Data Min Knowl Disc (2009) 19:132–172
DOI 10.1007/s10618-009-0129-2

# A novel hash-based approach for mining frequent itemsets over data streams requiring less memory space

**En Tzu Wang · Arbee L. P. Chen**

Received: 1 April 2008 / Accepted: 2 March 2009 / Published online: 17 March 2009
Springer Science+Business Media, LLC 2009

**Abstract** In recent times, data are generated as a form of continuous data streams in many applications. Since handling data streams is necessary and discovering knowledge behind data streams can often yield substantial benefits, mining over data streams has become one of the most important issues. Many approaches for mining frequent itemsets over data streams have been proposed. These approaches often consist of two procedures including continuously maintaining synopses for data streams and finding frequent itemsets from the synopses. However, most of the approaches assume that the synopses of data streams can be saved in memory and ignore the fact that the information of the non-frequent itemsets kept in the synopses may cause memory utilization to be significantly degraded. In this paper, we consider compressing the information of all the itemsets into a structure with a fixed size using a hash-based technique. This hash-based approach skillfully summarizes the information of the whole data stream by using a hash table, provides a novel technique to estimate the support counts of the non-frequent itemsets, and keeps only the frequent itemsets for speeding up the mining process. Therefore, the goal of optimizing memory space utilization can be achieved. The correctness guarantee, error analysis, and parameter setting of this approach are presented and a series of experiments is performed to show the effectiveness and the efficiency of this approach.

Responsible editor: M.J. Zaki.

E. T. Wang
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, ROC
e-mail: m9221009@em92.ndhu.edu.tw

A. L. P. Chen (✉)
Department of Computer Science, National Chengchi University, Taipei, Taiwan, ROC
e-mail: alpchen@cs.nccu.edu.tw

**Keywords** Data stream · Data mining · Frequent itemset · Hash-based approach · False positive

## 1 Introduction

Rapid advances in network communications and software/hardware technologies bring huge amounts of data in a form of continuous *data streams*. A data stream is an unbounded sequence of data persistently generated at a high speed. For instance, network traffic, web logs, data from sensor networks, and financial transactions are some categories of data streams. As in traditional databases, techniques of data mining can be applied to discover the underlying knowledge behind data streams. However, data streams are naturally accompanied with several characteristics and restrictions, increasing the degree of complexity of the mining.

First, it is impossible to store an entire data stream in memory even on disk because data streams are unboundedly generated. Therefore, each element in a data stream should be inspected at most once to establish a *synopsis* or *sketch* for the mining. A synopsis or sketch is a structure used to summarize the data stream. Second, the utilization of memory for the mining process such as the space for the synopsis or sketch should be restricted in a reasonable amount. Third, since the past data cannot be saved in memory, approximate results of the mining algorithms with accuracy guarantees are necessary. Finally, since a data stream is continuously generated, both the processing time of maintaining a synopsis or sketch for the data stream and the processing time of the mining algorithm should be as transient as possible. In order to satisfy the above requirements for mining over data streams, the traditional mining algorithms for large databases cannot be directly applied.

In the past few years, the problems of mining frequent items or frequent itemsets over data streams are actively studied. The original problem of mining frequent itemsets attempts to discover associations among items within transactions stored in a database (Agrawal and Srikant 1994). For instance, an itemset may be "milk, bread, support = 10%," meaning that 10% of the transactions in the database contain both milk and bread. As displayed in the above example, *support* of an itemset is a measure showing how much probability a transaction contains the itemset. Users can give a *minimum support threshold* to exclude the itemsets with low probabilities. If the support of an itemset equals or exceeds the minimum support threshold, it is called a *frequent itemset*. As discussed in Lin et al. (2005), according to the degree of importance of the recent data, three models of mining frequent item(set)s over data streams are discussed, including the*sliding window model*, the*time-fading model*, and the*landmark model*.

### 1.1 Related work

The approaches in the sliding window model (Cheng et al. 2006; Chi et al. 2004; Golab et al. 2003; Jiang and Gruenwald 2006; Lin et al. 2005; Li et al. 2006; Leung and Khan 2006; Lee and Ting 2006; Mozafari et al. 2008; Wang et al. 2007b) emphasize the importance of the data contained in a sliding window. The approaches in

Golab et al. (2003), Lee and Ting (2006) and Wang et al. (2007b) focus on finding frequent items, while those in Cheng et al. (2006), Lin et al. (2005), Leung and Khan (2006) and Mozafari et al. (2008) focus on finding frequent itemsets, and those in Chi et al. (2004), Jiang and Gruenwald (2006) and Li et al. (2006) focus on finding closed frequent itemsets. According to the characteristic of a sliding window, two categories, the *count-based* and the *time-based* (Golab et al. 2003) windows, are discussed. In the count-based window, the number of transactions in a window is fixed despite the generating speeds of the transactions. On the other hand, in the time-based window, the number of time units in a window is fixed, making the number of the transactions generated in the window distinct. For both categories of windows, when a transaction is out of the window due to window sliding, its contributions to the support counts of the associated item(set)s are eliminated.

The approaches in the time-fading model (Chang and Lee 2003; Giannella et al. 2004; Lee and Lee 2005) emphasize the sensitivity of time, that is, the recent data are weighted higher than the earlier data. A *decay* mechanism is introduced in Chang and Lee (2003) and Lee and Lee (2005) to achieve the goal of the time-fading model. Users can define how important the recent data are by giving a decay rate with a range from 0 to 1. When a transaction arrives, the new size of the data stream equals the original size multiplied by the decay rate plus one as contributed by the current transaction. In addition, the support counts of an itemset are computed similarly as computing the size of the data stream. In addition to the decay mechanism, the *tilted-time window* is also used to satisfy the requirements of the time-fading model. In the concept of the tilted-time window, the current transaction is held by the current window with the smallest time scale and the former transactions are held by the windows with larger time scales. *FP-stream* (Giannella et al. 2004), which is a variant of *FP-tree* (Han et al. 2000), is an algorithm to find frequent itemsets under a tilted-time window for answering time-sensitive queries.

In the landmark model, users obtain frequent item(set)s from transactions between the *landmark*, which is a particular time point designating the start of the system, and the current time. The approaches based on the landmark model are roughly categorized into *false negative* oriented (Yu et al. 2004) and *false positive* oriented types (Charikar et al. 2002; Cormode and Muthukrishnan 2003; Demaine et al. 2002; Jin and Agrawal 2005; Jin et al. 2003; Karp et al. 2003; Li et al 2004; Manku and Motwani 2002; Wang et al. 2007a). The false negative oriented approaches aim to yield mining results with *no false alarms*, that is, all mining results are truly frequent item(set)s while some truly frequent item(set)s may not be discovered. Yu et al. (2004) propose an *$\varepsilon$-Decoupling* approach based on the concept of *Chernoff bound* for mining frequent itemsets. The number of *false negative frequent itemsets* can be controlled by a given parameter $\delta$ with a range from 0 to 1 and therefore, the *recall rate* (the percentage of truly frequent itemsets returned) of the approach can be guaranteed.

Alternatively, the false positive oriented approaches aim to find all truly frequent item(set)s. As a result, some non-frequent item(set)s may also be returned. Two approaches extended from the *majority algorithm* (Fischer and Salzberg 1982) for mining frequent items over data streams are proposed in Demaine et al. (2002) and Karp et al. (2003). In these approaches, a certain number of counters are used to monitor the items. When an item arrives, the counter monitoring it is increased by one. If the item is not monitored by any counters and some available counters exist, the item is monitored by one of the available counters. Otherwise, all counters are decreased by one and the items with their counts equaling zero are no longer monitored. Jin and Agrawal (2005) develop an *in-core* algorithm, also extended from the majority algorithm, to mine frequent itemsets. The principles of the majority algorithm are used to maintain the itemsets with a length of two, and the Apriori property (Agrawal and Srikant 1994) is employed in finding the itemsets with longer lengths. However, a buffer with an unlimited size is used to store transactions for generating the frequent itemsets with longer lengths, which is not suitable for *online-mining* over data streams.

Dang et al. propose the EStream algorithm operating in the *online-processing mode* to find the frequent itemsets over data streams (Dang et al. 2008). In contrast to the online-processing mode, the *batch-processing mode* adopted by such approaches as (Manku and Motwani 2002; Yu et al. 2004) processes transactions in batches. Although EStream is the first online-processing algorithm which provides an error guarantee for the support counts of frequent itemsets, it is required setting the length of the longest itemsets in advance. Without any background knowledge to the mining results, it is difficult to set a proper length for the longest itemsets to find *all* the frequent itemsets. Moreover, the EStream algorithm is neither a false negative oriented approach nor a false positive oriented approach. That is, the set of the mining results provided by EStream may contain the itemsets not truly frequent and may not contain all the truly frequent itemsets.

The hash-based approaches for mining frequent items over data streams are discussed in Charikar et al. (2002), Cormode and Muthukrishnan (2003), Jin et al. (2003) and Wang et al. (2007a). In these approaches, several hash functions are used to hash the items into their corresponding counters in the hash table, making an item in the data stream associated with a set of counters. Notice that in these approaches, several items may be associated with a common counter. The *maintaining procedure* of the hash table in these hash-based approaches is that, when an item arrives in the data stream, its associated counters are all increased by one. Consequently, the support counts of the items can be estimated by using the values saved in their associated counters. However, the current hash-based approaches only provide solutions for mining frequent items. In the *hCount* method proposed in Jin et al. (2003), when users want to find frequent items, that is, *at the mining stage*, *all items* are hashed to obtain their support counts, and then the received support counts are checked to see whether they are no less than the minimum support threshold multiplied by the current size of the stream. If we regard an itemset as an item, and apply the hCount method to mine frequent itemsets, the processing time in the mining stage is enormous since there is a huge number of candidate itemsets to be checked to see whether they are frequent.

Manku and Motwani (2002) develop the *Lossy Counting* algorithm operating in the batch-processing mode for mining frequent itemsets over data streams. The principle of Lossy Counting is to promptly prune the itemsets with *low frequencies* and to keep only the itemsets with *high frequencies*. However, it needs to retain all itemsets with their supports larger than $\varepsilon$ to guarantee that the estimate support counts of an itemset is less than its true support counts by at most $\varepsilon \times N$, where $\varepsilon$ is a predefined error parameter and $N$ is the current length of the stream. Since the error parameter $\varepsilon$ is often set much smaller than the minimum support threshold, the number of non-frequent itemsets with supports between $\varepsilon$ and the minimum support threshold is massive, causing the memory space needed to be huge. Based on the principle of the Lossy Counting algorithm, Li et al. design an algorithm named DSM-FI (Li et al 2004) for mining frequent itemsets over data streams. DSM-FI reduces the memory utilization by keeping transactions and the sub-transactions projected from the transactions in *prefix tries*. However, DSN-FI needs to enumerate the itemsets from the prefix tries to check whether they are frequent, causing the processing time in the mining stage to be huge. In addition, different from Lossy Counting, it only prunes the *items* with their frequencies smaller than $\varepsilon$ from the prefix tries, limiting the power of the pruning strategy. This is because almost all of the items have high frequencies in most of the cases.

Calders et al. propose a new measure named *Max-frequency* to define the frequencies of item(set)s over data streams (Calders et al. 2006, 2007). The Max-frequency measure different from the mentioned three models is described as follows. The current time to any point in the past of the stream can be regarded as a window. An itemset in each window has its corresponding frequency. The current frequency of an itemset is defined as its maximal frequency over all possible windows in the stream. This measure can avoid *missing seasonal behavior* which occurs as a result of the improper window size set in the sliding window model.

In this paper, we study the problem of mining frequent itemsets over data streams in the landmark model. Most of the approaches in this model operate in the batch-processing mode. Different from that, our approach adopts the online-processing mode discussed in Dang et al. (2008) and solves the problem of needing to set the maximal length of the frequent itemsets to be found in advance as in Dang et al. (2008) for finding all the frequent itemsets. In addition, different from most of the approaches which are *deterministic*, our approach is *probabilistic*. This means, our approach guarantees that the estimate support counts of an itemset possess an error no more than $\varepsilon \times N$ with a confidence given by users.

## 1.2 Main contributions

We propose a novel hash-based approach which operates in the online-processing mode in this paper. This hash-based approach combines the principles of hCount (Jin et al. 2003) and Lossy Counting (Manku and Motwani 2002). By continuously hashing all subsets contained in the current transaction, an entire data stream can be compressed into a new synopsis consisting of a hash table and frequent itemsets. The mining results can be yielded by processing the synopsis. In order to avoid the drawback of Lossy Counting, *a hash table with a fixed size* is used in the synopsis to store the information of the support counts of all itemsets. According to the information retained in the synopsis, the support counts of the non-frequent itemsets can be estimated by a novel estimating technique, avoiding the need to keep the non-frequent itemsets in memory. On the other hand, for extending the hCount method to mine frequent itemsets, the frequent itemsets are kept in the synopsis to quickly output the mining results. The merit of our approach lies on the fact that the disadvantages of hCount and Lossy Counting can be complementary to their advantages.

The contributions of this work are summarized as follows. First, a new synopsis is designed by skillfully integrating the principles of Lossy Counting and hCount to solve the problem of retaining the non-frequent itemsets, which wastes the memory space. Second, based on the newly proposed synopsis, a novel technique for estimating the support counts of the non-frequent itemsets is developed, leading to a high precision of the mining results. Third, the *Chernoff bound* is applied to decide the size of the hash table used in the new synopsis, making the boundary of the size of the hash table tight. Finally, a series of experiments is performed and the space-efficiency of our approach revealed.

### 1.3 Roadmap

The remainder of this paper is organized as follows. Section 2 introduces the basic concepts of this approach. The principal algorithms and its correctness guarantee are described in Sect. 3. In Sect. 4, the parameter setting and accuracy guarantee analysis of this approach are discussed. The experiment results are presented and analyzed in Sect. 5 and finally, Sect. 6 concludes this work.

## 2 Preliminaries

The basic concepts of this hash-based approach are introduced in this section, including the problem definition, the introductions to hCount and Lossy Counting, and the concepts of the hash function used in this approach.

### 2.1 Problem definition

The problem we study in this paper is specified as follows. Let $I = \{i_1, i_2, \ldots, i_M\}$ be a set of literals called *items*. A *data stream* $D = \{t_1, t_2, t_3, \ldots\}$, is an unbounded sequence of *transactions*, where each transaction $t_i$ is a set of items such that $t_i \subseteq I$. In addition, an $itemset\, X$ is also a set of items such that $X \subseteq I$. We say that a transaction $t$ contains an itemset $X$, if $X \subseteq t$.

**Definition 1** (*landmark*) The *landmark* of the data stream is a particular time point at which the system starts. That is, from the landmark, the system starts to generate transactions which form a data stream.

**Definition 2** (*true support counts of an itemset*) The *true support counts* of an itemset $X$ equal the number of transactions containing $X$ from the landmark to the current time.

Let $\sigma$ be a given parameter called *minimum support threshold* such that $\sigma \in (0, 1]$, and $N$ be the current length (size) of the data stream if the newly arrived transaction is $t_N$.

**Definition 3** (*truly frequent itemset*) An itemset is defined as a truly frequent itemset in a data stream $D$ with respect to $N$ if and only if its true support counts are no less than $\sigma \times N$.

**Definition 4** (*immediate subset*) Let $e$ be an itemset with a length denoted as $|e|$. The *immediate subset* of $e$ is a subset of $e$ with a length $|e| - 1$.

For example, *{1, 2, 3}* is an immediate subset of *{1, 2, 3, 4}*.

Given a data stream $D$ with a length of $N$ and three user-defined parameters including a minimum support threshold $\sigma$, an error parameter $\varepsilon$, and a level of confidence (probability) $\rho$ such that $\varepsilon, \rho \in (0, 1)$, the goal of this paper is to design an algorithm operating in the online-processing mode for finding all the frequent itemsets from $D$. In addition, the returned mining results must satisfy the following constraints: (1) all of the truly frequent itemsets of $D$ with respect to $N$ must be returned and (2) the true support counts of an itemset are less than its estimate support counts provided by the algorithm by at most $\varepsilon \times N$ with a confidence level of $\rho$.

## 2.2 Basic concepts of hCount and lossy counting

Our hash-based approach is designed by combining the principles of hCount (Jin et al. 2003) and Lossy Counting (Manku and Motwani 2002). These two approaches are separately introduced in Subsects. 2.2.1 and 2.2.2.

### *2.2.1 Introduction to hCount*

Jin et al. propose a hash-based approach named hCount (Jin et al. 2003) for finding frequent items over data streams. A hash table $S[r][h]$ is used in hCount, having $h$ hash functions $H_i(x)$, $1 \le i \le h$, $H_i(x) \in [0, r)$, where $x$ is an item and $r$ is a positive integer. Each entry in the hash table is a counter. Accordingly, $x$ is associated with $h$ counters $\{S[H_1(x)][1], S[H_2(x)][2], \ldots, S[H_h(x)][h]\}$ in the hash table. Notice that how to set $h$ and $r$ is detailed in Jin et al. (2003). Initially, all counters in the hash table are all zeros. When an item arrives, its associated counters are increased by one. If users make a request for finding frequent items, the associated counters of each item are retrieved, and the minimal value in these counters is viewed as its support counts to check whether it is frequent. Maintaining the hash table as a sketch for a data stream in hCount is very efficient but it may spend much time on the mining stage.

As mentioned in Sect. 1, the hCount method can be easily extended to mine frequent itemsets over data streams, if each itemset is handled as a unique item. Moreover, the Apriori property can also assist in pruning some candidate itemsets in the mining stage to reduce the mining time. We extend the hCount method accordingly as shown in Figs. 1 and 2. However, according to our implementation of the extended hCount method, it takes hours to find the frequent itemsets. This is because the extended hCount method takes a lot of time to check which itemsets are frequent. Obviously, the extended hCount method is not suitable for mining frequent itemsets over data streams.

### *2.2.2 Introduction to lossy counting*

Manku and Motwani propose the Lossy Counting algorithm for finding frequent items and itemsets over data streams in Manku and Motwani (2002). For finding frequent

```
Maintenance of the extended hCount
Input: a data stream $D$ with respect to $N$
Output: a hash table of the extended hCount, $HT$
1: $\forall$ new transaction $t \in D$
2:     $N = N + 1$
3:     $\forall$ itemset $e \in t$
4:         Hash $e$ into a series of counters positioned in $HT$ by using a series of
           hash functions
5:         All the associated counters are increased by one
```

**Fig. 1** Maintenance of the extended hCount method

```
Finding frequent itemsets by extending hCount
Input: the hash table $HT$, the minimum support threshold $\sigma$, and the current
length of the data stream $N$
Output: frequent itemsets
1: $\forall$ single item $i$
2:     Hash $i$ to find its associated counters and choose the minimal value from
       the counters.
3:     If (the minimal value $\geq \sigma \times N$)
4:         Position $i$ into the large-1 set
5: Use items in the large-1 set to generate the candidate-2 set
6: $\forall$ itemset $e \in$ candidate-k set
7:     If (all immediate subsets of $e$ are frequent)
8:         Choose the minimal counts from the immediate subsets
9:         Hash $e$ to find its associated counters and choose the minimal value
           from the counters
10:        If (Min (minimal counts, minimal value) $\geq \sigma \times N$)
11:            Position $e$ into the large-k set
12:            The support counts of $e$ = Min (minimal counts, minimal value)
13: Use itemsets belonging to the large-k set to generate the candidate-(k+1) set.
    The generating method is the same as how it does in the Apriori algorithm
14: Return $\bigcup_{\forall k}$ large-k set
```

**Fig. 2** Finding frequent itemsets by the extended hCount method

items, the stream is conceptually divided into *buckets* with a size of $\lceil 1/\varepsilon \rceil$. The buckets are labeled with an ID number starting with 1, and the current bucket ID is $b_{\text{current}} = \lceil \varepsilon N \rceil$, where $N$ is the current length of the data stream. The synopsis of Lossy Counting for finding frequent items consists of entries with a form of $(x, f, \Delta)$, where $x$ is an item, $f$ is the estimate support counts of $x$, and $\Delta$ is the maximum possible error in $f$. When an item $x$ arrives, we check whether $x$ is stored in the synopsis and increase its estimate support counts by one if it is. Otherwise, a new entry $(x, 1, b_{\text{current}} - 1)$ is created in the synopsis. In addition, an entry $(x, f, \Delta)$ will be deleted if $f + \Delta \leq b_{\text{current}}$ whenever $N \equiv 0 \bmod \lceil 1/\varepsilon \rceil$. When users request the frequent items, those items with their estimate support counts $f \geq (\sigma - \varepsilon) \times N$ are returned.

For finding frequent itemsets, the Lossy Counting algorithm operates in the batch-processing mode, which is similar to that for finding frequent items. Let $\beta$ be the number of buckets in the current batch. For each entry $(e, f, \Delta)$ kept in the synopsis of Lossy Counting for finding frequent itemsets, where $e$ is an itemset, we update $f$

by adding the number of occurrences of $e$ in the current batch. In addition, if $f$ plus $\Delta$ of the updated entry is no more than $b_{\text{current}}$, it will be deleted. If an itemset $e$ is not kept in the synopsis and has the estimate support counts $f \geq \beta$ in the current batch, a new entry $(e, f, b_{\text{current}} - \beta)$ will be kept. Obviously, each itemset with its true support counts exceeding $\varepsilon \times N$ must be kept in the synopsis of Lossy Counting, causing the memory required to be huge.

### *2.2.3 Drawbacks of hCount and lossy counting*

As discussed above, if the hCount method (Jin et al. 2003) is directly applied to mine frequent itemsets over data streams, it will take a long time due to enormous amounts of candidate itemsets to be checked to see if they are frequent. Therefore, if the frequent itemsets are retained and updated during the maintaining process of the synopsis, only the stored itemsets need to be checked at the mining stage, which greatly reduces the mining time. On the other hand, since the Lossy Counting algorithm (Manku and Motwani 2002) needs to keep all the itemsets with their true support counts exceeding $\varepsilon \times N$ to provide a deterministic guarantee: the estimate support counts of an itemset are less than the true support counts by at most $\varepsilon \times N$. However, the non-frequent itemsets with their true support counts between $\varepsilon \times N$ and $\sigma \times N$ kept in the system will substantially degrade the memory utilization. Therefore, if the non-frequent itemsets kept in the system can be compressed into a structure with a fixed size, it may effectively reduce the required memory space. The hash-based approach proposed in this paper is encouraged by hCount and Lossy Counting, which combines their advantages to overcome their drawbacks.

## 2.3 Basic concepts of the hash function

In this subsection, the hash function used in our approach is introduced. In this approach, a transaction in the data stream is represented as a *binary vector* in which each element represents a unique item. The length of the binary vector representing a transaction is equal to the number of all possible items, $M$, appearing in the data stream. To a binary vector $V$ representing a transaction $t$, an element of $V$, $V_i$, equals 1 if item $i$ is purchased in the transaction $t$, and 0 otherwise. The binary vector $V$ can be regarded as a unique number $\sum_{i=1}^{M} 2^{i-1} \times V_i$, expressed by the binary system. Moreover, itemsets contained in transactions are also represented as the binary vectors. Each itemset can also be identified by a number expressed by the binary system.

A hash function, $\text{H}(x) = (a \times x + b) \bmod m$, is used in this hash-based approach, where $a$, $b$ are arbitrary integers and $m$ is the number of entries in the hash table, determined according to a user's confidence level and the error parameter, to be detailed in Sect. 4. When a new transaction arrives, all subsets of the transaction will be regarded as unique numbers and hashed into the entries of the hash table. That is, let $e$ be one of the subsets of the transaction. $e$ is represented as a unique number $e_{bv}$ calculated as discussed above. $\text{H}(e_{bv})$ can be regarded as the identifying number of the entry in the hash table, into which $e$ is hashed.

## 3 The mining approach: hMiner

The hash-based approach for online-mining frequent itemsets over data streams, named *hMiner,* is introduced in this section. By continuously hashing all subsets of a newly arrived transaction into certain entries, entire information of a data stream can be compressed into a newly designed synopsis. The information stored in the synopsis can be exploited to estimate the support counts for itemsets to identify the frequent ones. If an itemset is recognized as frequent, it will be saved in the synopsis, and deleted otherwise. What kind of information is saved in the synopsis and how it assists in identifying frequent itemsets are detailed in the Subsects. 3.1 and 3.2. In addition, the remainder of this section contains the algorithm for mining frequent itemsets and the correctness guarantee of this approach.

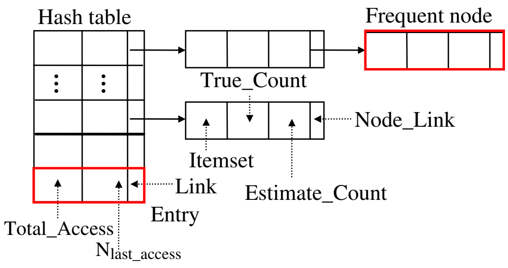
### 3.1 Data structure of hSynopsis

The data structure used in the hMiner approach, named *hSynopsis,* is shown in Fig. 3. It is composed of two components, a *hash table* and *frequent nodes*. The hash table is employed in summarizing the whole data stream while the frequent nodes are used to keep the information of the frequent itemsets.

The hash table has $m$ *entries*, and each contains three fields including *Total_Access*, $N_{\text{last\_access}}$ and *Link*. These three fields are explained as follows. (1) The Total_Access field is *an accumulated counter* maintaining *the number of accesses* to this corresponding entry in the data stream. For example, as an itemset in the current transaction is hashed into the entry $b$, the Total_Access field of $b$ will be increased by one. (2) The $N_{\text{last\_access}}$ field keeps the length of the data stream, that is, the number of the transactions in the stream*, at the time when this corresponding entry was last accessed.* For example, suppose that the current length of the data stream is $N$. As an itemset in the current transaction is hashed into the entry $b$, the value kept in the $N_{\text{last\_access}}$ field of $b$ will be substituted by $N$. This is because when the last access to $b$ in the data stream occurred the length of the stream equals $N$. Finally, (3) the Link field of an entry links to a list of frequent nodes.

A frequent node (f-node) consists of four fields including *Itemset*, *True_Count*, *Estimate_Count*, and *Node_Link*. These four fields are explained as follows. (1) The Itemset field in an f-node indicates the itemset this f-node identifies. Since we only keep the information of frequent itemsets, we need to check whether a non-frequent

**Fig. 3** The hSynopsis data structure



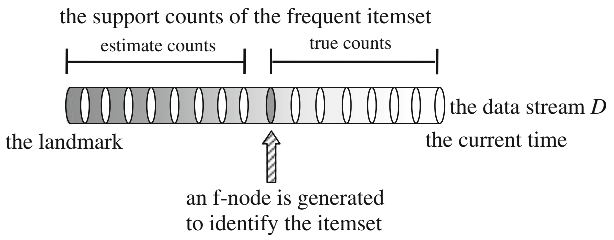

the support counts of the frequent itemset
estimate counts
true counts
the data stream *D*
the landmark
the current time
an f-node is generated to identify the itemset

**Fig. 4** The concepts of true counts and estimate counts

itemset will become a frequent itemset to generate an f-node to identify it. Since the number of the occurrences of the non-frequent itemset is not recorded, we have to estimate its number of the *previous occurrences* to decide whether it should be identified by an f-node. As shown in Fig. 4, we can see that the support counts of a frequent itemset identified by an f-node can be separated into *true counts* and *estimate counts*. The demarcation of the two parts is at the time when the f-node is generated to identify the itemset. (2) The True_Count field in an f-node is an accumulated counter maintaining the true counts of the support counts of the itemset identified by the f-node. As an f-node is newly generated, the default value of its True_Count is 1. The True_Count field of an f-node will be increased by one if the current transaction contains the itemset identified by the f-node. (3) In opposition to the True_Count field, the Estimate_Count field keeps the estimate counts of the support counts of the itemset identified by this f-node. Finally, (4) the Node_Link field links to the next f-node in the entry.

Overall, the information kept in an entry and its list of f-nodes help to estimate how many itemsets are hashed into this entry and how many support counts these itemsets have. From this estimation, whether an itemset is frequent can be determined. The functions and designs of the fields introduced above are integrated into the maintenance of hSynopsis, to be detailed in the following section.

In order to conveniently demonstrate and explain the algorithms of this approach, several notations are introduced as follows. Let $b$ be an identifier of an entry. Then, $b$. Total_Access denotes the value of the Total_Access field of $b$, $b.N_{\text{last\_access}}$ denotes the value of the $N_{\text{last\_access}}$ field of $b$, and $b$.Link denotes the list of f-nodes of $b$. Let *fn* be an identifier of an f-node. Then, *fn*.Itemset denotes the itemset *fn* identifies, *fn*.True_Count denotes the value of the True_Count field of *fn*, and *fn*.Estimate_Count denotes the value of the Estimate_Count field of *fn*.

## 3.2 Maintenance of hSynopsis

Figure 5 shows a flowchart of the maintenance of hSynopsis. The number of the transactions in the data stream from the landmark to the current time is denoted as $N$ which can be viewed as an accumulated counter. As can be seen, all subsets of a newly arrived transaction will be hashed to update the corresponding entries in the hash table. The kernel principle of hMiner is that, when an itemset $e$ is hashed into an entry $b$ in the hash table, the information kept in $b$ will help to determine whether $e$ is frequent. If $e$ is determined to be frequent, an f-node identifying $e$ will be generated and kept in

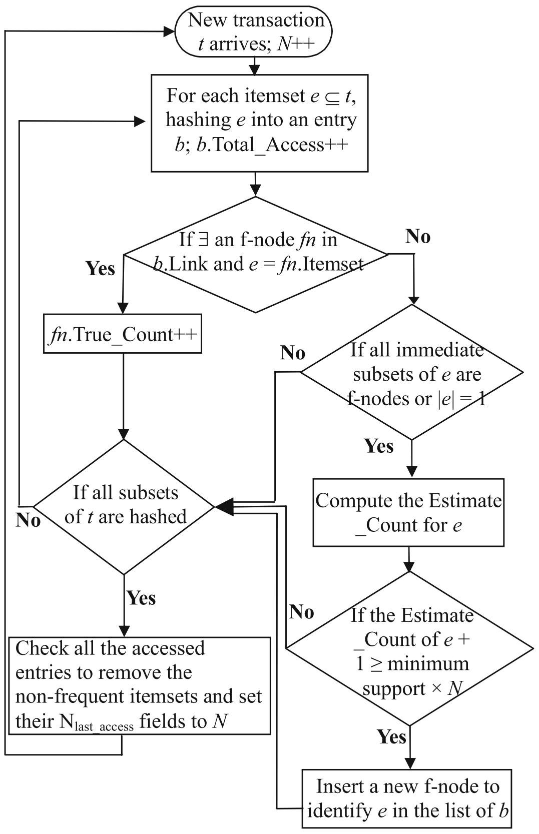

New transaction $t$ arrives; $N$++
For each itemset $e \subseteq t$, hashing $e$ into an entry $b$; $b$.Total_Access++
If ∃ an f-node $fn$ in $b$.Link and $e = fn$.Itemset
Yes
No
$fn$.True_Count++
If all immediate subsets of $e$ are f-nodes or $|e| = 1$
No
Yes
If all subsets of $t$ are hashed
No
Yes
Compute the Estimate _Count for $e$
If the Estimate _Count of $e$ + 1 ≥ minimum support × $N$
No
Yes
Check all the accessed entries to remove the non-frequent itemsets and set their $N_{last\_access}$ fields to $N$
Insert a new f-node to identify $e$ in the list of $b$

**Fig. 5** Flowchart of hSynopsis maintenance

the list of $b$.Link. On the contrary, if $e$ is determined to be non-frequent, no f-node identifying $e$ will be kept in $b$.Link.

The maintenance of hSynopsis can be decomposed into two phases: identifying the frequent itemsets and removing the non-frequent itemsets. A complete example is shown in Subsect. 3.2.3 following the description of the maintenance of hSynopsis in Subsects. 3.2.1 and 3.2.2.

### *3.2.1 Phase I: identify the frequent itemsets*

When a new transaction $t$ arrives, the size of the data stream, $N$, is increased by one. Moreover, all of the itemsets contained in $t$ are enumerated, sorted into an increasing order of length, and then sequentially processed (hashed).

Before explaining the operations on the maintenance of hSynopsis, we first introduce a new term named *working space*. The working space $W$ keeps the whole information of the entries which are accessed in the current transaction but before the entries in the hSynopsis are updated in the current transaction. In addition, $W$ will release all the entries retained in it after the operations to handle the current transaction complete. For example, suppose that the itemsets contained in the current transaction are hashed into the entries $a$, $b$, and $c$. Then, the replicas of the entries $a$, $b$, and $c$ in hSynopsis will be kept in $W$ before the information of the entries in hSynopsis is updated in the current transaction. Since the Estimate_Count field records the estimate for the pervious occurrences of an itemset, the information of the current transaction should not be taken into account. $W$ is therefore used to compute the value to be kept in the Estimate_Count field. Moreover, $W$ can also be used to recognize which entries in the hash table are accessed for the current transaction.

After hashing an itemset $e$ in the current transaction into an entry $b$, whether $b$ is retained in the working space $W$ is checked. If the accessed entry $b$ is not retained in $W$, a replica of $b$, named $b_W$, will be positioned in $W$. After confirming that $b_W$ is in $W$, $b$. Total_Access in the hash table is increased by one. If $e$ can be discovered in an f-node *fn* in the list of $b$.Link, *fn*. True_Count is increased by one. Otherwise, the estimate for the previous occurrences of $e$ (Estimate_Count for$e$) needs to be computed in one of the following two conditions: (1) all immediate subsets of $e$ are identified by the f-nodes linked by their corresponding entries in hSynopsis, and (2) $|e|$ equals one. According to the Apriori property (Agrawal and Srikant 1994), an itemset $e$ with a length $|e|$ becomes a candidate itemset if all of the immediate subsets of $e$ are frequent itemsets. Therefore, if an itemset is not kept in the list of its corresponding entry, this property is used to determine whether this itemset can be a candidate itemset. If it is a candidate itemset, the Estimate_Count for it needs to be computed. On the other hand, to an itemset $e$ with a length of one, that is, $|e| = 1$, it has no immediate subsets. As a result, if the itemset is not kept by an f-node, we need to compute Estimate_Count for it. The detailed method to compute Estimate_Count for an itemset is shown in Fig. 6 and explained in the following.

```
**Computing Estimate_Count for an itemset**
**Input:** an entry $b$, the minimum support threshold $\sigma$, and the working space $W$
**Output:** the Estimate_Count for the itemset $e$
**Variable:** $c_{lb}$, $c_{ub}$, and $n_{lb}$ where $c_{lb} \le c \le c_{ub}$, $n_{lb} \le n$
1: Find $b_W$ from $W$
2: $c_{ub} = b_W.\text{Total\_Access} - \sum_{fn \in b_W.Link} fn.\text{True\_Count}$
3: $c_{lb} = Max(0, b_W.\text{Total\_Access} - \sum_{fn \in b_W.Link} (fn.\text{True\_Count} + fn.\text{Estimate\_Count}))$
4: If $(\lceil s \times b_W.\text{N}_{\text{last\_access}} \rceil - 1 = 0)$
5:     Estimate_Count = 0
6: Else
7:     $n_{lb} = \lceil c_{lb} / (\lceil s \times b_W.\text{N}_{\text{last\_access}} \rceil - 1) \rceil$
8:     If $(n_{lb} = 0)$
9:         Estimate_Count = $c_{ub}$
10:    Else
11:        Estimate_Count = $c_{ub} - (n_{lb} - 1)$
13:    While (Estimate_Count $\ge \sigma \times b_W.\text{N}_{\text{last\_access}}$)
14:        Estimate_Count = Estimate_Count − 1
```

**Fig. 6** Computing Estimate_Count

*Computing the estimate counts* (*Estimate_Count*). Because the Estimate_Count field in an f-node is used to record the estimate of the previous occurrences of an itemset, and the information of $b$ in hSynopsis is updated in the current transaction, the information of $b_W$ in the working space $W$ should be used to compute Estimate_Count for $e$. Two variables $n$ and $c$ are introduced to assist in computing Estimate_Count for an itemset $e$. The variable $n$ is used to identify *the number of different itemsets* hashed into $b_W$ but not in the list of $b_W$.Link, that is, the non-frequent itemsets (at the time when the length of the stream equals $b_W.N_{\text{lsat\_access}}$). On the other hand, the variable $c$ is used to denote *the sum of the previous true support counts of all non-frequent itemsets hashed into* $b_W$. $c$ is used to divide among the non-frequent itemsets hashed into $b_W$ to estimate a reasonable number of the previous occurrences for a non-frequent itemset.

Since the real values of $n$ and $c$ cannot be determined according to the information retained in an entry (in $W$), we develop a mechanism to assist in discovering boundaries of $c$ and $n$. Let $c_{lb}$, $c_{ub}$ and $n_{lb}$ be the lower bound of $c$, the upper bound of $c$ and the lower bound of $n$, respectively, that is, $c_{lb} \le c \le c_{ub}$ and $n_{lb} \le n$. Assume that all itemsets identified by the f-nodes in $b_W$.Link was not contained in any transactions in the stream before their related f-nodes were generated. In other words, their numbers of the previous occurrences are all zeros. Therefore, $c_{ub}$ can be obtained by

$$c_{ub} = b_W.\text{Total\_Access} - \sum_{fn \in b_W.Link} fn.\text{True\_Count} \tag{1}$$

On the other hand, if the Estimate_Count fields in all f-nodes in the list of $b_W$.Link are estimated accurately, that is, the estimate counts are all equal to the previous true support counts, $c_{lb}$ can be obtained by

$$\begin{aligned} c_{lb} = \text{Max}(0,\ & b_W.\text{Total\_Access} \\ & - \sum_{fn \in b_W.Link} (fn.\text{True\_Count} + fn.\text{Estimate\_Count})) \end{aligned} \tag{2}$$

Moreover, if each itemset hashed into $b_W$ but not in the list of $b_W$.Link has *the maximum allowable counts*, i.e. the maximal value less than $\sigma \times b_W.N_{\text{last\_access}}$, the lower bound of $n$ can be derived from

$$n_{lb} = \left\lceil c_{lb} / \left( \left\lceil \sigma \times b_W.N_{\text{last\_access}} \right\rceil - 1 \right) \right\rceil \tag{3}$$

Since those itemsets was not kept in $b_W$.Link at the time of the last access of $b_W$, that is, $b_W.N_{\text{last\_access}}$, the reasonable counts for them must be less than the minimum support threshold $\sigma$ multiplied by $b_W.N_{\text{last\_access}}$, making the maximum allowable counts to equal $\lceil \sigma \times b_W.N_{\text{last\_access}} \rceil - 1$.

After obtaining $c_{ub}$ and $n_{lb}$, Estimate_Count for $e$ can be derived from *distributing the sum of true support counts of all non-frequent itemsets hashed into b, that is, c, among the number of the different non-frequent itemsets hashed into b, that is, n.* In order to avoid missing any truly frequent itemsets, we adopt an *over-distribution strategy*. The over-distribution strategy has two strong assumptions: (1) $e$ must be one of the contributions to $n_{lb}$ and (2) to the other $n_{lb} - 1$ itemsets, each of them only contributes one count to $c_{ub}$. In essence, by giving all the available counts to $e$, we can achieve the goal of over-distribution. Therefore, Estimate_Count for $e$ is equal to $c_{ub} - (n_{lb} - 1)$. For example, if $c_{ub}$ is 5 and $n_{lb}$ is 3, the distribution is that the estimate of the support counts for $e$ is 3 and the other two itemsets respectively have a count of one. Although the over-distribution strategy attempts to make the estimate counts to equal or exceed the previous true support counts of an itemset, Estimate_Count for $e$ should never exceed or equal $\sigma \times b_W.N_{\text{last\_access}}$. This is because $e$ was non-frequent at the time of the last access of $b_W$, that is , $b_W.N_{\text{last\_access}}$. Therefore, Estimate_Count for $e$ should be continuously decreased until it is smaller than $\sigma \times b_W.N_{\text{last\_access}}$, if the result of the over-distribution is no less than $\sigma \times b_W.N_{\text{last\_access}}$. Notice that more than one itemset in a transaction may be hashed into a common entry in the hash table. For example, itemsets *{1, 4}* and *{2, 3}* in the transaction *{1, 2, 3, 4}* are both hashed into the entry $b$. If both of them need to compute Estimate_Count, according to the over-distribution strategy, itemsets *{1, 4}* and *{2, 3}* will have *the same* Estimate_Count.

If $|e|$ equals one, the value calculated as above is the *final Estimate_Count* for $e$. However, to $e$ with a length more than one, the computed value is only a *candidate Estimate_Count*. The Apriori property mentioned before is again considered. The sum of the value kept in the True_Count field and the value kept in the Estimate_Count field in the corresponding f-node should be calculated for all immediate subsets of $e$ by using the information in hSynopsis rather than $W$. The *minimal sum* minus one, which is contributed by the current transaction, will be used to compare with the candidate Estimate_Count. The smaller one is selected to be the *final Estimate_Count* for $e$. According to the over-distribution strategy, the candidate Estimate_Count must exceed or equal the previous true support counts of $e$, and according to the Apriori property, the previous support counts of the immediate subsets of $e$ must also exceed or equal the previous true support counts of $e$. Therefore, the smaller one is selected to be the final Estimate_Count, making the estimate counts for $e$ more accurate. If the final Estimate_Count for $e$ plus one, which is contributed by the current transaction, exceeds or equals $\sigma \times N$, a new f-node (itemset $= e$, True_Count $= 1$, Estimate_Count $=$ the final Estimate_Count) is inserted into the list of $b$.Link.

### *3.2.2 Phase II: remove the non-frequent itemsets*

Since the current size of the data stream $N$ will be increased as time goes by, some frequent itemsets may become non-frequent. Therefore, the f-nodes must be checked to decide whether they should be kept in the lists of the entries. However, if all entries in hSynopsis are checked, the processing time of the phase II for handing a transaction will be enormous. Since some entries have been accessed in the current transaction, these entries can be easily checked.

After hashing all subsets of the current transaction into hSynopsis, the entries accessed in hSynopsis should be inspected to remove the f-nodes with the value kept in the True_Count field plus the value kept in the Estimate_Count field smaller than $\sigma \times N$ from their lists. Moreover, the $N_{\text{last\_access}}$ fields of the corresponding entries in hSynopsis should also be substituted by $N$. These operations in the phase II not only

```
Maintenance of hSynopsis
Input: a data stream $D$, a working space $W$, and the minimum support threshold $\sigma$
Output: an updated hSynopsis $H$
Initially: $W = \phi$
1:  $\forall$ new transaction $t \in D$
2:      $N = N + 1$
3:      Phase I: $\forall$ itemset $e \in t$
4:          Hash $e$ into a certain entry $b$ positioned in $H$.
5:          If (the corresponding entry of $b$, $b_W$, is not found in $W$)
6:              Store a replica of $b$, namely $b_W$, in $W$
7:          $b$.Total_Access = $b$.Total_Access + 1
8:          If ($\exists$ an f-node $fn$ in $b$.Link and $e = fn$.Itemset)
9:              $fn$.True_Count = $fn$.True_Count + 1
10:         Else
11:             If (all immediate subsets of $e$ are f-nodes in $H$ or $|e| = 1$)
12:                 Use $b_W$ in $W$ to compute the candidate Estimate_Count
13:                 If ($|e| > 1$)
14:                         Use the information in $H$ to compute True_Count +
                            Estimate_Count for all immediate subsets of $e$ and chose
                            the minimal sum − 1 to be the minCount
15:                         Estimate_Count = $Min$ (candidate Estimate_Count,
                            minCount)
16:                 Else
17:                     Estimate_Count = candidate Estimate_Count
18:           If (Estimate_Count + 1 $\geq \sigma \times N$)
19:                 Insert an f-node (Itemset = $e$, True_Count = 1, Estimate_Count)
                    into $b$.Link.
20:     Phase II: $\forall$ entry $b_W \in W$
21:         Check the corresponding of $b_W$, $b$, in $H$.
22:         $\forall$ f-node $fn \in b$.Link
23:             If ($fn$.Estimate_Count + $fn$.True_Count < $\sigma \times N$)
24:                 Delete $fn$ from $b$.Link.
25:             $b$.$N_{last\_access} = N$
26:         Delete $b_W$ from $W$
```

**Fig. 7** Maintenance of hSynopsis

remove the non-frequent itemsets but also assist in deciding Estimate_Count for an itemset in the phase I, since the $N_{\text{last\_access}}$ field updated in the phase II can help to compute the estimate counts of the non-frequent itemsets.

As can be seen, after the phase II terminates, the value kept in the True_Count field plus the value kept in the Estimate_Count field of all f-nodes in the list of each entry in hSynopsis exceeds or equals the value in the $N_{\text{last\_access}}$ field of the entry multiplied by the minimum support threshold. It means that *the f-nodes kept in the lists of the entries are related to the* $N_{\text{last\_access}}$ *fields of their corresponding entries*. This responds to the procedure of computing Estimate_Count for an itemset. In essence, the phase I and the phase II in the maintenance process of hSynopsis are rigidly related. The detailed algorithm of the maintenance of hSynopsis is shown in Fig. 7.

### *3.2.3 An example for hSynopsis maintenance*

An example of the maintenance process of hSynopsis is shown in Fig. 8a–e. Suppose that the minimum support threshold is equal to 0.4, the itemsets including *{1}*, *{2}*, *{5}*, and *{1, 5}* are all hashed into the same entry *b* and all entries of hSynopsis are

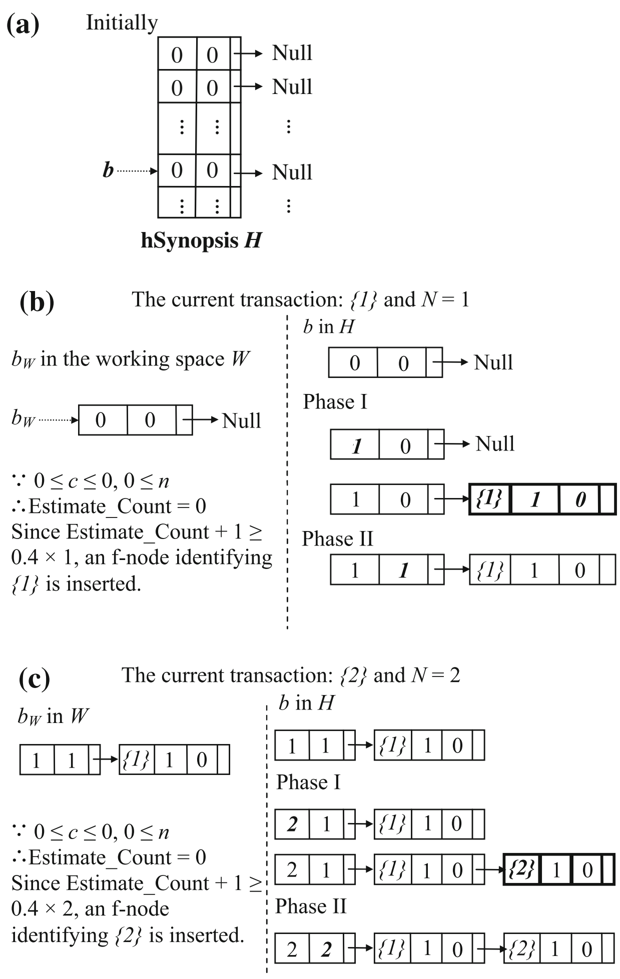

(a)
Initially
Null
b
hSynopsis H
(b)
The current transaction: {1} and N = 1
b in H
b_W in the working space W
Null
Phase I
{1}
Phase II
∵ 0 ≤ c ≤ 0, 0 ≤ n
∴Estimate_Count = 0
Since Estimate_Count + 1 ≥ 0.4 × 1, an f-node identifying {1} is inserted.
(c)
The current transaction: {2} and N = 2
b_W in W
b in H
{2}
∵ 0 ≤ c ≤ 0, 0 ≤ n
∴Estimate_Count = 0
Since Estimate_Count + 1 ≥ 0.4 × 2, an f-node identifying {2} is inserted.

**Fig. 8** **a** An example of the maintenance of the hSynopsis. **b** The current transaction is *{1}*. **c** The current transaction is *{2}* . **d** The current transaction is *{1, 5}*. **e** The current transaction is *{2}*

initially set to zero or null as shown in Fig. 8a. Four transactions, explicitly, *{1}*, *{2}*, *{1, 5}*, and *{2}* are sequentially generated in the data stream. Figure 8b–e show the status of entry $b$ in hSynopsis and the status of entry $b_W$ in the working space $W$. As shown in Fig. 8d, several itemsets contained in the current transaction may be hashed into the same entry. This condition does not affect the correctness of this approach since the previous information of an entry is retained in $W$ and exploited to compute Estimate_Count for an itemset.

## 3.3 Mining frequent itemsets

The algorithm of the mining process is shown in Fig. 9. By scanning the hash table in hSynopsis and inspecting all f-nodes listed in each entry in the hash table, the frequent

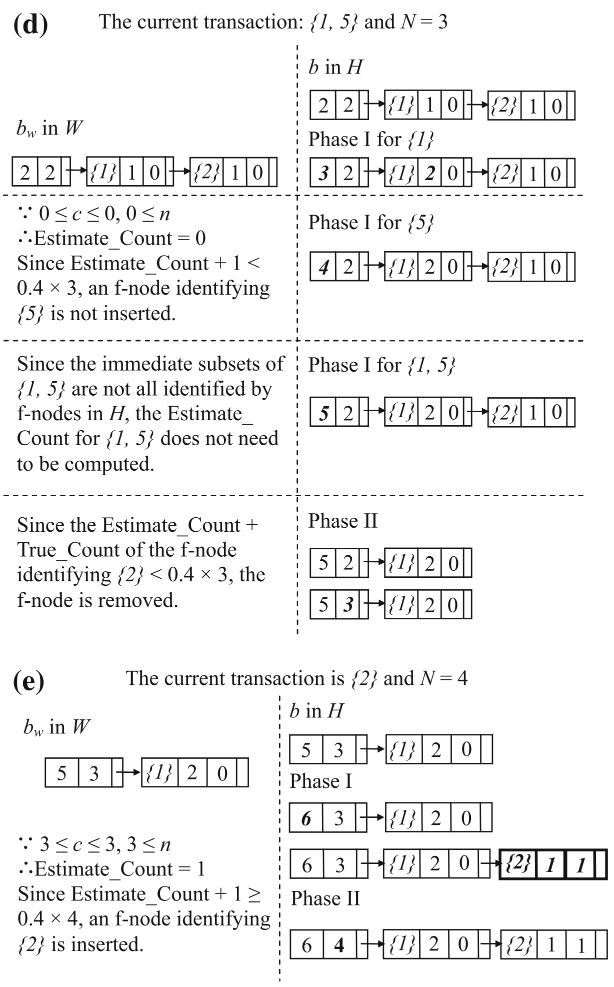

(d) The current transaction: {1, 5} and N = 3
b in H
2 2 {1} 1 0 {2} 1 0
b_w in W
2 2 {1} 1 0 {2} 1 0
Phase I for {1}
3 2 {1} 2 0 {2} 1 0
∵ 0 ≤ c ≤ 0, 0 ≤ n
∴Estimate_Count = 0
Since Estimate_Count + 1 < 0.4 × 3, an f-node identifying {5} is not inserted.
Phase I for {5}
4 2 {1} 2 0 {2} 1 0
Since the immediate subsets of {1, 5} are not all identified by f-nodes in H, the Estimate_ Count for {1, 5} does not need to be computed.
Phase I for {1, 5}
5 2 {1} 2 0 {2} 1 0
Since the Estimate_Count + True_Count of the f-node identifying {2} < 0.4 × 3, the f-node is removed.
Phase II
5 2 {1} 2 0
5 3 {1} 2 0
(e) The current transaction is {2} and N = 4
b in H
b_w in W
5 3 {1} 2 0
5 3 {1} 2 0
Phase I
6 3 {1} 2 0
∵ 3 ≤ c ≤ 3, 3 ≤ n
∴Estimate_Count = 1
Since Estimate_Count + 1 ≥ 0.4 × 4, an f-node identifying {2} is inserted.
6 3 {1} 2 0 {2} 1 1
Phase II
6 4 {1} 2 0 {2} 1 1

**Fig. 8** continued

```
Mining frequent itemsets
Input: hSynopsis H, σ, and N
Output: the mining results
1: ∀ entry b in hSynopsis
2:     If (b.N_last_access ≥ σ × N)
3:         ∀ f-node fn ∈ b.Link
4:             If (fn.True_Count + fn. Estimate_Count ≥ σ × N)
5:                 Return fn.itemset
```

**Fig. 9** Mining frequent itemsets

itemsets can be returned to users on demand. Before trying to inspect f-nodes listed in an entry, checking the $N_{\text{last\_access}}$ field in the entry can help to determine whether the list of the entry should be inspected or not. Obviously, if the value kept in the

$N_{\text{last\_access}}$ field of an entry is smaller than $\sigma \times N$, the f-node list of the entry does not need to be checked. This is because the support counts of an itemset in the list of the entry are at most equal to the value kept in the $N_{\text{last\_access}}$ field which is smaller than $\sigma \times N$. To an f-node, if the value of its True_Count field plus the value of its Estimate_Count field exceeds or equals $\sigma \times N$, the itemset it identifies will be returned to users. In the set of the mining results, all truly frequent itemsets will be contained but there may be some *false alarms*.

## 3.4 Correctness guarantee

The correctness of hMiner is discussed in this subsection. By extending the following lemma, we will show that the set of the mining results provided by hMiner has *no false dismissals*, that is, all truly frequent itemsets in the data stream with respect to the length of $N$ are returned to users.

**Lemma 1** *Suppose that the minimum support threshold is* $\sigma$*, the current length of the data stream is N, and an entry b in hSynopsis is accessed for handling the current transaction. After the operations on handling the current transaction terminate, the itemsets kept in all f-nodes in b.Link must form a set of frequent itemsets hashed into b, with no false dismissals.*

*Proof* When the operations on handling the current transaction terminate, the value of the Estimate_Count field plus the value of the True_Count field $\geq \sigma \times N$ must be held for each f-node in $b$.Link. Therefore, if the true support counts of an itemset can be proven to be smaller than or equal to Estimate_Count + True_Count (the value of the Estimate_Count field plus the value of the True_Count field) of the corresponding f-node, the itemsets kept in all f-nodes in $b$.Link forming a set of frequent itemsets hashed into $b$, with no false dismissals can be guaranteed.

Suppose that there have been $k$ transactions to access $b$ and the current transaction is the $k$th one. The above condition can be proven by *induction on k*. *Basis*: Initially, $b$.Total_Access, $b.N_{\text{last\_access}}$, and $b$.Link are respectively set to 0, 0, and null. When $k = 1, 0 \leq c \leq 0$ and $0 \leq n$, which implies that Estimate_Count = 0. Indeed, since the transaction is the first transaction to access $b$ in the data stream and there are no previous transactions to access $b$, the previous true support counts of an itemset must be 0. The induction basis is true.

*Induction step*: The induction hypothesis is to assume that when $k = K$, the above condition is held, that is, the true support counts of the itemsets identified by all the corresponding f-nodes in $b$.Link are smaller than or equal to Estimate_Count + True_Count of the corresponding f-nodes. Let the value in the $N_{\text{last\_access}}$ field of $b$ be $N_K$, while $k = K$. According to the maintenance process of hSynopsis, Estimate_Count + True_Count of all f-nodes in $b$.Link must exceed or equal $\sigma \times N_K$. When $k = K + 1$, let the size of the data stream be $N_{K+1}$. Two conditions are separately considered. One condition is that *an itemset contained in the transaction and hashed into b can be found to be identified by an f-node in b.Link.* The value kept in the True_Count field of the corresponding f-node is increased by one. Since its True_Count + Estimate_Count $\geq$ the true support counts of the itemset identified by it when $k = K$, its True_Count + 1 + Estimate_Count must also $\geq$ the true support counts of the itemset identified by it when $k = K + 1$. Although the f-node may be removed as its True_Count + 1 + Estimate_Count < $\sigma \times N_{K+1}$, it still holds the correlation between the true support counts and the estimate support counts.

Another condition is that *an itemset contained in the transaction and hashed into b cannot be found in b.Link.* As previously discussed, the estimate of the pervious true support counts (Estimate_Count) for the itemset is computed. The computation of Estimate_Count can guarantee that the computed value of Estimate_Count for the itemset is no less than its previous true support counts. Therefore, if an f-node identifying the itemset is inserted in $b$.Link in the current transaction, its Estimate_Count + 1 is no less than $\sigma \times N_{K+1}$ and its true support counts equaling the previous true support counts + 1 will be smaller than or equal to Estimate_Count + 1.

Consequently, by the principle of induction, the true support counts of the itemset in the stream is smaller than or equal to Estimate_Count + True_Count of the corresponding f-node in $b$, making the set of the itemsets identified by all f-nodes in $b$.Link to have no false dismissals. □

**Corollary 1** *Since all entries in hSynopsis can form their own sets of frequent itemsets with no false dismissals, the itemsets in the union of their own sets of frequent itemsets, with Estimate_Count + True_Count no less than the minimum support threshold multiplied by the current length of the data stream must form the set of mining results with no false dismissals.*

## 4 Accuracy guarantee analyses

The accuracy guarantee of hMiner is discussed in this section. We concentrate on analyzing the error provided by the hash table of hSynopsis in Sect. 4.1 and discuss the accuracy guarantee of the whole approach in Sect. 4.2.

### 4.1 Parameter setting and error analysis of the hash table

The number of entries in the hash table, $m$, is decided according to *a user's confidence level* and *an error parameter*, which directly affects the error provided by hMiner.

**Lemma 2** *Suppose that $\varepsilon$ is the error parameter, $\rho$ is the level of confidence, N is the current length of a data stream, and m is the number of entries in the hash table. $\varepsilon$ and $\rho$ are both given by users such that $\varepsilon, \rho \in (0, 1)$. The estimate support counts of each itemset provided by hMiner possess an error no more than $\varepsilon \times N$ with a confidence level at least $\rho$, if m is set to $e/\varepsilon^2 M \times (e^L - 1) \times ln((1 - 2^M)/ln\ \rho)$, where L is the average length of the transactions in the data stream and M is the number of different items appearing in the data stream.*

*Proof* If the f-node part is excluded in hSynopsis, that is, only a hash table is contained in hSynopsis, the hMiner approach is converted into a special case of hCount (Jin et al. 2003) with only a hash function. In the following discussion, the part of f-nodes is excluded to simplify the error analysis.

Let $X$ be a random variable to denote the length of a transaction in the data stream and let $X_1, X_2, \ldots, X_M$ be a sequence of Poisson trials with $\Pr(X_i = 1) = p_i, \forall i = 1$ to $M$, where a random variable $X_i$ denotes the appearing of item $i_i$ in a transaction. $X_i = 1$ means $i_i$ is purchased and 0, otherwise. Obviously,

$$X = \sum_{i=1}^{M} X_i \tag{4}$$

As a result, $E[X] = E\left[\sum_{i=1}^{M} X_i\right] = \sum_{i=1}^{M} E[X_i] = \sum_{i=1}^{M} p_i$. Since the average length of transactions is $L$, that is $E[X] = L$, we can then obtain the following equation

$$L = \sum_{i=1}^{M} p_i \tag{5}$$

As we know, a transaction with a length of $k$ may generate $2^k - 1$ accesses to the hash table. We can then expect that $E[2^X - 1]$ hash accesses may occur in a transaction.

$$\begin{aligned} E[2^X - 1] &= E[2^X] - 1 \\ &= E\left[2^{\sum_{i=1}^{M} X_i}\right] - 1 \\ &= E\left[2^{X_1} \times 2^{X_2} \times \cdots \times 2^{X_M}\right] - 1 \\ &= E[2^{X_1}]E[2^{X_2}] \cdots E[2^{X_M}] - 1 \end{aligned} \tag{6}$$

By definition of expectation of a random variable, $E[2^{X_i}]$ can be computed.

$$E[2^{X_i}] = p_i \times 2 + (1 - p_i) = 1 + p_i \le e^{p_i} \tag{7}$$

Thus, from Eqs. 6 and 7,

$$\begin{aligned} E[2^X - 1] &= E[2^{X_1}]E[2^{X_2}] \cdots E[2^{X_M}] - 1 \\ &\le e^{p_1} e^{p_2} \cdots e^{p_M} - 1 \\ &= e^{\sum_{i=1}^{M} p_i} - 1 \\ &= e^L - 1 \end{aligned} \tag{8}$$

Therefore, there is a total of $(e^L - 1)N$ hash accesses if the current length of the data stream is equal to $N$.

If the hash function is universal, these hash accesses to a certain entry $b$ can be viewed as realizations of $(e^L - 1)N$ i.i.d. random variables $Y_1, Y_2, Y_3, \ldots, Y_{(e^L - 1)N}$. Each of them will follow a Bernoulli distribution, $B(1, 1/m)$, where $1/m$ is attached to $Y_i = 1$ and $(1 - 1/m)$ is attached to $Y_i = 0, \forall i = 1$ to $(e^L - 1)N$. Let $Y$ be a random variable to denote the error for the true support counts of an itemset in hMiner.

As can be seen, several itemsets may be hashed into the same entry, thus making $Y$ nonnegative. Moreover,

$$Y \le \sum_{i=1}^{(e^L-1)N} Y_i \tag{9}$$

Consider the worst case, all the hash accesses to a certain entry do not support a certain itemset, thus causing $Y = \sum_{i=1}^{(e^L-1)N} Y_i$. Then, after the processing of $N$ transactions in the data stream terminates,

$$\mathrm{E}[Y] = \sum_{i=1}^{(e^L-1)N} \mathrm{E}[Y_i] = \sum_{i=1}^{(e^L-1)N} \frac{1}{m} = \frac{(e^L-1)N}{m} \tag{10}$$

Since $Y_1, Y_2, Y_3, \ldots, Y_{(e^L-1)N}$ are Bernoulli trials with $\Pr(Y_i) = 1/m$, the following *Chernoff bound* holds: for any $\delta > 0$,

$$\Pr(Y \ge (1+\delta)\mu) \le \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \tag{11}$$

where $\mu = \mathrm{E}[Y]$. Further infer from Eq. 11 and substitute $\mu$ with $\frac{(e^L-1)N}{m}$, the following formulas can be obtained.

$$\Pr(Y - (1+\delta)\mu < 0) > 1 - \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \tag{12}$$

$$\Pr\left(Y - (1+\delta)\frac{(e^L-1)N}{m} < 0\right) > 1 - \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\frac{(e^L-1)N}{m}} \tag{13}$$

Let the error parameter $\varepsilon$ be $\frac{(1+\delta)(e^L-1)}{m}$. We can obtain that

$$m = \frac{(1+\delta)(e^L-1)}{\varepsilon} \tag{14}$$

Consider the worst case, a transaction in the data stream contains all the items, thus generating $2^M - 1$ itemsets. Let $\rho$ be the probability with which the errors for all itemsets satisfy Eq. 13. That is,

$$\begin{aligned}\rho &= \left(1 - \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\frac{(e^L-1)N}{m}}\right)^{2^M-1} \\ &\approx \exp\left(\left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\frac{(e^L-1)N}{m}} \times -(2^M-1)\right)\end{aligned} \tag{15}$$

Inference of the formula for $\rho$ is as follows.

$$\begin{aligned}
&\ln \rho = \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\frac{(e^L-1)N}{m}} \times -(2^M-1)\\
&\Rightarrow \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{-\left(\frac{(e^L-1)N}{m}\right)} = \frac{-(2^M-1)}{\ln \rho}\\
&\Rightarrow -\left(\frac{(e^L-1)N}{m}\right)\ln\left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right) = \ln\left(\frac{-(2^M-1)}{\ln \rho}\right)\\
&\Rightarrow 1 = \frac{\ln\left(\frac{-(2^M-1)}{\ln \rho}\right)}{-\left(\frac{(e^L-1)N}{m}\right)\ln\left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)} \qquad (16)
\end{aligned}$$

Multiplying the left-hand-sides and the right-hand-sides of Eqs. 14 and 16, respectively, we can obtain that

$$\begin{aligned}
&m \times 1 = \frac{(1+\delta)(e^L-1)}{\varepsilon} \times \frac{\ln\left(\frac{-(2^M-1)}{\ln \rho}\right)}{\left(-\frac{(e^L-1)N}{m}\right)\ln\left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)}\\
&\Rightarrow m = \frac{(1+\delta)(e^L-1)}{\varepsilon} \times \frac{\ln\left(\frac{-(2^M-1)}{\ln \rho}\right)}{\left(-\frac{(e^L-1)N}{\frac{(1+\delta)(e^L-1)}{\varepsilon}}\right)\ln\left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)}\\
&\Rightarrow m = \frac{(e^L-1)\ln\left(\frac{-(2^M-1)}{\ln \rho}\right)}{\varepsilon^2 N} \times \frac{(1+\delta)^2}{-\ln\left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)} \qquad (17)
\end{aligned}$$

Since $N$ is much larger than $M$, by substituting $N$ with $M$, we can obtain that

$$m \le \frac{(e^L-1)\ln\left(\frac{-(2^M-1)}{\ln \rho}\right)}{\varepsilon^2 M} \times \frac{(1+\delta)^2}{\ln\left(\frac{(1+\delta)^{(1+\delta)}}{e^{\delta}}\right)} \qquad (18)$$

In addition, by assuming $\delta \ll 1$, we can obtain the following formula.

$$\begin{aligned}
&\frac{\ln\left(\frac{(1+\delta)^{(1+\delta)}}{e^{\delta}}\right)}{(1+\delta)^2} = \frac{\ln(1+\delta)}{(1+\delta)} - \frac{\delta}{(1+\delta)^2} \approx \frac{\ln(1+\delta)}{(1+\delta)} \approx e^{-1} \qquad (19)\\
&\Rightarrow \frac{(1+\delta)^2}{\ln\left(\frac{(1+\delta)^{(1+\delta)}}{e^{\delta}}\right)} \approx e \qquad (20)
\end{aligned}$$

From Eqs. 18 and 20, we can obtain that

$$m \leq \frac{e}{\varepsilon^2 M} \times (e^L - 1)\ln\left(\frac{-(2^M - 1)}{\ln \rho}\right) \tag{21}$$

Therefore, the estimate support counts of each itemset possess an error no more than $\varepsilon \times N$ with a probability at least $\rho$, if $m$ is set to $e/\varepsilon^2 M \times (e^L - 1) \times \ln((1 - 2^M)/\ln \rho)$. □

### 4.2 Discussion on accuracy guarantee of hMiner

From Lemma 1 and Corollary 1, all truly frequent itemsets must be produced as answers by hMiner. That is because the estimate supports of itemsets provided by hMiner must be no less than their true supports. In addition, from Lemma 2, the estimate support counts of each itemset possess an error no more than $\varepsilon \times N$ with a probability at least $\rho$, if $m$ is properly set. In other words, Lemma 2 shows that the estimate support counts of each itemset possess an error larger than $\varepsilon \times N$ with a probability at most $1 - \rho$. Lemma 1 demonstrates that hMiner is a false positive oriented approach which returns all truly frequent itemsets while Lemma 2 indicates that how much hMiner over-estimates for support counts.

In essence, hSynopsis consists of a hash table and f-nodes. The hash table provides the error guarantee for supports of itemsets while the f-nodes speed up the mining stage. In reality, the capability of the part of f-nodes in hSynopsis not only speeds up the mining stage but also enhances the precision of hMiner.

Suppose that the part of f-nodes is excluded in hSynopsis, that is, hSynopsis becomes a special case of the sketch of hCount, using only one hash function. In addition, the $N_{\text{last\_access}}$ field and the Link field of an entry in hSynopsis can also be removed because their functions relate to f-nodes. In this case, the maintenance of this "reduced" hSynopsis will become very easy. When an itemset is hashed into an entry $b$, $b$. Total_Access is increased by one. In addition, the support counts of an itemset are set to the value of the Total_Access field of which the itemset is hashed into. For example, if an itemset $e$ is hashed into the entry $b$, $b$.Total_Access is regarded as the support counts of $e$. On the other hand, to a "normal" hSynopsis, in the over-distribution strategy for computing Estimate_Count, $c_{ub}$ used to be distributed among the non-frequent itemsets is equal to $b_W$.Total_Access $- \sum_{fn \in b_W.Link} fn$.True_Count. It shows another capability of the part of f-nodes, which is helping to provide a more precise estimate support counts, since the *true counts* for the itemsets in f-nodes are directly kept. Therefore, although the error guarantee of hMiner is provided by the hash table in hSynopsis, the part of f-nodes can further improve the precision of the mining results.

## 5 Performance evaluation

In this section, the experiment results for evaluating hMiner are presented and analyzed.

## 5.1 Performance criteria

According to the characteristics and restrictions of data streams discussed in Sect. 1, several criteria related to the accuracy, execution time, and memory utilization of the mining algorithms are proposed to evaluate the performances of this approach.

**Criterion 1** (*precision*). Since hMiner is one of the false positive oriented approaches, all frequent itemsets must be returned but several non-frequent itemsets may also be outputted. Therefore, *precision* is introduced to be a measure of accuracy of mining algorithms. *Precision* is measured by $|R_{\text{apriori}} \cap R_{\text{method}}|/|R_{\text{method}}|$, where $R_X$ denotes the set of results mined by the method $X$ and $|R_X|$ denotes the number of the frequent itemsets contained in $R_X$.

**Criterion 2** (*recall*). Since hMiner is compared with the other approaches which may not find all the truly frequent itemsets, *recall* is introduced to be another measure of accuracy of mining algorithms. *Recall* is measured by $|R_{\text{apriori}} \cap R_{\text{method}}|/|R_{\text{apriori}}|$, where $R_X$ denotes the set of results mined by the method $X$ and $|R_X|$ denotes the number of the frequent itemsets contained in $R_X$. Obviously, as proven in Lemma 1, the recall of hMiner must be 1.

**Criterion 3** (*memory*). Data are generated promptly in data streams, making memory utilization for mining algorithms to be one of the most critical issues. Therefore, the memory space for a mining algorithm to keep its synopsis or sketch should be measured, which is denoted as *memory*.

**Criterion 4** (*mining-time*). The processing time of mining frequent itemsets from a synopsis or a sketch saved in memory is denoted as *mining-time*.

**Criterion 5** (*maintaining-time*). The time of maintaining a synopsis or a sketch for a mining algorithm is also considered to evaluate its efficiency. Therefore, the average processing time to handle a transaction for maintaining the synopsis or sketch is introduced to be a measure, denoted as *maintaining-time*.

## 5.2 Experiment setup

The in-core algorithm proposed in Jin and Agrawal (2005) is an algorithm designed for mining frequent itemsets over data streams. However, in that approach, the transactions need to be kept in an unlimited buffer for multiple scanning to generate the longer frequent itemsets, causing it unsuitable for the online-mining process in which a transaction is inspected at most once. In addition, the DSM-FI approach proposed in Li et al (2004) keeps the transactions and the sub-transactions projected from the transactions in *prefix tries*, and enumerates the itemsets from the prefix tries to check whether they are frequent, causing the processing time in the mining stage to be huge, which is also unsuitable for the online-mining process. As EStream (Dang et al. 2008) is the first online-processing algorithm which provides an error guarantee for the support counts of the frequent itemsets, and hMiner also operates in the online-processing mode, hMiner is compared with EStream in the experiments. In addition, as Lossy

Counting (Manku and Motwani 2002) is one of the false positive oriented approaches and it is also the most representative algorithm for mining frequent item(set)s in the landmark model of the data streams, comparing the performance of hMiner with Lossy Counting is also performed in the experiments.

*Buffer-Trie-SetGen* (BTS) implementation discussed in Manku and Motwani (2002) is utilized to implement the Lossy Counting algorithm in the experiments. As hMiner and EStream both operate in the online-processing mode, which means that a newly arrived transaction is immediately processed, Lossy Counting is then transformed into operating in the online-processing mode in the experiments. The Buffer in the BTS implementation transfers to keep only one transaction and the TRIE (the synopsis of Lossy Counting) is changed to be saved in memory but not on disk. Since there is only one transaction in the Buffer, it is easy for the SetGen to enumerate all subsets contained in the transaction without needing a complex implementation of Heap. As a result, the Lossy Counting algorithm for finding frequent itemsets operates in the online-processing mode similarly as the Lossy Counting algorithm for finding frequent items. That is, when a newly arrived transaction is received, it is immediately processed and whenever $N \equiv 0 \bmod \lceil 1/\varepsilon \rceil$, an entry $(e, f, \Delta)$ in the synopsis (TRIE) will be deleted if $f + \Delta \leq b_{\text{current}}$, where $e$ is an itemset, $f$ is the estimate support counts of $e$, $\Delta$ is the maximum possible error in $f$, and $b_{\text{current}}$ is the current bucket ID.

All of the algorithms are implemented in C++ and all the experiments are performed on a PC with the Intel Pentium 4 3.2GHz CPU, 3GB of main memory, and under the Window XP operating system. Six datasets are tested in the experiments. One is a real web log dataset named Calgary-HTTP (Arlitt and Williamson 1996) and the others are generated by the IBM synthetic data generator (Agrawal and Srikant 1994). The datasets used in the experiments are described in Table 1 (see Fig. 10). All factors used in the experiments are summarized in Table 2.

According to Lemma 2, the estimate support counts of each itemset provided by hMiner possess an error no more than $\varepsilon \times N$ with a confidence level at least $\rho$, if $m$ is set to $e/\varepsilon^2 M \times (e^L - 1) \times \ln((1 - 2^M)/\ln \rho)$. Indeed, setting $m$ as above can satisfy the accuracy guarantee but in reality, the part of f-nodes is excluded and the worst cases, *i.e. assuming transactions containing all the distinct items and assuming that all accesses to an entry are the error for the support counts of an itemset*, are considered in the proof of Lemma 2. Obviously, Lemma 2 provides a theoretical analysis but if $m$ is set accordingly, it may waste the memory space because it is almost impossible for a transaction to contain all the distinct items, which results in generating all the possible $2^M - 1$ itemsets in the real applications and moreover, the part of f-nodes also helps to reduce the error for support counts. Since most of the users concern more on the number of the truly frequent itemsets found and the number of the found itemsets which are true, providing the error guarantee as mentioned above turns into evaluating precision (Criterion 1) and recall (Criterion 2) in the experiments. Therefore, under some given assumptions, a heuristic method of setting $m$ to $\tau \times e/\varepsilon \times (2^L - 1) \times \ln\left(\binom{M}{L}(1 - 2^L)/\ln \rho\right)$ in the experiments is described as follows, where $\tau$ is a coefficient and can be adjusted using historical data. As can be seen, when $\varepsilon M$ is greater than 1, $e/\varepsilon^2 M \times (e^L - 1) \times \ln((1 - 2^M)/\ln \rho)$ is smaller than $e/\varepsilon \times (e^L - 1) \times \ln((1 - 2^M)/\ln \rho)$. Then, by assuming that the variation of the lengths of transactions is very small, that is, the lengths of all transactions are nearly equal

**Table 1** Test datasets

| Dataset | Description |
|---|---|
| T4.I4.D500K | The default test dataset, which has an average transaction size of 4 with an average maximal itemset size of 4, and the number of distinct items = 20,000 |
| T4.I4.D200K | An average transaction size of 4 with an average maximal itemset size of 4, and the number of distinct items = 10,000 |
| T5.I4.D200K | An average transaction size of 5 with an average maximal itemset size of 4, and the number of distinct items = 20,000 |
| T6.I4.D200K | An average transaction size of 6 with an average maximal itemset size of 4 and the number of distinct items = 25,000 |
| T7.I4.D100K | An average transaction size of 7 with an average maximal itemset size of 4 and the number of distinct items = 25,000 |
| Calgary-HTTP | A real dataset on tracing web logs, with an average transaction size of 1.7. The distribution of this dataset is shown in Fig. 10 |

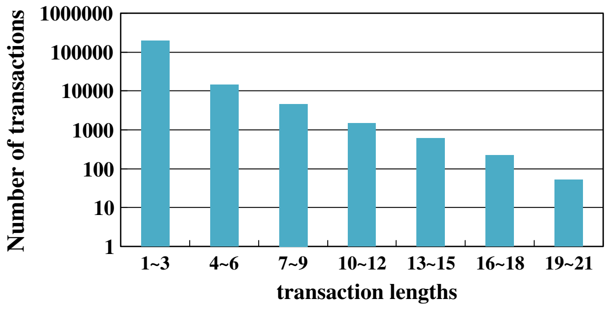


**Fig. 10** The distribution of Calgary-HTTP

to $L$, a transaction may generate $2^L - 1$ accesses and the number of combinations of itemsets $2^M - 1$ is then transformed into $\binom{M}{L}(2^L - 1)$. Therefore, $m$ is set to $\tau \times e/\varepsilon \times (2^L - 1) \times \ln\left(\binom{M}{L}(1 - 2^L)/\ln \rho\right)$ experimentally, which saves more memory space than setting $m$ theoretically. A proper $\tau$ can be found heuristically as follows. Collect the transactions of a data stream for some period of time as its historical data and then apply hMiner to the collected historical data and adjust $\tau$ to obtain a high precision rate for the mining results found from the historical data. The relation between $\tau$ and the precision of the mining results from the historical data of T4.I4.D500K is listed in Table 3. Then, a proper $\tau$ equaling 0.1, which provides an acceptable precision, e.g. >0.9, is used. For the other datasets, we use the same method to adjust $\tau$ and decide the sizes of the hash tables.

**Table 2** Experiment factors

| Factor | Default | Range | |
|---|---|---|---|
| $\rho$ | 0.9 | 0.1–0.9 | $\rho$: The confidence level |
| $\varepsilon$ | 0.0009 | 0.0001–0.0009 | $\varepsilon$ : The error parameter |
| $\sigma$ | 0.02/0.006 | – | $\sigma$: The minimum support threshold |
| $N$ | 500 K | 500–2,000 K | $N$: The size of the data stream |
| $k$ | 5/15 | – | $k$: The longest itemsets to be found in EStream |
| | | | $\rho$, $\varepsilon$, $\sigma$, and $k$ are given by users. |

**Table 3** relation between $\tau$ and precision ($P$) while using T4.I4.D500K

| $\tau$ | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.06 | 0.07 | 0.08 | 0.09 | 0.1 | 0.2 | 0.3 | 0.4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P$ | 0.54 | 0.86 | 0.54 | 0.86 | 0.94 | 0.80 | 0.91 | 0.95 | 0.88 | 0.96 | 0.96 | 0.98 | 0.98 |

## 5.3 Experiment results

The experiment results are presented and analyzed in this subsection.

### *5.3.1 Experiment results on a varying $\rho$*

The first set of experiment results shown in Figs. 11, 12, 13, 14 describes the relation between the performance criteria discussed in Sect. 5.1 and the confidence level $\rho$. As $\rho$ increases, the memory used in hMiner shown in Fig. 13 also increases. This is because the number of entries in the hash table of hSynopsis is positively correlated to $\rho$. However, since the effect of $\rho$ is at the scale of logarithm, the memory used slowly increases. In addition, since the mining-time of hMiner is highly related to the number of entries in the hash table of hSynopsis, which is increased slowly as $\rho$ increases, the curves in Fig. 12 are almost stable. The curves in Fig. 11 are also almost stable due to the same distribution of the test dataset.

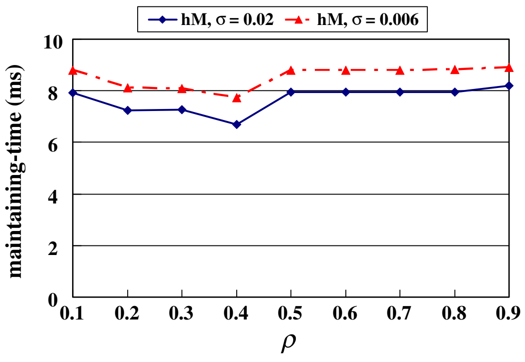


**Fig. 11** Relation between $\rho$ and *maintaining-time*

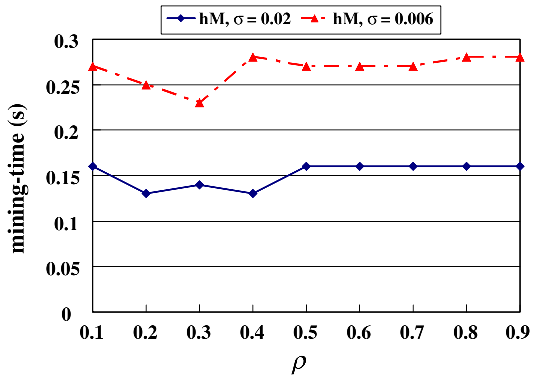


**Fig. 12** Relation between $\rho$ and *mining-time*

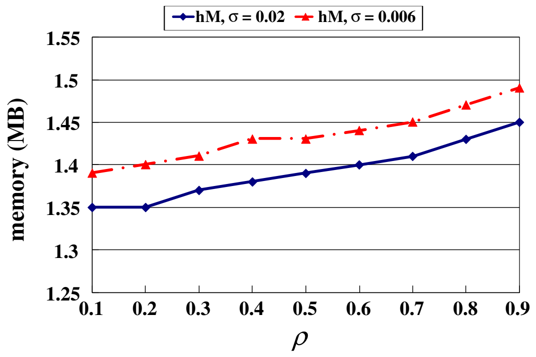


**Fig. 13** Relation between $\rho$ and *memory*

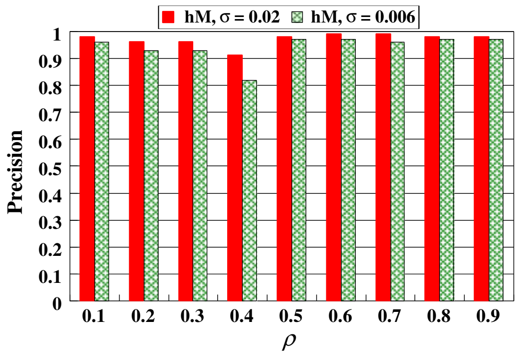


**Fig. 14** Relation between $\rho$ and *precision*

From Figs. 11, 12, 13, 14, we find that the curve of "hM, $\sigma = 0.006$" is always higher than that of "hM, $\sigma = 0.02$." This is because under the condition of low minimum support threshold, the number of f-nodes kept in hSynopsis is increased, directly affecting the maintaining-time, mining-time, and memory used in hMiner. Figure 14 shows the precision of hMiner. It is high (>0.9) in most of the cases. However, it deserves to be mentioned that when $\rho$ is equal to 0.4, its precision falls and its maintaining-time falls too. It may be caused by the chosen hash function. If the hash is not universal and many itemsets fall into a common entry, the precision of hMiner is decreased. On the other hand, as the probability of collisions happening in handling a transaction is increased due to the non-universal hash function, the maintaining-time decreases. This is because whenever a collision occurs during processing a transaction, the computed Estimate_Count can be shared among the itemsets in the transaction hashed into the same entry, leading to a reduction of the maintaining-time.

### 5.3.2 *Experiment results on a varying $\varepsilon$*

The second set of experiment results shown in Figs. 15, 16, 17, 18, 19 describes the relation between the performance criteria and the error parameter $\varepsilon$. As shown in Fig. 15, the maintaining-time of Lossy Counting and that of hMiner increase as $\varepsilon$ decreases. To Lossy Counting, as $\varepsilon$ decreases, more itemsets are kept in its synopsis, increasing the processing time of adding the information of a transaction to the synopsis of Lossy Counting. On the other hand, the size of the hash table in hSynopsis becomes large as $\varepsilon$ decreases, decreasing the probability of collisions happening in handling a transaction. As a result, the maintaining-time of hMiner increases.

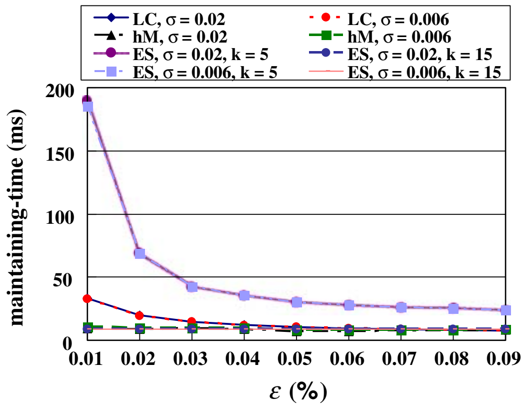


**Fig. 15** Relation between $\varepsilon$ and *maintaining-time*

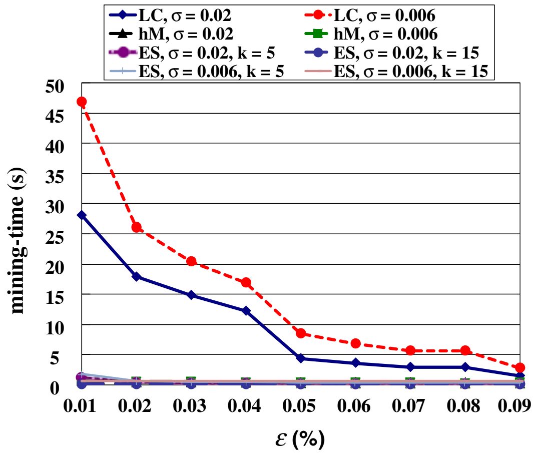


**Fig. 16** Relation between $\varepsilon$ and *mining-time*

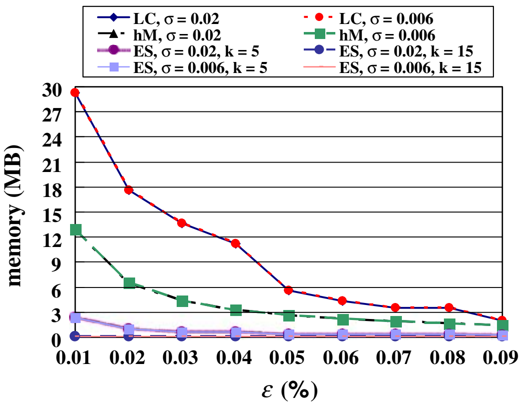


**Fig. 17** Relation between $\varepsilon$ and *memory*

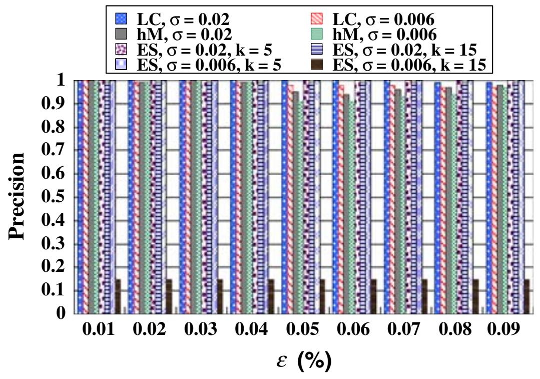


**Fig. 18** Relation between $\varepsilon$ and *precision*

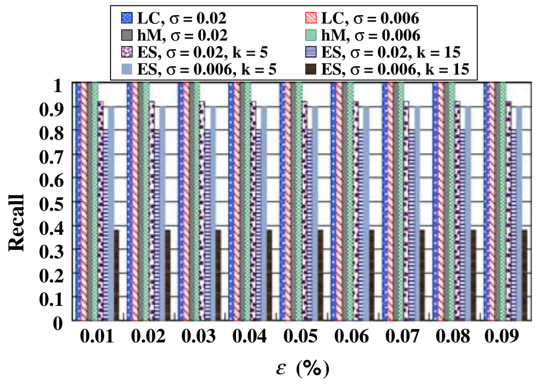


**Fig. 19** Relation between $\varepsilon$ and *recall*

Referring to Fig. 15, the maintaining-time of EStream with $k = 15$ is almost fixed under various values of $\varepsilon$. In reality, whenever $N \equiv 0 \bmod \lceil 2^{k-2}/\varepsilon \rceil$, where $k$ is the parameter in EStream indicating the length of the longest itemset to be found, the itemsets with low support counts are pruned from the synopsis of EStream. The maintaining-time and the memory used in EStream indeed relate to $\varepsilon$. However, in

Fig. 15, since the bucket size of EStream with $k = 15$ is at least *nine millions*, much larger than the size of the test dataset, the effect of $\varepsilon$ on the performance criteria does not appear. While comparing to Lossy Counting, the bucket size of Lossy Counting is at most *ten thousands* in the experiments, which is much smaller than that of EStream with $k = 15$. It implies that since the term of $\lceil 2^{k-2}/\varepsilon \rceil$ exponentially increases as $k$ increases, when $k$ is set to a large value for finding *all* the frequent itemsets, the pruning strategy is almost useless. That is, the pruning is done infrequently. To EStream with $k = 5$, since its bucket size is at most eighty thousands, we can find that its maintaining-time also increase as $\varepsilon$ decreases.

Moreover, from Fig. 15, we find that the maintaining-time of EStream with $k = 15$ is less than those of Lossy Counting and hMiner. Although EStream attempts to process all itemsets contained in a transaction, as an itemset in the transaction is not kept in its synopsis, the supersets of the itemset do not need to be processed. In addition, by setting the boundaries of support counts for itemsets in advance, EStream avoids keeping the itemsets with low support counts in the synopsis, thus reducing the memory used as shown in Fig. 17 and decreasing the maintaining-time and mining-time as respectively shown in Figs. 15 and 16.

In EStream, it sets a set of boundaries of support counts for distinct length of itemsets. For example, suppose that $k = 15$, then we have Table 4 computed as follows. $J$ is an index factor to indicate the lengths of itemsets. Boundary[$J$] is a boundary of support counts corresponding to an itemset with a length of $J$. When $J = 1$, Boundary[$J$] is set to 0. Boundary[$J$]$=2^{k-1}(1 - 1/2^{J-1})$, $\forall J = 2$ to $k$. All itemsets with a length of one (items) are directly kept in the synopsis in EStream. Moreover, an itemset $e$ with a length $|e|$ will be kept in its synopsis, if in the current bucket, (1) the support counts of its immediate subsets just kept in the current bucket exceed Boundary[$|e|$] minus Boundary[$|e| - 1$] ( $\delta_{\text{boundary}}$ *for* $|e|$ *in short*), and (2) the support counts of its immediate subsets kept from the previous bucket exceed Boundary[$|e|$]. From the above equation, we know that the boundaries for the support counts of the itemsets computed in EStream only relate to the parameter $k$ despite the minimum support threshold and the size of the current stream. When $\sigma \times N$ is small, e.g. smaller than $\delta_{\text{boundary}}$ for $|e| =$ two (i.e. $8192 - 0$), many frequent itemsets cannot pass the boundaries, resulting in a low recall as "ES, $\sigma = 0.006$, $k = 15$" shows in Fig. 19. This situation may become worse while the minimum support threshold is set to a smaller value or the parameter $k$ is set to a larger value. However, without any background knowledge of the mining results, if we want to find all the frequent itemsets, $k$ is intuitively set to a large value. Therefore, a low recall becomes the main drawback of EStream in practice. On the other hand, EStream will over-estimate the support counts for those itemsets kept in it synopsis, thus resulting in a low precision as "ES, $\sigma = 0.006$, $k = 15$" shows in Fig. 18.

**Table 4** Boundaries for support counts of itemsets in EStream with $k = 15$

| Length of itemsets | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | … | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Boundary | 0 | 8,192 | 12,288 | 14,336 | 15,360 | 15,872 | 16,128 | 16,256 | … | 16,382 | 16,383 |

**Table 5** Boundaries for support counts of itemsets in EStream with $k = 5$

| Length of itemsets | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Boundary | 0 | 8 | 12 | 14 | 15 |

EStream with $k = 5$ uses less memory than hMiner and still results in high precision and recall. Since the boundaries of EStream with $k = 5$, as shown in Table 5, help to refuse the itemsets with low supports to be kept in its synopsis, the memory used in EStream is small. However, the goal is to find *all* the frequent itemsets rather than finding the frequent itemsets with lengths up to 5. The itemsets with longer lengths such as 6, 7, 8, and 9 in the test dataset cannot be found due to setting $k = 5$. The high recall of EStream with $k = 5$ comes from the fact that the ratio of the number of the frequent itemsets with lengths more than 5 to the number of total frequent itemsets is small. If the ratio is high, the recall of EStream with $k = 5$ may become low. The maintaining-time of EStream with $k = 5$ is higher than those of hMiner and Lossy Counting. This is because for processing an itemset, it needs to check all of the immediate subsets of the itemsets in the synopsis constructed by prefix tries. Although hMiner needs to do this too, checking via hashing is more efficient than checking via traversing tries.

As shown in Figs. 16 and 17, the mining-time and memory used in hMiner and Lossy Counting increase as $\varepsilon$ decreases. The number of the entries in hSynopsis increases as $\varepsilon$ decreases in hMiner, therefore increasing the memory space and the mining-time. To Lossy Counting, more itemsets are kept in the synopsis as $\varepsilon$ decreases, causing the memory used and the mining-time to increase. Referring to Fig. 17, the synopsis of Lossy Counting needs more memory space than that of hMiner because the itemsets with the supports exceeding $\varepsilon$ need to be saved in memory. However, retaining the non-frequent itemsets with supports between $\varepsilon$ and $\sigma$ uses too much memory space. On the other hand, to hMiner, the entire data stream is compressed and saved in hSynopsis consisting of a hash table with a fixed size and the f-nodes retaining the frequent itemsets. This fixed-sized structure used in hMiner limits the memory space for the non-frequent itemsets which are kept in Lossy Counting.

Since the memory used in Lossy Counting is only affected by $\varepsilon$, the *line of* "*LC*, $\sigma = 0.02$" and the *line of* "$LC, \sigma = 0.006$" in Fig. 17 are overlapped. So are those of EStream. On the other hand, $\sigma$ may also affect the memory used in hMiner since the number of f-nodes is decided by $\sigma$. From Fig. 17, we also find that the *line of* "$hM, \sigma = 0.02$" and the *line of* "$hM, \sigma = 0.006$" are almost overlapped. This is because the memory used by the hash table of hSynopsis is much more than that of the f-nodes, making the effect of $\sigma$ to be limited. Although hMiner uses less memory space, as shown in Fig. 18, the precision of hMiner is almost the same as that of Lossy Counting.

### *5.3.3 Experiment results on scalability*

The third set of experiment results shown in Figs. 20, 21, 22, 23, 24 describes the effects of the number of transactions (scalability) on these approaches. Under the same data distribution, the maintaining-time, the mining-time and the memory used in hMiner are almost constant. The maintaining-time of Lossy Counting and that of

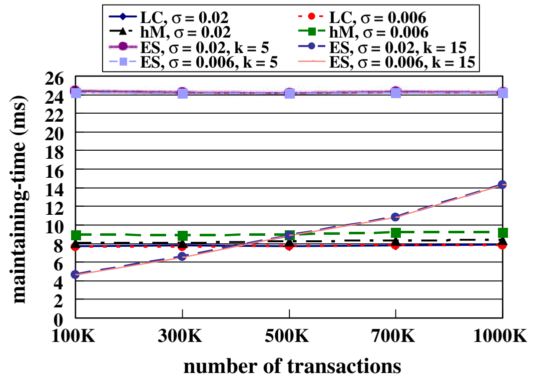


**Fig. 20** Relation between *scalability* and *maintaining-time*

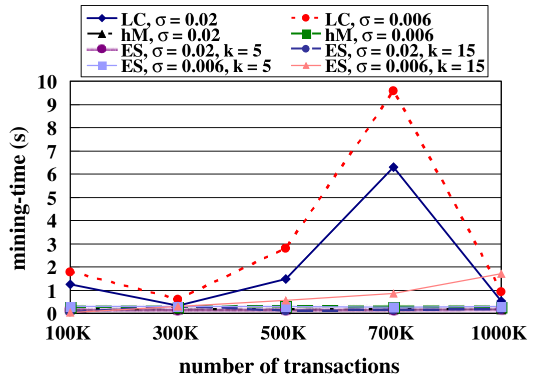


**Fig. 21** Relation between *scalability* and *mining-time*

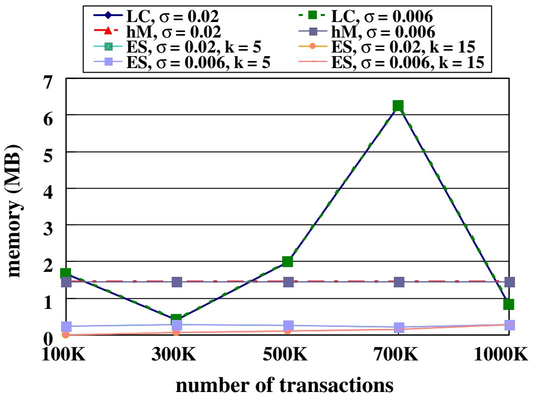


**Fig. 22** Relation between *scalability* and *memory*

EStream with $k = 5$ as shown in Fig. 20 are also almost fixed under different sizes of the data. Different from Lossy Counting which only need to directly add the information of the current transaction to the prefix tries, hMiner needs additional computation on estimation, causing the maintaining-time of hMiner is a little higher than that of Lossy Counting.

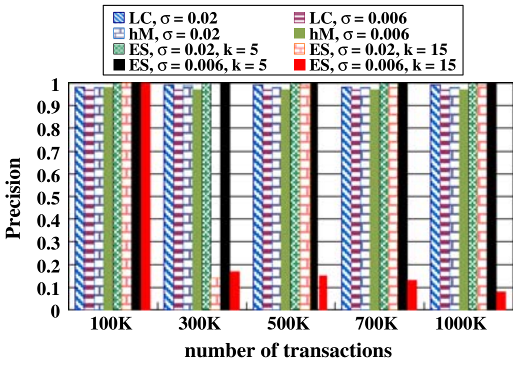


**Fig. 23** Relation between *scalability* and *precision*

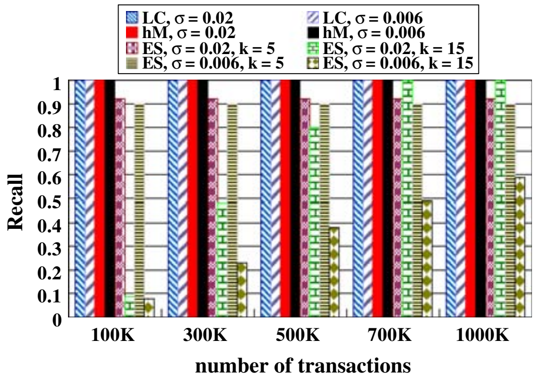


**Fig. 24** Relation between *scalability* and *recall*

From Fig. 20, it is also found that the maintaining-time of EStream with $k = 15$ increases as the size of data increases. As more transactions are generated, the support counts of more itemsets exceed the boundaries mentioned above in EStream with $k = 15$, causing the maintaining-time and memory used to increase. For the same reason, the recall of EStream shown in Fig. 24 gets higher as the number of transactions becomes larger. Since the bucket size of EStream with $k = 15$ is near nine millions, we expect that the maintaining-time and the memory used in EStream with $k = 15$ must still increase as the size of dataset increases until the bucket is full. In addition, since EStream uses the structure of prefix tries to keep the itemsets, checking whether all immediate subsets of an itemset are kept in the synopsis may spend much time. Therefore, although the memory used in EStream with $k = 15$ slowly increases as the size of the dataset increases, the maintaining-time of EStream with $k = 15$ drastically increases.

As shown in Figs. 21 and 22, the curves of the mining-time and the memory used in Lossy Counting change drastically. Obviously, the more itemsets are saved in the synopsis of Lossy Counting, the more mining-time it needs. Theoretically, as the stream size increases, the memory used in Lossy Counting also increases. Whenever $N \equiv 0 \bmod \lceil 1/\varepsilon \rceil$, the itemsets with small support counts are removed from the synopsis of Lossy Counting, making the memory used to decrease. It is the reason why the curve of Lossy Counting in Fig. 22 drastically changes. Since the memory used in Lossy Counting may drastically change, it is difficult to evaluate the maximum required memory, causing an *out of memory* risk to exist. Moreover, the mining-time of hMiner is much shorter than that for Lossy Counting because the current frequent itemsets can be directly determined from the f-nodes kept in the hSynopsis of hMiner.

As shown in Fig. 23, the precision of hMiner and that of Lossy Counting are almost equal to one. On the other hand, the precision of EStream with $k = 15$ under $\sigma = 0.006$ seems to decrease as the stream size increases, and the precision of EStream with $k = 15$ under $\sigma = 0.02$ seems to be low only in the case of 300 K. In addition, as shown in Fig. 24, the recall of hMiner and that of Lossy Counting must equal one due to the design principles. On the other hand, the recall of EStream with $k = 15$ seems to increase as the stream size increases. In reality, all of the results on precision and recall of EStream are strongly related to the boundaries for support counts of itemsets as shown in Table 4. When the size of the stream is small e.g. 100 K, since the support counts of all items are smaller than $\delta_{\text{boundary}}$ for $|e| =$ two (i.e. 8192 − 0), only the itemsets with a length of one are kept in the synopsis of EStream, leading to a high precision but a very low recall. While $\sigma = 0.02$, as the size of stream increases (from 100 K to 300 K), the support counts of some items exceed $\delta_{\text{boundary}}$ for $|e| =$ two and then, the itemsets with longer lengths are kept in the synopsis. At the mining stage, for an itemset $e$ kept in the synopsis, $e$ is reported as frequent if the support counts of $e$ are no less than $\sigma \times N$ minus the boundary for the itemsets with a length $|e|$. It means that EStream attempts to over-estimate the support counts of the itemsets kept in the synopsis, causing the precision to decrease. Therefore, the precision of "ES, $\sigma = 0.02$, $k = 15$" is low in the case of 300 K. Notice that, the boundaries in Table 4 are set in advance, which are computed only according to the value of $k$ rather than $N$ or $\sigma$. As a result, while $\sigma \times N <$ those boundaries, the recall of EStream will be low. The values of $\sigma \times N$ in this experiment are listed in Table 6 for reference.

Setting the parameter $k$ directly affects the memory used, the precision, and the recall, even the maintaining-time of EStream. However, how to set a proper $k$ to find *all* frequent itemsets over data streams is difficult without any knowledge of the mining results, and intuitively setting $k$ to a large value may incur very low recall and precision, making EStream not workable in practice.

### *5.3.4 Experiment results on testing distinct datasets*

The experiment results of using distinct datasets to be the test data are shown in Figs. 25, 26, 27. All the parameters are set as the default values except testing Calgary-HTTP. While testing Calgary-HTTP, the minimum support threshold used is 0.1% and the

**Table 6** Values of the minimum support threshold multiplied by the stream size $N$

| Number of transaction (K): | 100 | 300 | 500 | 700 | 1, 000 |
|---|---|---|---|---|---|
| $\sigma = 0.02$ | 1,635 | 6,673 | 10,000 | 13,341 | 20,000 |
| $\sigma = 0.006$ | 491 | 2,002 | 3,000 | 4,003 | 6,000 |

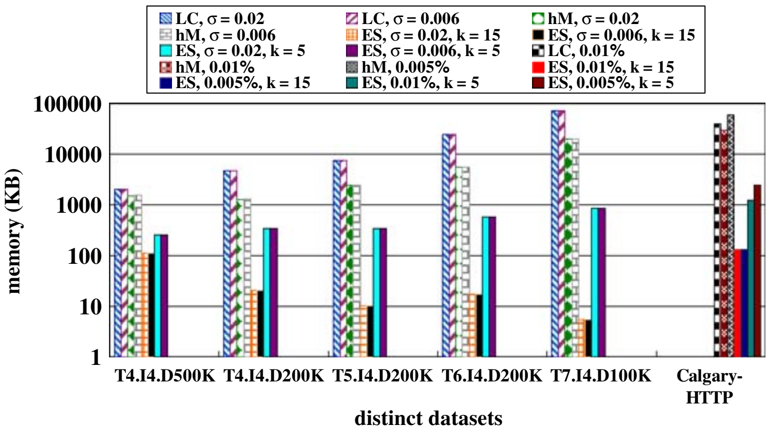


**Fig. 25** Relation between *distinct datasets* and *memory*

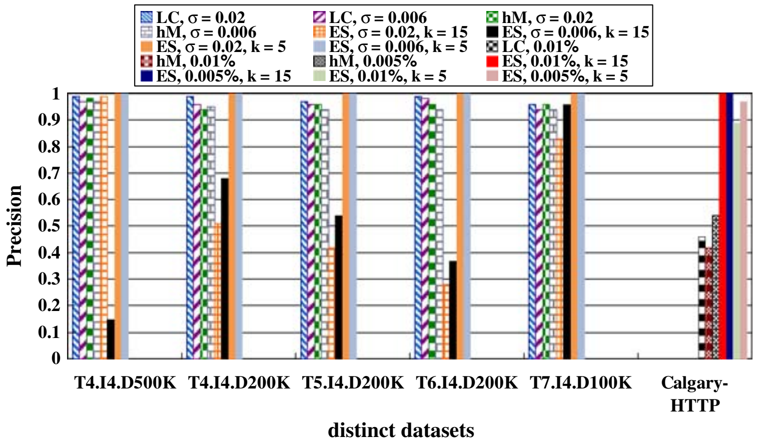


**Fig. 26** Relation between *distinct datasets* and *precision*

error parameters $\varepsilon$ are set to 0.01% and 0.005%. Under these settings, the maximal length of the truly frequent itemsets found is 5. As mentioned above, since the boundaries for support counts of itemsets computed in EStream do not relate to the size of the data stream and the minimum support threshold, the recall of EStream with $k = 15$ shown in Fig. 27 is still very low. Although EStream uses less memory space than hMiner, its precision and recall are not acceptable.

EStream with $k = 5$ has high precision and recall in Figs. 26 and 27. However, since the goal is to find all the frequent itemsets rather than finding the frequent itemsets with lengths up to 5, that is, the goal of EStream, reporting only the frequent itemsets with lengths up to 5 is unacceptable. If the ratio of the number of frequent itemsets with lengths longer than 5 to the total number of frequent itemsets increases, the recall of EStream with $k = 5$ must become low. When the test dataset used is Calgary-HTTP, under the condition of setting the minimum support threshold to 0.1%, the maximal

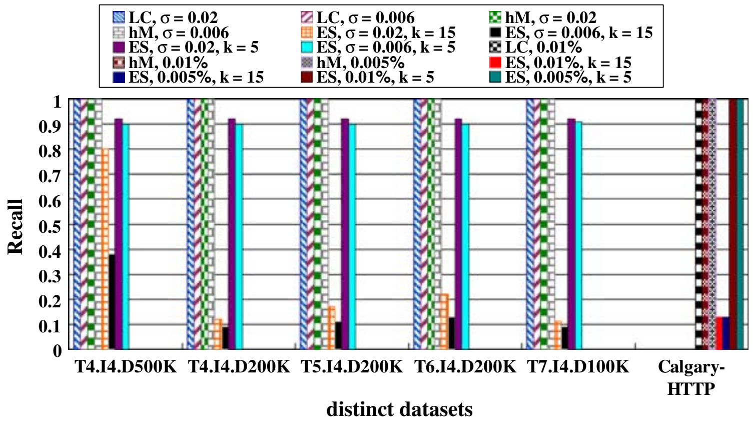


**Fig. 27** Relation between *distinct datasets* and *recall*

length of the truly frequent itemsets found is 5, which exactly matches $k$ set to 5 in EStream, resulting in high precision and recall. It is lucky. However, in fact, EStream is not designed to find all frequent itemsets. How to set a proper $k$ without any background knowledge to the mining results is the most difficult part for applying EStream to find all frequent itemsets because $k$ will directly affect the precision and recall of EStream. A small $k$ limits EStream to find the truly frequent itmesets with lengths longer than $k$ while a large $k$ even results in very low precision and recall.

From Fig. 26, it can be found that the precision of hMiner is almost the same as that of Lossy Counting. Since Lossy Counting is a deterministic algorithm, that is, it guarantees that the errors of the supports of all itemsets are no more than $\varepsilon$ with 100% confidence, its precision is a little higher than that of hMiner. However, the memory used in hMiner is much less than that of Lossy Counting as shown in Fig. 25.

According to the effect of the data distribution of Calgary-HTTP, many itemsets have supports between $\sigma - \varepsilon$ and $\sigma$. Therefore, the precisions of Lossy Counting and hMiner are around 40% under $\varepsilon = 0.01\%$ while the test data used is Calgary-HTTP. However, when $\varepsilon$ is set to 0.005% the precision of hMiner becomes higher. To Lossy Counting, since its required memory space cannot be satisfied due to too many nodes in TRIE under $\varepsilon = 0.005\%$, causing the system to crash. Therefore, no experiment results of Lossy Counting with $\varepsilon = 0.005\%$ are shown for Calgary-HTTP in Figs. 25, 26, 27.

### 5.3.5 *Discussion on the limitations of hMiner*

From the experiment results, the limitations of hMiner are summarized as follows. (1) Different from the other approaches which are only affected by $\varepsilon$, the maintaining-time and memory used in hMiner is also affected by $\sigma$. As $\sigma$ decreases, the number of f-nodes kept may become larger, thus resulting in using more memory space. The processing time on checking whether an itemset is kept in f-nodes and that on computing Estimate_Count also increase, causing the increase of the maintaining-time. In addition, as f-nodes are used to keep frequent itemsets, they may be frequently inserted, deleted, and reinserted if the status of the frequent changes drastically. However, while Lossy Counting works in the online-processing mode, the nodes kept in the synopsis also need to face similar problems. They need to be inserted as transactions arrive, deleted as the conceptual bucket is full, and reinserted as the new transactions arrive. (2) As the lengths of transactions increase, the maintaining-time of hMiner also increases because hMiner needs to process all subsets of transactions. However, the memory used in hMiner is less than that used in Lossy Counting since a few long transactions may incur a huge number of subsets needing to be kept. (3) hMiner compresses the information of a whole data stream into a hash table with a fixed size and additionally keeps f-nodes to indicate the frequent itemsets. In order to speed up the maintaining procedure of hSynopsis, only the f-nodes kept in the entries accessed by the current transaction are checked. Therefore, some *out-of-date* frequent itemsets may be kept in the f-nodes. We can periodically scan the whole hSynopsis, set all the $N_{\text{last\_access}}$ fields to $N$ which is the current size of the data stream, and remove the f-nodes with the sum of the value kept in its True_Count field and the value kept in its Estimate_Count field being smaller than $\sigma \times N$(these f-nodes correspond to the out-of-date frequent itemsets). If all truly frequent itemsets cannot be kept in main memory, the above procedure is useless, and hMiner becomes inefficient. However, the performance of the other approaches will get even worse under this condition as they need to keep the non-frequent itemsets.

## 6 Conclusions

In this paper, we propose the first hash-based approach, hMiner, operating in the online-processing mode for mining frequent itemsets over data streams. A newly designed data structure, hSynopsis, based on the principles of the Lossy Counting algorithm and the hCount method is used in hMiner. The entire information of the data stream can be compressed and saved in hSynopsis by hashing all the subsets of the current transaction. Moreover, the current frequent itemsets can be quickly determined from the f-nodes of hSynopsis. Rooted in hSynopsis, a novel technique is provided to estimate the support counts of the non-frequent itemsets, which makes a high precision of hMiner. From the experiments, it can be seen that EStream may result in a very low recall due to an improper parameter setting, which indicates the longest itemsets to be found. However, how to set a proper parameter in EStream for finding all the frequent itemsets is difficult without any background knowledge of the mining results. Moreover, the experiment results reveal that under almost the same precision, hMiner needs less memory space than Lossy Counting to maintain the synopses. In hMiner, the information of the non-frequent itemsets is compressed into the hash table. On the contrary, the memory space to save the information of the itemsets in Lossy Counting may drastically change. In summary, when the memory space of the environment is viewed as an important factor, hMiner is a better choice.

In this paper, we only consider to discover frequent itemsets from a single stream. However, in many applications such as analyzing the transactions from a chain of retail stores, or monitoring a large scale network with multiple routers, it may involve *multiple streams*. It implies that mining over multiple streams can discover useful knowledge. A naïve method to achieve the goal of mining over multiple streams is to collect all the data and process them in a central server (i.e., handling the multiple streams as a single stream). However, the enormous amounts of data can bring extensive computation, causing the processing time to be unacceptable. Therefore, we are currently extending hMiner to discover the global frequent itemsets from multiple streams in a distributed manner.

## References

Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: Bocca JB, Jarke M, Zaniolo C (eds) Proceedings of the 20th international conference on very large databases (VLDB1994), Santiago, Chile, pp 487–499

Arlitt M, Williamson C (1996) Web server workload characterization: the search for invariants. In: Proceedings performance evaluation review, vol 24, No. 1, pp 126–137

Calders T, Dexters N, Goethals B (2006) Mining frequent items in a stream using flexible windows. In: Cama J, Klinkenberg R, Aguilar J (eds) Proceedings of ECML/PKDD 2006 workshop on knowledge discovery from data streams (IWKDDS), Berlin, Germany, pp 87–96

Calders T, Dexters N, Goethals B (2007) Mining frequent itemsets in a stream. In: Proceedings of the seventh IEEE international conference on data mining (ICDM'07), Omaha, USA, pp 83–92

Chang JH, Lee WS (2003) Finding recent frequent itemsets adaptively over online data streams. In: Getoor L, Senator TE, Domingos P, Faloutsos C (eds) Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining (KDD2003), Washington, DC, USA, pp 487–492

Charikar M, Chen K, Farach-Colton M (2002) Finding frequent items in data streams. In: Widmayer P, Ruis FT, Bueno RM, Hennessy M, Eidenbenz S, Conejo R (eds) Proceedings of the 29th international colloquium on automata, languages and programming (ICALP'02), Málaga, Spain, pp 693–703

Cheng J, Ke Y, Ng W (2006) Maintaining frequent itemsets over high-speed data streams. In: Ng WK, Kitsuregawa M, Li J, Chang K (eds) Proceedings of the 10th Pacific-Asia conference on knowledge discovery and data mining (PAKDD 2006), Singapore, pp 462–467

Chi Y, Wang H, Yu PS, Muntz RR (2004) Moment: maintaining closed frequent itemsets over a stream sliding window. In: Proceedings of the fourth IEEE international conference on data mining (ICDM'04), Brighton, UK, pp 59–66

Cormode G, Muthukrishnan S (2003) What's hot and what's not: tracking most frequent items dynamically. In: Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems (PODS2003), San Diego, CA, pp 296–306

Dang XH, Ng WK, Ong KL (2008) Online mining of frequent sets in data streams with error guarantee. Knowl Inf Syst 16(2):245–258

Demaine E, Lopez-Ortiz A, Munro JI (2002) Frequency estimation of Internet packet streams with limited space. In: Möhring RH, Raman R (eds) Proceedings of the 10th European symposium on algorithms (ESA2002), Rome, Italy, pp 348–360

Fischer MJ, Salzberg SL (1982) Finding a majority among N votes: solution to problem 81-5. J Algorithm 3(4):362–380

Giannella C, Han J, Pei J, Yan X, Yu PS (2004) Mining frequent patterns in data streams at multiple time granularities. In: Kargupta H, Joshi A, Sivakumar K, Yesha Y (eds) Data mining next generation challenges and future directions. AAAI Press, Menlo Park, CA, pp 191–212

Golab L, DeHaan D, Demaine ED, López-Ortiz A, Munro JI (2003) Identifying frequent items in sliding windows over on-line packet streams. In: Proceedings of the first ACM SIGCOMM Internet measurement conference (IMC'03), Florida, USA, pp 173–178

Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Chen W, Naughton JF, Bernstein PA (eds) Proceedings of the 2000 ACM SIGMOD international conference on management of data (SIGMOD'00), Dallas, TX, USA, pp 1–12

Jiang N, Gruenwald L (2006) CFI-stream: mining closed frequent itemsets in data streams. In: Eliassi-Rad T, Ungar LH, Craven M, Gunopulos D (eds) Proceedings of the 12th ACM SIGKDD international conference on knowledge discovery and data mining (KDD'06), Philadelphia, USA, pp 592–597

Jin R, Agrawal G (2005) An algorithm for in-core frequent itemset mining on streaming data. In: Proceedings of the fifth IEEE international conference on data mining (ICDM'05), Houston, TX, USA, pp 210–217

Jin C, Qian W, Sha C, Yu JX, Zhou A (2003) Dynamically maintaining frequent items over a data stream. In: Proceedings of the 12th ACM international conference on information and knowledge management (CIKM'03), New Orleans, LA, USA, pp 287–294

Karp RM, Papadimitriou CH, Shenker S (2003) A simple algorithm for finding frequent elements in streams and bags. ACM Trans Database Syst 28(1):51–55

Lee D, Lee W (2005) Finding maximal frequent itemsets over online data streams adaptively. In: Proceedings of the fifth IEEE international conference on data mining (ICDM'05), Houston, TX, USA, pp 266–273

Lee LK, Ting HF (2006) A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In: Vansummeren S (ed) Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems (PODS'06), Chicago, USA, pp 290–297

Leung CKS, Khan Q (2006) DSTree: a tree structure for the mining of frequent sets from data streams. In: Proceedings of the sixth IEEE international conference on data mining (ICDM'06), Hong Kong, China, pp 928–932

Li HF, Lee SY, Shan MK (2004) An efficient algorithm for mining frequent itemsets over the entire history of data streams. The first international workshop on knowledge discovery in data streams, in conjunction with ECML/PKDD 2004, Pisa, Italy

Li HF, Ho CC, Kuo FF, Lee SY (2006) A new algorithm for maintaining closed frequent itemsets in data streams by incremental updates. In: Proceedings of IEEE international workshop on mining evolving and streaming data (ICDM workshops 2006), Hong Kong, China, pp 672–676

Lin CH, Chiu DY, Wu YH, Chen ALP (2005) Mining frequent itemsets from data streams with a time-sensitive sliding window. 2005 SIAM international conference on data mining (SDM'05), Newport Beach, CA

Manku GS, Motwani R (2002) Approximate frequency counts over data streams. In: Proceedings of the 28th international conference on very large databases (VLDB2002), Hong Kong, China, pp 346–357

Mozafari B, Thakkar H, Zaniolo C (2008) Verifying and mining frequent patterns from large windows over data streams. In: Proceedings of IEEE 24th international conference on data engineering (ICDE'08), Cancún, México, pp 179–188

Wang SY, Hao XL, Xu HX, Hu YF (2007a) Finding frequent items in data streams using ESBF. In: Proceedings of the 2007 international workshop on high performance data mining and application (HPDMA 2007), in conjunction with PAKDD 2007, Nanjing, China, pp 244–255

Wang SY, Xu HX, Hu YF (2007b) Finding frequent items in sliding windows over data streams using EBF. In: Proceedings of the eighth ACIS international conference on software engineering, artificial intelligence, networking, and parallel/distributed computing (SNPD 2007), Qingdao, China, pp 682–687

Yu JX, Chong Z, Lu H, Zhou A (2004) False positive or false negative: mining frequent itemsets from high speed transactional data streams. In: Nascimento MA, Özsu MT, Kossmann D, Miller RJ, Blakeley JA, Schiefer KB (eds) Proceedings of the 30th international conference on very large databases (VLDB2004), Toronto, Canada, pp 204–215