Information Processing Letters 110 (2010) 257–260

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Finding frequent items over sliding windows with constant update time

Regant Y.S. Hung *, Lap-Kei Lee, H.F. Ting [1]

Department of Computer Science, The University of Hong Kong, Hong Kong

ARTICLE INFO

Article history:
Received 13 May 2008
Received in revised form 9 January 2009
Accepted 19 January 2009
Available online 29 January 2010
Communicated by K. Iwama

Keywords:
Approximation algorithms
On-line algorithms

ABSTRACT

In this paper, we consider the problem of finding ϵ-approximate frequent items over a sliding window of size N. A recent work by Lee and Ting (2006) [7] solves the problem by giving an algorithm that supports $O(\frac{1}{\epsilon})$ query and update time, and uses $O(\frac{1}{\epsilon})$ space. Their query time and memory usage are essentially optimal, but the update time is not. We give a new algorithm that supports $O(1)$ update time with high probability while maintaining the query time and memory usage as $O(\frac{1}{\epsilon})$.

1. Introduction

In many applications such as network monitoring, telecommunications and financial monitoring, data arises in the form of a continuous stream of data items. For example, a flow in a network is a stream of TCP/UDP packets containing source/destination addresses. One of the problems that are important to these applications is finding the frequent items in a data stream. In the past, a lot of effort has been devoted to finding frequent items in the whole data stream seen thus far (e.g., [2–6,8,9]).

Data items are time-sensitive in most applications; we are only interested in identifying those frequent items in a sliding window covering the most recently seen N items of the data stream. Arasu and Manku [1] is the first to consider the following *ϵ-approximate frequent items problem* over a sliding window of size N: Let θ and ϵ be a user-specified threshold and a relative error bound respectively. We are asked to maintain some data structure that allows us to produce, at any time, a set B of items that satisfies the following properties: (1) B only contains items that occur at least $(\theta - \epsilon)N$ times in the sliding window, and (2) any item that occurs more than θN times in the sliding window must be in B.

Due to practical concern, an algorithm for data stream processing has to (1) process the data stream in only one pass, (2) use a small amount of memory, and (3) has small query and update times. For the ϵ-approximate frequent items problem over sliding windows, Arasu and Manku [1] gave an algorithm that supports $O(\frac{1}{\epsilon}\log\frac{1}{\epsilon})$ query and update time, and uses $O(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon})$ space. Recently, Lee and Ting [7] gave a simpler algorithm that reduces the query and update time to $O(\frac{1}{\epsilon})$, and the memory usage to $O(\frac{1}{\epsilon})$ space.

The query time and memory usage of the algorithm by Lee and Ting [7] are essentially optimal, while the update time is not. Their update time depends on ϵ, and more precisely, is $\Theta(\frac{1}{\epsilon})$. This motivates us to devise algorithms to reduce the update time to $O(1)$ while keeping the query time and space requirement in $O(\frac{1}{\epsilon})$. Demaine et al. [3] and Karp et al. [6] considered the ϵ-approximate frequent items problem over the whole data stream, instead of sliding window. We note that they proposed an interesting technique such that their data structures support constant update time with high probability. Roughly speaking, their approach can work on counters of the data items, where each counter only decreases at most one in any update. However, this technique cannot be applied directly to the algorithm in [7] for sliding windows; in an update, some

* Corresponding author.
E-mail addresses: yshung@cs.hku.hk (R.Y.S. Hung), lklee@cs.hku.hk (L.-K. Lee), hfting@cs.hku.hk (H.F. Ting).

[1] This research was supported in part by Hong Kong RGC Grant HKU-7163/07E.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2009.01.027

counter may decrease by a value depending on ϵN, which is much greater than one.

Our result. In this paper, we give a new algorithm for the ϵ-approximate frequent items problem over sliding windows. Then we show that the above technique by Demaine et al. [3] and Karp et al. [6] can be applied on the algorithm such that it has $O(1)$ update time with high probability, while maintaining the query time and memory usage as $O(\frac{1}{\epsilon})$.

Organization of the paper. In Section 2, we give a new algorithm for finding ϵ-approximate frequent items over a sliding window, which uses $O(\frac{1}{\epsilon})$ space and supports $O(\frac{1}{\epsilon})$ update time. In Section 3, we describe the technique adopted by Demaine et al. [3] and Karp et al. [6] for reducing the update time. In Section 4, we show how to extend the algorithm given in Section 2 so that it uses $O(\frac{1}{\epsilon})$ space, supports $O(1)$ update time (with high probability) and $O(\frac{1}{\epsilon})$ query time.

2. An algorithm that supports $O(\frac{1}{\epsilon})$ update time

We now give an algorithm which finds frequent items over a sliding window using $O(\frac{1}{\epsilon})$ space and requiring $O(\frac{1}{\epsilon})$ update time. Note that this algorithm is modified from the algorithm of Lee and Ting [7] substantially. The change is crucial for reducing the update time in Section 4. We will explain the changes throughout this section.

First, the framework of our algorithm differs from the one in [7]. Let $W_{p,q}$ denote the window covering the set of items: from the p-th item to the q-th item for some integer $p \leqslant q$. Recall that *sliding window* is defined as the window covering the N most recently arrived items. Suppose the r-th item is the most recently arrived item. Then $W_{r-N+1,r}$ is the current sliding window. The window slides whenever a new item arrives. In [7], it maintains a set of *window counters* for the current window $W_{r-N+1,r}$. Here, window counter is a data structure that is good in estimating the *frequency* (i.e., number of occurrences) of an item in $W_{p,q}$ for some specific p and q. If the $(r+1)$-st item arrives, it will maintain the window counters for the new window $W_{r-N+2,r+1}$. In our approach, we maintain two sets of window counters at any time, one set for $W_{(i-1)N+1,iN}$ and another set for $W_{iN+1,(i+1)N}$, where i is some integer such that $iN+1 \leqslant r \leqslant (i+1)N$. Such a change is crucial and necessary for reducing the update time to constant in Section 4. We will explain the advantage of this change after describing how to implement a window counter.

2.1. Window counter

Let us describe a useful data structure called window counter, that is proposed in [7]. For any item a and any $i \leqslant j$, let $f_{i,j}(a)$ be the frequency of item a in the window $W_{i,j}$. Window counter is used to estimate the frequency $f_{i,j}(a)$ of an item a for some specific i and j. Consider a window counter C_a of a particular item a. The window counter C_a comprises two variables: ℓ and d, and a queue Q.[2] The counter samples the position of item a with a sampling rate of λ. Suppose the r-th item is the most recently arrived item and $Q = \langle q_1, q_2, \ldots, q_{|Q|}\rangle$ where $q_1 < q_2 < \cdots < q_{|Q|}$. Conceptually, $f_{q_i+1,q_{i+1}}(a) = \lambda$ for any integer $1 \leqslant i < |Q|$ and $f_{q_{|Q|}+1,r}(a) = \ell$. The value of the window counter C_a is defined as $|Q|\lambda + \ell - d$, where we will set d later.

Suppose we keep a counter C_a for item a in the window $W_{p,q}$ where $p < q$. We initialize and maintain C_a as follows.

Initialization: In the beginning, we set ℓ and d to zero and Q is empty.

Update: Suppose the r-th item arrives where $p \leqslant r \leqslant q$. If this item is not a, we do not update C_a. Otherwise, we increase the value of C_a by one as follows: (i) Increase ℓ of C_a by one. (ii) If ℓ equals λ, we reset ℓ to zero and insert one entry r at the front of Q.

Suppose more than q items have arrived in the stream. C_a will not be updated for any item arrived after the q-th item, since C_a is maintained for the window $W_{p,q}$ only as mentioned before. For $p \leqslant k \leqslant q$, the estimated frequency $\hat{f}_{k,q}(a)$ on $f_{k,q}(a)$ is defined as $(n(k)-1)\lambda + \ell$ where $n(k)$ equals to the number of entries in Q whose values are greater or equal to k. (Note that the estimated frequency $\hat{f}_{k,q}(a)$ may be different to the value of counter C_a, which is defined as $|Q|\lambda + \ell - d$.) It is obvious to see that

$$f_{k,q}(a) - \lambda \leqslant \hat{f}_{k,q}(a) \leqslant f_{k,q}(a). \tag{1}$$

Note that compared with the original window counter proposed in [7], there is a critical change in the way we update the window counter: The original window counter needs to remove the "expired entry" (i.e., the positions not in the sliding window) in Q since at any time the algorithm in [7] maintains the set of window counters for the current sliding window, which shifts with the arrival of new items. Our approach does not need to delete the expired entry in Q. Thus, the value of C_a does not change dramatically as the one in [7], in which the value of counter will decrease by λ whenever an expired entry in Q is removed. Our approach ensures that the value of counter will either increase by at most one, or decrease by at most one when an item arrives. Such a property will be found very useful afterwards when we want to reduce the update time.

2.2. Sketch

Suppose the r-th item is the most recently arrived item, and i is the integer such that $iN+1 \leqslant r \leqslant (i+1)N$. Recall that we maintain one set of window counters, or we call it *sketch* for simplicity, for $W_{(i-1)N+1,iN}$ and another sketch (another set of window counters) for $W_{iN+1,(i+1)N}$. Let S_j denote the sketch for $W_{(j-1)N+1,jN}$ for any integer j. Then S_i is used to estimate the frequency $f_{r-N+1,iN}(a)$ of every item a in $W_{r-N+1,iN}$ while S_{i+1} is used to estimate

[2] There is a slight change in the data structure compared with the one in [7]. This change is for simplifying the analysis.

$f_{iN+1,r}(a)$ for every item a. Thus, after the arrival of the (iN)-th item, we do not modify the sketch S_i any more.[3] When the r-th item arrives, only S_{i+1} is updated.

We now describe how to maintain the set of window counters in a sketch for $W_{p,q}$ for some integers $p \leqslant q$ in such a way that we can answer the query correctly at any time. Since memory space is limited, we cannot maintain a counter for every item that may appear in the stream. At any time, we keep at most m window counters for m different items (m will be set later). When an item a' arrives, we first search for the window counter of a'. If this counter exists, we update this counter as described. Otherwise, we initialize the new counter $C_{a'}$ and update it as described. Then if there exists exactly m counters, we carry out a *batch decrement*: we decrease the value of every counter by one. This can be done by increasing the value of d of every counter by one. Any counter whose value equals zero will be removed. After this step, we will have less than m counters again. Suppose at least q items have arrived. When we need to estimate the frequency of a in the window $W_{k,q}$ for any $p \leqslant k \leqslant q$, we check if the counter C_a exists. If C_a does not exist, the estimated frequency of a is zero. Otherwise, the estimated frequency $\hat{f}_{k,q}(a)$ is the same as defined previously (i.e., $(n(k)-1)\lambda + \ell$). Together with inequality (1), we have the following guarantee in our scheme:

$$f_{k,q}(a) - \lambda - d_{\max} \leqslant \hat{f}_{k,q}(a) \leqslant f_{k,q}(a),$$

where $d_{\max}$ denotes the maximum possible number of batch decrements carried out. Then it is obvious to see that $d_{\max} \leqslant \lceil \frac{q-p+1}{m} \rceil$ since each batch decrement decreases the value of m from all the counters and the total amount can be deducted by batch decrements is at most $q-p+1$. Thus, we have

$$f_{k,q}(a) - \lambda - \left\lceil \frac{q-p+1}{m} \right\rceil \leqslant \hat{f}_{k,q}(a) \leqslant f_{k,q}(a). \tag{2}$$

For every sketch, we set $\lambda = \epsilon N/4$ and set $m = 4/\epsilon$.[4] Let $W_{r-N+1,r}$ denote the current sliding window. Recall that we maintain S_i for $W_{(i-1)N+1,iN}$ and S_{i+1} for $W_{iN+1,(i+1)N}$ where $iN+1 \leqslant r \leqslant (i+1)N$. For any item a, its estimated frequency in the sliding window, $\hat{f}_{r-N+1,r}(a)$, is defined as $\hat{f}_{r-N+1,iN}(a) + \hat{f}_{iN+1,r}(a)$. We can calculate the former term from S_i and calculate the latter term from S_{i+1}.

Lemma 1. *Consider any sliding window $W_{r-N+1,r}$. For any item a, we have*

$$f_{r-N+1,r}(a) - \epsilon N \leqslant \hat{f}_{r-N+1,r}(a) \leqslant f_{r-N+1,r}(a).$$

Proof. Note that S_{i+1} is actually maintaining the window counters for $W_{iN+1,r}$. Thus, we can apply inequality (2) for the upper and lower bounds of $\hat{f}_{iN+1,r}(a)$. Together with the facts that $\lambda = \epsilon N/4$ and $m = 4/\epsilon$, the lemma follows. □

Theorem 2. *Let θ be the user-specified threshold. We solve the ϵ-approximate frequent items problem by returning items whose estimated frequencies in the sliding window are no less than $(\theta - \epsilon)N$.*

Proof. For any item a returned, we have $f_{r-N+1,r}(a) \geqslant \hat{f}_{r-N+1,r}(a) \geqslant (\theta - \epsilon)N$. Consider any item e whose actual frequency $f_{r-N+1,r}(e) \geqslant \theta N$. By Lemma 1, $\hat{f}_{r-N+1,r}(e) \geqslant f_{r-N+1,r}(e) - \epsilon N \geqslant (\theta - \epsilon)N$. Therefore, e must be returned, implying all items with a frequency at least θN are returned. □

3. The technique for reducing update time

Note that Demaine et al. [3] and Karp et al. [6] suggested an approach to reduce the update time for finding ϵ-approximate frequent items in the whole data stream. As stated in [3], such a way of implementation supports constant update time with high probability. We describe their approach in this section.

Suppose we maintain a set of $O(\frac{1}{\epsilon})$ counters for estimating the frequency of the items. First, we adopt dynamic perfect hashing in order to keep a hash table of size $O(\frac{1}{\epsilon})$ such that we can access every counter in $O(1)$ time and update the hash table in $O(1)$ time with high probability. Second, we maintain c doubly linked lists, where c is the largest value among all the counter. Counters with the same value are put in the same doubly linked list. We also maintain a linked list $L = (v_1, v_2, \ldots, v_i, \ldots, v_c)$ of c pointers. For any i, pointer v_i points to the doubly linked list of counters with the value i. Moreover, every counter in this linked list have a pointer pointing back to v_i. To increase the value of a counter with value i by one, we use two steps: (i) access the counter by accessing the hash table in $O(1)$ time, and then (ii) remove the counter from the doubly linked list pointed by v_i and add it to the doubly linked list pointed by v_{i+1}. If there is a batch decrement, we will move the head of L to point to v_2 and then delete the pointer v_1. Garbage collection of v_1 and the corresponding doubly linked list will be carried out in subsequent rounds by the system. All these steps are carried out in constant time per item arrival. If in one of the next rounds there is an arrival of an item whose counter should have been removed but not yet garbage collected, the absence of v_1 can tell our algorithm that this counter should have actually been removed. We also update the hash table by removing this item from the hash table. Note that we can also handle such a process of the garbage collection manually so that we do not need to rely on the system for the garbage collection.

Such a data structure supports constant update time, but the space becomes $O(\frac{1}{\epsilon} + c)$ instead of $O(\frac{1}{\epsilon})$ where c is the maximum count of an item. We can have a slight modification in our algorithm such that the space is reduced to $O(\frac{1}{\epsilon})$: we replace any maximal subsequence of L, say $v_i, v_{i+1}, \ldots, v_{i+k}$, where these pointers do not point to any counter, with a pointer from v_{i-1} to v_{i+k+1} with

[3] When there are more than $(i+2)N$ items arrived, we can discard S_i for saving the space because S_i is useless for answering a query afterwards.

[4] For simplicity, we assume that $\epsilon N/4$ and $4/\epsilon$ are integers.

a field length of $k+2$. Then we can get the value of the counter by summing the field lengths of the pointers starting from v_1. It is easy to verify that the update time remains $O(1)$.

4. Algorithm supports constant update time

In this section, we describe how to extend the algorithm given in Section 2 using the technique described in Section 3 such that the extended algorithm supports constant update time with high probability.

We implement our sketch by the following technique. Upon the arrival of the r-th item a,

1. Search for the counter of item a by the hash table. If there is no such a counter in the hash table, we create one for item a;
2. Increase the counter C_a by 1: (i) remove the counter from the doubly linked list pointed by v_i and add it to the one pointed by v_{i+1} where i is the value of the counter originally. (ii) In that counter, we also increase ℓ by one; if ℓ equals λ, we will add one entry r to the queue Q and set ℓ to zero; and
3. Carry out batch decrement if there are exactly $\frac{4}{\epsilon}$ counters: we move the head of L to point to v_2 and then remove v_1 and all the counters in that doubly linked list. These will be garbage collected in the subsequent rounds. Note that we do not update the variable d in every counter since this costs $O(\frac{1}{\epsilon})$ time. In fact, we do not have such a variable d in any counter in our implementation. This variable is just used for conveying the concept easier.

It can be verified easily that the space required is still $O(\frac{1}{\epsilon})$, each update takes $O(1)$ time and Theorem 2 still applies in our new implementation. The following shows that we answer the query correctly in $O(\frac{1}{\epsilon})$ time.

Theorem 3. *We answer the query correctly in $O(\frac{1}{\epsilon})$ time.*

Proof. By Theorem 2, we can answer the query correctly by returning the items whose estimated frequencies are no less than $(\theta-\epsilon)N$. However, checking the estimated frequency of every item takes too long time because there are too many items. Note that we only need to check the items for which a counter is maintained since all other items must not be the answer to the query.

We can have a faster approach as follows. For every counter C_a that is kept for item a in S_{i+1}, we check whether there also exists a counter C_a for a in S_i. This can be done in constant time since we maintain a hash table for the counters kept in any sketch. Note that when we calculate the estimated frequency of an item in the sketch, we need to scan the queue Q of the counter. Thus, for any item a kept in S_{i+1}, we can sum up the estimated frequencies of the item in both sketches in $O(|Q'|+|Q''|)$ time where we let Q' and Q'' denote the queues maintained for a in S_i and S_{i+1} respectively. Since there are at most $O(\frac{1}{\epsilon})$ counters kept in a sketch, we check the hash table for all the counters in S_i for $O(\frac{1}{\epsilon})$ times, and thus, the total time needed is $O(\frac{1}{\epsilon})$. Since there are $O(\frac{1}{\epsilon})$ entries in all the queues in all the counters in every sketch, we use $O(\frac{1}{\epsilon})$ time for estimating the frequencies of all the $O(\frac{1}{\epsilon})$ items kept. So the total time required is $O(\frac{1}{\epsilon})$ time. Similarly, for every counter $C_{a'}$ that is kept for item a' in S_i, we check whether there also exists a counter $C_{a'}$ for a' in S_{i+1}, and get the sum of the estimated frequencies of the item in both sketches. This also takes $O(\frac{1}{\epsilon})$ time. Thus, the total time complexity for answering a query is $O(\frac{1}{\epsilon})$. □

References

[1] A. Arasu, G.S. Manku, Approximate counts and quantiles over sliding windows, in: Proc. Symposium on Principles of Database Systems (PODS), 2004, pp. 286–296.
[2] G. Cormode, S. Muthukrishnan, An improved data stream summary: The count-min sketch and its applications, Journal of Algorithms 55 (1) (2005) 58–75.
[3] E.D. Demaine, A. López-Ortiz, J.I. Munro, Frequency estimation of Internet packet streams with limited space, in: Proc. European Symposium on Algorithms (ESA), 2002, pp. 348–360.
[4] S. Ganguly, A. Majumder, CR-precis: A deterministic summary structure for update data streams, in: Proc. International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE), 2007, pp. 48–59.
[5] L. Golab, D. DeHaan, A. López-Ortiz, E.D. Demaine, Finding frequent items in sliding windows with multinomially-distributed item frequencies, in: Proc. International Conference on Scientific and Statistical Database Management (SSDBM), 2004, pp. 425–426.
[6] R.M. Karp, S. Shenker, C.H. Papadimitriou, A simple algorithm for finding frequent elements in streams and bags, ACM Transactions on Database Systems (TODS) 28 (1) (2003) 51–55.
[7] L.K. Lee, H.F. Ting, A simpler and more efficient deterministic scheme for finding frequent items over sliding windows, in: Proc. Symposium on Principles of Database Systems (PODS), 2006, pp. 290–297.
[8] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proc. Very Large Data Bases (VLDB) Conference, 2002, pp. 346–357.
[9] J. Misra, D. Gries, Finding repeated elements, Science of Computer Programming 2 (1982) 143–152.