine with 8 GB main memory running Windows 7 Professional Edition.

### 7.1 Experimental Settings

The synthetic datasets are generated based on Zipfian distributions. The Zipfian parameter value is from 0.8 to 2 (the default value is 1.0). We also used two real datasets widely evaluated in the data stream research [13]. The Kosarak is an anonymized click-stream on a Hungarian online news portal<sup>1</sup>. The Retail contains retail market basket data from an anonymous Belgian store [1]. In our experiments, we consider every single item in sequential order. Kosarak (Retail) is of all the items 8019015 (908576) and distinct items 41270 (16470).

We verify the efficiency with respect to **Time** (Each algorithm is run for 20 times and the average response time is reported), **Precision** (The fraction of the items identified by the algorithm that are actually frequent), and **Recall** (All the frequent items are detected due to the overly estimated count (cf. Theorem 1)).

### 7.2 Experimental Results

To verify the scalability, we varied one parameter in each set of experiments while setting other parameters to their default values.  $\phi$  is from 0.0001 to 0.01 (the default value is 0.001),  $m$  is from 1000 to 4000 (the default value is 1000) and  $\tau$  is from 0.97 to 1.00 (the default value is 0.995).

**Performance Overview:** In terms of the response time, SSQ yields better performance than state-of-art solutions, due to the Quasi-heap data structure proposed which can save the response time by up to 80%. In terms of the precision, FSSQ performs the best among all methods due to the sketch structure which can get a better bound on the estimation count error.

Figure 2 shows the response time and the precision by varying the cardinality of the items. Notably, the response time of all methods increases as  $n$  becomes larger. We find that the counter-based algorithms are faster than the sketch-based algorithms; however, the counter-based algorithms is less accurate than the sketch-based algorithms as discussed in Sect. 6.

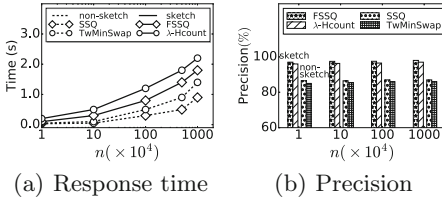
It is obvious that the response time decreases when the data becomes more skewed (cf. Fig. 3(a)). The cost of processing an existing item in the Quasi-heap is less than that of processing a new item in the Quasi-heap. The probability of receiving an existing item becomes higher when the data is more skewed.

<sup>1</sup> Frequent Itemset Mining Dataset Repository <http://fimi.cs.helsinki.fi/data/>.

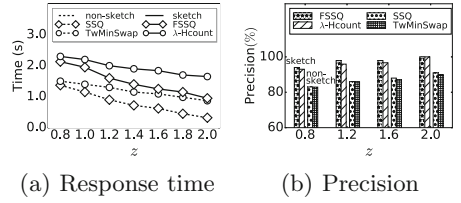
Figures 4, 5 and 6 show the experiments conducted  $\phi$ ,  $\tau$ , and memory, respectively. Due to the space limit, each figure only reports one set of experiments as all methods perform similarly on three datasets. For instance, SSQ is 35 %–40 % faster than TwMinSwap on average for  $\phi$  and  $\tau$  on all three datasets. Figure 6 shows that the precision increases when we have more space.

### 7.3 Effectiveness of Quasi-heap

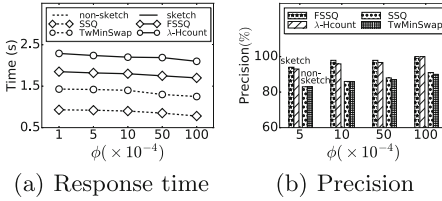
We also performed an experiment to investigate the advantage of the Quasi-heap (SSQ) as compared to the ordinary heap (SS). We reported the number of comparisons performed in the heap and the response time. Figure 7(a) shows that the performance between the Quasi-heap and the ordinary heap. For example, when the stream size is  $10^7$  and the data skew is 3.0, the number of comparisons in the Quasi-heap is  $2k$  which is 118 times smaller than  $236k$  in the ordinary heap. From Fig. 7(b), we can find that the Quasi-heap reduces the response time by up to 80 % since huge amount of unpromising comparisons are delayed by the *delaySorting* method.



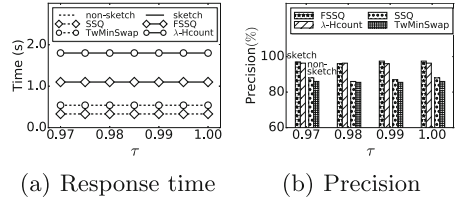
**Fig. 2.** Effect of  $n$  on synthetic



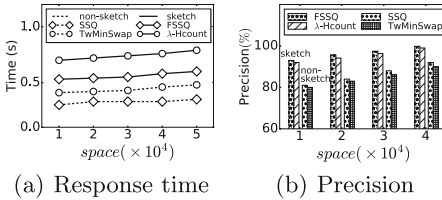
**Fig. 3.** Effect of  $z$  on synthetic



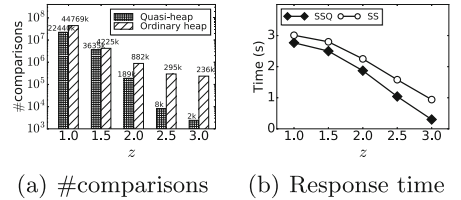
**Fig. 4.** Effect of  $\phi$  on synthetic



**Fig. 5.** Effect of  $\tau$  on Kosarak



**Fig. 6.** Effect of memory on retail



**Fig. 7.** Effectiveness of Quasi-heap

## 8 Conclusion and Future Work

In this paper, we focused on the problem of finding frequent items in data streams with a time decay model. In order to reduce the maintenance cost of the ordinary heap, we proposed a Quasi-heap data structure with a delayed sorting operation and invented two algorithms based on it. We extensively evaluated our methods on three datasets. In the future, we plan to extend the proposed methods in a distributed environment especially for data whose volume does not fit into memory of a stand-alone machine.

**Acknowledgement.** This work was supported by the public key plan of Zhejiang Province (2014C23005), National Science and Technology Supporting plan (2013BAH62F02 and 2013BAH27F01), China mobile research fund of education ministry (mcm20130671), the cultural relic protection science and technology project of Zhejiang Province, University of Macau RC (MYRG2014-00106-FST), and NSFC of China (61502548).

## References

1. Brijs, T., Swinnen, G., Vanhoof, K., Wets, G.: Using association rules for product assortment decisions: a case study. In: SIGKDD, pp. 254–260. ACM (1999)
2. Chakrabarti, A., Cormode, G., McGregor, A.: A near-optimal algorithm for computing the entropy of a stream. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 328–335. Society for Industrial and Applied Mathematics (2007)
3. Chang, J.H., Lee, W.S.: Finding recent frequent itemsets adaptively over online data streams. In: SIGKDD, pp. 487–492. ACM (2003)
4. Chen, L., Mei, Q.: Mining frequent items in data stream using time fading model. *Inf. Sci.* **257**, 54–69 (2014)
5. Chen, L., Zhang, S., Tu, L.: An algorithm for mining frequent items on data stream using fading factor. In: COMPSAC, vol. 2, pp. 172–177. IEEE (2009)
6. Chen, L., Zou, L.J., Tu, L.: A clustering algorithm for multiple data streams based on spectral component similarity. *Inf. Sci.* **183**(1), 35–47 (2012)
7. Cormode, G., Hadjieleftheriou, M.: Finding the frequent items in streams of data. *Commun. ACM* **52**(10), 97–105 (2009)
8. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. *J. Algorithms* **55**(1), 58–75 (2005)
9. Cormode, G., Shkapenyuk, V., Srivastava, D., Xu, B.: Forward decay: a practical time decay model for streaming systems. In: ICDE, pp. 138–149. IEEE (2009)
10. Homem, N., Carvalho, J.P.: Finding top-k elements in data streams. *Inf. Sci.* **180**(24), 4958–4974 (2010)
11. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. *TODS* **28**(1), 51–55 (2003)
12. Lim, Y., Choi, J., Kang, U.: Fast, accurate, and space-efficient tracking of time-weighted frequent items from data streams. In: CIKM, pp. 1109–1118. ACM (2014)
13. Manerikar, N., Palpanas, T.: Frequent items in streaming data: an experimental evaluation of the state-of-the-art. *Data Knowl. Eng.* **68**(4), 415–430 (2009)
14. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: PVLDB, pp. 346–357. VLDB Endowment (2002)

15. Mei, Q.L., Chen, L.: An algorithm for mining frequent stream data items using hash function and fading factor. In: *Applied Mechanics and Materials*, vol. 130, pp. 2661–2665. Trans Tech Publications (2012)
16. Metwally, A., Agrawal, D., Abbadi, A.E.: An integrated efficient solution for computing frequent and top-k elements in data streams. *TODS* **31**(3), 1095–1133 (2006)
17. Shaker, A., Senge, R., Hüllermeier, E.: Evolving fuzzy pattern trees for binary classification on data streams. *Inf. Sci.* **220**, 34–45 (2013)
18. Tong, Y., Zhang, X., Chen, L.: Tracking frequent items over distributed probabilistic data. *World Wide Web* **19**(4), 1–26 (2015)
19. Zhang, S., Chen, L., Tu, L.: Frequent items mining on data stream based on time fading factor. In: *AICI*, vol. 4, pp. 336–340. IEEE (2009)

Web Technologies and Applications

18th Asia-Pacific Web Conference, APWeb 2016,

Suzhou, China, September 23-25, 2016. Proceedings,

Part II

Li, F.; Shim, K.; Zheng, K.; Liu, G. (Eds.)

2016, XXII, 601 p. 236 illus., Softcover

ISBN: 978-3-319-45816-8