Information Sciences 257 (2014) 54–69

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

# Mining frequent items in data stream using time fading model

Ling Chen [a,b,*], Qingling Mei [a]

[a] Department of Computer Science, Yangzhou University, Yangzhou 225127, China
[b] State Key Lab of Novel Software Tech, Nanjing University, Nanjing 210093, China

A R T I C L E I N F O

*Article history:*
Received 24 March 2012
Received in revised form 26 August 2013
Accepted 1 September 2013
Available online 10 September 2013

*Keywords:*
Stream data mining
Frequent data item
Fading factor
Hash function

A B S T R A C T

We investigate the problem of finding frequent items in a continuous data stream, and present an algorithm named $\lambda$-*HCount* for computing frequency counts of stream data based on a time fading model. The algorithm uses $r$ hash functions to estimate the density values of stream data items. To emphasize the importance of recent data items, a time fading factor is used. For a given error bound, our algorithm can detect approximate frequent items under a certain probability using limited number of memory space. The memory requirement only depends on the number of different data items and the number of hash functions used. Experimental results on synthetic and real data sets show that our algorithm outperforms other methods in terms of accuracy, memory requirement, and processing speed.

## 1. Introduction

In recent years, more attentions have been paid to stream data mining. Detecting frequent data items is an important task in stream data analysis. Frequency is a fundamental characteristic in many data mining tasks such as association rule mining and iceberg queries. It has applications in many areas such as sensor data mining, business decision support, analysis of web query logs, direct marketing, network measurement, and internet traffic analysis. Correspondingly, the stream data could be stock tickers, bandwidth statistics for billing purposes, network traffic measurements, web-server click streams, and data from sensor networks.

Traditional mining algorithms assume a finite dataset and multiple scans on the data. For the stream data applications, the volume of data is usually too large to be stored in memory or to be scanned for more than once. For data streams, data items can only be sequentially accessed, and random access is prohibited. Therefore, traditional frequent item mining algorithms are not applicable to stream data. Furthermore, because of the high throughput of the data streams, possibly in the speed of gigabytes per second, any feasible algorithm for detecting frequent data item must perform data processing and query fast enough so as to match the speed of arriving data in the stream. In addition, the algorithm can use only limited memory space and store only the sketch or synopsis of the data items in the stream.

Solutions for finding frequent items in stream data have been proposed recently. Several algorithms use random sampling [10,12,13,16,17,20,25,32,34,39,45] to estimate the frequencies of the data items. For example, the *Sticky Sampling* [34] algorithm presented by Manku and Motwani is a sampling based algorithm for computing an $\varepsilon$-deficient synopsis over a data stream. It is a probabilistic one-pass algorithm that provides an accuracy guarantee on the set of frequent data items and their frequencies reported. The second class of such algorithms are deterministic algorithms [1,5,11,19,27,31,37,38]. The *MG* algorithm by Misra and Gries [37] is a well-known deterministic algorithm to detect frequent stream data.

* Corresponding author at: Department of Computer Science, Yangzhou University, Yangzhou 225127, China. Tel.: +86 514 87870026; fax: +86 514 87887937.
*E-mail addresses:* yzulchen@gmail.com (L. Chen), mql859@163.com (Q. Mei).

0020-0255/$ - see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ins.2013.09.007

In many applications, recent data in the stream is more meaningful. For instance, in an athlete ranking system, more recent records typically should carry more weight. One way to handle such problem is to use a sliding window model [2,4,18,22,30,42]. In this model, only the most recent data items in a time period of a fixed length are stored and processed, and only the frequent data items in this period are detected. The advantage of this method is that it can get rid of the stale data and only consider the fresh data, which are more meaningful in many cases. To emphasize the importance of the recent data, time fading model [8,35,43,44] also can be used for frequency measures in data stream. In this model, data items in the entire stream are taken into account to compute the frequency of each data item, but more recent data items contribute more to the frequency than the older ones. This is achieved by introducing a fading factor $\lambda$ ($0 < \lambda < 1$). A data item which arrived $n$ time points in the past is weighted $\lambda^n$. Thus, the weight is exponentially decreasing. In general, the closer to 1 the fading factor $\lambda$ is, the more important the history is taken into account. There are two advantages of the time fading model over the sliding window model. One is that in the time fading model, frequency takes into account the old data items in the history, while the sliding window model only observes within a limited time window and entirely ignores all the data items outside the window. This is undesirable in many real applications. The second is that in the time fading model, when more data arrive continuously, the frequency changes smoothly without a sudden jump which may occur in the sliding window model.

In real world applications, since there could be a huge number of different data items in the stream, it is impracticable to set a counter for each data item. But when using limited memory space, we cannot keep exact frequency counts for all possible items. Usually, an error bound is predefined by the user, and we can obtain an approximate result using less memory space. Therefore, it is necessary to tackle the problem of finding the right form of data structure and related construction algorithm so that the required frequency counts can be obtained with a bounded error for unbounded input data and limited memory. In solving this problem, we may end up facing a dilemma. That is, by setting a small error bound, we achieve high accuracy but suffer in terms of efficiency. On the contrary, a bigger error bound improves the efficiency but seriously degrades the mining accuracy. Therefore, we need to achieve good balance between the accuracy of the results and memory requirement. In this work, we investigate the problem of detecting approximate frequent items using limited number of memory space under a certain probability.

In this paper, we present an algorithm called *λ-HCount* for computing frequency counts over a user specified threshold on a data stream based on the time fading model. A fading factor $\lambda$ is used to emphasize the importance of the more recent data items. For a given error $\varepsilon$ and a threshold $s$ of density, our algorithm can detect $\varepsilon$-approximate frequent items using $\frac{e(1-\lambda)}{\varepsilon^2}\ln\left(-\frac{M}{\ln p}\right)+\frac{r}{s-\varepsilon}$ memory space under the probability of $p$, here$M$ is the number of different data items, $r$ is the number of hash functions used. Experimental results on synthetic and real data sets show that our algorithm *λ-HCount* outperforms other methods in terms of accuracy, memory requirement, and processing speed.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 formally defines the problem and describes the time fading model. Section 4 describes the framework of our algorithm *λ-HCount* while Section 5 analyzes its space and time complexity. Section 6 reports and analyzes our experimental results and Section 7 gives conclusions.

## 2. Related work

Problems related to frequency estimate have been actively studied. Many algorithms for identifying frequent items and other statistics in the entire data stream have been proposed.

*Lossy Counting* [34] was among the first algorithms for finding frequent items from a data stream. *Lossy Counting* is a one-pass algorithm that provides an accuracy guarantee on the set of frequent data items and their frequencies reported. Given a user-specified support threshold $s$, and an error threshold $\varepsilon$, *Lossy Counting* guarantees that: (1) All items whose true frequency exceeds $sn$ are detected, where $n$ is the total number of data items processed. Namely, there are no false negatives. (2) No item whose true frequency is less than $(s\text{-}\varepsilon)n$ is output. (3) The estimated frequency of any item is at most $\varepsilon n$ less than its true frequency. Homem and Carvalho [21] presented an algorithm for identifying the $k$ most frequent elements by merging the commonly used counter-based and sketch-based techniques. The algorithm also provides guarantees on the expected error estimate, order of elements and the stochastic bounds on the error. Karp et al. [27], and Demaine et al. [11] applied a deterministic *MG* algorithm [37] to detect frequent stream data. They reduced the time for processing one data item in *MG* algorithm to $O(1)$ by managing all counters in a hash table. The algorithm can easily be adapted to find $\varepsilon$-approximate frequent items in the entire data stream without making any assumption on the distribution of the item frequencies. This algorithm needs $1/\varepsilon$ counters for the most frequent data items in the stream. Processing the arrival data items entails incrementing or decrementing some counters.

Many algorithms for frequent item detecting use random sampling. They make assumptions on the distribution of the item frequencies, and the quality of their results is guaranteed probabilistically. Whang et al. [39] proposed a probabilistic algorithm to estimate the number of distinct items in a large collection of data in a single pass. Golab et al. [17] gave an algorithm for the case when the item frequencies are multinomial-distributed. Gibbons and Matias [16] presented sampling algorithms to recognize top-$k$ queries. Liu et al. [32] presented an error-adaptive and time-aware maintenance algorithm for frequency counts over data streams. Manku and Motwani [34] advanced a sampling based algorithm called *sticky sampling* for computing an $\varepsilon$-deficient synopsis over a data stream of singleton items. The algorithm scans the stream data and randomly samples the data items based on three user-specified parameters: support $s$, error bound $\varepsilon$, and probability of failure $\delta$. To solve the non-stationary and imbalanced problem in data stream, Ghazikhani et al. [14] presented an online neural network model and two separate cost-sensitive strategies to handle the class imbalance problem.

Many algorithms use hashing technique to map the data items in the stream to a hash table stored in the main memory. Estan and Varghese [12] presented a sampling algorithm and a hash-based algorithm for frequent item detecting. Based on the hashing technique, Charikar et al. presented an algorithm named *Count Sketch*[7], which requires $O(k/\varepsilon^2 \log n)$ memory space and $O(k/\varepsilon^2 \log n)$ computation time to process one data item. The algorithm can output the items with frequency larger than $1/(k+1)$ under the probability of $1-\delta$. If we wish to only find items with count greater than $n/(k+1)$, the space used is $O(k/\varepsilon^2 \cdot \log n/\delta)$. In addition, the algorithm leads directly to a 2-pass algorithm for the problem of estimating the items with the largest change in frequency between two data streams. Cormode et al. proposed an algorithm, called *groupTest*[10], which can further reduce the space complexity. The algorithm needs only $O(k(\log k + \log(1/\delta))(\log m))$ counters to output all items with frequency above $1/k+1$ under the probability of $1-\delta$, here $m$ is the number of different data items. Jin et al. [25] advanced an algorithm *hCount*which uses $O(e/\varepsilon \cdot \ln(-m/\log \delta))$ memory and $O(\ln(-m/\log \delta))$ time for processing each data element. The algorithm can detect the $\varepsilon$-approximate results under the probability of $\delta$. Fang et al. [13] also advanced several algorithms based on hashing to compute iceberg queries, but each requires at least two passes of scans over the data stream.

In addition to randomized algorithms, many deterministic algorithms for detecting frequent item in data stream are also reported. Calders et al. [5] proposed an algorithm for mining frequent items in a data stream. They defined a new frequency measure such that the current frequency of a data item is its maximal frequency over all possible windows in the stream from any time point in the past until the current time. Lin et al. [31] proposed an adaptive frequency counting algorithm to handle bursty data in the stream. They used a feedback mechanism that dynamically adjusts mining speed to cope with the changing arrival rate. Greenwald and Khanna [19] considered the problem of $\varepsilon$-approximate quantitative summaries. Wang et al. [38] proposed an algorithm *EC* to find $\varepsilon$-approximate frequent items in a data stream. The space complexity is $O(\varepsilon^{-1})$ and the processing time for each item is $O(1)$ on average. Moreover, the frequency error bound of the results obtained by the proposed algorithm is $(1-s+\varepsilon)\varepsilon n$.

The algorithms mentioned above do not discount the effect of the old data, all data items in the whole history of the data stream are given equal weights. This is undesirable in solving many application problems where recent data in the stream is more important than the aged one. To emphasize the importance of the recent data, several data mining algorithms over sliding windows are recently proposed. Arasu and Manku [2] gave the first deterministic algorithm for finding $\varepsilon$-approximate frequent items over a sliding window. The algorithm requires $O((1/\varepsilon)\log(1/\varepsilon))$ time for each query/update and uses $O((1/\varepsilon)\log^2(1/\varepsilon))$ space. Their algorithm divides the sliding window into several possibly overlapping sub-windows with different sizes. The algorithm applies the *MG* algorithm to each of these sub-windows to find the frequent items. These sub-windows are organized into levels so that whenever there is a query on the frequent items, one can traverse these sub-windows efficiently to identify the frequent data items. In [23] Hung and Ting studied the space complexity of the $\varepsilon$-approximate quantizing problem, and proved that any comparison-based algorithm for finding $\varepsilon$-approximate quantity in a data stream needs an $\Omega((1/\varepsilon)\log(1/\varepsilon))$ space. Golab et al. [18] proposed an algorithm based on jumping-sliding windows using a pre-specified threshold $1/m$. The algorithm creates a "global counter" for data items in sliding window, along with a "local counter" for data items in each basic window, and records these data items in the basic window. Lee and Ting [30] proposed an approximate frequent stream data mining algorithm which requires $O(1/\varepsilon)$ space. Their algorithm needs $O(1/\varepsilon)$ processing time for frequency updating and query in each time step. Zhang and Guan [42] proposed a stream data frequency estimation algorithm over sliding windows. Their algorithm requires $O(1/\varepsilon)$ memory space and $O(1)$ time for a frequency query or updating. Lam and Calders [29] proposed an algorithm mining top-$k$ frequent items in a data stream with flexible sliding windows.

Another approach to emphasize the importance of the recent data is based on the time fading model [9,36]. In this model, a fading factor $\lambda \in (0,1)$ is used, and a data item which was received $n$ time points in the past is weighted $\lambda^n$. Therefore, all data items in the stream can contribute is weight to the frequency, but more recent data items contribute more than the older ones. Zhang et al. [43] presented an algorithm *FE* (Frequent-Estimating) based on the time fading factor model to detect the frequent data items in a stream using $O(l+\varepsilon^{-1})$ memory space, here $l$ is a constant. The processing time for each data item is $O(1)$. In [44], Zhang et al. also proposed an algorithm using a hash function to record the densities of the data items. Due to the address conflict, the result is much less accurate than that of using multiple hash functions. The algorithm uses a heap of size $\varepsilon^{-1}$ to store the possible frequent data items, and requires $O(\log\varepsilon^{-1})$ time for answering a query. The algorithm in [8] uses a hash function and a queue of size $\varepsilon^{-1}$ to store the possible frequent data items, and the time for answering a query is $O(1)$. In [35], an algorithm using multiple hash functions for detecting frequent items in data stream is presented, but the memory requirement and the accuracy of the results are not analyzed, and the real counts of the frequent items detected cannot be estimated. Other recent works on mining frequent items in data stream have been surveyed in [3,28,33].

Fading factor is also used in algorithms for mining frequent itemsets. For instance, a method named *estDec* [6] is presented to find recent frequent item sets adaptively over an online data stream. A prefix-tree lattice data structure is used for frequency counting and candidate pruning. In [15], a FP-tree-based data structure named *FP-stream* is presented to mine frequent patterns over multiple time granularities.

In some stream data mining research, Chernoff bound is used to estimate the error bound and the complexity of data mining algorithms. For instance, Wong and Fu [40] use Chernoff bound to estimate the error to guarantee the quality of the output by their algorithm for mining the top-$k$frequent itemsets. In [41], Zaki et al. presented a sampling method to mine the frequent data patterns and the association rules in a database. They use Chernoff bound to estimate the upper bound of the sampling size. Jothimani and Thanamani [26] presented a Chernoff bound based algorithm to mine the frequent itemsets in data stream using a sliding window. They use Chernoff bound to determine the length of the window. In this paper, we also use the Chernoff bound for estimating the number of memory requirements by our algorithm *λ-HCount* for frequent data item detecting in data stream.

## 3. Concepts and definitions

In this section we describe a data fading model by using a fading factor $\lambda$ to discount the frequencies of the old data in the stream. We also give a formal definition of the problem of detecting $\varepsilon$-approximate frequent items from data stream. In this paper, we use a standard stream model with discrete time steps labeled as 0, 1, 2, 3, ..., and only one data record $a(t) \in X$ arrives at the $t-$th time step ($t = 1,2,3,\ldots$), where $X = \{x_1, x_2, \ldots, x_m\}$ is a domain containing discrete values.

To emphasize the importance of recent data, we use a fading factor $\lambda \in (0,1)$ in calculating the data items' support counts. For each data item $x$, its support count decreases as $x$ ages. We call such modified support count the density of the data item. In each time step, the density of each data item is reduced by the fading factor $\lambda$.

**Definition 1** (*Density of a data item*). The density of a data item $x \in X$ at time $t$ is defined as

$$D(x,t) = \begin{cases} 0 & t = 0 \\ D(x,t-1)\lambda + \delta(t,x) & \text{otherwise} \end{cases} \tag{1}$$

Here,$\delta(x,t) = \begin{cases} 1 & a(t) = x \\ 0 & \text{otherwise} \end{cases}$, where $a(t)$ is the data item received at time $t$.

From the definition we can see that the density of each data item should be modified in every time step. However, we found that it is unnecessary to update the density values of all data items at every time step. Instead, it is possible to update the density of a data item only when this item is received from the data stream. For each item, the time when it was last received should be recorded. Suppose a data item $x$ is received at current time $t_n$, and the last time when $x$ was received before is $t_s$ ($t_n > t_s$), then the density of $x$ can be updated as follows:

$$D(x,t_n) = D(x,t_s)^{t_n - t_s} + 1 \tag{2}$$

The following lemma shows that summation of densities of all the data items in the stream has its limitation.

**Lemma 1.** *Let X(t) be the set of all the data items that are received at least once from time 0 to t, we have:*

$$(1) \quad \sum_{x \in X(t)} D(x,t) \leqslant \frac{1}{1-\lambda}, \quad \text{for any } t = 1,2,\ldots.$$

$$(2) \quad \lim_{t\to\infty} \sum_{x \in X(t)} D(x,t) = \frac{1}{1-\lambda} \tag{3}$$

Proof of Lemma 1 is shown in the appendix.

Since a data stream may consist of potentially huge volume of data items, the number of data items in the stream could become very large. Therefore, if an algorithm uses the count of the data items to detect the frequent items, the count of each item could overflow. From Lemma 1, we can see that when a fading factor is used, summation of the densities is independent of the number of the data items in the stream, and the density of each data item is within the range of $[0, 1/(1-\lambda)]$ and never overflows.

From Definition 1, we can see that the density of a data item is reduced over time by the fading factor. The aged data items have less density than the new ones. We also can see that densities of the data items are reduced more rapidly if the fading factor is set a smaller value. Setting the value of fading factor $\lambda$ depends on the requirement of the applications. In solving some real world problem, if historical data should have their influence to a certain extent, then $\lambda$ should be assigned a larger value. If the user in application has more interest in the recent data, then $\lambda$ should be assigned a smaller value. In the extreme case, if we set $\lambda = 1$, all the data received from the stream have equal influence on the result; if we set $\lambda = \varepsilon$, which is a positive number very close to zero, then only the most recent datum has influence on the result. The other consideration in setting the value of $\lambda$ is the steadiness of the stream data. If the values of data items in the stream change rapidly, we should assign $\lambda$ a larger value to make the results reflect the changes of the data in the stream. For instance, since the stock market data and the IP addresses accessed in the Internet change very fast, then $\lambda$ should be assigned a larger value in such applications. But for the meteorological and the geological data, since they change gradually, we should assign $\lambda$ a smaller value. Usually, the value of $\lambda$ is set in the range of 0.98 to 1.0 according to the steadiness of the stream data and the requirement of the applications.

Like most previous work, our *λ-Count* algorithm takes two user-specified parameters: a support threshold $s \in (0,1)$, and an error parameter $\varepsilon \in (0,1)$ such that $\varepsilon < s$.

**Definition 2** (*Frequent data item*). Let $s$ be a user specified threshold. At time $t$, a data item $x$ is a frequent one if its density $D(x,t)$ satisfies $D(x,t) \geqslant \frac{s}{1-\lambda}$.

Given $\varepsilon$ as a user specified relative error bound and $\varepsilon < s$, we are asked to detect some data items with density at least $\frac{s-\varepsilon}{1-\lambda}$, which are called $\varepsilon$-approximate frequent items.

Our algorithm outputs a list of $\varepsilon$-approximate frequent items along with their estimated densities and counts. Similar to *Lossy Counting*, the answers produced by our algorithm have the following guarantees:

1. All items whose densities exceed $\frac{s}{1-\lambda}$ will be output, which means there are no false negatives.
2. No item whose density is less than $\frac{s-\varepsilon}{1-\lambda}$ will be output.
3. The estimated density for each item is not greater than its actual density. The difference between the estimated density and the actual one is no more than $\frac{\varepsilon}{1-\lambda}$.

## 4. The algorithm λ-HCount

In this section, we will describe our algorithm *λ-HCount* in detail. The algorithm usesr hash functions $H_1(x), H_2(x), \ldots, H_r(x)$ to estimate the density values of stream data items. Assuming there are $M$ different data items in the stream, each of these $r$ hash functions maps an integer from $[0\ldots M-1]$ to $[1,m]$ uniformly and independently. A two dimensional array $D$ of size $r \times m$ is used to record the densities of the data items. Whenever a data item $x$ arrives in the data stream, $r$ hash function values $H_i(x)$ ($i$ = 1, 2, …, $r$) are calculated and the entries in $D$ involving $x$: $D[i,H_i(x)]$ ($i$ = 1, 2, …, $r$) are updated. We use *FNV*-1 and *FNV*-1*a* hash functions, which are designed to quickly hash lots of data while maintaining a reasonable collision rate. Multiple *FNV* hash functions are designed using different *FNV* offset basis.

In array $D$, each entry $D[i,H_i(x)]$ has two items: $D[i,H_i(x)]\cdot s$ which is the density of $x$, and $D[i,H_i(x)]\cdot t$ which is the last time the value of $D[i,H_i(x)]\cdot s$ was updated. Initial value of $D[i,H_i(x)]\cdot s$ is set as 0. Among these $r$ estimated densities of $x$, the one which is the least recently updated, i.e. the $D[i,H_i(x)]\cdot s$ with least $D[i,H_i(x)]\cdot t$ value, is used to estimate the real frequency of $x$. The structure of array $D$ is shown in Fig. 1.

Suppose $D[k,H_k(x)]\cdot t = \min_{1\leqslant i\leqslant r} D[i,H_i(x)]\cdot t$, then entry $D[k,H_k(x)]$ is the least recent modified one among entries $D[i,H_i(x)]$ ($i$ = 1, 2, …, $r$). Since after time $D[k,H_k(x)]\cdot t$, no data item $x$ was received in the system, all the other entries $D[i,H_i(x)]$ ($i \neq k$) contain no density of $x$ after time$D[k,H_k(x)]\cdot t$. Noticing that the most recent received data will have larger increment of density, $D[k,H_k(x)]\cdot s$ is most close to the real density of $x$. Therefore, we take $minD = D[k,H_k(x)]\cdot s$ as the estimated density of $x$. Since the value of $minD = D[k,H_k(x)]\cdot s$ may include the density values of other items which share the same hash address with $x$, the real density of $x$ is obviously not greater than its estimated value $minD$. Later we will prove that the error between the estimated frequency by algorithm *λ-HCount* and the real one will not exceed $\frac{\varepsilon}{1-\lambda}$ under a predefined probability greater than $p$.

Although the density values stored in array $D$ need to be continually modified, the algorithm *λ-HCount* does not update all the density values in $D$ at every time step. Instead, it updates the density of a data item only when an identical new data item is received from the stream. Therefore the item $D[i,j]\cdot t$ is used to record the last time the value of $D[i,j]\cdot s$ was updated. If at time $t$, $D[i,j]$ receives a new data item, it is updated as follows:

$$D[i,j]\cdot s = D[i,j]\cdot s \times \lambda^{t-D[i,j]\cdot t} + 1 \tag{4}$$

We use a double linked list $F$ to record the frequent items detected. To accelerate the process of updating the list $F$, it is organized as a hash table using a hash function $H$. For a data item $x$, its address in $F$ is $H(x)$. Entries in list $F$ are arranged in a queue structure in the descending order of their $t_x$ values, which is the last time when item $x$ was received. Therefore, the most recently arrived or updated data item is located at the tail of the queue, while the oldest one is at the head. The queue is constructed as a doubly linked list as shown in Fig. 2.

Each entry of list $F$ is a vector $[x, D_x, t_x, p_{succ}, p_{pre}]$, where $D_x$ is the estimated density of the data item $x$ at time $t_x$, $t_x$ is the last time when item $x$ was received, $p_{succ}$ and $p_{pre}$ are pointers to its successor and predecessor respectively.

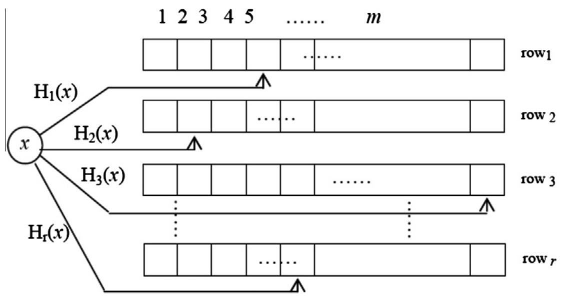

**Fig. 1.** Data structure of array $D$.

As an alternative way, we also can arrange the entries in $F$ according to their density values, and remove the entry with the least density instead of the least $t_x$ value from list $F$ when it is updated.

When a new data record $x$ is received from the stream, the algorithm sequentially calculates the values of $r$ hash functions $H_i\ (x)\ (i = 1,2,\ldots,r)$. Corresponding entries in array $D$ are refreshed accordingly. Then the algorithm obtains $minD = \min_{1\leqslant i\leqslant r}[k, H_i(x)] \cdot s\ [k,H_i(x)]\cdot s$ as the estimated density of $x$. If $D_{\min} > \frac{s-\varepsilon}{1-\lambda}$, the algorithm creates or updates its entry in list $F$. If the entry of $x$ already exits in list $F$,the algorithm modifies its density and $t_x$ value before moving it to the tail of the queue; otherwise, the algorithm creates a new entry for $x$ and inserts it to the tail of the queue. When the list $F$ is full, the algorithm deletes the item at the head of the queue to make room for the entry of new item $x$.

Framework of algorithm *λ-HCount* is as follows:

**Algorithm 1.** *λ-HCount*

```
Input: str: the data stream;
       $\lambda$: fading factor;
       $\varepsilon$: density error bound;
       s: density support threshold;
Output: F: list of frequent data items;
Begin
1. $t = 0$;
2. while not terminate condition do
3.   $t = t + 1$;
4.   receive a data item x from the data stream str; $t_{min} = \infty$
5.   For $k = 1$ to $r$ do
6.     calculate $H_k(x)$;
7.     $D[k, H_k(x)] \cdot s = \mathrm{D}[k, H_k(x)] \cdot s \times \lambda^{t-D[k,H_k(x)]\cdot t} + 1$;   $D[k, H_k(x)] \cdot t = t$
8.     If $D[k,H_k(x)]\cdot t \leqslant t_{\min}$ then $t_{\min} = D[k,H_k(x)]\cdot t$; $D_{min} = D[k,H_k(x)]\cdot s$ endif;
9.  End for;
10.  If $D_{\min} > \frac{s-\varepsilon}{1-\lambda}$ then
11.    If x is in F then change its entry into $(x,D_{min},t)$ and move it to the tail of the queue
12.   else
13.     If F is full then
14.       delete the item at the head of the queue; insert $(x,D_{min},t)$ to the tail of list F
15.     End if
16.   Endif
17  Endif
18 End while
End
```

Although the entries in array $D$ should be modified in every time step, algorithm *λ-HCount* only updates the entries related to the new data item received from the stream at the current time step. Output of the algorithm is the list $F$, which records the potential frequent data items and their densities. Since list$F$ is organized in a hash structure, we can quickly update it and use it to answer a query about the frequentness of the data items.

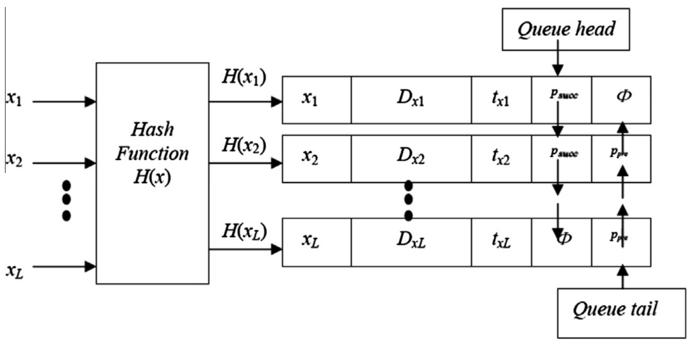

**Fig. 2.** Data structure of the double-linked list $F$.

## 5. Complexity of algorithm $\lambda$-HCount

### 5.1. Complexity of memory requirement

It can be seen from the framework of algorithm *$\lambda$-HCount* that its memory space required is consumed mainly by the arrays $D$ and $F$. We analyze the memory complexity of these arrays in the following theorems.

**Theorem 1.** *To detect all the $\varepsilon$-approximate frequent items in the stream under a predefined probability greater than p, array D in algorithm $\lambda$-HCoun t requires $\frac{e(1-\lambda)}{\varepsilon^2}\ln\left(-\frac{M}{\ln p}\right)$ memory space.*

Proof of Theorem 1 is shown in the appendix.

The values of $r$, $m$ determined by Theorem 1 guarantee that the probability for the error of the estimated densities of all the data items being less than $\frac{\varepsilon}{1-\lambda}$ is greater than $p$.

**Theorem 2.** *The list F requires at most $\frac{r}{s-\varepsilon}$ entries to store all the items with densities greater than $\frac{s-\varepsilon}{1-\lambda}$.*

Proof of Theorem 2 is shown in the appendix.

From Theorems 1 and 2 we know that algorithm *$\lambda$-HCount* requires at most $\frac{e(1-\lambda)}{\varepsilon^2}\ln\left(-\frac{M}{\ln p}\right)+\frac{r}{s-\varepsilon}$ memory space to detect all the $\varepsilon$-approximate frequent items in the stream under a probability greater than $p$. Since the memory requirement is independent of the length of the stream, algorithm *$\lambda$-HCount* always uses constant memory space despite of the unlimited size of the stream data.

### 5.2. Time complexity

When a data item is received from the stream, the algorithm should calculate $r$ hash functions and update $r$ entries in array $D$. Since $r = \mathrm{O}\left(\ln\left\lceil\frac{-M}{\ln p}\right\rceil\right)$, the time needed to update the hash table $D$ is $\mathrm{O}\left(\ln\left\lceil\frac{-M}{\ln p}\right\rceil\right)$ in processing one data item. When updating the list $F$ for a data item $x$, the algorithm first checks if it is already in the list by calculating a hash function, which can be finished in $O(1)$ time. Now we analyze the time complexity of the operations on the double linked list $F$:

- Delete an entry (Line 14 of Algorithm 1). Since we always delete the entry at the head of the queue, the time cost is $O(1)$.
- Insert an entry (Line 14 of Algorithm 1). Since we always insert the entry to the tail of the queue, the time cost is $O(1)$.
- Update an entry when receiving a new data item (Line 11 of Algorithm 1) from stream. Since we can use the hash function to get the address of the item, the time cost is $O(1)$.
- Move an entry to the tail of the queue (Line 11 of Algorithm 1). Given the doubly linked list structure of $F$, this operation can be done by first saving the vector of the entry to a buffer, deleting the entry from the queue, and then inserting the vector in the buffer to the tail of $F$. Since $F$ is a doubly linked queue, each of those operations costs $O(1)$ time.

Therefore, all the operations on list $F$ in processing a data item $x$can be finished in $O(1)$ time.

### 5.3. Query for Frequent Items

Based on the list $F$ built by *$\lambda$-Hcount* algorithm, an algorithm for answering a query is proposed in this section. In order to query all the $\varepsilon$-approximate frequent items, densities of all the entries in list $F$ should be checked, and the items with densities greater than $\frac{s}{1-\lambda}$ are output. Framework of the query algorithm is as follows:

**Algorithm 2.** *Query*($F$);

```
Input   F: double linked list;
        ε: density error bound;
        s: density support threshold;
Output  R: set of current frequent items;
Begin
1. $R = \Phi$;
2.  for all the entries in list F do
        /* Let the entry in F be $(x,D_x,t_x)$*/
3.    if $D_x\lambda^{t-t_x} > \frac{s}{1-\lambda}$ then
4.      $R = R \cup \{x\}$; estimate the count of x: $C(x,t) = (1-\lambda)tD_x\lambda^{t-t_x}$;
5.    end if
6. endfor
7. Output R
End
```

From the algorithm *Query* we can see that the computation time for answering each query is $\frac{s-\varepsilon}{1-\lambda}$, which is the size of list$F$. Because of the hash structure of list $F$, time for answering a query about the frequentness of a given data item is $O(1)$.

The quality of the output of Algorithm 2 is guaranteed by the following theorem.

**Theorem 3.** *The output of Algorithm 2 can satisfy the following conditions under a probability of p:*

(i) *The estimated density for each item is not greater than its actual density. The error of the estimated density is less than* $\frac{\varepsilon}{1-\lambda}$.
(ii) *All items whose real density exceeds* $\frac{s}{1-\lambda}$ *will be output; there are no false negatives.*
(iii) *No item whose density is less than* $\frac{s-\varepsilon}{1-\lambda}$ *will be output.*

Proof of Theorem 3 is shown in the appendix.

Theorem 3 shows that the algorithm *λ-HCount* can detect all the ε-approximate frequent items in data stream and there is no false negative in its output under a probability $p$.

In some real applications, the count of an item is required instead of its density. Line 4 of algorithm *Query*($F$) estimates the count of a frequent data item $x$ using its density. The following theorem shows the way to estimate the real count of a data item using the density obtained by algorithm *λ-Hcount.*

**Theorem 4.** *Let the count of item x at time t be C(x,t), and* $D_x(t)$ *be the density of x at time t, then we have* $C(x,t) \approx (1-\lambda)tD_x(t)$.

Proof of Theorem 4 is shown in the appendix.

Although several methods for detecting frequent items in data stream based on fading factor are reported recently [8,35,44], our algorithm *λ-HCount* has some advantages. In *λ-HCount* we use the most recently updated density to represent the real density of the estimated item, while in [35] this is simply estimated by the least density among the densities involved. In the model of fading factor, the most recently updated density is more accurate. Algorithms in [8,44] use only one hash function to record the densities of the data items, while in our algorithmλ-HCount,*r*hash functions are used. Due to the address conflict, the result by using one hash function is much less accurate than that of our algorithm *λ-HCount*. In Theorem 3, we theoretically proved that the algorithm *λ-HCount* can satisfy the 3 conditions under a probability of $p$, which ensure that there is no false negative in the results, the error of the results obtained is less than a given bound ε. The method in [8,35,44] has no such analysis and guarantee. Our method can detect the frequent items by estimating the real count of a data item using the most recently updated density, but in [8,35,44], the frequent items are detected only according to their densities. Although the method in [25] also uses multiple hash functions to detect the frequent items in data stream, it treated all the data stream with equal weights, while *λ-HCount* uses a fading factor to emphasize the importance of the more recent data items. In many applications, recent data in the stream is more meaningful. The method in [25] treated all the data in stream with equal weights, it is not suitable for many real applications. Due to the huge amount of data items in the stream, counts of the data in the method of [25] could be very large and cause overflow. But in our methods, since the density has a limitation $\frac{1}{1-\lambda}$, it never overflows despite of the huge amount of data in the stream.

## 6. Experimental results and analysis

We evaluate the quality and efficiency of our *λ-HCount* algorithm through extensive experiments on synthetic and real data sets. We also compare its performance against the algorithms *Frequent-Estimating*(FE) [43], *EC* [38], the fading factor model based *LC* algorithm [34] *λ-LC*, and the sliding window based algorithm by Lee and Ting named *Lee-Count* [30]. We focus on the algorithms' computing time, memory requirement, recall, precision and redundancy in handling data streams. All experiments were conducted on a PC with 512M memory, CPU1.7G, WINDOWS XP operating system, and coded using python2.6. In our experiments, we set parameters $p = 0.96$, $\lambda = 0.99$.

### *6.1. Test on synthetic data set*

We generate four datasets based on Zipf distributions, with parameters 0.5, 0.75, 1, 1.25, respectively. Each dataset contains ten million data records with 50,000 distinct items. We compare *FE*,*EC*, *λ-LC*, *Lee-Count* with *λ-HCount* on two error bound settings, $\varepsilon = 0.0005$ and $\varepsilon = 0.001$. We set parameters $r = 4$, $s = 0.01$. To simulate the real application, our algorithm reads the stream data from the peripheral device (hard disk) instead of storing them in the main memory.

Figs. 3 and 4 compare the memory requirements of the five algorithms. From the figures we can see that when $\varepsilon = 0.0005$, *λ-HCount* requires 81.1% less memory than *FE*, 65.4% less than *EC* and *Lee-Count*, and 86.7% less than *λ-LC* on average. When $\varepsilon = 0.001$, *λ-HCount* requires 71.2% less memory than *FE*, 31.3% less than *EC* and *Lee-Count*, and 54.9% less than *λ-LC* on average. Therefore, *λ-HCount* requires the least memory for all the different settings of Zipf parameters and ε.

From the figures we can see that the memory requirements of the five algorithms have no change for data sets with different Zipf distributions. The reason is that their space complexities are independent of Zipf parameters. The memory requirement of our algorithm *λ-HCount* only depends on the error bound ε, the probability $p$, the number of different data items $M$, and the number of hash functions $r$. It requires the same amount of memory for data sets with different Zipf distributions. Similarly, since the memory requirements of *EC*, *FE*, *Lee-Count* and *λ-LC* depend only on the error bound ε, the Zipf

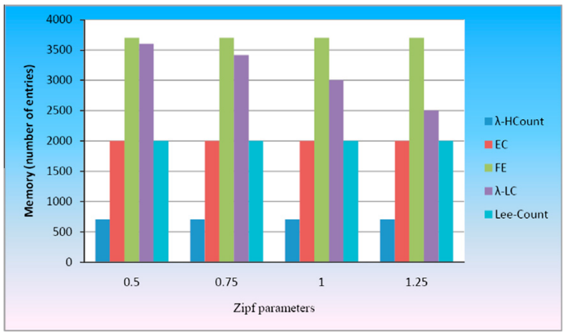

**Fig. 3.** Comparison of the memory requirements of the five algorithms on datasets with different distributions (10 M, $\varepsilon = 0.0005$).

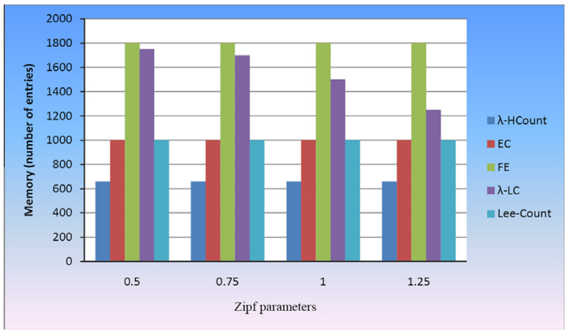

**Fig. 4.** Comparison of the memory requirements of the five algorithms on datasets with different distributions (10 M, $\varepsilon = 0.001$).

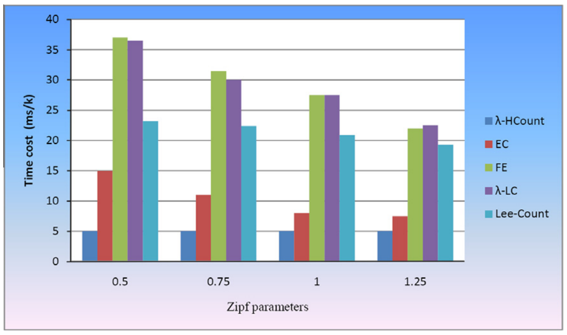

**Fig. 5.** Comparison of time costs of the five algorithms on different distributions of data (10 M, $\varepsilon = 0.0005$).

distribution does not affect their memory requirements. Although memory requirements of the five algorithms do not change for different Zipf distributions, algorithms *EC*, *FE*, *Lee-Count* and *λ-LC* require larger amount of memory than *λ-HCount*. Even though sampling sizes of *EC*, *FE*, *Lee-Count* and *λ-LC* are at least $O(1/\varepsilon)$, the large hidden constants in their space complexities cause greater memory requirements than *λ-HCount*.

Figs. 5 and 6 compare the average processing times by the five algorithms. In the figures, description "*mk/k*" of *y*-axis stands for the time cost (in microsecond) for 1 K (1024) data items by the algorithms. From the figures we can see that when $\varepsilon = 0.0005$, the average time cost of *λ-HCount* is 4.98 microsecond (*ms*) for 1 K data items, while that of *CE, FE, λ-LC* and *Lee-Count* are 10.52, 29.63, 29.38 and 21.37 ms, respectively. When $\varepsilon = 0.001$, the average time cost of *λ-HCount* is 2.43 ms for 1 K data items, while that of *CE, FE, λ-LC* and *Lee-Count* are 5.05, 19.18, 18.87 and 10.82 ms, respectively. Therefore, *λ-HCount* is much faster than the other four algorithms.

Since the algorithm *λ-HCount* uses the data structure of a hash table, it can access the entries in the table very fast. In fact, all the operations on the hash table, such as deleting, inserting, and updating an entry can be completed in $O(1)$ time. Also, since the entries in the table of frequent items are organized in a double linked queue structure, the algorithm can move an entry to the end of the queue in $O(1)$ time. But the algorithms *EC*, *FE*, *Lee-Count* and *λ-LC* need to do multiple decrement

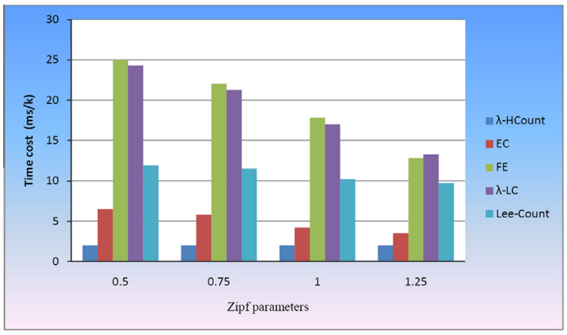

**Fig. 6.** Comparison of time costs of the five algorithms on different distributions of data (10 M, $\varepsilon = 0.001$).

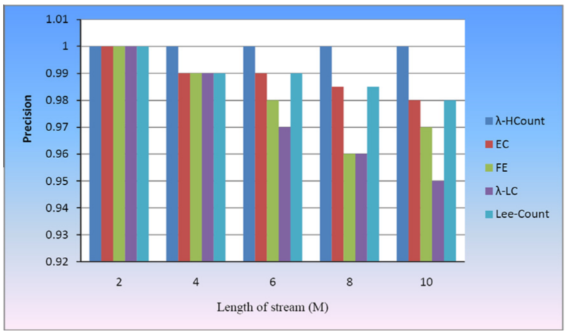

**Fig. 7.** Precisions of five algorithms on Zipf 1.25($\varepsilon = 0.001$).

operations which require at least $O(\varepsilon^{-1})$ or $O(\log\frac{1}{\varepsilon})$ time to process one data item. It could consume large amount of computation time if a small error bound $\varepsilon$ is used. Therefore, *λ-HCount* is much faster than the other four algorithms.

We also test and compare the quality of the results by the five algorithms in terms of recall and precision defined as follows.

$$\text{recall} = \frac{\text{number of the truly frequent items reported by the algorithm}}{\text{number of all the truly frequent items}}$$

$$\text{precision} = \frac{\text{number of the truly frequent items reported by the algorithm}}{\text{number of the frequent items reported by the algorithm}}$$

Since all the algorithms have no false negatives, all the frequent items can be detected, their recalls are all 100%. Fig. 7 shows the precisions of the five algorithms on synthetic data with Zipf parameter 1.25. From the figure we can see that the average precision of *λ-HCount* is close to 100%, while those of *FE*, *CE*, *Lee-Count* and *λ-LC* are 98.1%, 98.6%, 98.7% and 97.4% respectively. Therefore, *λ-HCount* has higher precision than the other four algorithms. From the figure we can also see that all the algorithms can achieve high precision, but precisions of algorithms *FE*, *EC*, *λ-LC* and *Lee-Count* decrease when the length of the stream increases, while *λ-HCount* obtains high precision close to 100% regardless of the length of the data stream.

Since the algorithms generate false positives, we also test the redundancies in the results by the three algorithms, which is defined as

$$\text{redundancy} = \frac{\text{number of the false frequent items reported by the algorithm}}{\text{number of the frequent items reported by the algorithm}}.$$

Table 1 shows the comparison of the redundancies in the results by the five algorithms. From Table 1 we can see that *λ-HCount* has almost no redundancy while redundancies of the other algorithms increase with the larger length of the data stream.

The reason for *λ-HCount* getting high quality results is that it uses the most recently updated density to represent the real density of the estimated item, while in other methods this is simply estimated by the least density among the densities involved. It is obvious that the most recently updated density is more accurate in the model with a fading factor. Although algorithm *FE* also uses the most recently updated density to represent the real density, since it uses only one hash function to record the densities of the data items, its result is much less accurate than that of our algorithm *λ-HCoun* which uses *r* hash functions.

### 6.2. Test on real data set

In this section, we test the algorithms *FE, EC* and *λ-HCount* using the real data set kosarak [24] downloaded from http://fimi.cs.helsinki.fi/data/. The kosarak data set consists of an anonymous click-stream of a Hungarian online news portal, which contains about 8 million individual data items and 41,270 distinct items. In the experiments, we set $r = 8$, $s = 0.005$, $\varepsilon = 0.0005$ and 0.001.

Figs. 8 and 9 show the comparison of the memory requirement of the three algorithms on kosarak data sets with different sizes setting$\varepsilon$ as 0.0005 and 0.001 respectively. From Figs. 8 and 9, we can see that when $\varepsilon = 0.0005$, *λ-HCount* requires 72.2% less memory than *FE* and 62.5% less memory than *EC*.When $\varepsilon = 0.001$, *λ-HCount* requires 46.4% less memory than *FE* and 31.8% less memory than *EC*. Therefore, *λ-HCount* has the least memory requirement.

From the figures we can see that the algorithms require the same amount of memory for equal sized data sets with different Zipf distributions. This is due to the fact that their space complexities only depend on the error bound $\varepsilon$ and other parameters, and are independent of the Zipf distribution. But algorithms*EC* and *FE* consume much more memory space than *λ-HCount* due to the large hidden constants in their space complexities $O(1/\varepsilon)$.

**Table 1**
Redundancy of five algorithms on Zipf 1.25 ($\varepsilon = 0.001$).

| Length of the stream | $2 \times 10^6$ | $4 \times 10^6$ | $6 \times 10^6$ | $8 \times 10^6$ | $10^7$ |
|---|---|---|---|---|---|
| *λ-HCount* | 0.0 | 0.0 | 0.0 | 0.0 | 0.001 |
| *EC* | 0.0 | 0.01 | 0.01 | 0.015 | 0.02 |
| *FE* | 0.0 | 0.01 | 0.02 | 0.04 | 0.03 |
| *λ-LC* | 0.0 | 0.01 | 0.03 | 0.04 | 0.05 |
| *Lee-Count* | 0.0 | 0.01 | 0.01 | 0.015 | 0.02 |

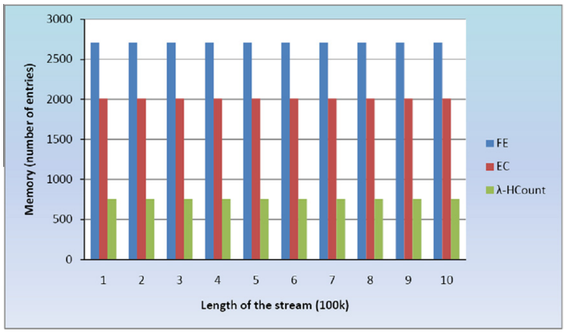

**Fig. 8.** Comparison of the memory requirements of the three algorithms on kosarak data ($\varepsilon = 0.0005$).

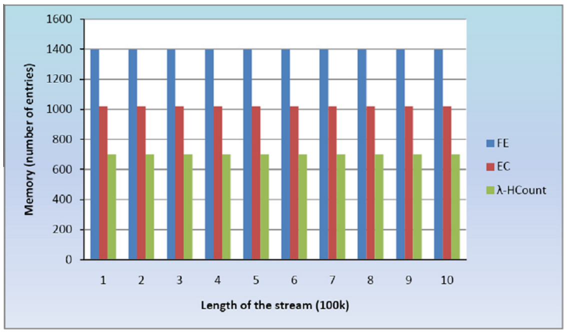

**Fig. 9.** Comparison of the memory requirements of the three algorithms on kosarak data ($\varepsilon = 0.001$).

The time costs (in microsecond) for 1 K (1024) data items by the three algorithms on kosarak data with different $\varepsilon$values are shown in Figs. 10 and 11. From the figures we can see that when $\varepsilon = 0.0005$, the average time cost of *λ-HCount* is 0.412 *ms* for 1 K data items, while that of *FE* and *CE* are 3.42 and 1.57 *ms*, respectively. When $\varepsilon = 0.001$, the average time cost of *λ-HCount* is 0.382 *ms* for 1 K data items, while that of *FE* and *CE* are 2.46 and 1.23 *ms* respectively. Therefore, it is obvious that *λ-HCount* is much faster than *FE* and *EC*.

The reason for *λ-HCount* geting the high speed is that it uses a double linked queue to store the frequent data items, which can accelerate the access to each data item, while *EC*, *FE*, *Lee-Count* and *λ-LC* need to do multiple decrement operations which requires at least O $(1/\varepsilon)$ time.

Our experimental results show that the recalls of the three algorithms on kosarak data are all 100%. In Fig. 12 we show precisions of *λ-HCount*and other two algorithms on kosarak data sets with different sizes. From the figures we can see that the average precision of *λ-HCount* is 0.978%, while that of *FE* and *CE* are 0.886% and 0.946%, respectively. Therefore, we can see that *λ-HCount* has higher precision than the other two algorithms.

Table 2 shows the comparison of the redundancies in the results by the three algorithms. From Table 2 we can see that our algorithm has the lowest redundancy among the three algorithms. This also demonstrates the high quality of the results by *λ-HCount.*

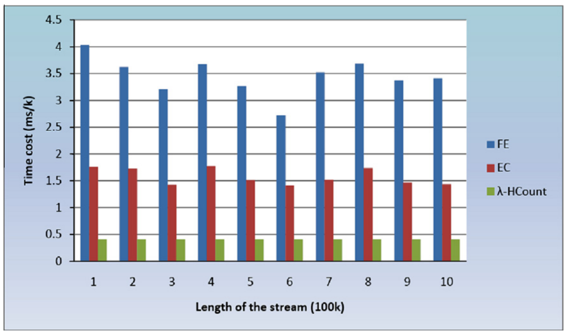

**Fig. 10.** Comparison of time costs of the three algorithms on kosarak data ($\varepsilon = 0.0005$).

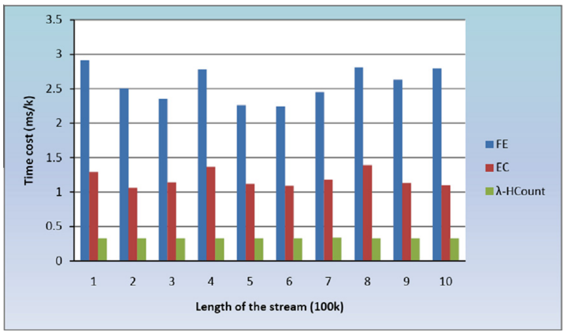

**Fig. 11.** Comparison of the time costs of the three algorithms on kosarak data ($\varepsilon = 0.001$).

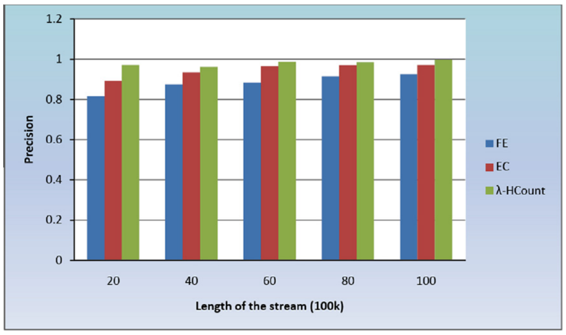

**Fig. 12.** Precisions of three algorithms on kosarak data ($\varepsilon = 0.001$).

**Table 2**
Redundancy of three algorithms on kosarak data ($\varepsilon = 0.001$).

| Length of the stream | $2 \times 10^6$ | $4 \times 10^6$ | $6 \times 10^6$ | $8 \times 10^6$ | $10^7$ |
|---|---|---|---|---|---|
| *λ-HCount* | 0.039 | 0.037 | 0.009 | 0.008 | 0.0 |
| *EC* | 0.108 | 0.087 | 0.041 | 0.015 | 0.014 |
| *FE* | 0.185 | 0.118 | 0.111 | 0.087 | 0.086 |

Results of all the experiments demonstrate that algorithm *λ-HCount* can get higher quality results using much less computation time than other methods. This can be explained by considering three factors: (1) *λ-HCount*uses a hash table to estimate the densities of data items. Since all the operations on an entry in the hash table can be completed in $O(1)$ time, the algorithm can modify the table very fast at every time step. (2) *λ-HCount*also uses a table to store the current frequent items detected. Since the entries in the table are organized in double linked queue, *λ-HCount* can modify any entry in the queue in $O(1)$ time. This accelerates the process on each data item. (3) To approximate the real density of the estimated item, λ-HCount uses the most recently updated density, instead of using the least density among the densities involved. Since the most recently updated density is more accurate, λ-HCount can obtain higher quality results than other methods.

## 7. Conclusions

A new algorithm *λ-HCount* for mining the frequent items in data stream is presented. The algorithm adopts a time fading factor $\lambda$ to emphasize the importance of the more recent data, and records the densities of the stream data items in hash tables so as to mine the frequent items in the data stream. For a given error $\varepsilon$ and a threshold of density $s$, our algorithm can mine $\varepsilon$-approximate frequent items under a probability greater than $p$ using $\frac{e(1-\lambda)}{\varepsilon^2}\ln\left(-\frac{M}{\ln p}\right) + \frac{r}{s-\varepsilon}$ memory space, here$M$ is the number of different data items, $r$ is the number of hash functions used. Experimental results with synthetic and real data sets show that the algorithm *λ-HCount* outperforms other methods in terms of accuracy, memory requirement, and processing speed.

## Acknowledgements

This research was supported in part by the Chinese National Natural Science Foundation under Grant Nos. 61379066, 61070047, 61070133 and 61003180, State Key Fundamentals Research (973) Project under Contract 2012CB316003, Natural Science Foundation of Jiangsu Province under Contracts BK2010318, BK21010134, and Natural Science Foundation of Education Department of Jiangsu Province under Contract 09KJB20013.

## Appendix A. 1. Proof of Lemma 1

**Lemma 1.** *Let $X(t)$ be the set of all the data items that are received at least once from time 0 to t, we have:*

(1) $\sum_{x \in X(t)} D(x,t) \leqslant \frac{1}{1-\lambda}$ *, for any t = 1, 2, ….*
(2) $\lim_{t\to\infty} \sum_{x \in X(t)} D(x,t) = \frac{1}{1-\lambda}$

**Proof.** For a given time $t$, $\sum_{x \in X(t)} D(x,t)$ is the summation of the densities of the $t+1$ data records that arrive at time steps 0, 1, …, $t$. For each time step $t'$, $0 \leqslant t' \leqslant t$, the data record contributes $\lambda^{t-t'}$ to the total density. Therefore, we have:

$$\sum_{x \in X(t)} D(x,t) = \sum_{t'=0}^{t} \lambda^{t-t'} = \frac{1-\lambda^{t+1}}{1-\lambda} \leqslant \frac{1}{1-\lambda}.$$

Also, it is clear that:

$$\lim_{t\to\infty} \sum_{x \in X(t)} D(x,t) = \lim_{t\to\infty} \frac{1-\lambda^{t+1}}{1-\lambda} = \frac{1}{1-\lambda}. \quad \square$$

## Appendix B. 2. Proof of Theorem 1

**Theorem 1.** *To detect all the $\varepsilon$-approximate frequent items in the stream under a predefined probability greater than p, array D in algorithm λ-HCount requires $\frac{e(1-\lambda)}{\varepsilon^2}\ln\left(-\frac{M}{\ln p}\right)$ memory space.*

**Proof.** For a data item $x$, its density can be estimated by $D(1,H_1(x))$, $D(2,H_2(x))$, ..., $D(r,H_r(x))$, where each associated entry contains not only net density $d_x$ for $x$ but also densities of some other items that are mapped to the same associated entry. Let $e_1$, $e_2$, ..., $e_r$ be the errors of the $r$entries for $x$, then we have

$$D(i,H_i(x)) = d_x + e_i \quad (i = 1,\ldots\ldots r)$$

Suppose the size of each hash table is $m$, then array $D$ has $r \times m$ entries. Since all the hash functions are well defined and can equilibriumly map the items into the hash tables, by Lemma 1 we know that the average density mapped to each entry is $\frac{1}{(1-\lambda)m}$. It suggests that the expected value of each error $e_i$ should not exceed $\frac{1}{(1-\lambda)m}$. Let $y$be a random variable indicating the value of the error $e_i$, its expected value should satisfy:

$$E(y) \leqslant \frac{1}{(1-\lambda)m} \tag{a1}$$

If we let $\mu = E(y)$ be the expectation of $y$, and $\delta$ be a constant such that $1 > \delta > 0$, then by Chernoff bound, we know

$$P_r[y > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \tag{a2}$$

By (a1) and (a2) we get

$$P_r\left[y > (1+\delta)\frac{1}{(1-\lambda)m}\right] < \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\frac{1}{(1-\lambda)m}}$$

The above formula shows that the probability for $> y\frac{1+\delta}{(1-\lambda)m}$ in one test is less than $\left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\frac{1}{(1-\lambda)m}}$. If we try $r$ times, the probability for all values being larger than $\frac{1+\delta}{(1-\lambda)m}$ is less than $\left[\left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\frac{1}{(1-\lambda)m}}\right]^r = \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\frac{r}{(1-\lambda)m}}$. In the other words, the probability that at least one of the records is less than $\frac{1+\delta}{(1-\lambda)m}$ is greater than $1 - \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\frac{r}{(1-\lambda)m}}$. Let $y_{min}$ be the minimum value of the $r$ errors, then

$$P_r\left[y_{\min} < \frac{1+\delta}{(1-\lambda)m}\right] > 1 - \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\frac{r}{(1-\lambda)m}}$$

Let $p$ be the predefined probability of the event that all $M$ items satisfy the above formula simultaneously. Then,

$$p = \left(1 - \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\frac{r}{(1-\lambda)m}}\right)^M \approx e^{-\left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\frac{r}{(1-\lambda)m}}.M} \tag{a3}$$

Let $\varepsilon = \frac{1+\delta}{m}$, then (a3) is the probability that the errors of the estimated densities of all the data items are less than $\frac{\varepsilon}{1-\lambda}$. From (10), it can be deduced that

$$\ln p = -\left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\frac{r}{(1-\lambda)m}} \times M$$

Since $\left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\frac{r}{(1-\lambda)m}} = -\frac{\ln p}{M}$,

We have $\frac{r}{(1-\lambda)m} = \frac{\ln\left(-\frac{\ln p}{M}\right)}{\delta-(1+\delta)\ln(1+\delta)} = \frac{\ln\left(-\frac{M}{\ln p}\right)}{(1+\delta)\ln(1+\delta)-\delta}$.

Therefore, $r = \frac{\ln(-\frac{M}{\ln p})\cdot m\cdot(1-\lambda)}{(1+\delta)\ln(1+\delta)-\delta}$.

Since $\varepsilon = \frac{1+\delta}{m}$, $m = \frac{1+\delta}{\varepsilon}$, we get $r = \frac{\ln\left(-\frac{M}{\ln p}\right).(1-\lambda)(1+\delta)}{[(1+\delta)\ln(1+\delta)-\delta]\varepsilon}$.

The number of the entries in $D$ is

$$r \times m = \frac{\ln\left(-\frac{M}{\ln p}\right).(1-\lambda)(1+\delta)^2}{[(1+\delta)\ln(1+\delta)-\delta]\varepsilon^2} \approx \frac{\ln\left(-\frac{M}{\ln p}\right)}{(1+\delta)\ln(1+\delta)}\cdot\frac{(1+\delta)^2}{\varepsilon^2}(1-\lambda) = \frac{\ln\left(-\frac{M}{\ln p}\right)}{\frac{\ln(1+\delta)}{1+\delta}}\cdot\frac{1-\lambda}{\varepsilon^2}$$

$$\approx \frac{e(1-\lambda)}{\varepsilon^2}\ln\left(-\frac{M}{\ln p}\right). \quad \square$$

## Appendix C. 3. Proof of Theorem 2

**Theorem 2.** *The list F requires at most $\frac{r}{s-\varepsilon}$ entries to store all the items with densities greater than $\frac{s-\varepsilon}{1-\lambda}$.*

**Proof.** By Lemma 1 we know that the summation of the densities of all the $M$ data items will not exceed $\frac{1}{1-\lambda}$. For each hash function $H_i(x)$, we have $\sum_{x\in X} D(i, H_i(x)) = \sum_{j=1}^{m} D(i,j) \leqslant \frac{1}{1-\lambda}$. This means summation of all the elements in the $i$th row of $D$ is not greater than $\frac{1}{1-\lambda}$. Assuming that in the $i$th row of $D$ there are $k$ elements with values greater than $\frac{s-\varepsilon}{1-\lambda}$, then $\frac{k(s-\varepsilon)}{1-\lambda} \leqslant \frac{1}{1-\lambda}$, and hence $k \leqslant \frac{1}{s-\varepsilon}$. It suggests that for $r$hash functions, there are at most $\frac{r}{s-\varepsilon}$ elements in $D$ with values greater than $\frac{s-\varepsilon}{1-\lambda}$. Therefore, list $F$ requires at most $\frac{r}{s-\varepsilon}$ memory space to store all the items with densities greater than $\frac{s-\varepsilon}{1-\lambda}$. □

## Appendix D. 4. Proof of Theorem 3

**Theorem 3.** *The output of Algorithm 2 can satisfy the following conditions under a probability of p:*

(i) *The estimated density for each item is not greater than its actual density. The error of the estimated density is less than $\frac{\varepsilon}{1-\lambda}$.*
(ii) *All items whose real density exceeds $\frac{s}{1-\lambda}$ will be output; there are no false negatives.*
(iii) *No item whose density is less than $\frac{s-\varepsilon}{1-\lambda}$ will be output.*

**Proof**

(i) Since the estimated density of each data item $x$ in list $F$ may include densities of other data items which share the same hash address, the estimated density of$x$ is not greater than its real density. By Theorem 1 we can see that the error of the estimated density is less than $\frac{\varepsilon}{1-\lambda}$ under a probability of $p$.
(ii) Let the real and estimated densities of data item $x$ be $d_x$ and $d_e$ respectively. If $d_x > \frac{s}{1-\lambda}$, by (i) we know that $d_e \geqslant d_x > \frac{s}{1-\lambda}$ under a probability of $p$. Since the estimated density $d_e > \frac{s}{1-\lambda}$, $x$ must be output by Algorithm 2.
(iii) Let $d_e$ be the estimated density of $x$, by (i) we know that $d_e - d_x < \frac{\varepsilon}{1-\lambda}$ under a probability of $p$. If the real density of $x$satisfies $d_x < \frac{s-\varepsilon}{1-\lambda}$, then $d_e < d_x + \frac{\varepsilon}{1-\lambda} < \frac{s-\varepsilon+\varepsilon}{1-\lambda} = \frac{s}{1-\lambda}$, $x$ will not be output by Algorithm 2. □

## Appendix E. 5. Proof of Theorem 4

**Theorem 4.** *Let the count of item x at time t be C(x,t), and $D_x(t)$ be the density of x at time t, then we have $C(x,t) \approx (1-\lambda)tD_x(t)$.*

**Proof.** By Theorem 3, we know $D_x(t) \approx \sum_{k=1}^{t} c_k \lambda^k$ with an error less than $\frac{\varepsilon}{1-\lambda}$, where $c_k$ is the real count of $x$ at time $k$:

$$c_k = \begin{cases} 1 & a(t) = x \\ 0 & \text{otherwise} \end{cases}.$$

Let $C_m(x,t) = C(x,t)/t$ be the average count of item $x$ at time $t$. Then we have $E\left[\sum_{k=1}^{t} c_k \lambda^k\right] = \sum_{k=1}^{t} E[c_k \lambda^k] = \sum_{k=1}^{t} E[c_k]\lambda^k = \sum_{k=1}^{t} C_m(x,t)\lambda^k = C_m(x,t)\sum_{k=1}^{t}\lambda^k = C_m(x,t)\frac{1-\lambda^t}{1-\lambda} \leqslant \frac{C_m(x,t)}{1-\lambda}$. Since $E\left[\sum_{k=1}^{t} c_k \lambda^k\right] \approx D_x(t)$, we get $\frac{C_m(x,t)}{1-\lambda} \approx D_x(t)$ and $C_m\ (x,t) \approx (1-\lambda)D_x(t)$. Noticing that $C_m(x,t) = C(x,t)/t$, we have $C(x,t) \approx (1-\lambda)tD_x\ (t)$. □

## References

[1] H. Akcan, A. Astashyn, H. Brönnimann, Deterministic algorithms for sampling count data, Data Knowledge Engineering 64 (2008) 405–418.
[2] A. Arasu, G. Manku, Approximate counts and quantiles over sliding windows, in: Proceedings of the 23rd ACM Symposium on Principles of Database Systems, 2004, pp. 286–296.
[3] R. Akbarinia, E. Pacitti, P. Valduriez, Best position algorithms for efficient top-k query processing, Information Systems 36 (2011) 973–989.
[4] C. Busch, S. Tirthapura, A deterministic algorithm for summarizing asynchronous streams over sliding windows, in: Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science (STACS 2007), Aachen, Germany, 2007, pp. 465–476.
[5] T. Calders, N. Dexers, B. Goethals, Mining frequent itemsets in a stream, in: Proceedings of the Seventh IEEE International Conference on Data Mining, 2007, pp. 83–92.
[6] J.H. Chang, W.S. Lee, Finding recent frequent itemsets adaptively over online data streams, in: ACM SIGKDD Conference, 2003, pp. 487–492.
[7] M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in: P. Widmayer, F.T. Ruiz, R.M. Bueno, M. Hennessy, S. Eidenbenz, R. Conejo (Eds.), Proc. of the Int'l Colloquium on Automata, Languages and Programming, Springer-Verlag, Malaga, 2003, pp. 693–703.
[8] L. Chen, S. Zhang, L. Tu, An algorithm for mining frequent items on data stream using fading factor, in: Proceedings of The IEEE International Computer Software and Applications Conference, 2009, pp.171–177.
[9] Y. Chen, L. Tu, Density-based clustering for real-time stream data, in: Proceedings of The Thirteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD-07), 2007. pp. 133–142.
[10] G. Cormode, Graham, Hadjieleftheriou, Marios, Finding the Frequent Items in Streams of Data, Communications of the ACM 52 (2009) 97–105.

[11] E.D. Demaine, A. Lopez-Ortiz, J.I. Munro, Frequency estimation of internet packet streams with limited space, in: Proceeding of the 10th Annual European Symposium on Algorithms, 2002, pp. 348–360.
[12] C. Estan, G. Varghese, New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice, ACM Transactions on Computer System 21 (2003) 270–313.
[13] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani J.D. Ullman, Computing iceberg queries efficiently, in: Proceedings of 24th International Conference on Very Large Data Bases, New York, USA, 1998, pp. 299–310.
[14] A. Ghazikhani, R. Monsefi, H.S. Yazdi, Online neural network model for non-stationary and imbalanced data stream classification, International Journal of Machine Learning and Cybernetics (2013), http://dx.doi.org/10.1007/s13042-013-0180-6.
[15] C. Giannella, J. Han, J. Pei, X. Yan, and P.S. Yu, Mining Frequent Patterns in Data Streams at Multiple Time Granularities. AAAI/MIT, 2003.
[16] P.B. Gibbons, Y. Matias, New sampling-based summary statistics for improving approximate query answer, in: Proc. SIGMOD, 1998, pp. 331–341.
[17] L. Golab, D. DeHaan, A. Lopez-Ortiz, E.D. Demaine. Finding frequent items in sliding windows with multinomially-distributed item frequencies, in: Proceedings of the 16th International Conference on Scientific and Statistical Database Management, 2004, pp. 425–426.
[18] L. Golab, D. DeHaan, E.D. Demaine, A. Lopez-Ortiz, J.I. Munro. Identifying frequent items in sliding windows over on-line packet streams, in: Proceedings of the Internet Measurement Conference, 2003, pp. 173–178.
[19] M. Greenwald, S. Khanna, Space-efficient online computation of quantile summaries, in: Proc. SIGMOD, 2001, pp. 58–66.
[20] Ming He, Yong-ping Du, p-top-*k* queries in a probabilistic framework from information extraction models, Computers, Mathematics with Applications 62 (2011) 2755–2769.
[21] N. Homem, J.P. Carvalho, Finding top-k elements in data streams, Information Sciences 180 (2010) 4958–4974.
[22] N. Homem, J.P. Carvalho, Finding top-k elements in a time-sliding window, Evolving Systems 2 (2011) 51–70.
[23] R.Y.S. Hung, H.F. Ting, An $\Omega(1/\varepsilon\log(1/\varepsilon))$ space lower bound for finding $\varepsilon$-approximate quantiles in a data stream, Lecture Notes in Computer Science 6213 (2010) 89–100.
[24] http://fimi.cs.helsinki.fi/data/.
[25] C. Jin, W. Qian, C. Sha, J.X. Yu, A. Zhou, Dynamically maintaining frequent items over a data stream, in: J. Carbonell (Ed.), Proc. of the 2003 ACM CIKM Int'l Conf. on Information and Knowledge Management, ACM Press, New Orleans, 2003, pp. 287–294.
[26] K. Jothimani, A.S. Thanamani, CB based approach for mining frequent itemsets, International Journal of Modern Engineering Research (IJMER) 2 (4) (2012) 2508–2511.
[27] R.M. Karp, S. Shenker, C.H. Papadimitriou, A simple algorithm for finding frequent elements in streams and bags, ACM Transactions on Database Systems 28 (2003) 51–55.
[28] B. Lahiri, S. Tirthapura, Identifying frequent items in a network using gossip, Journal of Parallel and Distributed Computing 70 (2010) 1241–1253.
[29] H.T. Lam, T. Calders, Mining top-k frequent items in a data stream with flexible sliding windows, in: Proceedings of KDD'10, ACM Press, Washington, DC, 2010, pp. 283–291.
[30] L. Lee, H. Ting, A simpler and more efficient deterministic scheme for finding frequent items over sliding windows, in: Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM Press, Chicago, USA, 2004, pp. 290–297.
[31] B. Lin, W.S. Ho, B. Kao, Chun-Kit Chui, Adaptive frequency counting over bursty data streams, in: Proceedings of the 2007 IEEE Symposium on Computational Intelligence and data mining, 2010, pp. 516–523.
[32] H. Liu, Y. Liu, J. Han, J. He, Error-adaptive and time-aware maintenance of frequency counts over data streams, in: Proceeding of WAIN 2006, Lecture Notes on Computer Science, vol. 4016, 2006, pp. 484–495.
[33] N. Manerikar, T. Palpanas, Frequent items in streaming data: an experimental evaluation of the state-of-the-art, Data and Knowledge Engineering 68 (2009) 415–430.
[34] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proc. of 28th Intl. Conf on Very Large Data Bases, 2002, pp. 346–357.
[35] Q.L. Mei, L. Chen, An algorithm for mining frequent stream data items using hash function and fading factor, Applied Mechanics and Materials (2012) 130–134.
[36] L. Tu, Y. Chen, Stream data clustering based on grid density and attraction, ACM Transactions on Knowledge Discovery from Data 3 (2009) 1–26.
[37] J. Misra, D. Gries, Finding repeated elements, Science of Computer Programming 2 (1982) 143–152.
[38] W.P. Wang, J.Z. Li, D.D. Zhang, L.J. Guo, An efficient algorithm for mining approximate frequent item over data streams, Journal of Software 18 (2007) 884–892.
[39] K.Y. Whang, B.T. Vander-Zanden, H.M. Taylor, A linear-time probabilistic counting algorithm for a database applications, ACM Transactions Database Systems 15 (1990) 208–229.
[40] R.C.W. Wong, A.W.C. Fu, Mining top-*K*frequent itemsets from data streams, Data Mining and Knowledge Discovery 13 (2006) 193–217.
[41] M.J. Zaki, S. Parthasarathy, W. Li, M. Ogihara, Evaluation of Sampling for Data Mining of Association Rules, Technical Report 617, Computer Science Department, University of Rochester, 1996.
[42] L. Zhang, Y. Guan, Frequency estimation over sliding windows, in: Proceedings of SIGKDD, 2007, pp. 1385–1387.
[43] S. Zhang, L. Chen, L. Tu, Frequent items mining on data stream based on time fading factor, in: Proceedings of 2009 International Conference on Artificial Intelligence and Computational Intelligence, 2009, pp. 336–340.
[44] S. Zhang, L. Chen, L. Tu, Frequent items mining on data stream using hash-table and heap, in: Proceedings of The IEEE International Conference on Intelligent Computing and Intelligent Systems, 2009, pp. 141–145.
[45] Y. Zhang, W. Zhang, X. Lin, B. Jiang, J. Pei, Ranking uncertain sky: the probabilistic top-k skyline operator, Information Systems 36 (2011) 898–915.