Approximately Mining Recently Representative Patterns on Data Streams*

Jia-Ling Koh and Yuan-Bin Don

Department of Information Science and Computer Engineering
National Taiwan Normal University
Taipei, Taiwan
Email: jlkoh@ntnu.edu.tw

Abstract. Catching the recent trend of data is an important issue when mining frequent itemsets from data streams. To prevent from storing the whole transaction data within the sliding window, the frequency changing point (FCP) method was proposed for monitoring the recent occurrences of itemsets in a data stream under the assumption that exact one transaction arrives at each time point. In this paper, the FCP method is extended for maintaining recent patterns in a data stream where a block of various numbers of transactions (including zero or more transactions) is inputted within each time unit. Moreover, to avoid generating redundant information in the mining results, the recently representative patterns are discovered from the maintained structure approximately. The experimental results show that our approach reduces the run-time memory usage significantly. Moreover, the proposed GFCP algorithm achieves high accuracy of mining results and guarantees no false dismissal occurring.

1. Introduction

The strategies for mining frequent itemsets in static databases have been widely studied over the last decade such as the Apriori[1], DHP[9], and FP-growth[4]. In addition to memory requirement and execution efficiency, the sheer size of mining results is a major challenge in frequent-pattern mining. In many cases, a high minimum support threshold may discover only commonsense patterns but a low one may generate an explosive number of output patterns, which severely restricts its usage. To solve this problem, the closed frequent itemsets [10] provided a lossless compression of the whole collection of patterns; however, its compression power is limited. For providing a general method for high-quality compression, the distance measure between two frequent itemsets was defined in [12] to find a representative pattern for each cluster of patterns. Two greedy algorithms, named RPglobal and RPlocal, respectively, were proposed. The RPglobal algorithm applied the greedy method to find representative patterns among the discovered frequent itemsets. For providing a scalable method, the RPlocal algorithm found the representative patterns locally during the process of pattern-growth. The experimental results showed that the RPlocal method, whose compression quality is very close to RP-global, is far more efficient than RP-global.

Recently, the data stream, which is an unbounded sequence of data elements generated at a rapid rate, provides a dynamic environment for collecting data sources. Since it is not feasible to store the past data in data streams completely, a method for

This work was partially supported by the R.O.C. N.S.C. under Contract No. 95-2221-E-003-011 and 95-2524-S-003-012.

providing the approximate answers with accuracy guarantees is required. The hash-based approach was proposed in [5], in which each item in a data stream owns a respective list of counters in a hash table, and each counter may be shared by many items. A new novel algorithm, called hCount, was provided to maintain frequent items over a data stream and support both insertion and deletion of items with a less memory space. Lossy-counting is the representative approach for mining frequent itemsets from data streams [8]. Given an error tolerance parameter ε, the Lossy-counting algorithm prunes the patterns with support being less than ε from the pool of monitored patterns such that the required memory usage is reduced. Consequently, the frequency of a pattern is estimated by compensating the maximum number of times that the pattern could have occurred before being inserted into the monitored patterns. It is proved no false dismissal occurs with Lossy-counting algorithm and the frequency error is guaranteed not to exceed a given error tolerance parameter.

In addition to the restriction of memory usage considered in the two works introduced previously, the time sensitivity issue is another important issue when mining frequent itemsets from data streams. It is likely that the embedded knowledge in a data stream will change quickly as time goes by. In order to catch the recent trend of data, the *estDec* algorithm [2] decayed the old occurrences of each itemset as time goes by to diminish the effect of old transactions on the mining result of frequent itemsets in the data steam. However, in certain applications, it is interested only the frequent patterns mined from the recently arriving data within a fixed time period. Under the assumption that exact one transaction arrives at each time unit, the sliding window method [3] defined the current sliding window to consist of the most recently coming w transactions in a data stream according to a given window size w. Consequently, the recently frequent itemsets were defined to be the frequent itemsets mined from the current sliding window. In addition to maintain the occurrence for the new transaction, the oldest transaction has to be removed from the maintained data structure when the window is sliding. However, all the transactions in the current sliding window need to be maintained in order to remove their effects on the current mining result when they are beyond the scope in the window.

To prevent from storing the whole transaction data within the sliding window, we proposed the frequency changing point (FCP) method for monitoring the recent occurrence of itemsets in a data stream [6]. The effect of old transactions on the mining result of recently frequent itemsets is diminished by performing adjusting rules on the monitoring data structure. The experimental results showed that our approach reduces the run-time memory usage significantly by comparing with one of [3]. Moreover, the proposed FCP algorithm achieves high accuracy of mining results and guarantees no false dismissal occurring.

In [7], a time-sensitive sliding window approach was also proposed for mining the recently frequent itemsets within the current sliding window in a data stream. However, a general assumption that a block of various numbers of transactions (zero or more transactions) is inputted into the data stream at each time unit was adopted. Accordingly, the recently frequent itemsets were discovered from the most recent w blocks of transactions. For each block of transactions, the frequent itemsets in the block were found and all possible frequent itemsets in the sliding window were collected in a PFP (Potential Frequent-itemset Pool) table. For each newly inserted pattern, the maximum number of possible lost counts was estimated. Moreover, a

discounting table was constructed to provide approximate counts of the expired data items. However, as the minimum support threshold is reduced, the number of frequent itemsets in a basic block will increase dramatically. Because of the increasing cost of table maintenance, the memory usage of PFP table will increase such that the execution efficiency of the algorithm goes down.

In this paper, the FCP algorithm proposed in [6] is extended for solving the same problem considered in [7]. The provided data structures and adjusting rules in [6] are modified accordingly. Moreover, to avoid generating redundant information in the mining results, the idea of local greedy method is applied to discover recently representative itemsets from the maintained structure. The experimental results show that our approach reduces the run-time memory usage significantly than the one of STW algorithm [7] when the minimum support threshold is low. Moreover, the proposed GFCP algorithm achieves high accuracy of mining results and guarantees no false dismissal occurring.

This paper is organized as follows. The related terms used in this paper are defined in Section 2 first. The provided data structure for monitoring recent patterns in a data stream is shown in Section 3. In Section 4, the proposed algorithm for discovering recently representative patterns from the maintained structure is introduced. The performance evaluation on the proposed algorithms and a related work is reported in Section 5. Finally, in Section 6, we conclude this paper.

2. Preliminaries

Let $I = \{i_1, i_2, \ldots, i_m\}$ denote the set of items in the specific application domain and a transaction is composed of a set of items in I. A ***basic block*** is a set of transactions arriving within a fixed unit of time. A data stream, $DS = [B_1, B_2, \ldots, B_t)$, is an infinite sequence of basic blocks, where each basic block $B_i = \{T_{i1}, T_{i2}, \ldots, T_{ij}\}$ is associated with an time identifier i and t denotes the time identifier of the latest basic block arriving currently. Let $DS[i,j]$ denote the multi-set of transactions collected from basic blocks in DS from time i to j. Under a predefined window size w, the ***current transaction window*** at time t, denoted as CTW_t, corresponds to $DS[t\text{-}w+1, t]$. The time identifier of the first basic block in CTW_t is denoted as CTW_t^{first}, that is $t\text{-}w+1$.

An itemset (or a pattern) is a set consisting of one or more items in I, that is, a non-empty subset of I. If itemset e is a subset of transaction T, we call T ***contains*** e. The number of transactions in CTW_t which contain e is named the ***recent support count*** of e in DS, denoted as $RC_t(e)$. The ***recent support*** of e, denotes as $Rsup_t(e)$, is obtained from $RC_t(e) / |CTW_t|$.

Given a user specified minimum support threshold between 0 and 1, denoted as S_{min}, an itemset e is called a ***recently frequent*** itemset in DS if $Rsup_t(e) \geq S_{min}$. Otherwise, e is a ***recently infrequent*** itemset in DS.

For any two patterns p_r and p, let $T(p)$ and $T(p_r)$ denote the sets of transactions which contain p and p_r, respectively. By extending the definition of pattern distances proposed in [12], if $p_r \supseteq p$, the distance between p and p_r in CTW_t is defined to be:

$$RD(p, p_r) = 1 - \frac{|T(p) \cap T(p_r)|}{|T(p) \cup T(p_r)|} = 1 - \frac{|T(p_r)|}{|T(p)|} = 1 - \frac{RC_t(p_r)}{RC_t(p)} = 1 - \frac{Rsup_t(p_r)}{Rsup_t(p)}.$$

Otherwise, $RD(p, p_r) = \infty$. If $RD(p, p_r)$ is less than a givenδvalue, it is called that p_r δ-

covers p and *p* is *δ-covered by* p_r in CTW_t. A recently frequent pattern *p* is named a ***recently representative pattern*** if *p* is not *δ*-covered by any recently frequent itemset.

3. Pattern Summarization Method

3.1 Frequency Changing Points

A most compact method to represent the occurrences of a pattern *p* is to keep the first appearing time, denoted as $p.t_s$, and an accumulated count of the pattern, denoted as *p.f*. Accordingly, $Rsup_t(p)$ is obtained from $p.f / |DS[CTW_i^{first}, t]|$ if $p.t_s$ is within current transaction window. Otherwise, $Rsup_t(p)$ is estimated by $p.f / |DS[p.t_s, t]|$. However, if $p.t_s$ is far earlier than CTW_i^{first}, it is not sure whether the estimated support stands for the recent trend of the pattern *p* accurately. According to the discussion proposed in [10], it is a critical point to set a new accumulated count of pattern *p* when the appearing frequency of *p* becomes infrequent.

Let t_1 denote the first time identifier when a pattern *p* appeared. The earliest time identifier *t'*, where $t' > t_1$, *p* appears in $B_{t'}$, and (the support count of *p* in $DS[t_1, t'\text{-}1]$) / $|DS[t_1, t'\text{-}1]| < S_{min}$, is named a ***frequency changing point*(*FCP*)** of *p*. If a frequency changing point of *p* has ever occurred, t_1 is set to be the latest frequency changing point of *p*. The frequency changing points of a pattern *p* are going to be used to reset the boundaries of intervals for accumulating the support counts of *p*.

3.2 Pattern Summarization Tree

For a pattern *p*, an 8-tuple (p, t_s, f, f_c, t_e, t_{pre}, C_d, *Rqueue*) summarization record as described as the following is used to represent the summarization of *p*'s occurrences.

(1) *p:* the corresponding itemset;
(2) t_s: the starting point of support count accumulation for *p*; initially, it is set to be the identifier of the basic block when *p* was inserted into the data structure;
(3) *f*: the accumulated support count of *p* in $DS[t_s, t]$;
(4) f_c: the support count of *p* in B_t;
(5) t_e: the time identifier of the latest basic block which contains *p*;
(6) t_{pre}: the latest time identifier which is a frequency changing point of *p*;
(7) C_d: the accumulated support count of *p* in $DS[t_s, t_{pre}\text{-}1]$;
(8) *Rqueue*: a queue consists of a sequence of (ct_1, ac_1), (ct_2, ac_2), …, (ct_n, ac_n) pairs, in which the time identifier ct_i is a frequency changing point of *p* for i=1,…,n. Besides, ac_1 denotes the support count of *p* in $DS[t_s, ct_1\text{-}1]$ and ac_i denotes the support count of *p* in $DS[ct_{i-1}, ct_i\text{-}1]$ for i=2,…,n.

A FP-tree-like structure, named pattern summarization tree (PS-tree), is adopted to organize the summarization records of itemsets which appear in current transaction window. However, to prevent from two scans over the data set, the items in a transaction are sorted according to their alphanumeric order instead of their frequency-descending order.

Moreover, an array named TranArray is constructed to maintain the number of transactions in each basic block for the recent blocks, which is used for computing the recent supports of itemsets.

3.3 Maintenance of PS-tree

On the whole, there are four sub-tasks for maintaining the PS-tree structure: (1) insert transactions in the new basic block into PS-tree, (2) remove the expired information of patterns, (3) check and record frequency changing points of patterns, and (4) update the TranArray. The pseudo codes of the whole process are shown in Figure.1.

```
Program PS-tree_maintenance
Input: DataStream DS, Window_size w
1. TranArray and PS_tree are initialized;
2. While (B_t is inputted) {
3.     New_block_insertion(B_t) ;
4.     For each node N in the PS_tree ;
5.       { Let p denote the associated itemset in N ;
6.         If (t > w)
7.           If (p.t_e < CTW_t^first) Remove_itemset(PS_tree) ;
8.           Else If (p.t_s < CTW_t^first) Reset_start_time(PS_tree) ;
9.         If (p.f_s ≠ t and p.f_c ≠ 0 )
10.          Check_change_point(PS_tree) ;}
11.    TranArray_update(TranArray) ;
12.    t = t+1 ; }
```

Figure 1: the pseudo codes of PS-tree maintenance

First, each transaction T in the newly inputted basic block is inserted into the PS-tree. The PS-tree is a trie structure, so there is one node for every common prefix of patterns. For each prefix-subset p of T, if the corresponding node exists in the PS-tree, $p.f$, $p.f_c$, and $p.t_e$ in the node are updated accordingly. Otherwise, a new node for storing the summarization record of p with initial setting is constructed

In the window sliding phase, for each monitored pattern p in the PS-tree, it is necessary to update the summarization information of p within current transaction window. It is indicated that a pattern p does not appear in CTW_t if $p.t_e$ is less than CTW_t^{first}. Therefore, such a pattern is pruned to prevent from storing the unnecessary patterns in the monitoring data structure.

Moreover, it is necessary to adjust the starting point of support count accumulation for the monitored pattern p if $p.t_s$ is less than CTW_t^{first}. If $p.Rqueue$ is empty for such a pattern p, it implies p remains a frequent itemset during the accumulation interval. Therefore, no false dismissal occurs when discovering recently frequent patterns according to the support of p in $DS[p.t_s, t]$ such that it is not necessary to adjust $p.t_s$. On the other hand, it implies there is one or more frequency changing points of p occurring if $p.Rqueue$ is not empty. Therefore, the frequency changing points of p are checked one by one to adjust $p.t_s$ to be a frequency changing point of p as approaching CTW_t^{first} as possible. Let (ct, ac) denoted the first frequency changing point pair got from $p.Rqueue$. It is applicable to adjust $p.t_s$ in the following three cases:

(1) $ct \le CTW_t^{first}$: it implies the support count accumulated in ac is beyond the scope of CTW_t.

(2) $ct > CTW_t^{first}$ and $ac=1$: it implies the occurring of p before ct is at $p.t_s$ only. Thus,

the actual starting point of p within CTW_t is set to be ct.

(3) $ct>CTW_t^{first}$, $ac>1$, and the previous time point when p appears before ct, denoted as t_e', is less than CTW_t^{first}: let X denote the support count of p in $B_{te'}$. The maximum value of t_e', denoted by $Max\_\ t_e$', must satisfy $(ac\text{-}X)/|DS[t_s, Max\_\ t_e'\text{-}1] \geq S_{min}$ and $(ac\text{-}X)/|DS[t_s, Max\_\ t_e'] < S_{min}$ because there is not any changing point occurring from $p.t_s$ to t_e'. Since the support count of p in $B_{te'}$ is not maintained, an upper bound of $Max\_\ t_e$', denoted by $UB\_\ Max\_\ t_e$', is estimated by $ac/|DS[t_s, UB\_\ Max\_\ t_e'\text{-}1] \geq S_{min}$ and $ac/|DS[t_s, Max\_\ t_e'] < S_{min}$. If $UB\_\ Max\_\ t_e'$ is less than CTW_t^{first}, it is indicated that all the possible value of t_e' must be less than CTW_t^{first}. Accordingly, ct is the first time p appears within CTW_t.

When satisfying any one of the situations enumerated above, the (ct, ac) pair is removed from $p.Rqueue$, the starting accumulation time $p.t_s$ is adjusted to be ct; and accumulated support counts $p.f$ and $p.C_d$ are modified accordingly.

The situation that $p.t_s$ and $p.f$ are not adjusted occurs when $p.Rqueue$ is not empty for a pattern p and the first changing point ct in $p.Rqueue$ does not satisfy any situation enumerated above. It is implied that $ct>CTW_t^{first}$, $ac>1$, and $t_e' \geq CTW_t^{first}$. In this case, p is frequent in $DS[p.t_s, CTW_t^{first}\text{-}1]$ because there is not any frequency changing point appearing between $p.t_s$ and ct. When judging whether p is recently frequent according to its support count in $DS[p.t_s, t]$, even though false alarm may occur, it is certain that no false dismissal will occur.

Moreover, for each pattern p monitored in the SP-tree, if p appears in current basic block ($p.f_c > 0$) and has ever appeared in any previous basic block ($p.t_s<t$), the checking of a frequency changing point of p is performed. If t is certified to be a frequency changing point of p, a frequency changing point pair (ct, ac) is inserted into $p.Rqueue$, where $ct=t$ and $ac=((p.f - p.f_c)\text{-}p.C_d)$. Besides, $p.t_{pre}$ and $p.C_d$ are updated to be t and $(p.f - p.f_c)$, respectively. After the checking, $p.f_c$ is reset to be 0. Finally, the number of transactions in current basic block is obtained, which is used to update the data in TranArray.

[Example 1]

Table 1: an example of data streams.

B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	…
ab *a* *a*	*b* *c* *c*	*ab*	*a*	*a* *b*	*c* *cd* *c* *cd*	*a* *b*		*a*	*cd*	…

Table 1 shows an example of data streams. Suppose S_{min} is set to be 0.5 and window size w is 4. The process of constructing the SP-tree is described as the following.

From time point t_1 to t_2, there was not any frequency changing point occurring for the monitored patterns. The constructed SP-tree and TranArray at t_2 are shown in Figure 2(a). For simplifying the figures, the header table and the links connecting the nodes with same items are now displayed in the figure.

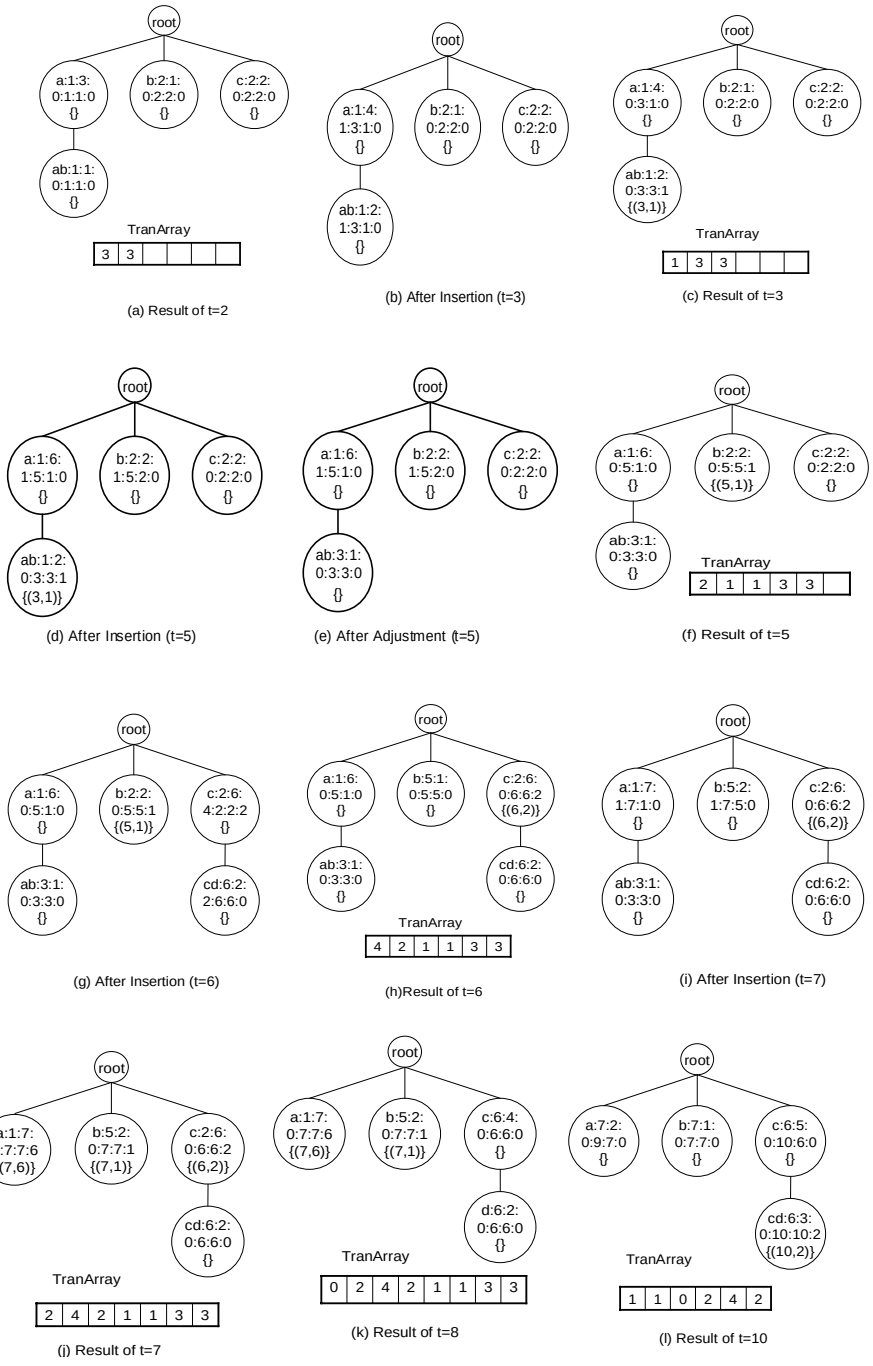

Figure 2: the maintenance of the monitoring data structure.

After processing B_3, time point t_3 is a frequency changing point of *ab*. Therefore, the changing point entry (3,1) is appended into *ab.Rqueue*. Besides, $ab.C_d$ is set to be $ab.f - ab.f_c$ =1, and $ab.t_{pre}$ is set to be 3. Accordingly, the resultant monitoring data structure is shown in Figure 2(c).

Continuing the similar processing, the resultant monitoring data structure at t_5 after inserting the patterns in B_5 into the data structure is shown in Figure 1(d). Then the value of $ab.t_s$ is adjusted because $ab.t_s$ is less than CTW_5^{first}(=2). According to the first frequency changing point pair of *ab*, which satisfies the first case of adjustment, $ab.t_s$ is reset to be 3, $ab.f$ is updated to be 1, and the pair is removed from *ab.Rqueue* as the result shown in Figure 2(e). After recording the frequency changing point pair, (5,1), of *b*, the result is shown in Figure 2(f).

After inserting the transactions in basic block B_6 as the result shown in Figure 2(g), since the first frequency changing point pair of *b* satisfies the second case of adjustment, $b.t_s$ and $b.f$ are adjusted to be 5 and 1, respectively. Moreover, *c.Rqueue* is updated as Figure 2(h) shows.

At time point t_7, the value of $ab.t_e$ is 3 (shown in Figure 2(i)) which less than CTW_7^{first}(=4). It is indicated that all the occurrences of *ab* are out of the range of current transaction window. Thus, the node of *ab* is removed from the PS-tree as the result shown in Figure 2(j).

In spite of no transaction being inputted at t_8, by satisfying the third case among the three conditions of adjustment, $c.t_s$ is adjusted to be 6 and $c.f$ is reset to be 6. The obtained result is shown in Figure 1(k), which shows the SP-tree correctly catches the recent occurrences of the monitored patterns *b*, *c*, *d*, and *cd* in the current transaction window. The estimated recent support of *a*, which is obtained according to its support in *DS*[1, 8] (7/16), is higher than its real recent support (2/8). However, after B_{10} is processed at time t_{10}, the summarization record of *a* will be adjusted as the monitoring data structure shown in Figure 1(e). Therefore, the estimation error is compensated after adjusting the starting point of support count accumulation.

4. Recently Representative Patterns Mining

According to the maintained SP-tree structure, the TD-FP Growth algorithm [11] is modified to perform on the structure to find all the patterns *p* with $Rsup_t(p) \geq S_{min}$ whenever needed. Moreover, the idea of RPlocal algorithm [12] is applied on the mining process to discover recently representative patterns.

The TD-FP Growth algorithm adopts a depth-first approach to perform pattern-growth. During the mining process, our approach constructs a *RP-tree* (Representative Pattern tree) to maintain the discovered representative patterns. The RP-tree is a trie structure, where each node contains a (*p*, *sup*) pair to denote a representative pattern *p* and its recent support, respectively.

A pattern *p* can be decided whether it is a representative pattern only when no more frequent pattern is grown from *p* or all the patterns grown from *p* have been decided. Therefore, a *RP-stack* is used to maintain those patterns which have not been certified to be representative patterns yet. There are five fields in each entry of *RP-stack*: (1) *p*: the itemset; (2) *sup*: the recent support of *p*; (3) *covered*: whether *p* is δ-covered by other patterns; (4) *cover_pattern*: the known maximum pattern which δ-covers *p*; (5) *cover_RP*: the known maximal representative pattern which δ-covers *p*.

For a newly discovered recently frequent itemset p, each pattern p' in *RP-stack* is a prefix of p. If p' has not been δ-covered by a representative pattern and it is δ-covered by p, $p'.covered$ is set to be 1. Besides, the pattern in $p'.cover\_pattern$ is replaced by p if p is larger than the original one.

In addition, the *RP-tree* is searched to find the representative patterns which δ-cover p. If more than one representative pattern δ-covers p, let p_r denote the maximum pattern among these satisfying patterns. Accordingly, $p.covered$ is set to be 2 and p_r is stored in $p.cover\_RP$.

Until the process of generating patterns grown from p is complete, the record of p is popped from *RP-stack*. According to the value stored in $p.covered$, whether p is a representative pattern is certified:

(1) $p.covered$=2: there is a representative pattern δ-covers p, *thus,* p is not a representative pattern.

(2) $p.covered$=1: p is δ-covered by the pattern p' stored in $p.cover\_patter$. If p' is found in the RP-tree, p is not a representative pattern because it is δ-covered by a representative pattern. Otherwise, p is certified to be a representative pattern and is inserted into the RP-tree.

(3) $p.covered$=0: there is not any pattern found to δ-cover p. Pattern p is certified to be a representative pattern and is inserted into the RP-tree.

5. Performance Study

The proposed GFCP (Generalized Frequency Changing Point) algorithm and Time-sensitive Sliding Window method (TSW in short) [7] were implemented using Visual C++ 6.0. The experiments have been performed on a 3.4GHz Intel Pentium IV machine with 1G megabytes main memory and running Microsoft XP Professional. Moreover, the datasets were generated from the IBM data generator [1], where each dataset simulates a data stream with each basic block consisting of 1000 transactions.

In the first part of experiments, the accuracy of mining results, execution time, and memory usage were measured to show the effectiveness and efficiency of the proposed GFCP algorithm for mining recently frequent itemsets by comparing with the ones of TSW algorithm. Furthermore, we extended the TSW algorithm by applying a local greedy method [12] to extract representative patterns from the discovered recently frequent itemsets. The performance of GFCP for mining recently representative patterns was observed in the second part of experiments by comparing with the one of the extended TSW algorithm.

[Experiment 1] To evaluate the effectiveness and efficiency of GFCP and TSW algorithms, the first part of experiments were performed on the dataset T5I4D100K with |N|=100. In this experiment, GFCP algorithm was performed to maintain the monitoring data structures, and TD_FP-Growth algorithm was performed once every 10 time points to find recently frequent itemsets.

[Exp.1-1] By comparing the mining results with the frequent itemsets found by Apriori algorithm on the corresponding CTW_t, the false dismissal rate(FDR), false alarm rate(FAR), and average support error(ASE) of the two algorithms were measured. Both GFCP and TSW guarantee that no false dismissal occurs, thus, the false dismissal rates of two algorithms are zeros. The false alarm rates of the two proposed algorithm at various time points are shown in Figure 3(a). In general, it is

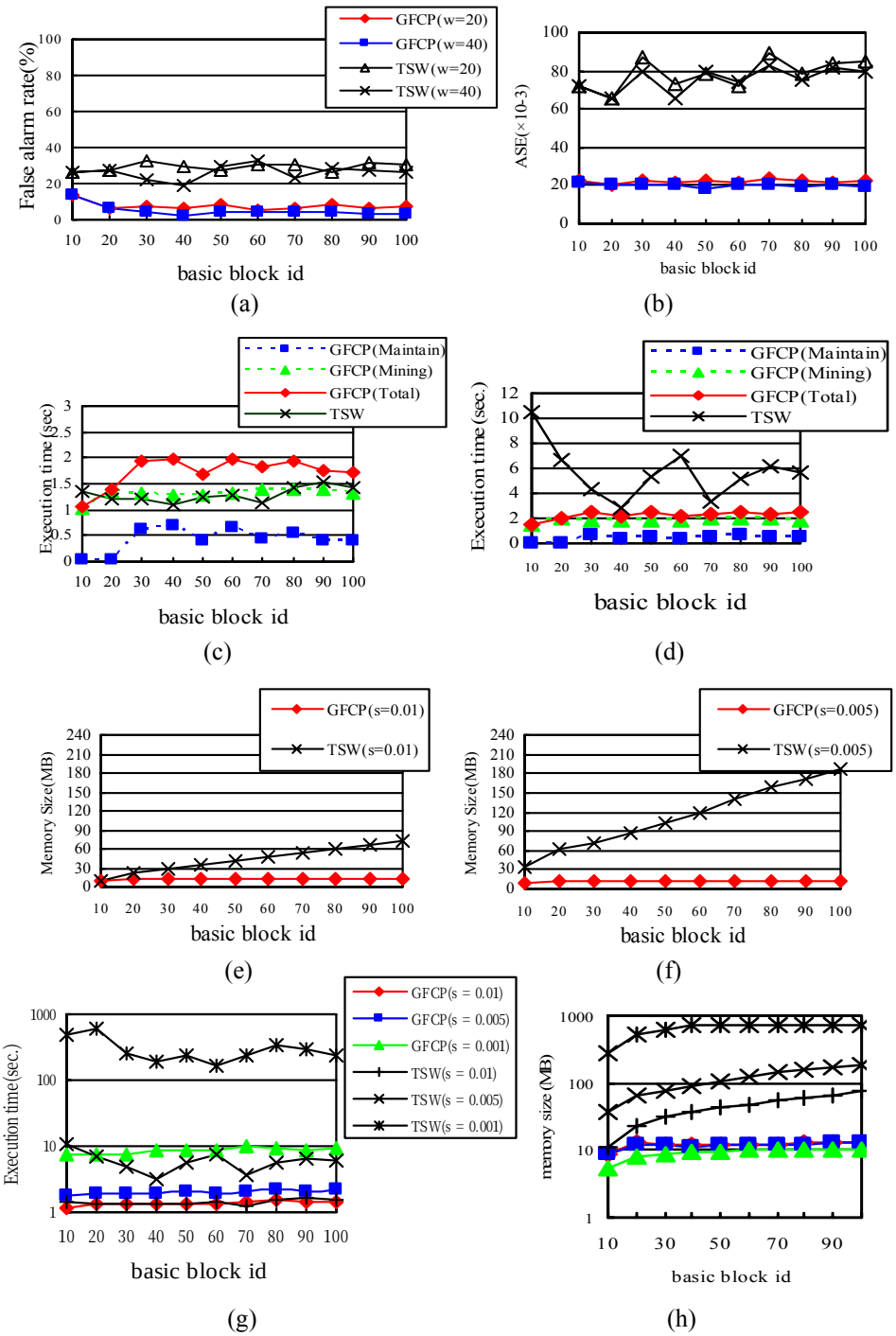

Figure 3: the experiment results.

reported that the FARs of the executions with window size 40 are less than the ones with window size 20. Besides, the FAR of FCP algorithm is below 10% on average, which is better than the one of TSW. From Figure 3(b) one may note that ASE($R_{GFCP}|R_{Apriori}$) keeps under 25×10^{-3} at different time points, which is about one third of ASE($R_{TSW}|R_{Apriori}$).

[Exp.1-2] In this experiment, GFCP and TSW algorithms are compared on their execution time (including the time of maintenance and mining) and maximum memory requirement. The dataset T5I4D1000K was experimented with window size =20. When S_{min} is varied to be 0.01 and 0.005, the results of execution time and maximum memory usages are shown in, Figure 3(c), 3(d), 3(e), and 3(f), respectively.

Although the total execution time of GFCP is more than the one of TSW algorithm slightly when S_{min} is 0.01, the execution efficiency of GFCP is much better than TSW when S_{min} is 0.005. This is due to the fact that the number of frequent itemsets in a basic block is increasing dramatically as the setting of S_{min} is reduced. Accordingly, the execution efficiency of TSW algorithm goes down because of the increasing cost of table maintenance. Regarding GFCP algorithm, little difference of maintenance time is observed between the two different settings of S_{min}. In addition, the effect of changing S_{min} on the mining time of GFCP is not significant. It should be noted that GFCP has better performance efficiency than TSW as S_{min} is smaller. By discarding the mining time, the maintenance time of GFCP is much less than the execution time of STW. These results suggest that, if it was not necessary to get mining results at any time, GFCP algorithm is a very efficient method for monitoring patterns in a data stream to discover recently frequent itemsets on demand. Furthermore, in contrast with TSW, the memory usage of GFCP is not sensitive to the setting of S_{min}, which also keeps stable at different time points. These results clearly demonstrate that GFCP is feasible for the streaming environment with a small memory.

[Experiment 2] In order to evaluate the GFCP algorithm and extended STW algorithm for mining recently representative patterns, the execution time and maximum memory usage were measured for different settings of S_{min}. The experiment was performed with window size=20 andδ=0.2. By varying the setting of S_{min} to be 0.01, 0.005, and 0.001, Figure 3(g) and 3(h) show the results of execution time and maximum memory usage, respectively.

The result shown in Figure 3(g) is in general agreement with the one of [Exp.1-2]. Especially, when S_{min} is set to be 0.001, the execution time of GFCP is 600 sec. less than the one of extended STW at most, which demonstrates the scalability of our approach. Clearly visible in Figure 3(h) is that the maximum memory usage of extended TSW is increasing as S_{min} is reduced. On the other hand, as a smaller S_{min} is set, the amount of frequency changing points maintained in the FP-tree is decreasing. This is the reason for the maximum memory usage of GFCP under S_{min} =0.001 is only 1/60 of the one required by TSW.

6. Conclusion

The sliding window approach proposes a good solution for providing time-sensitive mining of frequent itemsets in data streams. In this paper, a pattern summarization data structure based on the frequency changing point representation is provided to represent the occurrences of patterns in a data stream under a general input assumption. The effect of old transactions on the mining result of recently frequent

itemsets is diminished by performing adjusting rules on the monitoring data structure without needing to keep the whole transactions in the current sliding window physically. Moreover, to avoid generating redundant information in the mining results, the idea of local greedy method is applied to discover the recently representative patterns from the monitoring data structure. The experimental results demonstrate that the proposed GFCP algorithm achieves high accuracy for approximating the supports of recently frequent patterns and guarantees no false dismissal occurring. Not only the memory requirement by GFCP is significantly reduced by comparing with the one of TSW algorithm, but also the maintaining process of the monitoring data structure is very quick under various parameters setting. These results suggest that, if it was not necessary to get mining results at any time, GFCP algorithm is a very efficient method for monitoring patterns in a data streaming environment with a limited memory to discover recently representative patterns on demand.

References

[1] R. Agrawal and R. Srikant, “Fast Algorithms for Mining Association Rules,” in Proc. of Int. Conf. on Very Large Data Bases, 1994.

[2] J. H, Chang and W.S. Lee, “Finding Recent Frequent Itemsets Adaptively over Online Data Streams, ” in Proc. of the 9th ACM International Conference on Knowledge Discovery and Data Ming, 2003.

[3] J. H. Chang and W. S. Lee, “A Sliding Window Method for Finding Recently Frequent Itemsets over Online Data Streams, ” in Journal of Information Science and Engineering Vol. 20, pp753-762, 2004.

[4] J. Han, J. Pei, Y. Yin and R. Mao, “Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach”, Data Mining and Knowledge Discovery, 8(1):53-87, 2004.

[5] C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou, “Dynamically Maintaining Frequent Items Over a Data Stream,” in Proc. of the 12th ACM International Conference on Information and Knowledge Management, 2003.

[6] J. L. Koh and S. N. Shin, “An Approximate Approach for Mining Recently Frequent Itemsets from Data Streams,” in Proc. of 8th International Conference on Data Warehousing and Knowledge Discovery(DaWaK’06), 2006.

[7] C.H. Lin and D. Y. Chiu and Y.H. Wu and A.L.P. Chen "Mining Frequent Itemsets from Data Streams with a Time-Sensitive Sliding Window," in Proc. of SIAM Intl. Conference on Data Mining, 2005.

[8] G. S. Manku and R. Chen Motwani, “Approximate Frequent Counts over Data Streams, ” in Proc. of the 28th International Conference on Very Large Database, Hong Kong, China Aug, 2002.

[9] J.S. Park, M.S. Chen, and P.S. Yu, “An Effective Hash-based Algorithm for Mining Association Rules,” in Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD'95), May, pages 175-186, 1995.

[10] J. Pei, J. Han, and R. Mao "CLOSET: An efficient algorithm for mining frequent closed itemsets," in Proc. of ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, 2000.

[11] K Wang, L. Tang, J. Han, and J. Liu, “Top Down FP-Growth for Association Rule Mining, ” in Proc. of the 6th Pacific Area Conference on Knowledge Discovery and Data Mining, May 6-8, Taipei, Taiwan, PAKDD-2002.

[12] D. Xin, J. Han, X. Yan and H. Cheng, “Mining Compressed Frequent-Pattern Sets,” in Proc. of Int. Conf. on Very Large Data Bases (VLDB'05), 2005.