World Wide Web (2017) 20:1111–1133
DOI 10.1007/s11280-017-0433-5

Novel structures for counting frequent items in time decayed streams

Shanshan Wu[1] · Huaizhong Lin[1] · Leong Hou U[2] · Yunjun Gao[1] · Dongming Lu[1]

Received: 4 October 2016 / Revised: 10 December 2016 /
Accepted: 11 January 2017 / Published online: 24 January 2017

Abstract Identifying frequently occurring items is a fundamental 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 the time decay, where the importance of every arrival item is decreased over the time. To address the importance changes over time, we propose an innovative heap structure, named Quasi-heap, which maintains the item order using a lazy update mechanism. Two approximation algorithm, 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. To achieve better accuracy of frequency estimation for all the items in the stream, we introduce a new count-min-min (CMM) sketch structure, which can estimate the count of an item with almost error free. Extensive experiments conducted on both real-world and synthetic data demonstrate the superiority of proposed methods in terms of both efficiency (i.e., response time) and effectiveness (i.e., accuracy).

✉ Huaizhong Lin
linhz@zju.edu.cn

Shanshan Wu
wuss@zju.edu.cn

Leong Hou U
ryanlhu@umac.mo

Yunjun Gao
gaoyj@zju.edu.cn

Dongming Lu
ldm@zju.edu.cn

1 College of Computer Science and Technology, Zhejiang University, Hangzhou, China

2 Faculty of Science and Technology, University of Macau, Macau, China

Keywords Frequent item · Data stream · Time decay model · Data structure

1 Introduction

A data stream is a massive unbounded sequence of item 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 [9], classification [26], and mining frequent patterns [5, 30]. Finding frequent items [6, 11, 12, 15, 17, 18, 21, 23, 25, 28] 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 [1, 2, 5, 19, 30] and computing the entropy of a data stream [4].

In typical data stream scenarios, the item arrival rate is very high so that not all received items can be kept in the main memory. Thereby, typical solutions 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, prior studies mostly focus on how to answer the data stream problem approximately with an error bound.

Early solutions [11, 12, 15, 16, 18, 23, 25] of this problem are developed based on two traditional streaming models, the landmark model (i.e., the frequent items can be any item in the entire stream) and the sliding window model (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. In many applications, recent data in the stream is more meaningful. For instance, in an athlete ranking system, more recent records typically should be given more weight [16]. One way to address this problem is to use a sliding window model. However, the sliding window model partially address the item freshness but the items not occurring in the current window are completely ignored.

To address these, answering frequent items in *time decayed* data streams has received substantial attention from the community [7, 8, 13, 20, 24, 32, 33]. Under the *time decay* model, the weight of the received items is decreased over time and the frequent items are then computed based on the time decayed counts. This model preserves better completeness (i.e., every item is considered) and item freshness (i.e., 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 swap the affected item with its neighbors to maintain the order consistency. Under the time decay model, the weight of every item is updated over the time subject to the decay function. Such huge number of updates makes the frequent item problem challenging since the ordered list becomes difficult to maintain.

To address this challenge, we propose a new heap structure, named Quasi-heap, which maintains the order of unpromising items in the 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 stream. 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 SS [25]) and Filtered Space Saving with Quasi-heap (based on FSS [16]) to find the frequent items in time decayed data streams. Our improved algorithms answer the frequent item problem with reasonable memory space and guaranteed error bound. In addition, we theoretically analyze the algorithm complexity.
- Extensive experiments are conducted to demonstrate the superiority of our algorithms in terms of the running time and the estimation accuracy. More specifically, our algorithms reduce response time up to 80 % compared with the ordinary heap solutions and provide better estimation quality than prior studies.

A preliminary report of this work is published in [31]. The new contents in this manuscript include (1) studying a new count-min-min (CMM) sketch with higher estimation accuracy, compared to the previously well-known count-min sketch; and (2) supplementing some lemmas, theorems, and proofs which are left out in the preliminary conference version [31]; and (3) conducting enhanced experimental evaluation that investigates the efficiency of the proposed Quasi-heap, and adding several sets of experiments that shows the effectiveness of the proposed CMM sketch (CMM sketch can estimate the count of an item with almost error free).

The remainder of this paper is organized as follows. A survey of related work is presented in Section 2. Section 3 formulates the frequent items problem in time decayed data streams. Section 4 discusses and analyzes the Quasi-heap. Sections 5 and 6 depict two improved algorithms SSQ and FSSQ, respectively. Section 7 gives the theoretical analysis for our CMM sketch. Section 8 evaluates the proposed algorithms, and we conclude this paper in Section 9.

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 streams [10], counter-based and sketch-based.

The counter-based algorithms [15, 18, 23, 25] are deterministic algorithms which only monitor a subset of items from the data stream. These algorithms maintain a set of counters to track the frequent items over the subset. Space Saving [25], Lossy Counting [23], Frequent [15, 18] are the representative algorithms in this stream.

Another line of work is the sketch-based algorithms which use a set of array counters to estimate the frequency 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 counter of its corresponding sketches. To minimize the collision probability of the hash functions, we can increase the granularity of sketch (i.e., more counters are used). However, this will lead to huge memory consumption. CountSketch [6], Count-Min Sketch [11, 12], and FSS [16] are the representative algorithms in this stream.

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 remarkable attentions from the community recently [7, 8, 13, 20, 24, 32, 33]. Zhang et al. [32] proposed two ϵ-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. [8] proposed another ϵ-approximation algorithm, called Frequent-item Counting algorithm (FC), which finds the frequent data items based on the fading factor. It takes $O(1)$ time to maintain the answer for each arrival item by a hash function. Mei and Chen [24] proposed to estimate the frequency of items by multiple hash functions. However, their work did not give the analysis of the memory consumption and the result accuracy.

Recent developments attempted to improve the estimation accuracy by either exploiting the decay function or employing new data structure. Lim et al. [20] proposed a new ϵ-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 time by multiplying the decay rate. λ-HCount algorithm [7] 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 time. The main challenge under the backward decay function is that the weight of the existing items is constantly changed. To address this, Cormode et al. [13] 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 weight of the existing items becomes stable and the problem of finding the frequent items 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 the forward decay 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 prior studies.

3 Definitions and preliminaries

Table 1 summarizes the notations to be used in the rest of this paper. We use a standard stream model with discrete timestamp labeled as $< 0, 1, 2, 3, ... >$ and only one item a_i arrives at every timestamp. The current data stream D_n is the set of items that have been received so far, i.e., $D_n =< I_1, I_2, ..., I_n >$.

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 obsolete items.

Table 1 Summary of notation

Notation	Description
D_n	The data stream up to time n
$\|D_n\|$	Sum of the decayed counts of all items in D_n
a_i	An item in a data stream
I_t	The received item at time t
$C_t(a_i)$	The decayed count of the item a_i at time t
$c_t(a_i)$	The estimated decayed count of the item a_i at time t
τ	Time decay rate
ϕ	Frequency threshold
ϵ	Error tolerance parameter
n_k	The frequency of the k-th frequent item
m	The length of the monitored list
n	The number of all the items in a data stream
D	The number of all the distinct items in a data stream

Definition 1 (Decayed Count of An Item, $C_t(a_i)$[5]) Given a time decay rate τ $(0 < \tau \leq 1)$, $C_t(a_i)$ is the decayed count of item a_i at time t, i.e.,

$$C_t(a_i) = C_{t-1}(a_i) \times \tau + W_t(a_i) \tag{1}$$

where $C_1(a_i) = W_1(a_i)$ and $W_t(a_i)$ is a function that indicates the arrival of item a_i at time t. Specifically, if $I_t = a_i$, $W_t(a_i) = 1$; otherwise, $W_t(a_i) = 0$.

Based on Definition 1, we should update the decayed count of every item for each time step. We observe that (1) is a monotone decreasing function (i.e., decreased by a factor of τ) until the item is received again from the data stream. Accordingly, at the current time t, the decayed count of an arrival item a_i can be updated by

$$C_t(a_i) = C_{ut}(a_i) \times \tau^{(t-ut)} + 1$$

if the last updated time ut of a_i is recorded.

The following Definition 2 defines our frequent item problem in this work which aims at returning a set of highly occurring items subject to a frequency threshold ϕ. Specifically, an item a_i is in the result set if and only if its *normalized* decayed count is higher than ϕ and the normalization factor is the sum of all items $|D_n|$ $(= (1-\tau^n)/(1-\tau)$, which approaches to $1/(1-\tau)$ when $n \to \infty)$.

Definition 2 (ϕ-Frequent Item) Given a stream D_n of $< I_1, I_2, ..., I_n >$, and a frequency threshold ϕ, $0 < \phi \leq 1$. If the decay weight $C_n(a_i)$ is higher than $\phi \times |D_n|$, then a_i is a ϕ-frequent item.

We need huge memory to maintain the ϕ-frequent items exactly in a long running data stream, where the space complexity is $\Omega(D)$ space when the number of distinct items in the stream is D [10]. Thereby, typical solutions focus on answering this problem approximately subject to an error bound ϵ. Formally, the approximation version of the problem is defined as follows.

Definition 3 (ϵ-Approximate Decayed Frequent items) Given a stream D_n of $< I_1, I_2, ..., I_n >$, the ϵ-approximate frequent item set contains all items where their decayed counts are higher than $(\phi - \epsilon) \times |D_n|$.

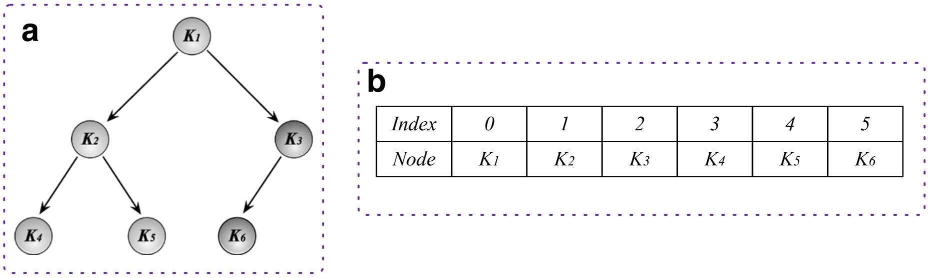
4 Quasi-heap

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, prior studies [25] replace the smallest item by the newly arrival item when the memory becomes full.

A heap is a partially sorted complete binary tree which is usually compactly stored in an array data structure, as shown in Figure 1. Although a heap is not completely in order, it conforms to a sorting principle: every node has a value less than or equal to both of its children (actually, the value of every node can also be larger than or equal to both of its children, but here we only use the heap with smaller value on the upper nodes, i.e., min-heap). More specifically, a file of keys $K_1, K_2, ..., K_m$ is a heap if $K_{\lfloor j/2 \rfloor} \leq K_j$, for $1 \leq \lfloor j/2 \rfloor < j \leq m$, thus $K_1 \leq K_2$, $K_1 \leq K_3$, $K_2 \leq K_4$, etc. It implies in particular that the smallest key appears on top of the heap, i.e., $K_1 = min(K_1, K_2, ..., K_m)$. An efficient approach to the heap creation has been suggested by R. W. Floyd [14].

In the landmark model, the count value always increases by 1 for each coming item. This fact ensures the updated time for each coming item is constant by using the linked list or heap data structure, since we only need to move counters between neighboring parent buckets in linked list or neighboring nodes in heap to keep correct order.

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. Suppose we use a typical structure to keep these counters, the complexity of each update takes $O(m)$ (for the linked list) and $O(\log m)$ (for the heap structure) time which is definitely too time consuming in a data stream environment. Hence, we propose a new data structure called Quasi-heap which aims at postponing unpromising sorting operations when the decayed count of an existing item is increased. In other words,



Index	0	1	2	3	4	5
Node	K1	K2	K3	K4	K5	K6

Figure 1 An example of an ordinary min-heap

we allow certain inconsistency in the Quasi-heap structure. An example of the Quasi-heap is given in the following Example 1.

Example 1 (An example of Quasi-heap) In Figure 2, we demonstrate the running procedures of the Quasi-heap. Each node consists of the item name and its decayed count. We adopt $\tau = 1$ for ease of presentation. Figure 2a shows a Quasi-heap that contains 12 items where the order is identical to that of the ordinary heap. Upon receiving the next sequence $< a, b, a, b, e, e >$, 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 are marked by thick lines in Figure 2b) 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, item c becomes the root node as it is the smallest item in the Quasi-heap (shown in Figure 2c). Then, item c is replaced by item n (cf. Figure 2d) where the *estimated count* of n is set to 4 (i.e., the count of the evicted item $c + 1$) as followed the suggestion of other counter-based algorithms [25].

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

```
   Input: c, a counter in the Quasi-heap;
          t, current time
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 $c.delay = 0$ then return;
3  if c is a leaf node then return;
4  delayedSorting (the left child counter of $c$);
5  delayedSorting (the right child counter of $c$);
6  let $sml$ be the smaller of the two child counters;
7  if $c.cnt > sml.cnt$ then
8      swap $c$ and $sml$ and $sml.delay \leftarrow 1$;
9  end
10 else if $c.cnt = sml.cnt$ then
11     if sml has larger estimated error than c then
12         swap $c$ and $sml$ and $sml.delay \leftarrow 1$;
13     end
14 end
15 $c.delay = 0$
```

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 (lines 4–5). After the recursive calls, if the count of the root is larger than the children, we swap the root with its child in lines 7–8. 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 9 of Algorithm 2 and Example 1).

We give the properties of Quasi-heap in the following Lemma 1 and Theorem 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.*

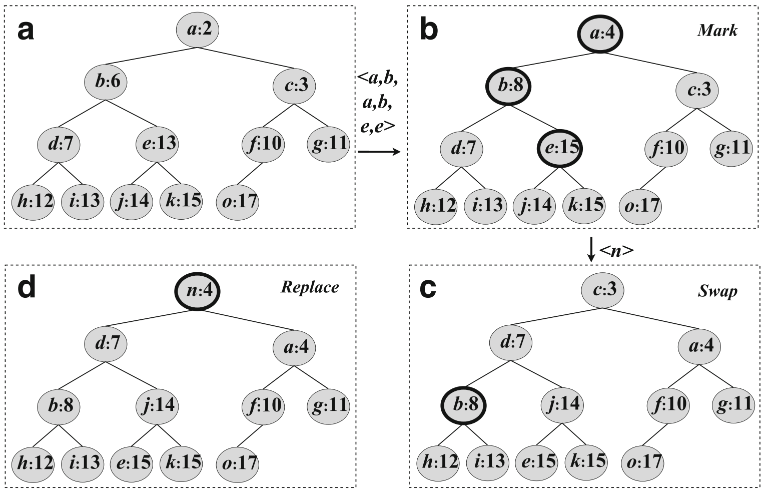


Figure 2 A running example of Quasi-heap

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 the item 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. (1)). If the item 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. □

According to the Lemma 1, we can get another property. If $c.delay = 0$, the decayed count of c is no larger than that of any descendent in its subtree. So, c has the smallest decayed count in its subtree.

Theorem 1 *After executed delayedSorting (i.e., Algorithm 1) on item c, c must have the minimum decayed count among its subtree.*

Proof We prove it by induction. Let ht be the height of the subtree with c as its root. When $ht = 1$, the statement is obviously true.

Suppose the statement is true for $ht - 1$, now we prove that it is also true for ht. If $c.delay = 0$, then c is already the minimum decayed count in its subtree. If $c.delay = 1$, Algorithm 1 is called for its two children counter. After the execution of algorithm 2 for the two children counter, they both have the minimum decayed count in their own subtrees by the assumption. By comparing and swapping counter c with the smaller of two children counter, the item kept in the counter c becomes the minimum decayed count in its subtree. □

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 take $O(1)$ 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 count is high) and it is more frequently received form 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 (SSQ)

Algorithm 2 SSQ

```
Input: t, the current time;
       D_n, a data stream;
       m, the length of the monitored list;
1  for each incoming item c at timestamp t do
2      if c is tracked in Quasi-heap then
3          $c.error \leftarrow c.error \times \tau^{t-c.ut}$;  $c.cnt \leftarrow c.cnt \times \tau^{t-c.ut} + 1$;  $c.delay \leftarrow 1$;  $c.ut \leftarrow t$;
4      end
5      if c is not tracked in Quasi-heap then
6          if Quasi-heap is full then
7              Let r be the root of the Quasi-heap;
8              delayedSorting(r);
9              $c.error \leftarrow r.cnt \times \tau^{t-r.ut} + 1$;  $c.cnt \leftarrow r.cnt \times \tau^{t-r.ut} + 1$;  $c.delay \leftarrow 1$;  $c.ut \leftarrow t$;
10             replace r by c;
11         end
12         if Quasi-heap is not full then
13             create a new counter c;
14             $c.error \leftarrow 0$;  $c.cnt \leftarrow 1$;  $c.delay \leftarrow 0$;  $c.ut \leftarrow t$;
15             insert and maintain c in the Quasi-heap;
16         end
17     end
18 end
```

We study a counter-based algorithm, SSQ (that is based on the SS algorithm [25]), to find the ϵ-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 12–15), 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.

We present two properties of our SSQ algorithm, which are based on the properties proposed in the Space-Saving algorithm [25] and the FE algorithm [32] with some minor modifications.

Lemma 2 *Among all m counters, the minimum counter value μ is no greater than* $(1-\tau^n)/m(1-\tau)$.

Proof The sum of estimated counts of all m items in the monitored list is no greater than the sum of decayed counts of all n items in the data stream, i.e., $\Sigma_i c_n(a_i) \leq (1-\tau^n)/(1-\tau)$.

$$m\mu \leq \Sigma_i c_n(a_i) \leq (1-\tau^n)/(1-\tau)$$

so, $\mu \leq (1-\tau^n)/m(1-\tau)$ □

Based on the Lemma 2, the SSQ algorithm can use confined space (i.e., setting $m = \lceil 1/\epsilon \rceil$) to find ϵ-Approximate frequent items by securing the error ratio at most $\epsilon \times |D_n|$ [25].

Theorem 2 (No False Negative) *For any item a_i with decayed count $C_n(a_i)$ greater than μ (the minimum counter value in Quasi-heap) is present in the Quasi-heap.*

Proof We prove the theorem by contradiction. Assume in 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' in the past. When a_i was evicted, its decayed count was $C_{t'}(a_i) = C_t(a_i)/\tau^{t-t'}$, which is larger than $\mu/\tau^{t-t'}$ (according to the assumption $C_t(a_i) > \mu$). Let $\mu_{t'}$ be the minimal counter value at time unit t', then $\mu_{t'}$ is no larger than $\mu/\tau^{t-t'}$. 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 estimated count of 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. □

6 Filtered space saving algorithm with quasi-heap (FSSQ)

In this section, we propose a sketch-based algorithm, Filtered Space Saving algorithm with Quasi-Heap (FSSQ) (that is based on the Filtered Space Saving algorithm [16]), 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 the width w and the depth d). The count-min sketch data structure used in this work is similar to that of the Count-Min algorithm [10]. The structure is conceptually described in Figure 3. 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: $h_1, ..., h_d : \{1, 2, ..., D\} \rightarrow \{1, 2, ..., w\}$. 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).

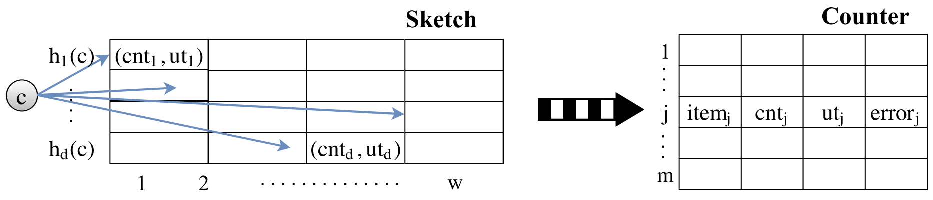


Figure 3 The data structures used in FSSQ algorithm

Algorithm 3 FSSQ

```
Input: $t$, the current time;
       $D_n$, a data stream;
       $m$, the length of the monitored list;
       $d$, the depth of the sketch;
       $w$, the width of the sketch;
       $s[x, y]$, the count in the sketch entry $(x, y)$
1  for each incoming item c at timestamp $t$ do
2      if $c$ is tracked in Quasi-heap then
3          $c.error \leftarrow c.error \times \tau^{t-c.ut}$;  $c.cnt \leftarrow c.cnt \times \tau^{t-c.ut} + 1$;  $c.delay \leftarrow 1$;  $c.ut \leftarrow t$;
4      end
5      if $c$ is not tracked in Quasi-heap then
6          if Quasi-heap is full then
7              delayedSorting(the root $r$ of the Quasi-heap);
8              Let $s[x, y]$ be the counter in the sketch entry $(x, y)$;
9              for $j = 1, ..., d$ do
10                 $s[j, h_j(c)].cnt \leftarrow s[j, h_j(c)].cnt \times \tau^{t-s[j,h_j(c)].ut} + 1$;
11                 $s[j, h_j(c)].ut \leftarrow t$;
12             end
13             $est \leftarrow min_{1\le j\le d} s[j, h_j(c)].cnt$;
14             if $est > r.cnt$ then
15                 for $j = 1, ..., d$ do
16                     $s[j, h_j(r)].cnt \leftarrow r.cnt \times \tau^{t-r.ut}$;
17                     $s[j, h_j(r)].ut \leftarrow t$;
18                 end
19                 $c.error \leftarrow r.cnt \times \tau^{t-r.ut}$;  $c.cnt \leftarrow r.cnt \times \tau^{t-r.ut} + 1$;  $c.delay \leftarrow 1$;  $c.ut \leftarrow t$;
20                 replace $r$ by $c$;
21             end
22         end
23         if Quasi-heap is not full then
24             create a new counter $c$;
25             $c.error \leftarrow 0$;  $c.cnt \leftarrow 1$;  $c.delay \leftarrow 0$;  $c.ut \leftarrow t$;
26             insert and maintain $c$ in the Quasi-heap;
27         end
28     end
29 end
```

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 the Quasi-heap becomes full, hence we omit to describe the similar parts. We first run *delayedSorting* from the root (line 7) in order to find the minimum item. Next we update the corresponding sketch entries by c (lines 9–11), and estimate its minimum value among d corresponding sketch entries (line 13). If the estimated minimum count is larger than the root of Quasi-heap, then we replace the root by the new item c (lines 19–20) and update the corresponding sketch entries by the evicted item r (lines 15–17).

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

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

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

Now we consider the situation when the Quasi-heap has been full. For a new coming item, if it is being monitored in the Quasi-heap, its counter in Quasi-heap increases and all the entries in the sketch remain unchanged. With the same decay rate and initially $\mu \geq min_{1\leq j\leq d} s[j, h_j(a_i)]$, the inequality is still true.

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 μ. However, at this case, a replacement is taken place. Hence, the conclusion is still true. □

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

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 μ according to Lemma 3, so the maximum overestimated error is no greater than μ. □

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 introduce indicator variables $I_{i,j,k}$ as follows.

$$I_{i,j,k} = \begin{cases} 1 & \text{if } a_i \neq a_k \wedge h_j(a_i) = h_j(a_k) \\ 0 & \text{if } otherwise \end{cases}$$

By pairwise independence of the hash functions, then the expectation of variables $I_{i,j,k}$ is

$$E(I_{i,j,k}) = Pr[h_j(a_i) = h_j(a_k)] \leq 1/w \leq \epsilon/e$$

Let $X_{i,j} = \Sigma_{k=1,\ldots,D}(I_{i,j,k} \times C_n(a_k))$. Since all $C_n(a_k)$ are non-negative, $X_{i,j}$ is a non-negative variable. By construction, $s[j, h_j(a_i)] = C_n(a_i) + X_{i,j}$. Thereby,

$$min_{1\leq j\leq d} s[j, h_j(a_i)] \geq C_n(a_i)$$

By pairwise independence of h_j, and linearity of expectation, we observe that

$$\begin{aligned} E(X_{i,j}) &= E(\Sigma_{k=1,\ldots,D}(I_{i,j,k} \times C_n(a_k))) \\ &= \Sigma_{k=1,\ldots,D}(E(I_{i,j,k}) \times C_n(a_k)) \\ &\leq \epsilon/e\Sigma_{k=1,\ldots,D} C_n(a_k) \leq \frac{\epsilon}{e \times (1-\tau)} \end{aligned}$$

By the Markov inequality,

$$Pr[X_{i,j} > eE(X_{i,j})] < E(X_{i,j})/eE(X_{i,j}) < 1/e$$

By combining these,

$$Pr[\forall j, X_{i,j} > eE(X_{i,j})] < e^{-d}(1 \leq j \leq d)$$

$$\begin{aligned} Pr[c_n(a_i) > C_n(a_i) + \epsilon/(1-\tau)] &= Pr[\forall j, s[j, h_j(a_i)] > C_n(a_i) + \epsilon/(1-\tau)] \\ &= Pr[\forall j, C_n(a_i) + X_{i,j} > C_n(a_i) + \epsilon/(1-\tau)] \\ &= Pr[\forall j, X_{i,j} > \epsilon/(1-\tau)] \\ &\leq Pr[\forall j, X_{i,j} > eE(X_{i,j})] < e^{-d} \leq \delta \end{aligned}$$

□

Theorem 3 *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 2 if a_i is present in the Quasi-heap. □

7 Count-Min-Min (CMM) sketch

In this section, we will introduce our Count-min-min (CMM) sketch which is an improved sketch based on the best well-known count-min (CM) sketch. CMM sketch provides a tighter bound of frequency estimation error.

In recent years, several different sketches [6, 11, 29] have been proposed in the data stream context to solve large-scale computation. These sketches in general consume reasonable space overhead and offer high accuracy result. Our CMM sketch lies in the same framework, and finds inspiration from these previous sketches. The common framework has been described in Figure 3 of Section 6.

Two well-known sketches are count sketch [6] and count-min sketch [11]. These two sketches both use multiple hash functions to define a projection from incoming items to a set of array counters. Each item is hashed by some hash functions into one or more values, which can be used to index the counters to update. The count sketch uses the mean or median of these estimates to achieve an estimation count. However, the count-min sketch uses the minimum count of all corresponding counters to estimate the count of an item.

In the count-min sketch, an array of $d \times w$ counters (each counter is initialized with zero) is maintained, along with d hash functions. When a new item c comes, it is projected by d hash functions into d corresponding counters in the array and each of which is incremented, shown in Figure 4. A query for the frequency of any item in the data stream reports the minimum count of all corresponding d counters.

According to Lemma 5, it is known that the counts estimated by the count-min sketch are overestimated. Now, when a new item c comes, we only increase the minimum count of these d corresponding counters, not each of them. Even so, it is still guaranteed that the frequency estimation of every item is overestimated. We call the improved count-min sketch for count-min-min (CMM) sketch, shown in Figure 5. We argue that the accuracy of the frequency estimation will be improved.

Noted that, we will increase all the counts when multiple corresponding counters all have the minimum value.

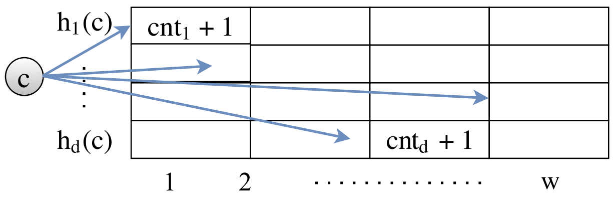


Figure 4 The update in the CM sketch

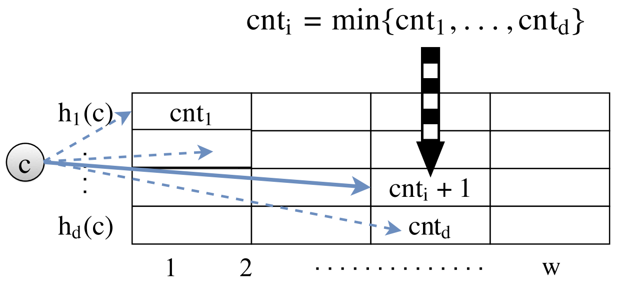


Figure 5 The update in the CMM sketch

On theoretical aspect, the following Lemma 6 will show that the counts estimated by the count-min sketch are still overestimated and the estimation error is tighter.

Lemma 6 *In CMM sketch, the estimate $c_n(a_i)$ has the following guarantees: $C_n(a_i) \leq c_n(a_i)$, and with probability at least $1-\delta$, $c_n(a_i) \leq C_n(a_i) + \frac{\epsilon}{(1-\tau)\times ln(1/\delta)}$.*

Proof We introduce indicator variables $I_{i,j,k}$ as follows.

$$I_{i,j,k} = \begin{cases} 1 & \text{if } a_i \neq a_k \wedge h_j(a_i) = h_j(a_k) \wedge s[j, h_j(a_k)] = min_{1\leq j\leq d} s[j, h_j(a_k)] \\ 0 & \text{if } otherwise \end{cases}$$

By pairwise independence of the hash functions, then the expectation of variables $I_{i,j,k}$ is

$$E(I_{i,j,k}) = Pr[h_j(a_i) = h_j(a_k) \wedge s[j, h_j(a_k)] = min_{1\leq j\leq d} s[j, h_j(a_k)]] \leq \frac{1}{wd} \leq \frac{\epsilon}{eln(1/\delta)}$$

Let $X_{i,j} = \Sigma_{k=1,...,D}(I_{i,j,k} \times C_n(a_k))$. Since all $C_n(a_k)$ are non-negative, $X_{i,j}$ is a non-negative variable. By construction, $s[j, h_j(a_i)] = C_n(a_i) + X_{i,j}$. Thereby,

$$min_{1\leq j\leq d} s[j, h_j(a_i)] \geq C_n(a_i)$$

By pairwise independence of h_j, and linearity of expectation, we observe that

$$\begin{aligned} E(X_{i,j}) &= E(\Sigma_{k=1,...,D}(I_{i,j,k} \times C_n(a_k))) \\ &= \Sigma_{k=1,...,D}(E(I_{i,j,k}) \times C_n(a_k)) \\ &\leq \frac{\epsilon}{eln(1/\delta)} \times \Sigma_{k=1,...,D} C_n(a_k) \\ &\leq \frac{\epsilon}{eln(1/\delta) \times (1-\tau)} \end{aligned}$$

By the Markov inequality,

$$Pr[X_{i,j} > eE(X_{i,j})] < E(X_{i,j})/eE(X_{i,j}) < 1/e$$

By combining these,

$$Pr[\forall j, X_{i,j} > eE(X_{i,j})] < e^{-d}(1 \leq j \leq d)$$

$$\begin{aligned}
Pr[c_n(a_i) > C_n(a_i) + \frac{\epsilon}{(1-\tau)\times ln(1/\delta)}] &= Pr\left[\forall j, s[j, h_j(a_i)] > C_n(a_i) + \frac{\epsilon}{(1-\tau)\times ln(1/\delta)}\right] \\
&= Pr\left[\forall j, C_n(a_i) + X_{i,j} > C_n(a_i) + \frac{\epsilon}{(1-\tau)\times ln(1/\delta)}\right] \\
&= Pr\left[\forall j, X_{i,j} > \frac{\epsilon}{(1-\tau)\times ln(1/\delta)}\right] \\
&\leq Pr[\forall j, X_{i,j} > eE(X_{i,j})] < e^{-d} \leq \delta
\end{aligned}$$

□

8 Experimental study

In this section, we empirically evaluate the efficiency of SSQ and FSSQ using both real and synthetic datasets. We compared our proposed solutions with the state-of-the-art solutions, TwMinSwap [20] (counter-based) and λ-HCount [7] (sketch-based). Most of the existing works arrange the items in the monitored list according to their updated time and remove the item with the least updated time when the monitored list is full. Hence, all implemented algorithms in our experiments also follow this convention. All methods were implemented in C++ and compiled using Microsoft Visual Studio 2012 compiler. All experiments were conducted on a 3.20 GHz Pentium PC machine with 8GB main memory running Windows 7 Professional Edition.

In the sequel, we first present the experimental setup in Section 8.1, and report the experimental results and our findings in Section 8.2, and a comparison between Quasi-heap and an ordinary heap is made to investigate the performance gains from Quasi-heap in Section 8.3, and then several sets of experiments are conducted to evaluate the effectiveness of our proposed CMM sketch in Section 8.4.

8.1 Experimental settings

We employed both synthetic and real datasets in our experiments. The synthetic datasets are generated based on Zipfian distributions. Table 2 shows every parameter and their values used in the experiments. (An appropriate Zipfian parameter is chosen so that the data is not overly skewed, which will make it very easy to distinguish frequent items. But at the same time it also guarantees that the number of frequent items which is above the threshold is not very little). We also used two real datasets, e.g., Kosarak and Retail, that are widely evaluated in data stream research [22, 27]. The Kosarak dataset is an anonymized click-stream on a Hungarian online news portal.[1] It consists of transactions, each of which has several items, expressed as integers. The Retail dataset contains retail market basket data from an anonymous Belgian store [3]. In our experiments, we consider every single item in sequential order. The detail statistics of the real datasets is listed in Table 3.

[1]Frequent Itemset Mining Dataset Repository, available at http://fimi.cs.helsinki.fi/data/ (last accessed on 17 November, 2016)

Table 2 Parameters

Parameter	Values	Default values
n	$10^4, 10^5, 10^6, 5 \times 10^6, 10^7$	10^7
z	0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0	1.0
ϕ	0.0001, 0.0005, 0.001, 0.005, 0.01	0.001
$space(bytes)$	$10^4, 2 \times 10^4, 3 \times 10^4, 4 \times 10^4, 5 \times 10^4$	4×10^4
τ	0.97, 0.975, 0.98, 0.985, 0.99, 0.995, 1.00	0.995

For fairness, we used common subroutines for similar tasks (e.g., hash tables) to increase comparability and allocated identical memory budget to all the algorithms. Based on FSS algorithm proposed by Homem and Carvalho [16], m_2, the number of counters in sketch-based algorithms, is almost half of m_1, the number of counters in counter-based algorithms. The budgeted memory for all algorithms are the same, i.e., $40 \times m_1 = 40 \times m_2 + 16 \times d \times w$ (40 bytes for every counter, 16 bytes for a unit in the sketch structure, the depth d of the sketch and the width w of the sketch). A large number of experiments also show that the conflict can be down to very low when d is 4. And in all experiments, the default value of m_1 is 800. We verify the performance of algorithms with respect to:

– **Time:** Each algorithm is run for 20 times and their average response time is reported.
– **Precision:** The fraction of the items identified by the algorithm that are actually frequent.
– **Recall:** The fraction of the actual frequent items that the algorithm identified. In all algorithms, all the frequent items are detected due to the overly estimated count (cf. Theorem 2).

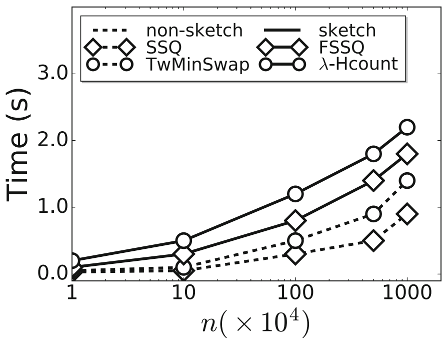
8.2 Experimental results

To verify the scalability of the algorithms, we varied one parameter in each set of experiments while setting other parameters to their default values. In our experiments, we showed the response time and precision of the algorithms as a function of stream size (n), data skew (z), frequency threshold (ϕ), space consumed and decay rate (τ) on three datasets, respectively.

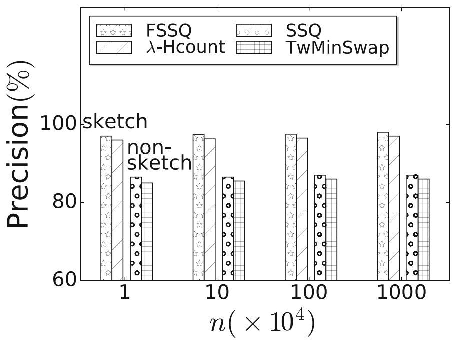
Performance Overview In terms of the response time, SSQ yields better performance than state-of-the-art solutions, due to the Quasi-heap data structure proposed which can save the response time by up to 80 %. In terms of the precision, FSSQ performs the best among all methods due to the sketch structure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

Table 3 Statistics of the real data streams

Dataset name	Size n	Distinct items (D)	Min	Max	Mean	Median	Standard deviation	Median
Kosarak	8019015	41270	1	41270	2387.2	640	4308.5	3.5
Retail	908576	16470	0	16470	3264.7	1564	4093.2	1.5



(a) Response time



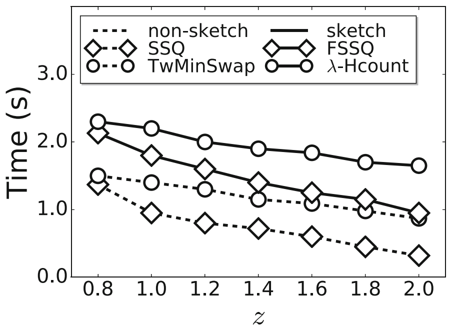
(b) Precision

Figure 6 Varying stream size on Synthetic dataset

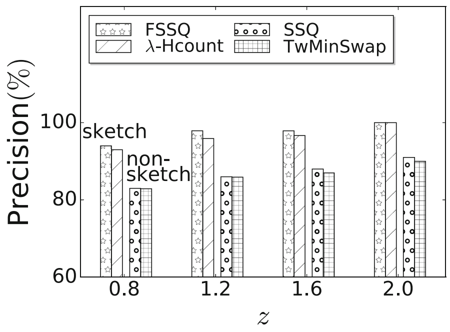
Figure 6 shows the response time and the precision by varying the cardinality of the items. Notably, the response time of all methods increases as the stream size becomes larger. We find that the counter based algorithms are faster than the sketch based algorithms; however, the counter based algorithms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

It is obvious that the response time decreases when the data becomes more skewed (cf. Figure 7a). That is because that 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. In other words, the skewness of data can simplify the problem as there are fewer frequent item candidates. Therefore, the response time of all methods decreases when the data become more skewed.

Figures 8 and 9 show the experiments conducted ϕ and τ, respectively. Due to space limit, each figure only reports two sets of experiments as all methods preform similarly on these three datasets. For instance, SSQ is 35–40 % faster than TwMinSwap on average for the frequency threshold ϕ and the decay rate τ in all three datsets. In Figure 9, we observe



(a) Response time



(b) Precision

Figure 7 Varying data skew on Synthetic dataset

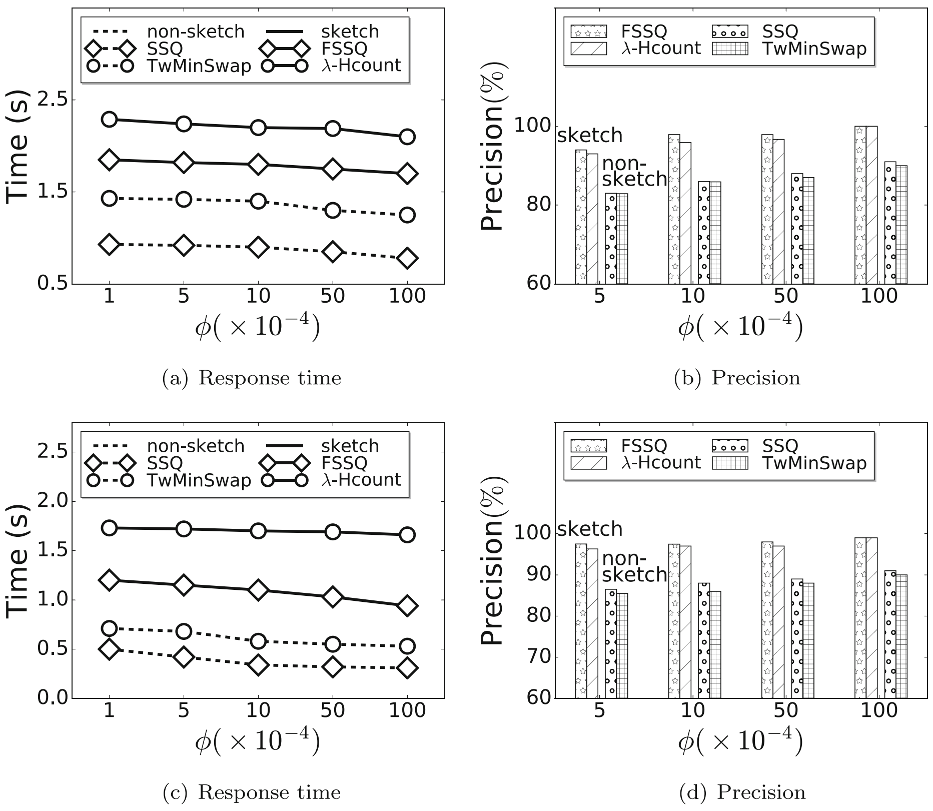


Figure 8 Varying frequency threshold on Synthetic dataset (**a** and **b**) and Kosarak datasets (**c** and **d**)

that the trend of the response time and precision almost keep unchanged as the decay parameter changes. There should not be mush significant dependence on this parameter, since the underlying problem is the same, just the input weight are being implicitly modified.

Figure 10 shows that the precision increases significantly when we have more space for the Quasi-heap and sketch based algorithms are more accurate than counter based algorithms. As an example in Figure 10b, SSQ and TwMinSwap find 86 % true frequent items when the space is set to 40000 bytes while FSSQ and λ-HCount find 99 % true frequent items using the same memory budget.

8.3 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 response time.

Figure 11a shows that the performance between the Quasi-heap and the ordinary heap. For example, when stream size is 10^6 and data skew is 1.5, the number of comparisons in Quasi-heap is 278K which is 35 % smaller than 429K in the ordinary heap. The advantage of our Quasi-heap becomes more significant when the data becomes more skewed. Figure 11c

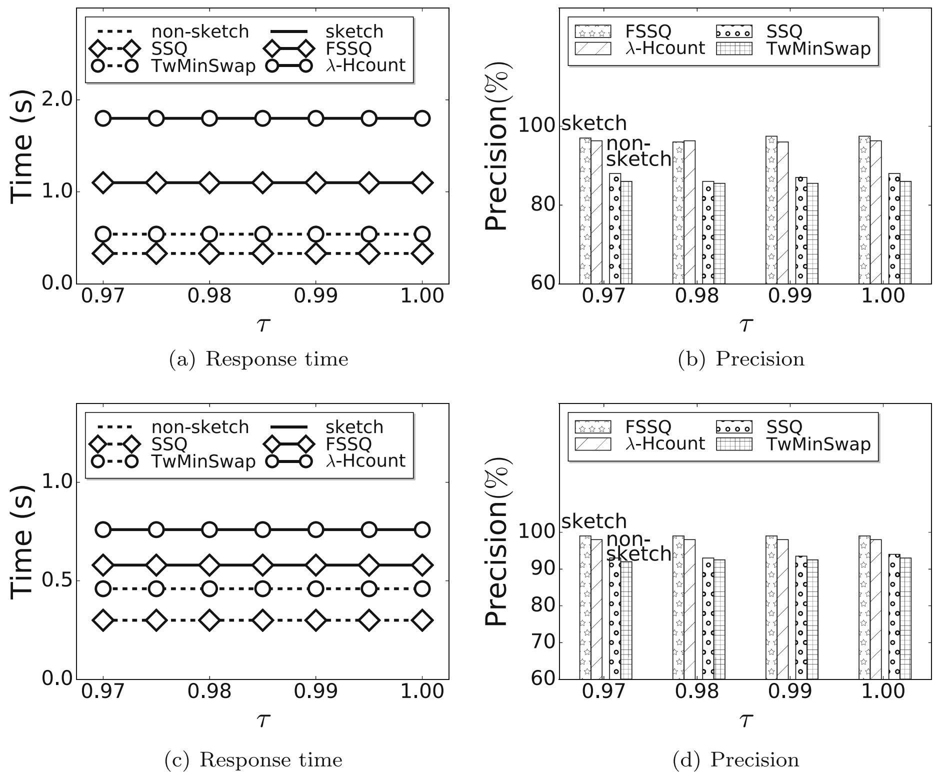


Figure 9 Varying decay rate on Kosarak dataset (**a** and **b**) and Retail datasets (**c** and **d**)

shows that the number of comparisons in Quasi-heap is only 2K when data skew is set to 3.0 which is 118 times smaller than 236K in the ordinary heap. From Figure 11b and d, we can find that the Quasi-heap reduce the response time by up to 80 % since huge amount of unpromising comparisons are delayed by the *delaySorting* method.

8.4 Effectiveness of CMM sketch

We theoretically discussed the estimation error of our improved count-min-min sketch (CMM) and the count-min sketch (CM) in Lemma 6 and Lemma 5, respectively. Next, we verify it by a set of experiments. For a more direct comparison, we use the CM sketch and CMM sketch to estimate the count of an item. The sketch depth is set to $d = 4$ and the width to $w = 2/\phi$, based on the analysis of the CM sketch[11].

Firstly, we define the estimation error rate of these two sketches by the following equation:

$$EER = \frac{Estimated\ count\ by\ sketch - True\ count}{True\ count}. \tag{2}$$

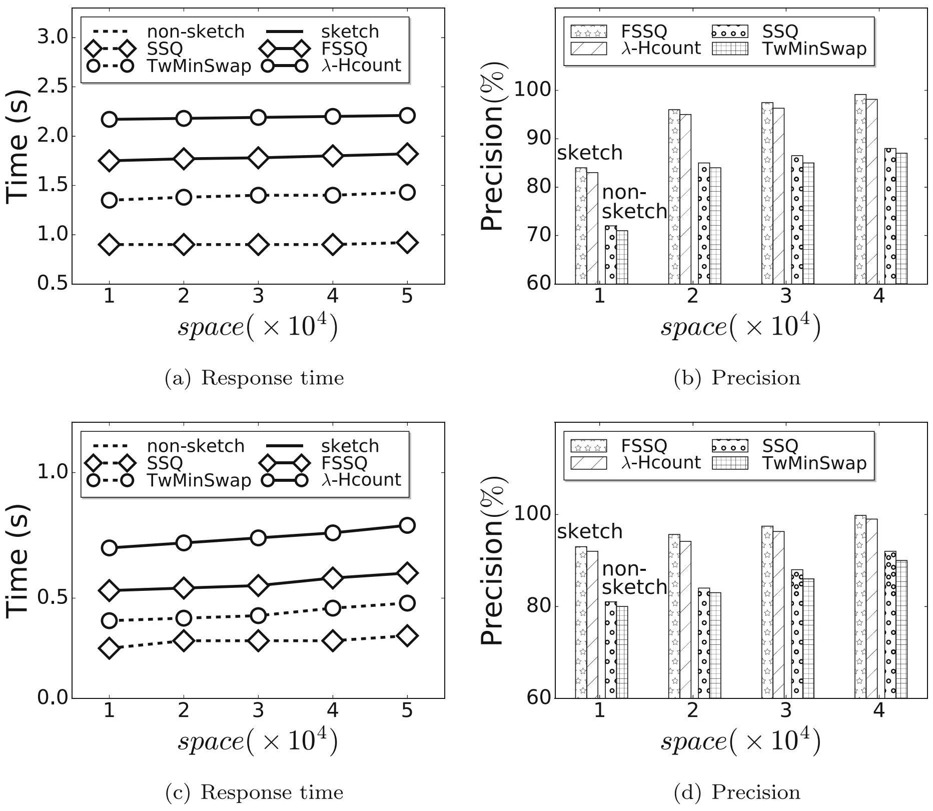


Figure 10 Varying memory space on Synthetic dataset (**a** and **b**) and Retail datasets (**c** and **d**)

Next, we define the reduced error rate by our CMM sketch, compared to CM sketch by the following equation:

$$EER' = \frac{Estimated\ count\ by\ CM - Estimated\ count\ by\ CMM}{Estimated\ count\ by\ CM}. \quad (3)$$

We used the CM sketch and CMM sketch to estimate the count of an item, and the results of the estimated counts and the error rate on Retail dataset and Kosarak dataset are shown in Tables 4 and 5, respectively. It is exciting that the estimated counts by our CMM sketch are always lower than that by the CM sketch. For example, from Table 4, the true count of the item '1' is 266, the estimated count by CM sketch is 3695 and the estimated error rate $ERR = (3695 - 266)/266 = 1289$ %, which is vastly overestimated. However, the estimated count by our CMM sketch is only 272 and the estimated error rate $ERR = (272-266)/266 = 2.25$ %, which is more accurate and acceptable in a data stream environment. The estimated error is reduced by $ERR' = (3695 - 272)/3695 = 92.64$ %,

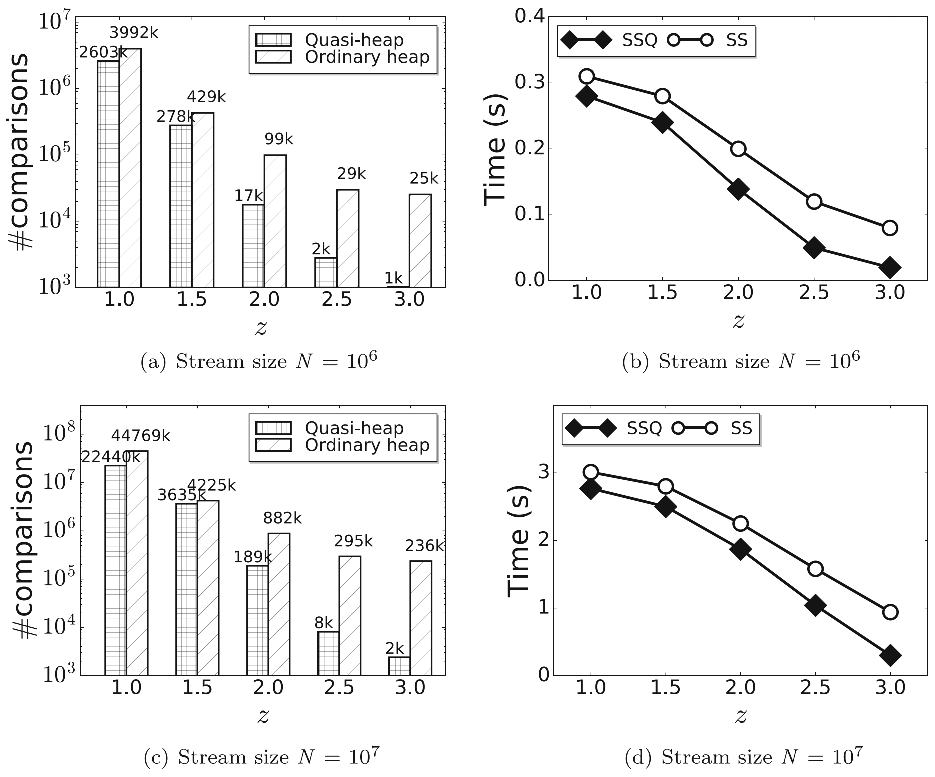


Figure 11 The number of comparisons varying data skew in Ordinary heap and Quasi-heap, respectively

which is a significant improvement. Even more, we can see that the estimated error rate of our CMM sketh EER sometimes reaches 0.

In summary, our CMM sketch can estimate the count of an item with little, and often with no error at all.

Table 4 Estimation error of CM sketch and CMM sketch on Retail

Item ID	True count	Estimated count by CM	EER of CM	Estimated count by CMM	EER of CMM	EER'
1	266	3695	1289 %	272	2.25 %	92.64 %
2	549	1001	83.24 %	562	2.36 %	43.86 %
3	8	213	2562 %	155	1837 %	27.23 %
10	712	801	12.5 %	712	**0**	11.11 %
20	5	63	1160 %	59	1080 %	6.35 %
30	540	612	13.33 %	540	**0**	11.76 %
100	54	198	211 %	161	198 %	18.67 %
200	239	412	72.38 %	240	0.41 %	41.75 %
300	37	91	145 %	79	113 %	13.19 %

Table 5 Estimation error of CM sketch and CMM sketch on Kosarak

Item ID	True count	Estimated count by CM	EER of CM	Estimated count by CMM	EER of CMM	EER'
1	197522	198725	0.61 %	197522	**0**	0.61 %
5	5930	6644	12.04 %	5930	**0**	10.75 %
10	294	2099	613 %	1073	264 %	48.89 %
50	1445	2241	55.08 %	1449	0.27 %	35.34 %
100	963	1350	40.18 %	1001	3.94 %	25.85 %
500	49	582	1087 %	576	1075 %	1.03 %
1000	2266	2916	28.68 %	2266	**0**	22.30 %

9 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. In order to improve the estimation accuracy, we propose a count-min-min (CMM) sketch structure based on the best well-known count-min sketch. We extensively evaluated our methods on three datasets. Our algorithms with the Quasi-heap reduce response time up to 80 % compared with the ordinary heap solutions and the proposed CMM sketch can estimate the count of an item with almost error free.

In the future, we intend to further study how to extend the proposed approaches to a distributed environment to handle greater scales of data streams, when a single machine is no longer capable of managing the large volumes of data and computation.

Acknowledgments This work was supported by National Science and Technology Supporting plan (2014BAK16B02, 2015BAH45F01), the cultural relic protection science and technology project of Zhejiang Province, NSFC 61502548 from NSF of China, grant MYRG2014-00106-FST and MYRG2016-00182-FST from UMAC RC.

References

1. Aouad, L.M., Le-Khac, N.A., Kechadi, T.M.: Performance study of distributed apriori-like frequent itemsets mining. Knowl. Inf. Syst. **23**(1), 55–72 (2010)
2. Boley, M., Grosskreutz, H.: Approximating the number of frequent sets in dense data. Knowl. Inf. Syst. **21**(1), 65–89 (2009)
3. 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)
4. 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)
5. Chang, J.H., Lee, W.S.: Finding recent frequent itemsets adaptively over online data streams. In: SIGKDD, pp. 487–492. ACM (2003)
6. Charikar, M., Chen, K., Farach-Colton, M.: Finding frequent items in data streams. In: Automata, Languages and Programming, pp. 693–703. Springer (2002)
7. Chen, L., Mei, Q.: Mining frequent items in data stream using time fading model. Inform. Sci. **257**, 54–69 (2014)
8. 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)

9. Chen, L., Zou, L.J., Tu, L.: A clustering algorithm for multiple data streams based on spectral component similarity. Inform. Sci. **183**(1), 35–47 (2012)
10. Cormode, G., Hadjieleftheriou, M.: Finding the frequent items in streams of data. Commun. ACM **52**(10), 97–105 (2009)
11. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms **55**(1), 58–75 (2005)
12. Cormode, G., Muthukrishnan, S.: What's hot and what's not: tracking most frequent items dynamically. ACM Trans. Database Syst. **30**(1), 249–278 (2005)
13. 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)
14. Floyd, R.W.: Algorithm 245: Treesort. Commun. ACM **7**(12), 701 (1964)
15. Golab, L., DeHaan, D., Demaine, E.D., Lopez-Ortiz, A., Munro, J.I.: Identifying frequent items in sliding windows over on-line packet streams. In: SIGCOMM, pp. 173–178. ACM (2003)
16. Homem, N., Carvalho, J.P.: Finding top-k elements in data streams. Inform. Sci. **180**(24), 4958–4974 (2010)
17. Jin, C., Qian, W., Sha, C., Yu, J.X., Zhou, A.: Dynamically maintaining frequent items over a data stream. In: CIKM, pp. 287–294. ACM (2003)
18. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst. **28**(1), 51–55 (2003)
19. Li, H.F., Huang, H.Y., Lee, S.Y.: Fast and memory efficient mining of high-utility itemsets from data streams: with and without negative item profits. Knowl. Inf. Syst. **28**(3), 495–522 (2011)
20. 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)
21. Lin, Z., Jiang, B., Pei, J., Jiang, D.: Mining discriminative items in multiple data streams. World Wide Web Journal **13**(4), 497–522 (2010)
22. 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)
23. Manku, G.S., Motwani, R.: Approximate Frequency Counts over Data Streams. In: VLDB, pp. 346–357. VLDB Endowment (2002)
24. 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 Publ (2012)
25. Metwally, A., Agrawal, D., Abbadi, A.E.: An integrated efficient solution for computing frequent and top-k elements in data streams. ACM Trans. Database Syst. **31**(3), 1095–1133 (2006)
26. Shaker, A., Senge, R., Hüllermeier, E.: Evolving fuzzy pattern trees for binary classification on data streams. Inform. Sci. **220**, 34–45 (2013)
27. Tantono, F.I., Manerikar, N., Palpanas, T.: Efficiently discovering recent frequent items in data streams. In: Scientific and Statistical Database Management, pp. 222–239. Springer (2008)
28. Tong, Y., Zhang, X., Chen, L.: Tracking frequent items over distributed probabilistic data. World Wide Web Journal, 1–26 (2015)
29. Wei, Z., Liu, X., Li, F., Shang, S., Du, X., Wen, J.: Matrix sketching over sliding windows. In: SIGMOD, pp. 1465–1480 (2016)
30. Woo, H.J., Lee, W.S.: Estmax: Tracing maximal frequent item sets instantly over online transactional data streams. IEEE Trans. Knowl. Data Eng. **21**(10), 1418–1431 (2009)
31. Wu, S., Lin, H., U, L.H., Gao, Y., Lu, D.: Finding frequent items in time decayed data streams. In: Apweb, pp. 17–29 (2016)
32. 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)
33. Zhang, S., Chen, L., Tu, L.: Frequent items mining on data stream using hash-table and heap. In: ICIS, vol. 1, pp. 141–145. IEEE (2009)