An efficient algorithm for mining high utility patterns from incremental databases with one database scan

Unil Yun a,*, Heungmo Ryang a, Gangin Lee a, Hamido Fujita b

a Department of Computer Engineering, Sejong University, Seoul, Republic of Korea
b Faculty of Software and Information Science, Iwate Prefectural University (IPU), Iwate, Japan

A R T I C L E   I N F O

Article history:
Received 13 September 2016
Revised 16 March 2017
Accepted 18 March 2017
Available online 20 March 2017

Keywords:
Data mining
High utility patterns
One database scan
Incremental mining
Utility mining

A B S T R A C T

High utility pattern mining has been actively researched as one of the significant topics in the data mining field since this approach can solve the limitation of traditional pattern mining that cannot fully consider characteristics of real world databases. Moreover, database volumes have been bigger gradually in various applications such as sales data of retail markets and connection information of web services, and general methods for static databases are not suitable for processing dynamic databases and extracting useful information from them. Although incremental utility pattern mining approaches have been suggested, previous approaches need at least two scans for incremental utility pattern mining irrespective of using any structure. However, the approaches with multiple scans are actually not adequate for stream environments. In this paper, we propose an efficient algorithm for mining high utility patterns from incremental databases with one database scan based on a list-based data structure without candidate generation. Experimental results with real and synthetic datasets show that the proposed algorithm outperforms previous one phase construction methods with candidate generation.

© 2017 Elsevier B.V. All rights reserved.

1. Introduction

There have been many efforts to analyze data and discover interesting knowledge [1–5], and data mining has made a significant contribution to data analysis. Pattern mining [6–11] is one of the data mining techniques and finds meaningful information hidden in huge databases as pattern forms. Although frequent pattern mining [12–15] has played an important role in the data mining field, this has a limitation that cannot fully reflect characteristics of real world databases to mining processes. To address this issue, utility pattern mining [16–24] has been actively studied. Utility pattern mining conducts pattern mining with the consideration of relative importance and non-binary occurrence of items. In contrast to frequent pattern mining, it is not an easy task for utility pattern mining to satisfy the anti-monotone property [6], a fundamental criterion for efficient pattern mining, where no valid super pattern is generated from an invalid one. Since this improves mining performance by effectively reducing search space, maintaining the property in mining processes is significant. To solve the problem, the overestimation concept [25] was proposed and employed in utility pattern mining, but it degrades

milling performance by extracting a large number of candidate patterns. For this reason, utility pattern mining without candidate generation has been researched [26–28].

In recent years, data have been incrementally generated in various real world applications such as retail markets and web services according to their continuous operations. Therefore, an appropriate method is necessary to discover useful information from such dynamic databases because previous static approaches are not suitable for the purpose. Incremental utility pattern mining [29–31] has been suggested to address this issue. Although these methods are more suited to mining high utility patterns from a database that is incrementally increased in a dynamic environment when compared to existing static ones, they generate candidate patterns in the mining process and require an additional process to identify actual patterns since they apply the overestimation concept, or they perform a pattern mining process without candidate generation, but there is a disadvantage in that they require a number of database scans for original or additional data. In other words, even if incremental utility pattern mining approaches have been suggested, previous approaches need at least two scans for incremental utility pattern mining irrespective of using tree or list structures. However, the approaches with multiple scans are actually not adequate for stream environments. In this paper, we propose an efficient algorithm for mining high utility patterns from incremental databases with one database scan based on a list-
2. Related work

2.1. Frequent pattern mining

Apriori [6] and FP-Growth [13] are Breadth First Search (BFS) and Depth First Search (DFS) based frequent pattern mining algorithms, respectively. Since the former requires a large number of database scans, the latter that scans databases only two times with a divide-and-conquer approach shows better performance. In general, FP-Growth-like methods outperform Apriori-like ones.

2.2. Static high utility pattern mining

Two-Phase [25] is the first algorithm that introduced and applied the overestimation concept in order to satisfy the anti-monotone property in utility pattern mining. Since the algorithm, Apriori-based FUM [32] and DCG+ [32] and FP-Growth-based UP-Growth [33] and UP-Growth+ [33], have been suggested. The approaches employing the overestimation model first extract valid candidate patterns satisfying the user defined minimum utility threshold at the first stage called Phase I and identify actual high utility patterns from the candidates at the final stage called Phase II. These approaches not only generate a large number of candidate patterns in the mining process but also perform an additional database scan to distinguish the actual pattern information from the mining process, which requires a high computational cost.

The methods applying the estimation model are required to conduct additional scanning in order to perform the identification process even if they construct their global data structures through a maximum of two database scans based on FP-Growth, so that a total of three scans are necessary. To solve the problem, a list data structure based method without candidate generation has been proposed. HUI-Miner [26] constructs a global data structure that stores utility information in the database of each candidate pattern called utility list through two scans of the database. In this case, the utility list for each candidate pattern is composed of entries corresponding to the number of transactions including the corresponding pattern. Each entry includes a tid that is an identifier of one of the transactions, an initl indicating a utility of the corresponding pattern in the transaction, and a rtul, which is the utility sum of items after the last item of the pattern in the transaction. After constructing the global data structure, the HUI-Miner algorithm performs a recursive mining process from the data structure to find out the high utility pattern information directly without generating candidates. In the recursive mining process, a list of local utilities for candidate patterns of length 2 or more is constructed, in which many tid comparison operations are performed. FHM [34] utilizes a data structure storing the overestimation utility information of candidates having a length of 2 and an estimated utility value that is not smaller than a user defined minimum utility called an Estimated Utility Co-Occurrence Structure (EUICS) improves mining performance by reducing computation. HUP-Miner [35] improves the performance of List-based high utility pattern mining by effectively reducing search space based on a list of partitioned utilities. d²HUP [28] is a newest list-based technique that generates candidates through a linear data structure called the Chain of Accurate Utility List (CAUL), a pattern enumeration strategy, and a high utility pattern growth approach combined with pruning by utility upper bound mining high utility patterns efficiently.

2.3. Incremental high utility pattern mining

IUM [36] and FIUM [36] are Apriori-based methods proposed for mining high temporal utility patterns from progressive databases. They apply the level-wise approach to perform the mining process with a number of constraints that result in the creation of database scans and candidates. IHUP [37] is an FP-Growth based algorithm for mining high utility patterns with the concept and improving mining performance of previous Apriori-based ones. Although IHUP has better performance than the previous ones, this method demands high computational time to identify actual high utility patterns from candidates because it generates a large number of candidates due to the applied overestimation concept. HUPID [38] solved the problem of IHUP by reducing overestimation utilities. However, this algorithm still needs an additional phase to extract actual high utility patterns from candidates. To improve mining performance of the previous overestimation concept based algorithms for incremental high utility pattern mining, we propose an efficient algorithm for mining high utility pattern mining without candidate generation on the basis of a list data structure. iCHUM [39] identifies promising items through two scans of original and additional data, reflects only the items in the data structure, and mines high utility pattern information from the constructed global data structure. The method also extracts candidates from the mining process by applying the overestimation concept and performs an additional scan to identify actual high utility patterns from the candidates. Moreover, since the promising item information is only stored, the approach has a disadvantage that additional data is required to update the promising and unpromising item information as the data is gradually added.

In order to solve the problem of the existing methods which need to identify actual patterns from the extracted candidate information in the process of mining the high utility pattern information from a dynamic database in an incremental environment, list-based methods without candidate generation have been proposed. HUI-list-INS [40] based on HUI-Miner is one of the methods for this purpose. It computes a TWU ascending order, which is the optimal sorting order, through two database scans and constructs a global data structure based on this information. Then, it mines high utility patterns from the data structure without generating candidates. Another method, EIH [41], reduces search space by reducing the number of local utility lists generated in the mining process based on FHM [34]. However, this method requires additional operations to create new utility lists for new data, and to merge them into data structures for existing original databases. Although these methods can discover high utility patterns in a dynamic environment without candidate generation, they have a disadvantage of requiring two database scans to determine the optimal TWU sorting order.

2.4. Differences from previous works

Our method is suggested for efficiently mining high utility pattern information from a dynamic database in an incremental
environment, and solves the following problems of existing techniques: (1) performance degradation caused by the overestimation concept [25] that is utilized by existing Apriori and FP-Growth based methods and (2) limitations of previous list based methods conducting two database scans for mining high utility patterns without candidate generation. That is, the proposed method has an advantage of mining high utility pattern information efficiently in a dynamic environment without candidate generation through only one scan to an incremental database. Based on the above contributions, we compare the proposed method with the existing methods for three aspects as follows: (1) static high utility pattern mining, (2) incremental high utility pattern mining based on the overestimation concept, and (3) incremental high utility pattern mining without candidate generation.

Static high utility pattern mining methods, FP-Growth based UP-Growth [33] and UP-Growth+ [33] and list structure based HUI-Miner [26], FHM [34], HUP-Miner [35], and dHUP [28], constructs their global data structures by performing two scans of a given database to mine high utility patterns satisfying a user-defined threshold. In particular, UP-Growth and UP-Growth+ require an additional scan to identify actual high utility patterns from candidates generated in the mining process by applying the overestimation concept. They only consider the static environment, so if new data is added, the existing data structure is removed and the mining process is repeated by performing two scans of the entire data including the added data. Thus, as database size increases, efficiency in a dynamic environment decreases. On the other hand, the proposed method not only constructs its data structure with a single database scan but also performs the mining process efficiently by reflecting only one scan of the added data without removing the existing data structure.

IUM [36] and FLM [36] based on Apriori, which are incremental high utility pattern mining methods with the overestimation concept, have a problem that requires a lot of database scans in the mining process. FP-Growth based IHUP [37] and HUPID [38], which are solutions to address the problem, construct their global tree data structures with only one database scan in contrast to the static methods, and then perform their mining processes. However, by using the overestimation concept, candidates are first extracted in the mining process, and an additional database scan is required to identify the actual high utility pattern information from the candidates. Although these methods can scan and reflect only one additional data in contrast to the static ones, they still have a limitation of requiring additional scanning due to the applied overestimation concept. iCHUM [46], an another incremental high utility pattern mining method based on FP-Growth, requires two database scans of original and additional data for building and updating the global data structure, and it conducts an additional scan for the utility pattern identification process. Hence, a total of three database scans is necessary. In addition, since the method stores only promising item information in its data structure, there is a constraint that whenever the data needs to be updated, unpromising item information needs to be reflected every time the data is updated. Based on the list data structure, the proposed method not only performs a full mining process with only one scan without candidate generation but also reflects newly added data efficiently.

Finally, HUI-list-INS [40] and EIHI [41], which are incremental high utility pattern mining methods based on a list data structure, discover high utility patterns from a dynamic database without candidate generation, whereas previous Apriori and FP-Growth based ones applying the overestimation concept require an additional database scan by generating candidates in their mining processes. However, when constructing their global list data structures, there is a disadvantage that they require scans of two original or additional data to compute a TWU ascending order, which is the optimized sorting order. In contrast, the proposed method not only builds and updates its global data structure with only one scan but also restructures the previously constructed data structure without additional database scans according to the optimal sorting order, through which the mining process can be performed efficiently. That is, existing methods, HUI-list-INS and EIHI, require a total of two database scans to construct their data structures according to the optimal sorting order and to mine high utility patterns without generating candidates in a dynamic environment. However, the proposed method not only constructs an optimized data structure with only one scan but also efficiently finds the same pattern information.

3. Incremental high utility pattern mining without candidate generation

In this paper, we suggest a list-based data structure for mining high utility patterns from incremental databases without candidate generation, construction and restructuring techniques for maintaining the proposed data structure, and an efficient algorithm for mining high utility patterns based on them. In the following subsections, we first provide preliminary knowledge for incremental high utility pattern mining and develop the proposed data structure. Moreover, we illustrate the construction, update, and maintenance techniques for our data structure and propose an incremental high utility pattern mining algorithm based on the list structure.

3.1. Preliminaries

In utility mining [10,34,42–46], a database, D, including a set of m distinct items, I = {i1, i2, ..., im}, is composed of n non-binary transactions, {T1, T2, ..., Tk}, in contrast to frequent pattern mining [6,13–15] with binary transactions. In addition, each transaction, Ti (Ti∈D and 1 ≤ i ≤ n), is a subset of I (|I| ≤ l) and has a unique id, called TID. Each item, ip (1 ≤ p ≤ m), has relative importance, called a candidate, and it is denoted as eu(ip). In addition, ip has a non-binary quantity value, called an internal utility, in a transaction, Td, and it is denoted as iu(ip, Td). The product of external and internal utilities of ip in Td, eu(ip) × iu(ip, Td), is defined as an item utility for the transaction, u(ip, Td). In Fig. 1, for instance, an item, A, has an external utility in the example database, eu(A)=3, and its non-binary occurrence, internal utility, in the first transaction, T1, is iu(A, T1)=1. In contrast to frequent pattern mining where a pattern, P = {i1, i2, ..., ik}, composed of k distinct items has a frequency value, the pattern has a utility value in utility mining.

**Definition 1. Utility of a pattern, P, in a transaction, Td, is u(P, Td) = Σiu(π, Td), where πp∈P and P ≤ Td.**

**Definition 2. Utility of a pattern, P, in a database, D, is u(P) = Σu(π, Td), where P ≤ Td and Td∈D.**

![Fig. 1. Example of non-binary database and profit table.](image-url)
Definition 3. Transaction utility of a transaction, \( T_d \), is \( tu(T_d) = \Sigma u(i_p, T_d) \), where \( i_p \in T_d \).

In the original database of Fig. 1, for example, since a pattern \((BC)\) is included in two transactions, \( T_2 \) and \( T_3 \), its utility in the database is the sum of \((BC)’s\) utilities in \( T_2 \) and \( T_3 \) according to Definition 2 as follows: \( u(BC) = u(BC, T_2) + u(BC, T_3) = u(B, T_2) + u(C, T_2) + u(B, T_3) + u(C, T_3) = 12 + 2 + 18 + 2 = 34 \). Besides, according to Definition 3, \( T_2 \) and \( T_3 \) have other items, and thus their transaction utilities are \( tu(T_2) = u(\{A, T_2\}) + u(B, T_2) + u(C, T_2) = 0 + 6 + 12 + 2 + 22 \) and \( tu(T_3) = u(B, T_3) + u(C, T_3) + u(D, T_3) + u(E, T_3) = 18 + 2 + 2 + 6 = 28 \).

The sum of utilities of all transactions in a database, \( D \), is \( u(D) = \Sigma tu(T_d) \), where \( T_d \in D \) and refers to a database utility. When given a minimum utility threshold, \( \delta \), from a user, a minimum utility, minutil, is defined as \( u(D) \times \delta \). If a pattern utility of \( P \) is no smaller than minutil, \( u(P) \geq \text{minutil} \), then \( P \) is a high utility pattern. Otherwise, it is called a low utility pattern. High utility pattern mining signifies the discovery of all patterns satisfying a given user-defined minimum utility threshold.

In the overestimation model [25] for maintaining the antimonotone property in utility mining, each pattern, \( P \), has an overestimated utility, called transaction weighted utilization \( (tuw) \), and it is defined as \( tuw(P) = \Sigma tu(T_d) \), where \( P \in T_d \) and \( T_d \in D \). That is, in mining processes, high transaction weighted utilization patterns with no smaller \( tuw \) values than a minimum utility value are extracted as candidates and actual high utility patterns are identified from them. Here, if an overestimation utility of \( P \) is no smaller than a threshold value, minutil, then it is a promising pattern. Otherwise, it is called an unpromising pattern. For instance, a database utility of the original database in Fig. 1 is \( u(D) = tu(T_1) + tu(T_2) + tu(T_3) + tu(T_4) = 9 + 22 + 28 + 10 = 69 \) and a minimum utility is \( \text{minutil} = u(D) \times \delta = 69 \times 0.5 = 34.5 \) when a given threshold is \( \delta = 0.5 \) (50%). Accordingly, \((BC)\) with a smaller utility than minutil is a low utility pattern in this case. Nevertheless, since its overestimated utility, \( tuw(BC) = tu(T_2) + tu(T_3) = 22 + 28 = 50 \), is larger than minutil, the pattern is extracted as a candidate in mining processes. Although \((BC)\) is not a high utility pattern, there is a possibility to mine a high utility pattern that is a super pattern of \((BC)\) because of its overestimation utility, and thus search space for all super patterns of \((BC)\) is checked.

In utility mining without candidate generation [26–28], meanwhile, the following definition is utilized to reduce search space without the overestimation concept.

Definition 4. When given a transaction, \( T_d \), containing a pattern, \( P \), a set of items after \( P \) in \( T_d \) is denoted as \( T_d/P \) and a remaining utility of \( P \) in \( T_d \) is defined as \( ru(P, T_d) = \Sigma u(i_p, T_d) \), where \( i_p \in T_d/P \).

In the approach mentioned above, for each pattern \( P \), a utility upper-bound is computed by the sum of its utility and remaining utility based on Definition 4, and an invalid search space with a smaller utility upper-bound than minutil is eliminated in a mining process. In Fig. 1, for example, a pattern, \((BD)\), is included in two transactions, \( T_2 \) and \( T_3 \), and its remaining utilities in the transactions are \( ru(BD, T_2) = 0 \) and \( ru(BD, T_3) = u(E, T_3) = 6 \). Thus, the pattern’s remaining utility in the database is \( ru(BD) = ru(BD, T_2) + ru(BD, T_3) = 6 \). In addition, since a pattern utility of \((BD)\) is \( u(BD) = u(B, T_2) + u(B, T_3) = 14 + 20 = 34 \), its utility upper-bound is \( u(BD) + ru(BD) = 34 + 6 = 40 \).

3.2. Overall process of incremental high utility pattern mining without candidate generation

Fig. 2 is an overall process of the proposed method. Our framework first constructs a global list-based data structure through one scan for the original database and restructures the data structure by sorting lists according to a twu ascending order. If a set of new transactions is added to the previous database, the proposed method scans only the added data once, updates the constructed and restructured data structure by reflecting the added data, and restructures the updated data structure again according to an updated twu ascending order. After the restructuring process for the global data structure with the twu ascending order, if a user requests high utility mining with a minimum utility threshold, our algorithm mines all high utility patterns satisfying the threshold recursively from the data structure without candidate generation.

3.3. Construction of global list-based data structure for original database

Our list based global data structure consists of sets of lists, called utility lists, where utility information for candidate patterns is stored and maintained. Each utility list is created for a candidate itemset, \( \{i_p\} \), with the length 1. Here, the utility list is composed of entries to store utility information of the pattern in transactions containing \( \{i_p\} \). That is, the number of entries in the utility list for \( \{i_p\} \) is the same with that of transactions including \( \{i_p\} \) in a given database. For each transaction, \( T_d \), containing \( \{i_p\} \), an entry with utility information of \( \{i_p\} \) in \( T_d \) has three elements: a TID of \( T_d \), a pattern utility of \( \{i_p\} \) in \( T_d \), \( u(i_p, T_d) \), and a remaining utility in \( T_d \) after \( \{i_p\} \), \( ru(i_p, T_d) \), according to Definition 4. The second and last elements are denoted as \( uu \) and \( ru \), respectively. In contrast to previous methods for static databases [26], the last element, \( ru \), is initialized as zero without computation when construction and update steps and calculated in a restructuring step after a sorting order is decided. For this purpose, \( twu \) values of items are computed in a transaction insertion step.

To deal with incremental databases efficiently, the proposed method constructs the global data structure through one database scan and our data structure creates utility lists for all database items since invalid items can be valid according to database increments. That is, in contrast to previous static methods that identify invalid items with two scans and build their global data structures with valid item information, our approach reflects all item information to the data structure. In summary, the data structures used in the proposed scheme are constructed in a single database scan, unlike the list-based ones used in existing schemes, for efficiently mining high utility patterns from a gradual database in a dynamic environment. In the process, all the promising and unpromising item information is stored, and the \( ru \) values are calculated at a later optimization, so that the values are initially initialized to zero.

The proposed algorithm firstly scans the original database once to construct the global list data structure. In the database scan, for each transaction, \( T_d = \{i_1, i_2, \ldots, i_k\} \), our method sorts items in the transaction according to an alphabetic order, \( T_d’ = \{i_1’, i_2’, \ldots, i_k’\} \), and processes them from the first item, \( i_1’ \). If there is no utility list for the processing item in the global data structure, the list is added first. Because the current processing transaction is \( T_d’ \), the proposed algorithm makes a new entry to store utility information of \( \{i_1’\} \) in \( T_d’ \) and attaches the entry to the list for \( \{i_1’\} \). After that, the first and second elements of the entry are set to TID of \( T_d’ \) and a utility of \( \{i_1’\} \) in \( T_d’ \), \( u(i_1’, T_d’) \). The last one, \( ru \), is assigned in a restructuring step according to an optimized sorting order, and thus its value is initialized as zero in this construction step. In a similar way, for the rest items in \( T_d’ \), our method makes their entries and sets utility information of the entries. In this stage, \( twu \) values for the items are increased by a transaction utility of \( T_d’ \). The construction step of the global data structure is finished after all transactions in the original database are processed.
Example 1. Consider the original database in Fig. 1. Our algorithm sorts items in the first transaction, \(T_1 = \{A, D, G\}\), according to the initial order, alphabetic order, and its result is \(T'_1 = \{A, D, G\}\). To insert the sorted transaction, the algorithm creates a utility list for a candidate pattern composed of the first item, \(\{A\}\), and an entry for \(T'_1\). Three elements of the entry, TID, iu, and ru, are set to 1, \(u(A, T'_1) = 3\), and 0, respectively. Through same way, the rest items, \(D\) and \(G\), are processed, utility lists for \(\{D\}\) and \(\{G\}\) are built, and entries for \(T'_1\) are added. Here, their TID elements are assigned as 1, iu values are set to \(u(D, T'_1) = 2\) and \(u(G, T'_1) = 4\), and ru are initialized as zero. Fig. 3(a) is a constructed global data structure by inserting the first transaction, \(T_1\). Then, the next transaction, \(T_2 = \{A, B, C, D\}\), is also sorted in the alphabetic order, and its result is \(T'_2 = \{A, B, C, D\}\). In its insertion step, only two utility lists for \(\{B\}\) and \(\{C\}\) are created since utility lists for \(\{A\}\), \(\{D\}\), and \(\{G\}\) are built when processing the first transaction, \(T_1\). After that, four entries are attached to the lists for \(\{A\}\), \(\{B\}\), \(\{C\}\), and \(\{D\}\) and their TID values are set to 2. In addition, their iuelements are assigned as \(u(A, T'_2) = 6\), \(u(B, T'_2) = 12\), \(u(C, T'_2) = 2\), and \(u(D, T'_2) = 2\), and ru are initialized as zero. Fig. 3(b) shows the result of inserting transactions from \(T_1\) to \(T_2\). The rest two transactions, \(T_3 = \{B, C, D, E\}\) and \(T_4 = \{A, D, F\}\), are also processed, and in this stage, utility lists for \(\{E\}\) and \(\{F\}\), four entries for \(T'_3\), and three entries for \(T'_4\) are made. Fig. 3(c) is a global data structure constructed with all the transactions, \(T_1, T_2, T_3, \text{and } T_4\), in the original database.

In the above example, whenever a sorted transaction, \(T_d'\), is inserted, twu values of items in \(T_d'\) are increased by \(tu(T_d')\), and after the construction of a global data structure, all twu values are computed and these are used to decide an optimized sorting order for the next restructuring step. Fig. 4 shows the updated twu values in the construction step of the global data structure in Fig. 3.

3.4. Restructure of constructed global list-based data structure

In dynamic environments, data are increased gradually and added to original databases, and in this stage, optimized sorting orders are changed and invalid patterns can be valid. Therefore, the proposed method in this paper utilizes all information for the construction of a global data structure in order to handle the dynamic environments in contrast to previous static methods with only valid information. Since the proposed method scans given data only once, the optimized sorting order is decided after the construction phase. In other words, the order is set with twu values updated in the phase. Utility lists created to store utility
Lemma 1. ru values computed based on the previous sorting order are invalid after the global data structure is restructured in accordance with the change of its sorting order.

Proof. Each ru value of an item in an entry refers to the sum of utilities of subsequent items after the item in a transaction according to Definition 4. In addition, item utilities in the transaction are not changed even if a sorting order is changed since a utility of each item is calculated by the product of the item’s quantity in the transaction and profit in a database and the quantity and profit are fixed values regardless of the sorting order. However, a set of subsequent items for the item is changed when the sorting order is updated, and accordingly the sum of utilities of the subsequent items. Hence, the ru value becomes invalid in accordance with the change of the sorting order.

The proposed algorithm employs tuw information updated in the construction step to decide an optimized sorting order, i.e., tuw ascending order. When the order, \( O = \{i_1', i_2', \ldots, i_m'\} \), where \( tuw(i_1') > tuw(i_2') > \ldots > tuw(i_m') \), is decided on the basis of tuw information, utility lists in the global data structure are rearranged according to \( O \). After that, our method traverses all utility lists starting from one for \( \{i_m'\} \) with the largest tuw value and updates ru values. In this traversal step, the proposed method uses a temporal array to compute sums of iuvalues with the same TIDs. That is, for each entry, a sum value in the array for its TID is increased by the entry’s iuvalue. Before raising the sum value, the entry’s ru value is assigned as the previous sum value. When the restructuring step completes, the temporal array is eliminated.

Example 2. Consider the constructed global data structure for the original database and tuw table in Figs. 3(c) and 4. In the tuw table of Fig. 4, G and D have the smallest and largest tuw values, respectively. Hence, their tuw ascending order is \( [G > F > E > A > B > C > D] \) and utility lists in Fig. 3(c) are reordered as shown in Fig. 5. After the rearrangement of utility lists, for the last utility list for (D), the proposed algorithm sets sum values in a temporal array to corresponding iuvalues. Since there are four entries in the list and there is no previous sum value in the array, their ru values have no change. Instead, sum values for TIDs, 1, 2, 3, and 4, are set to corresponding iuvalues, 2, 2, 2, and 1. Fig. 6(a) is the result of processing the last utility list for [D] in which the temporal array with four elements is also included. Then, our method deals with the next utility list for (C) with two entries for TIDs, 2 and 3. Because the temporal array already has two elements for the TIDs, ru values in the entries of the list are assigned as the stored sums, 2 and 2. In addition, the stored values increased by iuvalues in the entries, and Fig. 6(b) is the result of processing the utility lists for [D] and [C]. In a similar manner, ru values in a utility list for [B] are set to 4 and sums for TIDs, 2 and 3, are increased by 12 and 16. Fig. 6(c) shows the result after updating three lists for [D], [C], and [B]. When dealing with the next utility list for [A], sums of iuvalues, 2, 16, and 1, for three transactions, T1, T2, and T4, are used to set corresponding ru values, and then iuvalues in the list are added to the sums. Fig. 6(d) is the result when the process for the utility list for [A] is finished. As shown in Fig. 6(e), the global data structure is restructured by processing all the utility lists including the rest ones for (E), (F), and (G).
3.5. **Update of global list-based data structure**

Whenever a set of new transactions added to the current database, the proposed method reflects them to a global data structure by scanning the transactions only once and updates the data structure by restructuring it on the basis of changed utility information. That is, the newly added data are sorted in the current twu ascending order and then inserted in contrast to the original data reordered according to an alphabetic order. Compared with the existing methods, the proposed method scans the added data only once, rearranges them according to the current sorting order, reflects them in the constructed global data structure, and computes the changed optimal sorting order according to the update, thus rebuilding the data structure without further scanning. In other words, the data structure used in the proposed method is optimized efficiently according to the optimal sorting order with only one database scan of the original or additional data.

For each new transaction, \( T_d = \{i_1, i_2, \ldots, i_k\} \), the proposed method sorts each entry in the current twu ascending order and its result is \( T_d' = \{i'_1, i'_2, \ldots, i'_k\} \). Then, it inserts each entry, \( i'_p \), as a candidate pattern \( \{i'_p\} \). Here, our algorithm makes an entry for \( T_d' \) in a utility list for \( \{i'_p\} \) and assigns the first and second elements in the entry, \( \text{TID and } i_u \), as \( \text{Td} \)’s TID and \( \{i'_p\} \)’s TID. The last element, \( ru \), with zero is also not updated in this step. In addition, whenever a new transaction is inserted, \( twu \) values for all items in the transaction are increased by its transaction utility, \( tu(T_d') \). After inserting all transactions, our proposal decides a new twu ascending order and rearranges utility lists according to the order. Then, the algorithm conducts a restructuring step starting from the last utility list for a candidate pattern composed of an item with the largest twu value. The reason why all \( ru \) values in the global data structure are initialized as zero is the previous sorting order can be changed and in that case they have to be recomputed. In the restructuring step with the new data, a temporary array with accumulated \( iu \) values for TIDs is also utilized to set the \( ru \) values.

**Example 3.** Consider the added transactions in Fig. 1 and the constructed and restructured global data structure for the original database in Fig. 6(e). In Fig. 1, there are two data sets, \( db_{1+} \) and \( db_{2+} \), and each set contains two new transactions. Since the twu ascending order for the original database is \( G > F > E > A > B > C > D \), our method sorts the first transaction \( T_3 = \{B, C, F\} \) in \( db_{1+} \) according to the order, and its results is \( T_3' = \{F, B, C\} \). To insert the first item, \( F \), the proposed algorithm creates an entry for \( T_3' \) in the utility list for \( F \) and reflects its utility information to the entry. That is, the newly added entry’s TID, \( i_u \), and \( ru \) elements are set to 5, \( u(F, T_3') = 6 \), and 0, respectively. Entries for the rest items, B and C, are added to the list and their TID values are assigned as 5. In addition, \( iu \) elements of the entries are updated as \( u(B, T_3') = 6 \) and \( u(C, T_3') = 6 \) and \( ru \) values are initialized as zero. Fig. 7(a) is the result of processing the new transaction information to the global data structure. The last transaction, \( T_6 = \{C, D, E\} \), in \( db_{2+} \) is also processed and its sorted result, \( T_6' = \{E, C, D\} \) is inserted into the data structure as shown in Fig. 7(b). In the above update step, the twu table is updated by reflecting transaction utilities of \( T_5' \) and \( T_6' \) and accordingly, the twu ascending order is changed into \( G > F > E > A > B > C > D \). Based on the changed sorting order, \( ru \) values are updated and the utility lists are rearranged as shown in Fig. 5(a).

**Example 4.** Consider the added transaction data and the constructed and restructured global data structure in Fig. 1 and Fig. 9(a). To insert the last set of new transactions, \( db_{2+} \), our proposal processes them from the first transaction \( T_7 = \{C, E, F\} \). Since the previous changed sorting order by the insertion of \( db_{1+} \) is \( G > F > E > A > B > C > D \), \( T_7 \) is sorted in the order and its result is \( T_7' = \{F, E, C\} \). The method creates entries with TID 7 in three utility lists for \( F \), \( E \), and \( C \) and then updates their utility information for \( T_7' \). Besides, the entries’ \( iu \) elements are assigned as \( u(F, T_7') = 6 \), \( u(E, T_7') = 3 \), and \( u(C, T_7') = 2 \) and \( ru \) values are initialized as zero. In this insertion process of \( T_7' \), twu values of \( F, E \), and \( C \) are increased to \( tuu(F) = 39 \), \( tuu(E) = 47 \), and \( tuu(C) = 87 \) respectively as shown in Fig. 8. In the same way, the last transaction, \( T_8 = \{A, B, E\} \), is also processed, and its sorted result is \( T_8' = \{E, A, B\} \). After inserting the last data set, \( db_{2+} \), the sorting order finally becomes \( G > F > E > A > D > B > C \). Fig. 9 is the restructured global data structure with \( db_{1+} \) and \( db_{2+} \) according to the final order.

3.6. **Mining high utility patterns from global list structure**

After the global data structure is constructed with the original database or updated with increased data and then restructured according to a twu ascending order, the proposed method computes a minimum utility by multiplying total utility and user-defined threshold when a mining request is given. Then, our algorithm performs a series of mining processes recursively from a utility list for a promising candidate pattern composed of an item with the smallest twu value. Note that this series of processes is performed based on the previous approach, HUI-Miner [26], with utility lists.

Let \( \{i_1 > i_2 > \ldots > i_m\} \) and \( UL(i_p) \) be the current sorting order and a utility list for a candidate pattern, \( \{i_p\} \) \( (1 \leq p \leq m) \), with the length 1, respectively. The proposed method multiplies a minimum utility threshold, \( \delta \), that is given from a user and the database total utility, \( u(D) \), to obtain a minimum utility value, minutil. After that, it selects each utility list from the first one, \( UL(i_1) \), and performs mining operations recursively with utility lists after the current one in the global data structure. For the selected utility list for a candidate, \( \{i_p\} \), with the length 1, our algorithm compares the sum of \( iu \) values in the list with minutil and outputs \( ip \) as a high utility pattern if the sum is no smaller than minutil. In addition, if the sum of the sums of \( iu \) and \( ru \) values in the list is larger than or equal to minutil, then the method conducts mining processes for search space where a prefix pattern is \( \{ip\} \) with utility lists, \( \{UL(ip_1), UL(ip_1), \ldots, UL(ip_m)\} \), after \( UL(ip) \). The reason for this is that the sum refers to a utility upper-bound of patterns that can be generated from \( \{ip\} \). That is, they are super patterns and \( \{ip\} \) is their prefix pattern. It signifies that no super pattern of \( \{ip\} \) has a utility value larger than or equal to minutil. For a utility list for \( \{ip\} \) with no smaller utility upper-bound than minutil, the proposed method builds a utility list, \( UL(ip_{ipq}) \), with the length 2 based on each utility list, \( UL(\{ip\}) \) \( (1 \leq p < q \leq m) \), after \( UL(ip) \) in the global data structure. The following is a process to create \( UL(ip_{ipq}) \) with utility information of \( UL(ip_p) \) and \( UL(ip_q) \). For each
Fig. 6. Restructuring process for reordered global data structure in Fig. 5.
pair of entries with the same TID in both \(UL(i_p)\) and \(UL(i_q)\), our algorithm makes an entry for the TID in \(UL(i_p i_q)\) and sets its \(iu\) and \(ru\) values, \(iu(TID, UL(i_p i_q))\) and \(ru(TID, UL(i_p i_q))\), to the sum of \(iu\) values in the pair and their minimum \(ru\) value, respectively. That is, \(iu(TID, UL(i_p i_q)) = iu(TID, UL(i_p)) - iu(TID, UL(i_q))\) and \(ru(TID, UL(i_p i_q)) = \text{min}(ru(TID, UL(i_p)), ru(TID, UL(i_q)))\).

**Example 5.** Consider the whole database and the restructured global data structure in Fig. 1 and 9(b) and assume that a minimum utility threshold is set to 0.15 (15%). Since the total database utility is \(u(D) = 124\), a minimum utility is \(0.15 \times u(D) = 0.15 \times 124 = 18.6\). For the first utility list, \(UL(G)\), for \{G\}, the proposed method checks whether the sum of \(iu\) values, \(iu(1, UL(G)) = 4\), is no smaller than minutil or not and then does not extract \{G\} as a high utility pattern. In addition, since \{G\}'s utility upper-bound, \(iu(1, UL(G)) + ru(1, UL(G)) = 4 + 5 = 9\), is no larger than or equal to minutil, search space where its prefix pattern is \{G\} is not processed. In other words, there is no need to conduct a series of mining processes including creation steps for utility lists for super patterns of \{G\} with \(UL(F), UL(E), UL(A),\)
UL(D), UL(B), UL(C)) after UL(C). For the next utility list, UL(F), the proposed method decides to perform mining operations for search space where its prefix pattern is (F) by comparing the sum of iuvalues, iu(4, UL(F)) + iu(5, UL(F)) + iu(7, UL(F)) = 3 + 6 + 6 = 15, and its utility upper-bound, (iu(4, UL(F)) + iu(5, UL(F)) + iu(7, UL(F)) + ru(4, UL(F)) + ru(5, UL(F)) + ru(7, UL(F))) = 15 + 24 = 39, with minutil although (F) is not a high utility pattern. Accordingly, utility lists for candidate patterns with the length 2 for the prefix pattern (F) on the basis of utility lists after UL(F). Because UL(F) and UL(E) contain entries for TID 7, a utility list, UL(FE), with only one entry for the TID is built and the entry’s iuand ru values are iu(7, UL(FE)) = iu(7, UL(F)) + iu(7, UL(E)) = 6 + 3 = 9 and ru(7, UL(FE)) = min(ru(7, UL(F)), ru(7, UL(E))) = min(5, 2) = 2, respectively. Through the same manner, for UL(F) and UL(A), our algorithm creates UL(FA), adds an entry for TID 4 to the list, and sets the entry’s iuand ru values to iu(4, UL(FA)) = iu(4, UL(F)) + iu(4, UL(A)) = 3 + 6 = 9 and ru(4, UL(FA)) = min(ru(4, UL(F)), ru(4, UL(A))) = min(7, 1) = 1. Then, rest utility lists, UL(D), UL(B), and UL(C), are processed in a similar way to UL(F), and UL(FD), UL(FB), and UL(FC) are created. Fig. 10 shows the result of utility lists for candidate patterns with the length 2 where their prefix pattern is (F).

In the same way of building utility lists for candidate patterns with the length 2 in Example 5, the proposed method creates utility lists for other candidates with the length 2 where their prefix patterns are {E}, {A}, {D}, and {B}. The reason for this is that UL(E), UL(A), UL(D), and UL(B) have one or more subsequent utility lists and their utility upper-bounds are 18.6 that is no smaller than minutil. That is, the upper-bounds for super patterns of {E}, {A}, {D}, and {B} are 59, 46, 46, and 52 computed with utility information in UL(E), UL(A), UL(D), and UL(B). The reason no utility list for a candidate pattern with the length 2 from UL(C) is generated is there is no subsequent list after UL(C). Fig. 11(a), (b), (c), and (d) are created utility lists for candidate patterns with the length 2 where their prefix patterns are {E}, {A}, {D}, and {B}. Meanwhile, in the above steps, utility values of {E}, {A}, {D}, {B}, and {C} are computed with information about iuvalues as follows: u(E) = 18, u(A) = 21, u(D) = 8, u(B) = 42, and u(C) = 16. By comparing these utilities with minutil, (A) and (B) are extracted as high utility patterns. Overall, from the global data structure in Fig. 9, high utility patterns with the length 1, {A} and {B}, are generated and then search space for candidate patterns with the length 2 derived from prefix patterns, {F}, {E}, {A}, {D}, and {B}.

When utility lists for candidates with the length 2 are built from a utility list, UL(ip), for each pattern, {ip}, the proposed method creates utility lists for candidates with the length 3, UL(ipqkq+1), UL(ipqkq+2), ..., UL(ipqkm), where their prefix pattern is {ip, iq, k}. Here, if a utility of {ip, iq, k}, defined as the sum of iuvalues in entries of UL(ipqk), is no smaller than minutil, {ip, iq, k} is generated as a high utility pattern. However, if a utility upper-bound computed with information in UL(ipqk) is not larger than or equal to minutil, search space for super patterns of {ip, k} is not processed. The creation process of UL(ipqk) (1 ≤ p < q < r ≤ m) for a candidate pattern, {ip, iq, kr}, with the length 3 where its prefix is {ip, iq, k} is conducted as follows. For each pair of entries for the same TID in UL(ipqk) and UL(ipqk), our algorithm adds a new entry to UL(ipqk) and sets the entry’s iuand ru values. Although the ru value, ru(TID, UL(ipqk)), is assigned as min(ru(TID, UL(ipqk)), ru(TID, UL(ipqk))), the iuvalue, iu(TID, UL(ipqk)), cannot be calculated through the sum of iuvalues in UL(ipqk) and UL(ipqk), iu(TID, UL(ipqk)) + iu(TID, UL(ipqk)). The reason for this is that a utility of ip in a transaction, T, with the TID value is duplicated in iu(TID, UL(ipqk)) and iu(TID, UL(ipqk)). In other words, iu(TID, UL(ipqk)) + iu(TID, UL(ipqk)) is (u(ip, T) + u(qk, T)) + (u(ip, T) + u(qk, T)) = 2u(ip, T) + u(qk, T) + u(qk, T). Hence, iu(TID, UL(ipqk)) is set to the difference of the duplicated value, u(ip, T), from the sum, iu(TID, UL(ipqk)) + iu(TID, UL(ipqk)), as follows: iu(TID, UL(ipqk)) + iu(TID, UL(ipqk)) − iu(TID, UL(ipqk)).

Example 6. Consider the utility lists for candidates with the length 2 where their prefix is (F) in Fig. 10 and assume that minutil is 18.6 like in Example 5. The proposed method selects the first list for {FE} and makes utility lists for candidates that are super patterns of {FE} based on UL(FA), UL(FD), UL(FB), and UL(FC). In this stage, the method computes u(FE) through the sum of iuvalues of entries in UL(FE), and {FE} is not extracted as a high utility pattern since the utility is smaller than minutil. Moreover, a utility upper-bound of super patterns of {FE} is ru(7, UL(FE)) = ru(7, UL(FE)) = 9 + 2 = 11, and thus search space for the patterns is not processed. In other words, no mining process is conducted for utility lists for candidates with the length 3 based on UL(FE) and set of subsequent lists, UL(FA), UL(FD), UL(FB), and UL(FC). After that, no high utility pattern is outputted from UL(FA), UL(FD), and UL(FC) since utilities of {FA}, {FD}, and {FB}, u(FA) = 9, u(FD) = 4, and u(FB) = 12, are no larger than or equal to minutil. Besides, calculated their utility upper-bounds, ru(4, UL(FA)) + iu(4, UL(FA)) = 10, ru(4, UL(FD)) + iu(4, UL(FD)) = 4, and ru(5, UL(FB)) + iu(5, UL(FB)) = 18, are also smaller than minutil, and as a result search space for the three patterns is not traversed. For the last utility list, UL(FC), the proposed method generates {FC} as a high utility pattern because its utility is u(FC) = 20. But, there is no subsequent list after UL(FC) and its super patterns’ search space is not processed.

In Example 6, no super patterns of {FE}, {FA}, {FD}, {FB}, and {FC} is generated from their utility lists because their utility upper-bounds are smaller than minutil. However, {FC} only is a high utility pattern due to its larger utility, u(FC) = iu(5, UL(FC)) + ru(7, UL(FC)) = 12 + 8 = 20, then minutil. The next example is a process of dealing with utility lists for candidates with the length 2 where a prefix pattern is {E}. In the example, utility lists for candidate patterns with the length 3 in Fig. 11(a) due to their no smaller utility upper-bounds than minutil.

Example 7. Consider utility lists for candidates with the length 2 where their prefix patterns are {E}, {A}, {D}, and {B} in Fig. 11. The proposed method conducts a series of mining operations with UL(FA), UL(ED), UL(EB), and UL(EC) for {EA}, {ED}, {EB}, and {EC} in Fig. 11(a). In this stage, it selects the first list, UL(FA), computes a utility upper-bound of {EA} using information in subsequent lists, UL(ED), UL(EB), and UL(EC), before building utility lists for super patterns of {EA} with the length 3, and its result is iu(8, UL(FA)) + ru(8, UL(FA)) = 12 + 6 = 18. That is, search space for the super patterns is not processed. Since a utility value of {EA}, u(EA) = 12, derived based on iuvalues in UL(FA) is smaller than minutil, the pattern is not generated as a high utility pattern. Next, for UL(ED), a utility upper-bound of super patterns of {ED} is ru(3, UL(ED)) + ru(6, UL(ED))) = 15 + 24 = 39 and larger than minutil, and thus utility lists for {EDB} and {EDC} are created. However, {ED}'s utility is u(ED) = iu(3, UL(ED)) + iu(6,
UL(ED)) = 8 + 7 = 15. Hence, [ED] is not generated as a high utility pattern. Because there is a pair of entries for TID 3 in UL(ED) and UL(EB), the proposed method makes UL(EDB) containing an entry for the TID. Then, our algorithm sets the new entry's ru and ru values to ru(3, UL(ED)) + ru(3, UL(EB)) - ru(3, UL(E)) = 8 + 24 - 6 = 26 and min(ru(3, UL(ED)), ru(3, UL(EB))) = 2, respectively. In addition, UL(EDC) with two entries is built since UL(ED) and UL(EB) have entries for TIDs 3 and 6. The iu values of new entries in UL(EDC) are assigned as follows: iu(3, UL(EDC)) = iu(3, UL(ED)) + iu(3, UL(EB)) - iu(3, UL(E)) = 8 + 8 - 6 = 10 and iu(6, UL(EDC)) = iu(6, UL(ED)) + iu(6, UL(EB)) - iu(6, UL(E)) = 7 + 7 - 2 = 12. After that, their ru values are initialized as zero: ru(3, UL(EDC)) = min(ru(3, UL(ED)), ru(6, UL(EDC))) = 0 and ru(6, UL(EDC)) = min(ru(6, UL(ED)), ru(6, UL(EB))) = 0. Fig. 12(a) and (b) are the results of utility lists with the length 3 created from other lists for {ED} and {EB} in Fig. 11(a).

In the same manner for utility lists in Examples 6 and 7, the proposed method processes utility lists in Fig. 11(b), (c), and (d). Utilities of {AD} and {AB} obtained from utility lists for super patterns of {A} in Fig. (b) are 20 and 30, and thus they are generated as high utility patterns and their search space is traversed. The reason for is that utility upper-bounds of their super patterns are 34 and 32. In contrast, {AC}'s utility and a utility upper-bound of its supper patterns are 8, and thus not only {AC} is not extracted as a high utility pattern but also search space for the super patterns is not processed. Then, from utility lists for super patterns of {D}, {DB} is extracted as a high utility pattern with no smaller utility than minutil, and their search space is also traversed. Lastly, {BC} is only a high utility pattern generated from utility lists for candidate patterns in Fig. 11(d) where their prefix is {B}. Then, this mining process is finished since there is no subsequent list after UL(BC). Fig. 13 is the result of utility lists built from ones in Fig. 11(b) and (c).

In summary, from utility lists for candidate patterns with the length 2 in Fig. 11 where their prefix patterns are {E}, {A}, {D}, and {B}, six high utility patterns, {EB}, {EC}, {AD}, {AB}, {DB}, and {BC} are generated and then utility lists for {EDB}, {EDC}, {EBC}, {ADB}, {ADC}, {ABC}, and {DBC} are built.

As we can know, there is the difference between creations of utility lists for 2-itemsets and 3-itemsets when computing iu values.
of entries, but processes of building utility lists for \( k + 1 \)-itemsets from ones for \( k \)-itemsets \((k > 2)\) are the same.

**Example 8.** Consider utility lists for \((E\mathbf{B})\) and \((E\mathbf{C})\) in Fig. 12(a) where their prefix is \((E\mathbf{D})\). The proposed method first extracts \((E\mathbf{D})\) as a high utility pattern from a utility list for \((E\mathbf{D})\) because its utility, \(u(E\mathbf{D}) = iu(3, UL(E\mathbf{D})) = 26\), is larger than minutu. Besides, a utility list for \((E\mathbf{BC})\) based on \(UL(E\mathbf{B})\) and \(UL(E\mathbf{D})\) is created since a utility upper-bound of super patterns of \((E\mathbf{D})\) is \(ru(3, UL(E\mathbf{D})) = 26 + 2 = 28\). Fig. 14 shows the utility lists for \((E\mathbf{BC})\) built in accordance with the above process.

### 3.7. Algorithm for incremental mining of high utility patterns

Fig. 15 is the proposed method's algorithm composed of a main procedure to control the whole mining processes, a restructuring procedure to arrange a constructed or updated global data structure according to a TWU descending order, and a mining procedure to mine high utility patterns satisfying a given threshold from the restructured global data structure.

The main procedure reads each transaction in the original database or a newly inserted data and sorts items in the transaction in the current sorting order (lines 1 to 2). For each item in the sorted transaction, the procedure checks whether there is a utility list for a pattern composed of the item in the global data structure and builds the utility list if there is no list for the pattern (lines 3 to 5). Moreover, it adds an entry for the sorted transaction to a utility list for each item and then sets the entry's isand ru values to a utility of the item in the transaction and zero, respectively (lines 6 to 7). When the construction or update step of the global data structure for all transactions is finished, the main procedure calls the Restructure procedure to arrange utility lists in accordance with the optimal sorting order and update utility information in the lists (line 8). After completion of the restructuring step, when a mining request is given from a user, the main procedure calls the mining procedure to mine high utility patterns satisfying a user-defined threshold (lines 9 to 13).

The restructuring procedure firstly initializes all ru values in a set of utility lists, ULS, in the global data structure as zero and then reorders the lists in a TWU ascending order (lines 14 to 15). After that, the procedure selects each utility list in sequence and ru values of entries in the list with a temporal array (lines 16 to 19).

The last mining procedure takes a set of utility lists, ULS, for candidate patterns of the current prefix pattern, PPF, a set of high utility patterns, HUPS, a minimum utility value, minutu, and the depth of the current mining process, depth. For each utility list, \(UL(P)\), in ULS for a candidate pattern, \(P\), if a utility of \(P\) is no smaller than minutu, then the procedure adds \(P\) to HUPS (lines 20 to 22). Then, it creates a set of utility lists, CULS, for super patterns of \(P\) if their utility upper-bound is larger than or equal to minutu (lines 23 to 24). After that, for each subsequent utility list, \(UL(Q)\), after \(UL(P)\) in ULS, the procedure inserts a new utility list for \(P-Q\)-PPF into CULS (lines 25 to 26). In order to make entries in the utility list, for each pair of entries for the same TID in \(UL(P)\) and \(UL(Q)\), the mining procedure adds an entry for the TID to \(UL(P-Q\text{-PPF})\) and sets the entry's isand ru values (lines 27 to 32). Finally, the procedure calls itself recursively to conduct mining processes after all conditional utility lists are built and added to CULS when a prefix pattern is \(P\) (line 33).

An example of the progress of the proposed method according to the above algorithm is as follows. In the first step, the four transactions in the original database of Fig. 1 are inserted in order, and the global data structure is constructed as shown in Fig. 3(c). In this step, utility lists for candidate patterns with the length of 1 are key components, and the process of inserting each transaction according to the initial sorting order and constructing them are the key operations. In the second step, the constructed global data structure is restructured as shown in accordance to a TWU ascending order in Fig. 4, and Fig. 5 shows the result. Therefore, the key components and operations at this step are the restructured global utility lists according to the TWU ascending order and the process of rearranging the initial data structure. In the third step, ru values of the rearranged global data structures are recalculated according to the optimal sorting order as shown in Fig. 6. Thus, the rearranged utility lists are the key components, and the calculation process of the ru values is the key operation. In the fourth step, if there is incrementally added new data, as shown in Fig. 9(b), the data is reflected into the global data structure and utility lists in the data structure are rearranged, where key components and operations are the utility lists reflecting the new data, and the processes of reflecting the new data and rearranging the lists. In the final step, recursively mining process is performed with the restructured global data structure, and high utility patterns are discovered. Thus, the reconstructed global and local utility lists are core components, and the recursive mining process is vital operation.

### 3.8. Complexity analysis

In this subsection, we conduct time complexity analysis of the proposed method, LIHUP, with the previous ones, HUPID [38] and IHUP [37] which require only one scan to construct their global data structures, for the following two aspects: data structure construction or update, and pattern mining. Let \(N_o\) and \(N_i\) be the number of transactions in the original and increased databases composed of \(N_m\) items, respectively. Since all the compared methods scan once to construct or update their own data structures, if the length of the longest transaction is the same as the size of items in the databases, \(N_m\) items have to be processed to insert each transaction into the data structures. That is, time complexity of the process of inserting all transactions in the original or updated database is \(O(N_o \times N_m)\) or \(O(N_i \times N_m)\). Scanning a given database is the most time consuming process in pattern mining if the database size is as substantially huge as databases used in real world applications, and thus time complexity of the methods for this aspect are similar. In the mining process, the three methods, LIHUP, HUPID, and IHUP, find the same number of high utility patterns satisfying a given threshold recursively by following the pattern growth approach [13] where each conditional data structure is built for a candidate prefix pattern with no smaller utility upper-bound than the threshold. The difference between HUPID and IHUP is utility upper-bounds of candidates used in HUPID is much smaller than those of IHUP. It signifies that more local data structures are constructed by IHUP and its mining performance is worse than that of HUPID. That is, time complexity of mining high utility patterns for HUPID is better than that of IHUP. Meanwhile, these methods with one phase construction extract the huge number of candidates in their own mining processes, while LIHUP discovers all high utility patterns without candidate generation. Because the candidate generation requires an additional database scan for identifying actual high utility patterns from \(N_i\) candidates, its time complexity is \(O(N_0 + N_i) \times N_m \times N_o\) for the increased database. In contrast, there is no need for the proposed method to scan the database again. In summary, time complexity of the one phase construction
methods is much worse than that of the proposed one due to the additional database scan for the identification process with the huge number of candidate.

4. Performance evaluation

4.1. Experimental environment

In this section, we evaluate performance of methods for mining high utility patterns from incremental databases by storing utility information into global data structures through a single database scan and conducting mining operations with the data structure. That is, we compare our proposal (called LIHUP in the following subsections) with the methods with one phase construction, IHUP [37] and HUPID [38] that scan given data only once to construct or update their global data structures. All the algorithms were implemented in C/C++ and experiments were conducted on a Windows 7 OS with Intel 3.30 GHz CPU and 8.00GB main memory.

Moreover, to consider various characteristics of datasets, we used real and synthetic datasets as shown in Table 1 for incremental
high utility pattern mining. Firstly, for each 20% of a dataset, we constructed global data structures, restructured them, and extracted high utility patterns by conducting mining processes. Next, we updated utility information in the data structures by reflecting each rest 20% of the dataset, restructured them, and generated high utility patterns from the updated data structures with the same threshold setting. Note that, when the runtime of any algorithm was larger than 15,000 s, we terminated the algorithm since it was too late beyond compare.

In Table 1, |D|, |I|, and T_avg refer to the number of transactions in a dataset, that of items in the dataset, and the average length of the transactions. The following are descriptions of datasets used in our experiments for performance evaluation. Real datasets of the table, Chain-store, Retail, Connect, and Accidents were obtained from NU-MineBench 2.0 [47] and FIMI Repository (http://fimi.cs.helsinki.fi) and contain information about sales data of a retail market in California, market basket data of an anonymous Belgian retail store, connection data of online game users, and anonymous traffic accidents information. In addition, the first group of synthetic datasets from T104D200K to T104D100K has a characteristic of increasing database size and the last one from T10N10000L1000 to T40N40000L4000 has that of increasing the number of items. They were generated using the IBM synthetic data generator [6]. Meanwhile, the methods evaluated in this section’s experiments mine high utility patterns from non-binary incremental databases by considering relative item importance, and accordingly, internal and external utility information is necessary. Since all the datasets except for one of the real datasets, Chain-store, contain no utility information, we randomly generated internal and external utilities from 1 to 10 and from 0.01 to 1.00.

4.2. Performance evaluation of incremental high utility pattern mining on real datasets

In this part, we compare and analyze mining performance of the proposed list-based method, LIHUP, with the one phase construction ones, IHUP and HUPID, in terms of runtime and memory usage on the real datasets, Chain-store and Connect, in Table. Note that these methods reflect utility information in incremental databases to their data structures through only one scan and then discover high utility patterns.

Fig. 16 is the experimental results of LIHUP, HUPID, and IHUP in terms of runtime and memory usage with Chain-store, and the threshold is set to 0.0200%. In Fig. 16(a), runtimes of LIHUP and HUPID become slower with increase of database size, and that of IHUP is also proportional to the increase except when the size is raised from 20% to 40%. The reason for this is that they have to extract more high utility patterns satisfying the given condition from the more number of transactions. In the runtime result for Chain-store, the proposed method, LIHUP, shows the best performance for the increase in database size and HUPID with the improved overestimation model outperforms IHUP with the basic model. Moreover, we can observe that the gaps of runtimes between LIHUP and the previous methods become larger when the database size is increased. On the other hand, in the result of memory usage for Chain-store in Fig. 16(b), LIHUP requires the least amount of memory resources than the compared methods to discover the same set of high utility patterns and IHUP consumes more memory than HUPID due to its overestimation utilities. In addition, similar to the runtime in Fig. 16(a), we can see that the methods’ memory consumption is raised in accordance with the increase of the database size since they have to store more utility information to their data structures.

Fig. 17 shows experimental results for runtime and memory usage for LIHUP, HUPID, and IHUP using another sparse dataset, Retail. For this experiment, we set the minimum utility threshold to 0.1000%. Except for the IHUP results when the database size was 20%, the runtime performance of the comparison algorithms was proportional to the database size. The results show that the execution time of the IHUP has increased significantly because the applied and estimation techniques have generated too many candidate patterns in a small database, which necessitated a lot of computation in the actual high utility pattern identification process. On the other hand, in the memory performance result of this experiment, IHUP which performs many recursive operations and has the candidate pattern information stored in the memory during the process of generating a lot of candidate patterns and identifying the actual patterns from the results, showed the worst performance. On the contrary, the memory performance of the proposed method which performed the mining process without candidate generation was the best.

Next, Fig. 18 shows the results of the compared methods in terms of runtime and memory consumption for Connect and the
threshold is assigned as 80% in this experiment. In the result for a runtime aspect in Fig. 18(a), LIHUP shows outstanding performance compared to the other ones and IHUP is terminated when the database size is increased from 20% to 40% since it requires more than 15,000 s. That is, IHUP has the worst runtime performance and HUPID outperforms IHUP although it is slower than LIHUP. As we can observe from the runtime result for Chain-store in Fig. 16(a), the runtime gap between ours and the previous ones for Connect become worse with increase of the database size. As a result, we can derive that mining performance of HUPID and IHUP is substantially degraded, but LIHUP shows much better performance in incremental environments. Moreover, Fig. 17(b) shows that memory usage of all the methods becomes larger in accordance with increase of the database size because they have to store more information in their data structure. In the figure, LIHUP requires the least amount of memory resources and IHUP shows the worst memory performance.

Fig. 19 shows the experimental results for execution times and memory usage for LIHUP, HUPID, and IHUP using the last dense dataset, Accidents. In this experiment, we set the minimum utility threshold to 30%. As can be seen from the results of the experiments, execution time and memory usage performance deteriorate proportionally as the database size grows. In particular, the IHUP that generates the most candidate patterns due to the overestimation model has the worst performance in both aspects, while the proposed technique, LIHUP, has the best performance in all database sizes. The reason is that LIHUP builds and updates the data structure for the original and additional data with a single scan like the comparison schemes HUPID and IHUP, but does not require any additional scanning by not creating candidates in the mining process. This allows us to find high utility patterns more efficiently.

Table 2 shows the actual number of high utility patterns generated in the mining processes of the above experiments using the Chain-store and Retail data sets. From the table, we can see that all techniques, including the proposed technique LIHUP, yield the same mining results when given the same threshold. That is, the techniques for mining high utility patterns from a gradual database in a dynamic environment find a set of identical high utility patterns that satisfy a given threshold, resulting in
Hence, lesser LIHUP have figures, basic In threshold utility by threshold LIHUP that applied the results.

Table 2
Numbers of high utility patterns with chain-store and retail.

<table>
<thead>
<tr>
<th>Chain-store</th>
<th>Retail</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>#HUIs</td>
</tr>
<tr>
<td>Size</td>
<td>LIHUP</td>
</tr>
<tr>
<td>20%</td>
<td>2070</td>
</tr>
<tr>
<td>40%</td>
<td>1285</td>
</tr>
<tr>
<td>60%</td>
<td>1234</td>
</tr>
<tr>
<td>80%</td>
<td>1173</td>
</tr>
<tr>
<td>100%</td>
<td>1150</td>
</tr>
</tbody>
</table>

Fig. 19. Experimental results in terms of runtime and memory usage for accidents.

4.3. Scalability tests of incremental high utility pattern mining on synthetic datasets

In order to evaluate the scalability of the proposed method, LIHUP, and the previous ones, HUPID and IHUP, for increase of the database size and the number of items, we employ the groups of the synthetic datasets in Table 1, T10I4DxK (from T10I4D200K to T10I4D1000K) and TaNbLc (from T10N10000L1000 to T40N40000L4000). Moreover, for this evaluation, the scalability tests are conducted with two threshold settings, 0.05% forT10I4DxK and 0.02% for TaNbLc.

Fig. 20 shows the experimental results of the scalability tests of the proposed method with the previous ones, HUPID and IHUP, using the two dataset groups, T10I4DxK and TaNbLc. From the results, we can know that they require more runtime when the database size and the number of items become larger and LIHUP has the best scalability performance. Besides, mining performance of IHUP is the worst and substantially slower than the others. The reason for this is that it employs the basic overestimation concept and generates an enormous number of candidate patterns as a result.
Fig. 20. Experimental results for chain-store under varied minimum utility threshold.

Fig. 21. Experimental results for connect under varied minimum utility threshold.

Fig. 22. Experimental results for retail under varied minimum utility threshold.
5. Conclusions

In recent years, database volumes of various applications such as social network services and wireless sensor networks become larger gradually, and thus the minimum database scans are necessary to discover meaningful information from the incremental databases efficiently. This paper proposed an algorithm that constructs a global data structure through a single scan, restructures the data structure according to an optimal sorting order, and updates utility information in the restructuring step for mining high utility patterns efficiently. Moreover, we suggested a method to reflect newly added transactions to the previous data structure by scanning them, not the whole database, only once and optimize the updated data structure through a restructuring process. Comprehensive experiment results with real and synthetic datasets showed that the proposed method conducts mining operations more efficiently compared to the previous algorithms that require an additional database scan to identify actual high utility patterns from candidates although they construct and update their data structures through a single scan.

Acknowledgments

This research was supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (NRF No. 2015062051 and NRF No. 2015054624).

References


