An Efficient Mining Algorithm by Bit Vector Table for Frequent Closed Itemsets

Keming Tang
College of Information Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
Department of Computer Science, Yangzhou University, Yangzhou, China
Department of Software Engineering, Yancheng Teachers University, Yancheng, China
tkmchina@126.com

Caiyan Dai
Department of Computer Science, Yangzhou University, Yangzhou, China
daicaiyan@gmail.com

Ling Chen*
Department of Computer Science, Yangzhou University, Yangzhou, China
State Key Lab of Novel Software Technology, Nanjing University, Nanjing, China
lchen@yzcn.net

Abstract—**Mining frequent closed itemsets in data streams is an important task in stream data mining. In this paper, an efficient mining algorithm (denoted as EMAFCI) for frequent closed itemsets in data stream is proposed. The algorithm is based on the sliding window model, and uses a Bit Vector Table (denoted as BVTable) where the transactions and itemsets are recorded by the column and row vectors respectively. The algorithm first builds the BVTable for the first sliding window. Frequent closed itemsets can be detected by pair-test operations on the binary numbers in the table. After building the first BVTable, the algorithm updates the BVTable for each sliding window. The frequent closed itemsets in the sliding window can be identified from the BVTable. Algorithms are also proposed to modify BVTable when adding and deleting a transaction. The experimental results on synthetic and real data sets indicate that the proposed algorithm needs less CPU time and memory than other similar methods.**

Index Terms—**data mining, frequent closed itemsets, bit vector table, data stream, sliding window**

I. Introduction

Mining frequent itemsets from data streams is an important problem with wide applications in data streams analysis. Examples include stock tickers, bandwidth statistics for billing purposes, network traffic measurements, web-server click streams, transaction analysis in stocks and telecom call records, and so on. Unlike traditional data sets, data streams flow in and out of a computer system continuously with varying update rates. They are temporarily ordered, fast changing, massive and potentially infinite. For the stream data applications, the volume of data is usually too huge to be stored or to be scanned for more than once. Furthermore, since the data items can only be sequentially accessed in data streams, random data access is not practicable.

To improve efficiency of the mining process, Han[1] proposed an algorithm FP-growth (frequent-pattern growth) which uses a FP-tree and a head table *L* to find frequent itemsets. It is not practicable in data stream mining which allows only one time scan.

In solving many application problems on data stream, it is desirable to discount the effect of the old data. One way to handle such problem is to use sliding window models[2]. There are two typical models of sliding window[3]: milestone window model and attenuation window model. H.F.Li[4] made use of NewMoment to maintain the set of frequent closed itemsets in data streams with a transaction-sensitive sliding window. MOMENT by Chi[5] is also a typical algorithm which can decrease the size of the data structure. N.Jiang[6] proposed a novel approach for mining frequent closed itemsets over data streams. Y.Chi[7] introduced a compact data structure, i.e. the closed enumeration tree, to maintain a dynamically selected set of itemsets over a sliding window. The selected itemsets contain a boundary between frequent closed itemsets and the rest of the itemsets. F.J.Ao[8] presented an algorithm named FPCFI-DS for mining closed frequent itemsets in data streams. FPCFI-DS uses a single-pass lexicographical-order FP-Tree-based algorithm with mixed item ordering policy to mine the closed frequent itemsets in the first window, and updates the tree for each sliding window. J.Y.Wang[9] proposed an alternative mining task for mining top-*k* frequent closed itemsets of length no less than *min_l*. The BitTableFI algorithm by J.Dong[10] is based on a structure of BitTable. BitTable is a set of integer where every bit represents an item.

* corresponding author.
Email addresses: yzulchen@gmail.com (Ling Chen).

© 2011 ACADEMY PUBLISHER
doi:10.4304/jsw.6.11.2121-2128

Since BitTableFI only mines frequent itemsets, it generates huge amount of candidate itemsets. Furthermore, BitTableFI is just for mining the frequent itemsets from the static database, so it is obviously not suitable for the data stream.

In this paper, an efficient mining algorithm (denoted as EMAFCI) for frequent closed itemsets in data stream is proposed. The algorithm is based on the sliding window model, and uses a Bit Vector Table (denoted as BVTable) where the transactions and itemsets are represented by the column and row vectors respectively. The algorithm first builds the BVTable for the first sliding window. Frequent closed itemsets can be detected by pair-test operations on the binary numbers in the table. After building the first BVTable, the algorithm updates the BVTable for each sliding window. The frequent closed itemsets in the sliding window can be identified from the BVTable. The algorithm is also proposed to modify BVTable when adding and deleting a transaction. The experimental results on synthetic and real data sets indicate that the proposed algorithm needs less time CPU time and memory than other similar methods.

The rest of this paper is organized as follows. The next section describes the background of frequent closed itemset mining. In section 3, we introduce our algorithm EMAFCI. Section 4 shows the experimental results in testing EMAFCI. Finally, conclusions are given in Section 5.

II. Background

A. Frequent Closed Itemsets

Let $I = \{i_1, i_2, \cdots, i_m\}$ be a set of distinct data items, and a subset $X \subseteq I$ is called an itemset. Each transaction t is a set of items in I . A data stream $DS = \{(tid_1, t_1), ..., (tid_n, t_n), ...\}$ is an infinite sequence of transactions in which tid_k is the identifier of a transaction and $t_k \subseteq I (k = 1, 2, \cdots, n)$ is an itemset. For all transactions in a given window of the data stream, the support $sup(X)$ of an itemset X is defined as the number of transactions with X as a subset.

In general, the more transactions a sliding window has, a larger amount of frequent itemsets could be produced. In this case, there are many redundancies among those frequent itemsets. For example, in the frequent itemsets $\{acd, ad, a\}$, the only useful information is the set acd according to Apriori property, because it includes ad and a. Frequent closed itemset is a solution to this problem. A frequent itemset X is a closed one if it has no superset $Y \supset X$ so that $sup(X) = sup(Y)$. Frequent closed itemset is a condensed, i.e. both concise and lossless, representation of a collection of frequent itemsets.

B. Sliding Window

The basic idea of mining frequent closed itemset in the sliding window model is that it makes decisions from the recent transactions in a fixed time period instead of all the transactions happened so far. Formally, a new data element arriving at the time t will expire at time $t + w$, in which w is the length of the window. At every time step, when a new transaction comes to the window, the oldest one in the window should be deleted. Since the transactions in the window are updated over time, the frequent itemsets should be renewed accordingly.

III. The EMAFCI Algorithm

In this section, we illustrate the framework of the algorithm EMAFCI for mining frequent closed itemsets in data streams based on the model of sliding window. First we introduce the data structure of BVTable used in the algorithm.

A. The BVTablet

The EMAFCI algorithm is based on the data structure of BVTable. To compress the itemsets and the database, BVTable consists of a set of binary integer where each bit represents an item. It consists of three parts, the left, middle and right part.

The *i*th row of the BVTable is a vector (s_i, t_i, c_i) where binary integers s_i, t_i are the left and middle parts respectively, and the right part c_i is the support of the itemset corresponding to the *i*th row. In the left part of BVTable, each column represents an item and each row is a binary integer corresponding to a candidate itemset. Denote the *j*th bit of s_i, as s_{ij}. If the *j*th item is included in the *i*th itemset, then $s_{ij} = 1$, otherwise $s_{ij} = 0$. In the middle part of BVTable, each column represents a transaction in the current time window and each row is a binary integer indicating whether the itemset represented by this row is included in the transaction or not. Denote the *j*th bit of r_i, as r_{ij}. If the *i*th itemset is included in the *j*th transaction, then $r_{ij} = 1$, otherwise $r_{ij} = 0$. In the right part of the *i*th row, $c_i = H(r_i)$ is the support of the itemset corresponding to the *i*th row, here $H(r_i)$ is the number of bits "1" in r_j. Since c_i can be calculated easily from r_i, it doesn't need to be physically stored in the memory.

To mine the frequent closed itemsets from the current sliding window, the algorithm EMAFCI first builds a BVTable for all 1-itemsets that are denoted as L_1. Based on L_1, all the frequent 2-itemsets can be detected and L_2 can be built. Repeat this procedure until all the r-itemsets are detected, and here r is the maximum length of the transactions.

Here, L_1 consists of all the 1-itemsets that include the frequent and nonfrequent ones in order to store all the transactions in the current window.

Example 1: Let $I = \{a, b, c, d\}$ be the set of all the items, *minsup*=2, the size of a window $w = 4$. The set of transactions in the first window is $D = \{bc, ab, acd, acd\}$. The ItemLists L_1 and L_2 are shown as in TABLE 1 and TABLE 2 respectively.

© 2011 ACADEMY PUBLISHER

TABLE 1.
THE BVTABLE L_1

a	*b*	*c*	*d*	1	2	3	4	count
1	0	0	0	0	1	1	1	3
0	1	0	0	1	1	0	0	2
0	0	1	0	1	0	1	1	3
0	0	0	1	0	0	1	1	2

TABLE 2.
THE BVTABLE L_2

a	*b*	*c*	*d*	1	2	3	4	count
1	0	1	0	0	0	1	1	2
1	0	0	1	0	0	1	1	2
0	0	1	1	0	0	1	1	2

The left part of the 3rd row of L_1, which represents the item c, is compressed to 2 and its binary code is 0010. The middle part of this row is 11 and its binary form is 1011, which means that transactions 1, 3 and 4 include item *c*. Similarly in the left part of the 1st row in L_2 is 10 and its binary form is 1010 which corresponds to the itemset ac. The middle part of this row is 3, and its binary form is 0011, which means that transactions 3 and 4 include itemset ac.

Let n be the maximum length of the transactions, and w be the length of the sliding window, then each row of the table consists of $w+n$ bits. Since the right part can be obtained directly from the middle part, it doesn't need to be stored in the memory. Let r be the maximum number of bits of a binary integer in the system, an array of size $(w+n)/r$ is used to store each compressed data.

B. Framework of algorithm

The algorithm EMAFCI receives a transaction from the data stream at each time step, and forms a new sliding window by adding this new transaction into the window and emitting the oldest one. To identify the frequent closed itemsets in this new sliding window, EMAFCI should modify the BVTable accordingly. Since two adjacent windows are overlapped except the added and deleted transactions, the frequent closed itemsets of the two windows do not change abruptly. EMAFCI needs only to process the part of BVTable involving these two transactions. Therefore, procedures are proposed to modify BVTable when adding and deleting a transaction.

The framework of algorithm EMAFCI is as follows:

Algorithm: EMAFCI(D, L)
Input: D : the data stream;
Output: L : the BVTable;
Begin
  BuildFirstBVTable(D, n, L);
  while not the end of the stream **do**
    Receive a new transaction *x* from the stream;
    DeleteTransFCI(L);
    AddTransFCI(L , x);
  end while
End

C. Build the BVTable for the first window

The ItemList for the first window is constructed by a procedure BuildFirstBVTable(). First the BVTable for the 1-itemsets L_1 should be generated. Then the BVTable L_2 for the frequent 2-itemsets are generated from the frequent 1-itemsets by performing bitwise OR operation (denoted as $\bigcup$) in the left part of the BVTable and AND operation (denoted as $\bigcap$) in the middle part of the BVTable. Similarly, the frequent 3-itemsets and 4-itemsets are generated from the 2-itemsets. Iterate this procedure until all the frequent itemsets are detected. Let n be the maximum length of the transactions, so the maximum number of such iterations is $\log_2 n$. Among the frequent itemsets detected, the sub-itemsets with the same support are labeled "*", because they are non-closed frequent itemsets. Details of the operation are as follows.

We denote the BVTable after the *k*th iteration as L_k. Let $m = 2^{k-2}$, and denote the set of all frequent j-itemsets as C_m , then L_k consists of all the frequent itemsets $C_{m+1} \cdots C_{2m}$. To construct L_{k+1} , pair-test operation should be performed on each pair of frequent itemsets to generate possible larger frequent itemset. Let (s_i, t_i, c_i) and (s_j, t_j, c_j) be two frequent itemsets in L_k, then a pair-test operation can generate an itemset $(s_i \bigcup s_j, t_i \bigcup t_j, c)$, here $c = H(t_i \bigcap t_j)$ is the number of bits "1" in $t_i \bigcap t_j$.

In each iteration we generate the frequent itemsets in L_{k+1} based on L_k which consists of the frequent itemsets $C_{m+1} \cdots C_{2m}$, and entirely ignore the itemsets in $L_{k-1}, L_{k-2} ... L_1$.

Theorem 1 Let C_i be the set of all the frequent i-itemsets, all sets $C_j (2i > j > i+1)$ can be generated by pair-test operation only on the frequent itemset pairs in C_i regardless of $C_{i-1}, C_{i-2} ... C_1$.

Proof: Since for any frequent itemset I in $C_j (2i > j > i+1)$, all its i-item subsets are also frequent and must be included in C_i. We can partition I into two subsets I_1 and I_2 such that $|I_1| = |I_2| = i$ and $I_1 \bigcup I_2 = I$. It obvious that I_1 and I_2 are all in C_i and C_j, which can be generated by pair-test operations on itemset pair I_1 and I_2 in C_i.

Q.E.D.

Let the *i*th row in BVTable L_j be $L_{ji} = (s_i, r_i, c_i)$, where s_i, r_i and c_i are the left, middle and right parts of L_{ji} respectively. Denote $H(r)$ as the number of bits "1" in r. The framework of the algorithm BuildFirstBVTable(D, n, L) is described as follows:

© 2011 ACADEMY PUBLISHER

```
Algorithm BuildFirstBVTable( D, n, L )
Input:  D : the set of w transactions in the first
            sliding window;
        n : maximum length of the transactions
Output: the BVTable L ;
Begin
 Generate BVTable L_1 for the w transactions in D ;
 For j = 1 to log n - 1 do
   For i = 1 to |L_j| - 1 do
     For k = i + 1 to |L_j| do
        Let L_ji = (s_i, t_i), L_jk = (s_k, t_k); s = s_i ∪ s_k ;
        If the highest bit of s_i is larger than the
        highest bit of s_k then
           t = t_i ∩ t_k ;
        End if
        If H(t) ≥ minsup then
          Insert (s, t) into L_{j+1} ;
          If H(t) = H(t_i) then
             Label (s_i, t_i) with *
          End if
          If H(t) = H(t_k) then
             Label (s_i, t_i) with *
          End if
          If there is L_r = (s_r, t) and t_r = t then
             s = s ∪ s_r and label L_r * ;
          End if
          If there is L_r = (s_r, t_r) and s_r = s then
             t = t_r ∩ t and label (s_r, t_r) * ;
          End if
        End if
     End for
   End for
   check( L_{j+1} );
 End for
Delete all the entries in L_i marked with *;
End
```

The algorithm first generates BVTable L_1 for 1-itemsets. Then BVTable L_2, L_3, and $L_4 ... L_m (m \le \log n)$ are generated. In the *j*th iteration, all the pairs of rows in L_j are tested by the pair-test operation to generate larger itemset. If the new itemset is a frequent one, it is inserted into L_{j+1}. We should also detect whether the new generated itemset is a closed one. For instance, when the frequent 3-itemsets and 4-itemsets are obtained by pair-test operation on 2-itemsets, there may exist some frequent closed 4-itemsets which are the supersets and have the same supports of frequent 3-itemsets. In this case, such frequent 3-itemsets are not closed and should be deleted from L_{j+1}. So a procedure check(L_{j+1}) is used in algorithm BuildFirstBVTable(D, n, L).

Framework of the algorithm check(L_{j+1}) is described as follows:

```
Algorithm: check( L_j )
Input: the BVTable L_j ;
Output: modified BVTable L_j ;
Begin
  len = length(L_1) ;
  sort( L_1 , len);
  /*Rearrange the entries in L_i in ascending
  order of the number of items in the itemset*/
  For i = 1 to len - 1 do
    For k = i + 1 to len do
      Let L_ji = (s_i, t_i) , L_jk = (s_k, t_k) ;
      If s_j ∪ s_k = s_j then label L_jk *;
         Continue
      End if
    End for
  End for
End
```

Example 2: Let $I = \{a,b,c,d\}$ be the set of all the items, *minsup*=2, the size of a window $w = 4$. The set of transactions in the first window is $D = \{bc, ab, acd, acd\}$. We know the frequent items are a,b,c,d. L_1 is shown in TABLE 1. From L_1, L_2 = {1010, 1001, 0011} is obtained as shown in TABLE2. Using L_2, the BVTable L_3 for 3-itemsets is generated as shown in TABLE 3.

TABLE3.
THE BVTABLE L_3

a	*b*	*c*	*d*	1	2	3	4	count
1	0	1	1	0	0	1	1	2

Finally, we get the frequent closed itemsets $(a:3), (c:3), (b:2)$ and $(acd:2)$.

D. Deleting a transaction

An algorithm DeleteTransFCI(L) is presented to delete the oldest transaction from the current window. The framework of the algorithm is as follows:

```
Algorithm DeleteTransFCI( L )
Input: the BVTable L ;
Output: the updated BVTable L ;
Begin
 For j = 1 to log n do
   For i = 1 to |L_j| do
      T_i = T_i × 2 mod 2^w
      If j > 1 and H(T_i) < minsup then
         delete T_i ;
```

© 2011 ACADEMY PUBLISHER

End if
End for
End for
End

Example 3: In Example 1, for deleting the transaction bc, the items in the deleted transaction are b and c. After deleting the transaction, the supports of the itemset a,b,c and d are 3, 1, 2 and 2 respectively. The updated BVTable L_1 is as shown in TABLE 4.

TABLE 4.
THE BVTABLE L_1 FOR THE 1-ITEMSETS AFTER DELETING bc

a	b	c	d	1	2	3	4	count
1	0	0	0	0	1	1	1	3
0	1	0	0	0	1	0	0	1
0	0	1	0	0	0	1	1	2
0	0	0	1	0	0	1	1	2

Since the minimum support is 2, the item b is infrequent but is still in the table for further process. Then the BVTable L_2 for frequent 2-itemsets is updated as shown in TABLE 5.

TABLE 5.
THE BVTABLE L_2 FOR 2-ITEMSETS AFTER DELETING bc

a	b	c	d	1	2	3	4	count
1	0	1	0	0	0	1	1	2
1	0	0	1	0	0	1	1	2
0	0	1	1	0	0	1	1	2

In EMAFCI, the frequent 3-itemsets and 4-itemsets are generated from the 2-itemsets. The BVTable L_3 is updated as shown in TABLE 6.

TABLE 6.
THE BVTABLE L_3 FOR 3-ITEMSETS AND 4-ITEMSETS AFTER DELETING

a	b	c	d	1	2	3	4	count
1	0	1	1	0	0	1	1	2

After updating the BVTable L_3 for 3-itemsets and 4-itemsets, L_4 can be updated according to L_3.

E. Adding a transaction

When a new transaction is entering the window, the BVTable also should be maintained accordingly. An algorithm AddTransFCI(L , x) is presented to add a new transaction into the current window.

Framework of algorithm AddTransFCI(L , x) is shown as follows.

Algorithm: AddTransFCI(L , x);
Input: the BVTable L, the new transaction x to be added;
Output: the modified BVTable L ;
Begin
Add x as a new column in the middle part of L_1 , counts of the items in x are modified accordingly.
For $j=1$ **to** $\log n-1$ **do**
For $i=1$ **to** $|L_j|-1$ **do**
For $k=i+1$ **to** $|L_j|$ **do**
Let $L_{ji}=(s_i,t_i,c_i)$, $L_{jk}=(s_k,t_k,c_k)$; $s=s_i \bigcup s_k$
If $j>1$ *or* (c_i *>minisup* and c_k *>minisup*) **then**
If the highest bit of s_i is larger than the highest bit of s_k **then**
$t=t_i \bigcap t_k$;
End if
If $H(t)\geq minsup$ **then**
Inser (s,t) into L_{j+1}
If $H(t)=H(t_i)$ **then**
Label (s_i,t_i) with *
End if;
If $H(t)=H(t_k)$ **then**
Label (s_i,t_i) with *
End if;
If there is $L_r=(s_r,t)$ and $t_r=t$ **then**
$s=s\bigcup s_r$ and label L_r *
End if
If there is $L_r=(s_r,t_r)$ and $s_r=s$ **then**
$t=t_r \bigcap t$ and label (s_r,t_r) *
End if
End if
End for
End for
check(L_{j+1});
End for
Delete all the entries in L_i marked with *;
End

Example 4: Suppose in the Example 3, a new transaction abd is added, then the 1-itemsets in the added transaction are a,b and d . After adding the transaction, the supports of item a,b,c and d is 4, 2, 2, and 3 respectively. The updated BVTable L_1 is as shown in TABLE 7.

TABLE **7**.
THE BVTABLE L_1 FOR THE 1-ITEMSETS AFTER ADDING abd

a	b	c	d	2	3	4	5	count
1	0	0	0	1	1	1	1	4
0	1	0	0	1	0	0	1	2
0	0	1	0	0	1	1	0	2
0	0	0	1	0	1	1	1	3

© 2011 ACADEMY PUBLISHER

In the example, the BVTable for frequent 2-itemsets is updated from the 1-itemsets. TABLE 8 is the updated BVTable L_2.

TABLE 8.
THE BVTABLE L_2 FOR 2-ITEMSETS AFTER ADDING *abd*

a	*b*	*c*	*d*	2	3	4	5	count
1	1	0	0	1	0	0	1	2
0	0	1	0	0	1	1	0	2
0	0	0	1	0	1	1	1	3
0	0	1	1	0	1	1	0	2

From L_1 and L_2, L_3 for frequent 3 and 4-itemsets is updated as shown in TABLE 9.

TABLE 9.
THE BVTABLE FOR 3-ITEMSETS AND 4-ITEMSETS AFTER ADDING *abd*

a	*b*	*c*	*d*	1	2	3	4	count
1	0	1	1	0	0	1	1	2

Finally, we find that the frequent closed itemsets in the second window are $(a:4)$, $(ab:2)$ and $(acd:2)$.

In the algorithm AddTransFCI(L , x), only the vectors with the lowest bit 1 are mined to find the frequent closed itemsets in the added transactions. Therefore, the complexity of the algorithm is decreased, as we don't need to mine the entire BVTable.

IV. Experimental Results

In order to evaluate the performance of our algorithm EMAFCI, we test it and compare the memory requirement, the processing time for the first window and each sliding window with the algorithm Moment.

The experiments are performed on a Pentium 2.4GHz S4800A (AMD Opteron 880) CPU with 4GB RAM memory, 300GB hard drive. The algorithm is coded using the VC + + 6.0 on Linux operating system.

A. Data Set

In our experiments, we use the real database Mushroom and the synthetic databases proposed by Agrawal and Srikant for evaluating the algorithms.

Mushroom which can be downloaded from [13] is a dense dataset with 8124 transactions,. Database T40I5D10K which produces data simulating the transactions of retail stores is generated by using the synthetic data generator described by Agrawal et al.

B. Experimental results and analysis

(1) Mushroom

We have adopted the commonly used parameters: the number of transactions is set as 8124 while the size of window as 8000. We report the average performance over 124 consecutive sliding windows.

In Fig.1, the times for processing the first window by the algorithms of Moment and EMAFCI are compared. From Fig.1 we can see that when the *support=minsup/*8000 is set between 1 and 0.8, the processing time of EMAFCI is equal to that of Moment. But when the *support* is lower than 0.8, the time of Moment is much more than that of EMAFCI. For instance, when *support* is set as 0.2, time cost of Moment is more than 200s, while the time of EMAFCI is less than 50s.

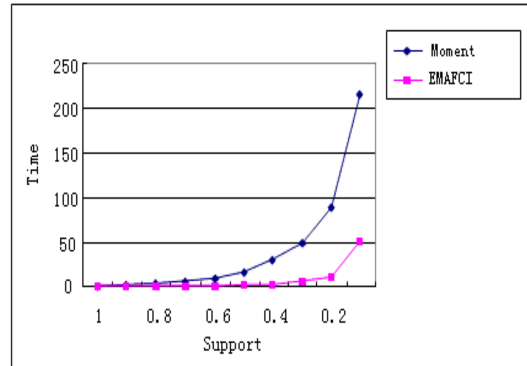


Figure 1. The time of processing the first window of Mushroom by EMAFCI compared with Moment

In Fig.2, the average time for processing a sliding window by the two algorithms are compared. From Fig.2, we can see that the average time for processing a sliding window by EMAFCI is much less than that of Moment especially when *support* is small. The time required for one window by EMAFCI is less than 0.04s, while the time of Moment increases very quickly and can go beyond 0.16s.

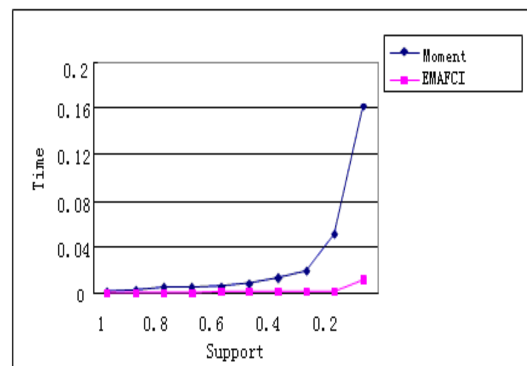


Figure 2. The average time of processing one window of Mushroom by EMAFCI compared with Moment

In Fig.3, the memory requirements by the algorithms of Moment and EMAFCI are compared. As we can see from Fig.3, the total memory required by EMAFCI is much less than that by Moment.

Since there are 10K transactions in the synthetic dataset T40I5D10K, we set the sliding window size as 5000 and perform the experiment on 100 consecutive sliding windows. Figs.4 to 6 show the processing time for the first window, average processing time for each sliding window, and the memory requirement on database T40I5D10K. We also compare the performance with that of algorithm Moment.

© 2011 ACADEMY PUBLISHER

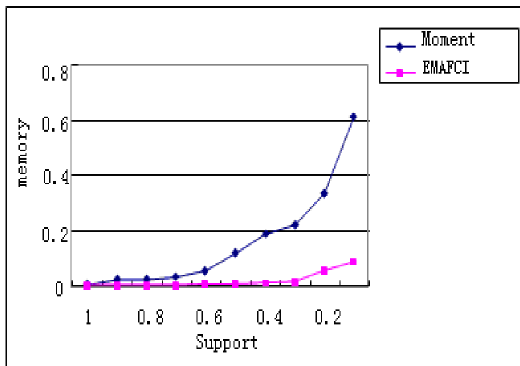


Figure 3. The memory required by EMAFCI compared with Moment on Mushroom

(2) T40I5D10K

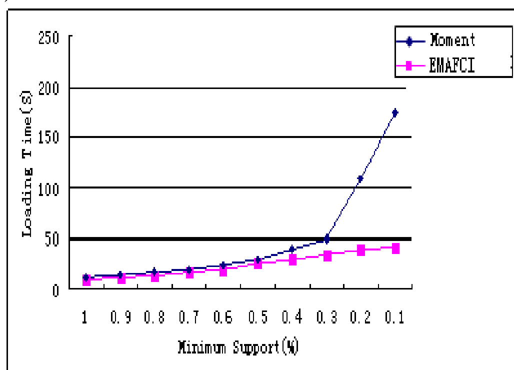


Figure 4. The time of processing the first window of T40I5D10K by EMAFCI compared with Moment

In Fig.4, the times of processing the first window by the algorithms of Moment and EMAFCI are compared. From Fig.4 we can see that the time of Moment is much more than that of EMAFCI. For instance, when *support=minsup/*5000 is set as 0.1, time cost of Moment is more than 150s, while the time of EMAFCI is less than 50s.

In Fig.5, the average time for processing a sliding window by the two algorithms is compared. From Fig.5 we can see that the average times for processing a sliding window by EMAFCI is much less than that of Moment especially when *support* is small. The time required for one window by EMAFCI is less than 0.06s, while the time of Moment increases very quickly and can go beyond 0.12s.

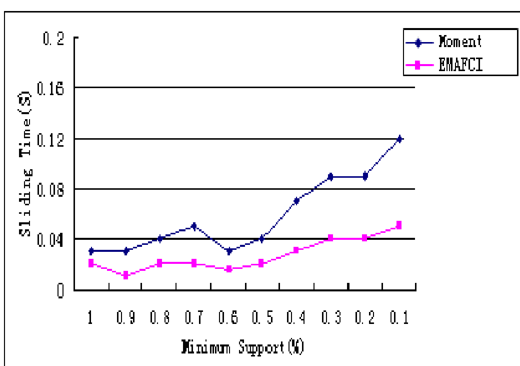


Figure 5. The average time of processing one window of T40I5D10K by EMAFCI compared with Moment

Fig.6 shows the comparison of memory requirements between algorithms Moment and EMAFCI. As we can see from Fig.6, the total memory required by EMAFCI is much less than that by Moment.

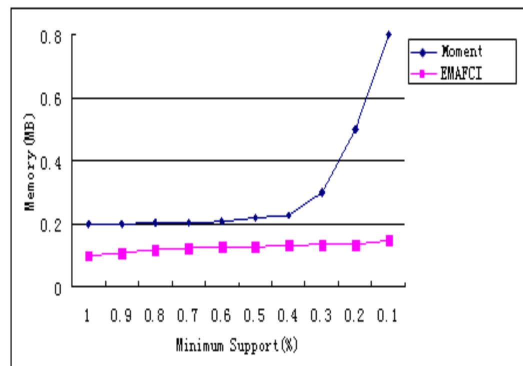


Figure 6. The memory required by EMAFCI compared with Moment on T40I5D10K

From the above experiments, we can see that algorithm EMAFCI requires much less computational time and memory space than Moment. The reason for EMAFCI getting such high performance is that it greatly reduces the search space and the storage of the itemsets in the EMAFCI.

V. Conclusion

An algorithm of EMAFCI is proposed for mining the closed frequent itemsets from data stream. The algorithm is based on the sliding window model, and uses a BVTable where the transactions and itemsets are recorded by the column and row vectors respectively. The algorithm first builds the BVTable for the first sliding window. Frequent closed itemsets can be detected by pair-test operations on the binary numbers in the table. After building the first BVTable, the algorithm updates the BVTable for each sliding window. The frequent closed itemsets in the sliding window can be identified from the BVTable. Algorithms are also proposed to modify BVTable when adding and deleting a transaction. The EMAFCI algorithm is implemented and compared its performance with Moment in terms of processing time and memory requirement. Our experimental results on both synthetic and real data show that EMAFCI is more effective with the guaranty of accuracy[12,13].

Acknowledgements

This research was supported in part by the Chinese National Natural Science Foundation under grant No. 61070047, Natural Science Foundation of Jiangsu Province under contract BK2008206, and The Graduated Student’s Research Innovation Project Jiangsu Province under contract CX08B_098Z.

References

[1] J.W.Han, J.Pei, Y.W.Yin, R.Y.Mao. Mining frequent patterns without candidate generation: frequent-pattern tree

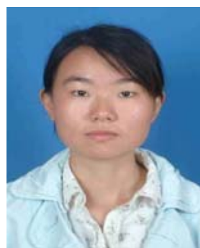
© 2011 ACADEMY PUBLISHER

approach, Data Mining and Knowledge Discovery, No.8, pp.53-87, 2004.
[2] K.T.Chuang, H.L.Chen, M.S.Chen. Feature-preserved sampling over streaming data. ACM Transactions on Knowledge discovery from data, Vol.2, No.4, Article 15, 2009.
[3] Y.Y.Zhu, D.Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. Proceedings of the 28th International Conference on VLDB, Hong Kong, China, pp.358-369, 2002.
[4] H.F.Li, C.C.Ho, S.Y.Lee. Incremental updates of closed frequent itemsets over continuous data streams. Expert Systems with Applications, Vol.36, pp.2451-2458, 2009.
[5] Y.Chi, H.Wang, P.S.Yu, R.R.Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. Proceedings of the 2004 IEEE International Conference on Data Mining. Brighton, UK, pp.59-66,2004.
[6] N.Jiang, L.Gruenwald. Research issues in data stream association rule mining. SIGMOD Record 35 (1), pp.14-19, 2006.
[7] Y.Chi, H.Wang, P.S.Yu, R.R.Muntz. Catch the moment: Maintaining closed frequent itemsets over a data stream sliding window. Knowledge and Information Systems, 10 (3), pp.265-294, 2006.
[8] F.J.Ao, J.Du, Y.J.Yan, B.H.Liu, K.D.Huang. An efficient algorithm for mining closed frequent itemsets in data Streams. Proceedings of the IEEE 8th International Conference on Computer and Information Technology, pp.37-42, 2008.
[9] J.Y.Wang, J.W.Han, Y.Lu, P.Tzvetkov. TFP: An efficient algorithm for mining Top-K frequent closed itemsets. IEEE Transaction on knowledge and Engineering, Vol.17, No.5, pp.652-664, 2005.
[10] J.Dong, M.Han. BitTableFI: An efficient mining frequent itemsets algorithm. Knowledge-Based Systems, Vol.20, pp.329-335, 2007.
[11] Dataset available at http://fimi.cs.helsinki.fi/.
[12] H.F.Li, H.Chen. Improve frequent closed itemsets mining over data stream with BitMap. Ninth ACIS international Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, pp.399-404, 2008.
[13] L.Chen, L.J.Zou, L.Tu. Stream data classification using improved fisher discriminate analysis. Journal of Computers. Vol.4, No.3, pp.208-214, 2009.

Keming Tang was born in Jianhu, Jiangsu, P.R. China, in October 13, 1965. He received master degree in engineering from Yangzhou University, P.R. China in 2002. Now, he is Ph. doctoral student of Nanjing University of Aeronautics and Astronautics.

He is currently associate professor of computer science, and the vice-dean of Information Science and Technology College, YanCheng Teachers University, Jiangsu Province, P.R. China. He has published more than 20 papers in journals including IEEE Transactions on CiSE and WISM, Journal of Computer Mathematics. His research interest includes data mining, peer to peer computing and software engineering.

He is a member of the Chinese Computer Society. His recent research has been supported by the Chinese National Natural Science Foundation.



Caiyan Dai, was born in Yancheng, Jiangsu, Sep 26, 1985. She received bachelor of engineering degree in computer education from Yangzhou University in 2004.

She is a master of Information Engineering College of Yangzhou University. Her research director is Data mining.

She is a student member of the Chinese Computer Society.

Ling Chen, was born in Baoying, Jiangsu, P.R. China, in September 10, 1951. He received B.Sc. degree in mathematics from Yangzhou Teachers' College, P.R. China in 1976.

He is currently professor of computer science, and the dean of Information Technology College, Yangzhou University, Jiangsu Province, P.R. China. He has published more than 120 papers in journals including IEEE Transactions on Parallel and Distributed System, Journal of Supercomputing, The Computer Journal. In addition, he has published over 100 papers in refereed conferences. He has also co-authored/co-edited 5 books (including proceedings) and contributed several book chapters. His research interest includes data mining, bioinformatics and parallel processing.

Prof. Chen is a member of IEEE and senior member of the Chinese Computer Society. His recent research has been supported by the Chinese National Natural Science Foundation, Chinese National Foundation for Science and Technology Development and Natural Science Foundation of Jiangsu Province, China. Prof. Chen has organized several national conferences and workshops and has also served as a program committee member for several major international conferences. He was awarded the Government Special Allowance by the State Council, the title of "National Excellent Teacher" by the Ministry of Education, and the Award of Progress in Science and Technology by the Government of Jiangsu Province.

© 2011 ACADEMY PUBLISHER