The Computer Journal Advance Access published February 16, 2012

doi:10.1093/comjnl/bxs010

An Efficient Frequent Itemset Mining Method over High-speed Data Streams

Mina Memar[1,*], Mahmood Deypir[2], Mohammad Hadi Sadreddini[1] and Seyyed Mostafa Fakhrahmad[3]

[1]*Department of Computer Science and Engineering, Shiraz University, Shiraz, Iran*
[2]*University of Aeronautical Science & Technology, Tehran, Iran*
[3]*Department of Computer Engineering, Islamic Azad University, Shiraz Branch, Shiraz, Iran*
**Corresponding author: mina.memar@gmail.com*

Frequent itemset mining over sliding window is an interesting problem and has a large number of applications. Sliding window is a widely used model for frequent itemset mining over data streams due to its capability to handle concept drift, its bounded memory and its low processing time. A sliding window-based algorithm requires an efficient data structure that can be updated as fast as possible when inserting and deleting transactions. Moreover, an innovative computing method is needed to provide the set of frequent patterns (FPs) with a little delay after the user issues a request for the mining results within a window. In this study, an efficient representation of the sliding window named blocked bit sequence is introduced which is aimed to store and maintain the content of the window. Moreover, by a novel technique this representation is exploited for efficiently extracting the set of FPs within the current window. Experimental evaluations on both real-life and synthetic data streams show that the proposed approach is faster than recently proposed algorithms in different phases of data stream mining.

Keywords: frequent itemsets; data stream mining; sliding window; bit sequence

Received 25 July 2011; revised 17 December 2011
Handling editor: Erol Gelenbe

1. INTRODUCTION

Mining frequent itemsets is an important task of data mining since it can be exploited in the other data mining functions such as association rules [1, 2], sequential patterns [3], classification [4] and clustering [5]. Moreover, it has a wide application area including business, science and industry. The problem of mining frequent itemsets in large databases was first proposed by Agrawal and Srikant [1] in 1993. An itemset or set of items is frequent when its frequency is not under a user-defined threshold. Traditional frequent itemset mining focused on the problem of mining static databases, which are stored in secondary storage, and can be scanned multiple times.

Owing to the growing number of data stream applications such as stock analysis, wireless sensor networks, web log analysis, telecommunication call records and network monitoring data, data stream mining has become an interesting topic. A data stream is a massive sequence of data elements that are continuously generated at a rapid rate and have a data distribution that changes with time. Owing to the characteristics of *stream data*, there are some challenges for data stream frequent itemset mining: (1) There is not enough time to perform multiple scans as in traditional data mining algorithms. Moreover, there is not enough space to store the whole stream data for future processing purposes. Therefore, a single scan of input data and low memory usage for the mining are necessary. (2) A data stream mining method needs to be adaptable to changes in underlying data distribution. (3) Data streams need to be processed as fast as possible. (4) Mining tasks over data streams should include an incremental process to be able to update the results as new data elements arrive.

There are mainly three data models in most previous works on mining frequent itemsets over data streams: landmark windows model [6], damped windows model [7] and sliding windows model [8]. The landmark model mines all frequent itemsets emerging from a particular point of time (called landmark) to the current time. The support count of an itemset in this model is the number of transactions containing it which had emerged from the landmark to the current time. Additionally, the landmark model is not aware of time; therefore, it does not distinguish between the new data items and old ones. The damped model assigns different weights to transactions such that new transactions have higher weights than older ones. All these approaches provide approximate answers for long-term data and adjust their storage requirement based on the available space. In the sliding window model, only the most recent transactions are used to determine the frequency of an item. To avoid having to count the supports on this window recurringly in every time point, the algorithm in fact updates the frequency of the itemsets based on the deletion of old transactions and insertion of new ones.

In this study, we propose an efficient single pass algorithm called *MFI-CBSW* (Mining Frequent Itemset within Circular Block Sliding Window). The *MFI-CBSW* algorithm uses the sliding window model to find frequent itemsets in a fixed number of recent transactions. The sliding window is composed of a sequence of blocks. Each block maintains a number of transactions. When the window is moving forward, the oldest block is disregarded and a new block containing newly generated transactions is appended to the window. An effective blocked bit sequence representation of items with a queue of non-zero block numbers helps us to store all transactions of the current window in a compressed format. Moreover, it is utilized for fast frequent itemset mining. Experimental results show that our algorithm is faster than recently proposed sliding window-based algorithms.

The rest of this paper is organized as follows. In Section 2, we give an overview of the related work. Section 3 states the problem. The previously proposed MFI-TRANSW [9] approach is reviewed in Section 4. Our proposed algorithm is presented in Section 5. Experimental evaluation and conclusion are described in Sections 6 and 7, respectively.

2. RELATED WORK

The first and most representative method for finding frequent itemsets in a static dataset is the Apriori algorithm [1]. This algorithm requires multiple scans of a database. Therefore, considering the requirements of the data stream processing, this is not suitable for finding frequent itemsets over an online data stream. Han *et al*. [10] proposed the frequent pattern tree (FP-tree) and the FP-growth algorithm. This algorithm reduces the number of database scans and eliminates the requirement for candidate generation. Introduction of this highly compact FP-tree structure led to a new avenue of research with regard to mining FPs with a prefix-tree structure. However, the static nature of an FP-tree and the requirement for two database scans limits the applicability of this algorithm to FP mining over a data stream.

The first algorithm named *Lossy Counting*, which is used for frequent itemsets over the entire history of a data stream, was proposed by Manku and Motwani [6]. The algorithm is a first single pass algorithm based on a well-known Apriori property proposed for data streams [1]. *Lossy Counting* uses a specific array to represent the lexicographic ordering of the hash tree, which is the popular method for candidate counting [1]. Chang and Lee [11] developed an algorithm (called *estDec*) for mining frequent itemsets in streaming data in which each transaction has a weight that decreases with the age. In other words, older transactions contribute less toward itemset frequencies. Giannella *et al*. [7] developed a FP-tree-based algorithm, called *FPstream*, to mine frequent itemsets at multiple time granularities by a novel tilted-time windows technique. *DSM-FI* [12] is a landmark-based algorithm. In this algorithm, every transaction is converted into smaller transactions and inserted into a summary data structure called *item-suffix frequent itemset forest*, which is based on the prefix-tree structure. In [8], the authors used the *Chernoff bound* to produce an approximate result of FPs over the landmark window. Zhi-Jun *et al*. [13] used a lattice structure, referred to as a *frequent enumerate tree*, which is divided into several equivalent classes of stored patterns with the same transaction-ids in a single class.

DSTree [14] and *CPS-Tree* [15] are two algorithms that use the prefix tree to store raw transactions of the sliding window. *DSTree* uses a fixed-tree structure in canonical order of branches while in *CPS-Tree* the prefix tree structure is reconstructed to control the amount of memory usage. Both of Leung and Khan [14] and Tanbeer *et al*. [15] perform the mining task using the *FP-Growth* [10] algorithm that was proposed for static frequent itemset mining.

Lin *et al*. [16] proposed a new method for mining FPs over the time-sensitive sliding window. In their method, the window is divided into a number of batches for which itemset mining is performed separately. At each time, a different number of transactions are received from a stream. The *SWIM* [17] is a pane-based algorithm in which frequent itemsets in one pane of the window are considered for further analysis to find frequent itemsets in the whole of the window. It keeps the union of FPs of all panes and incrementally updates their supports and prunes infrequent ones. It stores transactions of the window in the form of the prefix tree of each pane. Chang and Lee proposed the *estWin* algorithm [18], which finds recent FPs adaptively over transactional data streams using the sliding window model. This algorithm uses a reduced minimum support threshold named significance to early monitoring of itemsets before they become actually a frequent itemset.

In [19], Chi *et al*. proposed a closed frequent itemset mining algorithm for the sliding window model which uses the Closed Enumeration Tree (*CET*) to maintain the main information of itemsets. When the window slides, a new transaction arrives and an old transaction disappears; then a node is inserted and updated or deleted based on its type. In [20], Li *et al*. proposed the *NewMoment* algorithm with bit-sequence to denote the occurrence of an item within the sliding window's transactions. Furthermore, it maintains a data structure called *NewCET* to only store the closed frequent nodes. Therefore, the memory and running time costs are reduced. On the basis of the *Moment* algorithm, a new approach for non-derivable frequent itemset mining is proposed [21]. Similarly to closed itemsets, the set of non-derivable frequent itemsets is a compact representation of all frequent itemsets. In [21], an efficient algorithm called *IncSPAM* was proposed to maintain sequential patterns over the sliding window. The concept of bit-vector, Customer Bit-Vector Array with Sliding Window, is introduced to efficiently store the information of items for each customer. The representation can reduce the memory requirement and execution time for online maintenance. Jin and Agrawal [22] proposed an algorithm, called *StreamMining*, for in-core frequent itemset mining over online data streams.

In [9], Li and Lee proposed two Apriori-based algorithms, called *MFI-TransSW* and *MFI-TimeSW*, to find the complete set of frequent itemsets in online data streams with a transaction-sensitive sliding window and time-sensitive sliding window. In these algorithms, bit-sequences are used to keep the occurrence of items in recent sliding windows' transactions. This representation is used to reduce the time and memory required for window sliding. Our proposed approach is similarly to [9], however, it is more efficient in both sliding and mining phases.

3. PROBLEM STATEMENT

Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of items. A subset of I named X is denoted by an itemset. An itemset having k items is named a k-itemset. Each transaction $T = (t_{\mathrm{id}}, D)$ contains a unique number named t_{id} and an itemset named D. A transactional data stream $\mathrm{DS} = [t_1, t_2, \ldots, t_n)$ is a sequence of transactions received from a source in which t_i is the ith transaction of DS. Window W refers to a subsequence of DS between the ith and jth transactions, where $j > i$. The size of the window denoted by $|W|$ is equal to $j - i + 1$ transactions. A sliding window moves forward by inserting a fixed number of new transactions and deleting the same number of oldest ones. Therefore, the number of transactions remains fixed. This unit of insertion and deletion is named a block. The number of transactions of a block is called the size of the block and denoted by b. The support count of an itemset X in window W, $\sup(X)^{\mathrm{W}}$, is the number of transactions of the window containing the itemset. Let s be a user given the minimum support count threshold. An itemset X is frequent if $\sup(X)^{\mathrm{W}} \geq s$. Having a minimum support threshold, the problem is defined as finding all frequent itemsets of the last active window.

Example 1. In Fig. 1, 12 recently arrived transactions of a data stream DS are shown. The sliding window model is used to process this stream in which the window and block lengths are set to 9 and 3, respectively. Therefore, two windows W_1 and W_2 are shown on this data stream, each of which contains three blocks of transactions. Each block is depicted in a separate line. Assuming minimum support of 25%, frequent itemsets included in the first and second windows are shown in Fig. 1.

4. A REVIEW ON MFI-TRANSW

As mentioned previously, the proposed algorithm is based on the MFI-TRANSW [9]. Therefore, this section describes this algorithm. MFI-TRANSW uses a bit-sequence representation for storing and maintaining the occurrences of an item within transactions of a window. The aim of this data structure is to compress information in the memory and provide fast bit operations. The algorithm consists of three main phases: window initialization, window sliding and frequent itemset mining. In the window initialization phase, a number of transactions equal to the window size are read one by one from the input stream. For each item a bit-sequence tracks the occurrences of the item within incoming transactions. The window sliding phase starts after the initialization. The aim of this phase is to update the window content using new transactions. Upon arrival of a new transaction, the oldest transaction of the window is removed and the new transaction is inserted. For removing the oldest transactions, a right shift is performed on all bit sequences. Suppose that the order of each bit sequence is from the right to the left. After deleting the oldest transactions, for each bit sequence, based on the appearance of the corresponding item in the new transaction a 0 or 1 is placed at the leftmost position. The mining phase is started when a user issues a request. By applying the Apriori algorithm [1] on the bit sequences, frequent itemsets of the current window are extracted. For each item, the support count is equal to the number of '1's in its bit sequence. The support of candidate k-itemsets ($k > 1$) is calculated by first performing

Win ID		*DS*	Frequent Itemsets of W_1	Frequent Itemsets of W_2
1		$T_1(be)$, $T_2(abe)$, $T_3(ae)$	(a), (b), (c), (d), (e), (ae), (ad), (bc), (be), (ce), (de), (ade)	
	2	$T_4(bce)$, $T_5(bc)$, $T_6(ce)$		(a), (b), (c), (d), (e), (ac), (ad), (ae), (bc), (ce), (de), (ace), (ade)
		$T_7(ade)$, $T_8(e)$, $T_9(ade)$		
		$T_{10}(ace)$, $T_{11}(ac)$, $T_{12}(ace)$		
		⋮	⋮	⋮

FIGURE 1. An example data stream and the frequent itemsets over two consecutive windows.

AND operation on its $k-1$ subsets and then counting the number of 1's in the resultant bit sequence.

By using bit sequences, the amount of required memory for storing transactions is reduced, and moreover the sliding and mining processes are enhanced by exploiting bit operations. However, this algorithm suffers from a number of shortcomings. First, for each item, besides a bit sequence, this algorithm does not use extra information to enhance the sliding and mining processes. Secondly, the shift operations performed on large bit sequences is time inefficient. Thirdly, the sliding process is performed for each new transaction. It will be more efficient to increase the unit of insertion and deletion so that the same process can be applied on a block of transactions instead of one transaction.

5. PROPOSED ALGORITHM

In this study, a novel technique named CBSW is developed to efficiently maintain sliding window transactions over data streams. In this technique, for each item, a blocked bit sequence is stored and dynamically updated. The window sliding process is managed efficiently to reduce the cost of this process. A circular queue is used to track non-zero blocks of the window. The mining phase uses this queue for efficient extraction of frequent itemsets when the user submits a request.

5.1. Blocked bit sequence of items

In bit sequence representation of items proposed in [9], for each item X in a window W, a bit sequence named Bit(X) having length $|W|$ is created. In Bit(X), if the item appears in the ith transaction of the window, its corresponding bit will be set; otherwise it will be cleared. However, in our devised blocked bit sequence, each window is decomposed to a number of blocks with length b. Therefore, for each item, there are $n = |W|/b$ blocks in which the item occurrences are shown by setting the corresponding bits. A blocked bit sequence of an item X is called BBit(X). Similarly to the window size, block size is fixed during the input stream data processing and is given by the user. A window containing blocked bit sequence of items is named blocked window. On the basis of the definition of blocked bit sequence, the corresponding blocked window W_1 of Fig. 1 is as follows:

$\mathrm{BBit}(a) = \boxed{101}\,\boxed{000}\,\boxed{110}$, $\mathrm{BBit}(b) = \boxed{000}\,\boxed{011}\,\boxed{011}$,
$\mathrm{BBit}(c) = \boxed{000}\,\boxed{111}\,\boxed{000}$, $\mathrm{BBit}(d) = \boxed{101}\,\boxed{000}\,\boxed{000}$,
$\mathrm{BBit}(e) = \boxed{111}\,\boxed{101}\,\boxed{111}$.

In this blocked bit sequence, the ordering of blocks is from right to left. Each block can be regarded as a binary number, which is equivalent to a decimal number. As can be seen from the above paragraph, items b, c and d have some zero blocks. In our technique, for each item, a queue is used to store the indices of non-zero blocks. By numbering blocks of window from left to right, the values of the queue for each item are as follows:

$NzQ(a) = 1, 3$, $NzQ(b) = 1, 2$,
$NzQ(c) = 2$, $NzQ(d) = 3$,
$NzQ(e) = 1, 2, 3$.

Since we have a restricted number of non-zero blocks for each blocked bit sequence, storing these block numbers involves very small memory usage. FP mining over data streams using the proposed mechanism includes: window initialization, window sliding and mining. Using the novel circular blocked window mechanism, these phases are performed efficiently.

5.2. Window initialization

The first phase of our method is window initialization. With the incoming of first $|W|$ transactions from the input stream, this phase is started. In this phase, based on incoming transactions, blocked bit sequences and non-zero queues for items are created. Figure 2 shows the window content after the initialization.

As shown in Fig. 2, for each item, the blocked bit sequence shows the item occurrences in all blocks of the window. Non-zero blocks are stored in the corresponding queue of the item. Block numbers of the front and the rear of all queues are also held. These numbers are identical for all items and are used in the window sliding phase.

5.3. Window sliding

After incoming $|W|$ transactions and constructing blocked bit sequence of each item of the window, window initialization is completed. In the sliding window model, new coming transactions are inserted into the window and old transactions are removed from the window. This process is named window sliding. In the *MFI-TransSW* algorithm [9], each sliding of the window includes deletion of the oldest transaction and insertion of a new transaction. A right shift operation is applied on all bit sequences of items to remove information of the oldest transaction. For each item of the window, if the new transaction includes the item, the leftmost bit of the corresponding bit sequence is set to 1; otherwise it is set to 0. Consequently, the window content is updated and window sliding is performed. By deleting the oldest transaction and

Win ID	Transactions	Blocked bit sequences

W1	$T_1(be)$, $T_2(abe)$, $T_3(ae)$
	$T_4(bce)$, $T_5(bc)$, $T_6(ce)$
	$T_7(ade)$, $T_8(e)$, $T_9(ade)$
		{Bbit(a)= $\boxed{101}\,\boxed{000}\,\boxed{110}$, NzB(a)=1,3}
		{Bbit(b)= $\boxed{000}\,\boxed{011}\,\boxed{011}$, NzB(b)=1,2}
		{Bbit(c)= $\boxed{000}\,\boxed{111}\,\boxed{000}$, NzB(c)=2}
		{Bbit(d)= $\boxed{101}\,\boxed{000}\,\boxed{000}$, NzB(d)=3 }
		{Bbit(e)= $\boxed{111}\,\boxed{101}\,\boxed{111}$, NzB(e)=1,2,3}

		Front Block number =1 Rear Block Number=3

FIGURE 2. Window content after initialization.

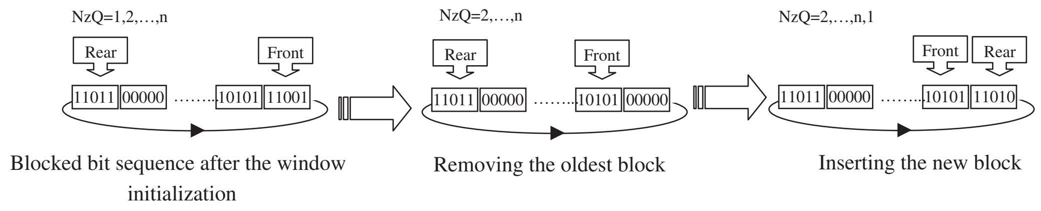


FIGURE 3. Window sliding process on sample blocked bit sequence.

Win ID	Transactions	Blocked bit sequences
W_2	*T_4(bce), T_5(bc), T_6(ce)* *T_7(ade), T_8(e), T_9(ade)* *T_{10}(ace), T_{11}(ac), T_{12}(ace)*	{BBit(*a*)=$\boxed{101}\ \boxed{000}\ \boxed{111}$, NzB (*a*)= 3,1} - { BBit(*b*)=$\boxed{000}\ \boxed{011}\ \boxed{000}$, NzB(*b*)=2} {BBit(*c*)=$\boxed{000}\ \boxed{111}\ \boxed{111}$, NzB(*c*)=2,1 } - { BBit(*d*)=$\boxed{101}\ \boxed{000}\ \boxed{000}$, NzB(*d*)=3} {BBit(*e*)=$\boxed{111}\ \boxed{101}\ \boxed{101}$, NzB(*e*)= 2,3,1}
		Front Block Number = 2 Rear Block Number =1

FIGURE 4. Window content after sliding from W_1 to W_2.

inserting a new transaction, the window size $|W|$ is preserved. The process of window sliding using right shift of all bit sequences is inefficient, especially when the window size is large. As an example, suppose that the bit sequence of a typical item e is $\text{Bit}(e) = 111101111$. By performing right shift operation, the bit sequence becomes $\text{Bit}(e) = 011110111$. As can be inferred, for a right shift, $|W| - 1$ operations are performed to move all bits to the right. Therefore, this process is inefficient for windows having large number of items. For example, for a window having length of 10000 and containing 1000 items, deleting the oldest transaction by right shift on all bit sequences needs 1000×9999 moves, which is a time inefficient operation.

To overcome this shortcoming, here we have utilized our blocked bit sequence to efficiently perform the window sliding process. We develop a new technique namely CBS, which significantly improves the window sliding process. In this technique, the unit of sliding is a block instead of a single transaction. Therefore, the sliding phase includes inserting a new block and deleting the oldest one from the window. In the CBS technique, each blocked bit sequence is regarded as a circular queue. Obviously, after the window initialization the front and rear numbers of all block queues are 1 and n, respectively. For removing the oldest block, in each blocked bit sequence, corresponding bits of the block are set to zero. In other words, the block located at the front of the queue is filled by zero. Additionally, if the removed block is a non-zero block, its block number will also be deleted from the corresponding non-zero queue. Subsequently, the front variable of the block circular queues is adjusted; that is, the value of the front variable is set to the next block number based on the rules of the circular queue. Similarly to the removing step, insertion of the new block is performed efficiently. Simply, for each item, the removed block is replaced with the new block and the new block will be located at the new rear of the block queue. If the inserted block is a non-zero block, its number is appended to the corresponding non-zero queue of the item. Figure 3 shows the window sliding process.

For example, consider the data stream of Fig. 2 once more. The window size and the block size are 9 and 3, respectively. This figure shows the first window containing transactions T_1 to T_9 after the window initialization. Figure 4 shows continuation of this stream after a window sliding. In this process, transactions T_1 to T_3 are removed from the window by setting the front blocks to zero. Subsequently, information about transactions T_{10} to T_{12} are represented in the block. On the basis of the previous front and rear block numbers, these values will, respectively, be 2 and 1 after sliding.

5.4. The mining process

By the user request, the mining phase is started to extract the set of frequent itemsets of the current window. Before describing the mining phase, the method that we use to compute the support of frequent itemsets is explained.

In *MFI-TransW* [9] method, the bit sequence of an itemset XY is constructed using its subsets X and Y using the AND operator as: $\text{Bit}(XY) = \text{Bit}(X)$ AND $\text{Bit}(Y)$ and then its support is computed by enumerating the number of '1's in the bit sequence. In our approach, using the non-zero queue of each item, the support of an itemset is calculated efficiently. To compute the support of an itemset, first its blocked bit sequence is constructed using its subsets. For an itemset XY, the item (X or Y) that has a smaller number of non-zero blocks is selected, according to the corresponding non-zero queues. Logical bitwise AND operation is then performed on the non-zero blocks of the selected itemset and their corresponding non-zero blocks in the other itemset. Other blocks that are not contained in the list of non-zero blocks of the selected itemset are filled by zero in the resulting block sequence of *XY*. The following pseudo-code describes this process.

```
if $|NzQ(X)| \leq |NzQ(Y)|$
    $\forall p \in NzQ(X): \ \mathrm{BBit}(XY)[p] = (\mathrm{BBit}(X) \text{ AND } \mathrm{BBit}(Y)),$
else
    $\forall p \in NzQ(Y): \ \mathrm{BBit}(XY)[p] = (\mathrm{BBit}(X) \text{ AND } \mathrm{BBit}(Y)).$
```

After constructing each block of the blocked bit sequence *XY*, if the block is not zero, its number will be inserted into the non-zero queue of the itemset. For example, consider two items *b* and *c* that are used to compute the support of *ab*. Since the number of non-zero blocks of *c* is smaller than the number of *b*'s non-zero blocks, it is selected.

$$\mathrm{Bbit}(b) = \boxed{000}\boxed{011}\boxed{011}, \ NzB(b) = 1, 2,$$
$$\mathrm{Bbit}(c) = \boxed{000}\boxed{111}\boxed{000}, \ NzB(c) = 2.$$

Then the blocked bit sequence of the itemset *bc* is computed as follows:

$$\mathrm{Bbit}(bc)[2] = (\mathrm{Bbit}(b)[2] \text{ AND } \mathrm{Bbit}(c)[2]) = 011 \text{ AND } 111 = 011$$

Since $\mathrm{Bbit}(bc)[2] > 0$, its number is inserted into the non-zero queue of the itemset, and the final result is

$$\mathrm{Bbit}(bc) = \boxed{000}\boxed{011}\boxed{000}, \ NzB(bc) = 2.$$

To extract frequent itemsets from the current window, the first step is identifying frequent single items. As described previously, the number of '1's in the corresponding non-zero block is equal to the support of an item. If this number exceeds the minimum support threshold, it is inserted into the set of frequent items (FI_1). To find other itemsets, similarly to the *Eclat* algorithm [23] we have used a depth first method of traversing the prefix tree of itemsets. This approach is a recursive process in which, at each depth, the candidate itemsets are generated by joining frequent itemsets of the previous level. After joining, the support of the generated itemset is computed using the method illustrated above. The process of itemset generation continues until no further frequent itemset can be generated. Subsequently, the generation process in the reverse direction (from bottom to up) produces other frequent itemsets in the next branches. This depth first process and the resulting prefix tree generated as the result of the recursive process are shown in Fig. 5.

For example, consider candidate itemset generation and frequent itemset mining within W_2 of Fig. 4. Suppose that the minimum support is 0.2 and thus the minimum frequency of the window size 9 becomes 2 ($0.2 \times 9 = 1.8 \cong 2$). As mentioned previously, the first step is finding frequent items by computing the support of all single items using the described approach. The resulting support values of items are $\mathrm{Sup}(a) = 5$, $\mathrm{Sup}(b) = 2, \mathrm{Sup}(c) = 6, \mathrm{Sup}(d) = 2l, \mathrm{Sup}(e) = 7$, all of which are frequent since they exceed the minimum support threshold. The set of frequent items (FI_1) and the corresponding blocked bit sequences are shown in Fig. 6.

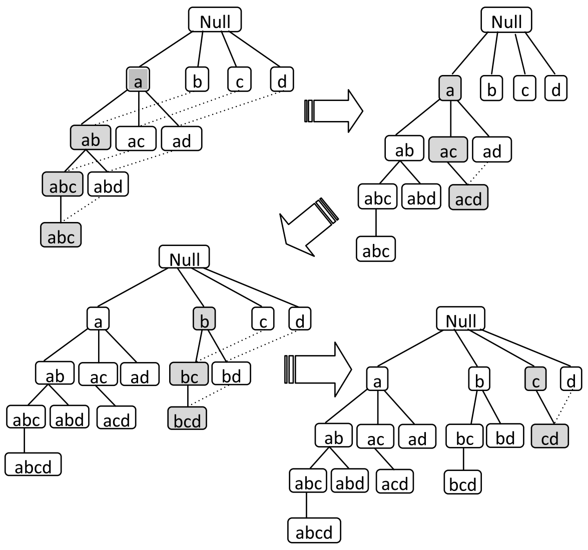


FIGURE 5. Depth first mining process.

FI_1	{BBit(*a*)=$\boxed{101}\boxed{000}\boxed{111}$, NzB (*a*)= 3,1} - { BBit(*b*)=$\boxed{000}\boxed{011}\boxed{000}$, NzB(*b*)=2}
	{BBit(*c*)=$\boxed{000}\boxed{111}\boxed{111}$, NzB(*c*)=2,1 } - { BBit(*d*)=$\boxed{101}\boxed{000}\boxed{000}$, NzB(*d*)=3}
	{BBit(*e*)=$\boxed{111}\boxed{101}\boxed{101}$, NzB(*e*)= 2,3,1}

FIGURE 6. Frequent items of W_1.

After generating FI_1, the first item *a* in alphabetical order must be processed in a depth first manner to compute all frequent itemsets including item *a*. By joining item *a* with other items according to the above approach and computing their support, 2-itemsets including item *a* are found. Subsequently, this process is repeated for *ac*; that is, *ac* is jointed with the other siblings of *a* including *ad* and *ae* to generate *acd* and *ace*. After computing their support, *ace* is found as a frequent 3-itemset since its support is 2, which equals the minimum support threshold. Since there is no frequent 3-itemset in this depth, the process stops for this branch and we return to the previous depth and the next itemset in FI_2 (*ad*). The steps followed to find frequent itemsets of W_2 are shown in Fig. 7. In this figure, CI_k and FI_k show candidate and frequent *k*-itemsets, respectively.

6. EXPERIMENTAL EVALUATION

In this section, the performance of the *MFI-CBSW* algorithm is empirically evaluated and compared with the recently proposed *MFI-TransSW* [9] and *CPS-TREE* [15]. All programs were written in C++ using Microsoft Visual Studio 2008. In *MFI-CBSW*, the length of each block is set to 32 since the length of integer is 32 bits. All experiments are performed on a system

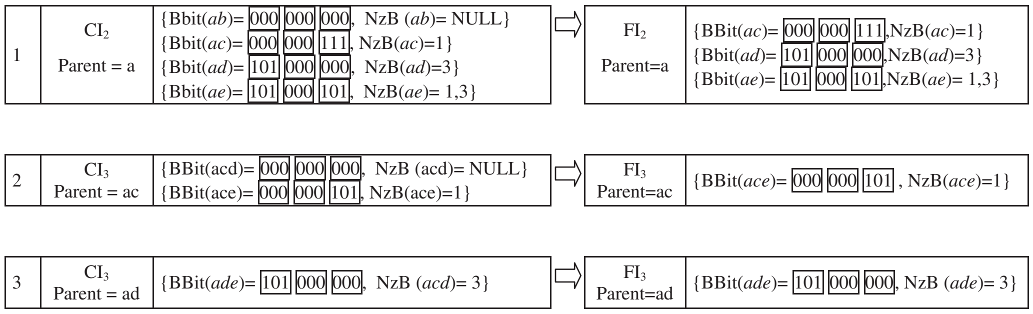

1
CI_2
Parent = a
{Bbit(ab)= 000 000 000, NzB (ab)= NULL}
{Bbit(ac)= 000 000 111, NzB(ac)=1}
{Bbit(ad)= 101 000 000, NzB(ad)=3}
{Bbit(ae)= 101 000 101, NzB(ae)= 1,3}
FI_2
Parent=a
{BBit(ac)= 000 000 111,NzB(ac)=1}
{Bbit(ad)= 101 000 000,NzB(ad)=3}
{Bbit(ae)= 101 000 101,NzB(ae)= 1,3}
2
CI_3
Parent = ac
{BBit(acd)= 000 000 000, NzB (acd)= NULL}
{BBit(ace)= 000 000 101, NzB(ace)=1}
FI_3
Parent=ac
{BBit(ace)= 000 000 101 , NzB(ace)=1}
3
CI_3
Parent = ad
{BBit(ade)= 101 000 000, NzB (acd)= 3}
FI_3
Parent=ad
{BBit(ade)= 101 000 000, NzB (ade)= 3}

FIGURE 7. Process of finding frequent itemsets of W_2.

Dataset	#Trans.(T)	#Items(I)	MaxTL(MTL)	AvgTL(ATL)	(ATL/I)) ×100
BMS-POS	515,597	1657	164	6.53	0.39
Kosarak	990,002	41,270	2498	8.10	0.02
T40I10D100K	100,000	1000	77	39.61	4.20

FIGURE 8. Dataset characteristics.

with a 2.4 GHZ CPU and having 4 GB RAM, which uses Windows XP operating system. Window size and minimum support thresholds are two important factors that directly affect the performance of every FP mining algorithm for data streams. Here, by varying these parameters, we compare all algorithms in terms of runtime and memory usage. Both real and synthetic datasets are used for the evaluation. Some statistical information about the datasets used in the experimental analyses is provided in Fig. 8. First two datasets are real and the last dataset is an artificial dataset. BMS-POS contains several years' worth of point-of-sale data from a large electronics retailer. Kosarak is a dataset of click-stream data from a Hungarian online news portal. The T40I10D100K is created by the synthetic data generator described in [1], where the parameters T, I and D represent the average transaction size, average maximal potentially FPs and the number of transactions, respectively. In Fig. 8, columns named #Trans and #Items are the number of transactions and number of distinct items that appeared in different transactions, respectively. The fourth column shows the maximum length of transactions of each dataset and the fifth column depicts the average length of transactions of datasets. The last column of this figure shows the percentage of total distinct items that appear in each transaction in each dataset. This parameter provides a measure of whether the dataset is sparse or dense. In general, a sparse dataset contains fewer items per transaction and many distinct items in total. A dense dataset, in contrast, has many items per transaction with few distinct items. Therefore, when the value of this parameter for a dataset is relatively low (e.g. less than or equal to 10.0), we define the dataset to be sparse. Otherwise, it is considered a dense dataset.

In the first experiment, the average sliding time of all algorithms are compared. The window sliding time is measured for different values of window sizes. For computing the average sliding time, first the total time for sliding over each dataset is measured and then this time is divided by the number of active windows to process the dataset. This process is repeated for each size of window for all algorithms on T40I10D100K, *kosarak* and BMS-POS datasets. The results are shown in Fig. 9. The horizontal axis shows that the window size and vertical axis represent the sliding time. This figure shows that the proposed algorithm has faster sliding time compared with both *MFI-TRANSW* and *CPS-Tree* algorithms. As can be inferred from Fig. 9, as the window size increases, the average sliding time of *CPS-Tree* and *MFI-TRANSW* also increases. However, the sliding time of *MFI-CBSW* remains almost constant. Therefore, the window sliding time of *MFI-CBSW* is independent of the window size. In the proposed algorithm, for window sliding, a block is cleared and filled by new information for each item of the window. The size of the window does not affect this process. However, in *MFI-TRANSW* the required time to perform a right shift on a whole bit sequence is directly related to the size of the bit sequence, which is proportional to the size of the window. As the size of bit sequences increases, the right shift processing times increase, which leads to a larger sliding time. For *CPS-Tree*, for the sliding process, it is required to traverse the tree, remove branches related to old transactions and also insert new branches for added transactions. As the size of the window increases, the required time to traverse a larger tree also increases. Moreover, *CPS-Tree* needs to perform tree reconstruction to maintain support descending order of items. This reconstruction takes a longer time for a larger tree. Therefore, *CPS-Tree* has larger sliding time for larger windows.

According to Figs 8 and 9, the average sliding times of the algorithms on each dataset directly depend on the number of items within the dataset. For example, average sliding times of the algorithms on the Kosarak dataset are greater than those of the others. However, this increase is not that great for MFI-CBSW.

In the second experiment, the time required to extract complete sets of FPs are compared. This experiment is also repeated for different window sizes. In this experiment, after each sliding, a mining is applied on the active window. The

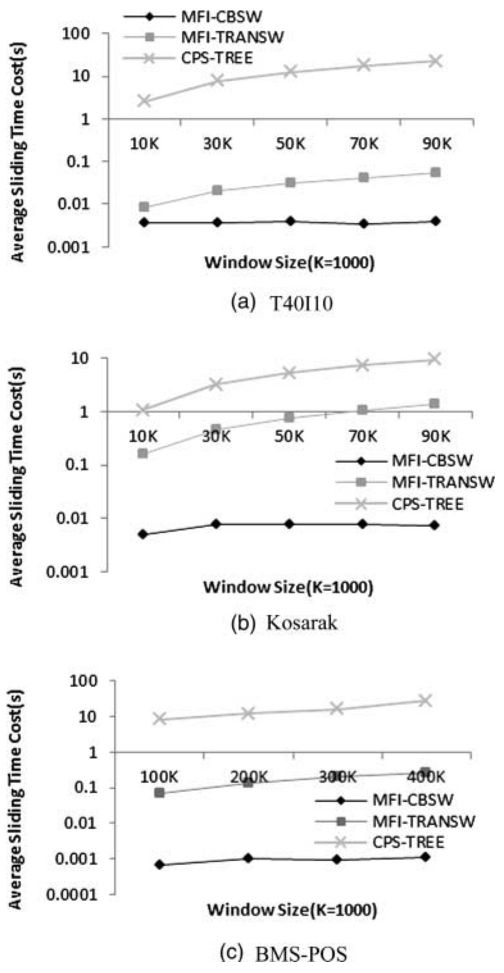


FIGURE 9. Average runtime of the window sliding phase.

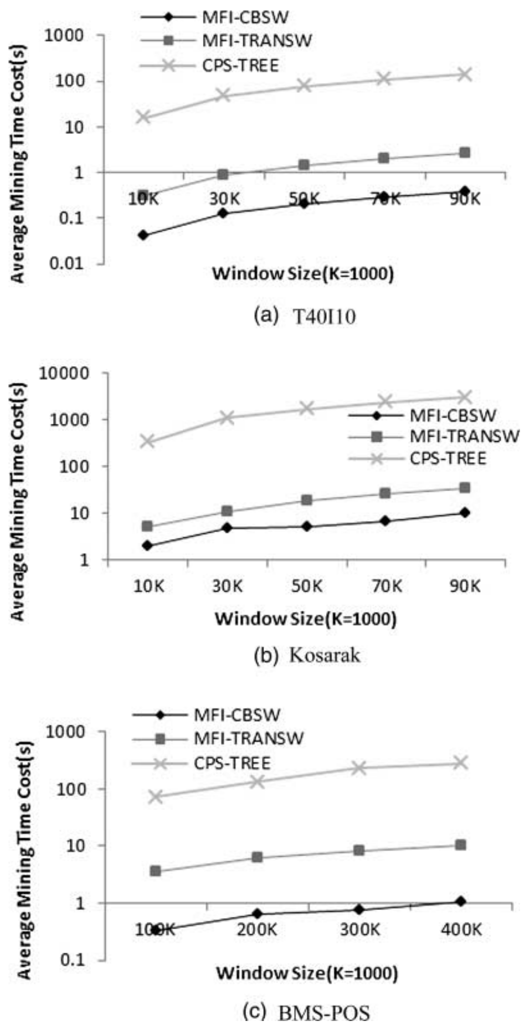


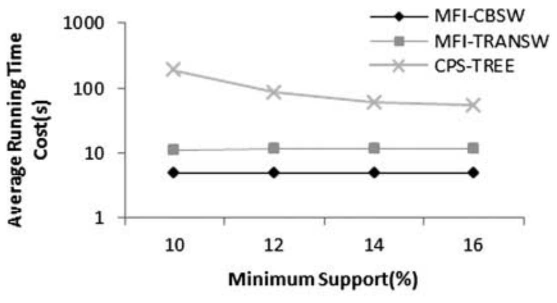
FIGURE 10. Average runtime of the mining phase.

average mining time of a window is measured and the results are shown in Fig. 10. Horizontal and vertical axes show the window size and the mining time, respectively. As shown in this figure, the average mining time of *MFI-CBSW* is significantly lower than other algorithms for all window sizes. The reason for this superiority of *MFI-CBSW* and *MFI-TRANSW* with respect to the *CPS-Tree* is the use of bit operations for computing supports. Moreover, *MFI-CBSW* utilizes zero blocks for fast support computations and merging itemsets, which lead to a better mining time than *MFI-TRANSW*.

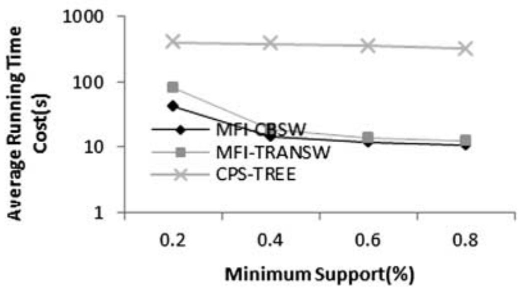
According to Fig. 10 and dataset information represented by Fig. 8, for all algorithms, the average mining time for the Kosarak dataset is greater than for both T40I10 and BMS-POS datasets. This is due to the larger number of items contained in this dataset. Other properties of the datasets do not have significant effect on sliding and mining times of the algorithms. Although these datasets have different characteristics, *MFI-CBSW* is superior to other algorithms for all of them, which shows the generality of the proposed algorithm in terms of sliding and mining times.

In the third experiment, total runtimes of all algorithms are compared together. The total time includes window initialization, window sliding and the mining process for each dataset by each algorithm. For this experiment, the window size is fixed to 150 K for all datasets and average runtimes are measured using the number of active windows. This experiment is performed for different values of the minimum support threshold to see the impact of this parameter on the runtime of each algorithm. The results are plotted on Fig. 11, where each subfigure compares the runtime of all algorithms on one dataset. As can be seen from Fig. 11, *MFI-CBSW* has better runtime on all datasets in comparison with other algorithms. This is due to enhancements of the algorithm in the sliding and mining phases.

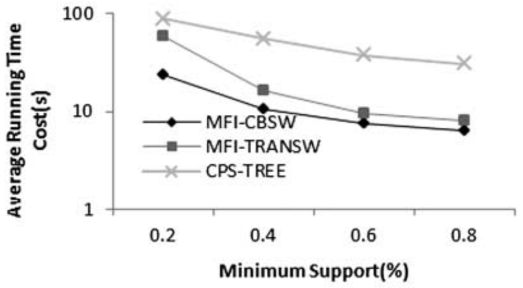
In the last experiment, memory usages of all algorithms are compared. The memory required to maintain the window transactions for different window sizes is reported here, since this parameter directly impacts the memory requirements. In



(a) T40I10

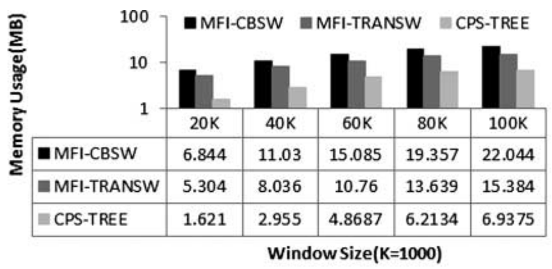


(b) Kosarak



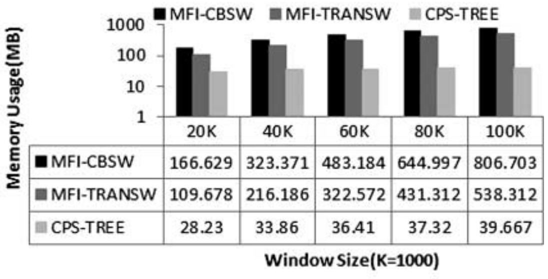
(c) BMS-POS

FIGURE 11. Average runtime comparison.



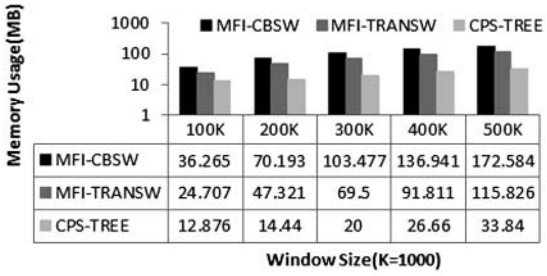
	20K	40K	60K	80K	100K
■ MFI-CBSW	6.844	11.03	15.085	19.357	22.044
■ MFI-TRANSW	5.304	8.036	10.76	13.639	15.384
■ CPS-TREE	1.621	2.955	4.8687	6.2134	6.9375

(a) T40I10



	20K	40K	60K	80K	100K
■ MFI-CBSW	166.629	323.371	483.184	644.997	806.703
■ MFI-TRANSW	109.678	216.186	322.572	431.312	538.312
■ CPS-TREE	28.23	33.86	36.41	37.32	39.667

(b) Kosarak



	100K	200K	300K	400K	500K
■ MFI-CBSW	36.265	70.193	103.477	136.941	172.584
■ MFI-TRANSW	24.707	47.321	69.5	91.811	115.826
■ CPS-TREE	12.876	14.44	20	26.66	33.84

(c) BMS-POS

FIGURE 12. Memory usage comparison.

this experiment, for all datasets the peak of memory usage for each window size is measured and depicted in Fig. 12.

As shown in Fig. 12, the *CPS-Tree* has better memory usage for all datasets and all window sizes. This is due to high prefix sharing among branches of the tree. Moreover, it performs restructuring on the tree to preserve support descending order of items for high prefix sharing of branches. The memory usage of *MFI-CBSW* is worse than *MFI-TRANSW* since it maintains additional queue for storing zero block numbers. However, this additional data structure is utilized by *MFI-CBSW* for better sliding and mining times. On the basis of the memory usage data of Fig. 12, for different window sizes, the difference between the amounts of required space for MFI-CBSW and MFI-TRANSW is fixed. Although MFI-CBSW has larger memory requirement, its memory usage does not prohibitively increase as the window size becomes larger and the additional data structure of MFI-CBSW can be easily stored in the memory of current computing systems.

7. CONCLUSION

In this paper, an efficient single-pass algorithm called *MFI-CBSW* for mining frequent itemsets over a sliding window for data streams is proposed. The sliding window is composed of a sequence of blocks. Each block maintains a number of transactions. When the window is moving forward, the oldest block is discarded and a new block containing newly generated transactions is appended to the window. In this algorithm, an innovative data structure named blocked bit sequence along with a non-zero block number queue is used. This effective blocked bit sequence representation of items with a queue of non-zero block numbers helps to store all transactions of the current window in a compressed format. This data structure enhances the efficiency of the window sliding and mining processes. Moreover, a novel technique named CBSW is used to efficiently maintain sliding window transactions over data streams. Circular block movement enhances the process of sliding even further by reducing the cost of this process. A queue is used to track non-zero blocks of the window. Utilizing non-zero block numbers during the mining prevents the block bit operations for which the results are predetermined. The mining phase uses this queue for efficient extraction of frequent itemsets when the user submits a request.

Experimental evaluations show the superiority of the proposed algorithm in terms of window sliding and mining runtimes. However, the algorithm involves little increase in the memory usage due to storing of non-zero block numbers. This extra memory usage is not prohibitive since it does not increase significantly for large window sizes.

As future work, instead of storing the whole of a bit sequence including zero blocks and a non-zero block number queue, an efficient hash table can be used to maintain non-zero block information. In this hash table, block numbers and blocks are keys and values, respectively. By using this approach both of our main data structures can be combined together in order to produce a more concise data structure.

FUNDING

The partial financial support by the Iranian research institute of Information and Communication Technology (ICT) is kindly acknowledged.

REFERENCES

[1] Agrawal, R. and Srikant, R. (1994) Fast Algorithms for Mining Association Rules in Large Databases. *Proc. VLDB*, Santiago de Chile, Chile, pp. 487–499.

[2] Kargupta, H. *et al*. (2004) VEDAS: A Mobile and Distributed Data Stream Mining System for Real-Time Vehicle Monitoring. *Proc. SDM*. Orlando, Florida, USA.

[3] Pei, J., Han, J, Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U. and Hsu, M. (2004) Mining sequential patterns by pattern-growth: the PrefixSpan approach. *IEEE Trans. Knowl. Data Eng*., **16**, 1424–1440.

[4] Hulten, G., Spencer, L. and Domingos, P. (2001) Mining Time-Changing Data Streams. *Proc. KDD*, San Francisco, CA, USA, pp. 97–106.

[5] Guha, S., Mishra, N., Motwani, R. and O'Callaghan, L. (2000) Clustering Data Streams. *Proc. IEEE Symp. Foundations of Computer Science*, Redondo Beach, CA, USA, pp. 359–366.

[6] Manku, G.S. and Motwani, R. (2002) Approximate Frequency Counts over Data Streams. *Proc. VLDB*, Hong Kong, China, pp. 346–357.

[7] Giannella, C., Han, J., Pei, J., Yan, X. and Yu, P.S. (2002) Mining Frequent Patterns in Data Streams at Multiple Time Granularities. *Proc. NSF Workshop on Next Generation Data Mining*, Marriott, Inner Harbor, Baltimore.

[8] Yu, J.X., Chong, Z., Lu, H., Zhang, Z. and Zhou, A. (2006) A false negative approach to mining frequent itemsets from high speed transactional data streams. *Inf. Sci.*, **176**, 1986–2015.

[9] Li, H. and Lee, S., (2009) Mining frequent itemsets over data streams using efficient window sliding techniques. *Expert Syst. Appl.*, **36**, 1466–1477.

[10] Han, J., Pei, J. and Yin, Y. (2000) Mining Frequent Patterns Without Candidate Generation. *Proc. SIGMOD Conf.*, Dallas, Texas, USA, pp. 1–12.

[11] Chang, J. and Lee, W. (2004) Decaying obsolete information in finding recent frequent itemsets over data stream. *IEICE Trans. Inf. Syst.*, **87**, 1588–1592.

[12] Li, H.-F., Lee, S.-Y. and Shan, M.-K. (2004) An Efficient Algorithm for Mining Frequent Itemsets over the Entire History of Data Streams. *Proc. Int. Workshop on Knowledge Discovery in Data Streams*, Pisa, Italy.

[13] Zhi-Jun, X., Hong, C. and Li, C. (2006) An Efficient Algorithm for Frequent Itemset Mining on Data Streams. *Proc. ICDM*, Leipzig, Germany, pp. 474–491.

[14] Leung, C.K.S. and Khan, Q.I. (2006) DSTree: A Tree Structure for the Mining of Frequent Sets from Data Streams. *Proc. IEEE Int. Conf. Data Mining*, Hong Kong, China, pp. 928–932.

[15] Tanbeer, S.K., Ahmed, C.F., Jeong, B. and Lee, Y. (2009) Sliding window-based frequent pattern mining over data streams. *Inf. Sci.*, **179**, 3843–3865.

[16] Lin, C., Chiu, D., Wu, Y. and Chen, A.L.P. (2005) Mining Frequent Itemsets from Data Streams with a Time-Sensitive Sliding Window. *Proc. SDM*, Newport Beach, California, USA.

[17] Mozafari, B., Thakkar, H. and Zaniolo, C. (2008) Verifying and Mining Frequent Patterns from Large Windows over Data Streams. *Proc. ICDE Int. Conf.*, Cancun, Mexico, pp. 179–188.

[18] Chang, J.H. and Lee, W.S. (2005) estWin: online data stream mining of recent frequent itemsets by sliding window method. *Inf. Sci.*, **31**, 76–90.

[19] Chi, Y., Wang, H., Yu, P.S. and Muntz, R.R. (2004) Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window. *Proc. ICDM*, Brighton, UK, pp. 59–66.

[20] Li, H., Ho, C., Kuo, F. and Lee, S. (2006) A New Algorithm for Maintaining Closed Frequent Itemsets in Data Streams by Incremental Updates. *Proc. ICDM Workshops*, Hong Kong, China, pp. 672–676.

[21] Li, H. and Chen, H. (2009) Mining non-derivable frequent itemsets over data stream. *Data Knowl. Eng*., **68**, 481–498.

[22] Jin, R. and Agrawal. G. (2005) An Algorithm for In-Core Frequent Itemset Mining on Streaming Data. *Proc. ICDM*, Houston, Texas, USA, pp. 210–217.

[23] Zaki, M.J., Parthasarathy, S., Ogihara, M. and Li, W. (1997) New Algorithms for Fast Discovery of Association Rules. *Proc. KDD*, Newport Beach, California, USA, pp. 283–286.