An Approximate Scheme to Mine Frequent Patterns over Data Streams

Shanchan Wu

Department of Computer Science,
University of Maryland, College Park, MD 20742, USA
wsc@cs.umd.edu

Abstract. In this paper, we propose a scheme to mine frequent patterns from data streams. Our scheme can guarantee to run mining algorithm in the limited memory capacity when the data becomes large enough, and to keep the running time at the regular range when the time evolves. We emphasize the most recent data but do not discard all of the historical data. We propose efficient sampling and merging schemes to process incoming data streams quickly. We implement the schemes and mining frequent patterns algorithm. The analysis and experiments show that our algorithms are efficient.

1. Introduction

Frequent pattern mining in datasets has been studied extensively in data mining, with many algorithms proposed and implemented, For example Apriori[1], FP-growth[2], CLOSET[3], and CHARM[4]. Frequent pattern mining and its associated methods have been popularly used in association rule mining, sequential pattern mining, structured pattern mining, associative classification, frequent pattern-based clustering, and so on.

Recent emerging application, such as network traffic analysis, web click stream mining, power consumption measurement, sensor network data analysis, and dynamic tracing of stock fluctuation, call for study of a new kind of data, called stream data, where data takes the form of continuous, potentially infinite data streams, as opposed to finite, statically stored data set. However, it is challenging to mine frequent patterns in data streams because of the large and continuous data. Mining frequent patterns in data stream is a challenging job, much more difficult than mining frequent patterns in datasets. There are some reasons: (1) the data stream is potentially infinite, so we can't simply abandon some data just like dealing with datasets; (2) As the memory capacity of computers is finite, we can't keep all of the historical data stream; (3) We sometimes need timely answer and the response time should be small. These reasons make finding frequent patterns in data steam a hard job. Though some work has been done in this area, it is far less than sufficient.

There are some recent studies on mining data streams. Giannella et al. [5] have developed a frequent itemsets mining algorithm over data stream. They have proposed the use of tilted windows to calculate the frequent patterns for the most recent transactions based on the fact that users are more interested in the most recent

transactions. They use an incremental algorithm to maintain the FP-stream which is a tree data structure to represent the frequent itemsets. They conducted a number of experiments to prove the algorithm efficiency.

Manku and Motwani [6] have proposed and implemented an approximate frequency counts in data streams. The implemented algorithm uses all the previous historical data to calculate the frequent patterns incrementally.

P.A. Laur et al. [7] have proposed a statistical technique which biases the estimation of the support of sequential patterns or the recall, as chosen by the user, and limits the degradation of the other criterion. They replaced the conventional minimal support requirement for finding frequent sequential patterns by a statistical support.

Gaber, M, M. et al [8] reviewed the theoretical foundations of data stream analysis, and have critically reviewed mining data stream systems and techniques. They outlined and discussed research problems in streaming mining field of study.

Most of these previous work on mining frequent or sequential patterns over data streams focuses on how to technically deal with large historical data, but do not guarantee that they can run their algorithm in the limited memory capacity when the data becomes large enough, or do not keep the running time at the regular range when the time evolves. In this project, we will propose a method to mine frequent patterns with about the same cost time when the time evolves and the historical data gets large. We refer the ideas in [5] and [7]. Similar to [5] we focus more on most recent data, and like [7] we use some sample idea, but not the same technique. [5] uses tilted windows to focus most recent data. [7] uses statistical technique to change the minimal support.

We suppose to keep the data which we will mine on about the same size and on the other hand we try to avoid wasting too much time on sampling to keep the data around the same size. This will guarantee that we are able to run the mining algorithm with the limited memory capacity. And if keeping the data around the same size costs small time, then the total cost time mostly depends on the mining algorithm. One simple idea is that every time we use the most recent data, but it will lose the rest historical data. Another idea is that every time we sample the whole historical data, but this will cost a lot of time.

Suppose we want to keep the data with size S, from which we will find frequent patterns. Apparently we have the equation:

$S = S/2 + S/4 + S/8 + S/16 + \ldots + S/2^n \ldots + \ldots$

The idea is that we use the S/2 most recent items in the historical data without sampling, and sample half for the next S/2 items, and sample 1/4 for the next to the next S/2 items, again an again till the most original items. We will extend this equation to more generality, where the fading factor can be some value other than 1/2. We will discuss it more detail in section 3.

We will design an efficient mechanism to quickly sample data in this way, and combine this algorithm to the mining algorithm. FP-growth [2] is an efficient algorithm to mine frequent patterns. We use the FP-growth to mine frequent patterns. Mining frequent patterns will be discussed in section 5. How the properties of data

stream will affect our algorithm will be discussed in section 6. In that section make some hypotheses which will be tested in section 8.

We propose two methods in section 4 to implement the sampling and merging mechanism. One is to maintain the sampled historical data in hard disk, the other is to store the sampled historical data in a compact FP-tree and keep it in main memory. We will combine the sampling algorithm and mining algorithm. When new data streams come, we update the data and if necessary run the sampling algorithm. In this way the mining results reflects the most recent data and also historical data with less priority. We will talk about the mechanism when and how we run sample algorithm later. We do some experiments and analyze the performance.

2. Problem Definition

Let $I = \{a_1, a_2, a_3, \cdots a_m\}$ be a set of items, and a transaction database $DB = \{T_1, T_1, \cdots, T_n\}$, where $T_i (i \in [1 \cdots n])$ is a transaction which contains a set of items in I. The support (or occurrence frequency) of a pattern A, where A is a set of items, is the number of transactions containing A in DB. A pattern is frequent if A's support is no less than a predefined minimum support threshold ξ. Suppose $\theta = \frac{\xi}{|DB|}$, then a pattern is frequent if $support(A) \geq \theta \cdot |DB|$. $\theta \in (0,1)$ is a user specified minimum support. The problem of mining frequent patterns is to mine all patterns whose support is greater than, or equal to $\theta \cdot |DB|$. Each of them is called a frequent pattern.

In the case of data stream we have a partial storage of DB. Let us now assume that we are provided with a data stream. Let $DS = B_0, B_1, \cdots, B_n, \cdots,$ be an infinite sequence of batches, where each batch is associated with a timestamp t, i.e. B_t, and n is the identifier of the "Latest" batch B_n. Each batch B_i consists of a set of customer data transactions; that is $B_i = [T_1, T_2, \cdots, T_k]$. The length (L) of the data stream is defined as $L = |B_0| + |B_1| + \cdots + |B_n|$, where $|B_i|$ stands for the cardinality of the set B_i. Therefore, the problem of mining frequent patterns is to find frequent patterns A, verifying $\sum_{t=0}^{L} support_t(T) \geq \theta \times L$

As the data stream is potentially infinite, it is possible that some patterns which

might be frequent at some time but might not be frequent patterns again when time evolves, and it is also possible that some patterns which might not be frequent at some time but might be frequent patterns some time later. In the case the data stream update quickly, the mining frequent patterns algorithm which cost a lot of time are not practical because the algorithms do not get the most recent frequent patterns. During the long time execution of the algorithm, the data stream has been updated greatly. To limit the running time, one straightforward thinking is that we do not apply the mining algorithm to the whole historical data but rather to some sample data, and what we get is not the truly frequent patterns, but the estimations of the frequent patterns. Some new problems come up:

1. The sampling procedure should not cost lots of time when applied to potentially infinite data streams. If it costs lots of time, then it is useless.
2. How can the frequent patterns from the sampling data be a good estimation of the truly frequent patterns?

It is generally impossible to make a fully accurate prediction when some information is missing from the original data. And we can notice the different importance of the historical data with different time. For example, a shopping transaction stream could start long time ago, but some old items may have lost their attraction and fashion or seasonal products may change from time to time. In this paper we address how to make efficient sampling with balancing the importance of current data and without loss the historical information to mine frequent patterns from data stream.

3. Mining Data Stream with Limited Memory

Suppose the original mining algorithm can mine the data set with the size D, consuming the time and memory space under our tolerance. Then our target is to limit the data size not greater than and near to D, on which we run the mining algorithm. We introduce a fading factor λ, where $0<\lambda<1$. For the most recent M transactions, we use the original transactions; for the next older M transactions, we sample λM; and so on. Then we have:

$$D = M + \lambda M + \lambda^2 M + \cdots + \lambda^i M + \cdots \lambda^n M = \frac{M(1-\lambda^n)}{1-\lambda}$$

If the data stream is infinite, i.e. $n \to +\infty$, then $M = \frac{(1-\lambda)D}{1-\lambda^n} \to (1-\lambda)D$

We apply $M = (1-\lambda)D$ as a window size in which we do the same dense sampling. Let $DS = B_0, B_1, \cdots, B_n, \cdots,$ be a data stream, where each batch B_i consists of a set of customer data transactions. Suppose we begin running processing data stream as the data stream starts. The processing of the data stream is as follow:

Input: *Data stream* $DS = B_0, B_1, \cdots, B_n, \cdots;$

fading factor λ *; window size* M

Initialize: $D_t = \phi$, $D_{t1} = \phi$

$S_{t1} = 0$

ProcessDataStream (DS)
{

waiting next new coming batch B_i *and suppose* $B_i = [T_1, T_2, \cdots, T_k]$

for each transaction T_j *in* B_i

{

$S_{t1} = S_{t1} + 1$; $D_{t1} = D_{t1} \cup T_j$

if($S_{t1} = M$ *) then{*

$D_t = MergeAndSample(D_t, D_{t1}, \lambda)$; $S_{t1} = 0$; $D_{t1} = \phi$

}

}

ProcessDataStream (DS)

}

In the *ProcessDataStream(DS)*, for every new transaction, we add it to the D_{t1}, and increase S_{t1}, the size of D_{t1}. If S_{t1} is less than window size M, we do nothing and turn to next transaction. Else, We merge and sample D_{t1} and D_t, and then set the result to D_t.

We have two ways to implement $MergeAndSample(D_t, D_{t1}, \lambda)$. We can first merge D_{t1} to D_t and then do sampling, or we can first sample D_{t1} and D_t separately and then merge them. We will talk more about this in next section.

Every time we run mining algorithm, we use both D_{t1} and D_t as data source. So our results will reflect both the most recent data and the historical data.

4. Sampling and Merging

In this section, we talk about the scheme of sampling and merging. We have two choices. One is that we keep the sampled historical data and the most recent data in

the disk, the other is that we keep the sampled historical data in the main memory using a compact tree structure.

When we use disk to store both parts of data, the merging operation is just combine to parts of data sets together. After merging, we do sampling operation on the merged file. The sampling operation is as following:

Algorithm (Disk-Sample: Sampling transaction file with possibility λ *)*
Input: transaction file Tran_file, and sampling density λ
Output: The sampled Tran_file
Method: Call Disk-Sample (FP-tree)
Procedure Disk-Sample(Tran_file, λ *)*
{
create an empty file tmp_file;
move the file pointer forward, get sufficient lines of data until get a complete transaction t. For every such transaction t select t with a random possibility λ *.*
If t is selected, then push t into a buffer. When the buffer is almost full, flush the buffer data into tmp_file. After finishing scanning Tran_file, flush the buffer again, remove Tran_file and rename tmp_file to Tran_file.
return Tran_file.
}

The advantage of storing the data in the disk is that it can store much more data than in memory and the operation is simple. The disadvantage is that it costs more time because of the IO operations.

We can use FP-tree [2] to store the sampled historical data. Our goal is to mine the frequent patterns from the sampled historical data and the most recent data. If we store all of the sampled historical data in the memory, this goal can be met. However, this will require too much space. So we only maintain those items in the historical transactions whose frequency is greater than some error threshold ε, and discard the other items. As a result, we no longer have exact frequency over the sampled historical data plus most recent data, rather than an approximate frequency. for any itemset I, $\hat{f}_I = f_I(D_t) + f_I(D_{t1})$, where $f_I(D_t)$ is the frequency of the sampled historical data and $f_I(D_{t1})$ is the frequency of the most recent data. The approximation is less than the actual frequency as described by the following inequality,

$$f_I - \varepsilon \cdot \lambda D \le \hat{f}_I \le f_I$$

Thus if we deliver all itemsets whose approximate frequency is larger than $(\delta - \varepsilon\lambda)D$, we will not miss any frequency itemsets in the sampled historical data and the most recent data. We use an example to describe the idea and propose the algorithm during

the discussion of the example.

Example. Let D=10, $\lambda = 0.5$, then M=5. Suppose error threshold $\varepsilon = 0.3$, then only those items having frequency at least 2 in the sampled historical data set are stored in the frequency tree. At sometime the sampled data is as following table:

Table1. Historical Sampled Transaction Datasets

Transaction ID	Items	(Ordered) items with frequency >= 2
1	f,a,c,d,g,k,i,m	f,c,a,m,k
2	a,b,c,f,m	f,c,a,b,m
3	b,f,h,j,k	f,b,k
4	b,c,o,s,p	c,b
5	a,f,c,e,l,m,n	f,c,a,m

Then FP-tree with respect to Table 1 is as Figure 1.

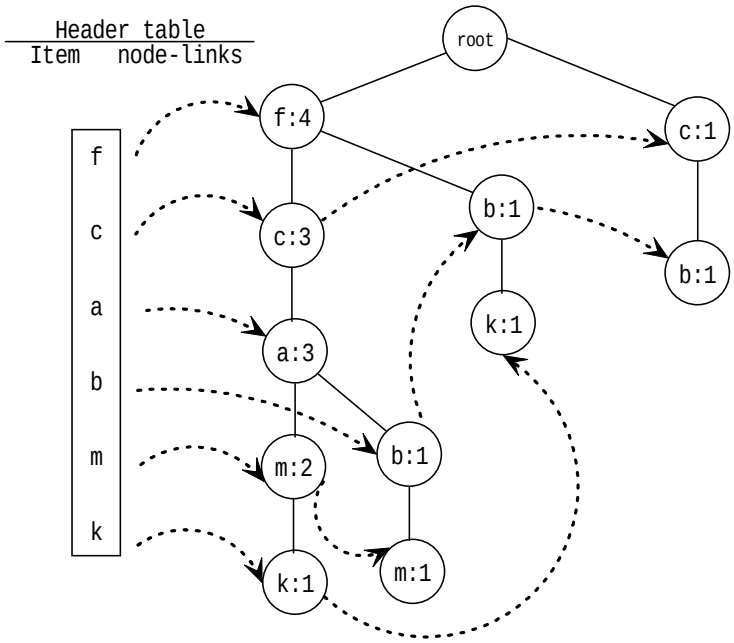

Figure 1. FP-Tree constructed from Table 1

After sometime, when new transactions come and the number of the new transactions hits M, then we should do the merge and sampling operation. Suppose some time later the accumulated new transactions are as Table 2. The number of new transactions has hit M where M is equal to 5.

Table2. New Transaction Data Sets.

Transaction ID	Items
6	f,a,c,d,g,k,i,m
7	a,b,c,f,j
8	b,f,h,j,k
9	b,f,c,j,s,p
10	a,f,c,e,l,m,n

As the historical data is stored FP-tree, the efficient operation is to first sample

FP-tree and new transaction data sets. Then merge them together. Sampling new transaction data sets is simple. We give every transaction the possibility of λ to be selected, in this example $\lambda = 0.5$. Now we give the algorithm of sampling FP-tree.

Algorithm (FP-Sample: Sampling FP-tree with possibility λ)
Input: FP-tree, and sampling density λ
Output: The new sampled FP-tree
Method: Call FP-Sample(FP-tree)
Procedure FP-Sample(Tree, λ)
{
for every node in the header table do{
Traverse its node-list. For every node V in the node-list,
$s = support(V) - \sum_{V_1 \in \text{children of } V} support(V_1)$ *if(* $s > 0$ *) then{*

$s1 = \sum_{i=1}^{s}$ (random with possibility λ to return 1 and 1-λ to return 0) ,

$\Delta s = s - s1$,
traverse node V back to root, deduct the support of every node by Δs,
if the support of any node during the traversing becomes to 0, then remove the node from the tree.
}
}
return Tree
}

Using FP-Sampling Algorithm, FP-tree is changed as showing in figure 2 (this is one possibility and there are some other possibilities of the new FP-tree).

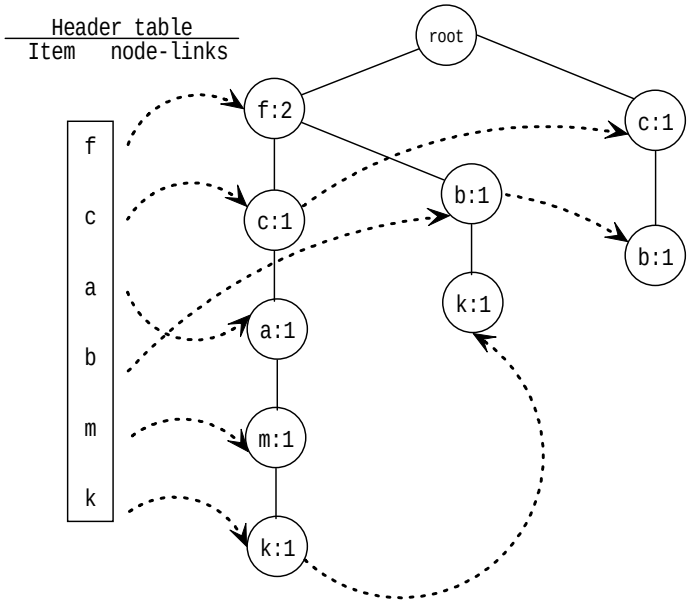

Figure 2. FP-tree after Sampling the FP-tree in figure 1.

Now we sample the recent data in table 2, we have table 3 (there are also some other possibilities).

Table3. New Transaction Data Sets After Being Sampled.

Transaction ID	Item set
7	a,b,c,f,j
9	b,f,c,j,s,p

Merging the dataset in Figure 2 and Table 3 only needs to scan the FP-tree and table3 twice. During the first scan, calculate the frequency of every item and discard those items whose frequency is less than 2. During the second scan, build up the new FP-tree and destroy the old FP-tree. The new FP-tree in figure 3 actually based on table 4, thought we do not actually maintain such a table.

Table4. Virtual Transaction Datasets About Figure 3

Transaction ID	Items	(Ordered) items with frequency >= 2
1	f, c, a, m, k	f, c, a, k
3	f, b, k	f, b, k
4	c, b	c, b
7	a, b, c, f, j	f, c, b, a, j
9	b, f, c, j, s, p	f, c, b, j

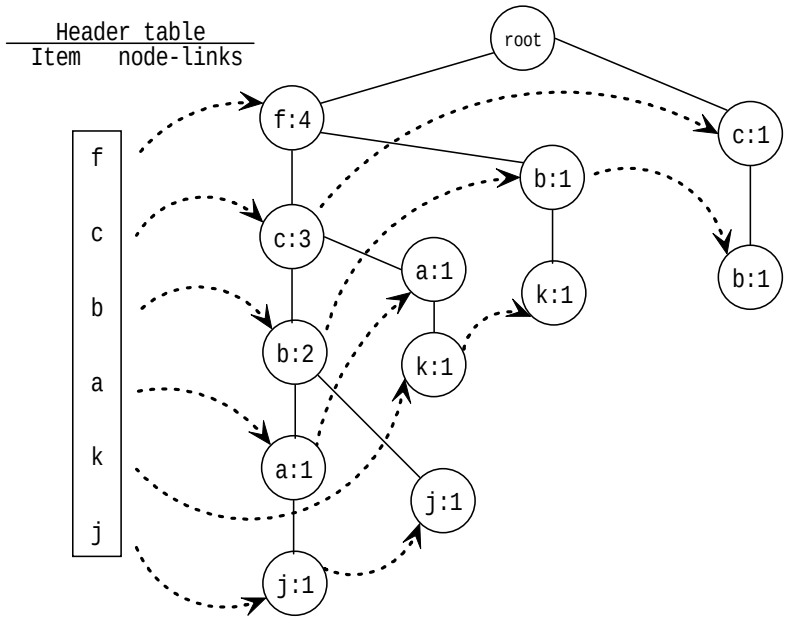

Figure 3. FP-tree After Merging FP-tree in Figure 2 and Data Sets in Table3.

5. Mining Frequent Patterns

We run the mining algorithms on the sampled data and most recent data.

To find frequent patterns over data streams, we examine the same problem in a

transaction database. To justify whether a single item a_i is frequent in a transaction database DB, one just need to scan the database once to count the number of transactions that a_i appears. One can count every single item a_i in one scan of DB. However, it is too costly to count every possible combination of single items. An efficient alternative proposed in the Apriori algorithm [1] is to count only those itemsets whose every proper subset is frequent. That is, at the k-th scan of DB, derive its frequent itemset of length k, and then derive the set of length (k+1) candidate itemset (i.e., whose every length k subset is frequent) for the next generation.

Han J. at al in [2] proposed a method that may avoid candidate generation-and-test and utilize a compact data structure called frequent-pattern tree, or FP-tree in short. To ensure the tree structure is compact and informative, only frequent length-1 items will have nodes in the tree, and the tree nodes are arranged in such a way that more frequently occurring nodes will have better chance of node sharing than less frequently occurring ones. Subsequent frequent-pattern mining will only need to work on the FP-tree instead of the whole data set.

We use the FP-growth [2] algorithm to mine frequent patterns from the sampled historical data and the most recent data. Next section we will talk about the properties of transaction data stream. We will discuss the importance of the properties and make some hypotheses which will be tested in section 7.

6. Properties of Transaction Data Stream

What will the properties of the data stream affect our sampling and merging algorithm? And how?

There are some properties of the data stream and data set which are very important and inevitably affect the time cost in data mining procedure. Apparently the size of the data size will greatly affect the cost time and this has been experimented by many data mining algorithms. We have tested the effect of the historical data size to our sampling and merging algorithm. And we showed that the sampling and merging algorithm based on FP-tree has a very good performance with different historical data size. Now we need to consider other properties of the data which will affect the performance of the algorithm, the most important of which are the number of distinct items and the average length of transactions. In some literature, these properties are omitted. For example, in [5] in the experiment they only used the transactions data with short average length, which is 3. Actually, short length transactions will cost much less time to construct and maintain the structure they used in their algorithm. When the length changes longer, it will increase the cost time. How these properties affect the algorithm is depended on the nature of the algorithm. An ideal algorithm should be robust with changing of these properties. Actually it is hard to reach such ideal situation, but we still seek to reduce their effect as much as possible. The performance of the sampling and merging algorithm based on disk, from the nature of the algorithm, will not be affected by these properties or very little. But as they have

worse performance than the algorithm based on FP-tree which is constructed in memory, and intuitively the FP-tree based algorithm will be affected by these properties, we need to test how these properties affect the performance.

Hypothesis 1. When the number of distinct items increases, the cost time will increase less sharply than linear function.

Hypothesis 2. When the average length of transactions increases, the cost time will increase less sharply than linear function.

We will do experiments to test the above two hypotheses. For Hypothesis 1, we change the number of distinct items and keep other properties unchanged, and test the different time cost; For Hypothesis 2, we change the average length of transactions and keep other properties unchanged, and test the different time cost.

If the hypotheses are falsified, in other words, the cost time increases more sharply than the linear function, we need to modify our algorithm. Especially when it increases exponentially, we need to reconstruct the data structure and redesign the algorithm. In reality, usually there are long transactions and large number of distinct items. We need to consider such situation and cannot just think of the situation where there are only short transactions and small number of distinct items. For example, in the supermarket, there are lots of items of goods, thousands of or even ten thousands of.

7. Performance Study and Experiments

Our experiment was written in C++, using Microsoft Visual C++2005. See Appendix to get a snapshot of the demo.

The data stream was generated by the IBM synthetic market-basket data generator, available at "http://www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/datasets/syndata.html". The stream was broken into batches of transactions and fed into the program through standard input.

. We implement the FP-tree mining algorithm. The demo has Multithreads, GUI thread for Setting parameters and running mining operations; cmd thread for inputting and processing data stream and for standard output.

We do the experiments to test the performance. When the number of the most recent transactions does not hit the window size M, no sample and merge operations take place, and one batch of transaction with 56k file size costs average of 1.4 second to process. The time cost for processing the data stream become larger when the window size hit M and sample and merge operations take place. We compare two types of sample and merge operations, one is disk based, the other is based FP-tree. We generate the test data with average transaction length of 10. We use the same most recent data and different size of historical data to compare the performance.

Most recent data: Trans num, 1k; file size 275k.
Error threshold $\varepsilon = 0.0005$. Number of distinct items: 100.

Table5. Properties of Test Data

Historical data	data1	data2	data3	data4	data5	data6	data7	data8
Trans num	1k	2k	5k	10k	15k	20k	25k	30k
File size	275k	487k	1213k	2774k	4086k	5518k	6837k	8173k
Fading factor λ	0.5	0.667	0.8	0.9	0.933	0.95	0.96	0.967

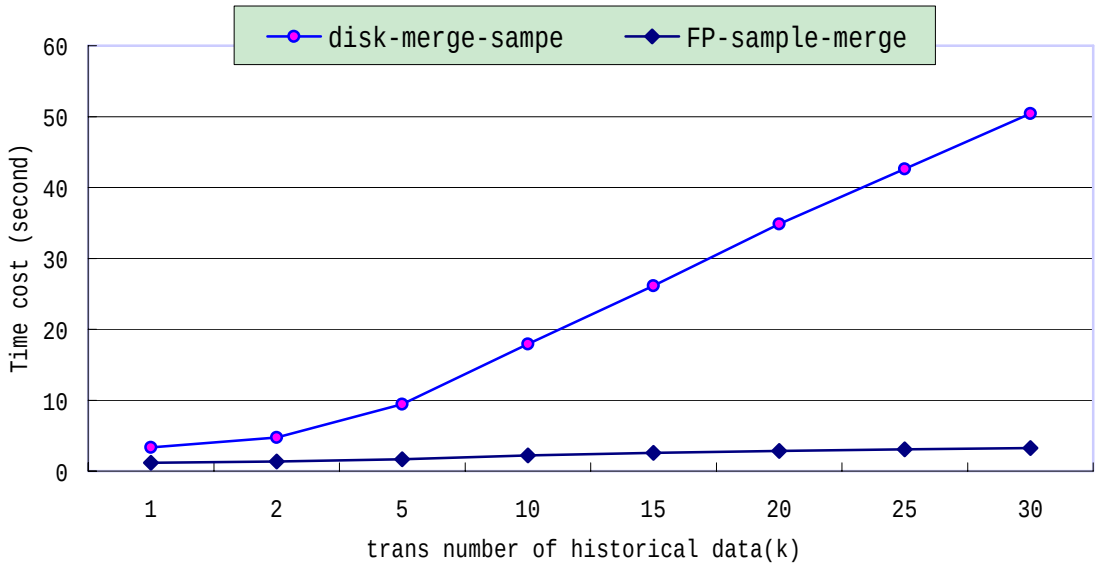

Figure 4. Compare the Performance of the Two Types of $MergeAndSample(D_t, D_{t1}, \lambda)$

From Figure4, we found that FP-tree based merge and sample is much more efficient when the historical data is greater than most recent data. And the time cost is much less than the time needed for the generating frequent patterns algorithm. For example it costs 176 seconds to run the FP-growth algorithm to generate frequent patterns over dataset with 10k transactions when we set minimum support as 0.005, and 360 seconds for dataset with 30k transactions.

The Fading factor reflects the important of the historical data.

When D is fixed, the greater fading factor, the more frequent sampling operations will take place. In the case sampling operations take place frequently, storing the sampled historical data in main memory in the form of a compact tree with error threshold will speedup processing data stream. Otherwise we can store both sampled historical data and most recent data in disk to maintain the original sampled data.

To test how the cost time changes for FP-Sample-Merger algorithm when the number of distinct items increases, we do the following experiments. We select most recent data: Trans num, 1k; Error threshold $\varepsilon = 0.0005$. Fading factor λ =0.5.

Transaction number of historical data, 1k. We use distinct items number 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000. We get Figure 5. This figure shows that when the distinct items increase, with other properties unchanged, the cost time does not appear to increase, or in other words, similar.

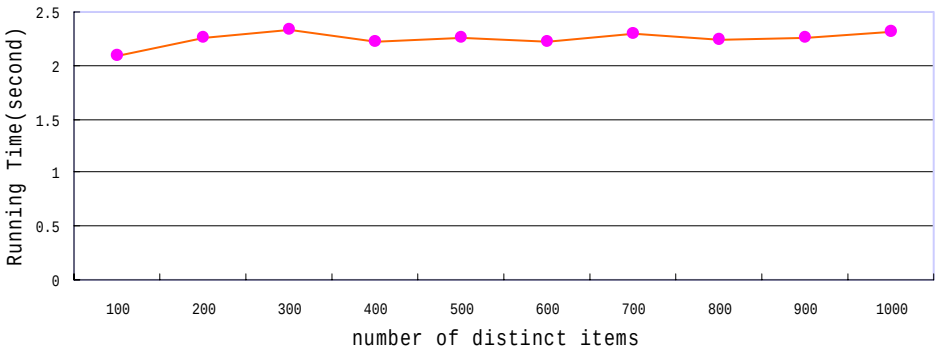

Figure 5. Different cost time with respect to different number of distinct items

To test how the cost time changes for FP-Sample-Merger algorithm when the average length of transactions increases, we do the following experiments. We select most recent data: Trans num, 1k; Error threshold $\varepsilon = 0.0005$. Fading factor λ =0.5. Transaction number of historical data, 1k. We use average length of transactions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14. We get Figure 6. This figure shows that when the average length of transactions increase, with other properties unchanged, the cost time increase linearly. The major cause should be that when the average length of transactions increase, the data for every transaction to be processed increase too, and the FP-tree becomes deeper.

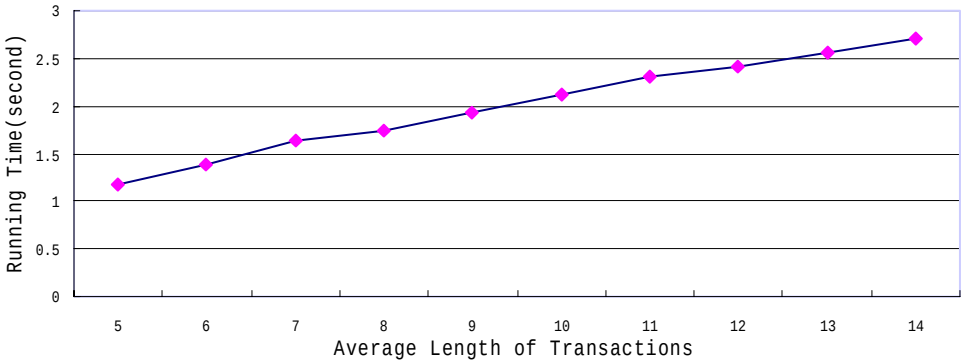

Figure 6. Different cost time with respect to different average length of transactions.

Figure 5 meets our hypothesis in section 6 very well. When the number of distinct items increases, the cost time will increase less sharply than linear function. Figure 6 does not meet our hypothesis in section 6 perfectly. The hypothesis says when the

average length of transactions increases, the cost time will increase less sharply than linear function. Here although the increase of cost time is not greater than linear function of average length of transactions, it is not less than linear function. We will try to find some methods to improve the algorithm to meet the standard of hypothesis 2 in section 6 in the future. One major time cost in the algorithm would be through IO to process most recent data. If we can improve this part significantly, the result will be much better.

8. Conclusions

In this paper we propose an approximate scheme to mine frequent patterns over Data Streams. We keep the data which we will mine on about the same size and on the other hand we try to avoid wasting too much time on sampling to keep the data around the same size. In this way we guarantee to run mining algorithm in the limited memory capacity when the data steam becomes large enough, and to keep the running time at the regular range when the time evolves. We propose two methods to do the merging and sampling operation. The one based on the compact tree structure is very efficient according to our experiments.

References

[1] Agrawal,R., and Srikant,r. 1994. Fast algoritm for mining association rules. In Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94), 487-499

[2] Han, J.; Pei, J.; and Yin, Y. 2000. Mining frequent patterns without candidate generation. In Porc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00), 1-12

[3]Pei, J. han, J. and Mao, R. 2000. CLOSET: An efficient algorithm for mining frequent closed itemsets. In proc. 2000 ACM-SIGMOD Int. Workshop Data Mining and Knowledge Discovery (DMKD'00), 11-20.

[4] Zaki, M.J., and Hsiao, C.J. 2002. CHARM: An efficient algorithm for closed itemset mining. In Proc. 2002 SIAM Int. Conf. Data Mining, 457-473

[5] C. Giannella, J. Han, J. Pei, X. Yan, and P.S. Yu, Mining Frequent Patterns in Data Streams at Multiple Time Granularities, in H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha (eds.), Next Generation Data Mining, AAAI/MIT, 2003.

[6] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In Proceedings of the 28th International Conference on Very Large Data Bases, Hong Kong, China, August 2002.

[7] Pierre-alain Laur, Richard Nock, Jean-emile Symphor. Mining Sequential Patterns on Data Streams: a Near-Optimal Statistical Approach. 2nd int. Workshop on Knowledge Discovery from Data streams (at ECML/PKDD 05). pp 29-40.

[8]Gaber, M, M., Zaslavsky, A., and Krishnaswamy, S., Mining Data Streams: A Review, ACM SIGMOD Record, Vol. 34, No. 1, June 2005, ISSN: 0163-5808.

Appendix

The demo Snapshot.

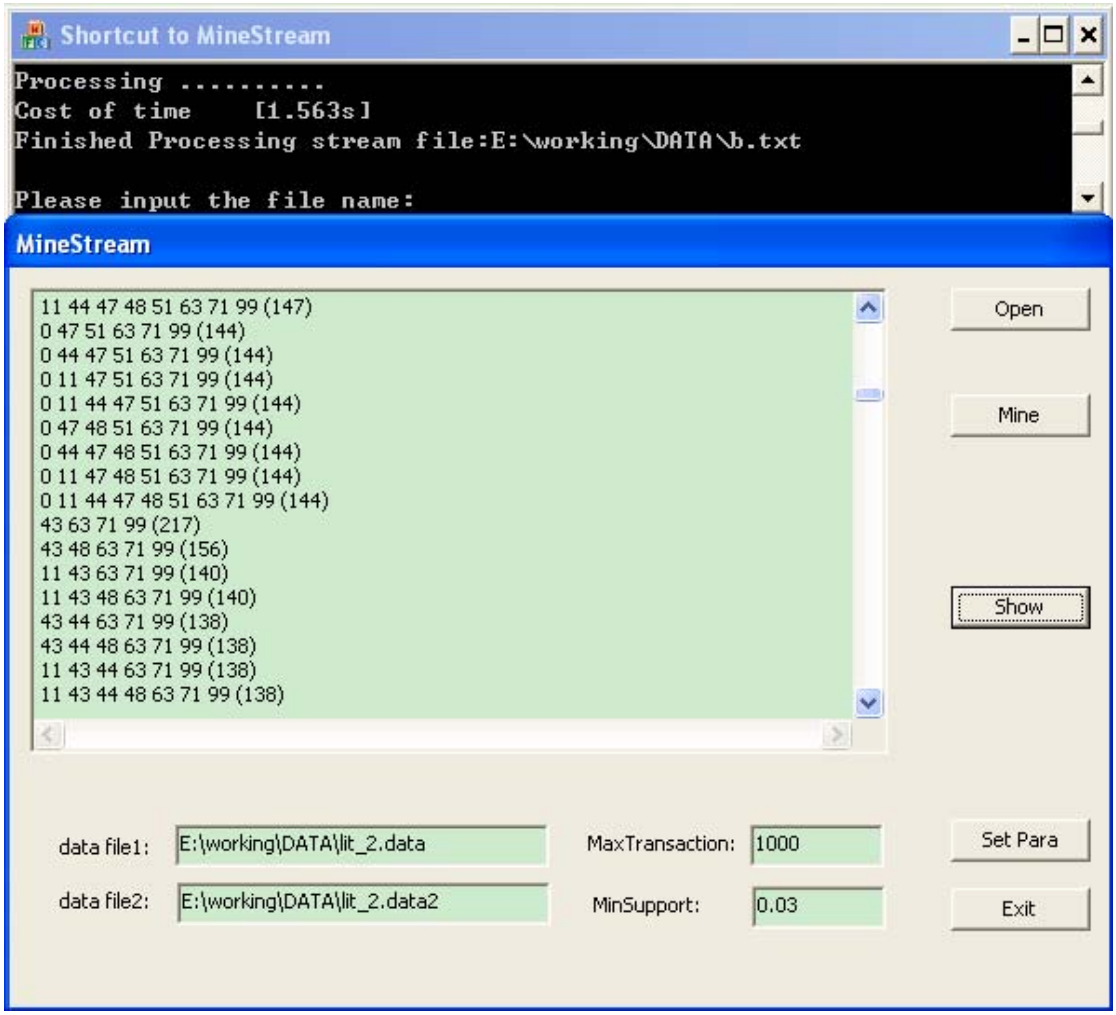

Figure 5. Snapshot of the demo