Finding Maximal Frequent Itemsets over Online Data Streams Adaptively

Daesu Lee
Department of Computer Science
Yonsei University, Korea
dslee@database.yonsei.ac.kr

Wonsuk Lee
Department of Computer Science
Yonsei University, Korea
leewo@database.yonsei.ac.kr

Abstract

Due to the characteristics of a data stream, it is very important to confine the memory usage of a data mining process regardless of the amount of information generated in the data stream. For this purpose, this paper proposes a ***CP-tree*** *(**Compressed-prefix tree**) that can be effectively used in finding either frequent or maximal frequent itemsets over an online data stream. Unlike a prefix tree, a node of a CP-tree can maintain the information of several itemsets together. Based on this characteristic, the size of a CP-tree can be flexibly controlled by merging or splitting nodes. In this paper, a mining method employing a CP-tree is proposed and an adaptive memory utilization scheme is also presented in order to maximize the mining accuracy of the proposed method for confined memory space at all times. Finally, the performance of the proposed method is analyzed by a series of experiments to identify its various characteristics.*

1. Introduction

With the rapid development of information technology, the amount of information has been increasing faster than ever in various domains. Furthermore, depending on the characteristics of an application domain, its information is produced in diverse forms. A data stream is one of such forms and is a massive unbounded sequence of data elements continuously generated at a rapid rate. Due to this reason, it is impossible to maintain all the elements of a data stream. Consequently, on-line data stream processing should satisfy the following requirements [6]. First, each data element should be examined at most once to analyze a data stream. Second, memory usage for data stream analysis should be restricted finitely although new data elements are continuously generated in a data stream. Third, newly generated data elements should be processed as fast as possible to produce the up-to-date analysis result of a data stream, so that it can be instantly utilized upon request. To satisfy these requirements, data stream processing sacrifices the correctness of its analysis result by allowing some error.

Recently, various algorithms [7,9,10] are actively proposed to extract different types of knowledge embedded in a data stream. The ***sticky sampling*** method [11], the ***Lossy Counting*** algorithm [11] and the ***estDec*** method [4] focus on finding frequent itemsets over a data stream. In the Lossy Counting algorithm, to reduce the memory usage of a mining process, the counts of frequent itemsets can be kept in a secondary storage and only a buffer for the batch-processing of transactions is kept in main memory. As the buffer is enlarged, more number of newly generated transactions can be batch-processed together, so that the algorithm is more efficiently processed. When the number of frequent itemsets is large, accessing the information of frequent itemsets in a secondary disk needs more time. Due to this reason, this algorithm is not appropriate for an online data stream.

For finding frequent itemsets, we have proposed the *estDec* method [4] to minimize the number of itemsets to be monitored. In this method, an itemset whose support is greater than a predefined significant support S_{sig} ($S_{sig} \leq S_{min}$) is regarded as a significant itemset where S_{min} is a given minimum support. Each significant item-set in a data stream is represented by an individual node of a prefix tree. Consequently, the resulting set of frequent itemsets in a data stream can be found instantly at any moment. As the number of significant itemsets in a data stream is increased, the size of the prefix tree that represents these itemsets become larger. Consequently, the memory usage of the prefix tree is also increased. Once the size of a prefix tree becomes larger than the size of available memory space, no new significant itemset can be inserted to the prefix tree, so that the *estDec* method will not work properly any longer.

An itemset tree [8] is proposed to reduce the memory usage of finding frequent itemsets in an incrementally enlarged data set. Unlike a prefix tree, a node of an itemset tree maintains the subset of items in an individual transaction. Given an itemset tree for a set of transactions, if there exists a node representing any subset of items in a newly added transaction *T*, the node is shared to represent the subset of the items in *T*. Only a node corresponding to the remaining items of *T* is newly inserted into the itemset tree. By sharing the common subsets of transactions in a data set, it is possible to reduce the required size of memory space. Consequently, the size of an itemset tree is smaller than that of a prefix tree for the same data set. However, like a prefix tree, an itemset tree has no mechanism to manipulate its size adaptively to confined memory space.

To cope with this drawback of conventional tree structures, this paper proposes a ***CP-tree*** (***Compressed-Prefix tree***) to replace the role of a prefix tree in the *estDec* method. In addition, this paper also introduces the extended version of the *estDec* method, namely *estDec*+ for a CP-tree. Unlike a prefix tree, two or more nodes of a prefix tree can be merged into a single node of a CP-tree as long as the support difference of their corresponding itemsets is within a predefined threshold called a *merging gap threshold* $\delta \in (0,1)$. In such a node of a CP-tree, only the counts of two representative itemsets are maintained while those of the remaining itemsets are estimated based on the counts of the representative itemsets. By adaptively controlling the value of δ, the number of nodes in a CP-tree can be changed. As the value of δ is increased, more significant itemsets can be represented by a single node of a CP-tree. Consequently, the size of the CP-tree is reduced while the mining result of the *estDec*+ method becomes less accurate. However, since the size of a CP-tree can be flexibly controlled by merging or splitting nodes, the *estDec*+ method can fully utilize confined memory space at all times. This capability is valuable when the number of significant itemsets is fluctuated heavily.

The rest of this paper is organized as follows: Section 2 reviews the *estDec* method. Section 3 proposes the structure and operations of a CP-tree in detail. Section 4 introduces the *estDec*+ method which employs a CP-tree to find frequent or maximal frequent itemsets over an on-line data stream. In Section 5, the performance of the *estDec*+ method is evaluated by a series of experiments. Finally, Section 6 concludes this paper.

2. Preliminaries

The *estDec* method examines each transaction in a data stream one by one without any candidate generation and keeps track of the occurrence count of an itemset in the transactions generated so far by a monitoring tree whose structure is a prefix tree [1,3]. Given the current data stream D_k, a prefix tree P_k has the following characteristics:

i) A prefix tree P_k has a root node n_{root} with a "*null*" value and each node except the root node has an item $i \in I$.
ii) Given a node n having an item $i_n \in I$ in a prefix tree P_k, let $s = n_{root} \rightarrow n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_v \rightarrow n$ be the sequence of nodes in the path from the root n_{root} to the node n and let each node n_j have its item $i_j \in I$ $(1 \le j \le v)$. The node n represents an itemset $e_n = i_1 i_2 \ldots i_v i_n$ and maintains the current count $C_k(e_n)$.

The *estDec* method has two major operations: ***delayed-insertion*** and ***pruning*** operations. Monitoring the count of a new itemset is started only in the following two cases. The first case is when a new 1-itemset appears in a newly generated transaction T_k. In this case, monitoring its count is instantly started by inserting it to the monitoring tree P_k without any estimation. The second case is when an insignificant itemset just becomes a significant one due to its appearance of T_k. Since it becomes an significant itemset, it should be inserted into P_k for further monitoring. To find such an n-itemset e $(n \ge 2)$, only when all of its $(n-1)$-subsets are currently maintained in P_k, the current support of the itemset e is estimated by those of its $(n-1)$-subsets. If the estimated support is greater than a predefined insertion support S_{ins}, the itemset e is inserted. Since the total number of the $(n-1)$-subsets is n, let $\{c_1, c_2, \ldots, c_n\}$ be the set of the current counts of the $(n-1)$-subsets monitored in P_k. The estimated current count $\hat{C}_k(e)$ of the itemset e is obtained by the largest one i.e. $\hat{C}_k(e) = max(c_1, \ldots, c_n)$. The upper bound of the estimation error for e is

$$|C_k(e) - \hat{C}_k(e)| = \max(c_1, ..., c_n) - \min(c_1, ..., c_n)$$

The above procedure is a ***delayed-insertion*** operation. The upper bound of this estimation error is proven to be ignorable when k becomes infinite [4]. In this paper, estimating the current count of such an insignificant itemset is called as *inserting-count estimation*.

On the other hand, a pruning operation is performed when the current support of an itemset maintained by P_k becomes less than a predefined pruning support S_{prn}. The itemset is regarded as an insignificant itemset that cannot be a frequent itemset in the near future. Upon identifying such an itemset, the node representing such an itemset and all of its descendent nodes are pruned from P_k based on the anti-monotonicity of a frequent itemset [4]. Since P_k is located in main memory, its size should be kept smaller than the confined space of main memory at all times. However, its size totally depends on the density of significant itemsets in the current data stream D_k with respect to S_{sig}. Therefore, once the size of a prefix tree exceeds to the size of the confined memory space, it is impossible to monitor any new significant itemset by the delayed-insertion operation. Due to this reason, the mining accuracy of the *estDec* method can be degraded without any upper bound in this situation.

3. Compressible Prefix Tree: CP-tree

3.1. CP-tree

To reduce the size of a prefix tree, the information represented by the prefix tree needs to be compressed. Two consecutive nodes by a prefix tree are merged in a CP-tree when the current support difference between their corresponding itemsets is less than or equal to a ***merging gap threshold*** $\boldsymbol{\delta} \in (0,1)$. Ultimately, a subtree of a prefix tree can be compressed into a node of a CP-tree.

Definition 1. A mergeable subtree
Suppose P_k be a prefix tree and S be a subtree of P_k. Let e_v denote the itemset represented by the root of S and e_j denote an itemset represented by a node of S. A leaf node of S is not necessarily to be a leaf node of P_k. Given a merging gap threshold δ, if all the nodes of the subtree S satisfy the following condition, the subtree S is a *mergeable subtree* and compressed into a node of a CP-tree Q_k that is equivalent to P_k.

$$|C_k(e_v) - C_k(e_j)|/|D|_k \leq \delta,\ 1 \leq j \leq |S|$$

where $|S|$ denotes the number of nodes in S. □

The detailed structure of a node in a CP-tree is defined in Definition 2.

Definition 2. CP-node structure
Given a mergeable subtree S of a prefix tree P_k in D_k, let a CP-tree Q_k be equivalent to P_k. To represent the information of S in Q_k, a node m of Q_k maintains the following four entries $m(\tau, \pi, c_S, c_L)$ as follows:

i) *item-list* τ : The items of the nodes in each level of S are lexicographically ordered and these level-wise lists of items are ordered according to their levels. Let $|\tau|$ denote the number of items in τ and the j^{th} item in τ is represented by $\tau[j]$ $(1 \leq j \leq |\tau|, |\tau|=|S|)$. The item $m.\tau[1]$ is corresponding to the itemset represented by the root node of S. This itemset is called as the *shortest itemset* of the node m and denoted by $m.e_S$. On the other hand, the last item $m.\tau[|S|]$ is corresponding to the itemset represented by the right-most leaf node in the lowest level of S. The itemset is called as the *longest itemset* of the node m and denoted by $m.e_L$.

ii) *parent-index list* π : π maintains an entry of a form $p.q$ where p denotes a node identifier of Q_k and q denote an index of the item-list τ of the node p. Suppose a node n_x with an item $x \in I$ is the parent of a node n_y with an item $y \in I$ in the mergeable subtree S. The nodes n_x and n_y of P_k are represented by their corresponding items x and y in the item-list τ of the node m. Let a and b denote the item-list indexes of the items x and y respectively, i.e., $m.\tau[a]=x$ and $m.\tau[b]=y$. The parent-child relationship of the nodes n_x and n_y in P_k is modeled by the two lists τ and π in the node m since $m.\tau[b]=y$ and $m.\pi[b]=m.a$ imply the parent of the item y is $m.\tau[a]=x$. On the other hand, suppose the parent of the root of S is a node n_z with an item z in P_k and the node n_z is in another mergeable subtree $\overline{S}$. If $\overline{S}$ is represented by a node $\overline{m}$ of Q_k and the item-list index of the node n_z in $\overline{m}$ is q_z i.e., $\overline{m}.\tau[q_z]=z$, the entry of $m.\tau[1]$ is set to $\overline{m}.q_z$.

iii) *largest count* c_L : the current count of the shortest itemset e_S.

iv) *smallest count* c_S : if $|S|=1$, $c_S=c_L$. Otherwise, the current count of the longest itemset e_L. □

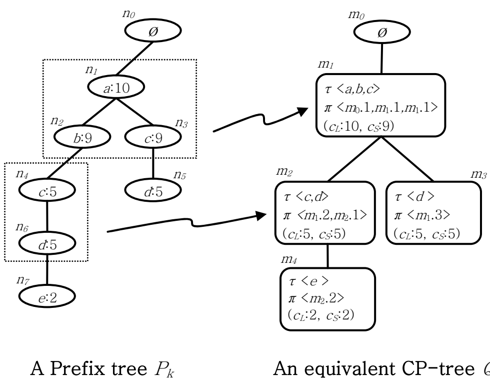


Figure 1. A CP-tree and its equivalent prefix tree (S_{min}=0.1, δ=0.2, $|D|$=10)

Figure 1 shows a prefix tree P_k and its equivalent CP-tree Q_k. The subtree formed by the nodes n_1, n_2, and n_3 of P_k are compressed into the node $m_1(\tau=\langle a,b,c \rangle, \pi=\langle m_0.1, m_1.1, m_1.1 \rangle, c_L, c_S)$ of Q_k. This is because the current support difference between the root node n_1 of the subtree and each of its child nodes n_2 and n_3 is less than δ. The root of the subtree represented by m_1 is $\tau[1]=a$ which is corresponding to the node n_1 of P_k. Its parent node $\pi[1]=m_0.1$ is the node m_0 of the CP-tree Q_k. The fact that the node n_1 is the parent of the node n_2 in P_k can be inferred by $m_1.\tau[2]=b$ and $m_1.\pi[2]=m_1.1$ which imply that the parent of the item b is $m_1.\tau[1]$ i.e. a. The shortest and longest itemsets of m_1 are a and ac respectively.

3.2. Merged-count Estimation

Given the item-list $m.\tau=<i_1,i_2,...,i_n>$ of a node m in a CP-tree, let e_{i_j} denote the itemset represented by i_j $(1\leq j\leq n)$. By the two counters c_S and c_L of the node, it is possible to trace the current supports of at most two itemsets i.e. the shortest and longest itemsets e_{i_1} and e_{i_n} as precisely as the *estDec* method does. Therefore, if more than three itemsets are compressed into a single node, the current counts of the remaining itemsets can be estimated by a formula

$$C(e_{i_j}) = \lceil c_L - f(m,j) \rceil \ (2 \leq j \leq n-1)$$

where $f(m, j)$ denotes a count estimation function that can model the count $C(e_{i_j})$ in terms of the counts c_L and c_S of the shortest and longest itemsets e_S and e_L. A function meaningful in an application domain can be employed to define the function $f(m, j)$.

To distinguish the inserting-count estimation defined in Section 2, the above estimation is called as *merged-count estimation*. When the current count of an itemset traced by a node of a CP-tree is estimated by the above mechanism, there must be an error count but the possible range of this error count is totally influenced by the value of δ.

3.3. CP-tree Maintenance

Given a minimum support S_{min} and a merging gap threshold δ, let Q_{k-1} denote a CP-tree at a data stream D_{k-1}. As in the *estDec* method, when a new transaction T_k is generated, those paths of Q_{k-1} that are induced by the items of T_k are traversed respectively and the counts of all the nodes in the paths are updated. Traversing a CP-tree is virtually the same as traversing a prefix tree in [4]. The items of T_k are lexicographically ordered and matched with Q_{k-1} by a depth-first manner as in a prefix tree. Among the items in the item-list τ of a node m, a *leaf-level item* is defined as an item i whose item-list index j does not appear in the parent-list of the node m. i.e. $j\neq m.\pi[l]$ $\forall l$, $1\leq l\leq |\tau|$ where $m.\tau[j]=i$.

Upon visiting a node m of a CP-tree, let m_p denote its parent node and i_p denote the last matched leaf-level node in m_p. If $m.\pi[1]=m_p.i_p$, the first item $m.\tau[1]$ is compared with those remaining items of T_k that are not yet matched. If the above condition is not met or the item $m.\tau[1]$ is not matched, the search is terminated and return to the parent node of the node m. If $m.\tau[1]$ is matched, the largest count c_L of the node m is incremented by one. Let the item $m.\tau[1]$ be the first common item ci_1 in the item-list $m.\tau$ and the remaining items of T_k. Among the remaining items of T_k, only those items that are after the item ci_1 are considered to be matched further. Subsequently, find the second common item ci_2 in both the remaining items of τ and those items in T_k that are considered to be matched. Let the item-list indexes of the common items ci_1 and ci_2 be j_1 and j_2. Only when the parent of ci_2 is ci_1, i.e. $m.\pi[j_2]=m.j_1$, let ci_2 be a new ci_1 and find a new ci_2 by the same manner until there is no such ci_2 in $m.\tau$. If the above recursive search in the node m is terminated by reaching one of the leaf-level items of the node m, the child nodes of the node m are searched continuously. When the last ci_2 item is not an leaf-level item, the next common item in the remaining items of the two lists is searched by the same way. This procedure is recursively repeated until there is no item to be matched in either of the lists. Furthermore, only when the last item $\tau[|\tau|]$ is matched as a leaf-level item, the smallest count $m.c_S$ is also increased. If none of the leaf-level items are matched, the search is terminated and returned to the parent of the node m. The traversing algorithm is presented in Section 4.

While traversing a node m of Q_{k-1}, two major operations: *node-merge* and *node-split* can be performed additionally. Let $\overline{m}$ denote the parent node of m. A node-merge operation is only invoked in the following two cases. One is when the current support difference between the shortest itemset of $\overline{m}$ and the longest itemset of the node m becomes less than or equal to δ i.e. $(\overline{m}.c_L - m.c_S)/|D|_k \leq \delta$. This case happens only when the difference between the two counts remains the same and only $|D|_k$ is increased. The other one is when a new significant itemset e is identified by the inserting-count estimation process, so that a new node for the itemset needs to be inserted as a child of the node m. If the current support difference of the estimated support of the new significant itemset and the largest itemset of the node m is less than or equal to δ, the new node is merged into the node m. The detailed steps of a node-merging operation are described in Figure 2. On the other hand, a node m of a CP-tree is split into two different nodes when the support differ-ence between its shortest and longest itemsets becomes greater than δ, i.e., $(m.c_L-m.c_S)/|D|_k$ $>\delta$. The difference is enlarged when only $m.c_L$ is increased. When a node is split, each of the leaf-level items of the node $m.\tau$ is separated as an individual node of a CP-tree. The detailed steps of a node-split operation are described in Figure 3.

```
node_merge($m_1$ and $m_2$)
1 append $m_2.\tau$ to the end of $m_2.\tau$;
2 let $a \leftarrow m_1.|\tau| + 1$;
3 append $m_2.\pi[1]$ to $m_1.\pi[a]$
4 for each entry in $m_2.\pi[j]$ ($2 \le j \le |m_2.\tau|$) where $m_2.\pi[j] = m_2.q$
    5 add an entry $m_1.(q + m_1.|\tau|)$ to $m_1.\pi[a + j - 1]$;
6 make the child nodes of $m_2$ be those of $m_1$;
7 Prune $m_2$ from $Q_k$;
```

Figure 2. Node_merge operation

```
node_split($m$)
let $b_1,\ldots b_w$ be the leaf-level items of a node $m$ and $q_1,\ldots q_w$ be their item-list indexes
1 for each $b_k$ ($1\le k\le w$)
2   create a child node $m_k$ of the $m$ by initializing;
3   $m_k.\tau[1] \leftarrow b_k$;
4   $m_k.\pi[1] \leftarrow m.q_k$;
5   $m_k.c_L \leftarrow$ estimate count of $e_{b_k}$;
6   eliminate those entries from $m.\tau$ and $m.\pi$ corresponding to $b_k$;
7 $m.c_S \leftarrow$ estimate count of the new longest itemset of $m$;
```

Figure 3. Node_split operation

4. Finding Maximal Frequent Itemsets

In this section, a modified version *estDec*+ of the *estDec* method is proposed. The proposed method is basically based on the *estDec* method but the underlying memory structure of significant itemsets is changed from a prefix tree to a CP-tree.

4.1. *estDec*+ Method

In the *estDec* method, the weight of information in a data stream is differentiated over time by a decay mechanism [4], so that it can find recently frequent itemsets over the data stream [4]. To concentrate on developing a mining method based on a CP-tree over a data stream, this paper does not mention about the decay mechanism precisely. However, the same decay mechanism applied to a prefix tree in the *estDec* method can also be employed to a CP-tree in the *estDec*+ method. Furthermore, the *estDec*+ method employs delayed-insertion and pruning operations to trace the current supports of only significant itemsets. However, the two thresholds: an insertion support S_{ins} and a pruning support S_{prn} used in the *estDec* method are denoted by a *significant support* $S_{sig}\in(0,S_{min})$ in this paper. In other words, delayed-insertion and pruning operations in the *estDec*+ method are performed with respect to S_{sig}.

If every node which represents a significant itemset is allowed to be merged, an infrequent but significant itemset e, i.e. $S_{sig}\le S_k(e)\le S_{min}$ can be merged with frequent itemsets. This can cause false positive or false negative errors in the course of merged-count estimation. To reduce these types of error, a node whose current support is less than a predefined threshold called a merging threshold S_{merge} ($\ge S_{min}$) is not considered as a candidate for a node-merge operation. Since a node of a CP-tree can represent multiple itemsets together, the current supports of all the itemsets induced by a node are estimated to find any maximal frequent itemset in the node. As the gap between S_{merge} and S_{min} is enlarged, the possibility that a node representing a maximal frequent itemset is not merged with any other node becomes high. Consequently, the counts of all the maximal frequent itemsets hardly include any estimation error caused by the merged-count estimation.

Like the *estDec* method, the *estDec*+ method consists of four phases: parameter updating, node restructuring, itemset insertion, and frequent itemset selection. When a new transaction T_k in a data stream D_{k-1} is generated, these phases except the frequent itemset selection phase are performed in sequence. The frequent itemset selection phase is performed only when the up-to-date result set of frequent or maximal frequent itemsets is requested.

Parameter updating phase: The total number of transactions in the current data stream D_k is updated.
Count updating & node restructuring phase: This phase is performed by traversing Q_{k-1} according to the lexicographic order of the items in T_k. For each visited node m, its smallest and largest counts $m.c_S$ and $m.c_L$ may be incremented as described in Section 3.3. If the updated support of the shortest itemset $m.e_S$ becomes less than S_{sig} i.e. $m.c_L/|D|_k<S_{sig}$, the node m and all of its descendent nodes are pruned since all the itemsets represented by these nodes are turned out to be insignificant. If the updated support of the longest itemset $m.e_L$ is less than S_{merge} or the support difference between the itemsets $m.e_S$ and $m.e_L$ becomes greater than δ i.e. $m.c_S/|D|_k<S_{merge}$ or $(m.c_L-m.c_S)/|D|_k>\delta$, the node m is split. On the other hand, if the updated support of $m.e_L$ is greater than S_{merge} and the support difference between $m.e_S$ and and $m^*.e_L$ of its parent node m^* is less than or equal to δ i.e. $m.c_S/|D|_k\ge S_{merge}$ and $(m^*.c_L-m.c_S)/|D|_k\le\delta$, these two nodes m and m^* are merged.
Itemset insertion phase: The itemset insertion phase is performed to insert any new significant itemset which has not been maintained in Q_{k-1}. As in the *estDec* method, every single item should be maintained by Q_{k-1}. Consequently, when T_k contains any new item that is not in Q_{k-1} yet, a new node m for the item $i\in T_k$ is inserted as follows:

$$m.\tau=<i>,\ m.\pi=<root>,\ m.c_S = m.c_L = 1$$

Subsequently, any insignificant item whose current support is less than S_{sig} is filtered out in the transaction T_k. Let the filtered transaction be denoted by $\overline{T}_k$. The monitoring tree Q_{k-1} is traversed for the filtered transaction $\overline{T}_k$ once again to find out any new significant itemset induced by the items in $\overline{T}_k$. By the same way as in the *estDec* method, for each significant *n*-itemset $e=i_1i_2\ldots i_n$ ($n \geq 1$) represented by a node *m* of Q_k, every (n+1)-itemset $\bar{e}$ which is $\bar{e} \cup i_{n+1} \in \overline{T}_k$ is examined. First of all, check whether all of its *n*-subsets of the itemset $\bar{e}$ are currently maintained in Q_{k-1}. If the above condition is satisfied, the current support of the itemset $\bar{e}$ is estimated as $\hat{C}(\bar{e})$ as described in Section 2. If $\hat{C}(\bar{e}) \geq S_{sig}$, a new node *w* corresponding to the itemset $\bar{e}$ is inserted to Q_k. The detailed description of this estimation process is presented in [4]. The entries of the new node *w* are initiated as follows:

$$w.\tau = \langle i_{n+1} \rangle,\ w.\pi = \langle m.q \rangle,\ w.c_S = w.c_L = \hat{C}(\bar{e})$$

where *q* denotes the item-list index of the item i_n in the node *m*.

Maximal frequent itemset selection phase: This phase retrieves all the currently frequent or maximal frequent itemsets by traversing the monitoring tree Q_k.

As in the *estDec* method, all the nodes whose largest counts c_L are less than $|D|_k * S_{sig}$ can be pruned altogether by traversing the entire monitoring tree. It is called a *force-pruning* operation, and can be performed periodically.

```
traverse($m$, $m_p$, $q$, $T$, $y$)
  $m_p$: the parent node of a node $m$
  $T[k]$: the $k$th item of a transaction $T$ in lexicographical order
  $q$: the item-list index of the last leaf-level item $T[y-1]$ in node $m_p$
--------------------------------------------------------------------------
if $m.\pi[1] = m_p.q$ and $m.\tau[1] = T[y]$ then
  $m.c_L \leftarrow m.c_L + 1$;
  if $(m.c_L / |D|_k) < S_{sig}$ then
    pruning $m$; // eliminate $m$ and all of its descendent nodes
  else
    $y \leftarrow y + 1$; $x \leftarrow 2$;
    $ci_1 \leftarrow m.\tau[1]$;
    $ci_2 \leftarrow$ find_com_item($m.\tau$, $x$, $T$, $y$);
    while $m.\pi[ci_2] = m.v$ and $x \leq |\tau|$ and $y \leq |T|$ do
      if $ci_2$ is a leaf-level item then /* $v \cong m.\tau[ci_1]$ */
        if $x = |\tau|$ then
          $m.c_S \leftarrow m.c_S + 1$;
          if $(m_p.c_L - m.c_S)/|D|_k \leq \delta$ and $m.c_S/|D|_k \geq S_{merge}$ then
            node_merge($m_p$, $m$);
        if $(m.c_L - m.c_S)/|D|_k > \delta$ then
          node_split($m$);
        if $\exists$ an unvisited child $m_c$ of $m$
          traverse($m_c$, $m$, $w$, $T$, $y$); /* $m.\tau[ci_2] \cong w$ */
        else return;
      else
        $ci_1 \leftarrow ci_2$;
        $ci_2 \leftarrow$ find_com_item($m$, $\tau$, $x$, $T$, $y$);
--------------------------------------------------------------------------
find_com_item($m.\tau$, $x$, $T$, $y$) returns the first common item $ci$ in the
items $m.\tau[i]$ ($x \leq i \leq |m.\tau|$) and $T[j]$ ($y \leq j \leq |T|$) where $x$ and $y$ are
updated, s.t. $\tau[x-1] = T[y-1] = ci$
```

Figure 4. Traverse operation

4.2. Adaptive Memory Utilization

Since information embedded in a data stream is more likely to be changed over time, the number of currently significant itemsets is continuously varied. However, the size of memory space for a CP-tree is confined. In order to minimize the estimation error caused by the merged-count estimatioin, it is very important to keep the value of δ as small as possible. The size of a CP-tree is inversely proportional to the value of δ. In order to adaptively control the memory utilization of the *estDec*+ method, the value of δ should be dynamically changed in the parameter update phase of the *estDec*+ method.

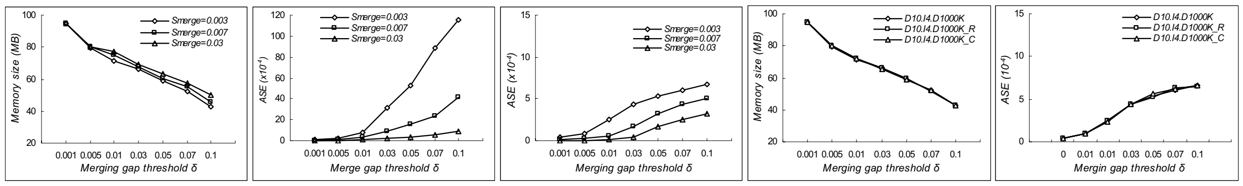
Based on the ratio of the current memory usage over given confined memory space, the value of δ is dynamically changed in the parameter updating phase of the *estDec*+ method. Let M_U and M_L denote the upper and lower bounds of desired memory usage respectively for given confined memory space M_A. Whenever the current memory usage M_C satisfies the following conditions, the new value δ^{new} of a merging gap threshold is adjusted adaptively as follows:

$$\delta^{new} = \begin{cases} \delta^{old} + \alpha & \text{if } M_C > M_U \\ \delta^{old} - \alpha & \text{if } M_C < M_L \end{cases} \quad (M_A > M_U > M_L > 0)$$

where α denotes the step-wise increment of δ for each adaptation and is defined by a user. As the values $M_A - M_U$ and $M_U - M_L$ become larger, the value of α can also be larger. As long as the current memory usage M_C of a CP-tree becomes greater than the upper bound M_U, the value of δ is increased. As a result, more nodes can be merged and the size of the CP-tree is reduced. On the other hand, when M_C becomes less than the lower bound M_L, the value of δ is decreased to enhance the mining accuracy of the CP-tree, so that the size of the CP-tree is increased. By setting the lower bound M_L high enough, the memory utilization of the *estDec*+ method is kept high. On the other hand, by setting the upper bound M_U low enough, the *estDec*+ method can be executed without causing any memory overflow.

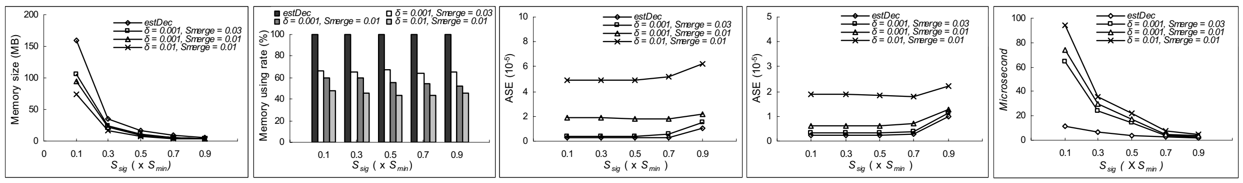
5. Performance Evaluation

In this section, the performance of the *estDec*+ method is analyzed by two data sets: *T10.I4.D1000K* and *WebLog*. The data set *T10.I4.D1000K* is generated by the same method as described in [2]. The data set *WebLog* is a real web-page access log data. The con-



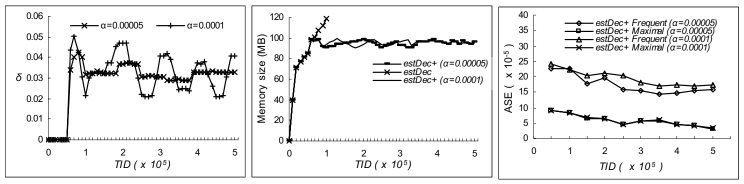
(a) Memory Usage **(b) Mining accuracy of frequent itemsets** **(c) Mining accuracy of maximal frequent itemsets** **(d) Memory Usage** **(e) Mining accuracy of maximal frequent itemsets**

Figure 5. Performance of the *estDec*+ method by varying *δ*



(a) Memory Usage **(b)Memory requirement** **(c) Mining accuracy of frequent itemsets** **(d) Mining accuracy of maximal frequent itemsets** **(e) Runtime**

Figure 6. Performance comparison of the *estDec*+ method with the *estDec* method



(a) Change of **(b) Memory usage** **(c) Mining accuracy**

Figure 7. Performance of the *estDec*+ method with a dynamic adaptation technique of *δ*

secutive web-pages accessed by a user are considered as a semantically atomic unit of activities, i.e., a transaction. The total number of items, i.e., the number of web-pages, is 545. The minimum, maximum, and average lengths of a transaction in the data set *WebLog* are 2, 30, and 5 respectively. In addition, the total number of transactions is 500,000. In all experiments, the transactions of a data set are looked up one by one in sequence to simulate the environment of an online data stream and a force-pruning operation is performed in every 1000 transactions. In addition, the value of S_{min} is set to 0.001 and the count estimation function $f(m,j)$ is defined as follows:

$$f(m,j) = (c_L - c_S) \times \sum_{l=1}^{|e_{i_j}|-|e_S|} \frac{1}{l} \Bigg/ \sum_{l=1}^{|e_L|-|e_S|} \frac{1}{l}$$

All experiments are performed on a 1.8 GHz Pentium PC machine with 512MB main memory running on Linux 7.3 and all programs are implemented in C.

Figure 5 shows the performance of the *estDec*+ method on the data set *T10.I4.D1000K* by varying *δ*. The value of S_{sig} is set to $0.1 \times S_{min}$ in this experiment. The memory usage of the *estDec*+ method is illustrated in Figure 5-(a) after the memory usage is stablized. To measure the accuracy of the *estDec*+ method, a term *average support error* $\boldsymbol{ASE(R_2|R_1)}$ [4] is employed. As the value of $ASE(R_2|R_1)$ gets smaller, the mining result R_2 is more similar to R_1. In Figure 5-(a), the memory usage of the *estDec*+ method gets smaller as the value of *δ* gets larger since more nodes are merged. The $ASE(R_{estDec+}|R_{Apriori\_MFI})$ in Figure 5-(c) is much smaller than the $ASE(R_{estDec+}|R_{Apriori})$ in Figure 5-(b) since most of maximal frequent itemsets are more accurately monitored without merged-count estimation. As the value of S_{merge} is set to be smaller for the same value of *δ*, the *ASE* is increased but the memory usage is decreased. This is because more nodes are merged as the value of *δ* is increased or the value of S_{merge} is decreased. Figure 5-(d) and 5-(e) shows that *estDec*+ method is order-independent. For this experiment, *T10.I4.D1000K_R* is generated with the reversely ordered transactions of *T10.I4.D1000K* while *T10.I4. D1000K_C* is generated with two consecutive datasets, i.e., one with the transactions having odd number *TID* and the other with the transactions having even number *TID*. The value of S_{merge} is set to 0.003.

In Figure 6, the performance of the *estDec*+ method is closely compared with that of the *estDec* method on the data set *T10.I4.D1000K*. In this experiment, the value of *δ* is fixed. For the same value of S_{sig}, the memory usage of the *estDec*+ method is always less than that of the *estDec* method. Figure 6-(b) shows the memory requirement of the *estDec*+ method. The requirement is represented by the ratio of the size of memory space required by the *estDec*+ method over that required by the *estDec* method to execute the same dataset. By varying the value of S_{sig}, Figure 6-(c) and Figure 6-(d) illustrate the mining accuracy of finding frequent and maximal frequent itemsets respectively. When the value of S_{merge} gets higher, the *ASE* of the *estDec*+ method becomes closer to that of the *estDec* method since less nodes corresponding to maximal frequent itemsets are merged. In Figure 6-(e), the average processing time per transaction is compared. It is inversely proportional to the memory usage. This is mainly because the processing time to interpret the information of the itemsets in a node of a CP-tree becomes longer as either the value of δ is larger or the value of S_{merge} is smaller.

Knowledge embedded in a data stream is more likely to be changed over time [5]. Figure 7 shows how the *estDec*+ method can adaptively maximize the utilization of confined memory space. In this experiment, the data set *WebLog* is used. The values of S_{min}, S_{sig}, and S_{merge} are set to 0.003, $0.1\times S_{min}$, and 0.003 respectively. Furthermore, the values of M_U, M_L, and M_A are set to 95MB, 85MB, and 100MB respectively. The initial value of δ is set to 0 and two different values of a user-defined increment α are used. The *estDec* method fails to be executed after the 1×10^{5th} transaction. This is because the size of its prefix tree becomes larger than that of the confined memory space. On the other hand, there is no problem to execute the *estDec*+ method by adjusting the value of δ adaptively for the same situation. Figure 7-(a) illustrates the trace of the value of δ in this experiment. As shown in Figure 7-(b), the memory usage of the *estDec*+ method is kept between M_U and M_L at all times. As expected, the value of δ is more widely fluctuated for the larger value of α. Figure 7-(c) shows the *ASE*s of the the *estDec*+ method in finding frequent and maximal frequent itemsets. The *ASE* of finding frequent itemsets is much higher than that of finding maximal frequent itemsets.

6. Concluding Remarks

For a given value of S_{min}, the total number of frequent itemsets can be varied continuously over time without any upper bound. On the other hand, the up-to-date mining result of an on-line data stream should be traced in real-time and available at any moment. For this reason, the current counts of all the significant itemsets are kept in main memory by the *estDec* method. However, it is impossible to guarantee all of them to be maintained in confined memory space at all times. To cope with this problem, a CP-tree is proposed in this paper. Although the proposed *estDec*+ method can be used to find either frequent itemsets or maximal frequent itemsets, it provides better accuracy for finding maximal frequent itemsets as illustrated in the experiments. By making the value of a merging threshold S_{merge} large enough, the *estDec*+ method can find the set of maximal frequent itemsets as accurately as the *estDec* method can do while the memory usage can be minimized. Although the average processing time of the *estDec*+ method is slightly increased, the proposed method successfully provides a way to accommodate the unpredictable number of significant itemsets generated in the future of a data stream not in a secondary storage but in confined memory space.

References

[1] R.C. Agarwal, C.C. Aggarwal, and V.V.V. Prasad. Depth First Generation of Long Patterns. In *Proc. of the 6th ACM SIGKDD*, pp. 108-118, 2000.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th VLDB*, pp. 487-499, 1994.

[3] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proc. of the ACM SIGMOD*, pp. 255-264, 1997.

[4] J.H. Chang and W.S. Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proc. of the 9th ACM SIGKDD*, pp. 487-492, 2003.

[5] G. Dong, J. Han, L.V.S. Lakshmanan, J. Pei, H. Wang, and P.S. Yu. Online Mining of Changes from Data Streams: Research Problems and Preliminary Results. In *Proc. of the Workshop on Management and Processing of Data Streams*, 2003.

[6] M. Garofalakis, J. Gehrke, and R. Rastogi. Querying and Mining Data Streams: You Only Get One Look. In *tutorial notes of the 28th VLDB*, 2002.

[7] S. Guha and N. Koudas. Approximating a Data Stream for Querying and Estimation: Algorithms and Performance Evaluation. In *Proc. of the 18th ICDE*, pp. 567-576, 2002.

[8] A. Hafez, J. Deogun, and V. V. Raghavan. The Item-Set Tree: A data Structure for Data Mining. In *Proc. of the 1st International Conference on Datawarehousing and Knowledge Discovery*, pages 183-192, 1999.

[9] G. Hulten, L. Spencer, and P. Domingos. Mining Time-Changing Data Streams. In *Proc. of the 7th ACM SIGKDD*, pp. 97-106, 2001.

[10] D. Lambert and J.C. Pinheiro. Mining a Stream of Transactions for Customer Patterns. In *Proc. of the 7th ACM SIGKDD*, pp. 305-310, 2001.

[11] G.S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proc. of the 28th VLDB*, pp. 346-357, 2002.