Computing and Informatics, Vol. 33, 2014, 609–632

IMPROVING EFFICIENCY OF INCREMENTAL MINING BY TRIE STRUCTURE AND PRE-LARGE ITEMSETS

Thien-Phuong Le

Pacific Ocean University, Nha Trang, Viet Nam
e-mail: phuonglt@pou.edu.vn

Bay Vo

Division of Data Science, Ton Duc Thang University
Ho Chi Minh City, Viet Nam
&
Faculty of Information Technology, Ton Duc Thang University
Ho Chi Minh City, Viet Nam
e-mail: vodinhbay@tdt.edu.vn

Tzung-Pei Hong

Department of Computer Science and Information Engineering
National University of Kaohsiung, Kaohsiung, Taiwan, R.O.C.
e-mail: tphong@nuk.edu.tw

Bac Le

Department of Computer Science
University of Science, VNU-HCM, Viet Nam
e-mail: lhbac@fit.hcmus.edu.vn

Dosam Hwang*

Department of Computer Engineering, Yeungnam University, South Korea
e-mail: dshwang@yu.ac.kr

Abstract. Incremental data mining has been discussed widely in recent years, as it has many practical applications, and various incremental mining algorithms have been proposed. Hong et al. proposed an efficient incremental mining algorithm for handling newly inserted transactions by using the concept of pre-large itemsets. The algorithm aimed to reduce the need to rescan the original database and also cut maintenance costs. Recently, Lin et al. proposed the Pre-FUFP algorithm to handle new transactions more efficiently, and make it easier to update the FP-tree. However, frequent itemsets must be mined from the FP-growth algorithm. In this paper, we propose a Pre-FUT algorithm (Fast-Update algorithm using the Trie data structure and the concept of pre-large itemsets), which not only builds and updates the trie structure when new transactions are inserted, but also mines all the frequent itemsets easily from the tree. Experimental results show the good performance of the proposed algorithm.

Keywords: Data mining, frequent itemset, incremental mining, pre-large itemset, trie

1 INTRODUCTION

Data mining has produced a variety of efficient techniques in recent years, and the approaches may be classified as those working on transaction databases, temporal databases, relational databases, and multimedia databases, among others. Many mining methods have also been proposed, such as techniques for association rules, classification, clustering and sequential patterns [4]. Among these, mining association rules in transaction databases is the most common approach in data mining [1, 2, 3, 4, 5, 8, 11, 12].

Many algorithms for mining association rules from transactions have been proposed, most of which are based on the Apriori algorithm [3], which generates and tests candidate itemsets in each level. However, this may require scanning databases iteratively, and cause high computational cost. An important data structure used in the Apriori algorithm is the hash-tree [3]. In order to improve performance of the Apriori algorithm, Bodon and Ronyai [6] adopted the trie data structure to replace hash-trees. In addition to Apriori-based algorithms, Han et al. [13] proposed the Frequent-Pattern tree (FP-tree) structure to efficiently mine frequent itemsets (FIs) without candidate generation. Both the Apriori and the FP-tree mining approaches utilize batch mining, which means that they must process all the transactions in a batch way.

In real-world applications, new transactions are usually inserted into databases incrementally. The first incremental mining algorithm was the Fast-Updated algorithm (called FUP), proposed by Cheung et al. [9]. Although the FUP algorithm could indeed improve mining performance for incrementally growing databases, the

* Corresponding author

original databases still needed be rescanned. Hong et al. then proposed the concept of pre-large itemsets to further reduce the need for rescanning [14]. A pre-large itemset is defined by two support thresholds. The upper support threshold is the same as that used in the conventional mining algorithms, while the lower one is defined as the lowest support ratio for an itemset to be treated as pre-large. An itemset with a support ratio below the lower threshold is thought of as a small itemset. This algorithm does not need to rescan the original database until a number of new transactions has been inserted. Since rescanning the database requires considerable computation time, the maintenance cost can thus be reduced with the pre-large-itemset algorithm.

Various algorithms have been developed based on the concept of pre-large itemsets [15, 16, 17, 26]. For example, Hong et al. modified the FP-tree structure and designed a fast updated frequent pattern tree (FUFP tree) [15] for handling newly inserted transactions based on the FUP algorithm [9]. The FUFP-tree structure they used was similar to the FP-tree structure, except that the links between parent nodes and their child nodes were bi-directional. After that, Lin et al. proposed the Pre-FUFP algorithm, which updated and constructed the FUFP tree when new transactions are inserted. In addition to transactions insertion, some algorithms for handling deleting and modifying transactions have also been developed [16, 17].

In this paper, we propose the Pre-FUT algorithm for handling newly inserted transactions based on the pre-large itemsets concept and the trie data structure [6]. The trie structure used in the proposed algorithm can not only keep candidates, but also frequent itemsets and pre-large itemsets. By adopting the trie structure and some pruning techniques, the process of generating candidates and determining supports becomes easier and faster. Based on trie, we can find FIs directly from the tree-building process. The experimental results also show that the proposed algorithm has good performance for incrementally handling inserted transactions.

The rest of this paper is organized as follows. Related works are reviewed in Section 2, and the proposed Pre-FUT maintenance algorithm is described in Section 3. An example to illustrate the proposed algorithm is given in Section 4, while the experimental results showing the performance of the proposed algorithm are provided in Section 5. Finally, the conclusion and directions for future work are presented in Section 6.

2 RELATED WORKS

In this section, some related research is briefly reviewed on the topics of the hash-tree data structure, trie data structure and pre-large-itemset algorithm.

2.1 Hash-Tree Data Structure

A hash-tree is a data structure that can reduce the time needed to determine the support of itemsets in the Apriori algorithm, and new candidate itemsets can be

stored in a hash-tree. A hash-tree is a rooted (downward), directed tree. A node of the hash-tree contains either a list of itemsets (a *leaf* node) or a hash table (an *interior* node). In an interior node, each bucket of the hash table points to another node. The root of the hash-tree is defined to be at depth 1. An interior node at depth d points to nodes at depth $d + 1$. Itemsets are stored in the leaves. When we add an itemset c, we start from the root and go down the tree until we reach a leaf. At an interior node at depth j, we decide which branch to follow by applying a hash function to the j^{th} item of the itemset. All nodes are initially created as leaf nodes. When the number of itemsets in a leaf node exceeds a specified threshold, the leaf node is converted to an interior one. Figure 1 shows a hash-tree, which contains five candidates of size 3. Here, the items are capital letters. The hash value of a letter is its sequential number in the English alphabet (0 for A, 1 for B, etc.). As G and M have the same hash value of 1, the sets $\{A, E, G\}$ and $\{A, E, M\}$ are stored in the same leaf. The root has two children and the tree has four leaves.

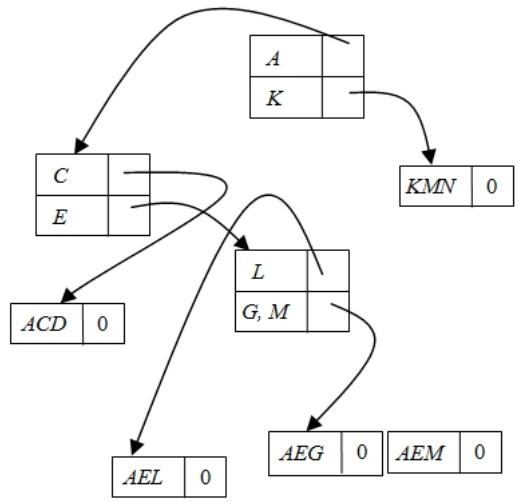

Figure 1. A hash-tree containing five candidates

Assume that we have a transaction t and we want to determine the support of the candidates in the hash-tree. Here, we do not generate and test every k-subset of t. Navigation in the hash-tree allows us to eliminate from consideration whole families of k-itemsets with a given prefix. If we are at an interior node and have reached it by hashing the item i, we hash on each item that comes after i in t and recursively repeat this task. When we arrive at a leaf, then we have to test explicitly if the candidates stored in the leaf are actually subsets of t. The benefit of using a hash-tree is that the number of explicit tests is much less than total number of candidates. A more detailed account of this process can be found in [3, 19].

2.2 Trie Data Structure

The trie data structure was originally introduced to store and retrieve words in a dictionary efficiently [25, 20]. A trie is a rooted, (downward) directed tree, like a hash-tree. The root is at depth 0, and a node at depth d can point to nodes at depth $d+1$. A pointer is also called an *edge* or a *link*, which is labeled by a letter. Each leaf represents a word which is the concatenation of the letters in the path from the root to the node. Note that if the first k letters are the same in two words, then the first k steps on the two paths are the same as well. The trie data structure is suitable to store and retrieve not only words, but any finite ordered sets.

Bodon and Ronyai [6] extended the trie structure to mine FIs with good performance. In their approach, the *trie* structure stores not only candidates, but also FIs. In their setting, a link is labeled by an item symbol, and the alphabet is thus the (ordered) set of all items J. A path in a trie is a candidate itemset which includes one or more links, with labels in an increasing order. Small itemsets are removed from the trie after the procedure of determining the support of the candidates finishes, and the *trie* only stores all the FIs. A k-itemset $C = \{i_1, i_2, \ldots, i_k\}$ can thus be viewed as a word $i_1, i_2, \ldots, i_k$ composed of letters from J, with $i_1 < i_2 < \ldots < i_k$. In the task of building the *trie* from the candidate itemsets, when inserting an itemset we have to start from the root node. If we get to a node which has no link labeled with the next letter i_k of the itemset, then we create a new node and a link pointing to it, whose label is i_k. We repeat this procedure till we reach the end of the itemset. Figure 2 presents an example of a trie that stores the candidates $\{A, C, D\}$, $\{A, E, G\}$, $\{A, E, L\}$, $\{A, E, M\}$, and $\{K, M, N\}$. The authors showed that their approach could improve the performance of generating candidates and determining supports in the following ways.

1. New candidates were generated from pairs of nodes which had the same parents. That is, the two itemsets were the same except for the last items.
2. Some pruning techniques were adopted to determine the supports of itemsets efficiently. When the supports of the candidate k-itemsets need to be determined, their method does not generate all k-subsets of a transaction t, but instead stops if possible. More precisely, if the current processing is at a node at depth d, then it moves forward on those links that have the labels $i \in t$ with indices less than $|t| - k + d + 1$. For example, assume transaction $t = \{B, C, E, K, M\}$ and the supports of the candidate 3-itemsets are to be determined by the above method. The notation $id\{X\}$ is used to represent the index of item X in transaction t. First, the processing starts at root, which has two links $\{A\}$ and $\{K\}$. Because $\{K\} \in t$, $id\{K\} = 3$, and $id\{K\} \geq |t| - k + d + 1 (= 5 - 3 + 0 + 1 = 3)$, it can be confirmed that there are not 3-subsets of t with the first item being $\{K\}$.

Next, the authors proposed an improvement to the procedure for finding supported candidates, based on the fact that superfluous moves are usually performed in a trie search in the sense that there are no candidates in some of the directions

explored. To illustrate this, consider the following example. Assume Figure 3 shows the result of a trie search, in which only candidate $\{A, B, C, D, E\}$ is generated after frequent 4-itemsets are determined.

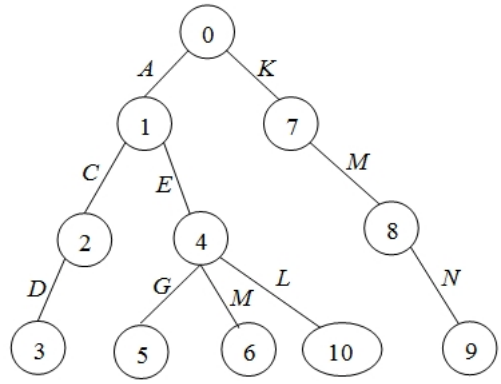

Figure 2. A *trie* data structure

When the supports of the candidate 5-itemsets need to be determined by the transaction $t = \{A, B, C, D, E, F, G, H, I\}$, every node of the *trie* needs to be visited, and thus 32 nodes are visited in this case. However, this is apparently unnecessary, since only one path leads to a node at depth 5. At a node with many links, the current processing mechanism has to decide which one to follow, because this will significantly affect the running time. To avoid superfluous traveling, at every node the length of the longest directed path that starts from it will be stored. When searching for k-itemset candidates at depth d, we move downward only if the maximal path length at this node is $k - d$. Note that this trick cannot be applied to a hash-tree, because a leaf at depth d does not necessarily store d-itemset candidates. Bodon and Ronyai also showed that storing counters at each node needs memory, but as their experiments proved, this approach can significantly reduce search time for candidate itemsets. More comparative results about the two data structures, hash-tree and trie, can be found in [6].

2.3 Some New Approaches for Incrementally Mining FIs

In this section, we will discuss in more detail approaches for incrementally mining FIs. As noted in Section 1, many incremental algorithms have been proposed, and each has advantages and disadvantages. There are currently two approaches to mining FIs in incremental databases. The first is based on generating candidates in each iteration, while the second does not need to do this. The first approach to incremental mining is known as the FUP algorithm [9], and it is similar to Apriori-like algorithms, which have to generate large number of candidates and repeatedly scan the database. In Thomas et al. [32], the negative border is maintained along

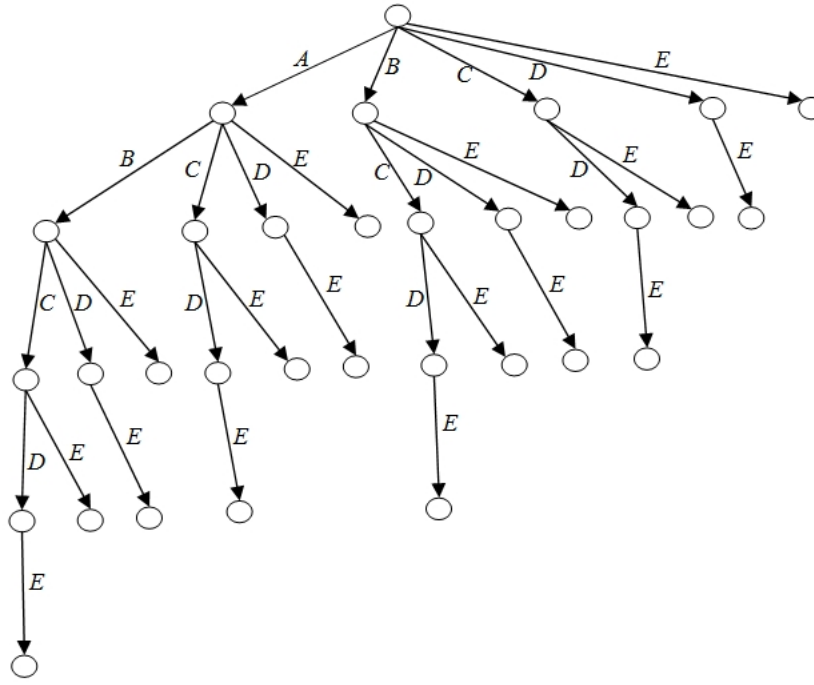

Figure 3. A *trie* with a single 5-itemset candidate

with the FIs to perform incremental updates, and this algorithm still requires a full scan of the whole database if an itemset outside the negative border gets added to the FIs or its negative border. This approach always generates candidates for each iteration. In the Apriori-based incremental algorithms, the tasks of generating candidates and determining their supports requires considerable time, and thus some techniques can be applied to overcome this weakness, such as using a hash-tree [3] or *trie* [6, 20].

The other approach to incremental mining is based on an FP-tree structure. This is a prefix-tree structure for storing compressed candidates and crucial information in transactions. An algorithm called AFPIM (Adjusting FP-tree for Incremental Mining) [24, 21] is designed to efficiently find new FIs based on adjusting the FP-tree structure. However, adjusting the FP-tree of the original database according to the changed transactions is computationally complex. A different algorithm, called EFPIM (Extending FP-tree for Incremental Mining) [27], has thus been proposed to find FIs with minimum re-computation when new transactions are added to or old transactions are removed from the database. This approach uses the structure EFP-tree (extended FP-tree), which is equal to the FP-tree when the algorithm is run for the first time and expanded during the next incremental mining. The EFP-tree of the original database is maintained to mine FIs without needing to rescan the

original database. However, the EFIPM algorithm still rescans the whole database when there is a frequent item in the updated database that does not appear in the EFP-tree built from the original database. This means that it is small in the original database. Therefore, the algorithm needs to rescan the original database to re-build EFP-tree.

In 2001, Hong et al. proposed the concept of pre-large itemsets [14]. A pre-large itemset is defined by two support thresholds, and in this approach the algorithm does not need to rescan the original database until a number of new transactions have been inserted. Based on the concept of pre-large itemsets, the maintenance for mining FIs in an incremental database can be reduced, and a number of related algorithms have been developed [15, 16, 17, 26, 22]. For example, Hong et al. modified the FP-tree structure and designed a fast updated frequent pattern tree (FUFP tree) [15] for handling newly inserted transactions based on the FUP algorithm [9]. In addition to transaction insertion, some algorithms have been developed to handle deleting and modifying transactions [16, 17]. In addition, the concept of pre-large itemsets has been also applied to mine sequential patterns in sequential databases. For example, Wang et al. [34, 35] used the concept of pre-large sequences for handling deleting and modifying transactions in sequential databases. In 2011, Hong et al. [18] applied the concept of pre-large sequences for mining sequential patterns in a sequential database when new transactions are frequently added into the database. Similar to mining FIs in an incremental database, the concept of pre-large sequences was proposed to reduce the need for rescanning original database. Like pre-large itemsets, pre-large sequences are defined by a lower support threshold and an upper support threshold that serve to avoid the direct movement of sequences from large to small and vice versa. The maintained algorithm does not require rescanning original databases until the accumulated amount of newly added sequences exceeds a safety threshold, which depends on the size of the database. Thus, as databases grow larger, the number of new transactions allowed before database rescanning is also required to grow. The pre-large-based approach thus becomes increasingly efficient as databases grow.

2.4 Pre-Large Itemset Algorithm

The pre-large itemsets algorithm was proposed by Hong et al. [14]. It is based on a safety threshold f to reduce the need to rescan the original databases to efficiently maintain the frequent itemsets. The safety number f of the inserted records is derived as follows:

$$f = \left\lfloor \frac{(S_u - S_l)d}{1 - S_u} \right\rfloor \tag{1}$$

where S_u is the upper threshold, S_l is the lower threshold, and d is the number of original transactions. A summary of the nine cases and their results are given in Table 1.

Cases 1, 5, 6, 8 and 9 will not affect the final large itemsets according to the weighted average of the counts. Cases 2 and 3 may remove existing large itemsets, and cases 4 and 7 may add new large itemsets. If we retain all large and pre-large itemsets with their counts after each pass, then cases 2, 3 and 4 can be handled easily. In addition, the ratio of the number of new transactions to the number of old transactions is usually very small in the maintenance phase, and this is more apparent as the database grows larger. It has been formally shown that an itemset in case 7 cannot possibly be large for the entire updated database as long as the number of new transactions is smaller than the number f [14].

Cases: Original – New	Results
Case 1: Large – Large	Always large
Case 2: Large – Pre-large	Large or pre-large, determined from existing information
Case 3: Large – Small	Large or pre-large or small, determined from existing information
Case 4: Pre-large – Large	Pre-large or large, determined from existing information
Case 5: Pre-large – Pre-large	Always pre-large
Case 6: Pre-large – Small	Pre-large or small, determined from existing information
Case 7: Small – Large	Pre-large or small, when the number of transactions is small
Case 8: Small – Pre-large	Small or pre-large
Case 9: Small – Small	Always small

Table 1. Nine cases and their results

3 PROPOSED PRE-FUT ALGORITHM

3.1 Notation

- D – the original database
- T – the set of new transactions
- U – the entire updated database, i.e., $D \cup T$
- d – the number of transactions in D
- t – the number of transactions in T
- S_l – the lower support threshold for pre-large itemsets
- S_u – the upper support threshold for large itemsets, $S_u > S_l$
- X – an itemset
- Tr^D – a *trie* storing the set of pre-large and large itemsets from D

- Tr^U – a *trie* storing the set of pre-large and large itemsets from $U = D \cup T$
- $count^T(X)$ – the number of occurrences of X in T
- $count^{Tr^D}(X)$ – the number of occurrences of X in Tr^D
- $count^{Tr^U}(X)$ – the number of occurrences of X in Tr^U

3.2 Pre-FUT Algorithm

As mentioned earlier, Bodon and Ronyai showed that generating candidates and determining item supports might become easy and fast with the aid of the trie data structure and pruning techniques [6, 23]. Hong et al. used the hash-tree data structure and the pre-large concept to efficiently mine FIs for incrementally inserted transactions [14]. In this paper, we adopt the *trie* data structure in Hong et al.'s method to further speed up the mining of frequent itemsets when new transactions are incrementally inserted into a database. We thus propose the Pre-FUT algorithm which is stated as follows.

INPUT: A lower support threshold S_l, an upper threshold S_u, a *trie* Tr^D storing large itemsets and pre-large itemsets derived from the original database consisting of $(d + c)$ transactions, and a set of t new transactions.

OUTPUT: A trie Tr^U storing large and pre-large itemsets, and rescanned itemsets from U.

The rescanned itemsets in the trie are small in the original database but pre-large or large in the new transactions. They are stored in the trie but not used to generate new candidates.

4 AN EXAMPLE TO ILLUSTRATE THE PROPOSED INCREMENTAL DATA MINING ALGORITHM

An example is given in this section to illustrate the proposed incremental data mining algorithm. Assume the initial data set includes the eight transactions shown in Table 2. For $S_l = 30\,\%$ and $S_u = 50\,\%$, the sets of large itemsets and pre-large itemsets for the given data are shown in Tables 3 and 4, respectively. A trie TrD storing the set of pre-large and large itemsets in the original database is shown in Figure 4.

Assume the two new transactions shown in Table 5 are inserted after the trie Tr^D is built. The Pre-FUT algorithm proceeds as follows, and the variable c is initially set at 0.

From line 1 of the algorithm, $f = \left\lfloor \frac{(S_u - S_l)d}{1 - S_u} \right\rfloor = \left\lfloor \frac{(0.5-0.3)8}{1-0.5} \right\rfloor = 3$

From lines 4 to 6, two newly inserted transactions are first scanned to get their 1-itemsets and counts. A trie Tr^U is then built from them, and the results are shown in Figure 5.

Algorithm 1 Pre-FUT algorithm

1: $f = \left\lfloor \frac{(S_u - S_l)d}{1 - S_u} \right\rfloor$
2: $k = 1$
3: Tr^U initially is the root node
4: add k-itemsets from the newly inserted transactions to Tr^U
5: **repeat**
6: scan T to determine $count^{Tr^U}(X) = count^T(X), \forall X \in Tr^U$ with $|X| = k$
7: **for all** itemset $X \in Tr^U$ with $|X| = k$ **do**
8: **if** X does not exist in Tr^D **then**
9: **if** $\left(\frac{count^{Tr^U}(X)}{t} \geq S_l \right)$ **then** ▷ Cases 7, 8, 9 in Table 1
10: mark X as a rescanned itemset in Tr^U
11: **end if**
12: **else**
13: remove X from Tr^U
14: **end if**
15: **end for**
16: **for all** itemset $X \in Tr^D$ with $|X| = k$ **do** ▷ Cases 1, 2, 3, 4, 5, 6 in Table 1
17: **if** X exists in Tr^U **then**
18: **if** $\left(\frac{(count^{Tr^D}(X) + count^{Tr^U}(X))}{d+t+c} \geq S_l \right)$ **then**
19: $count^{Tr^U}(X) = count^{Tr^U}(X) + count^{Tr^D}(X)$
20: **else**
21: remove X from Tr^U
22: **end if**
23: **else**
24: **if** $\left(\frac{count^{Tr^D}(X)}{d+t+c} \geq S_l \right)$ **then**
25: add itemset X to Tr^U with $count^{Tr^U}(X) = count^{Tr^D}(X)$
26: **end if**
27: **end if**
28: **end for**
29: **if** $(t + c > f)$ **then**
30: rescan the original database to determine whether the rescanned itemsets are large or pre-large
31: **end if**
32: $k = k + 1$
33: **until** $Trie\_gen(Tr^U, k) \neq \emptyset$
34: ▷ *Trie_gen* generates new candidate k-itemsets from Tr^U and stores them in Tr^U
35: **if** $(t + c > f)$ **then**
36: $d = d + t + c$
37: $c = 0$
38: **else**
39: $c = c + t$
40: **end if**

Lines 7 to 15 mark the 1-itemsets which are small in the original database. They are the 1-itemsets in Tr^U, which do not exist in Tr^D but are pre-large or large itemsets in the new transactions. In this example, only the 1-itemset $\{F : 1\}$ is small in the original database, but it is pre-large in the new transactions and is thus marked as a rescanned 1-itemset. The results are shown in Figure 6.

Lines 16 to 28 then process the 1-itemsets which are pre-large or large in the original database. In this example, the large or pre-large 1-itemsets are $\{A : 6, B : 5, C : 6, D : 3, E : 7\}$. Since they also exist in Tr^U, their total counts are calculated using $count^{Tr^U}(X) = count^{Tr^U}(X) + count^{Tr^D}(X)$. Table 6 shows the results.

Original database	
TID	*Items*
1	*ACE*
2	*ABDE*
3	*BCDE*
4	*ACE*
5	*ACE*
6	*ABC*
7	*BDE*
8	*ABCE*

Table 2. An example of an original database

Large itemsets					
1 item	Count	2 items	Count	3 items	Count
A	6	*AC*	5	*ACE*	4
B	5	*AE*	5		
C	6	*BE*	4		
E	7	*CE*	5		

Table 3. The large itemsets derived from the original database

Pre-large itemsets					
1 item	Count	2 items	Count	3 items	Count
D	3	*AB*	3	*BDE*	3
		BC	3		
		BD	3		
		DE	3		

Table 4. The pre-large itemsets derived from the original database

Original database	
TID	*Items*
9	*ABCD*
10	*CEF*

Table 5. Two new transactions

Items	*Count*
A	7
B	6
C	8
D	4
E	8

Table 6. Two new transactions

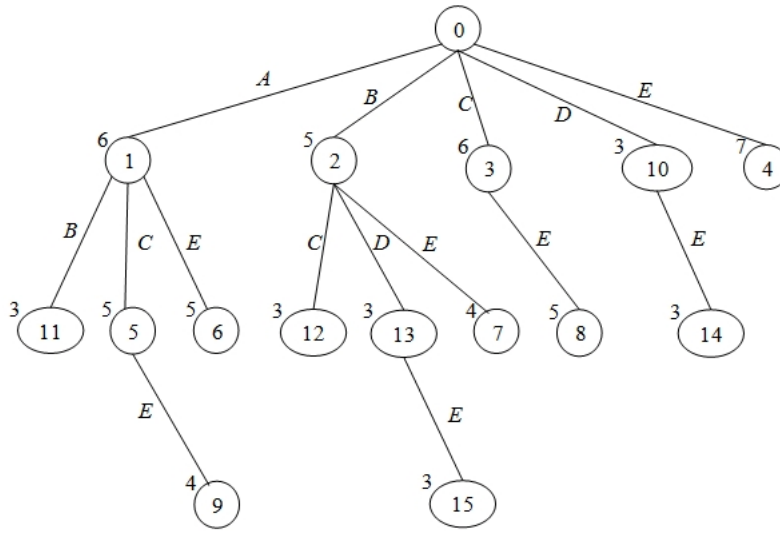

Figure 4. The *trie* Tr^D storing the set of pre-large and large itemsets in the original database

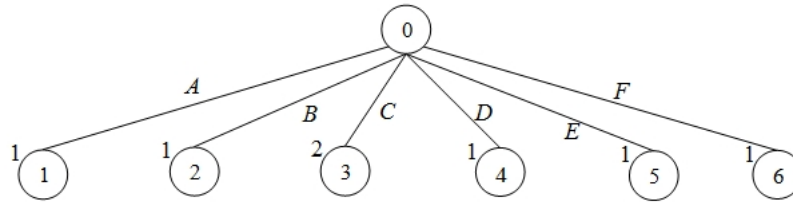

Figure 5. The *trie* Tr^U that is constructed

The new support ratios of A, B, C, D and E are then calculated. For example, the new support ratio of A is $7/(8+2+0)$, which is larger than 0.3. A is thus a pre-large itemset. In this example, the support ratios of A, B, C, D and E are greater than 0.3, so that their counts will be updated in the trie. The results are shown in Figure 7.

Since $t + c = 2 + 0 = 2 \leq f$, rescanning the original database is unnecessary (from lines 29 to 31), and nothing is done, and then at line 32 k is set at 2. In this algorithm, the sub-function $Trie\_gen(Tr^U, k)$ generates candidate k-itemsets with $k = 2$ from trie Tr^U. In this example, $Trie\_gen(Tr^U, k) \neq \emptyset$ with $k = 2$, such that the algorithm is repeated. Note that the marked nodes will not be used to generate new candidates. The results of the *trie* Tr^U after the new candidate 2-itemsets are generated with their counts from the new transactions are shown in Figure 8.

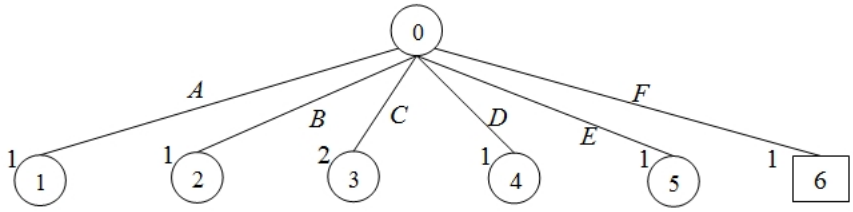

Figure 6. The *trie* Tr^U after the rescanned itemset is marked

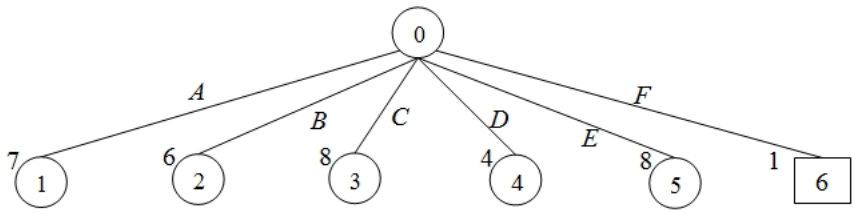

Figure 7. The *trie* Tr^U after line 28

Similar to the first iteration, lines 7 to 15 mark the 2-itemsets which are small in the original database. They are the 2-itemsets in Tr^U which do not exist in Tr^D but are pre-large or large itemsets in the new transactions. In this example, these 2-itemsets include $\{AD : 1, CD : 1\}$. The results are shown in Figure 9.

Lines 16 to 28 then process the 2-itemsets which are pre-large or large in the original database. In this example, these 2-itemsets are $\{AB : 3, AC : 5, AE : 5, BC : 3, BD : 3, BE : 4, CE : 5, DE : 3\}$. They exist in Tr^U, so their total counts are calculated using $count^{Tr^U}(X) = count^{Tr^U}(X) + count^{Tr^D}(X)$. In this example,

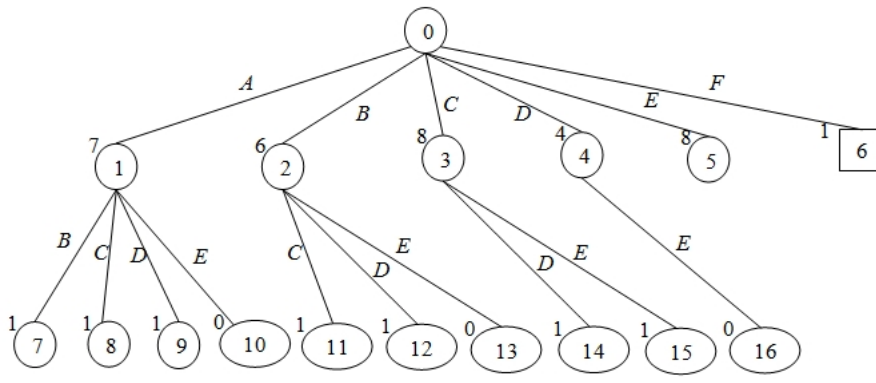

Figure 8. The *trie* Tr^U after line 33

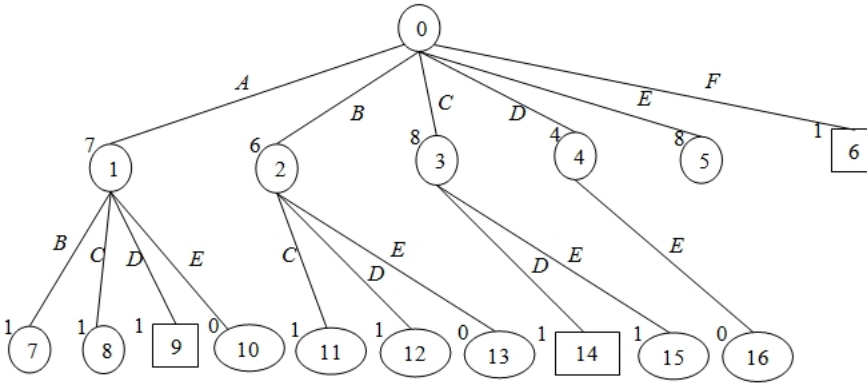

Figure 9. The *trie* Tr^U after the rescanned 2-itemsets are marked

the support ratios of $\{AB\}$, $\{AC\}$, $\{AE\}$, $\{BC\}$, $\{BD\}$, $\{BE\}$, $\{CE\}$ and $\{DE\}$ are greater than 0.3. The results are shown in Figure 10.

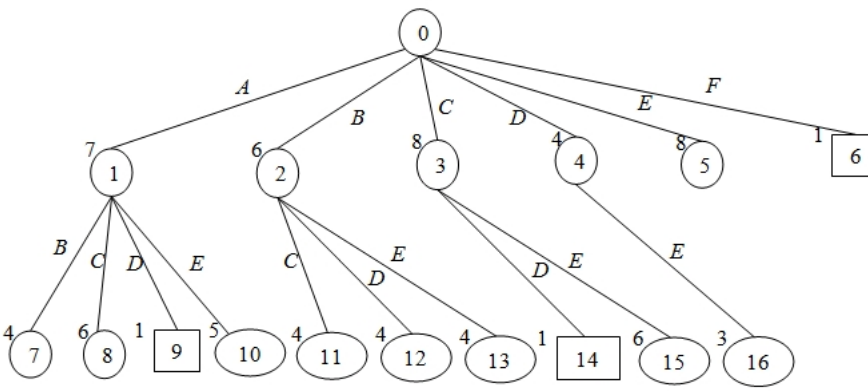

Figure 10. The *trie* Tr^U after line 28 for 2-itemsets

Since $t + c = 2 + 0 = 2 \leq f$, rescanning the original database is unnecessary, and nothing is done. Similarly, because $Trie\_gen(Tr^U, k) \neq \emptyset$ for $k = 3$, the algorithm is repeated. The results of the trie Tr^U after the new candidate 3-itemsets are generated with their counts from the two newly inserted transactions are shown in Figure 11.

Similarly, lines 7 to 15 mark the 3-itemsets which are small in the original database, and in this example only the 3-itemset $\{ABC : 1\}$ satisfies the condition. The other two 3-itemsets $\{ABE : 0\}$, $\{BCE : 0\}$ are removed from Tr^U. The results are shown in Figure 12.

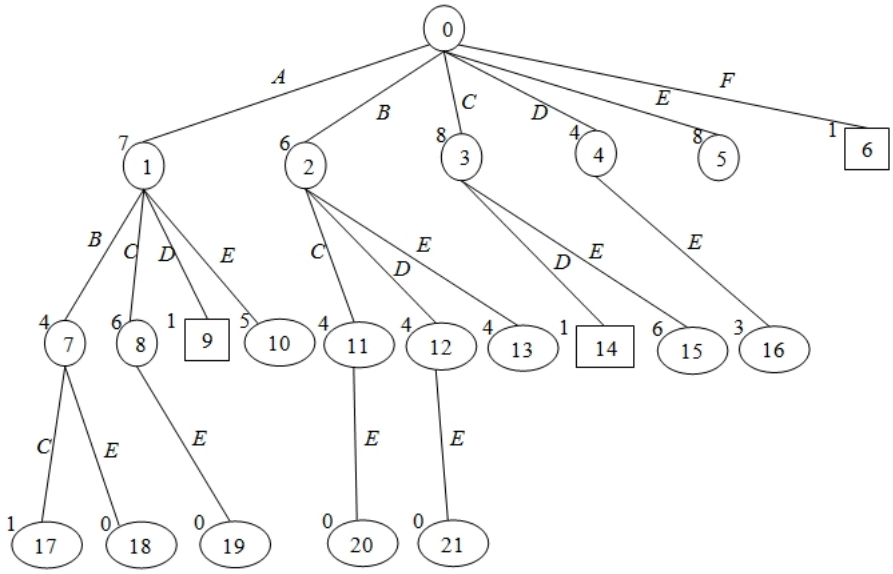

Figure 11. The *trie* Tr^U after line 33 for 3-itemsets

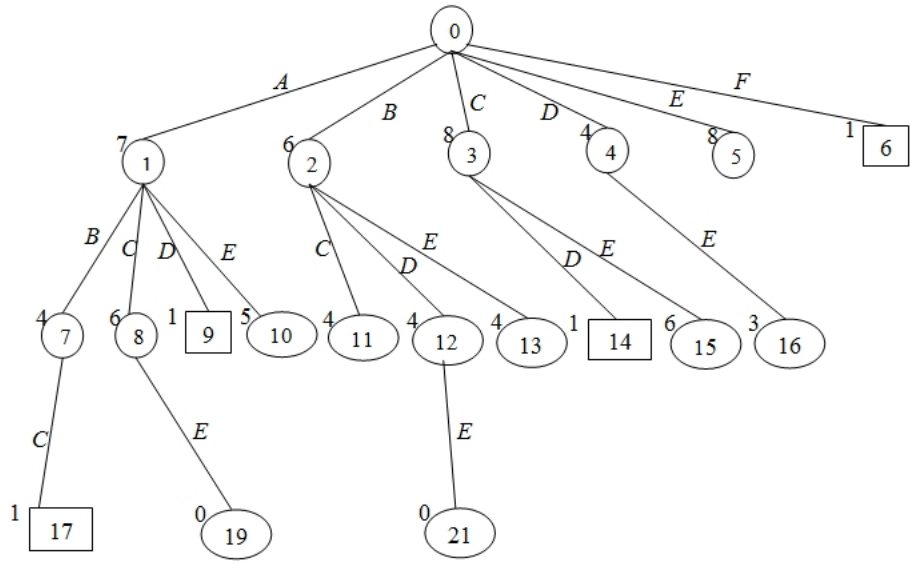

Figure 12. The *trie* Tr^U after line 15 for 3-itemsets

Lines 16 to 28 then process the 3-itemsets which are pre-large or large in the original database. In this example, the 3-itemsets include $\{ACE : 4, BDE : 3\}$. The results are shown in Figure 13.

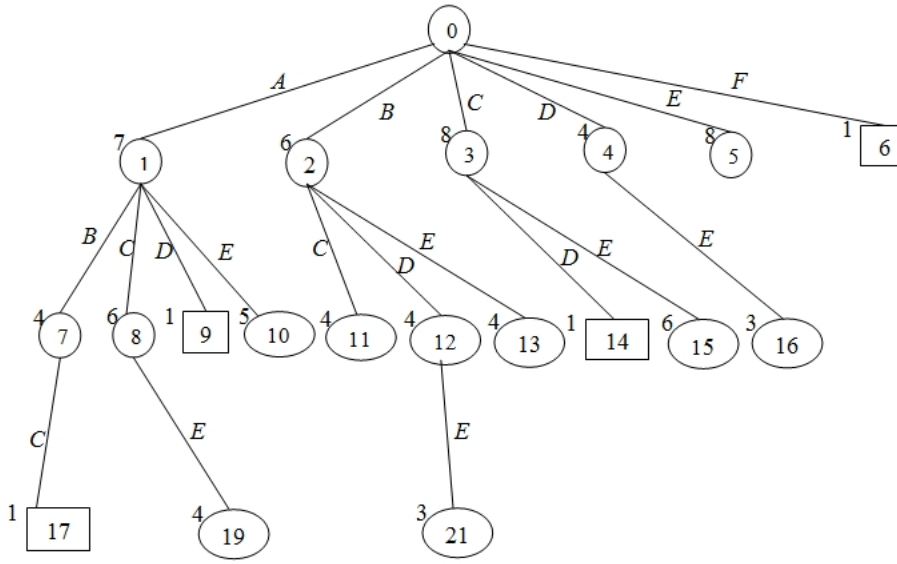

Figure 13. The *trie* Tr^U after line 28 for 3-itemsets

Because $Trie\_gen(Tr^U, k) = \emptyset$ for $k = 4$, the loop ends.

Lines 35 to 40 are then executed. Because $t + c = 2 \leq f$, $c = c + t = 2 + 0 = 2$ the result of the algorithm is thus the trie Tr^U that stores the pre-large and large itemsets for the whole updated transactions, as well as the rescanned itemsets. We can use Tr^U to tore the rescanned itemsets to mine next newly inserted transactions.

Note that the final value of c is 2 in this example and $f - c = 1$. It means that one more new transaction can be added without rescanning the original database.

5 EXPERIMENTAL RESULTS

Experiments were conducted to evaluate the performance of the proposed algorithm. All the algorithms were implemented on a PC with a Core 2 Duo (2×2 GHz) CPU and 2 GBs of RAM running Windows 7. All the algorithms were coded in C++. Two databases were used. One is Kosarak (with 990 002 transactions) the well-known and the other is T40I10D100K (with 100 000 transactions).

We compared the performance of the proposed Pre-FUT algorithm with that of the pre-large-itemset algorithm [14] which uses the hash-tree data structure. We do not compare our algorithm with the Pre-FUFP algorithm by Hong et al. [26], since they only computed the time for building and updating the FUFP tree when the new

transactions were inserted in their experiments. In our experiments, we compute the time for mining all FIs of the two algorithms. The first 900 000 transactions were extracted from the Kosarak database for offline mining. Each next 1 000 transactions were then sequentially used each time as new transactions for the experiments. The upper and the lower support thresholds were set at 1 % and 2 %, respectively. Figure 14 shows execution times required by the two algorithms for processing each inserted 1 000 transactions. It can be seen that the proposed Pre-FUT maintenance algorithm ran faster than the pre-large-itemset algorithm.

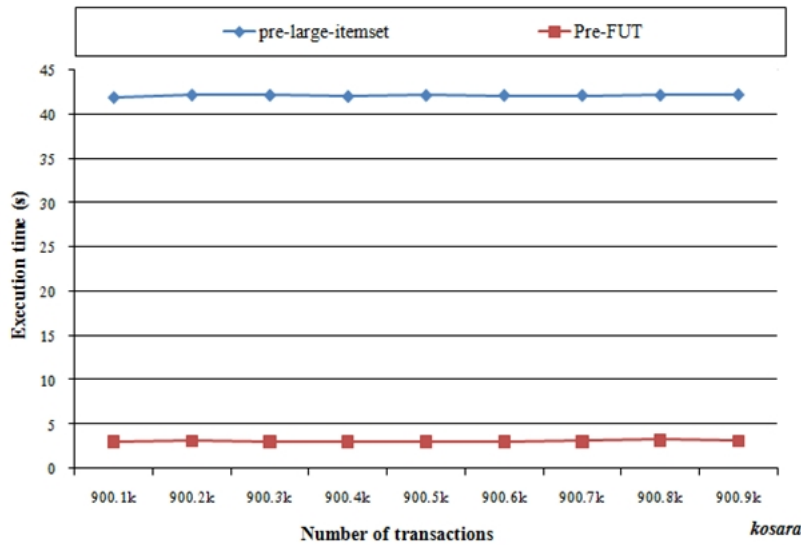

Figure 14. Comparison of the execution times for sequentially inserted transactions for Kosarak database

Next, the T40I10D100K database was used for the experiments. The first 90 000 transactions were extracted from the T40I10D100K database for offline mining. Each next 1 000 transactions were then sequentially used each time as new transactions for the experiments. The upper and the lower support thresholds were set at 5 % and 2 %, respectively. As mentioned above, when the number of inserted transactions reached the safety number, the original database was processed again. In the experiments with the T40I10D100K database, the safety number was calculated as $f = 90\,000 * (0.05 - 0.02)/(1 - 0.05) = 2\,842$. Figure 15 shows execution times required by the two algorithms for processing each 1 000 inserted transactions. It can be seen from the figure that the Pre-FUT maintenance algorithm ran faster than the pre-large-itemset algorithm, even in the case when the number of inserted transactions exceeded the safety threshold.

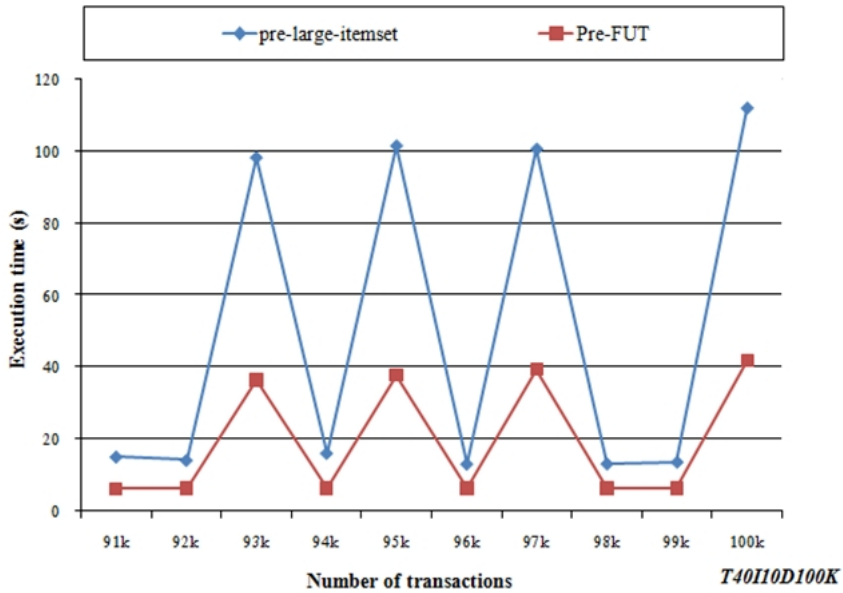

Figure 15. Comparison of execution times for sequentially inserted transactions for T40I10D100K database

The shapes of Figures 14 and 15 are different. In the first experiment with the Kosarak database, the safety threshold $f = 90\,0000 * (0.02 - 0.01)/(1 - 0.02) = 9\,183$ and 1 000 transactions will be inserted each time. The algorithm thus did not need to rescan the original database until the tenth batch of inserted new transactions. In the previous nine batches of inserted new transactions, each time the algorithm only needed to process the 1 000 inserted transactions to update the total counts of itemsets. Figure 14 shows comparative times of processing 1 000 new transactions each time in nine batches of inserted new transactions. In the second experiment with the T40I10D100K database, the safety threshold $f = 2\,842$. Because only 1 000 transactions were inserted each time, the algorithm needs to rescan the original database when the third batch of inserted new transactions arrive. The high points in Figure 15 are the cases when the algorithm had to rescan the original database. Processing the whole database would require more time than when processing only the inserted transactions. In real applications, the number of new inserted transactions is usually smaller than that in the original database, and thus the proposed approach is efficient.

6 CONCLUSION AND FUTURE WORK

This paper has adopted an alternative data structure for incremental data mining. The Pre-FUT algorithm has been proposed, which is based on the concept of pre-

large itemsets to reduce the number of database scans. It uses two user-specified upper and lower support thresholds to avoid small items becoming large in the updated database when transactions are inserted. All the tasks are processed and stored in a trie. With these strategies, the proposed approach can thus spend less execution time than the pre-large-itemset algorithm which is coded by the hash-tree data structure.

Hong et al. recently developed some algorithms for deleting and modifying transactions based on the pre-large itemsets concept [16, 17]. The results of our coding show that the task of inserting or deleting itemsets using the trie data structure can be processed very quickly and effectively. Moreover, we can mine FIs directly from the tree-building process by using the trie data structure. Therefore, in future, we will also extend our approach to mining tasks with deleted and modified transactions using the trie structure and the pre-large itemsets concept.

Acknowledgement

This work was supported by the Yeungnam University research grants in 2012.

REFERENCES

[1] Agrawal, R.—Imielinski, T.—Swami, A. N.: Mining Association Rules Between Sets of Items in Large Database. In SIGMOD93 Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, New York, USA, pp. 207–216.

[2] Agrawal, R.—Imielinski, T.—Swami, A. N.: Database Mining – A Performance Perspective. IEEE Transactions on Knowledge and Data Engineering, Vol. 5, 1993, No. 6, pp. 914–925.

[3] Agrawal, R.—Srikant, R.: Fast Algorithm for Mining Association Rules. In Proceedings of the International Conference on Very Large Data Bases, Santiago de Chile 1994, pp. 487–499.

[4] Agrawal, R.—Srikant, R.: Mining Sequential Patterns. In Proceedings of the Eleventh International Conference on Data Engineering, Taipei, Taiwan 1995, pp. 3–14.

[5] Agrawal, R.—Srikant, R.—Vu, Q.: Mining Association Rules with Item Constraints. In KDD97 Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, Newport Beach, California, USA 1997, pp. 67–73.

[6] Bodon, F.—Ronyai, L.: Trie – An Alternative Data Structure for Data Mining Algorithms. Mathematical and Computer Modeling, Vol. 38, 2003, No. 7-9, pp. 739–751.

[7] Chen, M. S.—Park, J. S.—Yu, P. S.: An Effective Hash Based Algorithm for Mining Association Rules. In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, NewYork, USA 1995, pp. 175–186.

[8] Chen, M. S.—Han, J.—Yu, S. P.: Data Mining – An Overview from a Database Perspective. IEEE Transactions on Knowledge and Data Engineering, Vol. 8, 1996, No. 6, pp. 866–883.

[9] Cheung, D. W.—Han, J.—Ng, V. T.—Wong, C. Y.: Maintenance of Discovered Association Rules in Large Databases – An Incremental Updating Approach. In Proceedings of the Twelfth IEEE International Conference on Data Engineering, New Orleans, Louisiana 1966, pp. 106–114.

[10] Cheung, D. W.—Lee, S. D.—Kao, B.: A General Incremental Technique for Maintaining Discovered Association Rules. In DASFAA97 Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, Melbourne, Australia 1997, pp. 185–194.

[11] Fukuda, T.—Morimoto, Y.—Morishita, S.—Tokuyama, T.: Mining Optimized Association Rules for Numeric Attributes. In Proceedings of the Fifteenth ACM SIGACTSIGMOD-SIGART Symposium on Principles of Database Systems, New York, USA 1996, pp. 182–191.

[12] Han, J.—Fu, Y.: Discovery of Multiple-Level Association Rules from Large Database. In Proceedings of the 21th International Conference on Very Large Data Bases, San Francisco, California 1995, pp. 420–431.

[13] Han, J.—Pei, J.—Yin, Y.: Mining Frequent Patterns without Candidate Generation. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas 2000, pp. 1–12.

[14] Hong, T. P.—Wang, C. Y.—Tao, Y. H.: A New Incremental Data Mining Algorithm Using pre-Large Itemsets. Intelligent Data Analysis, Vol. 5, 2001, No. 2, pp. 111–129.

[15] Hong, T. P.—Lin, C. W.—Wu, Y. L.: Incrementally Fast Updated Frequent Pattern Trees. Expert Systems with Applications, Vol. 34, 2001, No. 4, pp. 2424–2435.

[16] Hong, T. P.—Lin, C. W.—Wu, Y. L.: Maintenance of Fast Updated Frequent Pattern Trees for Record Deletion. Computational Statistics and Data Analysis, Vol. 53, 2009, No. 7, pp. 2485–2499.

[17] Hong, T. P.—Wang, C. Y.: An Efficient and Effective Association-Rule Maintenance Algorithm for Record Modification. Expert Systems with Applications, Vol. 37, 2010, No 1, pp. 618–626.

[18] Hong, T. P.—Wang, C. Y.—Tseng, S. S.: An Incremental Mining Algorithm for Maintaining Sequential Patterns Using pre-Large Sequences. Expert Systems with Applications, Vol. 38, 2011, No. 6, pp. 7051–7058.

[19] Jung, J. J.: An Evolutionary Approach to Query-Sampling for Heterogeneous Systems. Expert Systems with Applications, Vol. 37, 2010, No. 1, pp. 226–232.

[20] Jung, J. J.: Semantic Preprocessing for Mining Sensor Streams from Heterogeneous Environments. Expert Systems with Applications, Vol. 38, 2011, No. 5, pp. 6107–6111.

[21] Jung, J. J.: Constraint Graph-Based Frequent Pattern Updating from Temporal Databases. Expert Systems with Applications, Vol. 39, 2012, No. 3, pp. 3169–3173.

[22] Jung, J. J.: Semantic Annotation of Cognitive map for Knowledge Sharing Between Heterogeneous Businesses. Expert Systems with Applications, Vol. 39, 2012, No. 5, pp. 5857–5860.

[23] Jung, J. J.: Discovering Community of Lingual Practice for Matching Multilingual Tags from Folksonomies. Computer Journal, Vol. 55, 2012, No. 3, pp. 337–346.

[24] Koh, J. L.—Shied, S. F.: An Efficient Approach for Maintaining Association Rules Based on Adjusting FP-Tree Structures. In Proceedings of Database Systems for Advanced Applications, Jeju Island, Korea 2004, pp. 417–424.

[25] Knuth, D. E.: The Art of Computer Programming. Addison-Wesley 1968.

[26] Lin, C. W.—Hong, T. P.—Lu, W. H.: The Pre-FUFP Algorithm for Incremental Mining. Expert Systems with Applications, Vol. 36, 2009, No. 5, pp. 9498–9505.

[27] Lin, X.—Deng, Z.H.—Tang, S.: A Fast Algorithm for Maintenance of Associations Rules in Incremental Databases. In Proceedings of the Second International Conference on Advanced Data Mining and Applications, XiAn, China 2006, pp. 56–63.

[28] Mannila, H.—Toivonen, H.—Verkamo, A. I.: Efficient Algorithm for Discovering Association Rules. In AAAI Workshop on Knowledge Discovery in Databases 1994, pp. 181–192.

[29] Park, J. S.—Chen, M. S.—Yu, P. S.: Using a Hash-Based Method with Transaction Trimming for Mining Association Rules. IEEE Transactions on Knowledge and Data Engineering, Vol. 9, 1997, No. 5, pp. 812–825.

[30] Srikant, R.—Agrawal, R.: Mining Generalized Association Rules. In VLDB95 Proceedings of the 21th International Conference on Very Large Data Bases, San Francisco, California 1995, pp. 407–419.

[31] Srikant, R.—Agrawal, R.: Mining Quantitative Association Rules in Large Relational Tables. In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, New York, USA, pp. 1–12.

[32] Thomas, S.—Bodagala, S.—Alsabti, K.—Ranka, S.: An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases. In Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, Newport Beach, California 1997, pp. 263–266.

[33] Toivonen, H.: Sampling Large Databases for Association Rules. In Proceedings of the 22th International Conference on Very Large Data Bases, San Francisco, California 1996, pp. 134–145.

[34] Wang, C. Y.—Hong, T. P.—Tseng, S. S.: Maintenance of Sequential Patterns for Record Deletion. In Proceedings of the IEEE International Conference on Data Mining, San Jose, California 2001, pp. 536–541.

[35] Wang, C. Y.—Hong, T. P.—Tseng, S. S.: Maintenance of Sequential Patterns for Record Modification Using Pre-Large Sequences. In Proceedings of the IEEE Conference on Data Mining, Maebashi City, Japan 2002, pp. 693–696.

Thien-Phuong Le received the B. Sc. degree (2007) from Nha Trang University, the M. Sc. degree (2010) from University of Science, Ho Chi Minh City, Viet Nam. His research interests include artificial intelligence, soft computing, knowledge discovery and data mining.

Bay Vo received his Ph. D. degree in computer science from the University of Science, Vietnam National University of Ho Chi Minh, Vietnam in 2011. His research interests include association rules, classification, mining in incremental database, distributed databases and privacy preserving in data mining.

Tzung-Pei Hong received his Ph. D. degree in computer science and information engineering from National Chiao-Tung University in 1992. He served as the first director of the library and computer center, the Dean of Academic Affairs and the Vice President in National University of Kaohsiung. He is currently a distinguished professor at the Department of Computer Science and Information Engineering in NUK. He has published more than 400 research papers in international/national journals and conferences and has planned more than fifty information systems. He is also the board member of more than thirty journals and the program committee member of more than two hundred conferences. His current research interests include parallel processing, machine learning, data mining, soft computing, management information systems, and www applications.

Bac Le received the B. Sc. degree in 1984, the M. Sc. degree in 1990, and the Ph. D. degree in computer science in 1999. He is an Associate Professor, Vice Dean of Faculty of Information Technology, Head of Department of Computer Science, University of Science, Ho Chi Minh City. His research interests include artificial intelligence, soft computing, knowledge discovery and data mining.

Dosam Hwang is the corresponding author of this paper. He received his Ph. D. degree in natural language processing at Kyoto University. In 1987, he was chosen the best researcher at Korea Institute of Science and Technology. Since 2005 he has been the Professor at Yeungnam University in Korea. Currently he is working on natural language processing, machine translation, ontology, semantic web, and information retrieval. He has served for a number of international conferences and a technical committee for ISO.