DELAY: A Lazy Approach for Mining Frequent Patterns over High Speed Data Streams*

Hui Yang[1], Hongyan Liu[2], and Jun He[1]

[1] Information School, Renmin University of China, Beijing, 100872, China
{huiyang,hejun}@ruc.edu.cn
[2] School of Economics and Management, Tsinghua University, Beijing, 100084, China
hyliu@tsinghua.edu.cn

Abstract. Frequent pattern mining has emerged as an important mining task in data stream mining. A number of algorithms have been proposed. These algorithms usually use a method of two steps: one is calculating the frequency of itemsets while monitoring each arrival of the data stream, and the other is to output the frequent itemsets according to user's requirement. Due to the large number of item combinations for each transaction occurred in data stream, the first step costs lots of time. Therefore, for high speed long transaction data streams, there may be not enough time to process every transactions arrived in stream, which will reduce the mining accuracy. In this paper, we propose a new approach to deal with this issue. Our new approach is a kind of lazy approach, which delays calculation of the frequency of each itemset to the second step. So, the first step only stores necessary information for each transaction, which can avoid missing any transaction arrival in data stream. In order to improve accuracy, we propose monitoring items which are most likely to be frequent. By this method, many candidate itemsets can be pruned, which leads to the good performance of the algorithm, *DELAY*, designed based on this method. A comprehensive experimental study shows that our algorithm achieves some improvements over existing algorithms, *LossyCounting* and *FDPM*, especially for long transaction data streams.

1 Introduction

Data stream is a potentially uninterrupted flow of data that comes at a very high rate. Mining data stream aims to extract knowledge structure represented in models and patterns. A crucial issue in data stream mining that has attracted significant attention is to find frequent patterns, which is spurred by business applications, such as e-commerce, recommender systems, supply-chain management and group decision support systems. A number of algorithms [5-16] have been proposed in recent years to make this kind of searching fast and accurate.

* This work was supported in part by the National Natural Science Foundation of China under Grant No. 70471006,70621061, 60496325 and 60573092.

But how fast does the task need to be done on earth (*challenge1*)? The high speed of streams answers the question: the algorithms should be as fast as the streams flow at least, that is to say, it should be so fast as to avoid missing data to guarantee the accuracy of mining results. Former algorithms [5-16] usually divide this process into two steps. One is calculating the frequency of itemsets while monitoring each arrival of the date stream (*step1*), and the other is to output the frequent itemsets according to user's requirement (*step2*). Due to the large number of item combinations for each transaction occurred in data stream, the first step costs lots of time. Therefore, for high speed long transaction data streams, there may be not enough time to process every transaction arrived in stream. As a result, some transactions may be missed, which will reduce the mining accuracy.

This problem can also lead to another challenge. If calculating the frequency of itemsets while monitoring each arrival of the date stream, the longer the transaction is, the more inefficiently the algorithm performs (*challenge2*). Unfortunately the transactions in data streams are often large, for example, sales transactions, IP packets, biological data from the fields of DNA and protein analysis.

Finding long pattern is also a challenge for mining of the static data set. Maxpattern [18] and closed-pattern [17] are two kinds of solutions proposed to solve this problem. These methods could avoid outputting some sub-patterns of frequent itemsets.

But for data stream what we concern more is how to reduce the processing time per element in the data stream. So the key to the efficiency of the algorithm is to reduce the number of candidates. Some papers [9,12,16] proposed several solutions of pruning method. But they all prune candidates by itemsets. In order to prune the candidate itemsets, frequency of itemsets must be calculated, which turns to the *challenge1*.

In this paper, we try to address the challenges discussed above. Our contributions are as follows.

(1) We propose a new approach, a kind of lazy approach to improve the processing speed per item arrival in data stream. This method delays calculation of the frequency of each itemset to step 2. So, the step 1 only stores necessary information for each transaction, which can avoid missing any transaction arrival in data stream. Furthermore, these two steps could be implemented in parallel, as they can be done independently.
(2) In order to improve the accuracy of mining result, we propose monitoring items which are most likely to be frequent. By this method, many candidate itemsets can be pruned. Based on this method, we develop an algorithm, *DELAY*, which can prune infrequent items and avoid generation of many subsets of transactions, especially for long transactions.
(3) We conducted a comprehensive set of experiments to evaluate the new algorithm. Experimental results show that our algorithm achieves some improvements over existing algorithms, *LossyCounting* and *FDPM*, especially for long transaction data streams.

The rest of the paper is organized as follows. In section 2, we review related work. In section 3, we formally formulate the problem of mining frequent itemsets over streams. Section 4 describes the proposed approach and algorithm, and section 5 gives the experimental results. Finally, Section 6 concludes our paper.

2 Background and Related Work

Throughout the last decade, a lot of people have implemented various kinds of algorithms to find frequent itemsets [2,4,17-20] from static data sets. In order to apply these algorithms to data stream, many papers [5-9] fall back on partition method such as sliding windows model proposed by Zhu and Shasha [5]. By this method, only part of the data streams within the sliding window are stored and processed when the data flows in. For example, a *lossyCounting* (a frequent item mining algorithm) based algorithm [9] divides a stream into batches, in which data is processed in a depth-first search style. For simplicity, we call this algorithm *LossyCounting* too. Time-fading model is a variation of sliding window model, which is suitable for applications where people are only interested in the most recent information of the data streams, such as stock monitoring systems. This model is implemented in [10,11,13], which gets more information and consumes more time and space in the meantime.

For the infinite of stream, seeking exact solution for mining frequent itemsets over data stream is usually impossible, which leads to approximate solution of this mining task [9,12,16]. Algorithms of this kind can be divided into two categories: false-positive oriented and false-negative oriented. The former outputs some infrequent patterns, whereas the latter misses some frequent patterns. *LossyCounting* [9] is a famous false-positive approximate algorithm in data stream. Given two parameters: support s and error ε, it returns a set of frequent patterns which are guaranteed by s and ε. Algorithm *FDPM* [16] is a false-negative approximate algorithm based on *Chernoff Bound* and has better performance than *LossyCounting*.

The above algorithms all perform well when transactions in data stream are not large or the stream does not flow at a high speed, that is to say, these algorithms could not meet the challenges mentioned in section 1. This could be explained by too much computation on data, so it could happen that next transaction has been here before last transaction is finished.

3 Problem Definition

Let $I = \{i_1, \ldots, i_m\}$ be a set of items. An itemset X is a subset of I. X is called k-itemset, if $|X| = k$, where k is the length of the itemset. A transaction T is a pair $(tid; X)$, where tid is a unique identifier of a transaction and X is an itemset. A data stream D is an open set of transactions. N represents the current length of the stream. There are two user-specified parameters: a support threshold $s \in (0, 1)$, and an error parameter $\varepsilon \in (0, 1)$. Frequent-patterns of a time interval mean the itemsets which appear more than sN times. When user submits a query for the frequent-patterns, our algorithm will produce the answers that following these guarantees:

(1) All itemsets whose true frequency exceed $(s + \varepsilon)N$ are output.
(2) No itemset with true frequency less than sN is output.There are no false positive.
(3) Estimated frequencies are less than the true frequencies by at most εN.

4 A New Approach

The strongpoint of our approach is the quick reaction to data stream, so the algorithm could achieve good performance for data streams flowing at high speed. Our approach is also a two-step method. In the first step (step 1), we just store necessary information of stream in a data structure. Frequent itemsets are found until the second step (step 2), the query for them comimg. In the first step, some itemsets which do not exceed a threshold are pruned so as to save space. The criterion of pruning is whether the count of every item in the itemset exceeds a threshold. The second step is a pattern fragment growth step which is the same as the second step of *FP-growth*[19]. So in this paper we focus on the first step.

4.1 Data Structures

There are two main data structures in this algorithm: *List* and *Trie*. The *List* is used to store possible frequent items; and the *Trie* is used to store possible frequent itemsets.

***List*:** a list of counters, each of which is a triple of $(itemid, F, E)$, where *itemid* is a unique identifier of an item, F is the estimation of the item's frequency; and E is the maximum error between F and the item's true frequency.

***Trie*:** a lexicographic tree, every node is a pair of (P, F), where P is a pointer that points one counter of *List*. In this way the association between itemsets and items is constructed and that is why we could prune itemsets by items. F is the estimated frequency of itemset that consists of the items from the root of *Trie* down to this node.

***List.update*:** A frequent itemset consists of frequent items. So, if any item of an itemset is infrequent, then the itemset can not be frequent. Since data stream flows rapidly, frequent itemsets are changing as well. Some frequent itemsets may become non-frequent and some non-frequent itemsets may become frequent. Therefore, technique to handle *concept-drifting*[3] needs to be considered. In this paper, we dynamically maintain a *List*, in which every item's estimated frequency and estimated error is maintained by a frequency ascending order. The method used to update the *List* is based on *space-saving*[1].

***Trie.update*:** While updating *List*, the nodes which point the items deleted for becoming infrequent will also be deleted. This is just the method by which we implement itemsets' pruning by items. For each transaction arrived in the data stream, its subset consisting of items maintained in the *List* will be inserted into the *Trie* .

A example *List* and *Trie* for a stream with two transactions is shown in Fig 1 (a).

4.2 Algorithm *DELAY*

Based on the new approach, we develop an algorithm, *DELAY*, which is shown as followed.

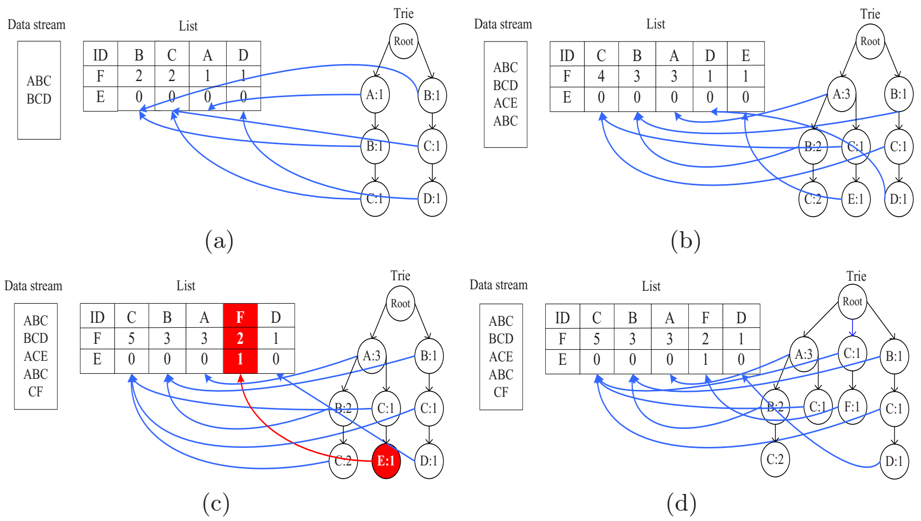
```
Algorithm: DELAY (data stream S, support s, error $\varepsilon$)
Begin
1. List.length = $\lceil m/\varepsilon \rceil$;
2. For each transaction t in S
3.     List.update (t);
4.     Delete the items from t which are not in the list;
5.     Insert t into trie;
6.     If user submits a query for frequent-patterns
7.         FP-growth(trie, s);
8.     end if
9. end for
End.
```

```
Procedure List.update (transaction t, tree trie)
Begin
1. for each item, e, in t
2.     If e is monitored, then increment the F of e;
3.     else
4.         let $e_m$ be the element with least frequency, min
5.         delete all node from trie which point to $e_m$;
6.         Replace em with e;
7.         Increment F;
8.         Assign $E_i$ the value min;
9.     end if
10. end for
End.
```

The main steps of *DELAY* are as follows. First, we define the length of *List*, l, to be $\lceil m/\varepsilon \rceil$ (line 1),m is the average length of transactions in data stream. Then, for every transaction t of data stream S, we update the *List* with the items of the transaction by procedure *List.update* (line 3). For those items which are not monitored in the *List*, delete them from t (line 4). Then insert transaction t into the *Trie* (line 5). Whenever a user submits a query for frequent itemsets, a procedure *FP-growth*[19] will be used to find and output answers using the information stored in *Trie* (line 6-8).

The procedure of *List.update* is similar to *space-saving*[1]. If we observe an item, e, that is monitored, we just increase its F (line 2). If e is not monitored, give it the benefit of doubt, and find from the *List* item e_m, the item that currently has the least estimated hits, *min* (line 4). Nodes of the *Trie* which point to item e_m are deleted (line 5). Then, item e_m is replaced by e (line 6). Assign F_m the value $(min+1)$ (line 7). For each monitored item e_i, we keep track of its over-estimation error, E_i, resulting from the initialization of its counter when it was inserted into the *List*. That is, when starting to monitor e_i, set $E_i = min$, the estimated frequency of the evicted item e_m.

An example of this algorithm is shown in Fig 1. Fig 1 (a) and (b) show the change of *List* and *Trie* without any item of *List* being replaced. In (c), with the coming of transaction with items CF, the number of unique items has exceeded the length of *List*, so the last item E in the *List* is replaced by F with frequency 2, and the nodes in *Trie* which point to E are deleted (the red one). In (d), transaction CF is inserted into *Trie*.



(a) List

ID	B	C	A	D
F	2	2	1	1
E	0	0	0	0

(b) List

ID	C	B	A	D	E
F	4	3	3	1	1
E	0	0	0	0	0

(c) List

ID	C	B	A	F	D
F	5	3	3	2	1
E	0	0	0	1	0

(d) List

ID	C	B	A	F	D
F	5	3	3	2	1
E	0	0	0	1	0

Fig. 1. An example

4.3 Properties of *DELAY*

Lemma 1. *For any item(e_i, F_i, E_i) monitored in* List, $E_i \leq \varepsilon N$

This is proved in [1].

Lemma 2. *Given a support threshold s, for a stream of length N, the error of frequent itemset E_p is bounded by εN, that is, $f_p - F_p \leq \varepsilon N$.*

Proof. If itemset p is frequent, then $F_p \geq sN$;

Assume $e_1(e_1, F_1)$ is the item with the least estimated frequency in p, and its real frequency is f_1.

Assume $e_2(e_2, F_2)$ is the item with the least real frequency in p, and its real frequency is f_2.

DELAY is a false-negative algorithm, so $f_i \geq F_i$.
So, $E_p = f_p - F_p = f_2 - F_1 \leq f_1 - F_1 \leq \varepsilon N$.

Theorem 1. *An itemset p with $f_p \geq (s + \varepsilon)N$, must be found by* DELAY.

Proof. For itemset p, according to lemma 2, $f_p \leq F_p + \varepsilon N$. If $f_p \geq (s + \varepsilon)N$ then $F_p \geq sN$, so our algorithm *DELAY* will output itemset p.

Theorem 2. *Time spent in step 1 is bounded by $O(Nm + Nt)$, where N is the length of data stream, m is the average length of transactions in data stream, and t is the time of inserting one itemset into* Trie.

Proof. In step 1 each transaction needs to be processed by *List.update* first. This part consumes time $O(Nm)$. Then the transaction is inserted into *Trie*. The time complexity of this part is difficult to estimate. Here we assume the time of inserting one itemset into *Trie* is t ignoring the difference of length between itemsets. For *DELAY*, one transaction one itemset needs to be inserted into the *Trie*. so the time complexity of this part is $O(Nt)$. So time spent in step 1 is bounded by $O(Nm + Nt)$. In the following we compare the times of inserting between *LossyCounting* [9] and *DELAY*.

4.4 Comparison with Existing Work

4.4.1 Comparison with *LossyCounting*

Theorem 2 proves that time spent in the step 1 by *DELAY* is about $O(Nm+Nt)$. *LossyCounting* [9] has a bound of $O((2^m)N + \frac{(2^m)N}{k\varepsilon}t)$, where m is the average length of transactions in data stream, k is the buffer size, N and t with the same definition of Theorem 2. Let's compare the time bound of *DELAY* and *LossyCounting*.

The time of inserting itemsets into *Trie* is difficult to estimate, so we just compare the times of inserting. The times of inserting in *DELAY* is N, that is one transaction one insertion into *Trie*, whereas the times of inserting itemsets in *LossyCounting* is $\frac{2^m N}{k\varepsilon}$. In *LossyCounting* itemsets whose frequencies exceed εk will be inserted into *Trie*, and every subsets of them will be inserted respectively too. $Nm + Nt < 2^m N + \frac{2^m N}{k\varepsilon}t$, when $k < \frac{2^m}{\varepsilon}$, that is, given $m = 30$, $\varepsilon = 0.001$, only when $k > 10^{12}$, *DELAY* will consume more time than *LossyCounting* in step 1, but 10^{12} is a huge number for memory space. So *DELAY* usually consumes less time. So we can see that the average length of transaction, m, is the determinant of which one performs well, which is also demonstrated in the following experiments.

4.4.2 Comparison with *FDPM*

FDPM proposed in [16] finds frequent patterns through two steps. First calculate the frequency of itemsets, and then prune them based on *Chernoff Bound*. The merit of it is in the space bound. As the method of calculating the frequency of itemsets is not given in the paper, we could not estimate the time bound. But as long as the algorithms need to calculate frequency of itemsets, they will consume more time than the step 1 of *DELAY* for the same data set, which will be proved in the following section.

5 Experimental Results

In this section we conducted a comprehensive set of experiments to evaluate the performance of *DELAY*. We focus on three aspects: time, space and sensitivity to parameter. Algorithm *DELAY* is implemented in C++ and run in a Pentium IV 2.4GHz PC with 1.0 G RAM and 40G Hard disk. In the experiments, we use the synthetic datasets generated by IBM data generator[21].

5.1 *DELAY*

In this section, we design two sets of experiments to test the performance of DELAY.

5.1.1 Time and Space

We fix $s = 1\%$, $\epsilon = s/10$, the average length of transaction $L = 30$, and vary the length of data stream from 100k to 1000k. Fig.2(a) shows the time used in step 1 (*step1 time*) and the time used in step 2(*step2 time*), and Fig.2(b) shows the memory consumption.

As shown in Fig.2(a) *step1 time* is linear with the length of data stream, accounting for only a small part of the total runtime, which means that this algorithm could deal with streams flowing at a high speed. Fig.2(b) shows that the increase of the memory consumed slows down with the increase of the length of stream.

This experiment proves that *DELAY* could potentially handle large-scale data stream of high speed, consuming only limited memory space.

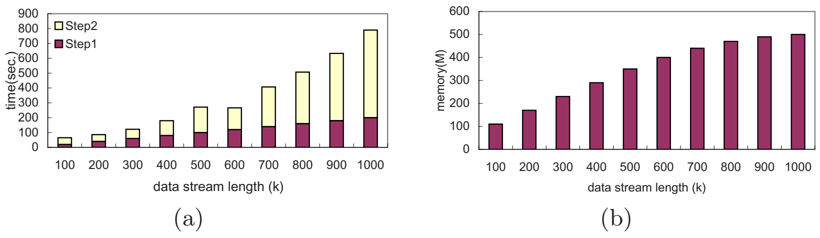


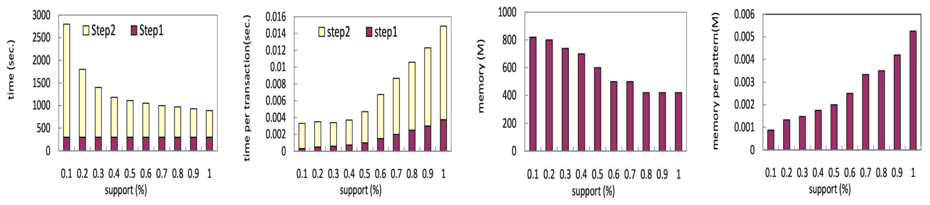
Fig. 2. Time and Space

5.1.2 Sensitivity to the Support Threshold

In this set of experiments, We generate dataset T30.I10.D1000k, set $\epsilon = s/10$, and vary s from 0.1% to 1%. Fig.3(a), (b), (c) and (d) show the total runtime, *step1* time per pattern, *step2* time per pattern, memory and memory per pattern respectively as the support varies.

As shown in Fig.3(a), *step1 time* remains almost stable as support varies. This is because *step1 time* is only relative to the length of stream, and *DELAY* do nothing different for different support value. Fig.3(b) show that the average run time per frequent pattern (itemset) decreases as support decreases though the total time is increased. Fig.3(c) and Fig.3(d) indicate that more memory is needed as support decreases, but the memory consumed per frequent pattern keeps steady on the whole.

The behavior of *DELAY* with the variation of error level, ϵ, is similar to the results of this set of experiments. Due to space limitation, we do not give the results here.



(a) (b) (c) (d)

Fig. 3. Sensitivity to the support threshold

5.2 Comparison with *LossyCounting* and *FDPM*

In this section, we compare *DELAY* with *LossyCounting* and *FDPM* in the following aspects: reaction time, total run time, memory requirements under different dataset size and support levels.

5.2.1 Average Length of Transactions

We fix $s = 1\%$, $\varepsilon = s/10$, the length of data stream = 100k, and vary average length of transaction from 10 to 60.

Fig.4 shows the change of run time of *DELAY*, *LossyCounting* and *FDPM* as the length of transactions increases. As shown in Fig.4, *DELAY* significantly outperforms *LossyCounting* and *FDPM* on the running time, and the excellence of *DELAY* is more evident when transaction becomes longer. The reason has been explained in section 4.3.

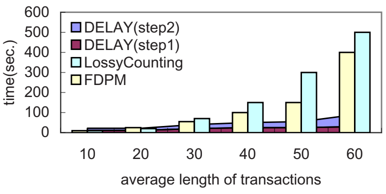


Fig. 4. Transaction length

5.2.2 Length of Stream

In this section we test the algorithms with two data sets: T30.I10.D?k and T15.I6.D?k. For this set of experiments, we fix $s = 1\%$, $\varepsilon = s/10$, and report the time and memory usage as the length of data stream increases from 100k to 1000k.

For T30.I10.D?k, Fig.5 shows the results. As shown in Fig.5(a), *DELAY* significantly outperforms *LossyCounting* and *FDPM* on the running time for the stream with relatively longer transactions. Fig.5(b) shows that the increasing speed of memory by *DELAY* as the increase of stream size is faster than *Lossy-Counting* and FDPM, but the ability of processing streams of higher speed is worthy of the sacrifice of memory.

For T15.I10.D?k, Fig.5(c) shows the results. As shown in Fig.5(c), though *DELAY* does not perform better than *LossyCounting* and *FDPM* do when transactions are short, but the *step1* time of *DELAY* shows that *DELAY* could work on streams flowing at high speed.

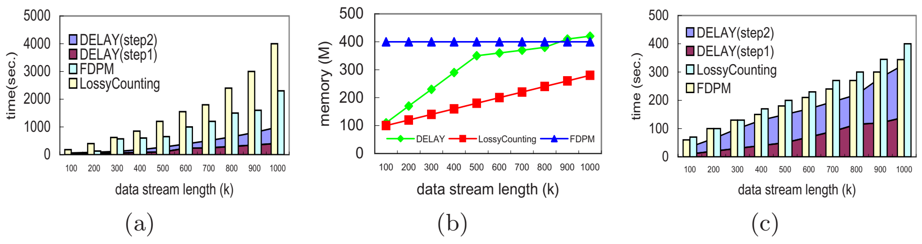


Fig. 5. Length of Stream

5.2.3 Support Threshold

In this section we test the algorithms with two data sets: T30.I10.D1000k and T15.I6.D1000k. We fix $\varepsilon = s/10$, and vary s from 0.1% to 1%.

For T30.I10.D1000k, Fig.6(a), (b), (c) and (d) show the results. As shown in Fig.6(a) and (b), *DELAY* outperforms *LossyCounting* and *FDPM* for streams of long transactions as support increases. Fig.6(c) and (d) show that *DELAY* consumes a bit more memory than *LossyCounting* and *FDPM*.

For T15.I10.D1000k, Fig.6(e) and (f) show the results. As shown in these Figures, the *step1 time* of *DELAY* is almost unchanged with the varying of support level. Though *DELAY* does not significantly outperform *LossyCounting* and *FDPM* for short transactions overall, *DELAY* could perform better when support level is relatively low.

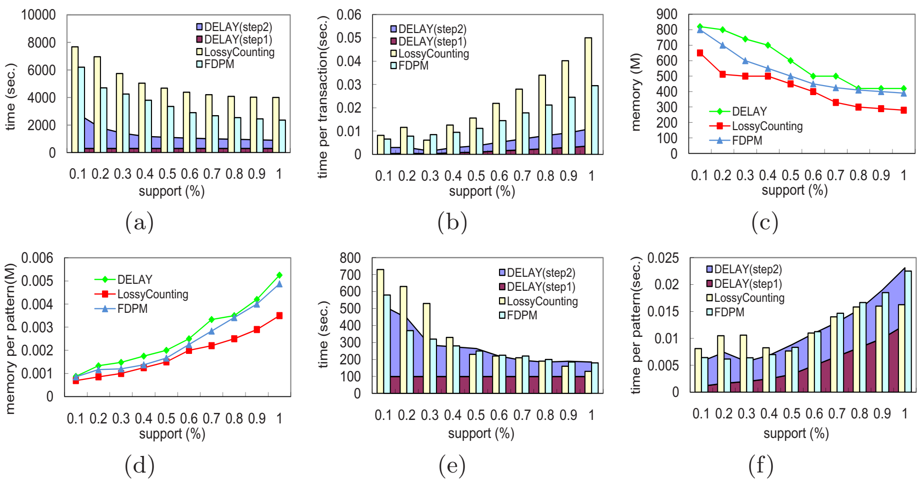


Fig. 6. Varying s

5.2.4 Recall and Precision

In order to evaluate the accuracy of these algorithms, we test the recall and precision for streams with different average length of transactions, different support threshold and different error rate. Recall and precision are defined as follows. Given a set A of true frequent itemsets and and a set B of frequent itemsets output by the algorithms. The recall is $\frac{|A \cap B|}{|A|}$ and the precision is $\frac{|A \cap B|}{|B|}$.

Fixing $s = 1\%$, $\varepsilon = s/10$, the length of data stream 1000k, we vary the average length of transactions from 10 to 50. Table 1 shows the recall and precision of *DELAY*, *LossyCounting* and *FDPM*. *DELAY* is a false-negative algorithm, which ensures its precisions to be 100%, *DELAY* achieves a little higher recall than *FDPM* on the average.

For dataset T30.I10.D1000k, setting $\varepsilon = s/10$, Table 2 lists the recall and precision of *DELAY*, *LossyCounting* and *FDPM* with different support thresholds. It shows that recall increase as support thresholds decrease. That is because the error of items is bound by εN, and the error distributes to lots of itemsets. When the support decreases, there will be more frequent itemsets, so the error per itemset becomes less.

For dataset T30.I10.D1000k, Table 3 shows the recall and precision of *Lossy-Counting* and *DELAY* as ε increases from 0.01% to 0.1% and support is fixed to be 1%. It tells us that *DELAY* can get high recall under the condition of maintaining precision to be 1.

Table 1. Varying transaction length

L	*DELAY*		*FDPM*		*LC*	
	R	P	R	P	R	P
10	1	1	1	1	1	0.89
20	1	1	1	1	1	0.6
30	0.96	1	0.93	1	1	0.54
40	0.92	1	0.93	1	1	0.51
50	0.9	1	0.9	1	1	0.42

Table 2. Varying support s

S %	*DELAY*		*FDPM*		*LC*	
	R	P	R	P	R	P
0.1	0.97	1	1	1	1	0.89
0.2	0.96	1	0.99	1	1	0.97
0.4	0.93	1	0.99	1	1	0.97
0.6	0.85	1	0.98	1	1	0.87
0.8	0.89	1	0.98	1	1	0.73
1	0.78	1	0.96	1	1	0.79

Table 3. Varying error ε

error%	*DELAY*		*FDPM*		*LC*	
	R	P	R	P	R	P
0.01	1	1	1	1	1	0.99
0.02	1	1	0.99	1	1	0.99
0.04	1	1	0.99	1	1	0.98
0.06	1	1	0.98	1	1	0.93
0.08	0.96	1	0.98	1	1	0.9
0.1	0.97	1	0.96	1	1	0.85

6 Conclusions

In this paper, we propose a lazy approach for mining frequent patterns over a high speed data stream. Based on this approach, we develop an algorithm, *DELAY*, which does not calculate the frequency of itemsets as soon as the data arrive in data stream like other algorithms, but only stores necessary information. The frequency is not calculated until the query for frequent itemsets comes, which can avoid missing any transaction arrival in data stream. This kind of delay also helps this method to perform well for long transaction data streams. In order to reduce the information needed to store, we propose monitoring items which are more likely to be frequent. In the meantime, this algorithm can also guarantee a predefined error rate.

References

1. Metwally, A., Agrawal, D., Abbadi, A.E.: Efficient Computation of Frequent and Top-k Elements in Data Streams. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363, Springer, Heidelberg (2004)
2. Bayardo Jr., R.J.: Efficiently Mining Long Patterns from Databases. In: Proceedings of the ACM SIGKDD Conference (1998)
3. Wang, H., Fan, W., Yu, P.S., Han, J.: Mining Concept-Drifting Data Streamsusing Ensemble Classifiers. In: ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (August 2003)
4. Agrawal, R., Imielinski, T., Swami, A.: Mining Association Rules between Sets of Items in Massive Databases. In: Int'l Conf. on Management of Data (May 1993)
5. Zhu, Y., Shasha, D.: StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In: Int'l Conf. on Very Large Data Bases (2002)
6. Chi, Y., Wang, H., Yu, P.S., Richard, R.: Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window. In: IEEE Int'l Conf. on Data Mining (November 2004)
7. Chang, J.H., Lee, W.S.: A Sliding Window Method for Finding Recently Frequent Itemsets over Online Data Streams. Journal of Information Science and Engineering (2004)
8. Cheng, J., Ke, Y., Ng, W.: Maintaining Frequent Itemsets over High-Speed Data Streams. In: Ng, W.-K., Kitsuregawa, M., Li, J., Chang, K. (eds.) PAKDD 2006. LNCS (LNAI), vol. 3918, Springer, Heidelberg (2006)
9. Manku, G.S., Motwani, R.: Approximate Frequency Counts over Data Streams. In: Int'l Conf. on Very Large Databases (2002)
10. Chang, J.H., Lee, W.S., Zhou, A.: Finding Recent Frequent Itemsets Adaptively over Online Data Streams. In: ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (August 2003)
11. Giannella, C., Han, J., Pei, J., Yan, X., Yu, P.S.: Mining Frequent Patterns in Data Streams at Multiple Time Granularities. In: Data Mining: Next Generation Challenges and Future Directions, AAAI/MIT Press, Cambridge (2003)
12. Li, H.-F., Lee, S.-Y., Shan, M.-K.: An Efficient Algorithm for Mining Frequent Itemsets over the Entire History of Data Streams. In: Int'l Workshop on Knowledge Discovery in Data Streams (September 2004)
13. Chang, J.H., Lee, W.S.: A Sliding Window Method for Finding Recently Frequent Itemsets over Online Data Streams. Journal of Information Science and Engineering (2004)
14. Charikar, M., Chen, K., Farach-Colton, M.: Finding Frequent Items in Data Streams. Theoretical Computer Science (2004)
15. Lin, C.-H., Chiu, D.-Y., Wu, Y.-H., Chen, A.L.P.: Mining Frequent Itemsets from Data Streams with a Time-Sensitive Sliding Window. In: SIAM Int'l Conf. on Data Mining (April 2005)
16. Yu, J.X., Chong, Z.H., Lu, H.J., Zhou, A.Y.: False positive or false negative: Mining frequent Itemsets from high speed transactional data streams. In: Nascimento, M.A., Kossmann, D. (eds.) VLDB 2004. Proc. of the 30th Int'l Conf. on Very Large Data Bases, pp. 204–215. Morgan Kaufmann Publishers, Toronto (2004)
17. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering frequent closed itemsets for association rules. In: Beeri, C., Bruneman, P. (eds.) ICDT 1999. LNCS, vol. 1540, pp. 398–416. Springer, Heidelberg (1998)

18. Bayardo Jr., R.J.: Efficiently mining long patterns from databases. In: Haas, L.M., Tiwary, A. (eds.) Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data. SIGMOD Record, vol. 27(2), pp. 85–93. ACM Press, New York (1998)
19. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Mining and Knowledge Discovery (2003)
20. Zaki, M.J.: Scalable algorithms for association mining. IEEE Transactions on Knowledge and Data Engineering 12(3), 372–390 (2000)
21. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proc. of 20th Intl. Conf. on Very Large Data Bases, pp. 487–499 (1994)