Received 21 April 2024, accepted 19 May 2024, date of publication 28 May 2024, date of current version 7 June 2024.

*Digital Object Identifier 10.1109/ACCESS.2024.3406562*

RESEARCH ARTICLE

# Incremental Top-k High Utility Pattern Mining and Analyzing Over the Entire Accumulated Dynamic Database

**CHANHEE LEE[1], HANJU KIM[1], MYUNGHA CHO[1], HYEONMO KIM[1], BAY VO[2], (Member, IEEE), JERRY CHUN-WEI LIN[3], (Senior Member, IEEE), PHILIPPE FOURNIER-VIGER[4], AND UNIL YUN[1]**

[1]Department of Computer Engineering, Sejong University, Seoul 05006, South Korea
[2]HUTECH University, Ho Chi Minh City, Vietnam
[3]Faculty of Automatic Control, Electronics and Computer Science, Department of Distributed Systems and IT Devices, Silesian University of Technology, 44-100 Gliwice, Poland
[4]College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China

Corresponding author: Unil Yun (yunei@sejong.ac.kr)

**ABSTRACT** Top-$k$ high utility pattern mining, which extracts the highest top-$k$ patterns that the users want to find, has been actively studied. Most previous studies in this domain have focused on static databases, where data insertions do not occur. In the real world, however, various applications continuously generate new data, and existing top-$k$ high utility pattern mining algorithms devised to process static databases cannot handle incremental databases. Although some methods can handle stream data, they have the limitation of processing a portion of the database rather than the entire accumulated database. In this paper, we suggest an efficient incremental mining method that discovers top-$k$ high utility patterns from the entire accumulated database. The proposed approach utilizes a list structure that stores minimal utility information required for the mining process and does not generate candidate itemsets. The suggested algorithm processes the incremental data with a single database scan and restructures the list for efficient mining. Moreover, four efficient threshold raising techniques along with a restoring technique are utilized to calculate the optimal threshold value in an accumulated incremental environment. The results of the experiments on runtime, memory, and scalability show that the suggested method efficiently processes the entire incremental database.

**INDEX TERMS** Data analysis, high utility pattern, incremental dynamic database, pattern analysis, Top-k.

## I. INTRODUCTION

As the application fields of information technology become more diverse, research on data analysis, such as clustering [1], classification [2], pattern mining [3], and pattern querying [4], [5], has been actively studied. In addition, the amount of generated data increases rapidly. Therefore, it is an important topic to analyze dynamic data efficiently. Pattern mining, a branch of data analysis, discovers hidden valuable patterns from huge databases. For example, there is unbalanced incomplete clustering [6], which is one of the latest techniques in the area of data clustering. It aims to handle information for datasets that have missing samples. Likewise, in a pattern mining area, there are some fields, such as uncertain pattern mining [7], that obtain useful patterns from uncertain or incomplete databases. Because of its suitability for analyzing databases, many applications, such as high utility pattern mining [8], high average utility pattern mining [9], erasable pattern mining [10], and differential sensor pattern mining [11], have been studied. Among them, high utility pattern mining expresses the utilities of itemsets by considering not only the occurrence of items but also their

The associate editor coordinating the review of this manuscript and approving it for publication was Marco Anisetti.

quantities and profits. High utility pattern mining is applied to various fields [12], such as e-commerce, medical, finance, and biomedical applications.

Traditional pattern mining methods receive a threshold from the user to extract result patterns. However, determining a proper threshold according to the characteristics of the dataset is a difficult task. Thresholds that are set too low extract an extravagant number of patterns and cause high runtime and memory usage. Therefore, the extracted patterns should be inspected again to find more meaningful patterns. In contrast, thresholds that are set too high discover few or no patterns. To solve this problem, top-*k* pattern mining algorithms have been proposed. Instead of the threshold, they receive *k*, the number of patterns to be extracted. Therefore, they can extract an ideal number of patterns without complicated threshold adjustments. Meanwhile, quantitative databases used in high utility pattern mining consist of the quantities and profits of items. Therefore, setting the threshold in high utility pattern mining is more difficult than in frequent pattern mining. To facilitate this, many top-*k* high utility pattern mining algorithms have been proposed. However, existing methods are not capable of processing dynamically accumulated incremental databases efficiently, where the data can only be scanned once.

As the IT industry has dramatically developed in recent years, the amount of accumulated data and the speed of accumulation are increasing rapidly. In such situations, real-time inserted incremental data must be scanned once. While various pattern mining algorithms [13], [14] have been proposed to process incremental databases, no incremental method for top-*k* high utility pattern mining has been studied, where the entire accumulated database is efficiently processed with one scan. Motivated by this, we propose an efficient Incremental Top-*k* High Utility Itemset mining algorithm (ITHUI) to discover top-*k* high utility patterns from accumulated incremental databases with a single scan using a list structure. To process the entire accumulated database in an incremental database situation, prior top-*k* high utility pattern mining methods scan the entire accumulated database twice and construct their data structures from scratch. However, ITHUI scans just the inserted database portion once and updates its data structure accordingly through a restructuring procedure. It also introduces a new database restoring process, which enables efficient threshold raising without the rescan of previous data. Thus, ITHUI has the advantage over existing methods in that it does not suffer from overheads in scanning massive amounts of previously accumulated data. This is especially relevant for rapidly accumulating databases in the IT industry and is magnified as the size of the database increases. To the best of our knowledge, this work is the first research on top-*k* high utility pattern mining that analyzes the entire accumulated incremental databases with a single incremental database scan. The contributions of this paper are as follows.

- An efficient method that mines top-*k* high utility patterns from accumulated incremental databases with single incremental database scans is proposed, which reflects new data into efficient list structures with minimal utility information.
- Reconstruction and restoring techniques for incremental database processing, threshold raising, and efficient mining are suggested in top-*k* high utility pattern mining.
- Runtime, memory usage, and scalability performance of the proposed algorithm measured using real and synthetic datasets show improvement over existing top-*k* high utility pattern mining methods.

The remaining sections of this paper are organized as follows. Section II presents the related work and specifies the differences between the proposed approach and the previous studies. Section III presents the problem definition and explains the procedures of the proposed approach for efficiently handling incremental databases. In Section IV, the results of the performance evaluations of the proposed algorithm are analyzed in detail. Finally, Section V concludes this paper.

## II. RELATED WORK

### A. HIGH UTILITY PATTERN MINING

Not only the occurrence of items but also the quantity and profit of items are handled in high utility pattern mining. Because of this advantage, various studies on utility pattern mining, such as on-shelf utility mining [15], cross-level high-utility itemset mining [16], and utility sequence mining [17], have been conducted. Furthermore, it is applied to real-world database analysis such as hill climbing [18] and longitudinal human studies [19]. However, unlike frequent pattern mining, high utility pattern mining has the limitation that it does not satisfy the anti-monotone property. To solve this problem, the two-phase algorithm [20] was proposed. It proposed transaction weighted utility (*TWU*) as a utility upper-bound to increase mining efficiency. However, since it performs in a generation-and-test manner with multiple database scans, many candidate patterns are created, and the performance is degraded. Afterward, tree-based algorithms, UP-Growth [21] and UP-Growth+ [22], reduced the number of candidate patterns through improved upper-bounds. Nevertheless, these tree-based high utility pattern mining methods still consume a lot of runtime and memory for generating candidate patterns. The tree structure is constructed through two database scans, but an additional database scan is required to extract result patterns among the candidate patterns. To overcome this limitation, HUI-Miner [23], a list-based algorithm, was proposed. HUI-Miner stores the actual utilities of items in a list structure and extracts high utility patterns without generating candidate patterns. Therefore, an extra database scan is not needed. Due to the benefits of the list structure, various list-based algorithms, such as FHM [24], HUP-Miner [25], and HMiner [26], were proposed.

Meanwhile, high utility pattern mining is applied in retail analysis [27] or database analysis of IoT systems [28], [29]. In these fields, data are continuously generated and accumulated. Therefore, real-time processing and efficient analysis of accumulated databases are important. Furthermore, high utility pattern mining, considering dynamic environments, has been actively researched in various application areas. ILDHUP [30] mines the high utility patterns, considering the arrival time of transactions. DSHUP [31] and RHUPS [32] extract high utility patterns from recent transactions through the damped window and sliding window concepts, respectively. Methods that employ the pre-large concept [33], [34] can reduce the number of mining processes in a dynamic environment by extracting large and pre-large patterns.

### B. TOP-K HIGH UTILITY PATTERN MINING

Determining an optimal threshold in quantitative databases where items are expressed as quantities and profits is difficult for users. Therefore, TKU [35], a top-*k* high utility pattern mining algorithm that receives the number of result patterns instead of the threshold, was proposed. TKU extracts potential top-*k* high utility patterns in phase 1 and scans the database again in phase 2 to find actual top-*k* high utility patterns. Thus, it creates much more potential patterns than actual patterns, which leads to inefficient runtime and memory usage. To overcome this problem, REPT [36], which uses improved threshold raising strategies, was proposed. However, REPT still generates many potential patterns compared to the actual patterns because it is a tree-based algorithm. Therefore, TKO [37] and KHMC [38], top-*k* high utility algorithms using a list structure, were proposed. TKO and KHMC no longer generate candidate patterns because their list structures store the actual utilities of items. After that, THUI [39] improved mining efficiency by using the list structures and proposing new threshold raising strategies.

Top-*k* high-utility pattern mining has been utilized in diverse areas. HUST [40] and TKUS [41] extract high utility patterns from sequence databases. LMSpan [42] combines top-*k* high utility pattern mining with the concept of time interval-based events with duration. TONUP [43] optimizes the data structure of d2HUP [44] for top-*k* high utility pattern mining. PTM [45] analyzes massive data using a prefix partitioning method. TKHUIM-GA [46] is an approach that utilizes genetic algorithms to mine top-*k* high utility patterns. TopHUI [47], THN [48], and TKN [49] find the top-*k* high utility patterns in quantitative databases containing negative utilities. FTKHUIM [50] extracts top-*k* high utility patterns by using new strategies. To quickly raise the threshold, it adopts a global priority queue and hash map. However, the above traditional top-*k* high utility pattern mining methods are designed to process static databases, which are fixed and do not include incremental data. Therefore, they are unsuitable for handling incremental environments where data are continuously added and accumulated. Sliding window-based top-*k* high utility pattern mining approaches [51], [52] were proposed to process stream data. However, since the sliding window model only considers recent data, they remove previous data and only address the most recent data in the window size. On the other hand, the proposed algorithm handles the entire accumulated incremental data while calculating the optimal threshold to extract top-*k* high utility patterns efficiently.

**TABLE 1.** Example database.

| *DB* | *TID* | *Transaction* | *TU* |
|---|---|---|---|
| *Original DB* | $T_1$ | $(b, 1)\ (c, 2)\ (d, 1)\ (f, 4)$ | 19 |
| | $T_2$ | $(a, 3)\ (e, 2)$ | 8 |
| | $T_3$ | $(a, 1)\ (b, 2)\ (d, 3)\ (f, 2)$ | 23 |
| | $T_4$ | $(a, 2)\ (b, 1)\ (c, 4)\ (e, 3)$ | 19 |
| | $T_5$ | $(b, 3)\ (c, 2)\ (d, 1)\ (e, 2)\ (f, 3)$ | 27 |
| | $T_6$ | $(a, 1)\ (d, 2)\ (e, 2)$ | 10 |
| *DB1+* | $T_7$ | $(a, 3)\ (b, 1)\ (c, 3)\ (d, 1)\ (e, 1)$ | 20 |
| | $T_8$ | $(a, 2)\ (b, 2)\ (c, 1)\ (d, 1)$ | 17 |

**TABLE 2.** Profit table.

| *Item* | *a* | *b* | *c* | *d* | *e* | *f* |
|---|---|---|---|---|---|---|
| *Profit* | 2 | 4 | 2 | 3 | 1 | 2 |

## III. MINING TOP-*K* HIGH UTILITY PATTERNS FROM ACCUMULATED INCREMENTAL DATABASES

### A. PRELIMINARIES

For general definitions of top-*k* high utility pattern mining, previous studies [35], [36] are referred to. A transaction database $D = \{T_1, T_2, \ldots, T_n\}$ consists of $n$ transactions. A set of $m$ different items appearing in the database is expressed as $I = \{i_1, i_2, \ldots, i_m\}$. Each transaction has a unique identifier, *TID*, and for any transaction $T_k$, $T_k \subseteq I$ is always true. An item $i_x$ in $T_k$ has the information of quantity which is called internal utility and notated as $iu(i_x, T_k)$. Table 1 shows the database used in the examples of the paper. The example database is composed of an original database and *DB1+*, where the latter is an incremental database. Table 2 is the profit table used in the examples. It consists of items and their profits. The profit value is the external utility of an item and is notated as $eu(i_x)$. In a transaction $T_k$, the utility of an item $i_x$, $U(i_x, T_k)$, is calculated as $iu(i_x, T_k) \times eu(i_x)$. Next, assume a pattern $X = \{i_1, i_2, \ldots, i_x\}$. $U(X, T_k)$, the utility of $X$ in transaction $T_k$, is defined as $\sum U(i_x, T_k)$, where $i_x \in X$ and $X \subseteq T_k$. In addition, $U(X)$, the utility of $X$ in a database $D$ is represented as $\sum U(X, T_k)$, where $X \subseteq T_k$ and $T_k \in D$.

*Definition 1 (Transaction Utility):* The transaction utility of a transaction $T_k$, which means the sum of the utilities of items in $T_k$, is expressed as $TU(T_k)$ and calculated as follows:

$$TU(T_k) = \sum U(i_x, T_k),\ \ where\ i_x \in T_k. \quad (1)$$

*Definition 2 (Transaction Weighted Utility):* The transaction weighted utility of a pattern $X$ is expressed as $TWU(X)$, which is the sum of the transaction utilities of the transactions where $X$ occurs. $TWU(X)$ is calculated by the formula below.

$$TWU(\mathrm{X}) = \sum TU(T_k),\ \ where\ X \subseteq T_k \text{ and } T_k \in D. \quad (2)$$

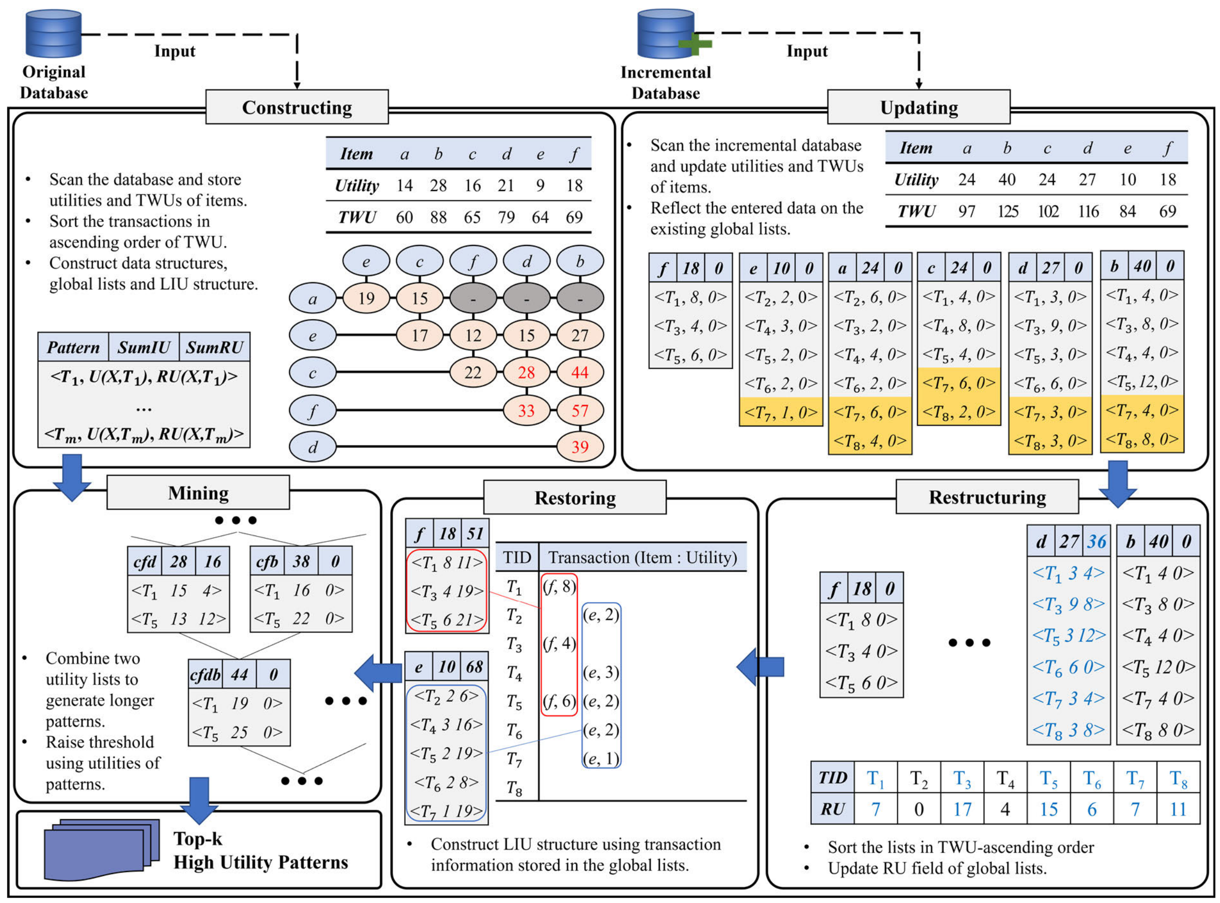

Original Database
Input
Constructing
Incremental Database
Input
Updating
Scan the database and store utilities and TWUs of items.
Sort the transactions in ascending order of TWU.
Construct data structures, global lists and LIU structure.
Scan the incremental database and update utilities and TWUs of items.
Reflect the entered data on the existing global lists.
Mining
Combine two utility lists to generate longer patterns.
Raise threshold using utilities of patterns.
Top-k High Utility Patterns
Restoring
Construct LIU structure using transaction information stored in the global lists.
Restructuring
Sort the lists in TWU-ascending order
Update RU field of global lists.

FIGURE 1. **Overall process of the proposed approach.**

*Definition 3 (Processing order of Items):* For an efficient mining procedure, items in a transaction are sorted in the ascending order of *TWU*. The *TWU*-ascending order considering incremental databases is defined as the processing order, where $\prec$ indicates the order of items.

*Definition 4 (Remaining Utility of a Pattern):* For a pattern $X$ and a transaction $T_k$ containing $X$, the set of items located after $X$ in $T_k$ is defined as $T_k\backslash X$. In $T_k$, the remaining utility of $X$ is expressed as $RU(X, T_k)$, and it is the sum of the utilities of items included in $T_k\backslash X$. In addition, the remaining utility of $X$ in database $D$ is expressed as $RU(X)$, and it is the sum of $RU(X, T_k)$ in the transactions where $X$ occurs.

$$RU(X, T_k) = \sum U(i_x, T_k),\ where\ i_x \in (T_k \setminus X). \tag{3}$$

$$RU(X) = \sum RU(X, T_k),\ where\ X \subseteq T_k\ and\ T_k \in D. \tag{4}$$

*Definition 5 (High Utility Pattern and Minimum Utility):* A high utility pattern is a pattern whose utility is not less than the minimum utility threshold (*minutil*). In top-*k* high utility pattern mining, *minutil* is divided into two types, current *minutil*, and optimal *minutil*. The current *minutil* indicates the threshold that is increased through threshold raising strategies. The optimal *minutil* represents the minimum utility threshold that mines $k$ result patterns for the user given $k$ value.

*Definition 6 (Top-k high utility patterns):* The $k$ patterns with the largest utility in a database $D$ are defined as top-*k* high utility patterns. That is, the top-*k* high utility pattern set contains patterns with utilities no less than the optimal *minutil*.

*Definition 7 (Ordered and Contiguous Pattern):* An ordered and contiguous pattern, $(i_x - i_y)$, is defined as a pattern that consists of the entire sequence of adjacent items from $i_x$ and $i_y$ when the items are sorted in processing order.

*Definition 8 (LIU structure [39]):* LIU (Leaf Itemset Utility) structure is a triangular matrix that stores the utility of all ordered and contiguous patterns, and it is used to raise the current *minutil*. The utility value of $(i_x - i_y)$ stored in the LIU structure is denoted as $LIU(x, y)$, and is defined as $LIU\left(i_x, i_y\right) = U\left(\left(i_x - i_y\right)\right) = \sum_{(i_x - i_y) \subseteq T} U\left(\left(i_x - i_y\right), T\right)$. For example, in sorted items $I = \{i_1, i_2, \ldots, i_p\}$, $LIU(i_3, i_6)$ means the utility value of the pattern with contiguous items $i_3$ to $i_6$, stored in the LIU structure. In other words, $LIU\ (i_3, i_6) = U\ (i_3 - i_6) = U(\{i_3 i_4 i_5 i_6\})$.

*Definition 9 (Underestimated Utility of a Pattern [39]):* Assume a pattern $X$ and its sub-pattern, $Y$. Pattern $X - Y$ is a set of items in $X$, excluding items in $Y$. The underestimated utility of pattern $X - Y$ is denoted as $UE(X - Y)$ and is defined as $UE(X - Y) = U(X) - \sum_{i \in Y} U(i)$. For example, when $X = \{i_3 i_4 i_5 i_6\}$ and $Y = \{i_4\}$, $UE(X - Y)$ is an underestimated utility of pattern $X - Y = \{i_3 i_5 i_6\}$.

## B. OVERALL PROCESS OF INCREMENTAL TOP-K HIGH UTILITY PATTERN MINING

Fig. 1 presents the overall process of ITHUI. ITHUI first scans an original database and calculates the utility and *TWU* values of items. At this time, RIU strategy for raising a threshold is used by using the utilities of the items. Then, items are sorted in *TWU*-ascending order, and list structures storing utility information of items and a LIU structure used for raising the threshold are constructed through the second database scan. Although *TWU* value of an item is less than current *minutil*, the item is not pruned because invalid patterns can change into valid patterns as the incremental data are inserted. Therefore, global lists for all items should be managed. However, unpromising patterns that cannot become the top-*k* high utility pattern are excluded from the construction of LIU, restructuring process, and mining process in order to improve performance. Utilizing the LIU structure, the threshold is raised again. Then, ITHUI creates longer patterns and mines the top-*k* high utility patterns. During the mining process, the threshold is increased by the actual utility of the patterns. When the incremental database is inserted, ITHUI scans newly inserted data and updates the global lists. Then, LIU structure is constructed using information stored in the global lists, and *minutil* is calculated again. Through the restructuring and mining processes, top-*k* high utility patterns are found in the dynamic databases.

## C. CONSTRUCTING DATA STRUCTURES FOR INCREMENTAL TOP-K HIGH UTILITY PATTERN MINING

One utility list corresponds to a pattern and consists of *Name*, *SumIU*, *SumRU*, and a set of tuples. *Name* represents the corresponding pattern, *SumIU* means the total utility of the pattern in transactions, and *SumRU* signifies the sum of remaining utilities of the pattern in transactions. The utility list of a pattern $X$ is denoted as $UL(X)$. Each tuple is composed of a transaction ID, a utility of a pattern in the transaction, and a remaining utility of a pattern in the transaction and denoted as $< TID, U(X, TID), RU(X, TID) >$. ITHUI calculates the utility and *TWU* values of the items by scanning the original database once. ITHUI uses the utilities of items to raise the threshold by RIU strategy [36]. If the number of items is more than or equal to $k$ inputted by a user, *minutil* is set to the $k$-th highest value among utilities of the items. For example, when $k$ is five, since the utility values of the items in the database are $\{(a, 14), (b, 28), (c, 16), (d, 21), (e, 9), (f, 18)\}$, *minutil* increases to 14, which is the utility of the item $\{a\}$. The *TWU*-ascending order of the items is as follows: $a \prec e \prec c \prec f \prec d \prec b$. Note that an item whose *TWU* is less than *minutil* should not be pruned because top-*k* high utility patterns can be generated from the pruned item as the incremental data are newly inserted. However, unpromising items, whose *TWU* values are less than *minutil* raised through the RIU strategy, can be excluded in the mining process. As a result, the algorithm can find valid patterns more efficiently.

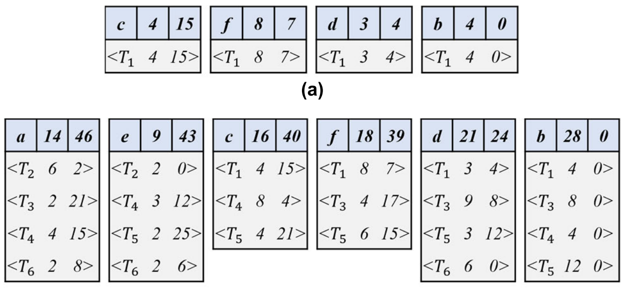


**FIGURE 2.** Construction of global utility lists: (a) Global lists after processing $T_1$, (b) Global lists after processing $T_6$.

*Lemma 1:* In the top-*k* high utility pattern mining that processes the dynamic environments, if a global list for an unpromising item in the original database is pruned, an inaccurate pattern may be discovered as the top-*k* high utility pattern when an incremental database is newly added.

*Rationale:* Let an original database be *ODB* and an incremental database be *IDB*. $TKH = \{P_1, P_2, \ldots, P_k\}$ is a set of top-*k* high utility patterns discovered from *ODB*, and $TKH = \{P'_1, P'_2, \ldots, P_k\}$ is a set of top-*k* high utility patterns extracted from the entire database, including *ODB* and *IDB*, when the global lists for the unpromising items were pruned in *ODB*. If *TKH* and $TKH'$ are sorted in utility descending order of the patterns, the minimum utility threshold in *ODB* is $U(P_k)$, and the minimum utility threshold in the entire database is $U(P'_k)$. In *ODB*, if *TWU* of an item $\{i_x\}$ is less than $U(P_k)$, the utility of any super pattern $\{X\}$ of $\{i_x\}$ is also less than $U(P'_k)$. In other words, the item and super patterns cannot become the result patterns in *ODB*. However, when the incremental data are inserted, the utility of a pattern may increase because the pattern may appear in the added transactions. After the incremental data are added, the utility of the pruned pattern $\{X\}$ is the total sum of utilities of $\{X\}$ in *ODB* and *IDB*, which is denoted as $U(X) = U(X, ODB) + U(X, IDB)$. However, when $TKH'$ is extracted, the utility of the pruned pattern in *ODB* may be not considered because the global list for the unpromising item was pruned. If $U(X, ODB)$ is greater than $U(P'_k) - U(X, IDB)$, the pattern $\{X\}$ can become the top-$k$ high utility pattern instead of the pattern $P'_k$. As a result, $TKH'$ may contain the imprecise pattern because the global lists for the unpromising items are not managed in memory. ■

ITHUI constructs global utility lists by scanning the sorted transactions from the last item to the first item. If a global utility list of the processed item has not yet been created, a utility list is generated, and a tuple storing TID, utility, and remaining utility is inserted into the list. If the global utility list for the processed item already exists, ITHUI does not create a utility list, makes a tuple, and just adds it to the constructed list. Since list construction begins with the last item, the remaining utility can be efficiently obtained by accu-

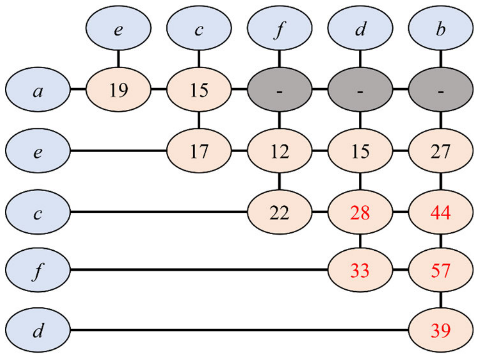


**FIGURE 3.** Constructed LIU structure.

mulating the utilities of the previous items. For efficiently calculating the remaining utility, ITHUI utilizes one variable that is reused for each transaction to store the utility sum of previous items. In this way, the *RU* values of list tuples can be calculated without the need to revisit previously processed data in the global list.

*Example 1:* Fig. 2 presents the process of generating the global utility lists for items in transactions which are sorted in *TWU*-ascending order. The construction process starts with the last item $\{b\}$ in the sorted transaction $T_1$. Since there is no utility list for the item $\{b\}$, $UL(b)$ is configured, and a tuple for $T_1$ is inserted into the list. $T_1$ is stored in the TID field, the utility field is calculated as $U(b, T_1) = iu(b, T_1) \times eu(b) = 1 \times 4 = 4$, and the remaining utility field is initialized to zero because there is no item behind $\{b\}$. Next, ITHUI processes the item $\{d\}$. A utility list for $\{d\}$ is also newly generated, and a tuple, $< T_1, 3, 4 >$, is added to $UL(d)$. The remaining utility of $\{d\}$ is set to the utility of $\{b\}$. This remaining utility value is remembered for future *RU* calculations. The remaining items $\{c\}$ and $\{f\}$ are processed in the same manner. The configured global lists after processing $T_1$ are illustrated in Fig. 2(a). Fig. 2(b) shows the configured global lists after the construction process is completed for all transactions in the original database.

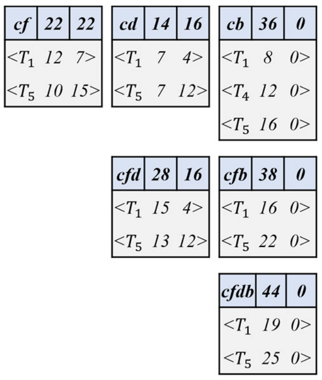
During the second database scan, ITHUI not only constructs the global lists but also creates the LIU structure [39]. While the algorithm traverses a sorted transaction during the database scan, it calculates the utility values of patterns consisting of contiguous items and stores them in the LIU structure. Whenever reading a transaction, the global utility lists are constructed, and at the same time, the values are accumulated into the LIU structure. Fig. 3 shows the constructed LIU structure after processing $T_6$. After constructing and updating the LIU structure, ITHUI utilizes the LIU-E strategy [39] that uses the utility values in the LIU structure to raise the threshold. ITHUI stores the utility values in the LIU structure, which are greater than or equal to *minutil*, into a priority queue. The element stored in the priority queue has a higher priority as the utility is larger. By not managing low priority values, the size of the priority queue remains $k$. For example, when $k$ is set to five, the values stored in the priority queue among the utility values shown in Fig. 3 are 57, 44, 39, 33, and 28. Then, *minutil* is raised to 28. Then, ITHUI raises the threshold using the LIU-LB strategy [39]. ITHUI raises the threshold by underestimating the utility of the sub-patterns of the ordered and contiguous patterns stored in the LIU structure according to Definition 9. The patterns are generated by eliminating items except the first and last items of the patterns managed in the LIU structure. The number of items to be removed is set to three or fewer so that a lot of patterns are created. For example, $UE(fb) = LIU(f, b) - U(d) = 57 - 21 = 36$. Since $UE(fb)$ is greater than *minutil*, the value is added to the priority queue. Then, values in the priority queue are 57, 44, 39, 36, and 33, and *minutil* is raised to 33. *Minutil*, increased through the LIU-E and LIU-LB strategies, can reduce the number of patterns that the algorithm should combine in the mining process. If *minutil* quickly increases to an optimal value, because the algorithm can predict more unpromising patterns in the mining process, it can skip the combining process for invalid patterns. In other words, the LIU-E and LIU-LB strategies increase *minutil* to a value close to the optimal value to optimize the mining process.

## D. MINING TOP-K HIGH UTILITY PATTERNS FROM GLOBAL LIST STRUCTURES

When a mining request is received, ITHUI conducts the mining process using the constructed global lists and current *minutil*. The mining process can be performed after the global lists are arranged in the ascending order of *TWU*s and the remaining utilities in the lists are updated. Let $\{i_1 \prec i_2 \prec \ldots \prec i_n\}$ be the current *TWU*-ascending order. The mining process of ITHUI is performed in DFS-manner. ITHUI combines the patterns with the same prefix and length. After processing all patterns that can be expanded from the item $\{i_1\}$, the prefix is changed into the next item, such as $\{i_2\}$, and pattern combining is performed recursively. The mining process is completed when there is no pattern that can be expanded.

*Lemma 2:* In the mining process, if an upper-bound of a pattern $X$, which is denoted as $UB(X)$ and defined as $U(X) + RU(X)$, is less than the current minimum utility threshold, the pattern and super patterns cannot become the top-$k$ high utility patterns.

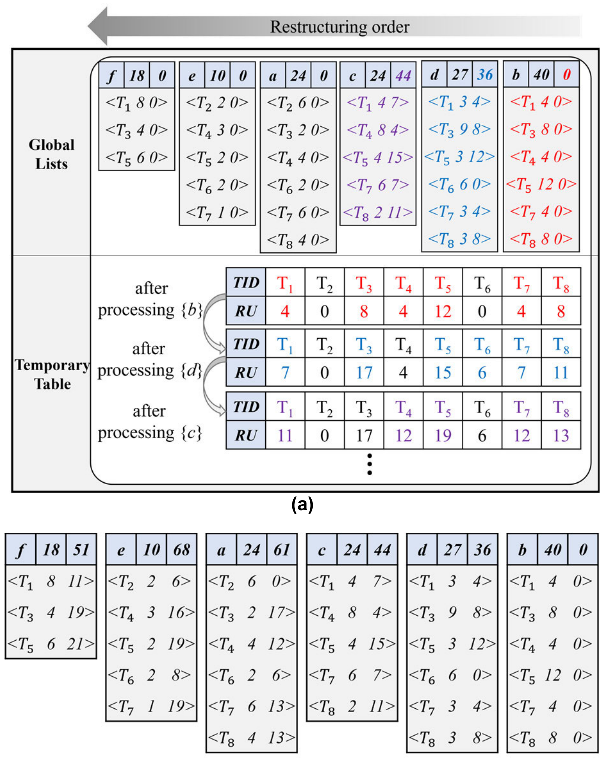
*Rationale:* An upper-bound of a pattern $\{X\}$ represents the maximum utility that a super pattern $\{X'\}$ of $\{X\}$ can have. Namely, $UB(X)$ is greater than or equal to the $U(X)$ and $U(X'P)$. Since the top-$k$ high utility pattern mining approaches increase the minimum utility threshold from zero to the optimal minimum utility threshold that extracts $k$ result patterns, the current minimum utility threshold is always less than or equal to the optimal minimum utility threshold during the mining process. If $UB(X)$ is less than the current minimum utility threshold, $U(X)$ and $U(X')$ are also less than the optimal minimum utility threshold. As a result, even if a pattern with an upper-bound lower than the current minimum utility threshold is pruned in the mining process, pattern loss does not occur. ■



cf 22 22: <T1 12 7> <T5 10 15>
cd 14 16: <T1 7 4> <T5 7 12>
cb 36 0: <T1 8 0> <T4 12 0> <T5 16 0>
cfd 28 16: <T1 15 4> <T5 13 12>
cfb 38 0: <T1 16 0> <T5 22 0>
cfdb 44 0: <T1 19 0> <T5 25 0>

**FIGURE 4.** Mining process performed on super patterns of {c}.

When a utility list for a pattern is visited, it is immediately determined whether the pattern is a top-*k* high utility pattern for the current *minutil*. This is achieved by checking the *sumIU* value in the utility list. Since this value is the pattern's actual utility, top-*k* high utility patterns can be discovered without candidate generation using the utility list structure. Extensions of patterns during mining are performed by combining two utility lists. Suppose a pattern $\{X\}$ and a pattern $\{Y\}$, which appears after $\{X\}$. Before combining $UL(X)$ with another utility list, $UB(X)$ is calculated to determine whether $X$ is extendable. This is done in constant time without traversing tuples of $UL(X)$ because *sumIU* and *sumRU* are stored in $UL(X)$. When combining $\{X\}$ and $\{Y\}$, the utility of the prefix pattern $\{P\}$ must be considered to avoid the duplication of the utility. When the length of $\{X\}$ and $\{Y\}$ is one, the prefix pattern $\{P\}$ is empty, and its utility is zero. Meanwhile, the remaining utility of the combined pattern $\{XY\}$ is set to the remaining utility of the pattern $\{Y\}$. For each pair of tuples with the same TID in both $UL(X)$ and $UL(Y)$, ITHUI creates a tuple $< TID, U(X, TID) + U(Y, TID) - U(P, TID), RU(Y, TID) >$ and inserts it into $UL(XY)$. Tuple pairs can be identified by traversing the tuples of each utility list in order just once, since the tuples are sorted in TID ascending order. During the construction of the utility list for the combined pattern, ITHUI reduces the upper-bound of long patterns to improve the mining performance by LA-Prune [25]. If the reduced upper-bound is less than *minutil*, ITHUI stops the expansion process of the pattern $\{XY\}$ and proceeds to the next pattern. The utility list structure makes LA-Prune trivial and efficient by managing the $U$ and $RU$ values of patterns per transaction, eliminating the need for additional calculations. While ITHUI performs the above mining process recursively, the RUC strategy [37] is applied, where *minutil* is raised using the actual utility of the pattern. ITHUI inserts the pattern whose utility is not less than *minutil* into the priority queue. If the size of the priority queue is greater than or equal to $k$, *minutil* increases to the $k$th utility value in the priority queue. During the mining process, if *minutil* is raised using the RUC strategy, because the algorithm can predict unpromising patterns using an upper-bound, the greater the *minutil*, the more quickly



Restructuring order
Global Lists
f 18 0: <T1 8 0> <T3 4 0> <T5 6 0>
e 10 0: <T2 2 0> <T4 3 0> <T5 2 0> <T6 2 0> <T7 1 0>
a 24 0: <T2 6 0> <T3 2 0> <T4 4 0> <T6 2 0> <T7 6 0> <T8 4 0>
c 24 44: <T1 4 7> <T4 8 4> <T5 4 15> <T7 6 7> <T8 2 11>
d 27 36: <T1 3 4> <T3 9 8> <T5 3 12> <T6 6 0> <T7 3 4> <T8 3 8>
b 40 0: <T1 4 0> <T3 8 0> <T4 4 0> <T5 12 0> <T7 4 0> <T8 8 0>
Temporary Table
after processing {b}
TID T1 T2 T3 T4 T5 T6 T7 T8
RU 4 0 8 4 12 0 4 8
after processing {d}
TID T1 T2 T3 T4 T5 T6 T7 T8
RU 7 0 17 4 15 6 7 11
after processing {c}
TID T1 T2 T3 T4 T5 T6 T7 T8
RU 11 0 17 12 19 6 12 13
(a)
f 18 51: <T1 8 11> <T3 4 19> <T5 6 21>
e 10 68: <T2 2 6> <T4 3 16> <T5 2 19> <T6 2 8> <T7 1 19>
a 24 61: <T2 6 0> <T3 2 17> <T4 4 12> <T6 2 6> <T7 6 13> <T8 4 13>
c 24 44: <T1 4 7> <T4 8 4> <T5 4 15> <T7 6 7> <T8 2 11>
d 27 36: <T1 3 4> <T3 9 8> <T5 3 12> <T6 6 0> <T7 3 4> <T8 3 8>
b 40 0: <T1 4 0> <T3 8 0> <T4 4 0> <T5 12 0> <T7 4 0> <T8 8 0>
(b)

**FIGURE 5.** Restructuring process: (a) Restructuring process performed on items {b}, {d}, and {c}, (b) Restructured global lists.

it can obtain valid patterns. Finally, ITHUI obtains optimal *minutil* that extracts the top-*k* high utility patterns when the mining process is finished, and patterns in the priority queue are provided to the user as the top-*k* high utility patterns.

*Example 2:* Fig. 4 presents the utility lists of the super patterns of item $\{c\}$. Since the upper-bound of the pattern $\{cf\}$, $UB(cf) = U(cf) + RU(cf) = 22 + 22 = 44$, is greater than *minutil*, patterns $\{cfd\}$ and $\{cfb\}$ are generated by extending $\{cf\}$. When $\{cf\}$ is combined with $\{cd\}$, tuples for $T_1$ and $T_5$ are inserted into the utility list for the pattern $\{cfd\}$ because both patterns are included in the transactions $T_1$ and $T_5$. The utility of $\{cfd\}$ for each transaction is calculated as follows, by considering the utility of the prefix pattern $\{c\}$: $U(cfd, T_1) = U(cf, T_1) + U(cd, T_1) - U(c, T_1) = 12 + 7 - 4 = 15$ and $U(cfd, T_5) = U(cf, T_5) + U(cd, T_5) - U(c, T_5) = 10 + 7 - 4 = 13$. The remaining utilities of $\{cfd\}$ are set to the remaining utilities of $\{cd\}$. Similarly, patterns $\{cf\}$ and $\{cb\}$ are combined. The utility list for the pattern $\{cfb\}$ is created, and tuples, $< T_1, 16, 0 >$ and $< T_5, 22, 0 >$, are added to the list. The pattern $\{cfd\}$ is not the top-*k* high utility pattern because its utility is less than *minutil*. However, the upper-bound of $\{cfd\}$ is greater than *minutil*, $\{cfdb\}$ is created by extending $\{cfd\}$. Since utilities of $\{cfb\}$ and $\{cfdb\}$ are greater than *minutil*, they are added to the priority queue. At this time, utility values stored in the priority queue are 44, 38, 36, and 33. The number of values stored in the priority queue is less than $k$, which was set to five, so *minutil* has not increased. After finishing the

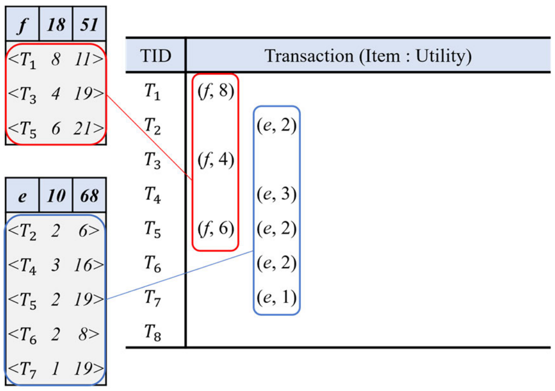


**FIGURE 6.** **Restoring process.**

mining for the original DB, the patterns in the priority queue are ({*fdb*} : 57) , ({*cfdb*} : 44) , ({*fb* : 42}), ({*db* : 39}) and ({*cfb* : 38}). The algorithm raises *minutil* to 38, which is the optimal *minutil*, using the RUC strategy.

## *E. UPDATING AND RESTRUCTURING GLOBAL LISTS FOR HANDLING INCREMENTAL DATA*

When incremental data occur, ITHUI restructures the configured global lists. ITHUI reflects the data on the global lists, obtains *TWU*-ascending order again, sorts the global lists in new *TWU*-ascending order, and updates the remaining utility field of utility lists. ITHUI updates the global lists by scanning the incremental data only once. Item information is stored in the form of a tuple. Each tuple stores TID, the utility of the pattern, and the remaining utility of the pattern. The remaining utility field is initialized to zero because the processing order may be varied depending on the transaction insertion. Reflecting incremental data in the list is the same as the construction process. Then, ITHUI performs the restructuring process to update the remaining utility values. A temporary table storing pairs of TID, and a remaining utility are utilized. The remaining utility values in the temporary table are initially set to zero. The restructuring process is conducted by reading the global lists from behind. While traversing tuples in the list, ITHUI finds each pair of a tuple and an element of the array with the same TID, updates the remaining utility of the tuple to the remaining utility in the array, and increases the remaining utility in the array by the utility of the tuple. When the above tasks are performed on all global lists, the restructuring process is completed.

*Example 3:* Fig. 5 displays the process of updating the remaining utility of the utility lists sorted in the new *TWU*-ascending order. The restructuring process begins with the last item $\{b\}$. The item $\{b\}$ appears in transactions $T_1$, $T_3$, $T_4$, $T_5$, $T_7$, and $T_8$. Since the remaining utility values in the temporary table are initialized to zero, all remaining utilities of $UL(b)$ are set to zero. Then, the remaining utility values in the temporary array increase by the utilities of $UL(b)$ to update the remaining utilities of the next item. The remaining utility values for $T_1$, $T_3$, $T_4$, $T_5$, $T_7$, and $T_8$ in the temporary table become 4, 8, 4, 12, 4, and 8, respectively. Thereafter, ITHUI sequentially processes the tuples of $UL(d)$. TID of the first tuple is $T_1$, and the remaining utility related to $T_1$ in the array is 4. Therefore, the remaining utility of $\{d\}$ in $T_1$ is set to 4. The remaining utility in the array increases by $U(d, T_3)$ to 7. TID of the second tuple is $T_3$, and the remaining utility for $T_3$ in the array is 8. Thus, $UL(d, T_3)$ is set to eight, and the remaining utility in the array becomes 17. In this way, all tuples of $UL(d)$ are updated. The remaining utility fields of $UL(c)$ are calculated in the same manner as above. Fig. 5(a) shows the restructuring process on the items $\{b\}$, $\{d\}$, and $\{c\}$. The above process is repeated to all global lists except the item whose *TWU* is less than *minutil*. The restructured global lists are presented in Fig. 5(b).

ITHUI recalculates the threshold after handling the incremental data in order to find the accurate top-*k* high utility patterns efficiently. ITHUI raises the threshold using the updated utility values of the items. In addition, the LIU structure used to increase the threshold is configured utilizing the information in the global utility lists. In an incremental environment, because the algorithm has to reflect information for a database in a single scan, it is efficient to use global utility lists for configuring the LIU structure. Since each utility list stores complete utility information in the transactions where the item appears and is sorted in *TWU*-ascending order, the LIU structure can be constructed using the global lists. Tuples of the utility list are traversed, and each pair of the name and the utility of the item is stored in a temporary transaction list. Here, items whose *TWU* value is less than the current *minutil* are skipped to minimize utility information management. This can be achieved efficiently since tuples are organized according to their corresponding item in the utility list structure. Fig. 6 shows the process of restoring transaction information from the restructured global lists. After restoring the database with a temporary transaction list, it is traversed to construct the LIU structure. The constructed LIU structure is then used to perform LIU-E and LIU-LB strategies for threshold raising. Finally, the temporary transaction lists and the LIU structure are deleted after raising the threshold.

*Lemma 3:* In accumulated incremental database environments, the incremental method demonstrates higher theoretical efficiency compared to static methods in top-*k* high utility pattern mining.

*Rationale:* Let an accumulated incremental database be *DB*, and a database that is to be newly inserted in *DB* be *db*. Both static and incremental methods process the entire accumulated database, $DB \cup db$, to mine top-*k* high utility patterns. However, static methods scan the entire accumulated $DB \cup db$ twice to prepare the data structure for mining, whereas the incremental method scans just the inserted *db* once to accomplish the same task. When the data structure is ready, both static and incremental methods proceed with the mining procedure. The incremental method may require the scanning of its data structure in memory for the data structure optimizations that happen during the two database scans of static methods. Nonetheless, this data structure scanning is

```
Algorithm 1 Insertion
Input: A set of the global utility lists, $GL$
An incremental data, $I_d$
Output: A set of the restructured global utility lists, $GL$
Variable: A temporary array, $TA$; A minimum utility threshold, $\delta$;
A triangular matrix, $LIU$; A temporary transaction list, $T_k-list$;

1.  For each transaction $T_k$ in $I_d$
2.      For each item $i$ in $T_k$
3.          Update $U(i)$ and $TWU(i)$
4.          If $UL_i$ does not exist in $GL$
5.              $UL_i$ = Create a new global utility list for $i$
6.          Insert a tuple $< T_k, U(i,T_k), 0 >$ into $UL_i$
7.  Raise $\delta$ to the $k$-th highest value among utilities of items
8.  Sort $GL$ in $TWU$-ascending order
9.  For each utility list $UL_i$ in $GL$// Restructure (Reverse order)
10.     For each tuple $T$ in $UL_i//T = \{T_k, U(i,T_k), RU(i,T_k)\}$
11.         Set $RU(i,T_k)$ of $T$ to remaining utility stored in $TA$
12.         Increase $TA(T_k)$ by $U(i,T_k)$ of $T$
13.     Update $SumIu$ and $SumRu$ of $UL_i$
14. For each utility list $UL_i$ in $GL$// Restoring
15.     If $TWU(i) < \delta$
16.         continue
17.     For each tuple $T$ in $UL_i$
18.         Insert a pair $(i, U(i,T_k))$ into $T_k-list$
19. For each $T_k-list$ // constructing LIU structure
20.     For each item $i$ in $T_k-list$// Reverse order
21.         $UpdateLIU(i,T_k)$
22. Raise $\delta$ using the LIU structure
23. Return $GL$
```

```
Algorithm 2 Mining
Input: A set of the global utility lists, $GUL$
Output: A set of top-$k$ high utility patterns, $TKH$
Variable: A priority queue, $PQ$; A set of conditional utility lists, $CL$; An upper-bound, $UB$; A minimum utility threshold, $\delta$;

1.  For each global utility list $GUL_p$ in $GUL$
2.      If $GUL_p.SumIU \geq \delta$
3.          $PQ = PQ \cup \{GUL_p.Name, GUL_p.SumIu\}$
4.          Raise $\delta$ to the $k$-th highest utility in $PQ$
5.      $UB = GUL_p.SumIu + GUL_p.SumRu$
6.      If $UB < \delta$
7.          continue
8.      $CL = \emptyset$
9.      For each utility list $GUL_q$ in $GUL//p \prec q$
10.         Generate a conditional utility list for $\{pq\}$, $CUL_{pq}$
11.         While tuple $T_p$ in $GUL_p \neq end$ and tuple $T_q$ in $GUL_q \neq end$
12.             If $T_p$ and $T_q$ have the same TID
13.                 $util = U(p, TID) + U(q, TID)$
14.                 $rutil = RU(q, TID)$
15.                 Insert a tuple $< TID, util, rutil >$ into $CUL_{pq}$ and Increase $T_p$ and $T_q$
16.             Else if $T_q$ is greater than $T_p$
17.                 Increase $T_p$
18.             Else
19.                 Increase $T_q$
20.         $CL = CL \cup CUL_{pq}$
21.     If $CL$ is not empty
22.         $Re-Mining(CL, GUL_p)$
23. $TKH = \{X | X \in PQ \text{ and } U(X) \geq \delta\}$
24. Return $TKH$
```

more efficient than scanning the entire database because the read speed of primary memory is faster than that of secondary memory. The incremental method thus has less overhead when processing database insertions. Moreover, as more data is inserted in a stream environment, the entire accumulated database size grows, thereby widening the efficiency gap between static and incremental methods with each incremental step. ■

## F. ANALYZING THE PROPOSED ALGORITHM

We describe two algorithms, *Insertion* and *Mining*, in detail. The proposed approach calls *Insertion* to handle the incremental data and *Mining* to extract results by extending patterns. Algorithm 1 presents the sub-procedure of ITHUI, *Insertion*. The utility information and utility lists are updated by scanning the incremental data once (Lines 01–06). The utility and *TWU* values of items increase (Line 03). The utility list is newly generated only if there is no utility list for the processed item. Then, a tuple that stores the utility information of the item is appended to the utility list (Lines 04–06). For the completeness of the results, the minimum utility threshold is raised to the updated utilities of the items (Line 07). The global utility lists are sorted in the newly obtained *TWU*-ascending order in order to calculate the remaining utility (Line 08). ITHUI conducts the restructuring process from the last utility list (Lines 09–13). The remaining utility of the tuple is set to the remaining utility in the temporary array and the remaining utility in the array increases by the utility of the tuple (Lines 11–12). The utility and remaining utility are obtained after the restructuring process (Line 13). From the updated global utility lists, the algorithm performs the restoring process to construct the LIU structure. (Lines 14–18). While traversing the global utility lists, if the *TWU* value of an item is less than *minutil*, the algorithm does not contain it in the temporary transaction list (Lines 15–16). It stores the name of the item and utility value by traversing tuples in the lists for the item whose *TWU* value is greater than or equal to *minutil* (Lines 17–18). Finally, ITHUI configures the LIU structure using the temporary transaction lists (Lines 19–21). The threshold is raised by utilizing the LIU structure (Line 22). The restructured global utility lists are returned to the main procedure (Line 23).

Algorithm 2 represents the sub-procedure of ITHUI, *Mining*. One global utility list is selected, the utility and upper-bound of the selected item are compared with the threshold, and longer patterns are created. If the pattern has a utility that is greater than current *minutil*, the pattern is inserted into the priority queue, and *minutil* increases to the *k*th utility value in the priority queue (Lines 02–04). The pattern is expanded when an upper-bound is not less than *minutil* (Lines 05–22). A set of utility lists for the combined patterns is initialized (Line 08). ITHUI performs the combination by selecting an item to be combined with the selected item (Lines 08–20). The utility list for the combined pattern is created (Line 10). ITHUI handles each tuple to find TID where the combined item appears (Lines 11–19). For each pair of tuples with the same TID, ITHUI creates a tuple storing utility information of the combined pattern and inserts it into the utility list of the combined pattern (Lines 12–15). If the compared TIDs of tuples are different, the tuple with a smaller TID moves to the next tuple (Lines 16–19). The created utility list is added to the set of conditional utility lists for the next mining process (Line 20). If the set is not empty, *Re-Mining* sub-procedure is recursively called to generate longer patterns (Lines 21–22). Finally, the result patterns are extracted from the priority queue by comparing the utility of the pattern with the optimal minimum utility threshold (Line 23). The result patterns are returned (Line 24).

### G. CORRECTNESS OF THE PROPOSED ALGORITHM

This subsection discusses the correctness of the proposed ITHUI method. ITHUI performs incremental top-*k* high utility pattern mining efficiently through a single scan of the inserted data. However, it must be certain that the proposed algorithm mines the same results as existing methods in the same domain that are designed to operate in static environments, such as THUI, TKO, and FTKHUIM. We provide the following proof regarding this matter. Assume an incremental database consisting of an original database, $DB_0 = \{T_1, T_2 \ldots, T_x\}$, and incremental databases, $DB_1 = \{T_{x+1}, T_{x+2} \ldots, T_y\}$, $DB_2 = \{T_{y+1}, T_{y+2} \ldots, T_z\}$, and so on. Next, assume an arbitrary pattern $X$ and let its utility value in an incremental database accumulated up to $DB$ for a static and incremental algorithm be $U_S(DB, X)$ and $U_I(DB, X)$, respectively. Then, when $DB_0$ is inserted, $U_S\,(DB_0, X) = \sum_{i=1}^{x} U(X, T_i)$ and $U_I\,(DB_0, X) = \sum_{i=1}^{x} U(X, T_i)$. When the first incremental database $DB_1$ is inserted, $U_S\,(DB_1, X) = \sum_{i=1}^{y} U(X, T_i)$ and $U_I\,(DB_1, X) = \sum_{i=1}^{y} U(X, T_i)$. Finally, when $DB_2$ is inserted, $U_S\,(DB_2, X) = \sum_{i=1}^{z} U(X, T_i)$ and $U_I\,(DB_2, X) = \sum_{i=1}^{z} U(X, T_i)$. Likewise, for any whole number $n$, $U_S\,(DB_n, X) = U_I\,(DB_n, X)$ is satisfied. According to *Definition 6*, if the utility value of the pattern is greater than or equal to an optimal *minutil*, it can be extracted as a top-*k* high utility pattern. Since our incremental method, ITHUI, and the existing static methods all use the same utility measure, ITHUI extracts the same results as previous top-*k* high utility pattern mining methods under the same circumstances. This proves the correctness of ITHUI and shows that ITHUI guarantees the correctness of mined results in an incremental environment.

**TABLE 3.** Characteristics of diverse datasets.

| *Dataset* | *\|D\|* | $T_{avg}$ | *\|I\|* | *Type* |
|---|---|---|---|---|
| Chess | 3,196 | 37.00 | 75 | Dense |
| Connect | 67,557 | 43.00 | 129 | Dense |
| Accidents | 340,183 | 33.81 | 468 | Dense |
| Bms-pos | 515,597 | 6.5 | 1,656 | Sparse |
| T50I4DxK | 200K - 1,000K | 50 | 1,000 | Synthetic |

## IV. PERFORMANCE EVALUATION

### A. EXPERIMENTAL SETTINGS

We performed various experiments and analyzed the performance of the proposed ITHUI algorithm, which is based on an efficient list structure. ITHUI is the first top-*k* high utility pattern mining method to efficiently process accumulated incremental databases, and the latest studies on top-*k* high utility pattern mining are all static algorithms that cannot process incremental databases. Therefore, the recent list-based top-*k* high utility pattern mining algorithms FTKHUIM [50], THUI [39], and TKO [37] were used for comparisons, and we modified them to operate on incremental databases for fair tests. The modified algorithms are marked with '-I', and for each incremental step, they treat the entire accumulated incremental database as a static database and scan it twice. This can show that ITHUI can handle accumulated incremental data more efficiently than static approaches. Although there are sliding window-based top-*k* high utility pattern mining approaches, they are unsuitable for handling entire accumulated incremental databases. All algorithms used for

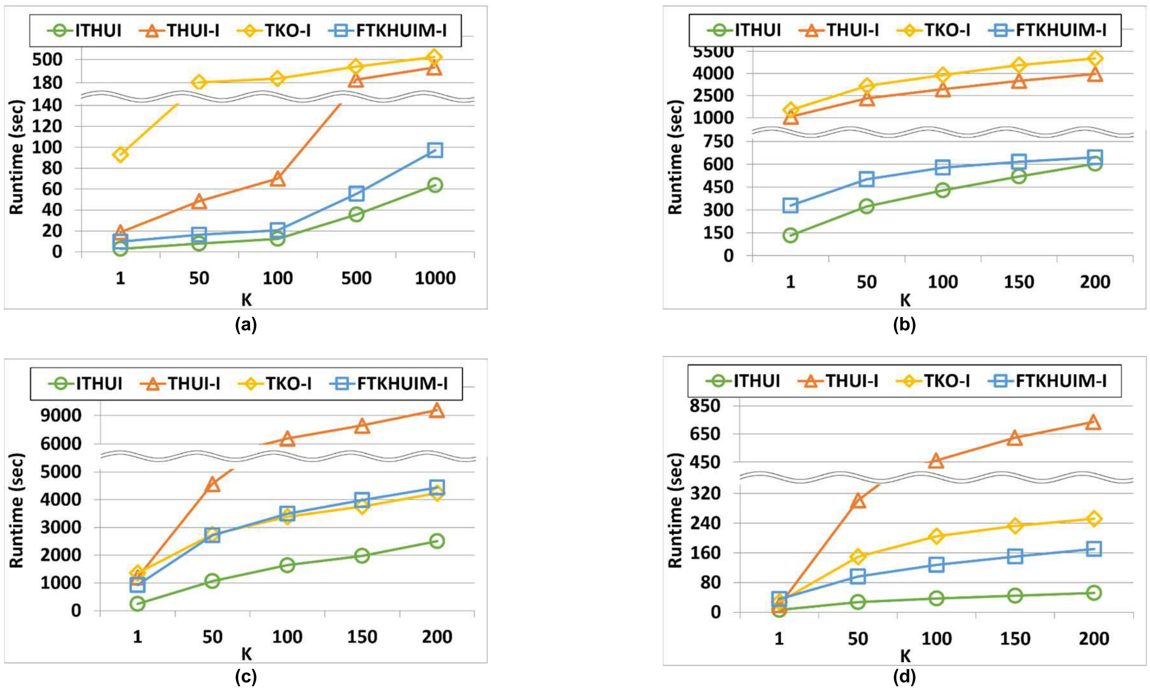


**FIGURE 7.** **Runtime evaluation under varied *k* values with four database insertions: (a) on Chess, (b) on Connect, (c) on Accidents, (d) on Bms-pos.**

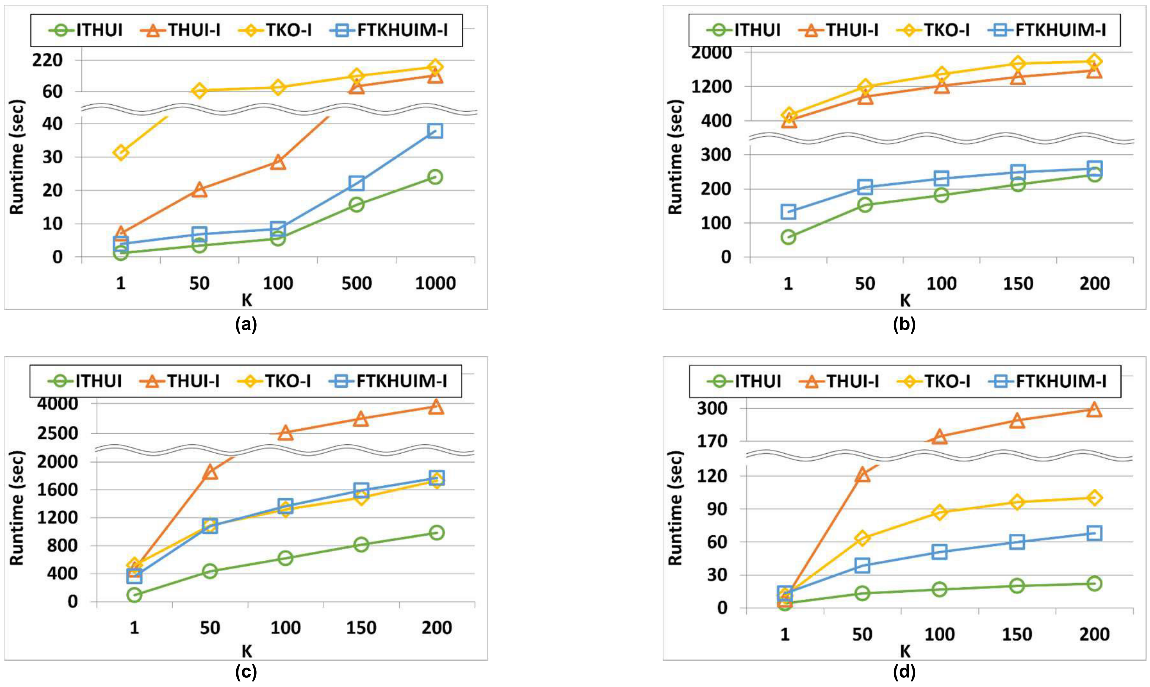


**FIGURE 8.** **Runtime evaluation under varied *k* values with one database insertion: (a) on Chess, (b) on Connect, (c) on Accidents, (d) on Bms-pos.**

performance evaluations were implemented in C/C++, and experiments were conducted on a computer with an Intel i7-6700K CPU @ 4.00 GHz and 32GB of RAM running Windows 10 64-bit operating system.

We used four real datasets and one synthetic dataset group for our experiments. Table 3 shows the characteristics of each dataset. $|D|$ denotes the number of transactions, $T_{avg}$ represents the average transaction length, and $|I|$ means the number of distinct items in the dataset. Bms-pos is a sparse dataset with relatively short transactions and a large number of items. Chess, Connect, and Accidents are dense datasets with relatively long transactions and few items. These real datasets can be accessed in the FIMI repository (http://fimi.cs.Helsinki.fi). T50I4DxK is a synthetic dataset

**TABLE 4. Runtime evaluation under the varied database insertions.**

| *Chess* | *Number of Database Insertions* | | | | | |
|---|---|---|---|---|---|---|
| *(K=1,000)* | 1 | 4 | 8 | 12 | 16 | 20 |
| ITHUI | 24.045 | 63.782 | 109.477 | 151.578 | 196.296 | 237.768 |
| FTKHUIM-I | 37.913 | 96.948 | 180.937 | 263.996 | 345.362 | 423.852 |
| *Connect* | *Number of Database Insertions* | | | | | |
| *(K=200)* | 1 | 4 | 8 | 12 | 16 | 20 |
| ITHUI | 241.348 | 603.523 | 1,022.75 | 1,444.33 | 1,993.66 | 2,359.26 |
| FTKHUIM-I | 259.775 | 645.937 | 1,190.94 | 1,704.68 | 2,272.26 | 2,785.17 |
| *Accidents* | *Number of Database Insertions* | | | | | |
| *(K=200)* | 1 | 4 | 8 | 12 | 16 | 20 |
| ITHUI | 984.578 | 2,506.42 | 4,000.49 | 5,781.46 | 7,737.57 | 9,792.36 |
| FTKHUIM-I | 1,772.03 | 4,429.15 | 8,448.11 | 12,025.5 | 15,903.8 | 19,810.1 |
| *Bms-pos* | *Number of Database Insertions* | | | | | |
| *(K=200)* | 1 | 4 | 8 | 12 | 16 | 20 |
| ITHUI | 22.049 | 52.452 | 87.367 | 125.247 | 161.649 | 198.301 |
| FTKHUIM-I | 68.063 | 170.592 | 326.321 | 473.745 | 612.616 | 751.748 |

**TABLE 5. Memory evaluation under the varied *k* values.**

| | *Chess* | *K* | | | | |
|---|---|---|---|---|---|---|
| *Four Database Insertions* | | 1 | 50 | 100 | 500 | 1,000 |
| | ITHUI | 12.922 | 14.09 | 14.355 | 17.27 | 20.953 |
| | THUI-I | 21.164 | 24.422 | 25.156 | 29.863 | 31.789 |
| | TKO-I | 26.781 | 27.559 | 28.152 | 29.293 | 29.094 |
| | FTKHUIM-I | 22.469 | 23.5 | 22.832 | 24.637 | 27.809 |
| | *Connect* | *K* | | | | |
| | | 1 | 50 | 100 | 150 | 200 |
| | ITHUI | 407.363 | 435.25 | 438.82 | 441.777 | 433.918 |
| | THUI-I | 751.203 | 767.574 | 799.441 | 811.387 | 814.484 |
| | TKO-I | 755.699 | 756.918 | 757.727 | 758.332 | 758.504 |
| | FTKHUIM-I | 476.434 | 512.973 | 515.344 | 521.203 | 535.477 |
| | *Accidents* | *K* | | | | |
| | | 1 | 50 | 100 | 150 | 200 |
| | ITHUI | 723.672 | 772.906 | 781.73 | 782.934 | 788.105 |
| | THUI-I | 1,392.38 | 1,524.26 | 1,556.04 | 1,623.9 | 1,639.22 |
| | TKO-I | 1,395.43 | 1,482.12 | 1,482.29 | 1,500.39 | 1,500.93 |
| | FTKHUIM-I | 1,513.5 | 1,654.04 | 1,659.82 | 1,654.85 | 1,647.05 |
| | *Bms-pos* | *K* | | | | |
| | | 1 | 50 | 100 | 150 | 200 |
| | ITHUI | 158.023 | 185.887 | 187.746 | 189.402 | 190.109 |
| | THUI-I | 152.727 | 244.469 | 252.332 | 254.969 | 256.75 |
| | TKO-I | 222.023 | 226.895 | 227.352 | 227.324 | 227.414 |
| | FTKHUIM-I | 217.047 | 419.207 | 431.098 | 435.52 | 438.457 |
| *One Database Insertion* | *Chess* | *K* | | | | |
| | | 1 | 50 | 100 | 500 | 1,000 |
| | ITHUI | 12.523 | 14.176 | 14.531 | 16.918 | 20.859 |
| | THUI-I | 20.793 | 24.188 | 25.551 | 29.285 | 31.016 |
| | TKO-I | 26.68 | 27.445 | 27.594 | 28.891 | 28.832 |
| | FTKHUIM-I | 22.32 | 22.125 | 22.652 | 24.77 | 27.398 |
| | *Connect* | *K* | | | | |
| | | 1 | 50 | 100 | 150 | 200 |
| | ITHUI | 407.051 | 425.879 | 431.820 | 433.355 | 430.773 |
| | THUI-I | 750.328 | 763.555 | 794.418 | 798.867 | 812.29 |
| | TKO-I | 752.285 | 753.375 | 753.84 | 754.102 | 753.801 |
| | FTKHUIM-I | 462.121 | 496.043 | 496.488 | 499.801 | 515.453 |
| | *Accidents* | *K* | | | | |
| | | 1 | 50 | 100 | 150 | 200 |
| | ITHUI | 719.27 | 771.98 | 784.406 | 779.902 | 786.273 |
| | THUI-I | 1,351.10 | 1,526.71 | 1,534.57 | 1,553.13 | 1,558.78 |
| | TKO-I | 1,388.95 | 1,475.41 | 1,475.23 | 1,493.41 | 1,493.09 |
| | FTKHUIM-I | 1,491.54 | 1,648.55 | 1,654.52 | 1,656.93 | 1,658.19 |
| | *Bms-pos* | *K* | | | | |
| | | 1 | 50 | 100 | 150 | 200 |
| | ITHUI | 157.801 | 185.094 | 187.484 | 188.547 | 189.379 |
| | THUI-I | 151.379 | 241.09 | 248.965 | 251.996 | 253.379 |
| | TKO-I | 223.051 | 226.562 | 226.875 | 227.008 | 226.996 |
| | FTKHUIM-I | 214.715 | 415.516 | 428.168 | 431.766 | 430.219 |

group that is used for the scalability test, where the number of transactions grows from 200K to 1,000K. T50I4DxK is obtained from the IBM generator. In all experiments, the original database accounts for 90% of the entire database, and the incremental databases occupy the remaining 10%. The incremental databases are divided into equal parts. After processing the original database, the algorithms handle each part as the newly inserted data. For our runtime and memory usage tests using real datasets, we measured the results in a situation where the incremental databases are treated as four and one parts. In short, we measured the runtime and memory usage performance of the algorithms when the number of database insertions was both four and one. Furthermore, we simulated a situation where the number of database insertions increased and compared the performance of ITHUI and the recent FTKHUIM-I using the four real datasets. Upon database insertion, the proposed algorithm scans only the newly inserted data, whereas THUI-I, TKO-I, and FTKHUIM-I scan the entire accumulated database twice since they were designed for static environments.

## B. RUNTIME TEST

In this subsection, the results of the runtime test for each algorithm are analyzed. Fig. 7 shows the runtime performance of each algorithm for the four real datasets with four database insertions. The results of the runtime test on Chess are presented in Fig. 7(a). The *k* value, which is the number of patterns that the user wants to extract, was increased from 1 to 1,000. As *k* increased, the runtime of ITHUI increased slightly, while the runtime of THUI-I and TKO-I grew rapidly. When the *k* value was greater than 500, the runtime of FTKHUIM-I also rose sharply, and the gap between FTKHUIM-I and ITHUI increased. Unlike TKO-I, ITHUI, FTKHUIM-I and THUI-I can reduce the upper-bound during the mining process by considering the utility information in the transactions where the combined pattern cannot appear. As a result, they required less runtime than TKO-I. Fig. 7(b) shows the runtime results for Connect. Here, *k* grew from 1 to 200. ITHUI, FTKHUIM-I and THUI-I, which utilize the LIU structure, had better runtime performance than TKO-I for all *k* values. ITHUI, FTKHUIM-I, and THUI-I efficiently raise the minimum utility threshold by using the actual utilities of the ordered and contiguous patterns managed in the LIU structure and estimating the utilities of patterns that are not stored in the LIU structure. Moreover, ITHUI can efficiently process incremental data with a single database scan through the restructuring and restoring procedures. Although the runtime gap between ITHUI and FTKHUIM-I is not great, if the number of incremental databases increases, the runtime

**TABLE 6.** Memory evaluation under the varied database insertions.

| *Chess* | *Number of Database Insertions* | | | | | |
|---|---|---|---|---|---|---|
| *(K=1,000)* | 1 | 4 | 8 | 12 | 16 | 20 |
| ITHUI | 20.859 | 20.953 | 21.293 | 20.91 | 21.281 | 21.344 |
| FTKHUIM-I | 27.398 | 27.809 | 28.848 | 30.402 | 30.066 | 30.062 |
| *Connect* | *Number of Database Insertions* | | | | | |
| *(K=200)* | 1 | 4 | 8 | 12 | 16 | 20 |
| ITHUI | 430.773 | 433.918 | 439.574 | 444.406 | 440.82 | 443.305 |
| FTKHUIM-I | 515.453 | 535.477 | 526.156 | 545.793 | 547.289 | 550.18 |
| *Accidents* | *Number of Database Insertions* | | | | | |
| *(K=200)* | 1 | 4 | 8 | 12 | 16 | 20 |
| ITHUI | 786.273 | 788.105 | 790.578 | 792.281 | 800.438 | 797.488 |
| FTKHUIM-I | 1,658.19 | 1,647.05 | 1,666.74 | 1,680.43 | 1,698.33 | 1,697.99 |
| *Bms-pos* | *Number of Database Insertions* | | | | | |
| *(K=200)* | 1 | 4 | 8 | 12 | 16 | 20 |
| ITHUI | 189.379 | 190.109 | 190.98 | 190.754 | 192.27 | 192.344 |
| FTKHUIM-I | 430.219 | 438.457 | 439.523 | 439.441 | 440.434 | 441.328 |

gap between them becomes greater. Fig. 7(c) represents the results of the runtime test performed on Accidents, where $k$ increased from 1 to 200. As the $k$ value grew, the optimal minutil decreased, so all algorithms processed more patterns and required more runtime. Nonetheless, ITHUI showed the best runtime performance out of all algorithms. The runtime results for Bms-pos are shown in Fig. 7(d), where $k$ increased from 1 to 200. ITHUI had the best runtime performance, followed by FTKHUIM-I, TKO-I, then THUI-I. Bms-pos is a large, sparse dataset with lots of items, so many items belong to the result patterns. As a result, as $k$ increased and more patterns were extracted, the runtime gap between ITHUI and the others increased, while ITHUI showed the smallest increase in runtime. Fig. 8 shows the runtime performance of each algorithm for the four real datasets with one database insertion. As seen from the line charts, the overall trend is analogous to that of the runtime test with four database insertions. However, the runtimes of all algorithms decreased because the number of database insertions and mining requests decreased from four to one. Nonetheless, it is clear that ITHUI's runtime performance exceeds those of FTKHUIM-I, THUI-I and TKO-I. Finally, Table 4 shows the results of the runtime test with a varied number of database insertions, where the values represent seconds. The test compares the runtimes of the proposed incremental ITHUI algorithm and the recent static FTKHUIM-I algorithm to demonstrate the performance of ITHUI in a situation where the number of database insertions increases. In this test, the $k$ value was set to 1,000 for Chess and 200 for the others, which are the largest $k$ values used in the runtime test with varied $k$ values. Likewise, the incremental database portion was also set to the same 10%. As seen from the table, ITHUI is faster than FTKHUIM-I in all experiments, and the runtime gaps increase as the number of database insertions increases. For example, in Connect, when database insertion occurred once, the runtime difference was around 18 seconds, and when it occurred twenty times, the difference was around 426 seconds. In conclusion, the runtime test demonstrates that

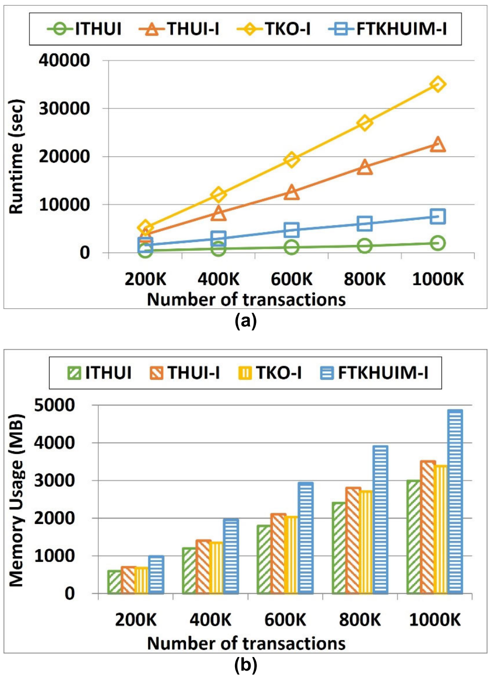


**FIGURE 9.** Scalability evaluation on T50I4DxK: (a) Runtime scalability, (b) Memory scalability.

ITHUI has better runtime performance than previous methods under the same conditions and that the proposed techniques are highly effective for handling accumulated incremental databases.

### C. MEMORY USAGE TEST

In this subsection, we measured the peak memory usage in MBs occupied by each algorithm while extracting the top-$k$ high utility patterns from each dataset. The top half of Table 5 shows the results of the memory usage test performed on the four real datasets with four database insertions. In each dataset, the range of $k$ values was the same as that used in the runtime test. In the dense datasets, which include Chess, Connect, and Accidents, as the $k$ value increased, the peak memory usage became larger. For Chess, THUI-I required less memory than TKO-I when $k$ was under 100 but spent similar or more memory when $k$ was over 500. In addition, FTKHUIM-I shows better memory efficiency than THUI-I and TKO-I. On the other hand, ITHUI always needs the least memory space, regardless of the $k$ value. For Connect, when $k$ was set to 1, ITHUI consumed 407.363 MB of memory, unlike other compared algorithms, which required at least 16% more memory. Increasing the $k$ value means that the optimal minimum utility threshold for extracting the result patterns becomes lower, and more patterns must be processed. Nevertheless, ITHUI had about 1.23 times, 1.87 times and 1.74 times better memory usage performance than FTKHUIM-I, THUI-I and TKO-I, respectively, when the $k$ value was the highest at 200. The memory usage results measured for Accidents also showed that ITHUI had the best memory usage performance with varying $k$ values. When the $k$ value was the highest at 200, the static algorithms, FTKHUIM-I, THUI-I, and TKO-I, occupied more than 1,500 MB of memory, but ITHUI required 788.105 MB of memory. Lastly, the sparse dataset, Bms-pos, is a large dataset with a large number of distinct items. For this dataset, the proposed ITHUI algorithm showed efficient memory usage in the experiments. Moreover, as $k$ increased, the memory usage of FTKHUIM-I and THUI-I quickly increased, while ITHUI increased relatively slowly. Compared with TKO-I, the proposed algorithm required less memory for all $k$ values, and when $k$ was the highest at 200, TKO-I required approximately 20% more memory space than ITHUI. The bottom half of Table 5 shows the results of the memory usage test performed on the four real datasets with one database insertion. The results were very close to those of the memory usage test with four database insertions. This is because in both tests, the entire accumulated database is processed in the last incremental step, where the peak memory usage is measured. Lastly, Table 6 shows the results of the memory usage test with a varied number of database insertions. This test was conducted in the same way as its runtime test counterpart. Results show similar results to the previous memory usage tests, since the peak memory is measured when the entire database is fully inserted. Nonetheless, the proposed ITHUI algorithm consumes the least amount of memory across all four real datasets in a situation where the number of database insertions increases. In summary, ITHUI had the overall best memory usage performance in accumulated incremental database environments across the $k$ values and number of database insertions.

### D. SCALABILITY TEST

In this subsection, we analyze the results of the scalability test using the T50I4DxK dataset group presented in Table 3. The number of database insertions was set to four, following the previous runtime and memory usage tests. Fig. 9 represents the scalability performance of the algorithms, where the $k$ value was set to 100. Fig. 9(a) presents the results of the runtime scalability test. As the number of transactions increased, the runtimes of each algorithm also increased. This is because the amount of data to be processed becomes larger depending on the size of the database. In addition, since ITHUI, THUI-I, TKO-I, and FTKHUIM-I are all list-based algorithms, the number of tuples stored in their list structures also increases when the size of the database grows. However, while the runtime of THUI-I, TKO-I, and FTKHUIM-I rose rapidly, the runtime of ITHUI grew slightly. This is because ITHUI is an incremental algorithm that manages the global lists by scanning the incremental data once, but the three static algorithms scan the entire database twice whenever new databases are inserted. In summary, ITHUI had the best runtime scalability performance, regardless of the number of transactions. The memory scalability performance of each algorithm for T50I4DxK is shown in Fig. 9(b). Memory usage also increased as the database size increased because the utility list structure stored more utility information as the number of transactions grew. ITHUI consumed less memory than THUI-I, despite the increase in the number of transactions. Moreover, the gap in memory usage between THUI-I and ITHUI increased as the size of the database became larger, which signified that ITHUI has better memory scalability performance. While TKO-I showed better memory performance than THUI-I, ITHUI still always required less memory than TKO-I for all database sizes. For FTKHUIM-I, it consumed the most memory in all cases because of its relatively complex list structure and additional data structures used for threshold raising strategies. As a result, the proposed ITHUI algorithm, which mines top-$k$ high utility patterns from accumulated incremental databases using a list structure, outperformed the state-of-the-art static algorithms THUI-I, TKO-I, and FTKHUIM-I in terms of scalability.

## V. CONCLUSION

In this paper, we suggested a novel method for discovering top-$k$ high utility patterns from continuously accumulated incremental databases. The suggested algorithm, ITHUI, stores the minimum utility information of patterns by constructing list structures. Since the utility list holds the actual utilities of patterns, candidate patterns are not created, and thus additional database scans are not required. Next, whenever an incremental database is inserted, ITHUI updates and restructures its global lists efficiently through a single scan. Additionally, it constructs the LIU structure corresponding to the changed *TWU*-ascending order through the introduced restoring process and raises the minimum utility threshold efficiently. Therefore, ITHUI is suitable for processing incremental data. We evaluated the performance of ITHUI in terms of runtime, memory, and scalability using diverse real and synthetic datasets. The experimental results showed that ITHUI processed incremental databases faster than state-of-the-art static algorithms while also consuming less memory. Moreover, ITHUI demonstrated great performance in a situation where the size of the database became larger. Our future work is to devise effective and efficient techniques in incremental top-$k$ high utility pattern mining, such as itemset pruning and threshold raising strategies, and also to develop more practical methods for the concept of top-$k$ in advanced pattern mining areas.

## REFERENCES

[1] W. Li, H. Zhang, J. Chen, P. Li, Y. Yao, X. Shi, M. Shibasaki, H. H. Kobayashi, X. Song, and R. Shibasaki, "Metagraph-based life pattern clustering with big human mobility data," *IEEE Trans. Big Data*, vol. 9, no. 1, pp. 227–240, Feb. 2023.

[2] Y. Chen, P. Yu, W. Chen, Z. Zheng, and M. Guo, "Meta-learning based classification for moving object trajectories in mobile IoT," *IEEE Trans. Big Data*, vol. 9, no. 2, pp. 584–596, Apr. 2023.

[3] C. Zhou, J. Xu, M. Jiang, D. Tang, and S. Wang, "Mining top-k frequent patterns in large geosocial networks: A mnie-based extension approach," *IEEE Access*, vol. 11, pp. 27662–27675, 2023.

[4] I. Mavroudopoulos and A. Gounaris, "SIESTA: A scalable infrastructure of sequential pattern analysis," *IEEE Trans. Big Data*, vol. 9, no. 3, pp. 975–990, 2023.

[5] C. Zhang, Q. Dai, Z. Du, W. Gan, J. Weng, and P. S. Yu, "TUSQ: Targeted high-utility sequence querying," *IEEE Trans. Big Data*, vol. 9, no. 2, pp. 512–527, Apr. 2023.

[6] Y. Cai, H. Che, B. Pan, M.-F. Leung, C. Liu, and S. Wen, "Projected cross-view learning for unbalanced incomplete multi-view clustering," *Inf. Fusion*, vol. 105, May 2024, Art. no. 102245.

[7] C. Lee, Y. Baek, J. C.-W. Lin, T. Truong, and U. Yun, "Advanced uncertainty based approach for discovering erasable product patterns," *Knowl.-Based Syst.*, vol. 241, Apr. 2022, Art. no. 108134.

[8] M. Han, N. Zhang, L. Wang, X. Li, and H. Cheng, "Mining closed high utility patterns with negative utility in dynamic databases," *Appl. Intell.*, vol. 53, no. 10, pp. 11750–11767, 2023.

[9] G. Li, T. Shang, and Y. Zhang, "Efficient mining high average-utility itemsets with effective pruning strategies and novel list structure," *Appl. Intell.*, vol. 53, no. 5, pp. 6099–6118, 2023.

[10] Y. Baek, U. Yun, H. Kim, H. Nam, G. Lee, E. Yoon, B. Vo, and J. C-W. Lin, "Erasable pattern mining based on tree structures with damped window over data streams," *Eng. Appl. Artif. Intell.*, vol. 94, Sep. 2020, Art. no. 103735.

[11] M. Z. A. Bhuiyan, J. Wu, G. M. Weiss, T. Hayajneh, T. Wang, and G. Wang, "Event detection through differential pattern mining in cyber-physical systems," *IEEE Trans. Big Data*, vol. 6, no. 4, pp. 652–665, Dec. 2020.

[12] W. Gan, J. C. Lin, P. Fournier-Viger, H.-C. Chao, V. S. Tseng, and P. S. Yu, "A survey of utility-oriented pattern mining," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 4, pp. 1306–1327, Apr. 2021.

[13] H. Kim, H. Kim, S. Kim, H. Kim, M. Cho, B. Vo, J. C.-W. Lin, and U. Yun, "An advanced approach for incremental flexible periodic pattern mining on time-series data," *Exp. Syst. Appl.*, vol. 230, Nov. 2023, Art. no. 120697.

[14] J. Kim, U. Yun, E. Yoon, J. C.-W. Lin, and P. Fournier-Viger, "One scan based high average-utility pattern mining in static and dynamic databases," *Future Gener. Comput. Syst.*, vol. 111, pp. 143–158, Oct. 2020.

[15] C. Zhang, Z. Du, Y. Yang, W. Gan, and P. S. Yu, "On-shelf utility mining of sequence data," *ACM Trans. Knowl. Discov. Data*, vol. 16, no. 2, pp. 1–31, 2021.

[16] N. T. Tung, L. T. T. Nguyen, T. D. D. Nguyen, P. Fourier-Viger, N.-T. Nguyen, and B. Vo, "Efficient mining of cross-level high-utility itemsets in taxonomy quantitative databases," *Inf. Sci.*, vol. 587, pp. 41–62, Mar. 2022.

[17] T. Truong, H. Duong, B. Le, P. Fournier-Viger, and U. Yun, "Mining interesting sequences with low average cost and high average utility," *Appl. Intell.*, vol. 52, no. 7, pp. 7136–7157, 2022.

[18] M. S. Nawaz, P. Fournier-Viger, U. Yun, Y. Wu, and W. Song, "Mining high utility itemsets with Hill climbing and simulated annealing," *ACM Trans. Manag. Inf. Syst.*, vol. 13, no. 1, pp. 1–22, 2021.

[19] A. Segura-Delgado, A. Anguita-Ruiz, R. Alcala, and J. Alcala-Fdez, "Mining high average-utility sequential rules to identify high-utility gene expression sequences in longitudinal human studies," *Exp. Syst. Appl.*, vol. 193, May 2022, Art. no. 116411.

[20] Y. Liu, W.-K. Liao, and A. Choudhary, "A two-phase algorithm for fast discovery of high utility itemsets," in *Proc. Adv. Knowledge Discovery Data Mining, 9th Pacific-Asia Conf.*, vol. 3518, 2005, pp. 689–695.

[21] V. S. Tseng, C.-W. Wu, B.-E. Shie, and P. S. Yu, "UP-growth: An efficient algorithm for high utility itemset mining," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Disc. Data Mining*, 2010, pp. 253–262.

[22] V. S. Tseng, B.-E. Shie, C.-W. Wu, and P. S. Yu, "Efficient algorithms for mining high utility itemsets from transactional databases," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 8, pp. 1772–1786, Aug. 2013.

[23] C.-W. Wu, P. Fournier-Viger, J.-Y. Gu, and V. S. Tseng, "Mining closed+ high utility itemsets without candidate generation," in *Proc. Conf. Technol. Appl. Artif. Intell. (TAAI)*, Nov. 2015, pp. 187–194.

[24] P. Fournier-Viger, C. Wu, S. Zida, and V. S. Tseng, "FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning," in *Proc. Int. Symp. Methodologies Intell. Syst.*, 2014, pp. 83–92.

[25] S. Krishnamoorthy, "Pruning strategies for mining high utility itemsets," *Exp. Syst. Appl.*, vol. 42, no. 5, pp. 2371–2381, 2015.

[26] S. Krishnamoorthy, "HMiner: Efficiently mining high utility itemsets," *Exp. Syst. Appl.*, vol. 90, pp. 168–183, Dec. 2017.

[27] A. Mondal, S. Saurabh, P. Chaudhary, R. Mittal, and P. K. Reddy, "A retail itemset placement framework based on premiumness of slots and utility mining," *IEEE Access*, vol. 9, pp. 155207–155223, 2021.

[28] J. M.-T. Wu, G. Srivastava, J. C.-W. Lin, Y. Djenouri, M. Wei, R. M. Parizi, and M. S. Khan, "Mining of high-utility patterns in big IoT-based databases," *Mobile Netw. Appl.*, vol. 26, no. 1, pp. 216–233, 2021.

[29] U. Yun, H. Kim, T. Ryu, Y. Baek, H. Nam, J. Lee, B. Vo, and W. Pedrycz, "Prelarge-based utility-oriented data analytics for transaction modifications in Internet of Things," *IEEE Internet Things J.*, vol. 8, no. 24, pp. 17333–17344, Dec. 2021.

[30] H. Nam, U. Yun, E. Yoon, and J. C.-W. Lin, "Efficient approach of recent high utility stream pattern mining with indexed list structure and pruning strategy considering arrival times of transactions," *Inf. Sci.*, vol. 529, pp. 1–27, Aug. 2020.

[31] H. Kim, U. Yun, Y. Baek, H. Kim, H. Nam, J. C.-W. Lin, and P. Fournier-Viger, "Damped sliding based utility oriented pattern mining over stream data," *Knowl.-Based Syst.*, vol. 213, Feb. 2021, Art. no. 106653.

[32] Y. Baek, U. Yun, H. Kim, H. Nam, H. Kim, J. C.-W. Lin, B. Vo, and W. Pedrycz, "RHUPS: Mining recent high utility patterns with sliding window–based arrival time control over data streams," *ACM Trans. Intell. Syst. Technol.*, vol. 12, no. 2, pp. 1–27, Apr. 2021.

[33] J. M.-T. Wu, Q. Teng, J. C.-W. Lin, U. Yun, and H.-C. Chen, "Updating high average-utility itemsets with pre-large concept," *J. Intell. Fuzzy Syst.*, vol. 38, no. 5, pp. 5831–5840, 2020.

[34] H. Kim, C. Lee, T. Ryu, H. Kim, S. Kim, B. Vo, J. C.-W. Lin, and U. Yun, "Pre-large based high utility pattern mining for transaction insertions in incremental database," *Knowl.-Based Syst.*, vol. 268, May 2023, Art. no. 110478.

[35] C. W. Wu, B.-E. Shie, V. S. Tseng, and P. S. Yu, "Mining top-k high utility itemsets," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2012, pp. 78–86.

[36] H. Ryang and U. Yun, "Top-k high utility pattern mining with effective threshold raising strategies," *Knowl.-Based Syst.*, vol. 76, pp. 109–126, Mar. 2015.

[37] V. S. Tseng, C.-W. Wu, P. Fournier-Viger, and P. S. Yu, "Efficient algorithms for mining top-K high utility itemsets," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 1, pp. 54–67, Jan. 2016.

[38] Q.-H. Duong, B. Liao, P. Fournier-Viger, and T.-L. Dam, "An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies," *Knowl.-Based Syst.*, vol. 104, pp. 106–122, Jul. 2016.

[39] S. Krishnamoorthy, "Mining top-k high utility itemsets with effective threshold raising strategies," *Exp. Syst. Appl.*, vol. 117, pp. 148–165, Mar. 2019.

[40] C. Liu and C. Guo, "Mining top-n high-utility operation patterns for taxi drivers," *Exp. Syst. Appl.*, vol. 170, May 2021, Art. no. 114546.

[41] C. Zhang, Z. Du, W. Gan, and P. S. Yu, "TKUS: Mining top-k high utility sequential patterns," *Inf. Sci.*, vol. 570, pp. 342–359, Sep. 2021.

[42] J.-W. Huang, B. P. Jaysawal, K.-Y. Chen, and Y.-B. Wu, "Mining frequent and top-k high utility time interval-based events with duration patterns," *Knowl. Inf. Syst.*, vol. 61, no. 3, pp. 1331–1359, 2019.

[43] J. Liu, X. Zhang, B. C. M. Fung, J. Li, and F. Iqbal, "Opportunistic mining of top-n high utility patterns," *Inf. Sci.*, vol. 441, pp. 171–186, May 2018.

[44] J. Liu, K. Wang, and B. C. M. Fung, "Direct discovery of high utility itemsets without candidate generation," in *Proc. IEEE 12th Int. Conf. Data Mining*, Dec. 2012, pp. 984–989.

[45] X. Han, X. Liu, J. Li, and H. Gao, "Efficient top-k high utility itemset mining on massive data," *Inf. Sci.*, vol. 557, pp. 382–406, May 2021.

[46] J. M. Luna, R. U. Kiran, P. Fournier-Viger, and S. Ventura, "Efficient mining of top-k high utility itemsets through genetic algorithms," *Inf. Sci.*, vol. 624, pp. 529–553, May 2023.

[47] W. Gan, S. Wan, J. Chen, C.-M. Chen, and L. Qiu, "TopHUI: Top-k high-utility itemset mining with negative utility," in *Proc. IEEE Int. Conf. Big Data*, Dec. 2020, pp. 5350–5359.

[48] R. Sun, M. Han, C. Zhang, M. Shen, and S. Du, "Mining of top-k high utility itemsets with negative utility," *J. Intell. Fuzzy Syst.*, vol. 40, no. 3, pp. 5637–5652, 2021.

[49] M. Ashraf, T. Abdelkader, S. Rady, and T. F. Gharib, "TKN: An efficient approach for discovering top-k high utility itemsets with positive or negative profits," *Inf. Sci.*, vol. 587, pp. 654–678, Mar. 2022.

[50] V. V. Vu, M. T. H. Lam, T. T. M. Duong, L. T. Manh, T. T. T. Nguyen, L. V. Nguyen, U. Yun, V. Snasel, and B. Vo, "FTKHUIM: A fast and efficient method for mining top-k high-utility itemsets," *IEEE Access*, vol. 11, pp. 104789–104805, 2023.

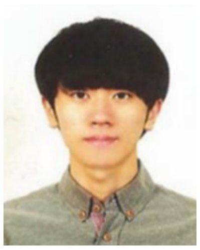
[51] H. Cheng, M. Han, N. Zhang, L. Wang, and X. Li, "ETKDS: An efficient algorithm of top-K high utility itemsets mining over data streams under sliding window model," *J. Intell. Fuzzy Syst.*, vol. 41, no. 2, pp. 3317–3338, 2021.

[52] W. Song, C. Fang, and W. Gan, "TopUMS: Top-k utility mining in stream data," in *Proc. Int. Conf. Data Mining Workshops (ICDMW)*, Dec. 2021, pp. 615–622.

**CHANHEE LEE** received the M.S. degree in computer engineering from Sejong University, Seoul, South Korea, in 2023. He is currently with Somansa. His research interests include data mining, data analytics, database systems, and artificial intelligence.

**HANJU KIM** received the B.S. degree in computer engineering from Sejong University, Seoul, South Korea, in 2023, where he is currently pursuing the M.S. degree. His research interests include data mining, data analytics, database systems, and artificial intelligence.

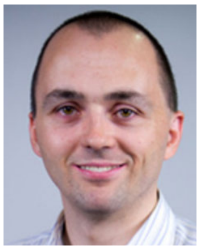
**MYUNGHA CHO** received the B.S. degree in computer engineering from Sejong University, Seoul, South Korea, in 2023, where he is currently pursuing the M.S. degree. His research interests include data mining, data analytics, database systems, and artificial intelligence.



**HYEONMO KIM** received the M.S. degree in computer engineering from Sejong University, Seoul, South Korea, in 2024. He is currently with Hanwha Systems. His research interests include data mining, data analytics, database systems, and artificial intelligence.

**BAY VO** (Member, IEEE) received the B.Sc., M.Sc., and Ph.D. degrees in computer science from the University of Science, Vietnam National University, Ho Chi Minh City, Vietnam, in 2002, 2005, and 2011, respectively. He is currently an Associate Professor and the Dean of the Faculty of Information Technology, HUTECH University, Ho Chi Minh City. His research interests include association rules, classification, mining in incremental database, distributed databases, and privacy preserving in data mining.

**JERRY CHUN-WEI LIN** (Senior Member, IEEE) received the Ph.D. degree from the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan, in 2010. He is currently a Professor with the Department of Distributed Systems and IT Devices, Silesian University of Technology, Poland. He has published more than 400 research articles in refereed journals, international conferences, 12 edited books, and 33 patents (held and filed, three U.S. patents). His research interests include data mining, soft computing, artificial intelligence/machine learning, and privacy preserving and security technologies.



**PHILIPPE FOURNIER-VIGER** received the Ph.D. degree in computer science from the University of Quebec, Montreal, in 2010. He is currently a Professor with the College of Computer Science and Software Engineering, Shenzhen University, China. He is also the Founder of the popular SPMF Open-Source Data Mining Library, which has been cited in over 430 research articles, since 2010. He has published over 140 research articles in refereed international conferences and journals, which have received over 1300 citations. His research interests include data mining, pattern mining, sequence analysis and prediction, text mining, e-learning, and social network mining. He has received the title of Youth 1000 Talent from the National Science Foundation of China.

**UNIL YUN** received the M.S. degree in computer science and engineering from Korea University, Seoul, South Korea, in 1997, and the Ph.D. degree in computer science from Texas A&M University, College Station, TX, USA, in 2005. From 1997 to 2002, he was with the Multimedia Laboratory, Korea Telecom, Seongnam-si, South Korea. After receiving the Ph.D. degree, he was a Postdoctoral Associate for almost one year with the Department of Computer Science, Texas A&M University. Then, he was a Senior Researcher with the Electronics and Telecommunications Research Institute, Gwangju, South Korea. In March 2007, he joined the School of Electrical and Computer Engineering, Chungbuk National University, South Korea. Since August 2013, he has been a Full Professor with the Department of Computer Engineering, Sejong University, Seoul. He has published more than 200 research articles in refereed journals and international conferences. His research interests include data mining, information retrieval, database systems, artificial intelligence, and digital libraries. Currently, he is an Associate Editor (Editorial Board Member) of *Knowledge-Based Systems*, *Plos One*, and *Electronics*.