Mining Recent Frequent Itemsets in Data Streams with Optimistic Pruning

Kun Li[1,2], Yongyan Wang[1], Manzoor Elahi[1,2], Xin Li[3], and Hongan Wang[1]

[1] Institute of Software, Chinese Academy of Sciences, Beijing, China, 100190
likun@ustc.edu
[2] Graduate University of Chinese Academy of Sciences, Beijing, China, 100049
[3] Department of Computer Science and Technology, Shandong University, Ji'nan, China, 250101

Abstract. A data stream is a massive unbounded sequence of transactions continuously generated at a rapid rate, so how to process the transactions as fast as possible in the limited memory becomes an important problem. Although it has been studied extensively, most of the existing algorithms maintain a lot of infrequent itemsets, which causes huge space usage and inefficient update. In this paper, a new algorithm, called OPFI-stream, is proposed to mine all accurate frequent itemsets from sliding window over data streams. The OPFI-stream algorithm maintains a dynamically selected set of itemsets in a prefix-tree based data structure. By using an optimistic pruning strategy, quite a lot of infrequent itemsets can be pruned during the construction and updates. Mining all frequent itemsets with accurate frequencies is just to traverse the tree. Experiments show that the performance is improved greatly even when the user-specified minimum support threshold is small.

Key words: frequent itemsets, optimistic pruning, data stream, sliding window

1 Introduction

Data stream is a new emerging class of applications in recent years, which is often continuous, unbounded, high-speed and has a data distribution changing with time [1]. Examples of such applications include financial analysis, network monitoring, sensor networks, telecommunication data management, and others.

Mining frequent itemsets becomes one of the most important problems in the data stream research area and presents new challenges. To mine data streams efficiently, a highly compact data synopsis is needed and multiple scans on data streams are not acceptable. Furthermore, recent data is more attractive than the old history in data streams. It is a challenge to mine frequent itemsets from recent data, since new items arrive and old items overdue with high speed. Mining data streams requires fast, real-time processing in order to keep up with the high-speed data arrival and mining results must be returned within short response time.

There are many algorithms on mining frequent itemsets, but most of them have some disadvantages. Traditional algorithms mining on static dataset usually have candidate generation, which often causes the problem of combinatorial explosion. Most algorithms mining on data streams discard the candidate generation process and use the apriori [2] property to prune some infrequent itemsets. But this pruning is far from enough because there are a large number of infrequent itemsets. These algorithms still maintain a lot of infrequent itemsets, and the performance degrades dramatically when the user-specified minimum support threshold is small.

In this paper, a new sliding window-based algorithm, called **O**ptimistic **P**runing strategy based **F**requent **I**temset mining in Data Stream (abbreviated as OPFI-stream), is proposed to mine all accurate frequent itemsets from data streams. There is also no candidate generation in the OPFI-stream algorithm. It uses a prefix-tree based data structure, called OPFI-tree, to maintain all frequent itemsets in the sliding window. When constructing and updating the tree, an optimistic pruning strategy is used to do the maximal pruning and prune most of the infrequent nodes. It is both time and space efficient in update and mining processes even when the user-specified minimum support threshold is small.

The rest of the paper is organized as follows. Section 2 describes the related work on mining frequent itemsets in data streams. Section 3 gives the problem formulations. The algorithm details are described in section 4. The experiments are shown in section 5 and the last section concludes the paper.

2 Related Work

Mining frequent itemsets is first introduced by Agrawal et al.[2], and appears as a basic step in many other mining problems such as mining association rules [2], clustering [3], classification [4], etc. Apriori [2], FP-tree [5], Eclat [6] are three typical algorithms of mining frequent itemsets on static dataset. And there are hundreds of follow-up research publications on mining frequent itemsets on static dataset. Most of these algorithms use a property, called apriori [2], to prune some infrequent itemsets. The apriori property means that an itemset is frequent only if all its sub-itemsets are frequent. Because traditional algorithms mine the static dataset and require multiple scans on the dataset, they are not suitable for mining data streams.

Researchers proposed many new algorithms on mining data streams during the recent years, and these algorithms can be classified as approximate or accurate and landmark window or sliding window. Lossy Counting [7] and FP-stream [8] are approximate algorithms of mining data streams in landmark window. LWF [9] is similar to Lossy Counting in finding frequent items first and then generating frequent itemsets. There are also some algorithms mining recent frequent itemsets in data streams. Chang et al. [10] proposed the estWin algorithm to find approximate frequent itemsets over sliding windows. James Cheng et al. [11] proposed the sliding window-based MineSW algorithm to mine approximate frequent itemsets. Moment [12] is the first algorithm to maintain all closed fre-

quent itemsets from sliding window over data streams. An itemset is closed if none of its proper supersets has the same frequency as it has. All frequent itemsets with accurate frequencies can be derived from the closed frequent itemsets. Some algorithms, such as Lossy Counting [7] and estWin [10], have candidate generation and suffer from the combinatorial explosion problem. The FP-stream [8] and Moment [12] algorithms maintain a lot of infrequent itemsets, which causes huge memory usage and inefficient update. So the OPFI-stream algorithm is proposed, which is sliding window-based and returns accurate results. It has no candidate generation and prunes most of infrequent itemsets.

3 Problem Formulation

Definition 1 (Itemset) *Let Σ be a set of items. We assume that there is a lexicographical order among the items in Σ and we use $a \prec b$ to denote that item a is lexicographically smaller than item b. An itemset, $X = \{x_1, x_2, ..., x_n\}$ is a subset of Σ. An itemset containing k items is called a k-itemset. Furthermore, an itemset can be represented by a sequence, wherein items are lexicographically ordered. For instance, $X = \{a, b, c\}$ is represented by abc, given that $a \prec b \prec c$. We also use $\prec$ to denote the lexicographical order between two itemsets, i.e., $ab \prec abc \prec ac$.*

Definition 2 (Transaction) *A transaction is a tuple, (tid, X), where tid is the id of the transaction and X is an itemset. The transaction supports an itemset, Y, if $X \supseteq Y$. For simplicity, we may omit the tid when it is irrelevant to the underlying idea of an algorithm.*

Definition 3 (Data Stream) *A transaction data stream is a sequence of incoming transactions. For simplicity, we use data stream instead of transaction data stream. A window, W, can be 1) either time-based or count-based, and 2) either a landmark window or a sliding window. In this paper, we use a count-based sliding window with fixed size of N, which always contains recent N transactions.*

Definition 4 (Frequent Itemset) *The frequency of an itemset X in W, denoted as $freq(X)$, is the number of transactions in W that support X. The support of X in W, denoted as $sup(X)$, is defined as $freq(X)/N$. Let minsup be the user-specified minimum support threshold, where $0 < minsup < 1$, we say itemset X is a Frequent Itemset in W, if $sup(X) \geq minsup$ or $freq(X) \geq minsup*N$.*

The problem of mining recent frequent itemsets is to mine all frequent itemsets with accurate frequencies from sliding window over data streams. And the results can be returned at any time.

4 The OPFI-stream Algorithm

In this section, OPFI-stream algorithm is proposed to dynamically maintain all frequent itemsets from the sliding window over data streams. The algorithm uses

a prefix-tree based structure, called Optimistic Pruning strategy based Frequent Itemsets tree (abbreviated as OPFI-tree). Most infrequent nodes in the tree can be pruned by the optimistic pruning strategy. All frequent itemsets and their accurate frequencies can be returned at any time by a simple traversal on the OPFI-tree. When the window moves, updating the OPFi-tree needs to delete the oldest transaction, so all transactions in the sliding window are maintained in the bit-sequences. A bit sequence of N bits is used to represent an item's occurrence information in the sliding window. If an item appears in the i-th transaction of the current window, the i-th bit of the item's bit sequence is set to be 1; otherwise, it is set to be 0. The bit-sequence representation uses small space and is very time efficient in computing itemset frequencies.

In order to help understand the algorithm better, we will first introduce the basic OPFI-tree, which has no optimistic pruning. There are three important phases on the basic OPFI-tree: constructing the tree in the window initialization phase, updating the tree in the sliding phase, and mining the tree at any time after the initialization phase. Then, we will apply the optimistic pruning strategy on the basic OPFI-tree. Along with the descriptions, we will analyze OPFI-tree's important properties and correctness.

4.1 The Basic OPFI-tree

The basic OPFI-tree is similar to the tree described in [13]. It can be proved [13] that all frequent itemsets are maintained in the basic OPFI-tree, but the proof details are omitted here for lack of space. The basic OPFI-tree is a prefix-tree based data structure. It is the OPFI-tree without the optimistic pruning strategy. A node in the basic OPFI-tree stores the following information: $item, frequency, isFrequent, children$. A path from the root to a node in the basic OPFI-tree represents an itemset, so we will use n_X to denote a node, where X is the itemset it represents. In the rest of the paper, we will say a node is (in)frequent if and only if its represented itemset is (in)frequent.

The basic OPFI-tree uses the apriori property [2] in construction and updates to prune some infrequent nodes. There are two corollaries for us to use the apriori property more efficiently, which can be derived from the apriori property directly.

Corollary 1 *If n_X is an infrequent node, then all children of n_X are also infrequent.*

Corollary 2 *Itemset $X = \{a_1, ..., a_i, a_j\}$ is frequent, $Y = \{a_1, ..., a_i, a_k\}$ is infrequent, where $a_1 \prec ... \prec a_i \prec a_j \prec a_k$, then $X \cup Y = \{a_1, ..., a_i, a_j, a_k\}$ is also infrequent.*

The first corollary indicates that all the children of an infrequent node should be pruned. Then we can see that all infrequent nodes are leaves in the basic OPFI-tree. The second corollary means that some children of a frequent node may be infrequent and should be pruned.

Because users are only interested in frequent itemsets, there is no need to maintain all itemsets in the basic OPFI-tree. However, it is impossible to know a

node changing from infrequent to frequent if we only maintain frequent itemsets. So the basic OPFI-tree maintains all frequent itemsets and a selected part of infrequent itemsets. Then the algorithm will monitor the node status changes. From corollary 1, all the children of an infrequent node should be pruned. When the infrequent node becomes frequent, it may have frequent children, so the subtree must be rebuild under this node. On the contrary, when a frequent node changes to infrequent, all its descendants must be deleted from the tree. However, most of nodes do not often change their status, and even if some status changes occur, the operations will be limited in a small subtree. So the cost of updates is small.

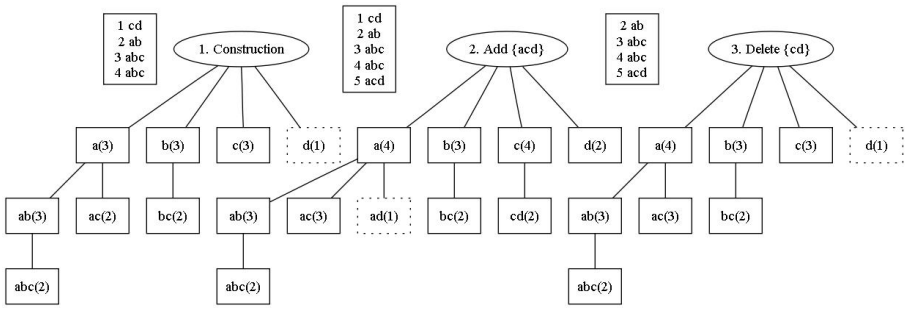


Fig. 1. Examples of Construction and Updates on the Basic OPFI-tree

The basic OPFI-tree is constructed in three steps:

1. Create a root node n_ϕ.
2. Create $|\Sigma|$ child nodes for n_ϕ in the lexicographical order, i.e., each $a \in \Sigma$ corresponds to a child node $n_{\{a\}}$.
3. Recursively do the construction on each child node $n_{\{a\}}$, where $a \in \Sigma$. The main principle in the construction is that if n_X is frequent, then creating child nodes for n_X by joining n_X and its frequent right siblings.

Figure 1 shows examples of construction and updates on the basic OPFI-tree, where $\Sigma = \{a, b, c, d\}$, $N = 4$, and $minsup = 0.5$. Let's take the constructed tree as an example, $n_{\{ab\}}$ is frequent and has a frequent right sibling $n_{\{ac\}}$, then a child node $n_{\{abc\}}$ is created for $n_{\{ab\}}$. We use $child(n_X)$ to denote the set of child nodes for n_X. However, when n_X and its right sibling n_Y are frequent, child $n_{\{X \cup Y\}}$ may be infrequent even with small frequency. The optimistic pruning strategy is designed to prune most of these infrequent nodes in $child(n_X)$.

4.2 The Optimistic Pruning Strategy

Although the basic OPFI-tree only maintains a selected part of infrequent itemsets in the sliding window, the number of infrequent itemsets it maintains is still

very large. So the optimistic pruning strategy is proposed to prune infrequent nodes with small frequencies. It is based on two observations:

Table 1. Number of Itemsets with Specified Frequencies

	1	2	3	4	5	6-10	11-15	16-20	21-30	31-50
T10I4D100K[a]	??	480202	73543	24525	10480	25235	8597	6724	7933	3058
BMS-WebView-1[a]	??	35292	2333	648	251	327	72	20	23	7

[a] Using 10000 transactions on T10I4D100K and 1000 on BMS-WebView-1.

- There are a large number of infrequent itemsets. The number of itemsets with small frequency is much bigger than the number of itemsets with large frequency. As be shown in table 1, as the frequency increases, the number of itemsets drops dramatically. There is an error when frequency is 1, because the number of itemsets with frequency 1 is too large to compute on our test machine.
- Itemsets do not often change their status dramatically, so we can optimistically think that an infrequent itemset with small frequency will seldom change its status to frequent.

In order to distinguish these infrequent nodes, OPFI-stream adds a new property, called $slackMinsup$, such that $0 \leq slackMinsup \leq minsup$. When creating a new child node in the basic OPFI-tree, if its support is smaller than $slackMinsup$, then it will be discarded. Obviously, all infrequent nodes will be pruned when $slackMinsup = minsup$, and no special pruning is done under the basic OPFI-tree when $slackMinsup = 0$. Although the maximal pruning is achieved when $slackMinsup = minsup$, it is not acceptable because some infrequent nodes with high support may change to frequent after some slides. Frequently adding the previously pruned nodes will degrade the update performance. So there is a compromise between the optimistic pruning strategy and the update performance. We will find in the experiments that what is the best value for $slackMinsup$ to balance the pruning and performance.

We use $opset(n_X)$ to denote the infrequent itemsets pruned by the optimistic pruning strategy under frequent node n_X. Then $estMaxsup$ property is added for n_X to estimate the maximal support of nodes in $opset(n_X)$. There are three operations on $estMaxsup$:

1. Initialization. It happens when the basic OPFI-tree creates subtree for the node n_X which is either newly created or originally infrequent. We use $child(n_X)$ to denote the direct children of n_X in the basic OP-tree, then we have

$$opset(n_X) = \{n_{X'} \mid n_{X'} \in child(n_X), sup(n_{X'}) < slackMinsup\} \quad (1)$$

Now the $estMaxsup$ is initialized as

$$estMaxsup = max\{sup(n_{X'}) \mid n_{X'} \in opset(n_X)\} \tag{2}$$

If there is no child node pruned by the optimistic pruning strategy, we have $opset(n_X) = \phi$ and $estMaxsup = 0$.

2. Update. There are three cases for the update operation:
   (a) If a child node of n_X is going to be added in the basic OPFI-tree but pruned by the optimistic pruning, and it has support bigger than $estMaxsup$, then update $estMaxsup$ as the support.
   (b) When n_X increases its support, $estMaxsup$ will also be increased. But $estMaxsup$ will not change if n_X decreases its support.
   (c) If a node's support decreases to be smaller than the $slackMinsup$, then the node is pruned and its parent's $estMaxsup$ is updated.
3. Re-Computation. When $estMaxsup$ increases to $minsup$, which means that some child nodes pruned ago maybe become frequent now, then the re-compute operation occurs. The algorithm will scan all its possible child nodes, and create new child nodes which have support bigger than $slackMinsup$, so the left child nodes will constitute the new $opset(n_X)$. $estMaxsup$ will also be update during the scan.

$estMaxsup$ will always meet the following requirement whatever operations occur:

$$estMaxsup \geq max\{sup(n_{X'}) \mid n_{X'} \in opset(n_X)\} \tag{3}$$

Equation 3 means that $estMaxsup$ will be never smaller than the maximal support of the child nodes pruned by the optimistic pruning. The algorithm can catch the moment that nodes in the *opset* become frequent, so no frequent itemset will be lost in the sliding window streams. But it is also possible that no new child node is added after the re-computation. Briefly, pruning infrequent nodes is optimistic but updating $estMaxsup$ is conservative.

Correctness Analysis

The most important thing in the optimistic pruning strategy is to maintain $estMaxsup$. Now we will prove that equation 3 holds.

Proof. We only need to prove that equation 3 holds after the three operations: initialization, update and re-computation.

When $estMaxsup$ is initialized or re-computed, it is set to the maximal support of nodes in the *opset*, so equation 3 holds obviously.

When $estMaxsup$ is updated after the initialization and re-computation, there are four cases to discuss.

1. If a child node of n_X is going to be added in the basic OPFI-tree but pruned by the optimistic pruning, and it has support bigger than $estMaxsup$, then $estMaxsup$ is updated as the support. Obviously, equation 3 holds.

2. When $sup(n_X)$ increases, nodes in $opset(n_X)$ may increase their support or remain the same. So increasing $estMaxsup$ will still make the equation 3 hold.
3. When $sup(n_X)$ decreases, nodes in $opset(n_X)$ may decrease their support or remain the same, and whether the node with the maximal support in $optset(n_X)$ decreases its support can't be determined. So $estMaxsup$ needs to remain the same, and equation 3 holds.
4. If a node's support decreases to be smaller than $slackMinsup$, then the node is pruned and added into *opset* of its parent. The parent's $estMaxsup$ is updated as the maximal support of nodes in *opset*, so equation 3 also holds.

Then we can prove that no frequent node is lost by the optimistic pruning strategy.

Theorem 1 *The optimistic pruning strategy does not lose any frequent nodes in the basic OPFI-tree.*

Proof. The proof can be divided into two steps:

Step 1. There is no frequent nodes pruned in the optimistic pruning process. The optimistic pruning strategy only prunes infrequent nodes with support smaller than $slackMinsup$, so there is no frequent node to be pruned.

Step 2. The algorithm can monitor the status changes of the infrequent nodes pruned by the optimistic pruning strategy. According to equation 3, when some nodes in *opset* change to be frequent, we have $estMaxsup \geq minsup$, so the re-computation operation will occur.

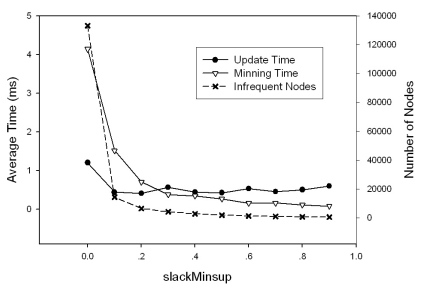
5 Experimental Results

We compare the performance of OPFI-stream against Moment [12], which returns accurate recent closed frequent itemsets in data streams. The experiments are done on a 2.8G Hz processor with 1G MB memory, running Windows 2003. In the experiments, the size of the sliding windows is fixed to 50,000, and the user-specified minimum support threshold is set from 0.1% to 1%. Under these settings, we will compare the performance of running time and maximal memory usage.

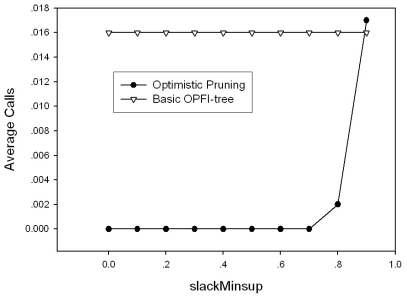
The first test dataset is a synthetic dataset, called T10I4D100K. It is generated using the synthetic data generator implemented by Agrawal et al. in [2]. T10I4D100K means that the average size of transaction is 10, the average size of maximal potentially frequent itemsets is 4, the total number of transactions in the dataset is 100,000. For simplicity in the following figures, we will use x to denote the value of $slackMinsup$ when $slackMinsup = x * minsup$.

In the first experiment, we will vary $slackMinsup$ from 0 to $0.9 * minsup$ when $minsup = 0.6\%$, to compare the average update time and mining time, the number of infrequent nodes in the tree. In order to see whether the optimistic pruning strategy is effective, we will compare the call times of two types of

BuildTree. The BuildTree described in [13] is used to construct the subtree under a frequent node. The first type is invoked when an infrequent node changes to frequent, and the second type is invoked in the re-computation step when $estMaxup$ increases to $minsup$.



(a) Average Update and Mining Time / Infrequent Nodes



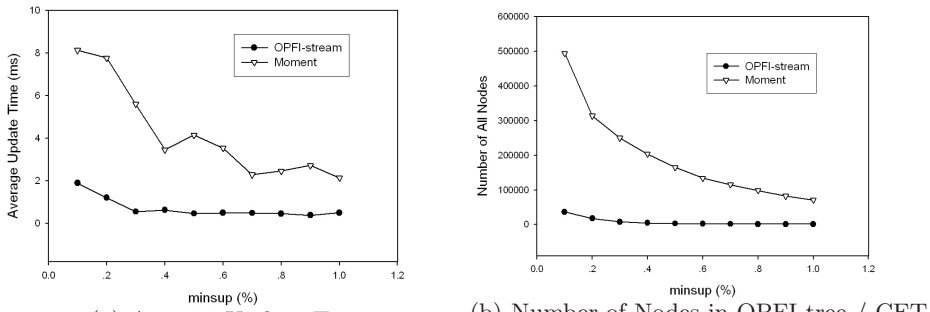
(b) Number of BuildTree Calls

Fig. 2. Performance of OPFI-stream on T10I4D100K when $slackMinsup$ Changes

Figure 2 shows the average update, mining time and number of infrequent nodes for OPFI-stream over 1000 sliding windows under different $slackMinsup$. As $slackMinsup$ increases in figure 2(a), the number of infrequent nodes in the OPFI-tree decreases, especially when $slackMinsup$ changes from 0 to $0.1 * minsup$, the number drops dramatically. It means that most itemsets have low frequencies. This also implies that the optimistic pruning strategy is very efficient to prune infrequent nodes. The average update and mining time decrease when $slackMinsup$ increases. The trend of mining time is similar to that of infrequent nodes, because mining in OPFI-stream is just to traverse the tree and the mining time depends on the total number of nodes in the tree. When $slackMinsup = 0$, the update time is much more than the rest, because the OPFI-tree is much bigger when $slackMinsup = 0$ and becomes the dominant factor. As $slackMinsup$ increases more than $0.6 * minsup$, the update time fluctuates. This is because that most infrequent nodes are pruned by the optimistic pruning strategy, but some of the nodes have support very close to $minsup$ and will be added soon. Figure 2(b) shows the number of calls to BuildTree when $slackMinsup$ changes. The first type of calls to BuildTree is not affected by the $slackMinsup$, and the calls caused by the optimistic pruning is few when $slackMinsup$ is small, but increases when $slackMinsup$ increases more than $0.8 * minsup$. This is the same trend like that of the average mining time as shown in figure 2(a). We can see that the number of BuildTree calls caused by the optimistic pruning is a little bigger when $slackMinsup = 0.9 * minsup$, this is because some infrequent nodes with support close to $minsup$ are also pruned but may change to frequent after some slides. Although the BuildTree needs to scan all transactions in the sliding

window, it can be seen from the experiments that the number of calls to it is very small, so the performance will be affected only slightly due to the BuildTree calls.

Then we will compare the update time and maximal memory usage for OPFI-stream and Moment. Mining in the Moment algorithm is to traverse an extra hash table and return closed frequent itemsets, so there is additional work to get all frequent itemsets from the closed frequent itemsets. So the mining time comparison is omitted here. OPFI-stream has two main data structures: OPFI-tree and bit-sequences, while the latter is determined by the number of items and the size of the sliding window. Moment also has two main data structures: CET and FP-tree, while the latter is used to store all transactions in the sliding window. In most cases, the memory usage of the bit-sequences in OPFI-stream is much smaller than that of the FP-tree in Moment. But they are just assistant data structures, the details are omitted here. So we will compare the number of all nodes in OPFI-tree and CET instead of maximal memory usage. From the first experiment, we will make a tradeoff between running time and space usage, so $slackMinsup$ is set to be $0.6 * minsup$.

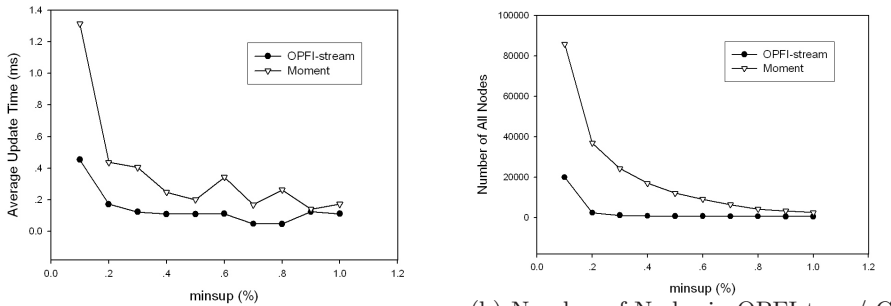


(a) Average Update Time (b) Number of Nodes in OPFI-tree / CET

Fig. 3. Performance Comparison on T10I4D100K when *minsup* Changes

Figure 3 shows the performance comparison of the OPFI-stream and Moment algorithms over 1000 sliding windows under different *minsup*. We can see that both the average update time and the number of all node in OPFI-tree are stable when *minsup* increases and much less than that of the Moment algorithm. When *minsup* is small, the number of nodes in OPFI-tree can be less than 2% of the number of nodes in CET. This is because that most infrequent nodes are pruned by the optimistic pruning strategy, which results in a very small OPFI-tree. Furthermore, the bit-sequence representation of the items in the sliding window is also very efficient in operations such as deleting old transaction, adding new transaction and counting frequencies for itemsets. So the average update time of OPFI-stream is much smaller compared to that of the Moment algorithm.

The second test dataset is a real dataset, called BMS-Webview-1, which contains several months of click stream data from an e-commerce web site. This dataset was used in KDDCUP 2000 [14]. In BMS-Webview-1, there are 59,602 transactions, 497 distinct items, and the maximal transaction size, the average transaction size are 267 and 2.5 respectively.



(a) Average Update Time (b) Number of Nodes in OPFI-tree / CET

Fig. 4. Performance Comparison on BMS-Webview-1 when *minsup* Changes

Figure 4(a) and 4(b) show the results on BMS-Webview-1. We can see that the trends in these two figures are nearly the same as that in figure 3(a) and 3(b). Both the time and space usage in the two algorithms for BSM-Webview-1 are smaller than that of T10I4D100K. This is because the average transaction size is smaller than that of the previous synthetic dataset.

From the experiments on both synthetic and real dataset, we can see that both the time and space performances of OPFI-stream outperform Moment. OPFI-stream has a good and stable performance even when *minsup* is small. Moment's performance is close to OPFI-stream when *minsup* is relative large, but degrades dramatically when *minsup* decreases smaller than 0.5%.

6 Conclusions

In this paper, an OPFI-stream algorithm is proposed to mine all accurate recent frequent itemsets in data streams. It uses a highly compact OPFI-tree to maintain all frequent itemsets and their accurate frequencies. Experimental studies show that the optimistic pruning strategy has pruned most infrequent nodes during the construction and updates, and the OPFI-tree is much smaller than the CET in the Moment algorithm. Besides the smaller memory usage, the experiments also show that it runs significant faster than the Moment algorithm. In the future, we will focus on adding batch updates in our proposed algorithm to improve the performance further.

References

1. Jiang, N., Gruenwald, L.: Research issues in data stream association rule mining. SIGMOD Record **35**(1) (2006) 14–19
2. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: VLDB '94: Proceedings of the 20th international conference on Very Large Data Bases, MorganKaufmann (1994) 487–499
3. Agrawal, R., Gehrke, J., Gunopulos, D., Raghavan, P.: Automatic subspace clustering of high dimensional data for data mining applications. In: SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM (1998) 94–105
4. Wang, K., Zhou, S., Liew, S.C.: Building hierarchical classifiers using class proximity. In: VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1999) 363–374
5. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM (2000) 1–12
6. Zaki, M.J.: Scalable algorithms for association mining. IEEE Transactions on Knowledge and Data Engineering **12**(3) (2000) 372–390
7. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases, VLDB Endowment (2002) 346–357
8. Giannella, C., Han, J., Pei, J., Yan, X., Yu, P.S.: Mining frequent patterns in data streams at multiple time granularities. In: In Proceedings of the NSF Workshop on Next Generation Data Mining. (2002)
9. Gaber, M.M., Krishnaswamy, S., Zaslavsky, A.: On-board Mining of Data Streams in Sensor Networks. In: Advanced Methods for Knowledge Discovery from Complex Data. Springer Berlin Heidelberg (2006) 307–335
10. Chang, J.H., Lee, W.S.: estwin: adaptively monitoring the recent change of frequent itemsets over online data streams. In: CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management, New York, NY, USA, ACM (2003) 536–539
11. Cheng, J., Ke, Y., Ng, W.: Maintaining frequent itemsets over high-speed data streams. In: PAKDD'06: Advances in Knowledge Discovery and Data Mining, 10th Pacific-Asia Conference. (2006) 462–467
12. Chi, Y., Wang, H., Yu, P.S., Muntz, R.R.: Moment: Maintaining closed frequent itemsets over a stream sliding window. In: ICDM '04: Proceedings of the 4th IEEE International Conference on Data Mining, Washington, DC, USA, IEEE Computer Society (2004) 59–66
13. Li, K., yan Wang, Y., Ellahi, M., an Wang, H.: Mining recent frequent itemsets in data streams. In: The 5th International Conference on Fuzzy Systems and Knowledge Discovery. (2008)
14. Zheng, Z., Kohavi, R., Mason, L.: Real world performance of association rule algorithms. In Provost, F., Srikant, R., eds.: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. (2001) 401–406