Available online at www.sciencedirect.com
ScienceDirect
Expert Systems with Applications 36 (2009) 2451–2458
Expert Systems with Applications
www.elsevier.com/locate/eswa

Incremental updates of closed frequent itemsets over continuous data streams

Hua-Fu Li [a,*], Chin-Chuan Ho [b], Suh-Yin Lee [b]

[a] *Department of Computer Science, Kainan University, Taiwan*
[b] *Department of Computer Science, National Chiao-Tung University, Taiwan*

Abstract

Online mining of closed frequent itemsets over streaming data is one of the most important issues in mining data streams. In this paper, we propose an efficient one-pass algorithm, NewMoment to maintain the set of closed frequent itemsets in data streams with a transaction-sensitive sliding window. An effective bit-sequence representation of items is used in the proposed algorithm to reduce the time and memory needed to slide the windows. Experiments show that the proposed algorithm not only attain highly accurate mining results, but also run significant faster and consume less memory than existing algorithm Moment for mining closed frequent itemsets over recent data streams.
© 2007 Elsevier Ltd. All rights reserved.

Keywords: Data mining; Data streams; Closed frequent itemsets; Single-pass mining; Incremental update

1. Introduction

Online mining of data streams is one of the most interesting research issues of data mining in recent years. Data streams have the unique characteristics as described below (Babcock, Babu, Datar, Motwani, & Widom, 2002; Golab & Özsu, 2003; Jiang & Gruenwald, 2006): (1) unbounded size of input data; (2) usage of main memory is limited; (3) input data can only be handled once; (4) fast arrival rate; (5) system cannot control the order data arrives; (6) analytical results generated by algorithms should be instantly available when users request; (7) errors of analytical results should be bounded in a range that users can tolerate.

Many previous studies contributed to the efficient mining of frequent patterns in streaming data (Chang & Lee, 2004a, 2004b; Chi, Wang, Yu, & Muntz, 2004; Giannella, Han, Pei, Yan, & Yu, 2003; Jin & Agrawal, 2005; Li, Lee, & Shan, 2004, 2005; Li, Ho, Shan, & Lee, 2006; Manku & Motwani, 2002; Teng, Chen, & Yu, 2003, 2004; Wong & Fu, 2005; Yu, Chong, Lu, & Zhou, 2004). According to the stream processing model (Zhu & Shasha, 2002), the research of mining frequent patterns over data streams can be divided into three categories: *landmark windows* (Jin and Agrawal, 2005, Li et al., 2004, 2005; Manku and Motwani, 2002; Yu et al., 2004), *sliding windows* (Chang & Lee, 2004b; Chi et al., 2004; Li et al., 2006; Teng et al., 2003, 2004; Wong & Fu, 2005), and *damped windows* (Chang & Lee, 2004a; Giannella et al., 2003), as described briefly as follows. In the landmark window model, knowledge discovery is performed based on the values between a specific timestamp, called *landmark*, and the present timestamp. In the sliding window model, knowledge discovery is performed over a fixed number of recently generated data elements which is the target of data mining. Two types of sliding widow, i.e., *transaction-sensitive sliding window* and *time-sensitive sliding window*, are used in mining data streams. The basic processing unit of window sliding of first type is an expired transaction while the basic unit of window sliding of second one is a time unit, such as a minute or an hour. In the damped window model, recent sliding windows are more important than previous ones.

* Corresponding author.
E-mail addresses: hfli@mail.knu.edu.tw, lihuafu@gmail.com (H.-F. Li), hocc@csie.nctu.edu.tw (C.-C. Ho), sylee@csie.nctu.edu.tw (S.-Y. Lee).

0957-4174/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2007.12.054

In Manku and Motwani (2002), Manku and Motwani developed two single-pass algorithms, sticky-sampling and lossy-counting, to mine frequent items over *offline* data streams with a landmark window. Moreover, Manku and Motwani proposed a lossy-counting based three module method BTS (Buffer-Trie-SetGen) for mining the set of frequent itemsets from *offline* data streams. Li et al. proposed prefix tree-based single-pass algorithms, called DSM-FI (Li et al., 2004) and DSM-MFI (Li et al., 2005), to mine the set of all frequent itemsets and maximal frequent itemsets over the entire history of *offline* data streams. Jin and Agrawal (2005) proposed an algorithm, called StreamMining, for in-core frequent itemset mining over *online* data streams. Yu et al. (2004) discussed the issues of false negative or false positive in mining frequent itemsets from high speed *offline* transactional data streams.

Chang and Lee (2004a) developed a damped window based algorithm, called estDec, for mining frequent itemsets in *online* streaming data in which each transaction has a weight decreasing with age. In other words, older transactions contribute less toward itemset frequencies, and it is a kind of damped windows model. Giannella et al. (2003) proposed a frequent pattern tree (abbreviated as FP-tree (Han, Pei, & Yin, 2000)) based algorithm, called FP-stream, to mine frequent itemsets at multiple time granularities by a novel titled-time windows technique. FP-stream focuses on *offline* data streams.

Chang and Lee (2004b) proposed a BTS-based algorithm, called SWFI-stream, for mining frequent itemsets in *online* data streams with a *transaction-sensitive* sliding windows model. Li et al. (2005) proposed a single-pass algorithm, called DSM-RMFI, based on DSM-MFI to find maximal frequent itemsets over *offline* data streams with a *time-sensitive* sliding window. Teng et al. (2003) proposed a regression-based algorithm, called FTP-DS, to find temporal patterns (frequent inter-transaction itemsets) across *multiple online* data streams in a time-sensitive sliding window. Teng et al. (2004) proposed a resource-aware algorithm, called RAM-DS, to mine temporal patterns in *multiple online* data streams with a time-sensitive sliding window. Li et al. (2006) proposed efficient algorithms, called MFI-TransSW and MFI-TimeSW, and to find the set of frequent itemsets in *online* data streams with a *transaction-sensitive* sliding window and *time-sensitive* sliding window, respectively. Wong and Fu (2005) proposed an efficient algorithm to mine *top-k frequent itemsets* in offline data streams with a *transaction-sensitive* sliding window without a user-defined minimum support constraint.

Chi et al. (2004) proposed a *transaction-sensitive* sliding window based algorithm, called Moment, which might be the first to find *frequent closed itemsets* (FCI) from *online* data streams with a transaction-sensitive sliding window. A summary data structure, called CET (closed enumeration tree), is used in the Moment algorithm to maintain a dynamically selected set of itemsets over a transaction-sensitive sliding window. These selected itemsets consist of closed frequent itemsets and a boundary between the closed frequent itemsets and the rest of the itemsets. CET can cover all necessary information because any status changes of itemsets (e.g. from infrequent to frequent or from frequent to infrequent) must be through the boundary in CET. Whenever a sliding occurs, it updates the counts of the related nodes in CET and modifies CET. Experiments of Moment show that the boundary in CET is stable so the update cost is little. However, Moment must maintain huge CET nodes for a closed frequent itemset. The ratio of CET nodes and closed frequent itemsets is about 30:1. If there are a large number of closed frequent itemsets, the memory usage of Moment will be inefficient.

The purpose of this work is on *closed frequent itemsets* mining over *online* data streams with a*transaction-sensitive* sliding window. An efficient algorithm, called NewMoment,[1] is proposed to mine the set of closed frequent itemsets over *online* data streams with a *transaction-sensitive* sliding window. Experiments show that the proposed NewMoment algorithm not only attain highly accurate mining results, but also run significant faster and consume less memory than Moment algorithm (Chi et al., 2004) for mining closed frequent itemsets over the most recent w transactions of a data stream.

The remainder of the paper is organized as follows. The problem is defined in Section 2. Section 3 presents the proposed NewMoment algorithm. Experiments are discussed in Section 4. Finally, we conclude this work in Section 5.

2. Problem definition

Let $\Psi = \{i_1, i_2, \ldots, i_m\}$ be a set of **items**. A **transaction** $T = (TID, x_1, x_2, \ldots, x_n)$, $x_i \in \Psi$, for $1 \leqslant i \leqslant n$, is a set of items, while n is called the **size** of the transaction, and TID is the unique identifier of the transaction. An **itemset** is a non-empty set of items. An itemset with size k is called a *k-itemset*. A **transaction data stream** $TDS = T_1, T_2, \ldots, T_N$ is a continuous sequence of transactions, where N is the TID of latest incoming transaction T_N.

A **transaction-sensitive window** (*TransSW*) in the transaction data stream is a window that slides forward for every transaction. The window at each slide has a fixed number, w, of transactions, and w is called the *size* of the window. Hence, the *current transaction-sensitive window* is $TransSW_{N-w+1} = [T_{N-w+1}, T_{N-w+2}, \ldots, T_N]$, where $N - w + 1$ is the window id of current *TransSW*. The **support** of an itemset X over *TransSW*, denoted as **sup**(X), is the number of transactions in *TransSW* containing X as a *subset*.

Definition 1 (*Frequent itemset*). An itemset X is called a **frequent itemset** (*FI*) if **sup**(X) $\geqslant s \cdot w$, where s is a user-defined minimum support threshold (MST) in the range of $[0, 1]$. The value $s \cdot w$ is called the **frequent threshold** (*FT*) of *TransSW*.

[1] A New algorithm for Maintaining Closed Frequent Itemsets by Incremental Updates.

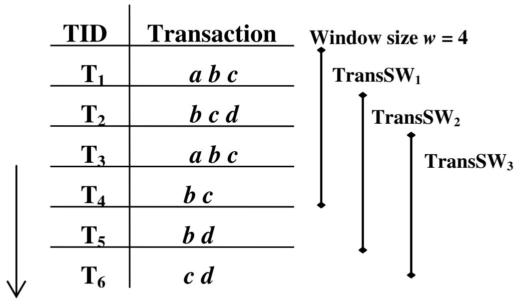

Fig. 1. Example Transaction-Sensitive Window.

Definition 2 (*Closed frequent itemset*). An itemset X is a **closed frequent itemset** if there exists no itemset X' such that (1) X' is a proper superset of X, and (2) every transaction containing X also contains X'.

Problem Statement: Given a transaction-sensitive window *TransSW*, and a minimum support threshold s, the problem is to mine the set of closed frequent itemsets in the most recent w transactions in a data stream.

Fig. 1 is an example transaction-sensitive window used in this paper. In Fig. 1, the size of the sliding window is 4. The first transaction-sensitive widow TransSW$_1$ consists of the transactions from T_1 to T_4. When the transaction with T_5 comes, the transaction-sensitive window eliminates the oldest transaction (T_1) from the current window and appends the incoming transaction (T_5). The second window TransSW$_2$ is the result after the first time of window sliding.

3. The proposed algorithm NewMoment

In this section, we introduce the proposed NewMoment algorithm. A bit vector based representation of items is used in the NewMoment algorithm to reduce the time and memory needed to slide the windows. A new summary data structure NewCET[2] based on a prefix tree structure is developed to maintain the essential information of closed frequent itemsets in the recent w transaction of a data stream.

3.1. Bit-vector representation of items

In the NewMoment algorithm, for each item X in the current *TransSW*, a *bit-sequence* with w bits, denoted as **Bit**(X), is constructed. If an item X is in the ith transaction of current *TransSW*, the ith bit of **Bit**(X) is set to be 1; otherwise, it is set to be 0.

For example, in Fig. 1, the first window $TransSW_1$ consists of four transactions: $\langle T_1,(abc)\rangle$, $\langle T_2,(bcd)\rangle$, $\langle T_3,(abc)\rangle$ and $\langle T_4,(bc)\rangle$, but the second window $TransSW_2$ consists of transactions: $\langle T_2,(bcd)\rangle$, $\langle T_3,(abc)\rangle$, $\langle T_4,(bc)\rangle$, and $\langle T_5,(bd)\rangle$. Because item a appears in the first and third transactions of $TransSW_1$, the bit-sequence of a, **Bit**(a), is 1010. Similarly, **Bit**(b) = 111, **Bit**(c) = 1111, and **Bit**(d) = 0100. The bit-sequences of all items in each window are listed in Table 1. The most left bit of a bit-sequence represents the oldest transaction in current window and the most right bit represents the newest transaction.

In the next section, we will introduce the methods to slide the transaction-sensitive windows using the bit-sequences of items.

3.2. Window sliding using bit-sequences

The bit-sequence is efficient in window sliding process. The sliding process consists of two steps: delete the oldest transaction and append the incoming transaction.

3.2.1. Delete the oldest transaction

In this step, the bit-sequences of items are used to left-shift one bit to delete the oldest transaction. For example, in Fig. 1, the bit sequence of item a, **Bit**(a), is 1010 in the first window TransSW$_1$. If transaction T_1 is deleted from TransSW$_1$, **Bit**(a) becomes 0100. Now the most left bit represents the transaction T_2. The most right bit is meaningless and is conserved for next step.

3.2.2. Append the incoming transaction

After deleting the oldest transaction from current transaction-sensitive window, we set the most right bit of each bit-sequence of items by checking the new incoming transaction T_N. We set the most right bit of the bit-sequence of item X to 1 if T_N contains X as a subset. Otherwise, we set the bit to 0.

For example, in Fig. 1, the bit-sequence of item a, **Bit**(a), becomes 0100 after deleting the expired transaction T_1. Because the incoming transaction T_5 does not contain item a, we set the most right bit of **Bit**(a) to 0, i.e., **Bit**(a) changes form 1010 to 0100. Similarly, **Bit**(c) changes from 1111 to 1110 and **Bit**(d) changes from 0100 to 1001.

In the next section, we introduce an efficient method to count the support of itemsets in the current transaction-sensitive window.

3.3. Counting support using bit-sequences

The concept of bit-sequence of item can be extended to itemset. For example, in Fig. 1, the bit-sequence of 2-itemset ab, **Bit**(ab), in the TransSW$_1$ is 1010. That means transactions T_1 and T_3 of TransSW$_1$ contain the itemset ab.

Table 1
Bit-sequences of items in each window

	TransSW$_1$	TransSW$_2$	TransSW$_3$
a	1010	0100	1000
b	1111	1111	1110
c	1111	1110	1101
d	0100	1001	0011

[2] New Closed Enumeration Tree.

The process of counting support of an itemset is described as follows. Assume that there are two k-itemsets X and Y and their corresponding bit-sequences **Bit**(X) and **Bit**(Y). The bit-sequence of the $(k+1)$-itemset $Z = X \cup Y$ can be obtained by the bitwise AND of **Bit**(X) and **Bit**(Y). For example, the bit-sequence of 2-itemset *ab*, **Bit**(*ab*), in the first window TransSW_1 is 1010 which can be obtained by bitwise AND the bit-sequences of items *a* and *b*, where **Bit**(a) = 1010 and **Bit**(b) = 1111.

In the next section, we propose an efficient approach to build the proposed summary data structure NewCET using bit-sequences of itemsets. Based on the bitwise AND of bit-sequences of itemsets, candidates can be efficiently generated when building NewCET.

3.4. Building the NewCET

The proposed summary data structure, called **NewCET** (New Closed Enumeration Tree), is an extended prefix tree structure. NewCET consists of three parts.

(1) *The bit-sequences of all 1-itemsets in the current transaction-sensitive window* $\mathit{TransSW}_{NN-w+1}$.
(2) *A set of closed frequent itemsets in* $\mathit{TransSW}_{NN-w+1}$.
(3) *A hash table*: For checking whether a frequent itemset is closed or not, we use a hash table to store all closed frequent itemsets with their supports as keys. Assume that there are two frequent itemsets X and Y. If the support of X is equal to the support of Y and $X \subseteq Y$, X and Y must be contained in the same set of transactions. That means the itemset X is not a closed frequent itemset. Moreover, the value of support is suitable to be the key of the hash table.

Similar to a prefix tree, each node n_I in the NewCET represents an itemset I. A child node, n_J, is obtained by adding a new item to I. But, NewCET only maintains a set of closed frequent itemsets, not all itemsets.

Fig. 2 gives the algorithm of building NewCET. In the building algorithm, each n_I has a corresponding bit-sequence, **Bit**(I), to store the support information in the current sliding window. Function *Build* is a depth-first procedure. *Build* visits the itemsets of the current NewCET in a lexicographical order. In the lines 1–2 of Fig. 2, function *Build* is performed if n_I is frequent and is not contained by other closed frequent itemsets. Function *leftcheck* uses the support of n_I as a hash key to speed up the checking. In the lines 3–5, if n_I passes the checking of the lines 1–2, *Build* generates all possible children of n_I with frequent siblings and creates their bit-sequences by bitwise AND bit-sequences of n_I and its frequent siblings. In the lines 6–7, *Build* recursively calls itself to check each child of n_I. In the lines 8–10, if there is no child of n_I with the same support as n_I, n_I is a closed frequent itemset and it is retained in the NewCET.

Fig. 3 shows the NewCET in the first window TransSW_1 when the function *Build* is in process. Although a bit-sequence is generated and a new tree node is created, only a branch of the tree is maintained in the main memory. This is because *Build* is a depth-first procedure. Besides the set of 1-itemsets, the maximum number of bit-sequences in the memory is 3, i.e., bit-sequences of the itemsets *ab*, *ac*, and *abc*. When the function *Build* is done, all bit-sequences of k-itemsets eliminated, where $k > 1$. The set of all closed frequent itemsets in the current transaction-sensitive window only retains their supports.

Fig. 4 shows the NewCET in the first transaction-sensitive window TransSW_1 when *Build* is done. The tree nodes

```
Build (n_I, N, s)
1:   if support(n_I) ≥ s · N then
2:       if leftcheck(n_I) = false then
3:           foreach frequent sibling n_K of n_I do
4:               generate a new child n_{I□K} for n_I;
5:               bitwise AND Bit(I ) and Bit(K ) to
                   obtain Bit(I □K);
6:           foreach child n_I′ of n_I do
7:               Build(n_I′, N, s);
8:           if a child n_I′ of n_I such that
                   support(n_I′) = support(n_I) then
9:               retain n_I as a closed frequent itemset;
10:              insert n_I into the hash table;
```

Fig. 2. Algorithm of building NewCET.

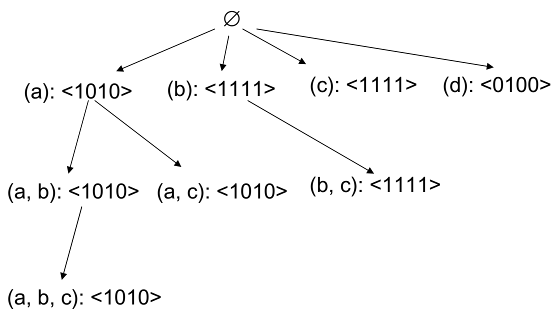

Fig. 3. NewCET in the first window TransSW_1.

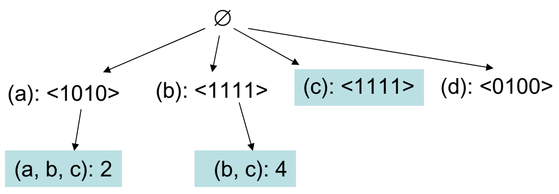

Fig. 4. NewCET in the first window TransSW_1 (tree nodes with shadow are closed frequent itemsets).

with shadow are closed frequent itemsets. For simplicity, the hash table is not displayed in this figure.

In Sections 3.5 and 3.6, we describe the methods to delete the oldest transaction and append a new incoming transaction in the current transaction-sensitive window, respectively.

3.5. Deleting the oldest transaction

Deleting the oldest transaction is the first step of window sliding. First of all, all bit-sequences of 1-itemsets are left-shifted one bit. Then, all items in the deleted transaction are kept. The process can be done by observing the most left bit before the left-shifting.

Fig. 5 gives the algorithm of deleting the oldest transaction after left-shifting all the bit-sequences of 1-itemsets. In the Fig. 5, the function *Delete* generates the prefix tree including the itemsets whose supports are $s \cdot N - 1$. This is because the supports of a set of closed frequent itemsets in previous window would be $s \cdot N$ and then becomes $s \cdot N - 1$ after the deletion.

Function *Delete* is a depth-first procedure. When the recursive calls of n_I's children return, *Delete* is performed, if n_I is a closed frequent itemset, the NewCET is maintained and the hash table is updated. In the lines 19 and 23, if n_I is closed frequent itemset in previous window, n_I is marked as a non-closed itemset. In this case, n_I will not be retained when the function *Delete* is done. Fig. 6 shows the NewCET after deleting the oldest transaction T_1.

3.6. Appending a new incoming transaction

Appending the incoming transaction is the second step of window sliding. All the bit-sequences of 1-itemsets are set their most right bit to 1 or 0 based on the incoming transaction T_N. We set the most right bit of the bit-sequence of itemset X to 1 if T_N contains X as a subset. Otherwise, we set the bit to 0.

Fig. 7 gives the algorithm of appending a new incoming transaction after setting the most right bit of each bit-sequence of 1-itemsets. Function *Append* is almost the same as *Build*. The only difference is in the lines 9–11. If the checked closed frequent itemsets are already in the NewCET, *Append* updates the NewCET and hash table. Fig. 8 shows the NewCET in the second window TransSW$_2$ after appending the incoming transaction T_5.

4. Performance evaluation of NewMoment

In this section, the experiments are performed to compare the proposed NewMoment algorithm with the Moment algorithm (Chi et al., 2004). The source code of Moment algorithm, denoted as MomentFP, is provided by Chi et al. (2004). All experiments are done on a 1.3 GHz Intel Celeron PC with 512 MB memory and running with Windows XP system. The proposed NewMoment algorithm is implemented in C++ STL and compiled with

```
Delete ($n_I$, $N$, $s$)

1:  if $n_I$ is not relevant to the deleted transaction then
2:      return;
3:  else if support($n_I$) ≥ ($s$·$N$ – 1) then
4:      foreach sliding $n_K$ of $n_I$ whose support ≥ ($s$·$N$ – 1)
          do
5:        generate a new child $n_{I□K}$ for $n_I$;
6:        bitwise AND Bit($I$) and Bit($K$) to obtain Bit($I$□$K$);
7:      foreach child $n_I'$ of $n_I$ do
8:          Delete($n_I'$, $N$, $s$);
9:      if support($n_I$) ≥ $s \cdot N$ then
10:         if leftcheck($n_I$) = false then
11:             if $n_I$ is closed frequent itemset
                      in previous sliding window then
12:                 update the support of $n_I$;
13:                 update $n_I$ in the hash table;
14:             else
15:                 retain $n_I$ as a closed frequent itemset;
16:                 insert $n_I$ into the hash table;
17:         else   /* leftcheck($n_I$) = true */
18:             if $n_I$ is closed frequent itemset
                      in previous sliding window then
19:                 mark $n_I$ as a non-closed frequent itemset;
20:                 eliminate $n_I$ from the hash table;
21:     else   / * support($n_I$) < $s$·$N$   */
22:         if $n_I$ is closed frequent itemset
                  in previous sliding window then
23:             mark $n_I$ as non-closed itemset;
24:             eliminate $n_I$ from the hash table;
```

Fig. 5. Algorithm of deleting the oldest transaction.

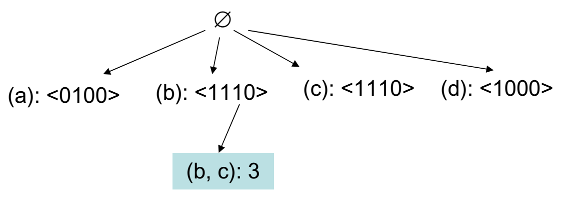

Fig. 6. NewCET after deleting the transaction T_1.

```
Append ($n_I$, $N$, $s$)
1:  if support($n_I$) ≥ $s \cdot N$ then
2:      if leftcheck($n_I$) = false then
3:          foreach frequent sibling $n_K$ of $n_I$ do
4:              generate a new child $n_{I\square K}$ for $n_I$;
5:              bitwise AND Bit($I$) and Bit($K$) to
                  obtain Bit($I\square K$);
6:          foreach child $n_I'$ of $n_I$ do
7:              Append($n_I'$, $N$, $s$);
8:          if a child $n_I'$ of $n_I$ such that
                  support($n_I'$) = support($n_I$) then
9:              if $n_I$ is closed frequent itemset
                      in previous sliding window then
10:                 update the support of $n_I$;
11:                 update $n_I$ in the hash table;
12:         else
13:                 retain $n_I$ as a closed frequent itemset;
14:                 insert $n_I$ into the hash table;
```

Fig. 7. Algorithm of appending the incoming transaction.

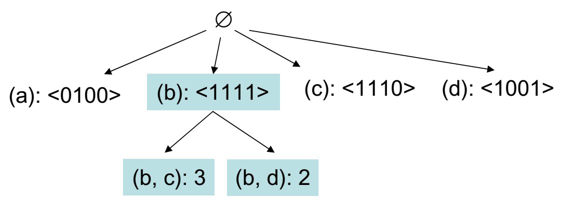

Fig. 8. NewCET after appending a new incoming transaction T_5 in the TransSW$_2$.

Visual C++ .NET compiler. Moreover, the synthetic data T10.I10.D200K is generated by the IBM synthetic data generator (Agrawal & Srikant, 1994). Parameters of synthetic data are listed in Table 2.

The performance measurements include memory usage, the loading time of the first window, and the average time of window sliding. Memory usage was tested by system tool to observe real memory variation. Average time of window sliding was reported over 100 consecutive sliding windows.

Table 2
Parameters of the synthetic data

Parameter	Value
Average items per transaction (T)	10
Number of transactions (D)	200 K
Number of items (N)	1000
Average length of maximal pattern (I)	10

4.1. Mining with different minimum supports

In the first experiment, the minimum support threshold is changed from 1% to 0.1%, and the size of sliding window is fixed to 100,000 (100 K) transactions.

Fig. 9 shows the memory usage with KB units. We can observe that memory used by Moment is more than 120 MB but used by NewMoment is about 15 MB. When the minimum support is down to 0.05%, the memory used by NewMoment is just 50 MB but memory of Moment is out of bound (more than 512 MB).

The maintaining data of NewMoment is much less than the one of Moment. NewMoment only maintains bit-sequences of 1-itemsets and closed frequent itemsets in current window. Experiment shows that NewCET is more compact than CET.

Fig. 10 shows the loading time the first window. In the first window, both NewMoment and Moment need to build a prefix (lexicographic) tree. We can observe that NewMoment is a little faster than Moment. The reason is that generating candidates and counting their supports with bit-sequences is more efficient than with an independent sliding window (in MomentFP, a FP-tree (Chi et al., 2004) is used).

Fig. 11 shows the average time of window sliding. In the experiment, NewMoment is a little slower than Moment because NewMoment do not use TID sum as another key to speed up left-check step. But we can observe that

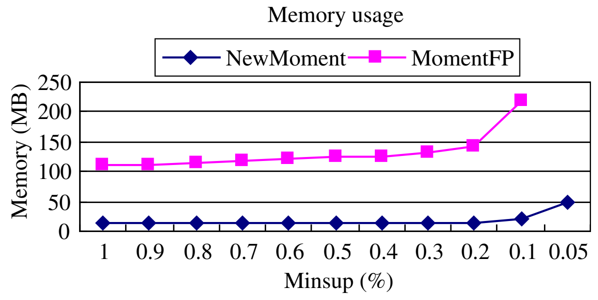

Fig. 9. Memory usage with different minimum supports.

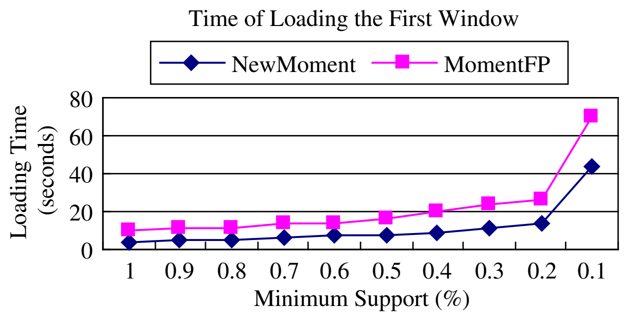

Fig. 10. Time of loading the first window with different minimum supports.

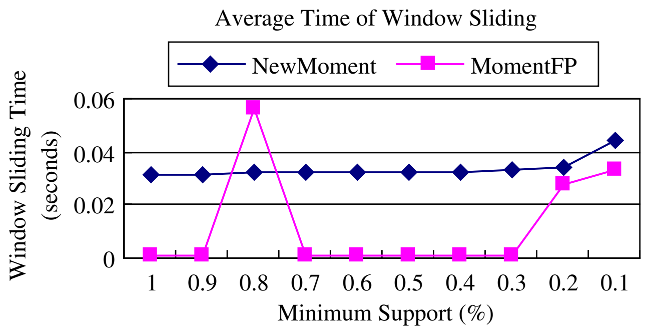

Fig. 11. Average time of window sliding with different minimum supports.

the difference is about 0.02 s. The steps of window sliding can be finished in one second for both algorithms and the difference is meaningless.

4.2. Mining with different window sizes

In the second experiment, the size of sliding window is changed from 10 K transactions to 100 K transactions, and the minimum support threshold *s* is fixed to 0.1%.

Fig. 12 shows the memory usage with KB units. Both NewMoment and Moment are linearly affected by the sizes of sliding windows. In this experiment, the proposed NewMoment algorithm outperforms the Moment algorithm in the memory requirement.

Fig. 13 shows the time of loading the first window. Although with the increasing sliding window size, each bit-sequence becomes larger, NewMoment is still faster than Moment in the experiment of loading time of the first window. The reason is that the processing time of bitwise AND operation between bit-sequences is almost not effected by the length of bit-sequence.

Fig. 14 shows the average time of window sliding. In this experiment, the time of window sliding of NewMoment and Moment is almost the same.

4.3. Mining with different number of items

NewMoment algorithm maintains bit-sequences of all items instead of independent sliding window structure maintained in MomentFP algorithm (Chi et al., 2004). In this section, several experiments are done to prove that with the increase of item types, NewMoment is still efficient in memory usage and running time. But, MomentFP is out of memory (more than 512 MB) when the number of items exceeds 3000.

In these experiments, the number of items is changed from 1000 to 10,000. The size of sliding window is set to 100,000 and minimum support threshold is set to 0.1%. Fig. 15 shows the memory usage with KB units. The memory usage of NewMoment and the number of items is linearly related. This result shows that NewMoment does not increase its memory usage suddenly when the number of items is large.

Fig. 16 shows the loading time the first window and Fig. 17 shows average time of window sliding. The results show that loading time and window sliding time also has linear relation with the number of items. Although loading time is more than 300 s when the number of items exceeds 9000, the process of loading the first window is only executed once. Average time of window sliding is still less than

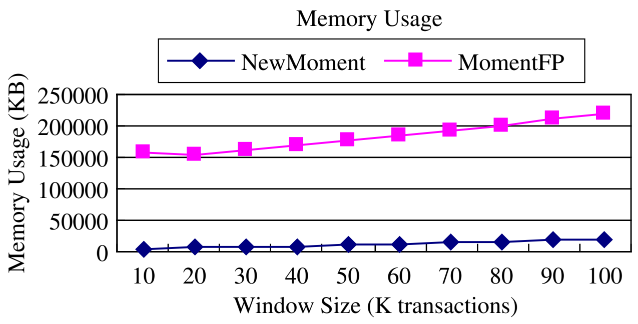

Fig. 12. Memory usage with different window sizes.

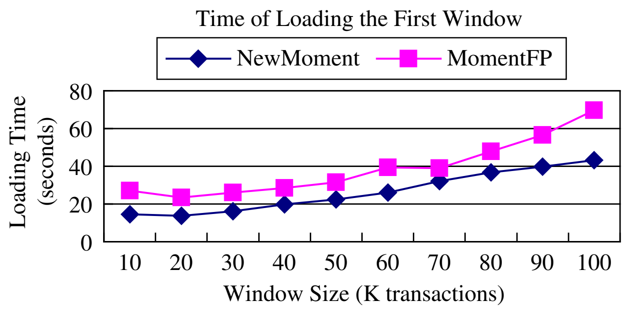

Fig. 13. Time of loading the first window with different sliding window sizes.

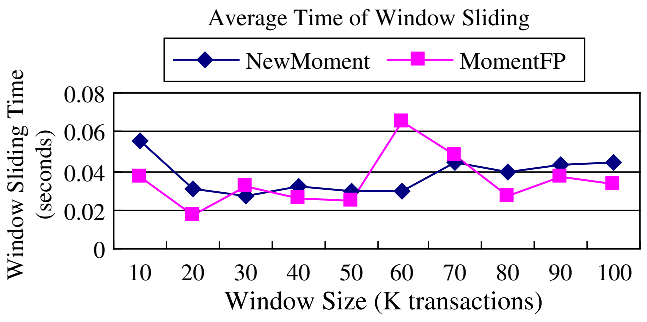

Fig. 14. Average time of window sliding with different sliding window sizes.

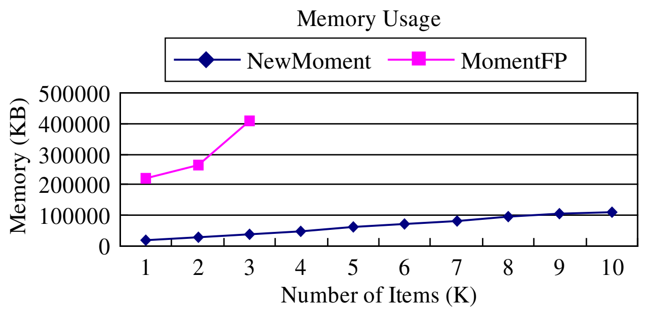

Fig. 15. Memory usage with different number of items.

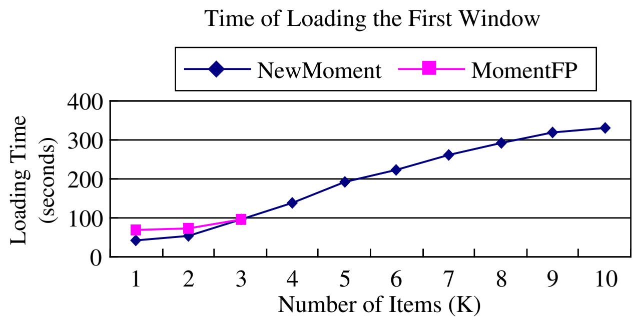

Fig. 16. Time of loading the first window with different number of items.

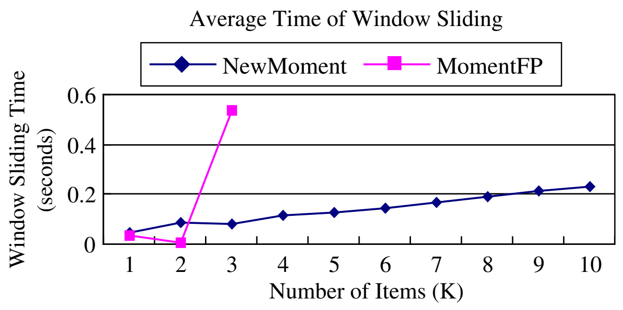

Fig. 17. Average time of window sliding with different number of items.

one second. It means that the proposed NewMoment algorithm is still efficient with a large number of items.

5. Conclusions

In this paper, we propose an efficient single-pass algorithm NewMoment to mine the set of closed frequent itemsets over data streams with a transaction-sensitive sliding window. In NewMoment algorithm, an effective bit-sequence representation is developed to reduce the memory requirement of the online maintenance of closed frequent itemsets generated so far. Experiments show that the proposed NewMoment algorithm outperforms the Moment, a state-of-art algorithm for mining the set of closed frequent itemsets over online data streams with a transaction-sensitive sliding window.

Acknowledgement

This paper was partially supported by the National Science Council of Taiwan, R.O.C. under research grant No. 95-2221-E-009-069-MY3 and No. NSC 96-2218-E-424-001.

References

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In *Proceedings of the 20th international conference on very large data bases*, (pp. 487–499).

Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (2002). Models and issues in data stream systems. In *Proceedings of the 21th ACM SIGMOD-SIGACT-AIGART symposium on principles of database systems*, (pp. 1–16).

Chang, J., & Lee, W. (2004a). Decaying obsolete information in finding recent frequent itemsets over data stream. *IEICE Transaction on Information and Systems, E87-D*(6).

Chang, J., & Lee, W. (2004b). A sliding window method for finding recently frequent itemsets over online data streams. *Journal of Information Science and Engineering, 20*(4).

Chi, Y., Wang, H., Yu, P., & Muntz, R. (2004). MOMENT: Maintaining closed frequent itemsets over a stream sliding window. In *Proceedings of the 4th IEEE international conference on data mining*, (pp. 59–66).

Giannella, C., Han, J., Pei, J., Yan, X., & Yu, P. S. (2003). Mining frequent patterns in data streams at multiple time granularities. In H. Kargupta, A. Joshi, K. Sivakumar, & Y. Yesha (Eds.), *Data mining: Next generation challenges and future directions*. AAAI/MIT.

Golab, L., & Özsu, M. T. (2003). Issues in data stream management. *ACM SIGMOD Record, 32*(2), 5–14.

Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation. In *Proceedings of the 2000 international conference on management of data*, (pp. 1–12).

Jiang, N., & Gruenwald, L. (2006). Research issues in data stream association rule mining. *ACM SIGMOD Record, 35*(1).

Jin, R., & Agrawal, G. (2005). An algorithm for in-core frequent itemset mining on streaming data. In *Proceedings of the 5th IEEE international conference on data mining*.

Li, H.-F., Lee, S.-Y., & Shan, M.-K. (2004). An efficient algorithm for mining frequent itemsets over the entire history of data streams. In *Proceedings of the first international workshop on knowledge discovery in data streams*.

Li, H.-F., Lee, S.-Y., & Shan, M.-K. (2005). Online mining (recently) maximal frequent itemsets over data streams. In *Proceedings of the 15th IEEE international workshop on research issues on data engineering*, (pp. 11–18).

Li, H.-F., Ho, C.-C., Shan, M.-K., & Lee, S.-Y. (2006). Efficient maintenance and mining of frequent itemsets over online data streams with a sliding window. In *Proceedings of the 2006 IEEE international conference on systems, man and cybernetics*.

Manku, G.S., & Motwani, R. (2002). Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on very large data bases*, (pp. 346–357).

Teng, W.-G., Chen, M.-S., & Yu, P.S. (2003). A regression-based temporal pattern mining scheme for data streams. In *Proceedings of the 29th international conference on very large data bases*, (pp. 93–104).

Teng, W.-G., Chen, M.-S., & Yu, P.S. (2004). Using wavelet-based resource-aware mining to explore temporal and support count granularities in data streams. In *Proceedings of the 4th SIAM international conference on data mining, April 22–24*.

Wong, R.C.W., & Fu, A. (2005). Mining top-k itemsets over a sliding window based on Zipfian distribution. In *Proceedings of 2005 SIAM international conference on data mining*.

Yu, J.-X., Chong, Z., Lu, H., & Zhou, A. (2004). False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *Proceedings of the 30th international conference on very large data bases*, (pp. 204–215).

Zhu, Y., & Shasha, D. (2002). StatStream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on very large data bases*, (pp. 358–369).