International Journal of Computational Intelligence Systems, Vol. 7, No. 1 (February 2014), 25-36

Sliding Window-based Frequent Itemsets Mining over Data Streams using Tail Pointer Table

Le Wang[1,2,3], Lin Feng [*1,2], Bo Jin[2]
1, School of Computer Science and Technology,
Dalian University of Technology,
Dalian 116024, P.R. China,
lelewater@gmail.com, fenglin@dlut.edu.cn
2, School of Innovation Experiment,
Dalian University of Technology,
Dalian 116024, P.R. China,
jinbo@dlut.edu.cn
3, School of Information Engineering,
Ningbo Dahongying University,
Ningbo, Zhejiang 315175, P.R. China.

Received 23 August 2011

Accepted 9 June 2013

Abstract

Mining frequent itemsets over transaction data streams is critical for many applications, such as wireless sensor networks, analysis of retail market data, and stock market predication. The sliding window method is an important way of mining frequent itemsets over data streams. The speed of the sliding window is affected not only by the efficiency of the mining algorithm, but also by the efficiency of updating data. In this paper, we propose a new data structure with a Tail Pointer Table and a corresponding mining algorithm; we also propose a algorithm COFI2, a revised version of the frequent itemsets mining algorithm COFI (Co-Occurrence Frequent-Item), to reduce the temporal and memory requirements. Further, theoretical analysis and experiments are carried out to prove their effectiveness.

Keywords: data mining; data streams; frequent itemsets; sliding window; tail pointer table

1. Introduction

Since Agrawal [1] developed the first algorithm Apriori for mining frequent itemsets from static sales dataset in 1994, new algorithms are proposed constantly for various sub-domains of frequent itemsets mining, such as those for traditional frequent itemsets [2, 3, 4, 5, 6] in certain datasets, high utility itemsets [7, 8, 9, 10, 11], frequent itemsets in uncertain datasets [12, 13, 14]. These approaches could be classified into two categories: level-wise approaches and pattern-Growth approaches. Apriori [1] is a classical level-wise approach; the FP-Growth (Frequent Pattern Growth) [2] algorithm is a classical pattern-Growth approach. However, in real world there are many data streams, such as wireless sensor data, transaction flows, call records, and so on. So it has been an important research issue in the field of data mining to mine frequent itemsets over data streams.

To handle continuous data streams, time-window is commonly used, and it is an efficient approach. Depending on the mechanism of the window algorithm, three window models can be used to mine frequent itemsets (or patterns): landmark window model, damped window model, and sliding window model. There are many research works on the sliding window-based approach, including maximal frequent itemset [15], closed

frequent itemsets [16, 17], complete frequent itemsets [18, 19, 20, 21], and so on. The runtime of such algorithms is mainly impacted by the speed of the mining frequent itemsets and the updating data. There are three classical algorithms for mining frequent itemsets over data streams based on the sliding window, such as DST (Data Stream Tree) [18], DSP [19], and CPS (Compact Pattern Stream) [20]. DST and DSP have the same structure, but their mining algorithms are different: DST mines frequent itemsets with FP-Growth [2], whereas DSP mines frequent itemsets with COFI (Co-Occurrence Frequent-Item) [3]; the COFI algorithm consumes less memory than FP-Growth according to the paper [19]; the memory requirement of CPS is smaller than DST/DSP, but the drawback is that it reconstructs trees more often, which consumes more time. Moreover, CPS also uses FP-Growth to mine frequent itemsets. However, these algorithms improve the speed of the mining process, but do not consider the efficiency of updating data.

In this paper, we propose a new data structure, called TPT-tree (Tail Pointer Table tree), to store the stream data of a window, it can improve the efficiency of updating data and costs less memory than DST/DSP; and propose a corresponding algorithm, called COFI2, for mining frequent itemsets over data streams.

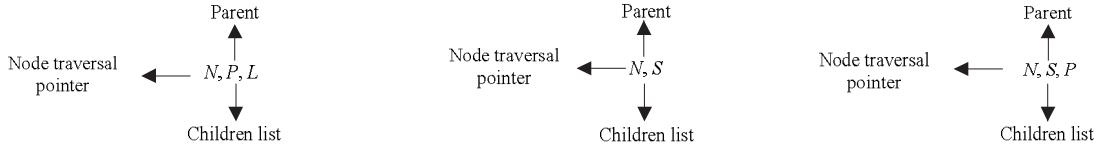
The organization of this article is as follows: Section 2 discusses related work; Section 3 provides a description of the problem and defines relevant terms; Section 4 introduces a structure TPT-tree and a corresponding algorithm; Section 5 shows the experimental results, and Section 6 gives conclusions.

2. Related work

The algorithms most closely related to our study are DST [18], DSP [19], and CPS [20], which mine exact frequent itemsets over data streams. The algorithms DST and DSP use the same data structure (DST-tree), and mine frequent itemsets using FP-growth and COFI [3] respectively; CPS creates trees with another data structure CPS-tree, and mines frequent itemsets using FP-growth. This section discusses the two data structures and the two frequent itemsets mining algorithms in detail.

2.1. *Data structures: DST-tree and CPS-tree*

To efficiently maintain data of a window to a tree and mine frequent itemsets from a window, the algorithms DST and CPS maintain data to DST-tree and CPS-tree respectively.



N: node name; *S*: total support number; *L*: ID of the last update pane

P: pane-counter[V_1,V_2,...V_w] (V_i: support number of node *N* at pane *i* ; *w*: number of panes in a window)

Fig. 1. Structure of DST-tree and CPS-tree

The node structure of DST-tree is shown in Fig. 1(a), where *N* is an *item name*, *L* is the *ID* of the pane by whose data this node is updated lastly, and *P* is a list of the support numbers of *w* panes (in the form of [V_1,V_2,.. V_w]; *w* is the window width). *P* and *L* are used to effectively delete obsolete data from a tree. Before adding new pane data into a tree, obsolete data should be deleted from the tree; and when adding new data into a tree, *P* and *L* should be updated. However, DST does not delete obsolete data before performing an operation of mining frequent itemsets, and therefore there are some invalid nodes on the DST-tree. The number of these nodes can increase along the sliding of the window, and the invalid nodes can affect the efficiency.

CPS-tree is proposed based on DST-tree. There are two types of nodes on CPS-tree: ordinary nodes and tail-nodes. The structure of an ordinary node is shown in Fig. 1(b), where *N* is the item name of the node, and *S* is total support number. The structure of a tail-node is shown in Fig. 1(c). When a transaction itemset is added into a CPS-tree, its last node is a tail-node, others are ordinary nodes. The difference between an ordinary node and a tail-node is that there is a pane-support at each tail-node; the tail-node has the same structure as that of a DST-tree's node. In this way, the CPS-tree saves space relative to the DST-tree. When a new pane of data arrives, CPS-tree is updated with the new data after the obsolete data are deleted. When deleting obsolete data, find all

tail-nodes by traversing the tree, and left-shift once the pane-support of each tail-node to leave the rightmost position for new data.

2.2. *Algorithms of mining frequent itemsets*

Han [2] proposed the FP-Growth algorithm that generates and mines frequent itemsets with the FP-tree. It is a classical pattern-Growth approach. It firstly constructs an FP-tree and a header table which only maintains frequent items, and items in the header table are sorted by their frequencies. Then process each item of the header table: firstly the item is added to the current conditional base (the conditional base is an itemset and is initialized as null) to generate a new conditional base, and construct a sub header table and a prefix tree for the new conditional base; then recursively process each item of the sub header table. During the processing, all new generated conditional bases are frequent itemsets. Mining of the *k*-itemset is changed to mining of the 1-itemset at the most *k* times by continuously building conditional FP-trees. The algorithm does not produce candidate itemsets; moreover, the database is scanned twice only.

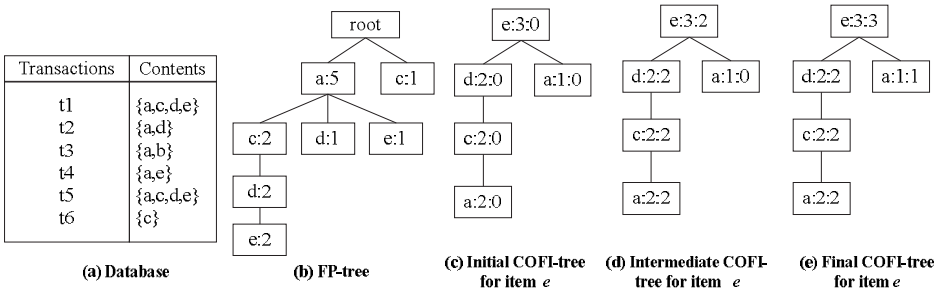


Fig. 2. Process of COFI mining

EI-Hajj and Zaiane [3] proposed the data structure COFI-Tree (Co-Occurrence Frequent-Item Tree) and the algorithm COFI for mining frequent itemsets. The key idea of this algorithm can be described as follows. First, scan database twice; the first scan aims to count frequency of each item; all items are sorted in the descending order of their frequencies; the second scan is to build a global FP-tree for maintaining the dataset. Once a global tree is built, it can be used to construct a COFI-tree for each frequent item *X*; frequent itemsets containing *X* can be discovered by the COFI-tree.

The following example shows the process of constructing a COFI-tree and mining frequent itemsets. Fig. 2(a) is an example database and Fig. 2(b) is its global FP-tree. To construct a COFI-tree for item "e", the branches, which are from the node "e" to the root, should be taken from the global FP-tree to count frequency (support number) of each item on these branches; the items are sorted by descending order of their frequencies to a local header table; the resulting local header table is {a:3, c:2, d:2}. All items of each branch have the same frequency as the support number of node "e" on the branch. All items of every branch are sorted by their local support numbers in ascending order and the sorted items are added into the COFI-tree, as shown in Fig. 2(c). The first number of every node of the COFI-tree is its support number and the second is its participation number, which have an initial value set as 0. The mining process starts with the first item in the local header table (i.e., the most frequent item locally), for example, item "a" of the local header table {a:3,c:2,d:2}; firstly find the first branch that contains node "a", then generate all combinations of all items on the branch (except the root node), and add the item "e" to each combination; and the frequency of each itemset (combination) is that of the node "a", that is 2. These itemsets are candidate itemsets and are saved. After this step, the participation numbers of all nodes on the branch should increase by 2, and the result is shown in Fig. 2(d). The second node "a" on the COFI-tree is processed in the same way, and we get a new candidate itemset {ae:1}. Each new candidate itemset should be compared with every one of the existing candidate itemsets. If it doesn't exist in the set, add it into the set; otherwise accumulate its support number. COFI goes on working till the participation number of each node on COFI-tree equals the support number; see Fig. 2(e).

3. Description of the problem

In this section, we provide the definitions for key terms that explain the concepts of mining frequent itemsets over data streams.

Let $I = \{i_1, i_2, i_3, \ldots, i_N\}$ be a set of literals, called item set. A set $X = \{i_j, \ldots, i_k\} \subseteq I$, ($j \leq k$ and $1 \leq j, k \leq N$) is called an itemset. An itemset with k items is denoted as a k-itemset. A transaction $T = (TID, Z)$ is a couple where TID is a unique identifier and Z is a subset of I. A transaction itemset Z is said to contain X, if $X \subseteq T.Z$. A transaction database $\{T_1, T_2, T_3, \ldots, T_n\}$ is called DB. The support number of an itemset X in DB is the number of transactions containing X.

A data stream DS (Data Stream) can formally be defined as an infinite sequence of transactions, $DS = \{T_1, T_2, \ldots, T_m, \ldots\}$, where T_i is the ith arrival of transactions. A sliding window W can be referred to as a set of all transactions between the ith and jth (where $j > i$) arrival of transactions and the size of W is $|W| = j - i$. Each window consists of a fixed number of panes, and a pane of data contains a fixed number of transactions. Let the window slide pane by pane, that is, each slide of the window introduces a new pane and removes the obsolete pane from the current window.

The support number of an itemset X in a window W is the number of transactions in W that contain X. Therefore, an itemset is called *frequent* in W if its support number is not less than the minimum support number, *min_sup*. Given DS, $|W|$, and *min_sup*, finding the itemsets in W, whose support numbers are not less than *min_sup*, is a problem of mining frequent itemsets from data streams wtih the sliding window mechanism.

Let $t = \{i_1, i_2, i_3, \ldots, i_k\}$ be a sorted transaction, where i_k is the tail item. If t is inserted into a tree in this order, then the node of the tree that represents the tail item is defined as a *tail-node* for t; the nodes that represent other items except the tail-item are defined as ordinary nodes. A node has a parent, but its parent does not have a pointer pointing to it, this node is called a *virtual node*.

4. TPT-tree

First, data streams are continuous and unbounded. Second, data streams are not necessarily uniformly distributed, and their distributions usually change with time. Thus we no longer have the luxury of performing multiple data scans to find frequent itemsets over data streams; it is essential to be efficient in terms of both time and memory during mining from data streams. To address this issue, a new data structure is proposed to improve the time- and memory-efficiency of such types of algorithms.

4.1. *Structure of TPT-tree*

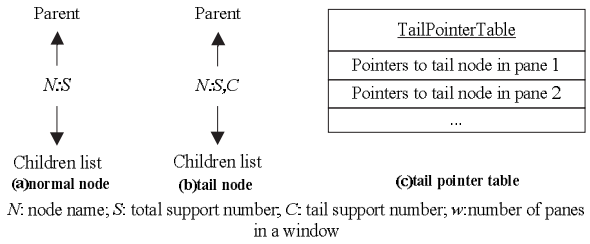


Fig. 3. Structure of TPT-tree

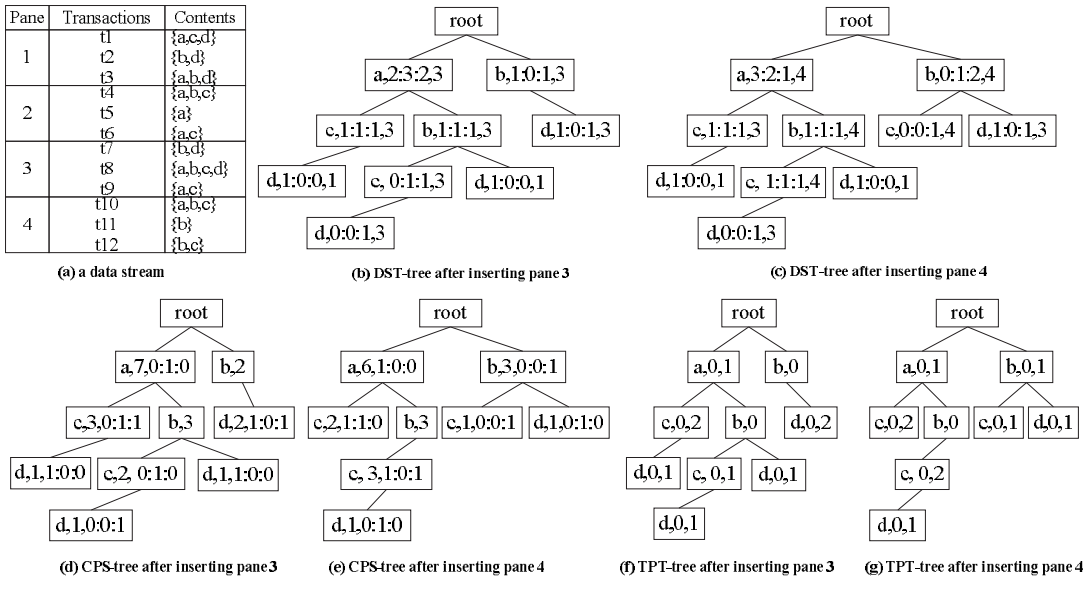


Fig. 4. A simple case study of the TPT algorithm

The structure of TPT-tree is illustrated in Fig. 3. There are two types of nodes: one is ordinary node, as shown in Fig. 3(a), where *N* is the item name of each node, and *S* is a support number (we call *S* the *total support number*); the other is tail-node, as shown in Fig. 3(b), where *C* is the *tail support number* of a tail-node (the *tail support number* of a tail-node *Tn* is the number of itemsets whose tail-nodes are *Tn*). The tail pointer table is shown in Fig. 3(c), which is used to delete obsolete data.

We use an example to illustrate construction of three types of tree structures: DST-tree, CPS-tree, and TPT-tree, as shown in Fig. 4.

Fig. 4(a) is an example data stream, with both the size of window and the size of pane are 3. The transaction itemsets are added to a DST-tree in lexicographic order; Fig. 4(b) is the result of the first three panes of data added to a DST-tree. When sliding to the second window, according to the algorithm DST, the fourth pane of data are appended to the tree, as shown in Fig. 4(c) (this is also the result when sliding to the second window); note that after the fourth pane of data are appended, the data of the first pane are not actually deleted from the tree, for example, there are two identical nodes "d,1:0:0,1".

The construction of CPS-tree is as follows: the first pane of data are added to a tree in lexicographic order; note that the last node of the transaction that is added to the tree is a tail-node, such as the node "d" in the transaction {a, c, d}. After the first pane is added, all items on the tree are re-sorted by descending order of their support numbers; now CPS-tree is converted into an FP-tree. The second pane of data is added to the tree in descending order of the support number in the first pane. After the second pane is added, again re-sort all items by descending order of their support numbers and reconstruct the CPS-tree to an FP-tree. The third pane of data is added by the same method. Fig. 4(d) shows the tree after the third pane is added.

Fig. 4(e) shows the tree after the fourth pane is added to the tree (in the same way as the third pane) and the first pane is deleted. The deletion process of the first pane is as follows: traverse the CPS-tree to find all tail-nodes; update the support number to be the value (total support number - tail-node's the leftmost pane-support) for all nodes between the tail-node and the root, and left-shift once the tail-node's pane-support list, leaving the rightmost position for recording new data.

When constructing a CPS-tree, if a tail-node that is to be added to the tree already has a corresponding ordinary node, we simply convert this ordinary node to a tail-node. For example, when adding transaction itemset {a} of the second pane, simply convert the node "a,3" under root to a tail-node "a,4,0:1:0", where "4" is total support number of the node, and "0:1:0" is the pane-support list for 3 panes of data, and the middle "1" is support number of the tail-node in the second pane.

Just as on a CPS-tree, the nodes on a TPT-tree are of two types: the ordinary nodes and the tail-nodes, as shown in Fig. 3. The construction of TPT-tree is as follows: when transaction itemsets are added to a tree in lexicographic order, we only update support number of each tail-node. For example, when adding transaction itemset {a,c,d} of the first pane, we only update the tail support number of the node "d" as "d,0,1", where "1" is tail support number of the node. Fig. 4(f) shows a TPT-tree after the first window data are appended; and Fig. 4(g) shows the tree after the processing of the second sliding window, that is, after the fourth pane is added and the first pane is deleted. The deletion process is discussed in detail in Section 4.2.

When a tail-node that is to be added to a tree already has a corresponding ordinary node, we also simply convert this ordinary node to a tail-node. For example, when adding transaction {b} of pane 4, simply convert the node "b,0" (as shown in Fig. 4(f)) under root to a tail-node "b,0,1"(as shown in Fig. 4(g)), where "1" is support number of the tail-node.

Theorem 1: Assume 3 types of data structures are constructed in the same order (e.g. lexicographic order or frequency order), TPT-tree requires the least memory for maintaining transaction itemsets.

Proof: Assume 3 types of data structures are constructed in the same order (e.g. lexicographic order), they have the same branches, but their node structures are different. Each node maintains support number of each pane on a DST-tree. On a CPS -tree, each ordinary node maintains a total support number, and each tail-node maintains support number of each pane and a total support number. On a TPT-tree, each ordinary node maintains a total support number, and each tail-node maintains only 2 support numbers no matter how much panes a window consists. Thus TPT-tree

requires the least memory for maintaining transaction itemsets.

4.2. *Data-updating algorithms*

When a new pane comes, two important processes occur: deleting obsolete data and adding new data.

The algorithm of deleting old data is shown in Fig. 5(a). Mainly its work is to process each obsolete tail-node which is recorded in the tail pointer table. If a tail-node is a virtual node, the support numbers of itself and its parent are deducted by one (line 3 and line 10); if its support number is 0 and it has no children node, it is deleted from the tree (line 15), and then we continue processing its parent: if its parent is also a tail-node and has a support number of 0 and has no children, it will be deleted, and its parent is processed (line 12-20).

The algorithm for adding new data is shown in Fig. 5(b). Transaction itemsets are sorted and added to the tree (line 3-4); then, the support number of its tail-node *TP* is increased by one (line 6), and the pointer *TP* is stored to the tail pointer table (line 7). The variable *P* in the algorithm is the looping counter.

```
Algorithm 1: deleting old data from T
Input: T , Tail_Pointer, w
Output: T
Method:
Begin
(1)  For each tail pointer TP in Tail_Pointer[w]
(2)       IF(TP.S = = -1) // a virtual node
(3)          TP.C = TP.C - 1;
(4)          TEMP = TP.parent;
(5)          IF(TP.C = = 0)
(6)             delete node TP from tree T;
(7)          End IF
(8)          TP = TEMP;
(9)       End IF
(10)      TP.C = TP.C - 1;
(11)
(12)      While (TP.C = = 0)
(13)           TEMP = TP.parent;
(14)           IF(TP has no child)
(15)              delete node TP from tree T;
(16)              TP = TEMP;
(17)           Else
(18)              Break;
(19)           End IF
(20)      End While
(21) End For
End
```

(a)Deleting obsolete data from a TPT-tree

```
Algorithm 2: Inserting data into T
Input: T ,Stream_data, Pane_size,
       Initial_sort_order, Tail_Pointer, w
Output: T
Method:
Begin
(1)  p = 0;
(2)  While (p != Pane_size)
(3)       Tr = transaction from the current location in Stream_data;
(4)       Insert Tr into T according to Initial_sort _order ;
(5)       TP = Tr's tail node;
(6)       TP.C = TP.C + 1;
(7)       Save TP to Tail_Pointer[w];
(8)       p = p + 1;
(9)  End While
End
```

(b)Inserting data into a TPT-tree

```
Algorithm 3: Mining frequent itemsets over cofi-tree
Input: T, ht (headtable), mini_support
Output: FI (frequent itemsets)
Method:
Begin
(1)  root = T.root;
(2)  For each item item1 in ht
(3)       CS = NULL; // a set for candidate itemsets
(4)       For each node node1 whose item name is
                item1
(5)            CS0 = all combinations of items between
                  node1 and root;
(6)            For each element X in CS0
(7)                 Y = X ∪ {item1} ∪ {root.N};
(8)                 Y.support = node1.S;
(9)                 Add itemset Y to CS ;
(10)           Enf For
(11)      End For
(12)      delete infrequent itemsets from CS;
(13)      FI = FI∪CS;
(14) End For
End
```

(c)COFI2 algorithm

Fig. 5. Algorithms based on TPT-tree and COFI2

Table 1. Differences in data processing methods among the three algorithms

Processing of one batch of data during window sliding	Adding data	Deleting data
DST/DSP	1. Sort items of all transaction itemsets; 2. Add sorted itemsets to a tree and modify support number of each node; 3. Record the updated batch of each node; 4. Modify support number of each node.	
CPS	1. Reconstruct a CPS-tree; 2. Sort items of a transaction itemset; 3. Add sorted itemsets to a tree and modify support number of each node; 4. Modify the header table.	1. Traverse a CPS-tree and delete obsolete data; 2. Modify the header table.
TPT	1. Sort items of all transaction itemsets; 2. Add sorted itemsets to the tree and only modify the tail support number of the tail-nodes; 3. Insert tail-nodes to tail pointer table.	1. Modify tail support numbers of tail-nodes through scanning the tail pointer table.

Table 1 shows a comparison of the three different processes for data updating of three types of data structure. We can see that CPS is the most complex one. When adding a new pane data, convert the tree to an

FP-tree and modify the header table; when deleting a pane, a traversal of the tree and modification of the header table are needed. A comparison of the runtime of TPT and DST is discussed in detail in Section 5.1.

4.3. *Frequent itemsets mining algorithm*

To improve the efficiency of frequent itemsets mining, a TPT-tree can be reconstructed to achieve as much prefix sharing as possible among transaction itemsets in the current window; after a TPT-tree is reconstructed, it can be as compact as an FP-Tree that is constructed using the same transaction itemsets. We adopt the tree reconstruction approach from that used in the CPS algorithm. The basic idea of reconstruction is the same as that in CPS, with a difference in processing tail-nodes; i.e., during the reconstruction of a tree, if the tail-node *A* is to be merged to the tail-node *B*, the following steps are performed:

(1) Accumulate *A*'s tail support number to *B*'s tail support number;
(2) Convert node *A* to a *virtual node*, and its parent is *B*.

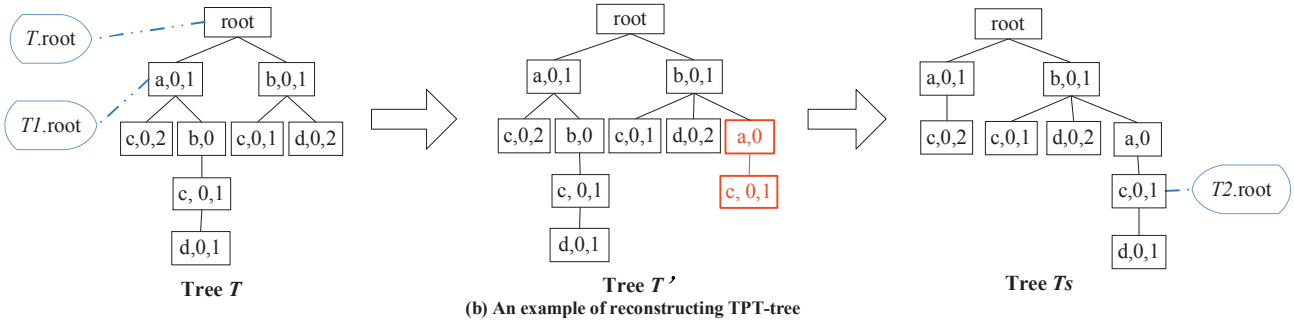
The purpose of converting *A* to a virtual node is to find node *A* from the TPT-tree and modify the tail support of both nodes *A* and *B* when deleting obsolete data containing *A*.

```
Algorithm 4: ReconstructingTree
Input: H, T
Output: Ts
//H: a header table maintaining items in descending order of support number
//T: a TPT-tree, items in each brach is not in order of H.
//Ts: a TPT-tree, items in each brach is in order of H.

Method:
Begin
(1)  root = T.root
(2)  For each branch Br in T
(3)      TN = the first tail-nodes in Br;
(4)      path Pj  is from tail-node TN to root
(5)          IF( Pj is a processed path)
(6)              Continue;
(7)          Else IF (Pj is a sorted path)
(8)              Process_path(Pj);
(9)          Else
(10)             Sort_path(Pj);
(11)             Proces_path(Pj);
(12)         End IF
(13)     End For
(14)   End For
(15) Return T;
End
```

```
Process_path(p)

Begin
(1)  A = tail node of path p;
(2)  For each node n in path p
(3)      For each sub-path sp from n to tail-node B
             // B ≠ A
(4)          IF (items in sp are at below of n in the
                 order of H)
(5)              IF (sp is a sorted path)
(6)                  Process_path(sp)
(7)              Else
(8)                  Sort_path(sp)
(9)              End If
(10)         Else // not at below of n in the order of
                 H
(11)             sp = path from n1 to B;
                 //n1 is a ancestor node of n, and
                 //items from n1 to B are at below of
                 //n1 in the order of H;
(12)             Sort_path(sp)
(13)         End IF
(14)     End For
(15) End For
End
```

```
Sort_path(p)
Begin
(1)  A = tail node of path p;
(2)  For each node Nd on path p from parent node of A
(3)      IF Nd just has one tail-node A
(4)          Delete node Nd from T;
(5)      Else
(6)          Break;
(7)      End IF
(8)  End For
(9)  Sort items in p in the order of H;
(10) Insert sorted items S into T at the below of
         node pn with a tail-node B;
(11) IF B is not a new node
(12)     IF B is not a tail-node
(13)         A.parent = B.parent; A.N = B.N;
(14)         Move B.children to A.children;
(15)     Else
(16)         B.C += A.C;
(17)         A.parent = B; A.S = -1; //A is  a virtual node
(18)     End IF
(19) Else
(20)     A.N = the last item in S;
(21)     A is as the tail node of S;
(22) End If
End
```

(a) The algorithm of reconstructing TPT-tree



(b) An example of reconstructing TPT-tree

Fig. 6. Reconstruction Algorithm of TPT-tree and an example

Fig. 6(a) shows the detailed algorithm of reconstructing a TPT-tree. Fig. 6(b) is an example of the reconstruction process for the tree in Fig .4(g) (this tree is now denoted as tree *T* in Fig. 6(b)). The following steps reconstruct tree *T*:

(1) Sort all items in *T* by the ascending order of their support numbers and get a sorted header table *H* = {b:6, a:5, c:5, d:3}.
(2) Process the first tail-node "a,0,1" on the branch "root-a-c" of tree *T*. There is only one item "a" on

the path "root-a", so no sorting is needed and we can process the sub tree whose root is "a,0,1" (the sub tree is denoted as $T1$, as shown in Fig. 6(b)):

(2.1) Process the tail-node "c,0,2" in $T1$. On the path from this tail-node to $T1$.root, the order of all items is identical with the order of H, and this tail-node has no children, so no further process is needed.

(2.2) Process another tail-node "c,0,1" in $T1$. Because the order of items on the path from this tail-node to $T1$.root is not identical with the order of H, we sort these items ("a", "b" and "c") by the order of H, add these sorted items ("b", "a" and "c") to the tree (the resulting tree is *T1* in Fig. 6(b)), and move the children nodes of the old tail-node to the new tail-node; and then, along the path from the old tail-node to *T1.root* ("c-b-a-root"), remove those nodes that have no children (that is, node "c" and "b" are removed consecutively). The resulting tree is the tree *T'* in Fig. 6(b). Next step is to process recursively the sub tree whose root is "c,0,1" (this sub tree is denoted as $T2$, as shown in Fig. 6(b)).

(2.2.1) $T2$ has one tail-node "d,0,1"; this tail-node has no children, and the items on the path from this node to $T2$.root are identical with the order of H, so no further process is needed.

(3) Using the above method，go on processing the tail-node "b,01" in tree T. The final resulting tree is tree *Ts* in Fig. 6(b).

After a TPT-tree is reconstructed, we can adopt COFI, COFI2 or FP-Growth for mining frequent itemsets of each window.

COFI2 is a revised version of COFI, to improve the efficiency of memory usage and runtime. When mining frequent itemsets containing a certain item, COFI maintains all candidate itemsets until all the nodes on the COFI-tree have been processed. Although a COFI-tree is pruned, the size of all combinations of all items may still be large if a branch contains many items; this situation will not only cost a large amount of memory, but also slow down the efficiency of matching identical itemsets among the candidate itemsets. To address this issue, our proposed approach COFI2 can maintain less number of candidates by generating candidate itemsets as late as possible and deleting infrequent itemsets as early as possible.

Fig. 2(c) is an example of the mining process of COFI. When generating all combinations of the items of a branches containing node "a", each itemset of the combinations must contain item "e", but may not contain item "a", and the number of the combinations is 8. However COFI2 requests that each itemset of the combination must contain both items "a" and "e"; and the size of the combination is 4. This is considered as generating candidate itemsets as late as possible. When dealing with the second node "a" on the COFI-tree, COFI2 generates only itemset {ae:1}. Then search the candidate itemsets for this itemset, and if it is found, we accumulate its support number; otherwise, we insert the itemset to the candidate itemsets. After finishing the processing of all nodes "a", the infrequent itemsets in the candidate itemsets can be deleted, because the itemsets that contain item "a" have been generated, and it is impossible to generate other itemsets that contain item "a" when processing other nodes. This way the infrequent itemsets can be deleted as early as possible without waiting for the whole tree to be processed.

Table 2. Differences between COFI and COFI2

Algorithms	Differences of candidate itemsets	Differences of tree structure
COFI	1. When processing an item on the COFI-tree, generate an itemset combination of all the items from the corresponding node to the root (inclusive) as the candidate itemsets; 2. After finishing the processing of all nodes on the COFI tree, delete the infrequent itemsets in the candidate itemsets.	Node contains participation count and support count.
COFI2	1. When processing an item on the COFI-tree, generate an itemset combination of all the items from the corresponding node to the root (including itself and the root) as the candidate itemsets; 2. After finishing the processing of this item, delete the infrequent itemsets from the candidate itemsets.	Node contains support count.

In addition, the nodes on the COFI-tree do not need a counter to store the participation count in algorithm COFI2. The COFI2 algorithm is shown in Fig. 5(c). When adding a new itemset Y to the set of itemsets CS, we only accumulate its support number if itemset Y exists in CS; otherwise insert this itemset to CS (line 9 in Fig. 5(c)).

Table 2 shows the comparison of COFI and COFI2. The main difference between the two algorithms is that (1) the time of generating and deleting candidate itemsets is

different, the sizes of the candidate itemsets are also different; (2) the tree structures are different.

Theorem 2: The performance of COFI2 outperforms COFI in terms of running time and space.

Proof: When an item Q on a COFI-tree is processed, each itemset of the combinations generated by COFI may not contain this item Q, but it must contain the item of the root; however each itemset of the combinations generated by COFI2 must contain Q and the item of the root. Because the items used to generated combination in two algorithms are same, the number of combinations generated by COFI2 must less than that by COFI. Meanwhile, because the number of combinations generated by COFI2 is less, the merging of two combinations would cost less time. In addition, the nodes on the COFI-tree of algorithm COFI2 do not maintain participation count; this reduces memory and processing time.

Summarizing the above, the performance of COFI2 outperforms COFI in terms of time and space.

5. Experimental analyses

In this section, we evaluate the performance of the proposed algorithm TPT (TPT-tree and COFI2) and compare it with the algorithms DST (DST-tree and COFI) on four datasets: *connect4*, *connect20*, *kosarakt* and *T40I10D100K*. The datasets *connect4*, *kosarakt* and *T40I10D100K* were obtained from FIMI Repository [22]; we take the first 20 dimensions of *connect4* as the dataset *connect20*. *connect4* and *connect20* are dense datasets; *kosarakt* and *T40I10D100K* are sparse datasets. *T40I10D100K* is a synthetic dataset generated by IBM Data Generator and other three datasets are real-world datasets. All algorithms were written in Java programming language. The configuration of the testing platform is as follows: Windows XP operating system, 2G Memory, Intel(R) Core(TM) i3-2310 CPU @ 2.10 GHz; Java heap size is 1G.

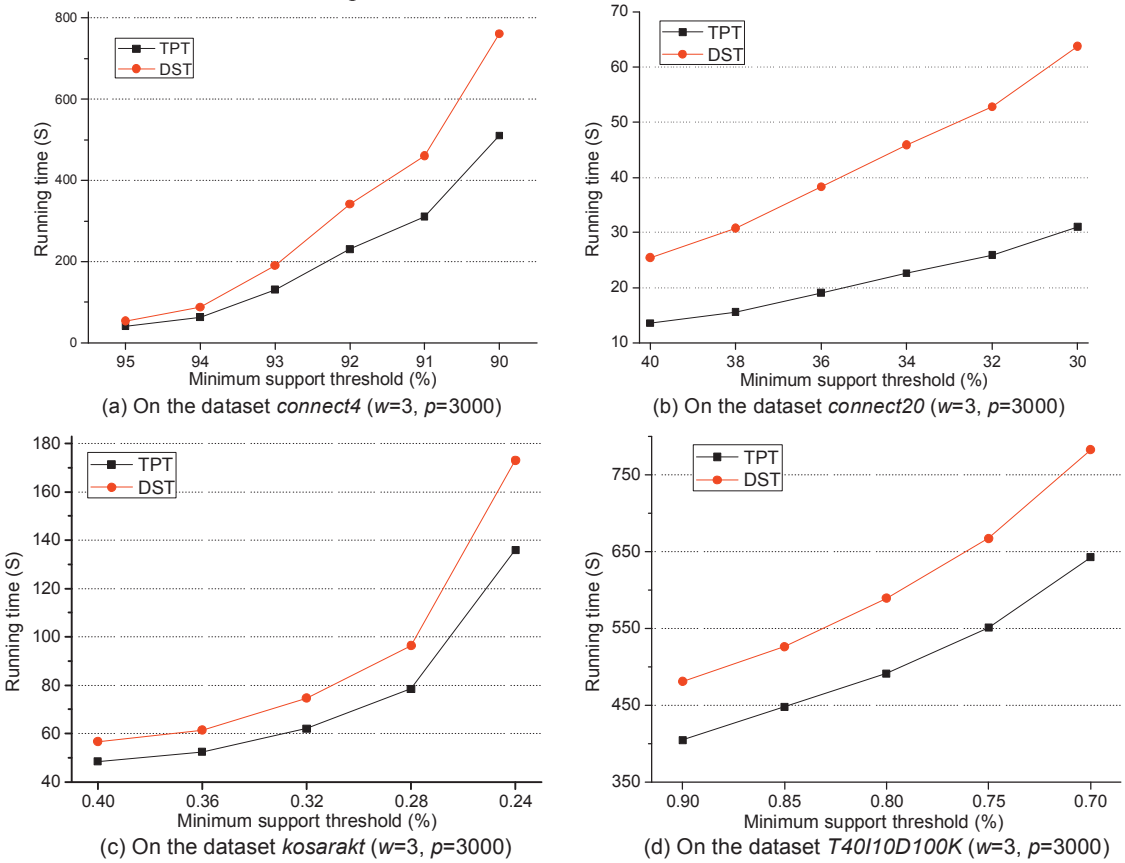


Fig. 7. Evaluation on varied minimum support threshold

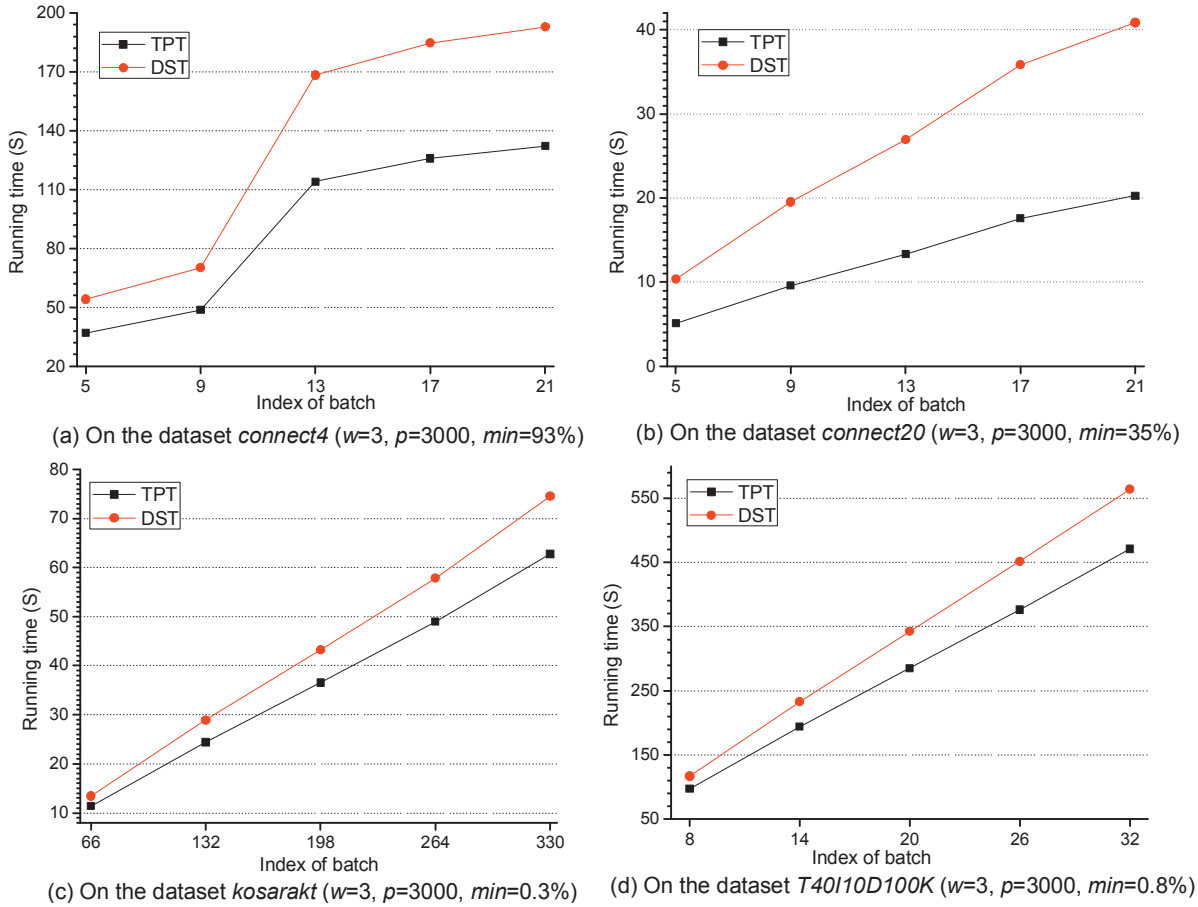


Fig. 8. Evaluation on accumulated batches

We evaluated the execution time of our algorithm under varied *minimum support threshold*. Fig. 7 shows the running time comparison of TPT and DST under four datasets, respectively. On the experiments of Fig. 7, the number of each batch data is set as 3000, thus the datasets *connect4*, *connect20*, *kosarakt* and *T40I10D100K* are divided into 23, 23, 330 and 33 panes, respectively; on these four datasets, mining operation was performed on 21, 21, 328 and 31 consecutive windows respectively; DST removed the obsolete nodes before performing each operation of mining frequent itemsets. Because there will be more frequent itemsets under a smaller minimum support threshold, the total running time will increase along with the decrease of the threshold; and from the results on Fig. 7, we can see that our algorithm TPT outperforms DST on different minimum support thresholds. For example, when the minimum support threshold (*min*) is 90% on the dataset *connect*4, TPT spends 510.141 seconds while DST spends 761.578 seconds.

Because data streams are continuous data flows along time, the accumulated performance of the algorithm should also be evaluated. Fig. 8 shows the running time comparison of TPT and DST under different batch index, and Fig. 9 is the running time comparison under varied batch-size.

From Fig. 8 we can see that the performance advantage of algorithm TPT accumulates along with the accumulation of batches, and it is stable.

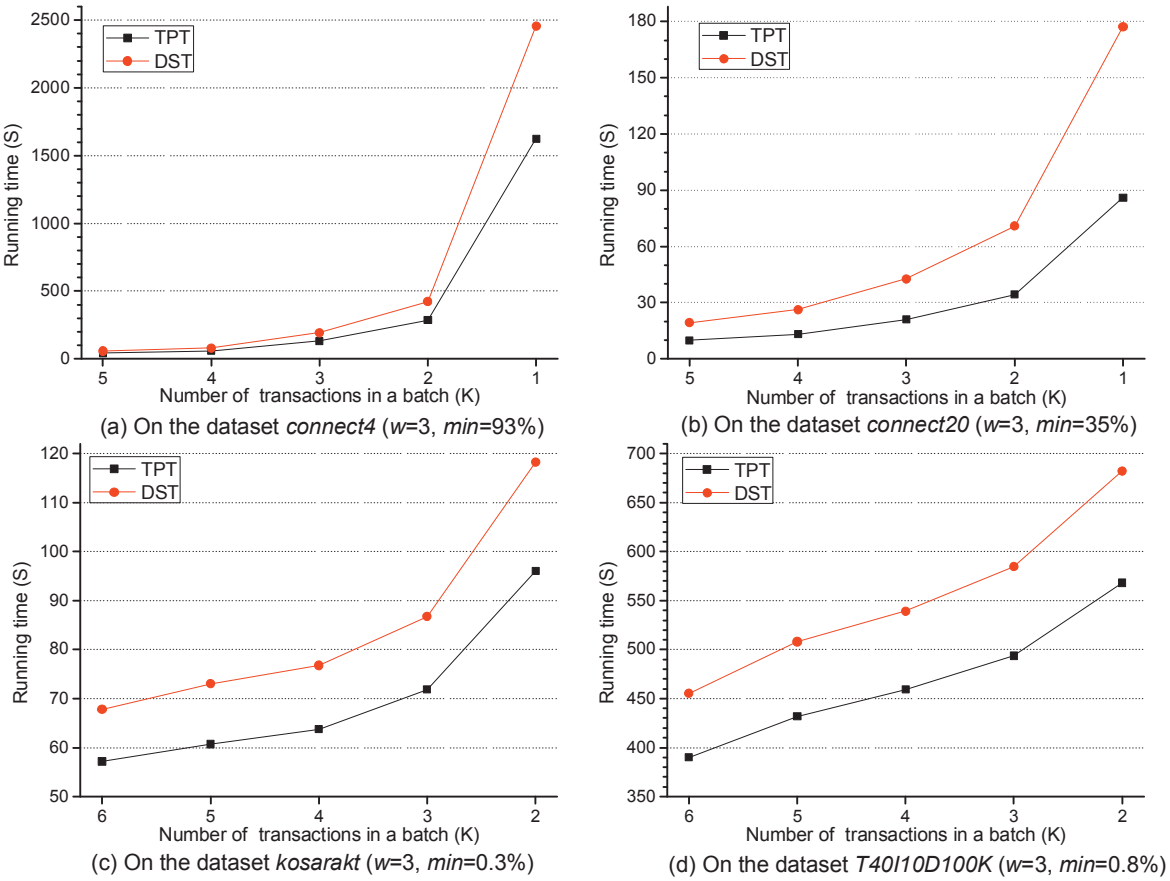


Fig. 9. Evaluation on varied batch-size

In Fig. 9, the running time decreases along with the increase of batch-size, because, for a fixed length dataset (which is the circumstance of our testing datasets), the more the number of transactions of each batch contains, the less the number of windows in a dataset will be. Fig. 9 shows that our algorithm TPT outperforms DST on varied batch-sizes. For example, when the batch-size (p) is 1000 on the dataset *connect*4, TPT spends 1625.25 seconds while DST spends 2454.734 seconds.

Concluding the above experiments, we can see that our proposed algorithm TPT has achieved a better performance than DST under varied minimum support thresholds and varied batch-sizes, and its advantage is stable along with the accumulation of the data flow process.

6. Conclusions

To improve the overall performance of the frequent itemsets mining algorithm over data streams, the efficiency of updating data and mining frequent itemsets should be considered. This study proposes a new data structure TPT-tree and gives the corresponding algorithm TPT. We also propose the algorithm COFI2 for mining frequent itemsets; meanwhile apply it to mine frequent itemsets over data streams. Theoretical and experimental analysis also shows that the performance of our proposed algorithm TPT (TPT-tree and COFI2) outperforms the algorithm DST (DST-tree and COFI) in terms of running time and memory space.

Acknowledgments

This work is supported by National Natural Science Foundation of P.R. China (Grant No. 61173163,

51105052), and Liaoning Provincial Natural Science Foundation of China (Grant No. 201102037).

References

1. R. Agrawal and R. Srikant, Fast algorithms for mining association rules in large databases, *in Proc. International Conference on Very Large Data Bases*, (Santiago, Chile, 1994), pp.487-487.
2. J. Han, J. Pei and Y. Yin, Mining frequent patterns without candidate generation, *in Proc. ACM SIGMOD International Conference on Management of Data*, (Dallas, TX, United states, 2000), pp.1-12.
3. M. El-hajj and O.R. Zaïane, COFI-tree mining: a new approach to pattern growth with reduced candidacy generation, *in Proc. IEEE International Conference on Frequent Itemset Mining Implementations*, 2003),
4. M. Song and S. Rajasekaran, A transaction mapping algorithm for frequent itemsets mining, *IEEE Transactions on Knowledge and Data Engineering* **4**(18) (2006) 472-481.
5. T. Hu, S.Y. Sung, H. Xiong, and Q. Fu, Discovery of maximum length frequent itemsets, *Information Sciences* **178**(1) (2008) 69-87.
6. B. Vo, T. Hong and B. Le, DBV-Miner: A Dynamic Bit-Vector approach for fast mining frequent closed itemsets, *Expert Systems with Applications* **39**(8) (2012) 7196-7206.
7. C.W. Lin, T.P. Hong and W.H. Lu, An effective tree structure for mining high utility itemsets, *Expert Systems with Applications* **38**(6) (2011) 7419-7424.
8. V.S. Tseng, C.W. Wu, B.E. Shie, and P.S. Yu, UP-Growth: An efficient algorithm for high utility itemset mining, *in Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (Washington, DC, United states, 2010), pp.253-262.
9. C.F. Ahmed, S.K. Tanbeer, B.S. Jeong, and Y.K. Lee, Efficient Tree Structures for High Utility Pattern Mining in Incremental Databases, *IEEE Transactions on Knowledge and Data Engineering* **21**(12) (2009) 1708-1721.
10. Y.C. Li, J.S. Yeh and C.C. Chang, Isolated items discarding strategy for discovering high utility itemsets, *Data and Knowledge Engineering* **64**(1) (2008) 198-217.
11. M. Liu and J. Qu, Mining high utility itemsets without candidate generation, *in Proc. 21st ACM International Conference on Information and Knowledge Management*, (Maui, HI, United states, 2012), pp.55-64.
12. C.W. Lin and T.P. Hong, A new mining approach for uncertain databases using CUFP trees, *Expert Systems with Applications* **39**(4) (2011) 4084-4093.
13. C.K. Leung and F. Jiang, Frequent pattern mining from time-fading streams of uncertain data, *in Proc. 13th International Conference on Data Warehousing and Knowledge Discovery*, (Toulouse, France, 2011), pp.252-264.
14. C.K. Leung, M.A.F. Mateo and D.A. Brajczuk, A tree-based approach for frequent pattern mining from uncertain data, *in Proc. 12th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, (Osaka, Japan, 2008), pp.653-661.
15. Z. Farzanyar, M. Kangavari and N. Cercone, Max-FISM: Mining (recently) maximal frequent itemsets over data streams using the sliding window model, *Computers & Mathematics with Applications* **64**(6) (2012) 1706-1718.
16. Y. Chi, H. Wang, S.Y. Philip, and R.R. Muntz, Catch the moment: maintaining closed frequent itemsets over a data stream sliding window, *Knowledge and Information Systems* **10**(3) (2006) 265-294.
17. F. Nori, M. Deypir and M.H. Sadreddini, A sliding window based algorithm for frequent closed itemset mining over data streams, *Journal of Systems and Software* (2012)
18. C.K.S. Leung and Q.I. Khan, DSTree: A tree structure for the mining of frequent sets from data streams, *in Proc. IEEE International Conference on Data Mining*, (Hong Kong, China, 2007), pp.928-932.
19. C. Leung and D. Brajczuk, Efficient Mining of Frequent Itemsets from Data Streams, *Sharing Data, Information and Knowledge* (2008) 2-14.
20. S.K. Tanbeer, C.F. Ahmed, B. Jeong, and Y. Lee, Sliding window-based frequent pattern mining over data streams, *Information Sciences* **179**(22) (2009) 3843-3865.
21. M. Deypir and M.H. Sadreddini, A dynamic layout of sliding window for frequent itemset mining over data streams, *Journal of Systems and Software* **85**(3) (2012) 746-759.
22. B. Goethals, Frequent itemset mining dataset repository, http://fimi.cs.helsinki.fi/data/, accessed Oct. 2010.