Information Sciences 278 (2014) 559–576

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

CP-tree: An adaptive synopsis structure for compressing frequent itemsets over online data streams

Se Jung Shin, Dae Su Lee, Won Suk Lee *

Department of Computer Science, Yonsei University, 134 Shinchon-dong, Seodaemun-gu, Seoul 120-749, Republic of Korea

ARTICLE INFO

Article history:
Received 29 April 2008
Received in revised form 1 September 2013
Accepted 8 March 2014
Available online 24 March 2014

Keywords:
Data mining
Frequent itemset
Frequent itemset compression
Data stream
Stream data mining

ABSTRACT

Due to the characteristics of a data stream, it is very important to confine the memory usage of a data mining process. This paper proposes a *CP-tree* (*Compressible-prefix tree*) that can be effectively employed in finding frequent itemsets over an online data stream. Unlike a prefix tree, a node of a CP-tree can maintain a concise synopsis that can be used to trace the supports of several itemsets together. As the number of itemsets that are traced by a node of a CP-tree is increased, the size of a CP-tree becomes smaller. However, the result of a CP-tree becomes less accurate since the estimated supports of those itemsets that are traced together by a node of a CP-tree may contain possible false positive or negative errors. Based on this characteristic, the size of a CP-tree can be controlled by merging or splitting the nodes of a CP-tree, which allows the utilization of a confined memory space as much as possible. Therefore, the accuracy of a CP-tree is maximized at all times for a confined memory space. Furthermore, a CP-tree can trace a concise set of representative frequent itemsets that can collectively represent the set of original frequent itemsets.

1. Introduction

In a data set of transactions, an itemset is frequent if the ratio of the count of transactions that contain the itemset over the total number of transactions is greater than or equal to a predefined support threshold called a *minimum support* S_{min}. A typical data mining task on finding frequent itemsets [1,2,4,12,17] for a finite data set often produces a huge number of similar frequent itemsets especially when either the items of a data set are heavily correlated or the value of a minimum support is quite low. In order to represent the entire set of frequent itemsets by a compact notation, closed or maximal frequent itemsets are introduced. An itemset is a *closed itemset* if no proper superset of the itemset has the same support as the support of the itemset. Therefore, a *closed frequent itemset* (*CFI*) can represent a number of frequent itemsets whose supports are the same. There are several algorithms [24,26,30] for finding the set of CFIs in a finite data set. Since the exact support of every frequent itemset can be recovered by CFIs, representing the set of frequent itemsets by its corresponding set of CFIs is lossless compression. However, the ratio of compression is quite restricted since the supports of most itemsets are different [25]. On the other hand, a frequent itemset is called a *maximal frequent itemset* (*MFI*) if it is not a subset of any other frequent itemset [3]. Several algorithms [5,14,20,28] have been proposed for finding MFIs. Given a data set, the set of MFIs is much smaller than the set of frequent itemsets, so that the compression ratio of MFIs may be quiet good but the supports of its subsets may be quite different one another. Consequently, unlike a CFI, an MFI cannot be used to represent a number of

* Corresponding author.
E-mail addresses: starofu@database.yonsei.ac.kr (S.J. Shin), dslee@database.yonsei.ac.kr (D.S. Lee), leewo@database.yonsei.ac.kr (W.S. Lee).

http://dx.doi.org/10.1016/j.ins.2014.03.074
0020-0255/

frequent itemsets whose supports are similar. Accordingly, a method of extracting a compressed set of frequent itemsets is proposed for a finite data [25]. It provides a flexible way to represent a number of similar frequent itemsets by a single representative frequent itemset.

The term *data stream* is used to denote the common characteristics of such a dataset. It is a massive, unbounded sequence of data elements continuously generated at a rapid rate. As such, it is impossible to maintain all the data elements of a data stream. Consequently, on-line data stream processing should satisfy the following requirements [13]. First, each data element is examined mostly 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, a newly generated data element should be processed in less than a fixed duration 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 accuracy by allowing some errors. Data mining over an online data stream should support a flexible trade-off between processing time and data mining accuracy. In addition, the granularity of input data should not be predetermined in order to catch the sensitive change of its result as quickly as possible. In order to evolving granular computing, [33] deals with several evolving fuzzy systems approaches which have emerged during the last decade and highlights the most important incremental learning methods used. These approaches are able to automatically adapt parameters, expand their structure and extend their memory on-the-fly, allowing on-line/real-time modeling. The problem of learning from data issued of time/spatial-based complex non-stationary processes is described in [32]. It presents efficient techniques, methods and tools able to manage, to exploit and to interpret correctly the increasing amount of data in environments that are continuously changing.

Network monitoring systems, sensor data systems, and fraud detection systems are usually designed to support the automatic analysis of continuously generated data elements and expect their analyzed results to be produced quickly. In a ubiquitous computing environment, identifying the current context of a user quickly is very important to provide the user with customized services proactively. For this purpose, the frequent contexts of a user can be recognized by tracing frequent itemsets over sensor data streams. Recently, various algorithms [7,8,11,15,23,27,29,31] are actively proposed to extract different types of patterns embedded in a data stream. Among these algorithms, *Sticky sampling* [21], *FTP-DS* [23], *Lossy Counting* [21], *estDec* [6], and *FDPM*-1[9] focus on finding frequent itemsets over a data stream. *Moment* [8] and *CFI-stream* [19] find the set of CFIs in those transactions that are within the current range of a sliding window over a data stream. In [31], the fundamentals of data stream mining and describe important applications, such as TCP/IP traffic, GPS data, sensor networks, and customer click streams.

In [6], we have proposed the *estDec* method for finding recently frequent itemsets over an online data stream. Among all itemsets in the transactions of a data stream, only those itemsets that can possibly be frequent itemsets in the near future are regarded as *significant itemsets* which are maintained in main memory by a lexicographic tree structure called a *prefix tree* [1,4]. An itemset is significant if its current support is greater than or equal to a user-defined threshold S_{sig} ($\leqslant S_{min}$). Patterns embedded in a data stream are more likely to be changed over time, so that the number of currently significant itemsets monitored by a prefix tree can be continuously varied. As the number of significant itemsets is increased, the size of a prefix tree that represents these itemsets becomes larger. Since a prefix tree is located in main memory, its size should not be larger than the size of a confined memory space at all times. However, the size of a prefix tree totally depends on the number of those itemsets that are currently significant. Therefore, once the size of a prefix tree becomes larger than that of the confined memory space, it is impossible to monitor any new significant itemset additionally. Because of this, the accuracy of the *estDec* method can be degraded without any upper bound in this situation.

In this paper, two itemsets are defined to be *similar* if one of them is a proper subset of the other and their support difference is less than or equal to a predefined threshold called a *merging gap threshold* $\delta \in (0,1)$. This similarity measure employed in this paper is a straight-forward generalization of a closed itemset. In other words, when $\delta = 0$, the longest itemset among two or more similar itemsets is a closed itemset. This paper proposes a *CP-tree* (*Compressible-Prefix tree*). Unlike a prefix tree where a significant itemset is represented by only a single node, 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 δ. Consequently, the supports of several similar itemsets can be monitored together by a single node of a CP-tree. Among the similar itemsets, only the counts of the two representative itemsets are maintained while those of non-representative itemsets are estimated based on the counts of the two representative itemsets. By adaptively controlling the value of δ, the number of nodes in a CP-tree can be changed flexibly. As the value 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 less accurate information is maintained in the CP-tree.

A CP-tree can be employed for two distinct purposes. One is to fully utilize a confined memory space. By adaptively adjusting the value of δ for the confined memory space, the precision of a CP-tree can be maximized at all times. The other is to compress the set of frequent itemsets instantly over an online data stream. All the similar itemsets of a node can be represented by the longer one of the two representative itemsets. The compression ratio of a CP-tree can also be flexibly controlled by changing the value of δ. Furthermore, by imposing another support threshold called a *merging threshold* S_{merge} ($\geqslant S_{min}$), it is possible to separate a CP-tree into two disjoint parts. One is the upper part where a node can monitor the supports of multiple significant itemsets together. The other is the lower part where a node can only monitor the support of a single significant itemset. Consequently, a CP-tree can trace those significant itemsets that are represented by the nodes of its lower part as accurately as a prefix tree does.

The rest of this paper is organized as follows: Related work is presented in Section 2. Section 3 reviews the *estDec* method. Section 4 proposes the structure of a CP-tree in detail. Section 5 introduces the extended version of the *estDec* method,

namely the *estDec+* method which employs a CP-tree to find frequent itemsets over an online data stream. In Section 6, the performance of the *estDec+* method is comparatively evaluated using a series of experiments. Finally, conclusions are drawn in Section 7.

2. Related work

Frequency counting algorithms over a data stream [7,10,21] allow some errors in their results in terms of either the set of frequent itemsets or the support of an individual frequent itemset. The *sticky sampling* [21] method uses a statistical sampling technique to estimate the count of itemsets while the *FTP-DS* [23] algorithm takes a regression-based approach to find frequent temporal patterns over data streams. Therefore, these two algorithms do not guarantee the upper bound of an error count in the estimated support of each itemset. In order to impose the upper bound of a false positive error, the *Lossy Counting* [21] algorithm finds the set of frequent itemsets over a data stream with respect to an *error parameter* ε. In the *Lossy Counting* algorithm, to reduce the memory usage of a mining process, the counts of frequent itemsets are kept in a secondary storage and only a buffer for temporarily holding transactions is kept in main memory. When the size of a buffer is enlarged, more transactions can be batch-processed together. As a result, the algorithm is more efficiently executed. Similarly, the *estDec* method also employs a *significant support* S_{sig} for this purpose. Consequently, the maximum possible false positive error of a frequent itemset found by these two algorithms is always less than ε or S_{sig} respectively. For both algorithms, the maximum possible error in the count of a frequent itemset is bounded by a user-defined parameter. While all of the above algorithms guarantee no false negative error in their results, the *FDPM*-1 algorithm [22] explores the possibility of allowing false negative errors in finding the set of frequent items over a data stream. Therefore, it may miss some of frequent items in its result but the amount of information needed to be held is greatly reduced.

Moment [8] finds the set of CFIs within the period of a sliding window over a data stream. It uses an in-memory data structure, called a *closed enumeration tree* (*CET*), which monitors not only the set of CFIs but also those itemsets that are in the boundary between the CFIs and the remaining itemsets. There are two different categories of an itemset. One is whether an itemset is either frequent or not. The other is whether an itemset is either closed or not. When a CFI and its subset appear in the same set of transactions, they have the same support. Consequently, all of currently CFIs are stored in a hash table whose key is constructed by (*support*,*tid_sum*) in order to identify the above two conditions quickly. The term *support* denotes the support of an itemset and the term *tid_sum* denotes the sum of the identifiers of those transactions that contain the itemset. The *Moment* algorithm defines specific actions to be taken when the category of a particular itemset is changed. Whenever a new transaction T is generated or a transaction T becomes out of the range of the sliding window, the *CET* is traversed according to the items of T. At the same time, the current *support* and the *tid_sum* of the itemset represented by each visited node are updated. In addition, the entry of the itemset in the hash table is also updated and checked to find out whether the current category of the itemset should be changed or not.

The *CFI-stream* algorithm [19] employs a data structure called a *DIrect Update* (*DIU*) tree that stores all closed itemsets in lexicographical order for a data stream. It performs two operations: an *addition* operation for a newly generated transaction and a *deletion* operation for the transaction that is deleted from the range of the sliding window. As in the *Moment* algorithm, the algorithm proposes a number of actions to be taken to the *DIU* tree for more precisely categorized cases. Since *CFI-Stream* stores all closed itemsets in the *DIU* tree regardless of their supports, its processing time and memory usage remain almost the same even if the value of a minimum support is changed.

In [25], two greedy methods: namely *RPglobal* and *RPlocal* are proposed to compress the set of frequent itemsets in a finite data set. The similarity between two itemsets is represented by a distance measure which denotes the ratio of the number of transactions that contain the two itemsets together over the number of transactions that contain at least one of the two itemsets. An itemset p is δ-*covered* by another itemset p' if the items of p is a subset of p' and the similarity between them is less than or equal to δ ($0 \leqslant \delta \leqslant 1$). A set of similar itemsets $\mathbf{P}$ is called as a δ-*cluster*. A single representative itemset $p_r \in \mathbf{P}$ replaces all the itemsets of $\mathbf{P}$ but the support information of every non-representative frequent itemset is lost. The *RPglobal* and *RPlocal* methods find a set of representative frequent itemsets that can replace all the frequent itemsets of a data set. The *RPglobal* method preserves the quality of compression but it requires high computational complexity. The *RPlocal* method sacrifices the theoretical quality bound of compression but it is far more efficient. The *RPglobal* method employs a greedy *set-covering* approach by employing an FP-tree-like structure [16]. However, when the number of frequent itemsets is increased, this method does not scale well. To cope with this problem, the *RPlocal* method finds locally good representative itemsets. However, the *RPlocal* method provides less accurate results since it cannot utilize the complete coverage information. The number of representative frequent itemsets can be flexibly controlled by the value of δ. These two methods cannot be applied to a data stream since they need to scan a data set multiple times.

3. Preliminaries

For finding frequent itemsets, a data stream can be viewed as an infinite set of continuously generated transactions as follows:

(i) Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of items that have ever been used as a unit information of an application domain.

(ii) An itemset e is a set of items such that $e \in (2^I - \{\varnothing\})$ where 2^I is the power set of I. For simplicity, an itemset $\{a,b,c\}$ is denoted by abc. In addition, the number of items in an itemset e is denoted by $|e|$ and an itemset with $|e|$ items is denoted by an $|e|$-itemset.
(iii) A transaction is a non-empty subset of I and each transaction has a unique transaction identifier TID. A transaction generated at the kth turn is denoted by T_k and its transaction identifier TID is k.
(iv) When a new transaction T_k is generated, the current data stream D_k is composed of all the transactions that have ever been generated so far, i.e., $D_k = \langle T_1, T_2, \ldots, T_k \rangle$ and the total number of transactions in D_k is denoted by $|D_k|$.

For finding frequent itemsets over an online data stream, we have proposed the *estDec* method [6] to minimize the number of itemsets to be monitored. The method 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 [1,4] as shown in Fig. 1. Given the current data stream D_k, a prefix tree P_k has the following characteristics:

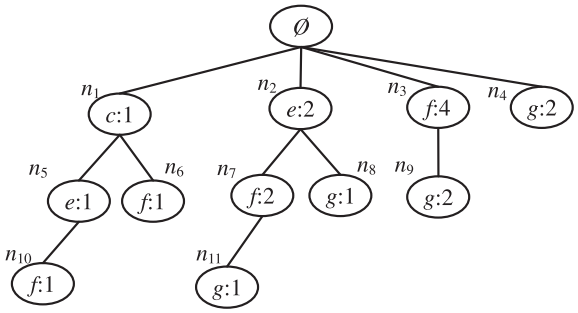
(i) A prefix tree P_k has a root node n_{root} with a "*null*" value and each node except the root node has an item $i \in I$.
(ii) Each node of a prefix tree consists of two fields: *item-name* and *count*. The *item-name* field holds the item of the node and the *count* field maintains the count of the itemset represented by the node.
(iii) For an itemset $e = i_1, i_2, \ldots, i_k$, the items $i_1, i_2, \ldots$, and i_k are lexicographically ordered and the last node of a path $(root) \rightarrow i_1 \rightarrow \ldots \rightarrow i_k$ $(i_k \in I, k \geqslant 1)$ in a prefix tree represents the itemset e.

Trends embedded in a data stream are more likely to be changed as time goes by. Identifying the recently changed significant pattern in a data stream quickly can provide valuable information for the analysis of the data stream. For this purpose, the effect of obsolete information in old transactions on the current result of a data stream should be eliminated effectively. To identify the recent change of a data stream, a decay mechanism [18] can be applied. In other words, the weight of information in each transaction is differentiated according to the time that the transaction is generated. A decay rate is the reducing rate of a weight for a fixed decay unit that determines the chunk of information to be decayed together. It is defined by two parameters: a *decay-base b* and a *decay-base-life h*. A *decay base b* determines the amount of weight reduction per decay unit and it is greater than or equal to 1. As the value of b becomes larger, the amount of weight reduction is increased. A *decay-base-life h* is defined by the number of decay units that make the current weight be $1/b$ when the weight of the current information is set to 1. Based on these two parameters, a *decay rate d* is defined as follows:

$$d = b^{-(1/h)} \quad (b \geqslant 1, h \geqslant 1)$$

Given a decay rate d, the approximate count $C_k(e)$ of an itemset e in the current data stream D_k is defined as follows:

$$\text{if } e \text{ appears in a new transaction } T_k \quad C_k(e) = C_{k-1}(e) \times d + 1 \tag{1}$$



(a) A Prefix tree P_k

node	itemset	node	itemset	node	itemset
root	Ø	n_4	*g*	n_8	*eg*
n_1	*c*	n_5	*ce*	n_9	*fg*
n_2	*e*	n_6	*cf*	n_{10}	*cef*
n_3	*f*	n_7	*ef*	n_{11}	*efg*

(b) itemsets in P_k

Fig. 1. A prefix tree P_k.

if e does not appear in a new transaction $T_k \quad C_k(e) = C_{k-1}(e) \times d$ (2)

The current approximate support $S_k(e)$ of an itemset e is the ratio of its current approximated count $C_k(e)$ over $|D_k|$. For a predefined minimum support S_{min}, an itemset e whose current support $S_k(e)$ is greater than or equal to S_{min} is a frequent itemset in the current data stream D_k.

Whenever a new transaction T_k is generated, all nodes that are corresponding to the itemsets of T_k are visited and their current counts are updated respectively. The *estDec* method examines each transaction in a data stream one-by-one without any candidate generation. It employs two major operations: *delayed-insertion* and *pruning* operations. For each of the two operations, it employs two different support thresholds: an insertion support S_{sig} ($\leqslant S_{min}$) and a pruning support S_{prn} ($\leqslant S_{sig}$). An itemset whose support is greater than or equal to S_{sig} is regarded as a *significant itemset*. 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 into the monitoring tree P_k without any estimation. The second case is when an itemset that used to be insignificant becomes significant due to its appearance of T_k. Since it becomes a significant itemset, it should be inserted into P_k for further monitoring. To find such an n-itemset e ($n \geqslant 2$), only when all of its $(n-1)$-subsets are maintained in P_k, the current support of the n-itemset e is estimated by those of its $(n-1)$-subsets.

Given a data set of transactions, two itemsets e_1 and e_2 are ***least exclusively distributed*** (***LED***) when they appear together in as many transactions as possible. Inversely, they are ***most exclusively distributed*** (***MED***) when they appear exclusively as much as possible. The maximum count $C^{max}(e_1 \cup e_2)$ of their union-itemset $e_1 \cup e_2$ is estimated by assuming that the two itemsets e_1 and e_2 are *LED*

$$C^{max}(e_1 \cup e_2) = \min(C(e_1), C(e_2))$$

where $C(x)$ denotes the count of an itemset x. When the two itemsets e_1 and e_2 ($e_1 \cap e_2 \neq$) are *MED*, the minimum count $C^{min}(e_1 \cup e_2)$ of their union-itemset is estimated as follows:

$$C^{min}(e_1 \cup e_2) = \max(0, C(e_1) + C(e_2) - C(e_1 \cap e_2))$$

Since the total number of the $(n-1)$-subsets of the n-itemset e is n, let $U_{n-1}(e)$ denote the set of the $(n-1)$-subsets and $\{c_1, c_2, \ldots, c_n\}$ denote the set of the current counts of the $(n-1)$-subsets. An n-itemset can be viewed as the union-itemset of its subsets. Therefore, the maximum count $C^{max}(e)$ of the n-itemset e is estimated by its $(n-1)$-subsets as follows:

$$C^{max}(e) = \min(c_1, \ldots, c_n)$$

For each distinct pair (α_i, α_j) of the $(n-1)$-subsets i.e., α_i and $\alpha_j \in U_{n-1}(e)$ $(\alpha_i \neq \alpha_j)$, when they are *MED*, the pair-wise minimum count $C^{min}(\alpha_i \cup \alpha_j)$ can be estimated. Among the ${}_nC_2$ number of pair-wise minimum counts, the largest count is the guaranteed appearance count i.e., the minimum count $C^{min}(e)$ of the n-itemset e. The upper bound of this estimation error $C^{max}(e) - C^{min}(e)$ is proven to be ignorable when k becomes infinite. The detailed description of this estimation is presented in [6]. Based on this estimation, the estimated current count $\widehat{C}_k(e)$ of the n-itemset e is set to the maximum possible count $C^{max}(e)$ i.e. $\widehat{C}_k(e) = C^{ma}x(e)$. If $\widehat{C}_k(e)/|D|_k \geqslant S_{sig}$, the itemset e is inserted into P_k. The above procedure is a ***delayed-insertion*** operation. A *pruning* operation is performed when the current support of an n-itemset ($n \geqslant 2$) maintained by P_k becomes less than S_{sig}. An itemset that used to be a significant itemset is regarded as an insignificant itemset when it cannot be a frequent itemset in the near future. Upon identifying such an itemset, the node representing the itemset and all of its descendent nodes are pruned together from P_k based on the anti-monotone property of a frequent itemset.

4. Compressible prefix tree: CP-tree

4.1. Structure of a CP-tree

To reduce the size of a prefix tree, the information represented in a prefix tree needs to be compressed as a concise synopsis. Two consecutive nodes in a prefix tree is merged into a node of a CP-tree when their corresponding itemsets are similar to each other. Ultimately, a subtree of a prefix tree can be merged into a node of a CP-tree. All the itemsets of the subtree are similar. Such a subtree is defined as a *mergeable subtree* by Definition 1 and the detailed structure of a node in a CP-tree is defined in Definition 2.

Definition 1 (*A mergeable subtree*). Let P_k be a prefix tree maintained by the *estDec* method in the current data stream D_k. Given a merging gap threshold δ, a *mergeable subtree* S of P_k is a maximal subtree rooted at an internal node e_r of P_k and every itemset e_j of S should satisfy the following constraint.

$$\forall\ e_j \in S, \quad |C_k(e_r) - C_k(e_j)|/|D_k| \leqslant \delta, \quad 1 \leqslant j \leqslant |S|$$

where $|S|$ denotes the number of nodes in S. A leaf node of S does not have to be a leaf node of P_k.

Definition 2 (*Structure of a CP-node*). Given a mergeable subtree S of a prefix tree P_k for the current data stream 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 $m(\tau, \pi, c_{\mathrm{I}}, c_{\mathrm{II}})$:

(i) *item-list* τ: In an item-list τ, the items of 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 jth item in τ is represented by $\tau[j]$ $(1 \leqslant j \leqslant |\tau|, |\tau| = |S|)$. 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 $m.e_{\mathrm{I}}$. On the other hand, the last item $m.\tau[|\tau|]$ 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 $m.e_{\mathrm{II}}$. All the similar itemsets of the subtree S are compressed into this longest itemset. Among the items of τ, those items that are represented by the leaf nodes of S are called as *leaf-level items*.

(ii) *parent-index list* π: A parent-index list π maintains an entry of a form $p.q$ where p denotes a node identifier of Q_k and q denotes 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 be a node n_z with an item z in P_k and the node n_z be in another mergeable subtree $\bar{S}$. Let $\bar{S}$ be represented by a node $\bar{m}$ of Q_k and the item-list index of the node n_z in $\bar{m}$ be q_z, i.e., $\bar{m}.\tau[q_z] = z$. The entry of $m.\pi[1]$ is set to $\bar{m}.q_z$.

(iii) *largest counter* c_{I}: It maintains the current count of the shortest itemset e_{I}.

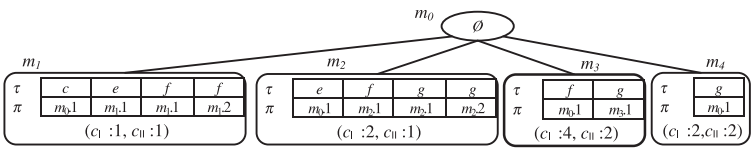
(iv) *smallest counter* c_{II}: If $|S| = 1$, $c_{\mathrm{II}} = c_{\mathrm{I}}$. Otherwise, it maintains the current count of the longest itemset e_{II}.

If every node of a CP-tree is allowed to be merged, an infrequent but significant itemset e, i.e., $S_{sig} \leqslant S_k(e) < S_{min}$ can be merged with some frequent itemsets. This may make either the itemset e be frequent or some of frequent itemsets be infrequent by the merged-count estimation. In order to avoid these types of an error, if the current support of the longest itemset of a node is less than a predefined threshold called a *merging threshold* S_{merge} $(\geqslant S_{min})$, the node is not considered as a candidate for a node-merge operation. As the gap between S_{merge} and S_{min} is enlarged, the size of a CP-tree is increased but the overall merged-count estimation errors in all frequent itemsets are decreased.

Fig. 2 shows a CP-tree Q_k which is equivalent to the prefix tree P_k in Fig. 1. The subtree formed by the nodes n_1, n_5, n_6, and n_{10} of the prefix tree P_k are compressed into the node $m_1(\tau = \langle c,e,f,f\rangle, \pi = \langle m_0.1, m_1.1, m_1.1, m_1.1, m_1.2\rangle, c_{\mathrm{I}}, c_{\mathrm{II}})$ 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_5, n_6 and n_{10} is less than δ. The item-list $\tau = \langle c,e,f,f\rangle$ and parent index list $\pi = \langle m_0.1, m_1.1, m_1.1, m_1.2\rangle$ in the node m_1 maintain the level-wise structural information of its corresponding mergeable subtree in P_k. More precisely, the root of the subtree represented by m_1 is $\tau[1] = c$ 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 Q_k. The fact that the node n_1 is the parent of the node n_5 in P_k can be inferred by $m_1.\tau[2] = e$ and $m_1.\pi[2] = m_1.1$. These two facts imply that the parent of the item e is $m_1.\tau[1]$, i.e., c. Similarly, the shortest and longest itemsets represented by the node m_1 are c and cef respectively. Their current counts are maintained by the two counters c_{I} and c_{II} of the node m_1. Likewise, the nodes n_2, n_7, n_8 and n_{11} of P_k are compressed into the node m_2 of Q_k. On the other hand, since there is no mergeable subtree for n_4, it is left to be uncompressed in Q_k. Fig. 2(b) shows how the itemsets of P_k are compressed by Q_k. The compression ratio is about 33.3% in terms of the number of itemsets. The detailed steps of a traverse operation are described in Fig. 3.

4.2. Merged-count estimation

Given the item-list $m.\tau = \langle i_1, i_2, \ldots, i_v\rangle$ of a node m in a CP-tree, let e_{i_j} denote the itemset represented by i_j $(1 \leqslant j \leqslant v)$. By the two counters c_{I} and c_{II} of the node, it is possible to trace the current supports of at most two similar itemsets as precisely as the *estDec* method does. They are the shortest and longest itemsets e_{i_1} and e_{i_v}. Therefore, if more than three similar itemsets are compressed into a single node, the current supports of the remaining itemsets e_{i_j} $(2 \leqslant j \leqslant v-1)$ should be estimated. The current count $C_k(e_{i_j})$ of such an itemset e_{i_j} can be estimated by a formula $C(e_{i_j}) = \lceil m.c_{\mathrm{I}} - f(m,j)\rceil$ where $f(m,j)$ denotes a count



(a) An equivalent CP-tree Q_k

	representative itemsets(e_{II})	itemsets in P_k
m_1	*cef*	*c, ce, cf, cef*
m_2	*efg*	*e, ef, eg, efg*
m_3	*fg*	*f, g, fg*
m_4	*g*	*g*

(b) Compressed itemsets in Q_k

Fig. 2. A CP-tree and its equivalent a prefix tree ($\delta = 0.5, S_{sig} = 0, S_{merge} = 0, d = 1$).

```
traverse(m, m_p, q, T)
m_p: the parent node of a node m
q: the item-list index of the last leaf-level item matched in the node m_p i.e.,
m_p.τ[q]
1  if m.π[1] = m_p.q and m.τ[1] ∈ T;
2      m.c_I ← m.c_I *d+ 1;
3      if (m.c_I / |D_k|) < S_sig
4          pruning m;          /* eliminate m and all of its descendent nodes */
5      else
6          i ← 1;
7          find the set of the i th level common items C^i;
8          while C^i ≠ ∅
9              i ← i + 1;
10             find the set of the i th level common items C^i;
11         if m.τ[|τ|] ∈ C^(i-1)
12             m.c_II ← m.c_II *d + 1;
13         if (m_p.c_I -m.c_II ) / |D_k| ≤ δ and m.c_II / |D_k| ≥ S_merge
14             node_merge(m_p, m);
15         else if (m.c_I -m.c_II ) / |D_k| > δ and m.c_II / |D_k| ≥ S_merge
16             node_split(m);
17         if ∃ a child node of m and ∃ a leaf-level item ∈ C^1 ∪ C^2 ∪ … ∪ C^(i-1)
18             for all leaf-level items m.τ[j]
19                 for all child nodes m_c of m
20                     traverse(m_c, m, j, T);
```

Fig. 3. Traverse operation.

estimation function that can model the count $C_k(e_{i_j})$ in terms of the two counters c_{I} and c_{II}. Any function meaningful in an application domain can be employed to define the function $f(m,j)$. For example, each of the following two functions $f_1(m,j)$ and $f_2(m,j)$ can be a possible candidate.

$$f_1(m,j) = (m.c_{\mathrm{I}} - m.c_{\mathrm{II}}) \times \frac{|e_{i_j}| - |e_{\mathrm{I}}|}{|e_{\mathrm{II}}| - |e_{\mathrm{I}}|}$$

$$f_2(m,j) = (m.c_{\mathrm{I}} - m.c_{\mathrm{II}}) \times \sum_{l=1}^{|e_{i_j}|-|e_{\mathrm{I}}|} \frac{1}{l} \Bigg/ \sum_{l=1}^{|e_{i_j}|-|e_{\mathrm{II}}|} \frac{1}{l}$$

The function $f_1(m,j)$ assumes that the count of an itemset e_{i_j} $(2 \leqslant j \leqslant v-1)$ is linearly decreased as the number of items in the itemset e_{i_j} is increased. On the other hand, the function $f_2(m,j)$ enlarges the decrement rate of the count of the itemset e_{i_j} as the number of items in the itemset is increased.

Example. Suppose a node m_1 of a CP-tree has its entries ($\tau = \langle b,c,d,e\rangle$, $\pi = \langle m_0.1, m_1.1, m_1.2, m_1.3\rangle$, $c_{\mathrm{I}} = 100$, $c_{\mathrm{II}} = 85$) and its parent node m_0 of m_1 have ($\tau = \langle a\rangle$, $\pi = \langle root\rangle$, $c_{\mathrm{I}} = 120$, $c_{\mathrm{II}} = 120$). Since $m_1.e_{\mathrm{I}} = ab$, $m_1.e_{\mathrm{II}} = abcde$, the count of an itemset abc represented by the second item c in $m_1.\tau$ can be estimated by the count estimation functions $f_1(m,j)$ and $f_2(m,j)$ respectively as follows:

(i) $f_1(m_1,j)$: $\lceil m_1.c_{\mathrm{I}} - f_1(m_1,j)\rceil = \left\lceil 100 - \left\{(100-85) \times \frac{3-2}{5-2}\right\}\right\rceil = 95$

(ii) $f_2(m_1,j)$: $\lceil m_1.c_{\mathrm{I}} - f_2(m_1,j)\rceil = \left\lceil 100 - \left\{(100-85) \times \sum_{l=1}^{3-2} \frac{1}{l} / \sum_{l=1}^{5-2} \frac{1}{l}\right\}\right\rceil = 92$

The above estimation is called as *merged-count estimation* in order to distinguish it from the inserting-count estimation defined in Section 3. The current count of a non-representative itemset traced by a node of a CP-tree is estimated by the above mechanism, so that it may contain a false negative error count but the possible range of this error is totally dependent on the value of a merging gap threshold δ. As this value is set to be larger, more nodes in a CP-tree can be merged. As a result, the size of the CP-tree can be smaller while the errors caused by the merged-count estimation can be increased. However, there exists an upper bound for the errors.

Theorem 1. *Given a merge gap threshold δ, suppose the support of an itemset e in the current data stream D_k is traced by a node m of a CP-tree Q_k. When the itemset e is not either the shortest or longest itemset of the node, its current support is estimated by the merged-count estimation as $S_k(e)$. For the real support $\overline{S}_k(e)$ of the itemset e, the support error $|\overline{S}_k(e) - S_k(e)|$ at the current data stream D_K should be always less than δ.*

Proof. Let $\overline{C}_k e$ and $C_k(e)$ denote the real and approximated counts of the itemset e in D_k. Since the count of the itemset e should be estimated, the itemset e is neither the shortest nor the longest itemset represented by the node m. Therefore, the following should be satisfied.

$$m.c_{\mathrm{I}} \leqslant \overline{C}_k(e) \leqslant m.c_{\mathrm{II}}, m.c_{\mathrm{I}} < C_k(e) < m.c_{\mathrm{II}}$$

Consequently, the error count of the itemset e is bounded by

$$|\overline{C}_k(e) - C_k(e)| \leqslant m.c_{\mathrm{I}} - m.c_{\mathrm{II}} \tag{3}$$

By Definition 1, the following is always true

$$(m.c_{\mathrm{I}} - m.c_{\mathrm{II}})/|D_k| \leqslant \delta \tag{4}$$

From (3) and (4), the current support error of the itemset e is bounded as follows:

$$|\overline{S}_k(e) - S_k(e)| = |\overline{C}_k(e) - C_k(e)|/|D_k| < (m.c_{\mathrm{I}} - m.c_{\mathrm{II}})/|D_k| < \delta. \qquad \square$$

5. Finding frequent Itemsets

5.1. CP-tree traversal

Traversing a CP-tree is virtually the same as traversing a prefix tree described in [6]. When a new transaction T_k is generated in a data stream D_{k-1}, the items of T_k are lexicographically ordered and matched with the CP-tree Q_{k-1} in a depth-first manner. Upon visiting a node m of a CP-tree, let m_p denote its parent node and an item i_p denote the last matched leaf-level item in m_p. Only when the item i_p of the parent node m_p is the first entry of the parent-index list, i.e., $m.\pi[1] = m_p.i_p$, check whether the first item $m.\tau[1]$ is one of the remaining items of T_k. If the above two conditions are not satisfied together, the search in the node m is terminated and the search in the parent node m_p is continued.

If the two conditions are satisfied, the largest counter c_{I} of the node m is updated by (1) since the shortest itemset e_{I} of the node m appears in the new transaction T_k. If the updated support of the shortest itemset $m.e_{\mathrm{I}}$ becomes less than S_{sig}, i.e., $m.c_{\mathrm{I}}/|D_k| < S_{sig}$, 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. Otherwise, among the paths of the mergeable subtree compressed into the node m, those paths that are induced by the remaining items of T_k are traversed in a level-wise manner. In the transaction T_k, only those items that are lexicographically after the item $m.\tau[1]$ are the candidates for further matching. An item $m.\tau[j]$ $(2 \leqslant j \leqslant m.\tau[|\tau|])$ is one of the first level common items if $m.\pi[j] = m.1$ and the item $m.\tau[j]$ is one of the candidates. Likewise, an item $m.\tau[j]$ $(2 \leqslant j \leqslant m.\tau[|\tau|])$ is one of the ith level common items if its parent item is one of the $(i-1)$th level common items and the item $m.\tau[j]$ is one of the candidates. When there is no next level common item, this level-wise search is terminated. Only when the last item $m.\tau[|\tau|]$ is one of the last level common items, the smallest counter $m.c_{\mathrm{II}}$ is updated by (1).

If the updated support of $m.e_{\mathrm{II}}$ is greater than or equal to S_{merge} and the support difference between $m.e_{\mathrm{I}}$ and $m_p.e_{\mathrm{II}}$ becomes less than or equal to δ, i.e., $m.c_{\mathrm{II}}/|D_k| \geqslant S_{merge}$ and $(m_p.c_{\mathrm{I}} - m.c_{\mathrm{II}})/|D_k| \leqslant \delta$, these two nodes m and m_p are merged. A *node-merge* operation is only invoked in the following two cases. One is when the current support difference between the shortest itemset of m_p and the longest itemset of the node m becomes less than or equal to δ, i.e., $m_p.c_{\mathrm{II}} - m.c_{\mathrm{I}} \leqslant \delta$. This case happens only when the difference between the two counters $m_p.c_{\mathrm{II}}$ and $m.c_{\mathrm{I}}$ remains the same and $|D_k|$ is increased. The other one is when a new significant itemset e is identified by the inserting-count estimation, so that a new node for the itemset needs to be inserted as a child of the node m. If the difference between the estimated support of the new significant itemset and the support of the shortest itemset in the node m is less than or equal to δ, the newly created node is instantly merged into the node m. The detailed steps of a node-merging operation are described in Fig. 4. On the other hand, if the updated support of the longest itemset $m.e_{\mathrm{II}}$ is greater than or equal to S_{merge} and the support difference between the itemsets $m.e_{\mathrm{I}}$ and $m.e_{\mathrm{II}}$ becomes greater than δ, i.e., $m.c_{\mathrm{II}}/|D_k| \geqslant S_{merge}$ and $(m.c_{\mathrm{I}} - m.c_{\mathrm{II}})/|D_k| > \delta$, the node m is split. A node m of a CP-tree is split when the current support difference between its shortest and longest itemsets becomes greater than δ, i.e., $(m.c_{\mathrm{I}} - m.c_{\mathrm{II}})/|D_k| > \delta$. The difference is enlarged when only the value of $m.c_{\mathrm{I}}$ is incremented. When the node m is split, every leaf-level item of the node m is separated as an individual node of a CP-tree. The detailed steps of a node-split operation are described in Fig. 5. This procedure is a *node-split* operation.

If the level-wise search in the node m is terminated by reaching at least one of its leaf-level items, all the child nodes of the node m are visited. Otherwise, the search in the node m is terminated and the search in the parent node m_p is continued. This procedure is recursively repeated until there is no item to be matched in T_k. Finally, all the paths induced by T_k are traversed.

Fig. 6 illustrates how a CP-tree is traversed when a new transaction T_k is processed. Traversing Q_{k-1} is started at the root node m_0 by depth-first search. Because of $m_1.\tau[1] \notin T_k$, the node m_2 is visited. Since $m_2.\tau[1] \in T_k$ and $m_2.\pi[1] = m_0.1$, the search in m_2 is continued and the item e becomes the first common item between $m_2.\tau$ and T_k. Since the shortest itemset e represented by m_2 is matched, the largest counter $m_2.c_{\mathrm{I}}$ is updated by (1) when the value of d is 1. To continue traversing the mergeable subtree corresponding to m_2, any second level common item between $m_2.\tau$ and T_k is searched. Since $m_2.\tau[2]$ is the same as the second item of T_k and $m_2.\pi[2] = m_2.1$, the item f becomes the second level common item. Therefore, the

```
node_merge(m_p, m)          /* m_p is the parent node of m */
append m.τ to m_p.τ;
m_p.π [m_p.|τ|+1];← m.π[1];
for each entry in m.π[j] (2 ≤ j ≤ m.|τ|)   /* let m.π[j] = m.q */
    v ← m_p.|τ| + q;
    m_p.π[m_p.|τ| + j] ← m_p.v;
for each child node m_c of m        /* let m_c.π[1] = m.r */
    w ← m_p.|τ| + r;
    m_c.π[1] ← m_p.w;
    make m_c be a child node of m_p;
prune m from Q_k;
```

Fig. 4. Node_merge operation.

```
node_split(m)
for each leaf-level item i                    /* let m.τ[j] = i */
    create a new node m*;
    m*.τ[1] ← m.τ[j];
    m*.π[1] ← m.π[j];
    m.τ[j] ← null;
    if j = |τ|
        m*.c_I ← m.c_II ; m*.c_II ← m*.c_I ;
    else
        m*.c_I ← estimate the current merged count of e_i;
        m*.c_c ← m*.c_I ;
    for each child node m_c of m
        if m_c.π[1] = m.j
            m_c.π[1] ← m*.1;
            make m_c be a child node of m*;
l ← l;
for k = 1 to m.|τ|
    if m.τ[k] ≠ null
        m.τ[l] ← m.τ[k];
        m.π[l] ← m.π[k];
        l ← l + 1;
m.c_II ←estimate the current merged count of the new
longest itemset of m;
```

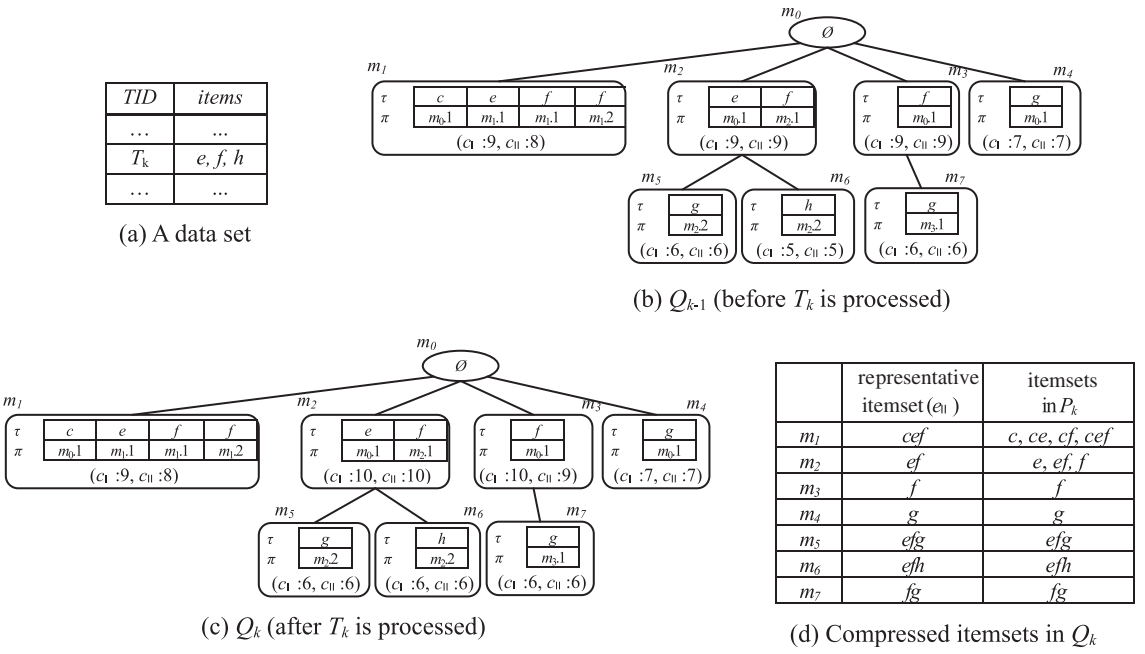
Fig. 5. Node_split operation.

itemset *ef* represented by $m_2.\tau[2]$ is the longest itemset of m_2, so that $m_2.c_{II}$ is also updated by (1). Since a leaf-level item is matched, the child node m_5 is visited subsequently but immediately returned to m_2 since $m_5.\tau[1] \notin T_k$. However, the counter $m_6.c_I$ of the node m_6 is updated by (1) similarly since $m_6.\pi[1] = m_2.2$ and $m_6.\tau[1] \in T_k$. Furthermore, the counter $m_6.c_{II}$ is also updated by (1) as well since the item *h* is also a leaf-level item in the node m_6. Subsequently, the node m_3 is visited and the counters $m_3.c_I$ and $m_3.c_{II}$ are updated similarly by the same reasons. The set of representative itemsets is shown in Fig. 6(d).

5.2. estDec+ method

The *estDec+* method consists of four phases: *parameter updating*, *node restructuring*, *itemset insertion*, and *frequent itemset selection*. Given a CP-tree Q_{k-1} for a data stream D_{k-1}, when a new transaction T_k is generated, these phases except the last phase are performed in sequence as illustrated in Fig. 7. The frequent itemset selection phase is performed only when the up-to-date result set of representative frequent itemsets is requested.

Phase (1) Parameter updating: The total number of transactions in the current data stream D_k is updated by (1).
Phase (2) Node restructuring: This phase is performed by traversing Q_{k-1} according to the lexicographic order of the items in T_k. While traversing, the nodes of Q_{k-1} are restructured.



	representative itemset (c_{II})	itemsets in P_k
m_1	*cef*	*c, ce, cf, cef*
m_2	*ef*	*e, ef, f*
m_3	*f*	*f*
m_4	*g*	*g*
m_5	*efg*	*efg*
m_6	*efh*	*efh*
m_7	*fg*	*fg*

(d) Compressed itemsets in Q_k

Fig. 6. CP-tree traversal ($\delta = 0.5$, $S_{sig} = 0$, $S_{merge} = 0$, $d = 1$).

Phase (3) Itemset insertion: The itemset insertion phase is performed to insert any new significant itemset which has not been maintained in Q_{k-1}. Every single item should be maintained. Consequently, when T_k contains any new item i that is not in Q_{k-1} yet, a new node m for the item $i \in T_k$ is inserted and initialized as follows:

$$m.\tau[1] = i, m.\pi[1] = \langle root \rangle, \quad m.c_{II} = m.c_I = 1$$

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 CP-tree is traversed for the filtered transaction $\overline{T}_k$ once again to find out any new significant itemset induced by the items of $\overline{T}_k$. For each significant n-itemset $e = i_1, i_2, \ldots, i_n$ $(n \geqslant 1)$ represented by a node m of Q_k, examine whether there exists any new significant $(n+1)$-itemset $\bar{e} = e \cup i_{n+1} \in \overline{T}_k$. First of all, check whether all of its n-subsets of the itemset $\bar{e}$ are currently maintained in Q_{k-1}. If this condition is satisfied, the current support of the itemset $\bar{e}$ is estimated by $C(\bar{e})$ as described in Section 3. If $C(\bar{e}) \geqslant S_{sig}$, a new node w corresponding to the itemset $\bar{e}$ is inserted to Q_k as a child of the node m. The entries of the new node w are initialized as follows:

$$w.\tau[1] = i_{n+1}, w.\pi[1] = m.q, \quad w.c_{II} = w.c_I = C(\bar{e})$$

where q denotes the item-list index of the item i_n in the node m. As mentioned in Section 5.1, the newly added node may be instantly merged into its parent node if their support difference is less than or equal to δ.

Phase (4) Frequent itemset selection: This phase retrieves all the currently frequent representative itemsets by traversing the CP-tree Q_k.

A *force-pruning* operation can be performed periodically in order to prune all the nodes whose largest counts c_I are less than $|D_k| \times S_{sig}$ altogether by traversing the entire CP-tree Q_k.

Fig. 8 illustrates how a CP-tree is constructed as a new transaction is processed. Given a data set in Fig. 8(a) and (b) shows the CP-tree Q_1 after the transactions T_1 is processed. If a prefix tree is used to represent T_1, 8 different nodes should be created. The transaction T_2 only makes the two counters c_{II} and c_I of the node m_3 be incremented. New significant itemsets, e.g. and *efg* are identified and inserted in the itemset inserting phase for T_3. The two newly inserted nodes are instantly merged into m_2 as shown in Fig. 8(d). This is because the support difference between the largest count of m_2 and the smallest count of m_5 is less than the merging gap threshold δ, i.e., $(m_2.c_I - m_5.c_{II})/|D_3| = (2-1)/3 \leqslant \delta$. Likewise, m_6 is also merged into m_2. After T_4 is processed, the support difference between the largest count of m_3 and the smallest count of m_7 becomes less than δ for the total number of transactions in D_4, i.e., $(m_3.c_I - m_7.c_{II})/|D_4| = (4-2)/4 \leqslant \delta$. Therefore, as shown in Fig. 2(a), the two nodes m_3 and m_7 shown in Fig. 8(d) are merged. Fig. 8(e) shows how a node-split operation is performed. After T_5 is processed, only the largest count c_I of the node m_3 is increased. As a result, the node m_3 is split into the two nodes m_3 and m_8 since the support difference of m_3 becomes greater than δ, i.e., $(m_3.c_I - m_3.c_{II})/|D_5| = (5-2)/5 > \delta$.

Input: A data stream D_k
Output: A complete set of frequent itemsets L_k

1 $Q_k \leftarrow \varnothing$; $|D_k| \leftarrow 1$;
2 for each *new transaction* T_k in D_k

// ***Phase 1) Parameter updating***
3 $|D_k| \leftarrow |D_{k-1}| + 1$;

// ***Phase 2) Node restructuring***
4 for each child node m of the root of Q_k
5 traverse(m, *root*, 1, T_k);

// ***Phase 3) Itemset insertion***
6 $\overline{T}_k \leftarrow \varnothing$;
7 for each item $i \in T_k$
8 if $i \notin Q_k$
9 insert a node m representing i into Q_k;
10 $m.\tau[1] \leftarrow i$; $m.\pi[1] \leftarrow$ <root> ; $m.c_{\mathrm{I}} \leftarrow 1$; $m.c_{\mathrm{II}} \leftarrow 1$;
11 else if $S_k(i) \geq S_{sig}$
12 $\overline{T}_k \leftarrow \overline{T}_k \cup \{i\}$;

13 for each itemset $e \in Q_k$
14 if $\exists$ an itemset $\overline{e} = e \cup \{i\}$ s.t. $i \in I$ and $i \notin e$ and $\overline{e} \in (2^{\overline{T}_k} - \{\varnothing\})$ and $\overline{e} \notin Q_k$
15 if all the $(|\overline{e}|-1)$-subsets of $\overline{e}$ are in Q_k
16 estimate $C(\overline{e})$;
17 if ($C(\overline{e})/|D_k|) \geq S_{sig}$
18 insert a node m representing $\overline{e}$ into Q_k;
/* let $m_p.\tau[q]$ represent the itemset e */
19 $m.\tau[1] \leftarrow i$; $m.\pi[1] \leftarrow m_p.q$; $m.c_{\mathrm{I}} \leftarrow C(\overline{e})$; $m.c_{\mathrm{II}} \leftarrow C(\overline{e})$;
20 if $\{(m_p.c_{\mathrm{I}} - m.c_{\mathrm{II}})/|D_k|\} \leq \delta$ and $(m.c_{\mathrm{II}} / |D_k|) \geq S_{merge}$
21 node_merge(m_p, m);

// ***Phase 4) Frequent itemset selection***
22 retrieve all of frequent representative itemsets in Q_k;

Fig. 7. *estDec+* method.

5.3. Memory usage adaptation

Although the number of significant itemsets over a data stream is continuously varied, the size of memory space for a CP-tree is physically confined. In order to minimize the estimation errors caused by the merged-count estimation, it is very important to keep the value of a merging gap threshold δ as small as possible. The size of a CP-tree is inversely proportional to the value of δ. If δ is set to be large, more nodes in a CP-tree can be merged, so that the memory usage of the *estDec+* method can be reduced. However, the larger the value of δ becomes, the less accurate the result of the *estDec+* method is. Given a confined memory space, the most accurate result of the *estDec+* method can be found when the confined memory space is utilized as much as possible at all times.

In order to adaptively control the memory utilization of the *estDec+* method, the value of δ should be dynamically adjusted in the parameter update phase of the *estDec+* method. As a naïve approach, after a merging gap threshold δ is updated from its old value δ^{old} to its new value δ^{new}, all the nodes of a CP-tree are traversed to restructure the CP-tree for the new value of δ^{new}. However, when the size of a CP-tree is large, this naïve approach requires long restructuring time, which can cause a severe problem for processing a data stream. Therefore, in order to reduce the restructuring time, only those paths of a CP-tree that are traversed by the new transaction T_k are restructured in Phase 2. As a result, a CP-tree is restructured incrementally by this approach.

Based on the ratio of the current memory usage over a given confined memory space, the value of δ is dynamically changed in the parameter updating phase after the total number of transactions in D_k is incremented. For a given confined memory space M_A, let M_U and M_L denote the upper bound and lower bound of desired memory usage respectively. Whenever the current memory usage M_C of a CP-tree satisfies the following conditions, the new value δ^{new} of a merging gap threshold is adjusted adaptively as follows:

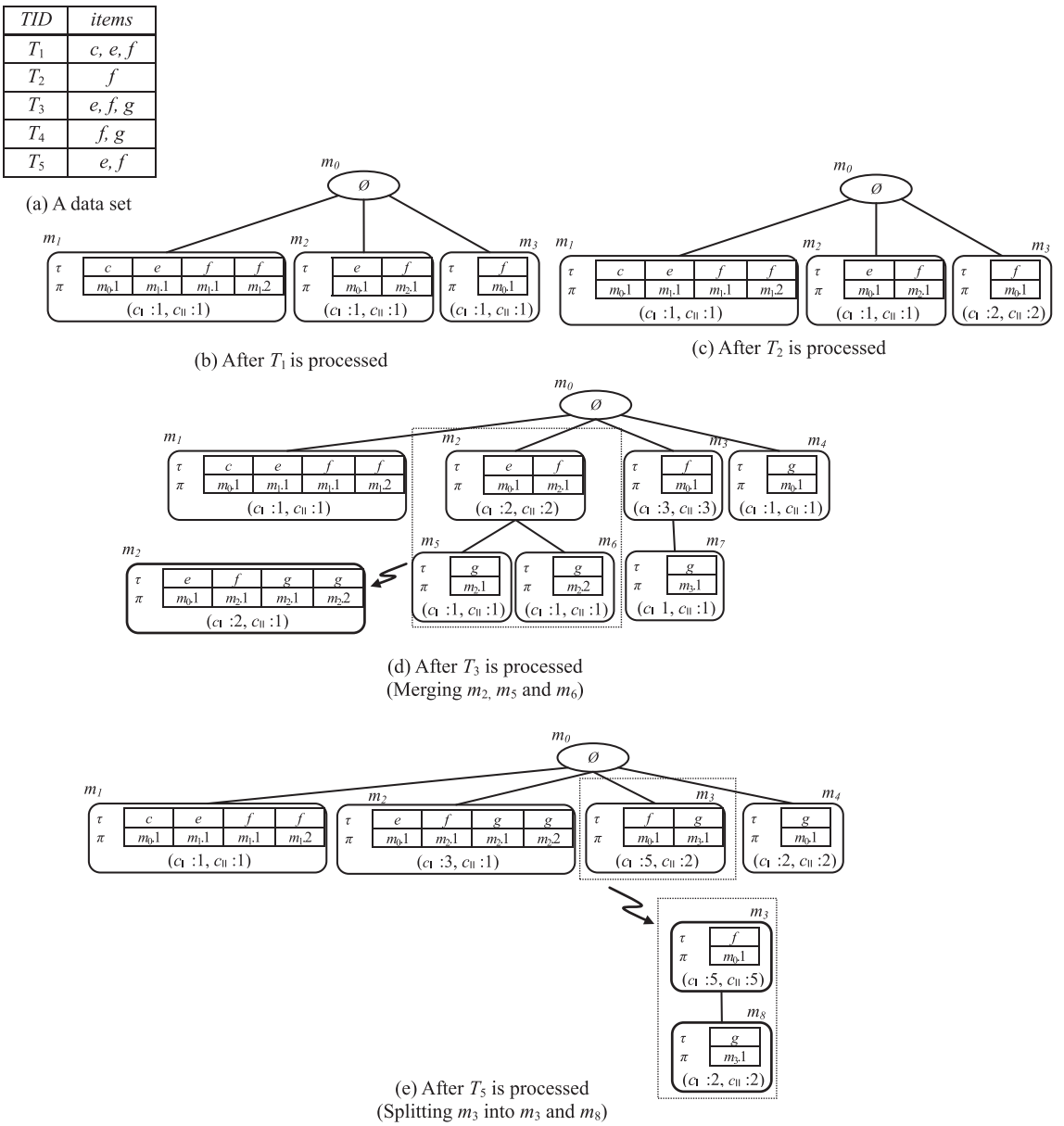


Fig. 8. Construction of a CP-tree ($\delta = 0.5$, $S_{sig} = 0$, $S_{merge} = 0$, $d = 1$).

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

where α denotes the step-wise increment of δ for each adaptation and is defined by a user. The value of α is set to be proportional to that of $M_A - M_U$. As long as the current memory usage M_C 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 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 stably without causing any memory overflow.

6. Performance evaluation

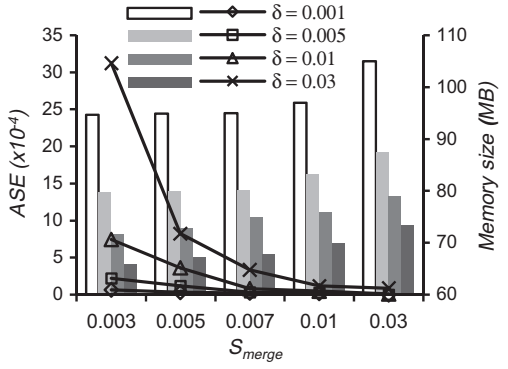
In this section, the performance of the *estDec+* method is analyzed using several data sets shown in Table 1. The data sets *T10.I4.D1000K*, *T5.I4.D1400K*, and *T15.I6.D100K* are generated by the same method as described in [2]. The data set *WebLog* is

a real web-page access log data. The consecutive web-pages accessed by a user are considered as a semantically atomic unit of activities, i.e., a transaction. It can provide valuable information for finding a set of web-pages that are frequently accessed at the same time. However, if a user does not access any web-page for a certain period of time, the corresponding transaction is considered to be terminated. The minimum, maximum, and average lengths of a transaction in the data set *WebLog* are 2, 30, and 5 respectively. The condition of most experiments are $S_{min} = 0.001$, $S_{sig} = 0.1 \times S_{min}$ and $d = 1$ on the data set *T10.I4.D1000K* unless they are specified differently. In addition, the count estimation function $f_2(m,j)$ in Section 4.2 is used. 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. All experiments are performed on a 1.8 GHz Pentium PC machine with 512 MB main memory running on Ubuntu Linux 5.1 and all programs are implemented in C.

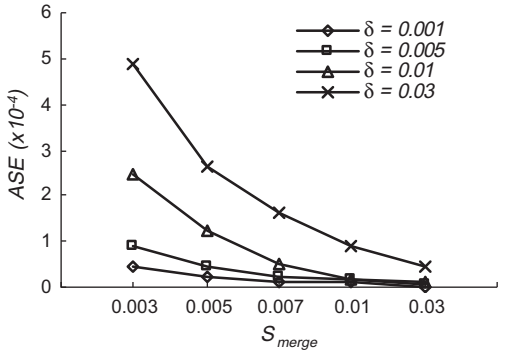
Fig. 9 shows the performance of the *estDec+* method on the data set *T10.I4.D1000K* by varying the values of δ and S_{merge}. The maximum memory usage of the *estDec+* method is illustrated in Fig. 9(a). To measure the relative accuracy of the *estDec+* method, a term *average support error* $ASE(R_2|R_1)$ [6] is employed. When two sets of mining results $R_1 = \{(e_i, S'_k(e_i))|S'_k(e_i) \geqslant S_{min}\}$ and $R_2 = \{(e_j, S''_k(e_j))|S''_k(e_j) \geqslant S_{min}\}$ are given for the same data stream D_k, the *average support error* $ASE(R_2|R_1)$ of R_2 with respect to R_1 is defined as follows:

Table 1
Data sets.

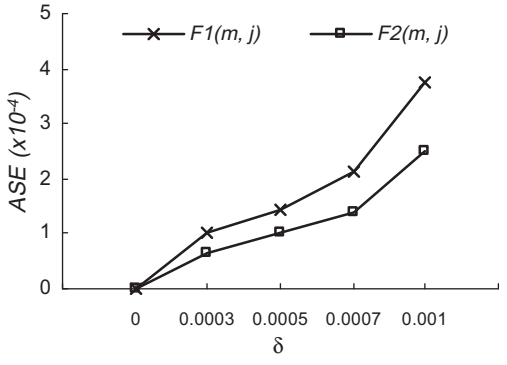
Data sets	# Of items	# Of transactions (K)	Avg. # of items in a transaction
T10.I4.D1000K	1000	1000	10
T5.I4.D1400K	1000	1400	5
T15.I6.D100K	1000	100	15
WebLog	545	500	5



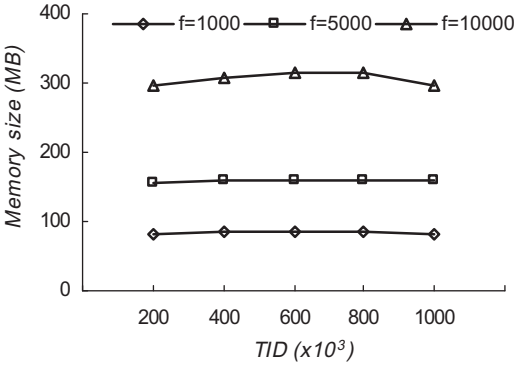
(a) Memory usage and Accuracy of frequent itemsets



(b) Accuracy of maximal frequent itemsets



(c) Effects of estimation functions



(d) Effects of a pruning period

Fig. 9. Performance of the *estDec+* method.

$$ASE(R_2|R_1) = \frac{\sum_{e_l \in R_1 - R_1 \cap R_2} S'_k(e_l) + \sum_{e_l \in R_1 \cap R_2} \{|S''_k(e_l) - S'_k(e_l)|\} + \sum_{e_l \in R_2 - R_1 \cap R_2} S''_k(e_l)}{|R_1|}$$

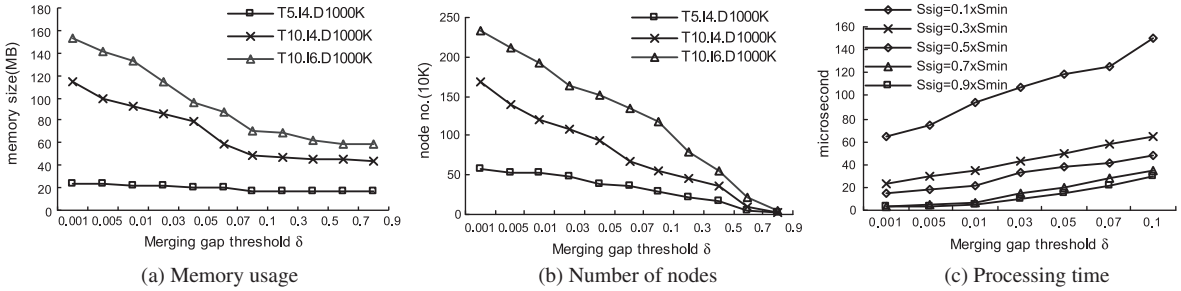
The average support error $ASE(R_2|R_1)$ is used to compare the accuracy of a result R_1 with that of another result R_2. The term $|R_1|$ denotes the number of itemsets in R_1. As the value of $ASE(R_2|R_1)$ gets smaller, the result R_2 is more similar to R_1. Fig. 9(a) shows $ASE(R_{estDec+}|R_{Apriori})$ for finding frequent itemsets by varying the value of a merging threshold S_{merge}. The terms $R_{estDec+}$ and $R_{Apriori}$ denote the results of the *estDec+* method and the *Apriori* algorithm [2] respectively. These two results are found after all the transactions of a data set are processed. The reason why the *Apriori* algorithm is adopted as comparison target is that it offers the entire result set with exact supports of itemsets. For the same value of S_{merge}, the *ASE* is increased but the memory usage is decreased as the value of δ is set to be higher. Similarly, for the same value of δ, the *ASE* is increased but the memory usage is decreased as the value of S_{merge} is set to be smaller. This is because more nodes in a CP-tree are merged as the value of δ is increased or the value of S_{merge} is decreased. Fig. 9(b) shows the performance of the *estDec+* method for finding maximal frequent itemsets. The $ASE(R_{estDec+}|R_{Apriori\_MFI})$ is much less than the *ASE* of finding frequent itemsets for the same values of δ and S_{merge}. This is because most of maximal frequent itemsets are more accurately monitored without the merge-count estimation. Fig. 9(c) shows the $ASE(R_{estDec+}|R_{Apriori\_MFI})$ for the two count estimation functions $f_1(m,j)$ and $f_2(m,j)$ in Section 4.2 IV.B. In this experiment, the value of S_{merge} is set to 0.001. The accuracy of $f_2(m,j)$ is better than that of $f_1(m,j)$. Fig. 9(d) shows the effect of the period of a pruning operation on the memory usage of the *estDec+* method. The values of δ and S_{merge} are set to 0.001 and 0.003 respectively. In this experiment, three pruning periods $f = 1000$, $f = 5000$, and $f = 10{,}000$ are compared. The pruning period $f = 1000$ means that a forced pruning operation is performed whenever 1000 new transactions are processed. The memory usage of the *estDec+* method is decreased as the period f is shortened. Consequently, if memory usage is the primary constraint in a mining process, a force-pruning operation should be performed more frequently.

Fig. 10 shows the compression capability of a CP-tree in terms of memory usage as well as the number of nodes by varying the value of a merging gap threshold δ. In this experiment, the value of S_{merge} is set to 0.001. Fig. 10(a) illustrates how much the required size of a CP-tree can be reduced for the three synthetic data sets. Fig. 10(b) illustrates how many nodes of a CP-tree can be reduced for the same conditions in Fig. 10(a). When the value of δ becomes larger, the size of a CP-tree as well as the number of nodes is decreased. However, notice that there is a lower bound for the size of a CP-tree since the size of a merged node is increased proportionally to the number of merged itemsets in the node. On the contrary, the number of nodes in a CP-tree can be flexibly reduced as shown in this figure. This indicates that the entire set of frequent itemsets can be flexibly compressed into a concise set of representative frequent itemsets. The effects of δ on the processing time of the proposed method are illustrated in Fig. 10(c). As expected, the processing time is proportional to the value of δ.

Fig. 11 shows the effects of a merging threshold S_{merge} on the $ASE(R_{estDec+}|R_{Apriori})$. Fig. 11(a) shows the accuracy of those frequent itemsets that are in the upper part of a CP-tree. In other words, the accuracy of those frequent itemsets whose supports are greater than or equal to S_{merge} is illustrated. As the value of S_{merge} is decreased, the accuracy is degraded. As shown in Fig. 11(b), the accuracy of the frequent itemsets in the lower part is much more accurate.

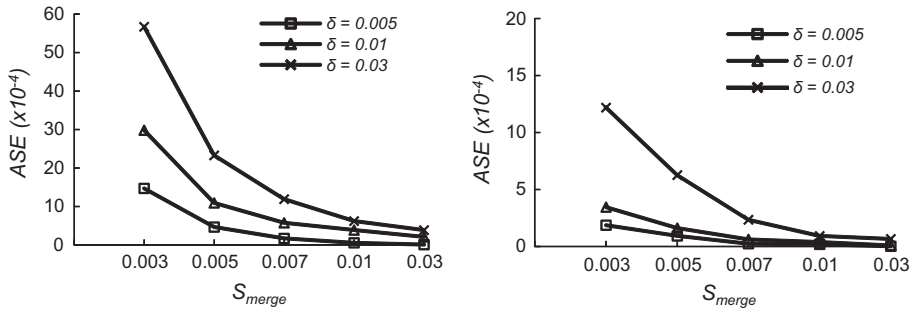
Fig. 12 shows how to confine the use of memory space adaptively by the *estDec+* method. In this experiment, the data set *WebLog* is used. The values of S_{min} and S_{merge} are set to 0.003. Furthermore, the values of M_U, M_L, and M_A are set to 95 MB, 85 MB, and 100 MB respectively. The initial 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 1×10^5 transactions. This is because the size of its prefix tree becomes greater than that of the confined memory space. On the other hand, the *estDec+* method can successfully execute the data set by adjusting the value of δ adaptively for the same situation. Fig. 12(a) illustrates the trace of the value of δ in this experiment. As shown in Fig. 12(b), the memory usage of the *estDec+* method is kept between the upper bound M_U and the lower bound M_L at all times. As expected, the value of δ is more widely fluctuated for the larger value of α. Fig. 12(c) shows the *ASEs* of the *estDec+* method in finding frequent itemsets. One more interesting fact is that the *ASE* is also affected by the value of α. When the value of α is set to be large, the *ASE* is increased.

In Fig. 13, the performance of the *estDec+* method is closely compared with that of the *estDec* method. For the same value of S_{sig}, the memory usage of the *estDec+* method is always less than that of the *estDec* method as shown in Fig. 13(a).



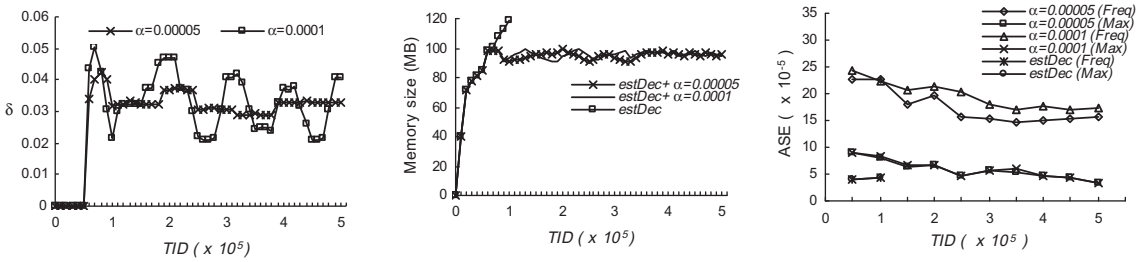
(a) Memory usage (b) Number of nodes (c) Processing time

Fig. 10. Performance of the *estDec+* method varying δ.



(a) Accuracy of the upper part (b) Accuracy of the lower part

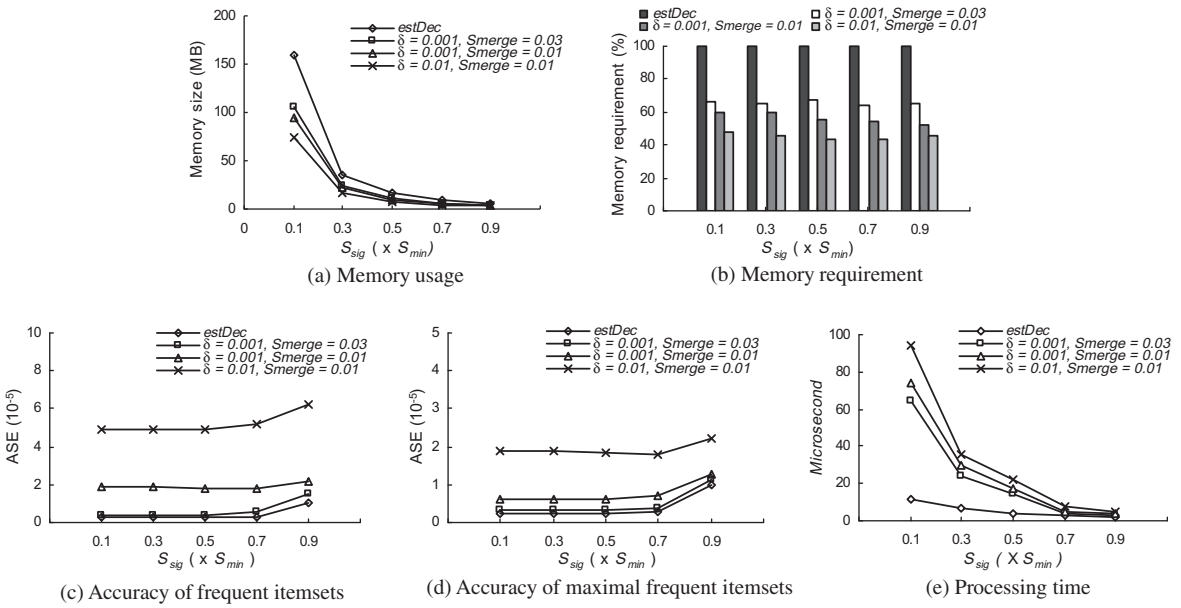
Fig. 11. Effects of S_{merge}.



(a) Adaptive change of δ (b) Memory usage (c) Accuracy

Fig. 12. Memory usage adaptation.

Fig. 13(b) shows the memory requirement of the *estDec+* method. The requirement is represented by the ratio of the memory space required by the *estDec+* method over the memory space required by the *estDec* method in order to execute the same dataset. By varying the value of S_{sig}, Fig. 13(c) and (d) shows the *ASE*s of the two methods. As either the value of S_{merge} is increased or the value of δ is decreased, the *ASE* of the *estDec+* method becomes closer to that of the *estDec* method since fewer nodes are merged. The average processing time per transaction shown in Fig. 13(e) is inversely proportional to the memory usage. This is because the processing time to interpret the information of itemsets represented by a node of a



(a) Memory usage (b) Memory requirement

(c) Accuracy of frequent itemsets (d) Accuracy of maximal frequent itemsets (e) Processing time

Fig. 13. Performance comparison.

CP-tree becomes longer as either the value of δ is increased or the value of S_{merge} is decreased. In Table 2, the performance of the *estDec+* method is compared with that of the *estDec* method on all the data sets. As the average number of items per transaction becomes larger, the two methods need more memory space since more itemsets should be maintained. For the same data set, the *estDec+* method requires much less memory space.

In Fig. 14, the performance of the *estDec+* method is shown by varying the values of δ and S_{merge}. The values of a decay-base *b* and a decay-base-life *h* are set to 2 and 10,000 respectively. In Fig. 14(a), the memory usages of both the decayed and undecayed *estDec+* methods are compared. For the same values of δ and S_{merge}, the memory usage of the decayed *estDec+* method is always less than that of the undecayed method since fewer significant itemsets are maintained by the CP-tree of the decayed *estDec+* method. Fig. 14(b) shows the *ASE*s of the two methods by varying the value of S_{merge}. The accuracy of the decayed method is measured relatively to the result $R_{dApriori}$ of the Apriori algorithm with the same decay mechanism. In other words, $ASE(R_{estDec+\_decay}|R_{dApriori})$ is used. The result of the decayed *estDec+* method is similar to that of the undecayed method for the same values of δ and S_{merge}.

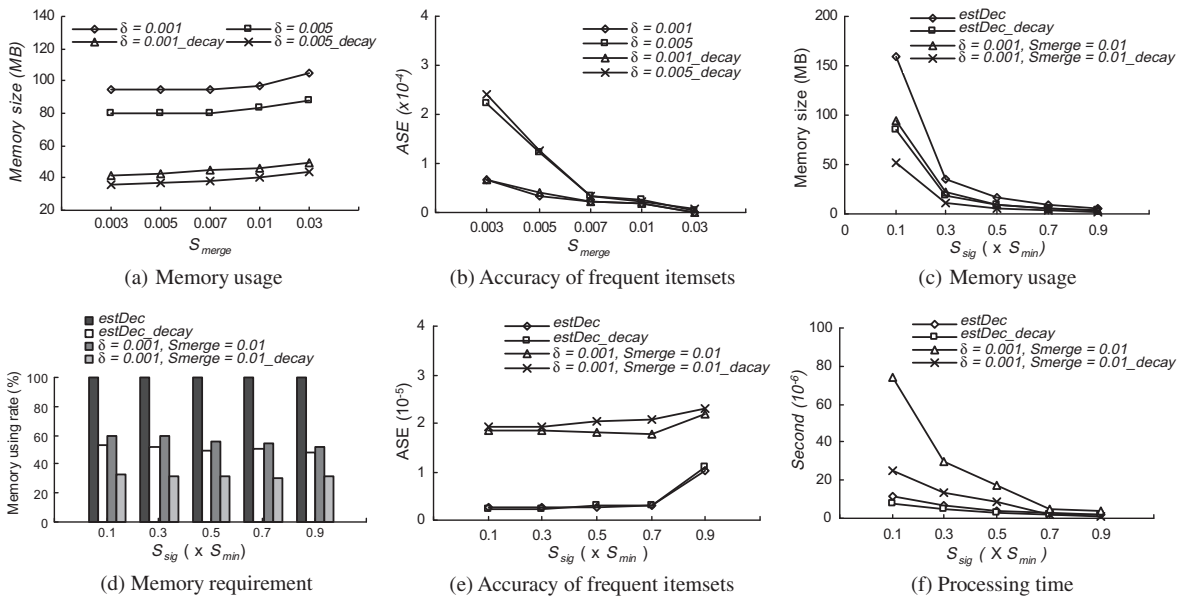
As shown in Fig. 14(c), for the same value of S_{sig}, the memory usage of the decayed *estDec+* method is always less than that of the undecayed *estDec* method. Fig. 14(d) shows the maximum memory requirement of each method. By varying the value of S_{sig}, Fig. 14(e) shows the *ASE*s of the two methods. Regardless of the value of S_{sig}, the *ASE* of the decayed *estDec+* method is also similar to that of the undecayed *estDec+* method for the same values of δ and S_{merge}. In Fig. 14(f), the average processing time per transaction is compared. For the same values of S_{sig}, δ, and S_{merge}, the processing time of the decayed *estDec+* method is always less than that of the undecayed *estDec* method. This is because more itemsets are maintained in the CP-tree of the undecayed *estDec* method.

In Fig. 15, the memory usages of *estDec+* and *CFI-Stream* [19] are compared for the datasets *T5.I4.D1000K* and *T10.I4.D1000K*. In this experiment, the values of δ and S_{merge} are set to 0 and 0.003 respectively. As the value of a decay base *b* is decreased, the *estDec+* method requires more memory space. This is because the old count of a significant itemset is decayed slowly, so that more itemsets are monitored by its prefix tree. Moreover, the memory usage of the *estDec+* method

Table 2
Performance comparison with various data sets.

	estDec		*estDec+*					
			$\delta = 0.001$, $S_{merge} = 0.003$		$\delta = 0.001$, $S_{merge} = 0.001$		$\delta = 0.01$, $S_{merge} = 0.001$	
	Memory	*ASE*	*Memory*	*ASE*	*Memory*	*ASE*	*Memory*	*ASE*
T10.I4.D1000K	158.52	0.257	104.98	0.326	94.98	0.619	74.96	1.865
T5.I4.D1400K	128.54	0.294	89.43	0.415	80.73	0.692	68.74	2.201
T15.I6.D100K	237.81	0.237	156.29	0.382	121.95	0.589	99.37	1.805
WebLog	127.04	0.251	91.60	0.319	82.31	0.628	70.48	1.927

Memory (MB), *ASE* (10^{-5}).



(a) Memory usage (b) Accuracy of frequent itemsets (c) Memory usage

(d) Memory requirement (e) Accuracy of frequent itemsets (f) Processing time

Fig. 14. Effects of information differentiation.

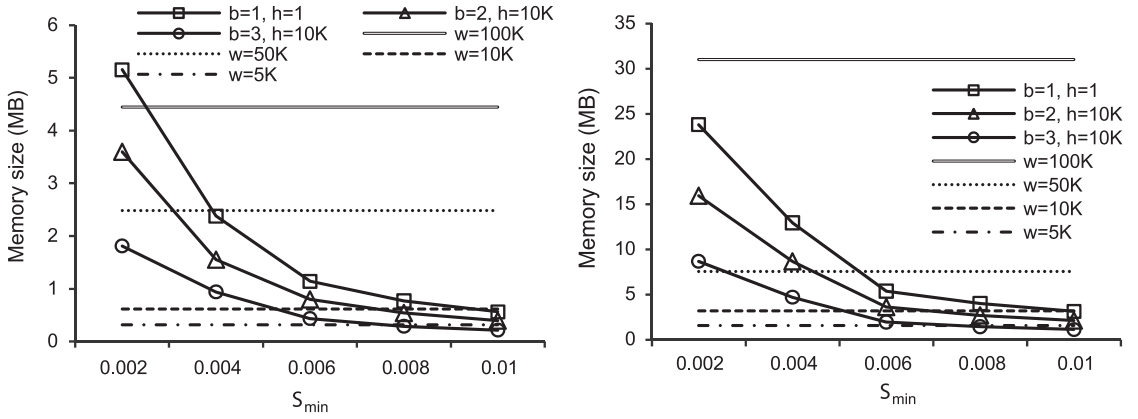


Fig. 15. Performance of the *estDec+* method and CFI-Stream.

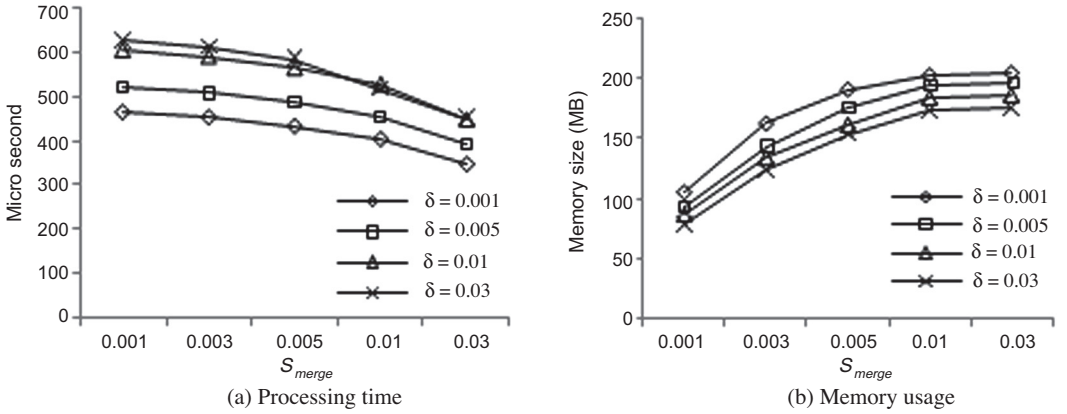


Fig. 16. Performance of the *estDec+* method for BMS-WebView-2.

decreases as the value of S_{min} is increased since the number of itemsets monitored by its prefix tree is decreased. On the other hand, the memory usage of *CFI-Stream* remains the same regardless of S_{min}. This is because *CFI-Stream* stores not only frequent closed itemsets but also infrequent closed itemsets in its *DIU tree*. The number of closed itemsets in a *DIU tree* tends to be increased as the size of a sliding window is enlarged, so that its memory usage is increased as well.

In order to illustrate the performance of the proposed method on a real data set, the *BMS-WebView-2* data set in the KDD-CUP 2000 competition is experimented in Fig. 16. The data set contains several months of clickstream data in two e-commerce web sites. Each transaction is a web session viewing various product-related pages. The total number of transactions, the maximum length and average length of a transaction are 77,512, 161 and 5 respectively. Fig. 16 shows the performance of the proposed method by varying the value of the merging gap threshold δ. The values of S_{min} and S_{sig} are set to 0.001% and 30% respectively. As either the value of δ is increased or the value of S_{merge} is decreased, the number of merged nodes in a *CP-tree* is increased. Consequently the memory usage is decreased and the processing time per transaction is slightly increased. This is because the processing of a merged node consumes much more time.

7. Concluding remarks

The up-to-date analysis result of an online data stream should be traced in real-time and available at any moment. For this purpose, the current counts of all significant itemsets are kept in main memory by the *estDec* method. However, for a given value of S_{min}, the total number of significant itemsets can be continuously varied over time without any upper bound. Due to this reason, it is impossible to guarantee that all of them are maintained in a confined memory space at all times. In order to solve this problem, this paper introduces a CP-tree that can restructure itself adaptively. Whenever the size of a CP-tree becomes too large, it is dynamically adjusted to fit into the confined memory space by sacrificing its accuracy. Based on this characteristic of a CP-tree, the proposed *estDec+* method can provide a way sustaining the process for tracing frequent

itemsets in case a confined memory space is overflowed. Furthermore, given an online data stream, a CP-tree can instantly compress the entire set of frequent itemsets into a concise set of representative frequent itemsets. Unlike CFI's and MFI's, the compression ratio can be controlled flexibly by the value of a merging gap threshold δ. Furthermore, the support of every non-representative itemset can be estimated as well.

Acknowledgements

This work was supported by the core research program (No. 2011-0016648) and NRL Program (No. R0A-2006-000-10225-0) of the Korea Science and Engineering Foundation (KOSEF) Grant funded by the Korea Government (MEST).

References

[1] R.C. Agarwal, C.C. Aggarwal, V.V.V. Prasad, Depth first generation of long patterns, in: Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2000, pp. 108–118.
[2] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings of the 20th International Conference on Very Large Data, Bases, 1994, pp. 487–499.
[3] R.J. Bayardo, Efficiently mining long patterns from databases, in: Proceedings of ACM Special Interest Group on Management of Data, 1998, pp. 85–93.
[4] S. Brin, R. Motwani, J.D. Ullman, S. Tsur, Dynamic itemset counting and implication rules for market basket data, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1997, pp. 255–264.
[5] D. Burdick, M. Calimlim, J. Gehrke, MAFIA: a maximal frequent itemset algorithm for transactional databases, in: Proceedings of the 17th International Conference on Data, Engineering, 2001, pp. 443–452.
[6] J.H. Chang, W.S. Lee, Finding recent frequent itemsets adaptively over online data streams, in: Proceedings of the 9th ACM SIGKDD International conference on Knowledge Discovery and Data Mining, 2003, pp. 487–492.
[7] M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in: Proceedings of the 29th International Colloquium on Automata, Language and Programming, 2002, pp. 693–703.
[8] Y. Chi, H. Wang, P. Yu, R. Muntz, Moment: Maintaining closed frequent itemsets over a stream sliding window, in: Proceedings of the 4th IEEE International Conference on Data Mining, 2004, pp. 59–66.
[9] Z. Chong, J.X. Yu, H. Lu, Z. Zhang, A. Zhou, False-negative frequent items mining from data streams with bursting, in: Proceedings of the 10th International Conference on Database Systems for Advanced Applications, 2005, pp. 422–434.
[10] M. Datar, A. Gionis, P. Indyk, R. Motwani, Maintaining stream statistics over sliding windows, in: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, 2002, pp. 635–644.
[11] E.D. Demaine, A. Lopez-Ortiz, J.I. Munro, Frequency estimation of internet packet streams with limited space, in: Proc. 10th Annual European Symposium on Algorithms, 2002.
[12] G. Dong, J. Han, L.V.S. Lakshmanan, J. Pei, H. Wang, P.S. Yu, Online mining of changes from data streams: research problems and preliminary results, in: Proceedings of the Workshop on Management and Processing of Data Streams, 2003.
[13] M. Garofalakis, J. Gehrke, R. Rastogi, Querying and mining data streams: you only get one look, in: Tutorial Notes of the 28th International Conference on Very Large Data, Bases, 2002.
[14] K. Gouda, M. Zaki, GenMax: an efficient algorithm for mining maximal frequent itemsets, Data Min. Knowl. Disc. 11 (2005) 1–20.
[15] S. Guha, N. Koudas, Approximating a data stream for querying and estimation: algorithms and performance evaluation, in: Proceedings of the 18th International Conference on Data, Engineering, 2002, pp. 567–576.
[16] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2000, pp. 1–12.
[17] C. Hidber, Online association rule mining, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1999, pp. 145–156.
[18] H. Javitz, A. Valdes, The NIDES Statistical Component Description and Justification, Annual Report, SRI International, 1994.
[19] N. Jiang, L. Gruenwald, CFI-stream: mining closed frequent itemsets in data streams, in: Proceedings of 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2006.
[20] D. Lin, Z.M. Kedem, Pincer-search: an efficient algorithm for discovering the maximum frequent set, IEEE Trans. Knowl. Data Eng. (2002).
[21] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proceedings of the 28th International Conference on Very Large Data, Bases, 2002, pp. 346–357.
[22] A. Savasers, E. Omiecinski, S. Navathe, An efficient algorithm for mining association rules in large databases, in: Proceedings of the 21st International Conference on Very Large Data, Bases, 1995, pp. 432–444.
[23] W-G. Teng, M-S. Chen, P.S. Yu, A regression-based temporal pattern mining scheme for data streams, in: Proceeding of the 29th International Conference on Very Large Database, Berlin, Germany, 2003.
[24] J. Wang, J. Han, J. Pei, Closet+: searching for the best strategies for mining frequent closed itemsets, in: Proceedings of 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2003.
[25] D. Xin, J. Han, X. Yan, H. Cheng, Mining compressed frequent-pattern sets, in: Proceedings of the 31st International Conference on Very Large Data, Bases, 2005.
[26] M. Zaki, C. Hsiao, Charm: an efficient algorithm for closed itemset mining, in: Proceedings of SIAM Conference on Data Mining, 2002.
[27] J. Cheng, Y. Ke, W. Ng, A survey on algorithms for mining frequent itemsets over data streams, Appeared Knowledge Inf. Syst. 16 (1) (2008) 1–27.
[28] Hua-Fu Li, Suh-Yin Lee, Man-Kwan Shan, Online mining (recently) maximal frequent itemsets over data streams, in: Research Issues in Data Engineering: Stream Data Mining and Applications, RIDE-SDMA 2005, 2005, pp. 11–18.
[29] K. Li, Y. Yan Wang, M. Ellahi, H. An Wang, Mining recent frequent itemsets in data streams, in: The 5th International Conference on Fuzzy Systems and Knowledge, Discovery, 2008.
[30] X. Liu, J. Guan, P. Hu, Mining frequent closed itemsets from a landmark window over online data streams, Comput. Math. Appl. 57 (6) (2009) 927–936.
[31] J. Gama, Knowledge Discovery from Data Streams, Chapman & Hall/CRC, Boca Raton, Florida, 2010.
[32] M. Sayed-Mouchaweh, E. Lughofer, Learning in Non-Stationary Environments: Methods and Applications, Springer, New York, 2012.
[33] E. Lughofer, Evolving Fuzzy Systems: Methodologies, Advanced Concepts and Applications, Springer, Berlin Heidelberg, 2011.