Efficient Tree Structures for High Utility Pattern Mining in Incremental Databases

Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer,
Byeong-Soo Jeong, and Young-Koo Lee, *Member*, *IEEE*

Abstract—Recently, high utility pattern (HUP) mining is one of the most important research issues in data mining due to its ability to consider the nonbinary frequency values of items in transactions and different profit values for every item. On the other hand, incremental and interactive data mining provide the ability to use previous data structures and mining results in order to reduce unnecessary calculations when a database is updated, or when the minimum threshold is changed. In this paper, we propose three novel tree structures to efficiently perform incremental and interactive HUP mining. The first tree structure, Incremental HUP Lexicographic Tree (IHUP_{L}-Tree), is arranged according to an item's lexicographic order. It can capture the incremental data without any restructuring operation. The second tree structure is the IHUP Transaction Frequency Tree (IHUP_{TF}-Tree), which obtains a compact size by arranging items according to their transaction frequency (descending order). To reduce the mining time, the third tree, IHUP-Transaction-Weighted Utilization Tree (IHUP_{TWU}-Tree) is designed based on the TWU value of items in descending order. Extensive performance analyses show that our tree structures are very efficient and scalable for incremental and interactive HUP mining.

Index Terms—Data mining, frequent pattern mining, high utility pattern mining, incremental mining, interactive mining.

1 Introduction

The initial solution of frequent pattern mining, candidate set generation-and-test paradigm of Apriori [1], [2], has revealed many drawbacks including that it requires multiple database scans and generates many candidate itemsets. FP-growth [12] solved this problem by introducing a prefix-tree (FP-tree)-based algorithm without candidate set generation-and-testing. Although frequent pattern mining [1], [2], [12], [13], [23], [24], [36] plays an important role in data mining applications, its two limitations are, first, it treats all items with the same importance/weight/price and, second, in one transaction, each item appears in a binary (0/1) form, i.e., either present or absent. However, in the real world, each item in the supermarket has a different importance/price and one customer can buy multiple copies of an item. Moreover, items having high and low selling frequencies may have low and high profit values, respectively. For example, some frequently sold items such as bread, milk, and pen may have lower profit values compared to that of infrequently sold higher profit value items such as gold ring and gold necklace. Therefore, finding only traditional frequent patterns in a database cannot fulfill the requirement of finding the most valuable itemsets/customers that contribute to the major part of the total profits in a retail business. This gives the motivation to develop a mining model to discover the itemsets/customers contributing to the majority of the profit.

Recently, a utility mining [5], [6], [7], [8] model was defined to discover more important knowledge from a database. We can measure the importance of an itemset by the concept of utility. We can handle the dataset with nonbinary frequency values of each item in transactions, and also with different profit values of each item. Therefore, utility mining represents real world market data. By utility mining, several important business area decisions like maximizing revenue or minimizing marketing or inventory costs can be considered and knowledge about itemsets/customers contributing to the majority of the profit can be discovered. In addition to our real world retail market, if we consider the biological gene database and Web click streams, then the importance of each gene or Web site is different and their occurrences are not limited to a 0/1 value. Other application areas, such as stock tickers, network traffic measurements, Web server logs, data feeds from sensor networks, and telecom call records can have similar solutions.

The previous works [5], [6], [7], [8], [10], [26], [27], [35] in this area are based on a fixed database and did not consider that one or more transactions could be deleted, inserted, or modified in the database. By using incremental and interactive high utility pattern (HUP) mining, we can use the previous data structures and mining results, and avoid unnecessary calculations when the database is updated or the mining threshold is changed. To understand the necessity of today's incremental databases, where additions, deletions, and modifications are very frequent operations,

- *C.F. Ahmed, S.K. Tanbeer, and B.-S. Jeong are with the Database Laboratory, Department of Computer Engineering, Kyung Hee University, 1 Seochun-dong, Kihung-gu, Youngin-si, Kyunggi-do 446-701, Republic of Korea. E-mail: {farhan, tanbeer, jeong}@khu.ac.kr.*
- *Y.-K. Lee is with the Data and Knowledge Engineering Laboratory, Department of Computer Engineering, Kyung Hee University, 1 Seochun-dong, Kihung-gu, Youngin-si, Kyunggi-do 446-701, Republic of Korea. E-mail: yklee@khu.ac.kr.*

Manuscript received 23 Aug. 2008; revised 7 Jan. 2009; accepted 30 Jan. 2009; published online 6 Feb. 2009.
Recommended for acceptance by L. Wang.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2008-08-0436.
Digital Object Identifier no. 10.1109/TKDE.2009.46.

we can consider a small example in the real world market. Consider customer X has bought three pens, four pencils, and one eraser and customer Y has bought one computer mouse. After sometime, customer Z may come to buy two breads and one milk, customer X may come and return two pencils, and customer Y may come to return the mouse. So, new transactions in the real world market can be frequently added, and the old transactions can be modified or deleted. As a result, we have to consider additions, deletions, and modifications of the transactions in our real world datasets.

Moreover, if the algorithms presented in the previous works want to calculate which patterns cover 20 percent of the total profit, then their internal data structures are designed in such a way that they can only calculate the asked amount. If the amount is 15 percent of the total profit, then previous algorithms have to build their data structures again. They cannot take any advantages from their previous design. However, in the real world, the businessmen need to repeatedly change the minimum threshold according to their business requirements. Thus, the *"build once mine many"* property (by building the data structure only once, several mining operations can be done) is required to solve this interactive mining problem.

Motivated from these real world scenarios, we propose efficient solutions to these major problems of the existing works. In this paper, we propose three new tree structures with the *"build once mine many"* property for high utility pattern mining in an incremental database. They exploit a pattern growth approach to avoid the problem of level-wise candidate generation-and-test strategy. We design the first incremental tree structure, Incremental High Utility Pattern Lexicographic Tree (IHUP_{L}-Tree), according to the item's lexicographic order. This tree can capture the incremental data without any restructuring operation, but the tree size cannot be guaranteed to be compact. The second incremental tree structure, Incremental High Utility Pattern Transaction Frequency Tree (IHUP_{TF}-Tree), is designed according to the transaction frequency (descending order) of items to obtain a compact tree. However, it is not guaranteed that the items having higher transaction frequency will also have a high utility value. If several low-utility items appear in the upper portion of the tree, then the mining time will be larger. To ensure the candidates of high utility items will appear before low-utility items in any branch of the tree, the third incremental tree structure, Incremental High Utility Pattern-Transaction-Weighted Utilization Tree (IHUP_{TWU}-Tree) is designed according to transaction-weighted utilization of items in descending order, which ensures that only nodes containing candidate items are participating in the mining operation. As a result, although the IHUP_{TF}-Tree requires the smallest amount of memory, the IHUP_{TWU}-Tree requires the smallest amount of overall running time due to its large time savings achieved during the mining phase. Moreover, the *"build once mine many"* property of the tree structures is very effective in interactive mining. Extensive performance analyses show that our proposed tree structures are very efficient and effective in interactive and incremental HUP mining, and they outperform the existing methods.

The remainder of this paper is organized as follows: In Section 2, we describe related studies. In Section 3, we describe the high utility pattern mining problem. In Section 4, we describe our proposed three tree structures for high utility pattern mining and show how they can handle additions, deletions, and modifications of transactions. Here, we also develop the tree construction and mining algorithm. In Section 5, mining performances of these tree structures are discussed. In Section 6, we describe the efficiency of our tree structures in incremental mining and show how the *"build once mine many"* property of our tree structures can be effective in interactive mining. In Section 7, our experimental results are presented and analyzed. Finally, in Section 8, conclusions are drawn.

2 Related Work

In the following sections, we will discuss research on frequent pattern mining, weighted frequent pattern mining, high utility pattern mining, and incremental and interactive pattern mining.

2.1 Frequent Pattern Mining

Let $I = \{i_1, i_2, \ldots\ldots i_m\}$ be a set of items and D be a transaction database $\{T_1, T_2, \ldots\ldots T_n\}$, where each transaction $T_i \in D$ is a subset of I. The support/frequency of a pattern $X\{x_1, x_2, \ldots\ldots x_p\}$ is the number of transactions containing the pattern in the transaction database. The problem of frequent pattern mining is to find the complete set of patterns satisfying a minimum support in the transaction database. The *downward closure* property [1], [2] is used to prune the infrequent patterns. This property states that if a pattern is infrequent, then all of its superpatterns must also be infrequent. The Apriori [1], [2] algorithm is the initial solution for the frequent pattern mining problem, but it suffers from the level-wise candidate generation-and-test problem and requires several database scans. For the first database scan, Apriori finds all one-element frequent itemsets, and based on that, generates the candidates for two-element frequent itemsets. In the second database scan, Apriori finds all of the two-element frequent itemsets, and based on that, generates the candidates for three-element frequent itemsets and so on.

FP-growth [12] solved this problem by using an FP-tree based solution without any candidate generation and using only two database scans. FP-array [13] technique was proposed to reduce the FP-tree traversals and it efficiently works especially in sparse datasets. One interesting measure h-confidence [22] was proposed to identify strong support affinity frequent patterns. Some other research [14], [15], [23], [24] has been done for frequent pattern mining. This traditional frequent pattern mining considers an equal profit/weight for all items and only binary occurrence (0/1) of the items in one transaction.

2.2 Weighted Frequent Pattern Mining

Research has been done for weighted frequent pattern mining [3], [4], [11], [20], [21] in binary databases where frequency of an item in each transaction can be either 1 or 0. The weight of a pattern P is the ratio of the sum of all its items' weight value to the length of P. Weighted frequent itemset mining (WFIM) [3] and Weighted interesting

patterns mining (WIP) [4] showed that the main challenge in this area is the weighted frequency of a pattern does not have the *downward closure* property. Consider item "*a*" has a weight of 0.6 and a frequency of 4, item "*b*" has a weight of 0.2 and a frequency of 5, and itemset "*ab*" has frequency 3. Then, the weight of itemset "*ab*" will be $(0.6 + 0.2)/2 = 0.4$ and its weighted frequency will be $0.4 \times 3 = 1.2$. The weighted frequency of "*a*" is $0.6 \times 4 = 2.4$ and that of "*b*" is $0.2 \times 5 = 1.0$. If the minimum weighted frequency threshold is 1.2, then pattern "*b*" is weighted infrequent, but its superpattern "*ab*" is weighted frequent. As a result, the *downward closure* property is not satisfied here. WFIM [3] and WIP [4] maintain the *downward closure* property by multiplying each itemset's frequency by the global maximum weight. In the above example, if "*a*" has the maximum weight of 0.6, then by multiplying it with the frequency of item "*b*," 3.0 is obtained. Therefore, pattern "*b*" is not pruned at the early stage and pattern "*ab*" will not be missed. However, pattern "*b*" is overestimated, and will be pruned later by using its actual weighted frequency. Maintaining *downward closure* property in high utility pattern mining is more challenging as it considers nonbinary frequency values of an item in transactions.

2.3 High Utility Pattern Mining

The Itemset Share approach [9] considers multiple frequencies of an item in each transaction. Share is the percentage of a numerical total that is contributed by the items in an itemset. The authors [9] define the problem of finding share-frequent itemsets and compare the share and support measures to illustrate that the share measure approach can provide useful information about the numerical values that are associated with transaction items, which is not possible using only the support measure. This method cannot rely on the *downward closure* property. The authors developed heuristic methods to find itemsets with share values above the minimum share threshold. Mining high utility itemsets [10] developed top-K objective-directed high utility closed patterns. The authors' definitions are different from our work. They assume the same medical treatment for different patients (different transactions) will have different levels of effectiveness. They cannot maintain the *downward closure* property but they develop a pruning strategy to prune low-utility itemsets based on a weaker *antimonotonic* condition.

The theoretical model and definitions of high utility pattern mining were given in [5]. This approach, called mining with expected utility (MEU), cannot maintain the *downward closure* property of Apriori and the authors of [5] used a heuristic to determine whether an itemset should be considered as a candidate itemset. Also, MEU usually overestimates, especially at the beginning stages, where the number of candidates approaches the number of all the combinations of items. This trait is impractical whenever the number of distinct items is large and the utility threshold is low. Later, the same authors proposed two new algorithms, UMining and UMining_H [6], to calculate the high utility patterns. In UMining, a pruning strategy based on utility upper bound property is used. UMining_H has been designed with another pruning strategy based on a heuristic method. However, some high utility itemsets may be erroneously pruned by their heuristic method. Moreover, these methods do not satisfy the *downward closure* property of Apriori, and therefore, overestimate too many patterns. They also suffer from excessive candidate generations and poor test methodology.

The Two-Phase [7], [8] algorithm was developed based on the definitions of [5] to find high utility itemsets using the *downward closure* property of Apriori. The authors have defined the *transaction-weighted utilization (twu)* and by that they proved it is possible to maintain the *downward closure* property. For the first database scan, the algorithm finds all the one-element transaction-weighted utilization itemsets, and based on that result, it generates the candidates for two-element transaction-weighted utilization itemsets. In the second database scan, it finds all the two-element transaction-weighted utilization itemsets, and based on that result, it generates the candidates for three-element transaction-weighted utilization itemsets, and so on. At the last scan, the Two-Phase algorithm determines the actual high utility itemsets from the high transaction-weighted utilization itemsets. This algorithm suffers from the same problem of the level-wise candidate generation-and-test methodology. CTU-Mine [26] proposed an algorithm that is more efficient than the Two-Phase method only in dense databases when the minimum utility threshold is very low. Another algorithm presents an approximation to solve the high utility pattern mining problem through specialized partition trees, called high-yield partition trees [27].

The isolated items discarding strategy (IIDS) [35] for discovering high utility itemsets was proposed to reduce the number of candidates in every database scan. IIDS shows that itemset share mining [9] problem can be directly converted to the utility mining problem by replacing the frequency value of each item in a transaction by its total profit, i.e., multiplying the frequency value by its unit profit. Applying IIDS, the authors developed efficient high utility itemset mining algorithms called FUM and DCG+ and showed that their technique is better than all previous high utility pattern mining techniques. However, their algorithms still suffer level-wise candidate set generation-and-test problem of Apriori, and require multiple database scans.

2.4 Incremental and Interactive Mining

Research [16], [17], [18], [19], [34], [36] has been done to develop techniques for incremental and interactive mining in the area of traditional frequent pattern mining, and they have shown that incremental prefix-tree structures such as CanTree [19], CP-tree [36], FUFP-tree [34], etc., are quite possible and efficient using currently available memory in the gigabyte range. The efficient dynamic database updating algorithm (EDUA) [32] is designed for mining databases when data deletion is performed frequently in any database subset. IncWTP and WssWTP algorithms [33] are designed for incremental and interactive mining of Web traversal patterns. However, these solutions are not applicable for incremental and interactive high utility pattern mining.

In the existing high utility pattern mining works, no one has proposed a solution for incremental mining, where many new transactions can be added, and existing transactions can be deleted/modified. Moreover, none of the data structures have the *"build once mine many"* property. Therefore, we propose new tree structures for incremental high utility pattern mining techniques containing the *"build once*

	TID \ ITEM	a	b	c	d	e	Trans. Utility
Original DB	T_1	2	2	0	0	0	34
	T_2	3	0	12	4	2	88
	T_3	0	0	15	0	3	66
	T_4	4	0	0	0	0	8
db_1^+	T_5	0	10	0	8	9	277
	T_6	0	7	3	0	4	142
db_2^+	T_7	1	0	2	0	1	15
	T_8	2	0	0	1	3	33

(a)

ITEM	PROFIT($) (per unit)
a	2
b	15
c	3
d	8
e	7

(b)

Fig. 1. Example of a (a) transaction database and (b) utility table.

mine many" property. Our algorithm maintains the *downward closure* property with a transaction-weighted utilization value of a pattern, and uses FP-growth mining operation to avoid level-wise candidate generation-and-test problem.

3 Problem Definition

We have adopted definitions similar to those presented in the previous works [5], [6], [7], [8]. Let $I = \{i_1, i_2, \ldots\ldots i_m\}$ be a set of items and D be a transaction database $\{T_1, T_2, \ldots\ldots T_n\}$, where each transaction $T_i \in D$ is a subset of I.

Definition 1. *The internal utility or local transaction utility value $l(i_p, T_q)$, represents the quantity of item i_p in transaction T_q. For example, in Fig. 1a, $l(c, T_2) = 12$.*

Definition 2. *The external utility $p(i_p)$ is the unit profit value of item i_p. For example, in Fig. 1b, $p(c) = 3$.*

Definition 3. *Utility $u(i_p, T_q)$ is the quantitative measure of utility for item i_p in transaction T_q defined by*

$$u(i_p, T_q) = l(i_p, T_q) \times p(i_p). \tag{1}$$

For example, $u(c, T_2) = 12 \times 3 = 36$ in Fig. 1.

Definition 4. *The utility of an itemset X in transaction T_q, $u(X, T_q)$, is defined by*

$$u(X, T_q) = \sum_{i_p \in X} u(i_p, T_q), \tag{2}$$

where $X = \{i_1, i_2, \ldots\ldots i_k\}$ is a k-itemset, $X \subseteq T_q$, and $1 \leq k \leq m$. For example, $u(ac, T_2) = 3 \times 2 + 12 \times 3 = 42$ in Fig. 1.

Definition 5. *The utility of an itemset X is defined by*

$$u(X) = \sum_{T_q \in D} \sum_{i_p \in X} u(i_p, T_q). \tag{3}$$

For example, $u(ac) = u(ac, T_2) + u(ac, T_7) = 42 + 8 = 50$ in Fig. 1.

Definition 6. *The transaction utility (tu) of transaction T_q denoted as $tu(T_q)$ describes the total profit of that transaction and is defined by*

$$tu(T_q) = \sum_{i_p \in T_q} u(i_p, T_q). \tag{4}$$

For example, $tu(T_1) = u(a, T_1) + u(b, T_1) = 4 + 30 = 34$ in Fig. 1.

Definition 7. *The minimum utility threshold δ is given by the percentage of total transaction utility values of the database. In Fig. 1, the summation of all the transaction utility values is 663. If δ is 30 or we can also express it as 0.3, then the minimum utility value can be defined as*

$$minutil = \delta \times \sum_{T_q \in D} tu(T_q). \tag{5}$$

Therefore, in this example, $minutil = 0.3 \times 663 = 198.9$ in Fig. 1.

Definition 8. *An itemset X is a high utility itemset if $u(X) \geq minutil$. Finding high utility itemsets means determining all itemsets X having criteria $u(X) \geq minutil$.*

The main challenge of facing high utility pattern mining areas is the itemset utility does not have the *downward closure* property. For example, if $minutil = 198.9$ in Fig. 1, then "*d*" is a low-utility item, as shown by $u(d) = 104$. However, "*de*" is a high utility itemset, as shown by $u(de) = 202$ according to Definition 8. We can maintain the *downward closure property* by transaction-weighted utilization.

Definition 9. *Transaction-weighted utilization of an itemset X, denoted by $twu(X)$, is the sum of the transaction utilities of all transactions containing X*

$$twu(X) = \sum_{X \subseteq T_q \in D} tu(T_q). \tag{6}$$

For example, $twu(ac) = tu(T_2) + tu(T_7) = 88 + 15 = 103$ in Fig. 1. The downward closure property can be maintained using transaction-weighted utilization. Here, for $minutil = 198.9$ in Fig. 1 as $twu(ac) < minutil$, any superpattern of "ac" cannot be a high-twu itemset (candidate itemset) and obviously cannot be a high utility itemset.

Definition 10. *X is a high transaction-weighted utilization itemset (i.e., a candidate itemset) if $twu(X) \geq minutil$.*

Definition 11. *The transaction frequency (tf) of an item i_p is $tf(i_p)$ and represents the number of transactions in which the item appears. The original frequency of i_p is $f(i_p)$, which denotes the actual number of occurrences of i_p in those transactions. For example, $tf(c) = 4$ as it appears in T_2, T_3, T_6, and T_7 and $f(c) = 12 + 15 + 3 + 2 = 32$.*

The transaction frequency of an itemset X is needed in our algorithm to speed up the second database scan to find high utility itemsets from high transaction-weighted utilization itemsets. For example, $twu(ce) = 311$, and $tf(ce) = 4$. After scanning T_7, we know that the "*ce*" pattern has been found four times, which is equal to its transaction frequency, and up to now $u(ce) = 166$. Thus, we can say that "*ce*" is a low-utility itemset for $\delta = 30\%$ and we do not need to consider "*ce*" in the rest of the database. So, we make the final decision for many patterns before the end of the database by using *tf*.

4 Our Proposed Tree Structures

In this section, at first, we will describe the construction process of the IHUP_L-Tree where items are arranged in lexicographic order. Subsequently, we will show how the IHUP_TF-Tree and IHUP_TWU-Tree structures can be constructed. While constructing, we will show how additions, deletions, and modifications are possible inside the tree

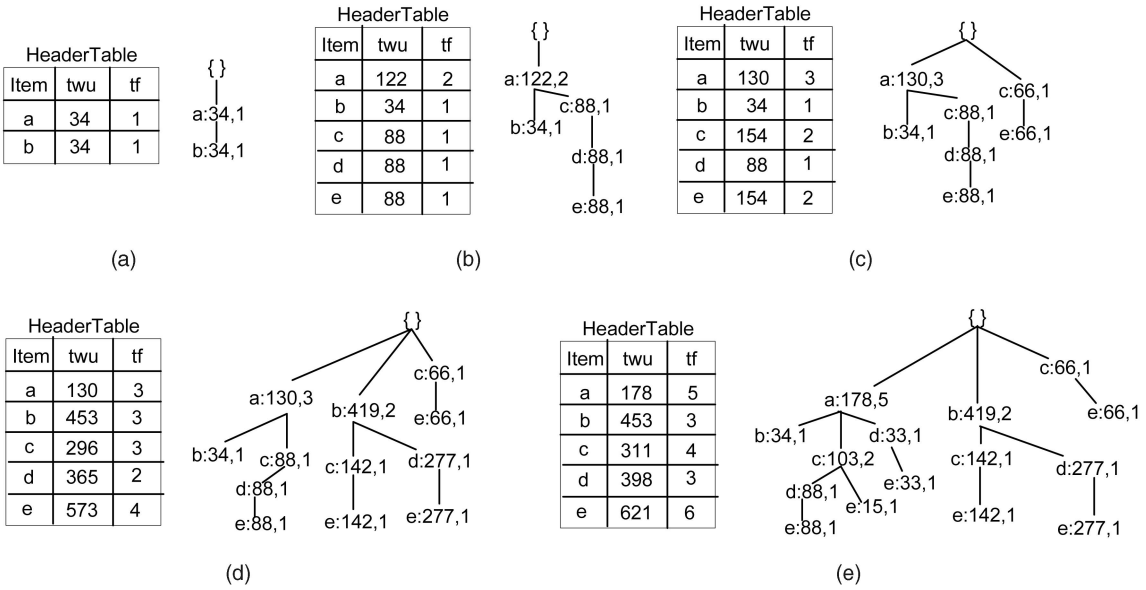

Fig. 2. Construction of the $\mathrm{IHUP_L}$-Tree (a) after inserting $\mathrm{T_1}$, (b) after inserting $\mathrm{T_2}$ (c) after inserting $\mathrm{T_3}$ and $\mathrm{T_4}$, (d) after inserting $\mathrm{db_1^+}$ ($\mathrm{T_5}$ and $\mathrm{T_6}$), and (e) after inserting $\mathrm{db_2^+}$ ($\mathrm{T_7}$ and $\mathrm{T_8}$).

structures. Finally, we describe our tree construction and mining algorithm. We will use the term IHUP-Tree to denote all three tree structures.

4.1 $\mathrm{IHUP_L}$-Tree Construction

In the first database scan, $\mathrm{IHUP_L}$-Tree arranges the items in lexicographic order and inserts them as a branch inside the tree. All the IHUP-Trees explicitly maintain *twu* and *tf* values in both the header table and the tree nodes. To facilitate the tree traversals, adjacent links are also maintained (not shown in the figures for simplicity).

Consider the transaction database and utility table in Fig. 1. From the very beginning, the database contains only four transactions. After that, the initial database is incremented by adding two groups of transactions. The construction process for the initial database (T_1-T_4) of the $\mathrm{IHUP_L}$-Tree is described in Figs. 2a, 2b, and 2c. The *tu* value of T_1 is 34 in Fig. 1. Fig. 2a shows that when T_1 is added to the tree, the first node in lexicographic order is "*a*" with $twu = 34$ and $tf = 1$. The second node is "*b*" with the same values. Fig. 2b shows when T_2 is added to the tree. The tu of T_2 is 88, so the branch in lexicographical order is "*a:88, 1,*" "*c:88, 1,*" "*d:88, 1,*" and "*e:88, 1.*" Here "*a*" is assigned the prefix-sharing with the existing node "*a,*" the *twu* value of node "*a*" is $34 + 88 = 122$, and the *tf* value is $1 + 1 = 2$. For other items, new nodes are created, as shown in Fig. 2b. After that, T_3 and T_4 are added to complete the initial database (Fig. 2c). By following the same process, db_1^+ (T_5 and T_6) and db_2^+ (T_7 and T_8) are inserted into the $\mathrm{IHUP_L}$-Tree, as shown in Figs. 2d and 2e.

We can also easily perform deletion and modification operations in the $\mathrm{IHUP_L}$-Tree. Suppose that we want to delete T_7, then the *tu* value of T_7 and one *tf* value can be easily decreased from the path "*a c e*" in the tree shown in Fig. 2e. Subsequently, as *twu* and *tf* values of node "*e*" in that path become zero, we have to delete node "*e*" from that branch. So, the modified *twu* and *tf* values of "*a*" will be 163 and 4, respectively, in that branch. In the same way, the modified *twu* and *tf* values of "*c*" will be 88 and 1, respectively, in that branch. Suppose that we do not want to delete T_7 but simply modify it. We want to reduce the quantity of items "*c*" from two to one. To perform this modification, the *twu* value of T_7 will be reduced by $1 \times 3 = 3$. We have to reduce amount three (*twu* value) from that branch "*a c e*."

As $\mathrm{IHUP_L}$-Tree always maintains the lexicographic order in both the header table and tree nodes, the following property is always true for it:

Property 1. *The ordering of items in the* $\mathrm{IHUP_L}$*-Tree is not affected in spite of changing the frequency of the items by additions, deletions, and modifications.*

4.2 $\mathrm{IHUP_{TF}}$-Tree Construction

To reduce the size of the $\mathrm{IHUP_L}$-Tree, we have to increase the prefix-sharing inside it. We develop the $\mathrm{IHUP_{TF}}$-Tree in order to achieve the desired compactness. In this tree, the nodes are arranged in descending order according to their transaction frequency so that items occurring in many transactions can be kept in the upper part of the tree, and therefore, higher prefix-sharing can be achieved. The $\mathrm{IHUP_{TF}}$-Tree can be constructed from an $\mathrm{IHUP_L}$-Tree using a path adjusting method [17] based on a bubble sort technique at any time. Any node may be split when it needs to be swapped with any child node having a count smaller than that node. If the support counts of both nodes are equal, a simple exchange operation between them is performed. After performing each operation, swapping nodes having the same items are merged.

Consider the transaction database and utility table in Fig. 1. In the original database (up to T_4), the $\mathrm{IHUP_{TF}}$-Tree is constructed in the same way as the $\mathrm{IHUP_L}$-Tree shown in Fig. 2c. Subsequently, according to the *tf* value, items are sorted in descending order and the new order is "*a c e b d*" as shown in Fig. 3a. Items in the transactions of db_1^+ are inserted in the tree in that order. When db_1^+ is finished, the tree is sorted according to the *tf* value of items, and the new order now is "*e a b c d,*" as shown in Fig. 3b. By the same

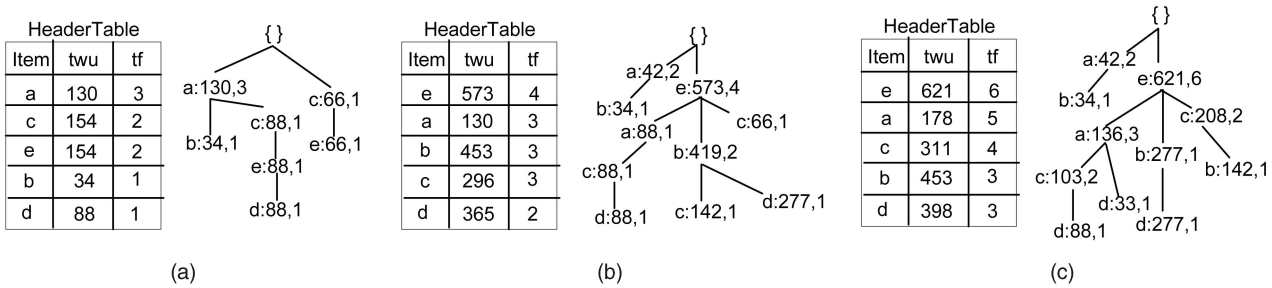

Fig. 3. Construction of the $\mathrm{IHUP_{TF}}$-Tree. (a) $\mathrm{IHUP_{TF}}$-Tree up to $\mathrm{T_4}$, (b) $\mathrm{IHUP_{TF}}$ up to $\mathrm{T_6}$, and (c) $\mathrm{IHUP_{TF}}$ up to $\mathrm{T_8}$.

fashion, db_2^+ is processed, the tree is restructured at the end of db_2^+, and the final order is "*e a c b d*," as shown in Fig. 3c. The $\mathrm{IHUP_{TF}}$-Tree in Fig. 3c has only 11 nodes (without root) compared to the 15 nodes of the $\mathrm{IHUP_L}$-Tree in Fig. 2e. Deletion and modification operations can be done in the same way as the $\mathrm{IHUP_L}$-Tree shown in Section 4.1.

4.3 $\mathrm{IHUP_{TWU}}$-Tree Construction

Several low-*twu* items can appear before the high-*twu* items in branches of the $\mathrm{IHUP_L}$-Tree and $\mathrm{IHUP_{TF}}$-Tree. For example, if *minutil* = 198.9 in Fig. 1, then "*a*" is a low-*twu* item. However, "*a*" appears before high-*twu* items "*b*," "*c*," "*d*," and "*e*" in the $\mathrm{IHUP_L}$-Tree and high-*twu* items "*b*," "*c*," and "*d*" in the $\mathrm{IHUP_{TF}}$-Tree. During the mining process, these noncandidate nodes incur a huge amount of delay (we will elaborately discuss in Section 5). This is the reason why we are proposing the $\mathrm{IHUP_{TWU}}$-Tree, which is designed according to the *twu* value in descending order. In the $\mathrm{IHUP_{TWU}}$-Tree, all candidate nodes are kept before the noncandidate nodes in every branch. The $\mathrm{IHUP_{TWU}}$-Tree can be constructed any time from an $\mathrm{IHUP_L}$-Tree using a path adjusting method [17] based on bubble sort technique. We will show in the mining section (Section 5) that the number of nodes participating in the mining operation in the $\mathrm{IHUP_{TWU}}$-Tree is always less than or equal to the other two tree structures, and therefore, it achieves the fastest mining time as well as the overall running time.

Consider the transaction database and utility table in Fig. 1. In the original database (up to T_4), the $\mathrm{IHUP_{TWU}}$-Tree is constructed in the same way as the $\mathrm{IHUP_L}$-Tree shown in Fig. 2c. Subsequently, according to the *twu* value of items, the tree is restructured. The new order is "*c e a d b*." Items in the transactions of db_1^+ are inserted in the tree in that order. When db_1^+ is added to the tree, the tree is restructured again according to the *twu* value of items. The new order now is "*e b d c a*." By the same fashion, db_2^+ is processed and no restructuring is required at the end of db_2^+ because the final order "*e b d c a*" remains the same, as shown in Fig. 4. The $\mathrm{IHUP_{TWU}}$-Tree in Fig. 4 has 13 nodes compared to 15 nodes in the $\mathrm{IHUP_L}$-Tree in Fig. 2e and 11 nodes in $\mathrm{IHUP_{TF}}$-Tree in Fig. 3c. Deletion and modification operations can be done in the same way as the $\mathrm{IHUP_L}$-Tree shown in Section 4.1.

The following property is true for all IHUP-Trees:

Property 2. *The twu and tf values of any node in an IHUP-Tree is greater than or equal to the sum of total counts of twu and tf values, respectively, of its children.*

4.4 IHUP-Tree Construction and Mining Algorithm

Fig. 5 describes the IHUP-Tree construction and mining algorithm. Our algorithm keeps the items in lexicographic, *tf* descending order, and *twu* descending order for $\mathrm{IHUP_L}$-Tree, $\mathrm{IHUP_{TF}}$-Tree, and $\mathrm{IHUP_{TWU}}$-Tree, respectively, in both header table and tree nodes. To maintain the sort order of $\mathrm{IHUP_{TF}}$-Tree and $\mathrm{IHUP_{TWU}}$-Tree, restructuring operation is performed using the path adjusting method [17] (based on bubble sort technique). A user-given parameter N is used here. This parameter indicates a percentage of database after which restructuring operation should be done. For example, if $N = 10\%$ (we can also write $N = 0.1$) and the database contains 100,000 transactions, then, after each 10,000 transactions, the restructuring operation will be done to achieve the desired sort order. As stated in Property 1, the $\mathrm{IHUP_L}$-Tree does not need this restructuring operation as it maintains the lexicographic order of items.

After scanning each transaction, our algorithm performs the insertion/deletion/modification operations explained in Sections 4.1-4.3. When the initial *DB* or any $db^+/db^-/db_{mod}$ is finished, our algorithm asks the user to input the mining threshold. Using the power of "*build once mine many*" property, it can perform many mining operations using different minimum thresholds without rebuilding the tree. It also keeps track of the last mining result to reduce the computation in mining operation and the number of candidates to be scanned in the database for the second time. It uses a pattern growth mining approach to avoid the level-wise candidate generation-and-test methodology of the existing algorithms.

5 Analysis of Mining Performance

In the FP-growth mining algorithm [12], when a prefix tree is created for a particular item, then all branches prefixing that item are taken with the frequency value of that item. Since the FP-tree contains only the global frequent items,

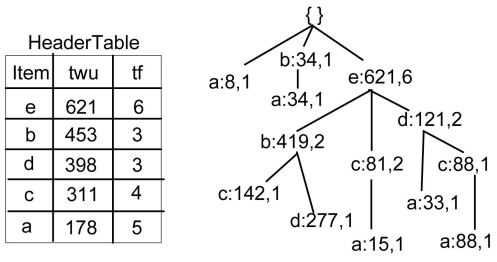

Fig. 4. Construction of the $\mathrm{IHUP_{TWU}}$-Tree upto T_8.

```
Input: DB, group of db+, db− and db_mod, δ, N (percentage of DB, after which restructuring operation should be done each time in
       the case of IHUP_TF-Tree and IHUP_TWU-Tree)
Output: High Utility Patterns
begin
    foreach transaction T_i do
        Sort the items inside T_i according to the current sort order
        Insert/Delete/Modify T_i into the tree
        Update the twu and tf in the header table H
        /* The following if block is only applicable for IHUP_TF-Tree and IHUP_TWU-Tree */
        if the next N % of database is scanned or end of database found then
            Perform bubble sort operation to restructure the tree and header table H.
        end
        if T_i is the last transaction of DB or any db+/db−/db_mod then
            while there is a mining request from user do
                Input δ from the user
                if previous δ > current δ or this is the first δ value for this IHUP-Tree then
                    foreach item α of H do
                        if twu (α) ≥ minutil then
                            Create Prefix-tree PT_α with its header table HT_α for item α
                            Call Mining (PT_α, HT_α, α)
                        end
                    end
                    if this is the first δ value for this IHUP-Tree then
                        Scan the database a second time to calculate high utility patterns from the candidate patterns
                    end
                    else
                        Scan the database a second time for only those candidate patterns which are not covered by the previous
                        result
                    end
                end
                else
                    Find the high utility patterns from previous result without mining and second database scan
                end
            end
        end
    end
end
Procedure Mining(T, H, α)
begin
    foreach item β of H do
        if twu(β) < minutil then
            Delete β from T and H to create conditional tree and header table
        end
    end
    Let CT be the Conditional tree of α created from T
    Let HC be the Header table of Conditional tree CT created from H
    foreach item β in HC do
        Add pattern αβ in the candidate pattern list
        Create Prefix-tree PT_αβ and Header table HT_αβ for pattern αβ
        Call Mining (PT_αβ , HT_αβ , αβ)
    end
end
```

Fig. 5. The IHUP-Tree construction and mining algorithm.

i.e., no branch in the tree contains infrequent items between frequent items, for the construction of the conditional tree for a particular item from its prefix tree, we have to eliminate the infrequent items associated with that particular item, but not any global infrequent items. However, if global infrequent items are inside the frequent items, then they will appear in the prefix trees of the frequent items and have to be deleted while creating the conditional trees. This increases the prefix and conditional tree creation time of the frequent items and the overall mining time.

Let us consider $minutil = 198.9$ for Fig. 1 and proceed to the mining operation for the tree structures we have proposed, then we can see the real picture of what happens if low-*twu* items are in between the high-*twu* items. For the specified minutil, item "*a*" is a low-*twu* item, but "*a*" appears between the high-*twu* items in the IHUP_{L}-Tree and IHUP_{TF}-Tree. So, the prefix-tree size of the high-*twu* items will be larger for this global low-*twu* item "*a*" and to create conditional trees of these high-*twu* itemsets, node "*a*" has to be deleted several times. This requires extra creation time for prefix trees and extra deletion time for creating conditional trees of high-*twu* itemsets. As a result, the mining operation will take more time. All of the high-*twu* items (*b*, *c*, *d*, and *e*) will have item "*a*" in their prefix tree. The prefix tree for the bottommost high-*twu* item "*e*" is shown in Fig. 6a. In addition to containing global low-*twu* items, prefix-sharing is less in the IHUP_{L}-Tree compared to the other two trees, i.e., items are scattered. Here, item "*e*" does not have any prefix-sharing and is scattered in six nodes. So, the prefix tree of the bottommost *twu* item "*e*" is larger than the other two trees.

A very interesting result is that although the IHUP_{TF}-Tree has 11 nodes (Fig. 3c) compared to 13 nodes (Fig. 4) in the IHUP_{TWU}-Tree, the prefix tree of its bottommost high-*twu* item "*d*" has 4 nodes (Fig. 6b) compared to 3 nodes (Fig. 6c) in the prefix tree of bottommost high-*twu* item "*c*" in

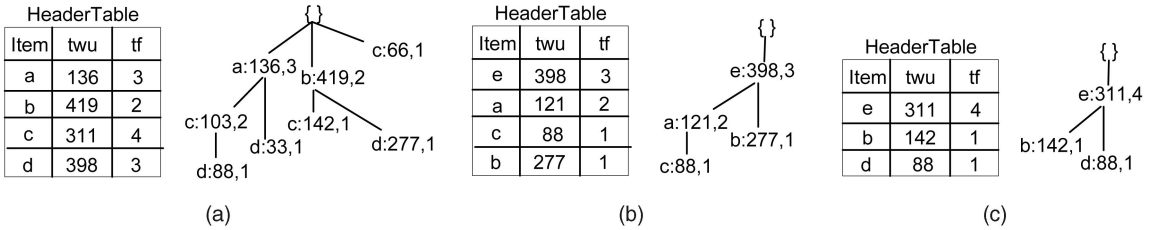

Fig. 6. Construction of Prefix trees for the bottommost high-*twu* item in mining operation. (a) Prefix tree for the bottommost item "e" in the $\mathrm{IHUP_L}$-Tree, (b) prefix tree for the bottommost item "d" in the $\mathrm{IHUP_{TF}}$-Tree, and (c) prefix tree for the bottommost item "c" in the $\mathrm{IHUP_{TWU}}$-Tree.

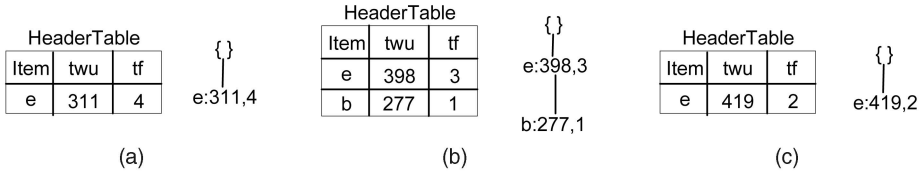

Fig. 7. Mining operation in the $\mathrm{IHUP_{TWU}}$-Tree. (a) Conditional tree for item "c," (b) prefix and conditional tree for item "d," and (c) prefix and conditional tree for item "b."

$\mathrm{IHUP_{TWU}}$-Tree. The reason behind this result is that 11 nodes are participating in mining operations in the $\mathrm{IHUP_{TF}}$-Tree (Fig. 3c), but only 8 nodes (Fig. 4) in the case of the $\mathrm{IHUP_{TWU}}$-Tree. If we observe Fig. 4, then we can see that the low-*twu* item "*a*" is present at the end of every branch. So, it will never appear in any prefix tree of other high-*twu* items. This property is always true. For example, if we increase the *minutil* from 198.9 to 320, then we can observe that low-*twu* items "*a*" and "*c*" appear at the bottommost part in every branch, so they will never appear in the prefix trees of other high-*twu* nodes, i.e., only 5 nodes will participate in mining operation for that minutil compared to 11 nodes in the $\mathrm{IHUP_{TF}}$-Tree.

Lemma 1. *The number of nodes participating in the mining operation of the* $\mathrm{IHUP_{TWU}}$*-Tree is always less than or equal to the number of nodes participating in the mining operation of the* $\mathrm{IHUP_{TF}}$*-Tree and* $\mathrm{IHUP_L}$*-Tree.*

Proof. The $\mathrm{IHUP_{TWU}}$-Tree arranges items in *twu* descending order; therefore, all the nodes containing low-*twu* item cannot appear in any prefix tree of the high-*twu* items. On the other hand, for the other two trees, these items may appear anywhere in the tree. As a result, they may appear in the prefix trees of other items. So, the number of nodes participating in the mining operation of the $\mathrm{IHUP_{TWU}}$-Tree cannot be greater than the number of nodes participating in the mining operation in the other two trees. □

Lemma 2. *The number of nodes N participating in the mining operation in the* $\mathrm{IHUP_{TWU}}$*-Tree decreases when the number of low-twu items increases for increased minutil values.*

Proof. N is equal to the total number of nodes of the $\mathrm{IHUP_{TWU}}$-Tree when the *minutil* value is zero. When *minutil* increases, some nodes containing low-*twu* items at the bottom of the tree cannot appear in any prefix tree of high-*twu* items. Suppose for *minutil* x, the number of nodes not participating in mining is y. If x increases by d_x, then y has to increase by d_y amount ($d_y \geq 0$). So, $y \leq y + d_y$. □

In the above discussion in this section, Lemmas 1 and 2 explain the reasons for performance variation of our proposed three tree structures. However, the final candidate patterns are the same for all three tree structures. Now, we complete the mining example for $\mathrm{IHUP_{TWU}}$-Tree. In the mining operation in the $\mathrm{IHUP_{TWU}}$-Tree for *minutil* 198.9, we construct the conditional tree of the bottommost high-*twu* item "*c*" (shown in Fig. 7a), from its prefix tree (shown in Fig. 6c) by deleting items "*b*" and "*d*" as they have low-*twu* values of 142 and 88, respectively. So, from Fig. 7a, we can generate candidate patterns $\{c, e : 311, 4\}$ and $\{c : 311, 4\}$. Subsequently, the prefix tree of the next item "*d*" is created in Fig. 7b. Both "*e*" and "*b*" have high-*twu* values with item "*d*," so this is also its conditional tree. Candidate patterns $\{b, d : 277, 1\}$, $\{b, d, e : 277, 1\}$, $\{d, e : 398, 3\}$, and $\{d : 398, 3\}$ are generated from the conditional tree of item "*d*" in Fig. 7b. In the same way, candidate patterns $\{b, e : 419, 2\}$ and $\{b : 453, 3\}$ are generated from the conditional tree of item "*b*" in Fig. 7c. Another candidate pattern $\{e : 621, 6\}$ is generated for the topmost item "*e*." A second database scan is required to find high utility itemsets from these nine high-*twu* itemsets. The five high utility itemsets are $\{b : 285\}$, $\{b, d : 214\}$, $\{b, d, e : 277\}$, $\{b, e : 346\}$, and $\{d, e : 202\}$. As mentioned in Section 3, the *tf* value is very useful also in the second database scan for stopping the monitoring of one particular pattern.

Observation 1. The existing recent algorithms such as Two-Phase, FUM, DCG+, etc., are based on Apriori-like algorithms, and therefore, suffer from the level-wise candidate set generation-and-test problem and need several database scans. At first, they scan the database to find single-element candidate patterns. Based on this result, they generate all the candidates for two-element patterns. For example, consider the database presented in Fig. 1 and minutil 198.9. Here "b," "c," "d," and "e" are single-element candidates. For two-element candidates, they generate all the combinations by taking two items at a time $\binom{4}{2}$ without knowing if those patterns appear in the database or not. So, if the number of single-element

candidates is 10,000, then they will generate $\binom{10,000}{2}$ two-element candidate patterns. In a similar way, they find the candidates of the three-element patterns, and so on. Therefore, the existing algorithms generate huge candidates and need to scan database several times.

Lemma 3. *If N_1 is the number of candidate itemsets generated by the IHUP-tree mining algorithm and N_2 is the number of candidate itemsets generated by Apriori-based high utility mining algorithms, then $N_1 \leq N_2$.*

Proof. A pattern $X\{x_1, x_2, \ldots\ldots x_n\}$ is a candidate high-*twu* itemset iff all of its subsets of length $n-1$ are high-*twu* itemsets in Apriori-based high utility algorithms. So, X may not be present in the database or it could have too low utility value to become a candidate. In the IHUP-Tree mining algorithm, if X is not present in the database, then it cannot appear in any branch of the tree, and therefore, it cannot appear as a candidate. Moreover, after determining X is a low-*twu* itemset, it is pruned. Therefore, the candidate set of the IHUP-Tree contains only true high-*twu* itemsets; hence, N_1 cannot be greater than N_2. □

Observation 2. When the maximum length of the candidate patterns increases, Apriori-based high utility mining algorithms have to scan database repeatedly. For example, in Fig. 1 and $\text{minutil} = 198.9$, the maximum length of a candidate pattern is 3(bde). Existing recent FUM and DCG+ algorithms scan the database three times to find all of the candidate and high utility itemsets. For the Two-Phase algorithm, one extra scan is required at the last for finding out the high utility itemsets from the candidate itemsets. Therefore, a total of four database scans are required for this example. For the maximum candidate length N, a total of N database scans could be required for the existing recent FUM and DCG+ algorithms and a total of $N+1$ database scans are required for the Two-Phase algorithm. On the other hand, the number of database scans required for our algorithm is totally independent of the maximum length of candidate patterns. Always a maximum of two database scans are required. As the minimum utility threshold decreases, the number of candidate patterns and their maximum length also increases. So, as the minimum utility threshold decreases, running time increases very sharply in Apriori-based algorithms.

6 Incremental and Interactive Mining

The IHUP-Trees are efficient for incremental mining. They require adjustment for only the last added/deleted/modified group of transactions. For example, after the creation of the IHUP_{TF}-Tree in Fig. 3a for the original database presented in Fig. 1a, we can perform mining operations. After that, when the database is incremented by adding one group of transactions (db_1^+), we can use the existing IHUP_{TF}-Tree (Fig. 3a) to capture these incremental data and construct the tree (Fig. 3b) for the incremented database. As a result, insertion cost for only the transactions in db_1^+ is needed rather than creating the whole tree structure from the very beginning. Now, we can perform mining operations on the current tree (Fig. 3b) to discover the resultant high utility patterns for the current incremented database (original DB + db_1^+ in Fig. 1a). In a similar way, deletion/modification operations can be handled and mining operations can be performed on the updated tree to discover the resultant high utility patterns for the current updated database.

The IHUP-Trees exploit the "*build once mine many*" property, and therefore, they are also efficient in interactive mining. This property means that after creating one tree, several mining operations can be performed using different minimum thresholds without rebuilding the tree. For example, after the creation of the IHUP-Trees in Figs. 2e, 3c, and 4, first $\delta = 30\%$ will be calculated, then after finding the result $\delta = 20\%$ or $\delta = 40\%$ can be calculated. The same operation can be done several times; hence, after the first time, we do not have to consider the tree creation time. Advantages in mining time and second database scanning time for determining high utility patterns from high-*twu* patterns are also realized. For example, after obtaining five high utility patterns from nine candidate high-*twu* patterns for $\delta = 30\%$, if we perform mining operation for larger values of δ, then the candidate set is a subset of the previous candidate set and the actual high utility itemsets can be defined without mining or scanning the database again. If δ is smaller than the previous trial, then the candidate set is a superset of the previous candidate set. Therefore, after mining, the database can be scanned only for patterns that did not appear in the previous candidate set.

7 Experimental Results

7.1 Experimental Environment and Datasets

To evaluate the performance of our proposed tree structures, we have performed several experiments on real-life datasets [28], [29] (*mushroom, retail*, and *kosarak*) using synthetic utility values, and a real-life dataset (*Chain-store*) using real utility values [30]. The performance of our algorithm was compared with the existing algorithms Two-Phase [7], [8], FUM, and DCG+ [35].

At first, we show the effectiveness of our tree structures in interactive mining using the *mushroom* and *retail* datasets. Next, we show the effectiveness of the tree structures in incremental mining using the *kosarak* dataset. These datasets (*mushroom, retail*, and *kosarak*) do not provide the profit values (external utility) or quantity of each item for each transaction (internal utility). Like the performance evaluation of the previous high utility pattern mining algorithms [7], [8], [26], [35], we have generated random numbers for the internal utility values ranging from 1 to 10. Observed from the real world datasets, most items are in the low-profit range. Therefore, the external utility of each item was heuristically chosen between 0.01 and 10, and randomly generated using a log-normal distribution. Some other high utility pattern mining research [7], [8], [26], [35] has adopted the same technique. Fig. 8 shows the external utility distribution of 2,000 distinct items using a log-normal distribution.

After that, we show the overall performance of the tree structures in the real-life dataset *Chain-store* containing real utility values [30] and runtime distribution of the IHUP-Trees. Finally, the memory usage and scalability of the tree structures are discussed. Our programs were written in Microsoft Visual C++ 6.0, and run with the Windows XP operating system on a Pentium dual-core 2.13 GHz CPU

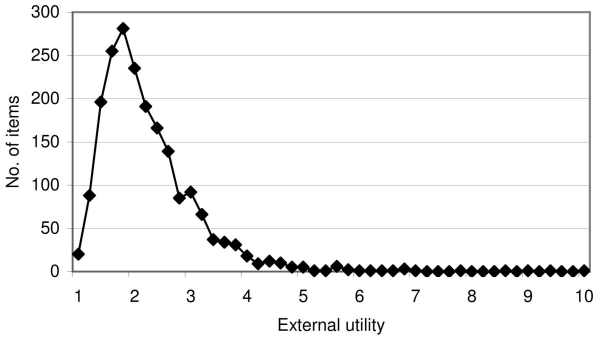

Fig. 8. External utility distribution for 2,000 distinct items using log-normal distribution.

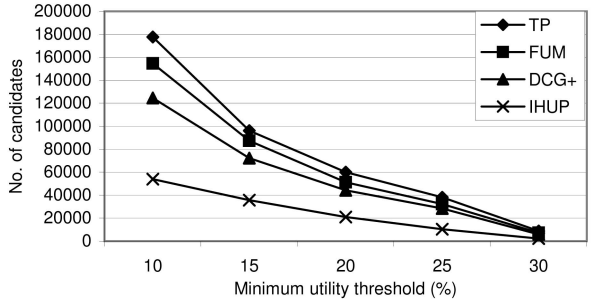

Fig. 9. Number of candidates comparison on the *mushroom* dataset.

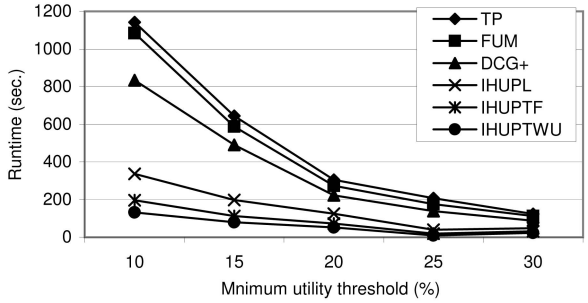

Fig. 10. Runtime comparison on the *mushroom* dataset.

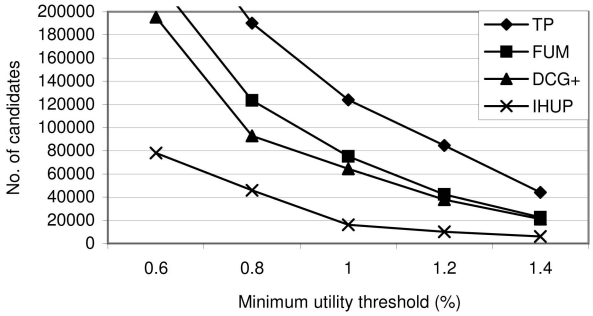

Fig. 11. Number of candidates comparison on the *retail* dataset.

with 2 GB main memory.

7.2 Effectiveness of the IHUP-Trees in Interactive Mining

The dataset *mushroom* includes descriptions of hypothetical samples corresponding to 23 species of mushrooms [29], [28]. It contains 8,124 transactions and 119 distinct items. Its mean transaction size is 23, and it is a dense dataset. Almost 20 percent $((23/119) \times 100)$ of its distinct items are present in every transaction, so it has long high utility patterns. As stated in Section 5, all the three tree structures have the same number of candidate patterns. Fig. 9 shows the comparison of candidate number and Fig. 10 shows the runtime comparison on the *mushroom* dataset. The minimum utility threshold range of 10 percent to 30 percent is used here. The value of N (described in Section 4.4) is 10 percent for both the IHUP_{TF}-Tree and IHUP_{TWU}-Tree in this dataset.

We have taken here the result in Fig. 10 for the worst case of interactive mining for our trees, i.e., we listed the threshold in descending order. As we stated in Section 6, our IHUP-Trees achieve benefit after the first mining threshold. Our trees do not have to be constructed after the first threshold. As the threshold decreases, mining operation is needed. However, as candidates of threshold 25 percent are a superset of the candidates of threshold 30 percent, we can easily save the second database scan time for the candidates of threshold 30 percent. High utility itemsets from them are already calculated for threshold 30 percent. We have to scan the database a second time only for the candidates newly added for threshold 25 percent. The best case occurs if we arrange the mining threshold in increasing order. Then, candidates of threshold 20 percent are a subset of the candidates of threshold 10 percent. Therefore, without mining and a second database scan, we can find the resultant high utility itemsets from the previous result. In that case, after the first mining threshold, the computation time for other thresholds is negligible compared to that required for the first. Figs. 9 and 10 also reflect our analysis presented in Observation 2 in Section 5, i.e., the number of candidates and runtime differences between the existing methods and our algorithm become larger as δ decreases.

The dataset *retail* is provided by Tom Brijs, and contains the retail market basket data from an anonymous Belgian retail store [31], [28]. It contains 88,162 transactions and 16,470 distinct items. Its mean transaction size is 10.3, and it is a large sparse dataset. Around 0.0625 percent $((10.3/16{,}470) \times 100)$ of its distinct items are present in every transaction. Fig. 11 shows the comparison of candidate number and Fig. 12 shows the runtime comparison on the *retail* dataset. The minimum utility threshold range of 0.6-1.4 percent is used here. The value of N is 10 percent for both the IHUP_{TF}-Tree and IHUP_{TWU}-Tree in this dataset. For the worst-case analysis of the interactive mining, we have also given the value of δ here in descending order. As stated in Observation 1 in Section 5, Figs. 11 and 12 show the effect of too many distinct items in *retail* dataset over the level-wise Apriori-based existing algorithms.

7.3 Effectiveness of the IHUP-Trees in Incremental Mining

We have tested the effectiveness of the IHUP-Trees in incremental mining on the *kosarak* dataset. The dataset *kosarak* was provided by Ferenc Bodon and contains click-stream data of a Hungarian online news portal [28]. It contains 990,002 transactions and 41,270 distinct items. Its mean transaction size is 8.1, and it is a large sparse dataset. Around 0.0196 percent $((8.1/41{,}270) \times 100)$ of its distinct items are present in every transaction.

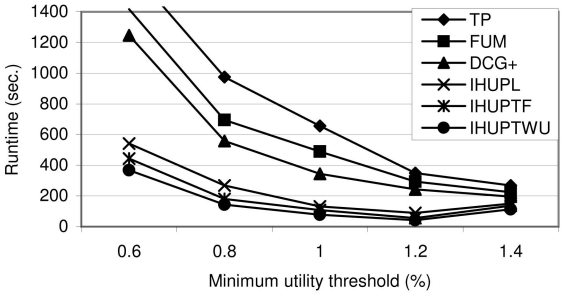

Fig. 12. Runtime comparison on the *retail* dataset.

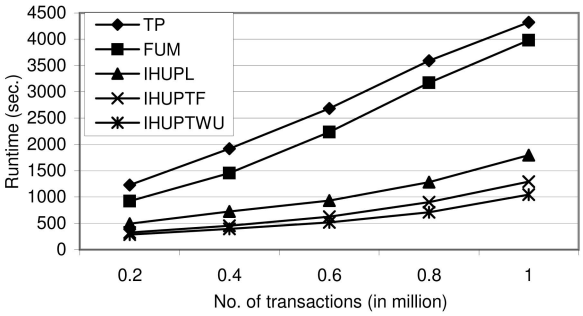

Fig. 13. Database increasing by $db^{+} = 0.2M$ on the *kosarak* dataset.

At first, we created the IHUP-Trees for 0.2 million transactions of this dataset, and then performed a mining operation with $\delta = 5\%$. Another 0.2 million transactions were added in the tree and the mining operations were performed again with the same utility threshold. In the same manner, all of the transactions in the *kosarak* dataset were added and the mining operation was performed in the IHUP-Trees for each stage at $\delta = 5\%$, as shown in Fig. 13. After adding each db^{+}, we restructured the IHUP_{TF}-Tree and IHUP_{TWU}-Tree before mining. It is obvious in Fig. 13 that as the database increases in size, the tree construction and mining time also increase. After adding all of the db^{+}, we performed the deletion operation in that tree. Here db^{-} size is 0.1 million. At first, 0.1 million transactions were deleted and the mining operation was performed using $\delta = 5\%$. The same operation was repeated four times as shown in Fig. 14. After deleting each db^{-}, we have restructured the IHUP_{TF}-Tree and IHUP_{TWU}-Tree before mining. It is also obvious in Fig. 14 that as the database size decreases, the tree construction and mining time also decrease. We have compared the performance of our tree structures with the existing Two-Phase and FUM algorithm. As DCG+ maintains an extra array for each candidate [35], we could not keep all its candidates in each pass in the main memory. Figs. 13 and 14 show that our tree structures outperform the existing algorithms.

7.4 Effectiveness of the IHUP-Trees in Real-Life Dataset with Real Utility Values

In this section, we use a real-life dataset adopted from NU-MineBench 2.0, a powerful benchmark suite consisting

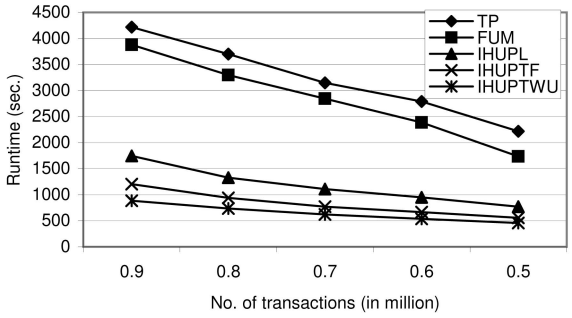

Fig. 14. Database decreasing by $db^{-} = 0.1M$ on the *kosarak* dataset.

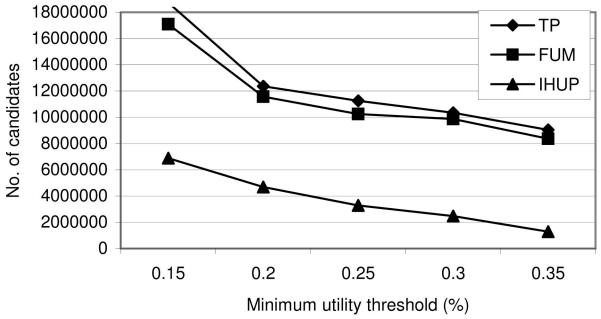

Fig. 15. Number of candidates comparison on the *Chain-store* dataset.

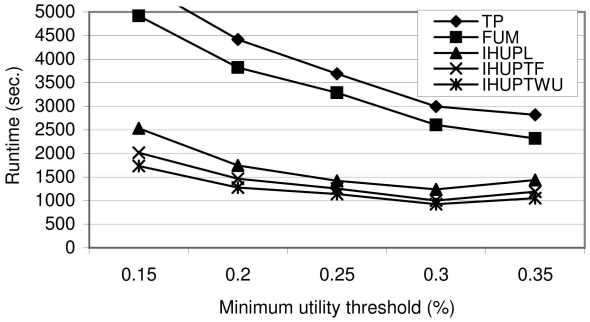

Fig. 16. Runtime comparison on the *Chain-store* dataset.

of multiple data mining applications and databases [30]. This dataset called *Chain-store* was taken from a major chain in California and contains 1,112,949 transactions and 46,086 distinct items [30], [35]. The dataset's utility table stores the profit for each item. The total profit of the dataset is $\$26{,}388{,}499.80$.

The mean transaction size of the real-life *Chain-store* dataset is 7.2, and it is a large sparse dataset. Around 0.0156 percent $((7.2/46{,}086) \times 100)$ of its distinct items are present in every transaction. Figs. 15 and 16 show the comparison of candidate number and the runtime, respectively, on this dataset. The minimum utility threshold range of 0.15-0.35 percent is used here. The value of N is 10 percent for both the IHUP_{TF}-Tree and IHUP_{TWU}-Tree in this dataset. We have compared the performance of our tree structures with the existing Two-Phase and FUM algorithm. As DCG+ maintains an extra array for each candidate [35], we could not keep all its candidates in each

TABLE 1
Runtime Distribution (in Seconds) of the IHUP-Trees

Dataset	Tree	Tree construction time	Tree restructuring time	Mining time	2nd DB Scan time	Total time
mushroom	IHUPL	5.715	0	30.755	11.15	47.62
$\delta = 30\%$	IHUPTF	5.593	0.828	14.309	11.06	31.79
	IHUPTWU	5.608	0.782	5.05	11.83	23.27
retail	IHUPL	47.756	0	52.938	49.867	150.561
$\delta = 1.4\%$	IHUPTF	46.624	9.681	30.24	49.801	136.346
	IHUPTWU	46.736	9.348	6.183	49.903	112.17
Chain-store	IHUPL	325.69	0	728.876	385.36	1439.926
$\delta = 0.35\%$	IHUPTF	323.483	77.052	405.671	385.82	1192.026
	IHUPTWU	324.78	76.983	265.741	384.562	1052.066

pass in the main memory. Figs. 15 and 16 show that our tree structures outperform the existing algorithms.

7.5 Runtime Distribution of the IHUP-Trees

Table 1 shows the runtime distribution for the three tree structures. As the IHUP_{L}-Tree does not require a restructuring operation, the restructuring time is zero. With a small restructuring cost, IHUP_{TF}-Tree and IHUP_{TWU}-Tree achieve faster mining time compared to the IHUP_{L}-Tree. According to the analysis of Section 5, the IHUP_{TWU}-Tree achieves fastest mining as well as overall runtime (shown in Table 1).

7.6 Memory Usage and Scalability of the IHUP-Trees

Prefix-tree-based frequent pattern mining techniques [16], [17], [18], [19], [34], [36] have shown that the memory requirement for the prefix trees is low enough to use the current available gigabyte-range memory. We have also handled our tree structures very efficiently and kept it within this memory range. Our prefix-tree structure can represent the useful information in a very compressed form because transactions have many items in common. By utilizing this type of path overlapping (prefix-sharing), our tree structures can save memory space. Moreover, by using a pattern growth approach, we generate a much smaller number of candidates compared to the existing algorithms. Table 2 shows that our prefix-tree structures outperform the existing algorithms in memory usage. Moreover, IHUP_{TF}-Tree requires the lowest amount of memory.

It is shown in Fig. 16 that our tree structures have easily handled the 46,086 distinct items and more than 1 million transactions in the real-life *Chain-store* dataset. Fig. 13 also shows that they have efficiently handled the 41,270 distinct items and around 1 million transactions in the *kosarak* dataset. Therefore, these experimental results demonstrate the scalability of our tree structures to handle large number of distinct items and transactions.

TABLE 2
Memory Comparison (in Megabytes)

Dataset	IHUPL	IHUPTF	IHUPTWU	FUM
mushroom	0.721	0.415	0.486	2.256
retail	16.761	13.067	13.819	31.73
kosarak	208.237	178.134	183.856	454.296
Chain-store	272.94	236.875	245.671	528.348

8 Conclusions

The key contribution of this paper is to provide an efficient research method for high utility pattern mining for handling incremental databases, while considering many insertions, deletions, and modifications with the currently available memory size. Three variations of our tree structure have been proposed. Among them, the IHUP_{L}-Tree is very simple and easy to construct and handle, as it does not require any restructuring operation in spite of incremental updating of the databases. We have shown that with a small restructuring cost, we have achieved an IHUP_{TF}-Tree that requires less memory and an IHUP_{TWU}-Tree that requires less time to execute than previous methods. We have used a pattern growth approach to avoid the level-wise candidate generation-and-test methodology. All of the tree structures have the "*build once mine many*" property and are highly suitable for interactive mining. All three tree structures require maximum two database scans. Extensive performance analyses show that our tree structures are very efficient for incremental and interactive high utility pattern mining and they outperform the existing algorithms in both execution time and memory usage. Moreover, they are scalable for handling a large number of distinct items and transactions.

Acknowledgments

The authors would like to express their deep gratitude to the anonymous reviewers of this paper. Their useful comments have played a significant role in improving the quality of this work.

References

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules Between Sets of Items in Large Databases," *Proc. 12th ACM SIGMOD*, pp. 207-216, 1993.

[2] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. 20th Int'l Conf. Very Large Data Bases (VLDB '94)*, pp. 487-499, 1994.

[3] U. Yun and J.J. Leggett, "WFIM: Weighted Frequent Itemset Mining with a Weight Range and a Minimum Weight," *Proc. Fifth SIAM Int'l Conf. Data Mining (SDM '05)*, pp. 636-640, 2005.

[4] U. Yun, "Efficient Mining of Weighted Interesting Patterns with a Strong Weight and/or Support Affinity," *Information Sciences*, vol. 177, pp. 3477-3499, 2007.
[5] H. Yao, H.J. Hamilton, and C.J. Butz, "A Foundational Approach to Mining Itemset Utilities from Databases," *Proc. Fourth SIAM Int'l Conf. Data Mining (SDM '04)*, pp. 482-486, 2004.
[6] H. Yao and H.J. Hamilton, "Mining Itemset Utilities from Transaction Databases," *Data and Knowledge Eng.*, vol. 59, pp. 603-626, 2006.
[7] Y. Liu, W.-K. Liao, and A. Choudhary, "A Two Phase Algorithm for Fast Discovery of High Utility of Itemsets," *Proc. Ninth Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD '05)*, pp. 689-695, 2005.
[8] Y. Liu, W.-K. Liao, and A. Choudhary, "A Fast High Utility Itemsets Mining Algorithm," *Proc. First Int'l Conf. Utility-Based Data Mining*, pp. 90-99, 2005.
[9] B. Barber and H.J. Hamilton, "Extracting Share Frequent Itemsets with Infrequent Subsets," *Data Mining and Knowledge Discovery*, vol. 7, pp. 153-185, 2003.
[10] R. Chan, Q. Yang, and Y.D. Shen, "Mining High Utility Itemsets," *Proc. Third IEEE Int'l Conf. Data Mining (ICDM '03)*, pp. 19-26, 2003.
[11] F. Tao, "Weighted Association Rule Mining Using Weighted Support and Significant Framework," *Proc. Ninth ACM SIGKDD*, pp. 661-666, 2003.
[12] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," *Data Mining and Knowledge Discovery*, vol. 8, pp. 53-87, 2004.
[13] G. Grahne and J. Zhu, "Fast Algorithms for Frequent Itemset Mining Using FP-Trees," *IEEE Trans. Knowledge and Data Eng.*, vol. 17, no. 10, pp. 1347-1362, Oct. 2005.
[14] J. Wang, J. Han, Y. Lu, and P. Tzvetkov, "TFP: An Efficient Algorithm for Mining Top-K Frequent Closed Itemsets," *IEEE Trans. Knowledge and Data Eng.*, vol. 17, no. 5, pp. 652-664, May 2005.
[15] C. Lucchese, S. Orlando, and R. Perego, "Fast and Memory Efficient Mining of Frequent Closed Itemsets," *IEEE Trans. Knowledge and Data Eng.*, vol. 18, no. 1, pp. 21-36, Jan. 2006.
[16] W. Cheung and O.R. Zaïane, "Incremental Mining of Frequent Patterns without Candidate Generation or Support Constraint," *Proc. Seventh Int'l Database Eng. and Applications Symp. (IDEAS '03)*, pp. 111-116, 2003.
[17] J.-L. Koh and S.-F. Shieh, "An Efficient Approach for Maintaining Association Rules Based on Adjusting FP-Tree Structures," *Proc. Ninth Int'l Conf. Database Systems for Advanced Applications (DASFAA '04)*, pp. 417-424, 2004.
[18] X. Li, Z.-H. Deng, and S. Tang, "A Fast Algorithm for Maintenance of Association Rules in Incremental Databases," *Proc. Advanced Data Mining and Applications (ADMA 06)*, pp. 56-63, 2006.
[19] C.K.-S. Leung, Q.I. Khan, Z. Li, and T. Hoque, "CanTree: A Canonical-Order Tree for Incremental Frequent-Pattern Mining," *Knowledge and Information Systems*, vol. 11, no. 3, pp. 287-311, 2007.
[20] W. Wang, J. Yang, and P.S. Yu, "WAR: Weighted Association Rules for Item Intensities," *Knowledge Information and Systems*, vol. 6, pp. 203-229, 2004.
[21] U. Yun, "Mining Lossless Closed Frequent Patterns with Weight Constraints," *Knowledge-Based Systems*, vol. 20, pp. 86-97, 2007.
[22] H. Xiong, P.-N. Tan, and V. Kumar, "Hyperclique Pattern Discovery," *Data Mining and Knowledge Discovery*, vol. 13, pp. 219-242, 2006.
[23] J. Dong and M. Han, "BitTableFI: An Efficient Mining Frequent Itemsets Algorithm," *Knowledge-Based Systems*, vol. 20, pp. 329-335, 2007.
[24] M. Song and S. Rajasekaran, "A Transaction Mapping Algorithm for Frequent Itemsets Mining," *IEEE Trans. Knowledge and Data Eng.*, vol. 18, no. 4, pp. 472-481, Apr. 2006.
[25] Y.-H. Wen, J.-W. Huang, and M.-S. Chen, "Hardware-Enhanced Association Rule Mining with Hashing and Pipelining," *IEEE Trans. Knowledge and Data Eng.*, vol. 20, no. 6, pp. 784-795, June 2008.
[26] A. Erwin, R.P. Gopalan, and N.R. Achuthan, "CTU-Mine: An Efficient High Utility Itemset Mining Algorithm Using the Pattern Growth Approach," *Proc. Seventh IEEE Int'l Conf. Computer and Information Technology (CIT '07)*, pp. 71-76, 2007.
[27] J. Hu and A. Mojsilovic, "High Utility Pattern Mining: A Method for Discovery of High Utility Itemsets," *Pattern Recognition*, vol. 40, pp. 3317-3324, 2007.
[28] Frequent Itemset Mining Dataset Repository, http://fimi.cs.helsinki.fi/data/, accessed Jan. 2008.
[29] UCI Machine Learning Repository, http://kdd.ics.uci.edu/, accessed Jan. 2008.
[30] J. Pisharath, Y. Liu, J. Parhi, W.-K. Liao, A. Choudhary, and G. Memik, NU-MineBench Version 2.0 Source Code and Datasets, http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html, accessed June 2008
[31] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets, "Using Association Rules for Product Assortment Decisions: A Case Study," *Proc. Fifth ACM SIGKDD*, pp. 254-260, 1999.
[32] S. Zhang, J. Zhang, and C. Zhang, "EDUA: An Efficient Algorithm for Dynamic Database Mining," *Information Sciences*, vol. 177, pp. 2756-2767, 2007.
[33] Y.-S. Lee and S.-J. Yen, "Incremental and Interactive Mining of Web Traversal Patterns," *Information Sciences*, vol. 178, pp. 287-306, 2008.
[34] T.-P. Hong, C.-W. Lin, and Y.-L. Wu, "Incrementally Fast Updated Frequent Pattern Trees," *Expert Systems with Applications*, vol. 34, pp. 2424-2435, 2008.
[35] Y.-C. Li, J.-S. Yeh, and C.-C. Chang, "Isolated Items Discarding Strategy for Discovering High Utility Itemsets," *Data and Knowledge Eng.*, vol. 64, pp. 198-217, 2008.
[36] S.K. Tanbeer, C.F. Ahmed, B.-S. Jeong, and Y.-K. Lee, "CP-Tree: A Tree Structure for Single Pass Frequent Pattern Mining," *Proc. 12th Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD '08)*, pp. 1022-1027, 2008.

Chowdhury Farhan Ahmed received the BS and MS degrees in computer science from the University of Dhaka, Bangladesh, in 2000 and 2002, respectively. Currently, he is working toward the PhD degree in the Department of Computer Engineering, Kyung Hee University, South Korea. From 2003 to 2004, he was a faculty member at the Institute of Information Technology, University of Dhaka, Bangladesh. In 2004, he became a faculty member in the Department of Computer Science and Engineering, University of Dhaka, Bangladesh. His research interests include the areas of data mining and knowledge discovery.

Syed Khairuzzaman Tanbeer received the BS degree in applied physics and electronics and the MS degree in computer science from the University of Dhaka, Bangladesh, in 1996 and 1998, respectively. Currently, he is working toward the PhD degree in the Department of Computer Engineering, Kyung Hee University, South Korea. Since 1999, he has been a faculty member in the Department of Computer Science and Information Technology, Islamic University of Technology, Dhaka, Bangladesh. His research interests include data mining and knowledge engineering.

Byeong-Soo Jeong received the BS degree in computer engineering from Seoul National University, Korea, in 1983, the MS degree in computer science from the Korea Advanced Institute of Science and Technology, Korea, in 1985, and the PhD degree in computer science from the Georgia Institute of Technology, Atlanta, in 1995. In 1996, he joined the faculty at Kyung Hee University, Korea, where he is currently an associate professor at the College of Electronics and Information. From 1985 to 1989, he was on the research staff at Data Communications Corporation, Korea. From 2003 to 2004, he was a visiting scholar at the Georgia Institute of Technology, Atlanta. His research interests include database systems, data mining, and mobile computing.

Young-Koo Lee received the BS, MS, and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology, Korea. He is currently a professor in the Department of Computer Engineering, Kyung Hee University, Korea. His research interests include ubiquitous data management, data mining, and databases. He is a member of the IEEE, the IEEE Computer Society, and the ACM.