Future Generation Computer Systems 95 (2019) 221–239

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Efficient approach for incremental high utility pattern mining with indexed list structure

Unil Yun [a], Hyoju Nam [a], Gangin Lee [a], Eunchul Yoon [b,*]

[a] Department of Computer Engineering, Sejong University, Seoul, Republic of Korea
[b] Department of Electronics Engineering, Konkuk University, Seoul, Republic of Korea

HIGHLIGHTS

- We suggest efficient incremental utility pattern mining.
- Novel indexed list structures are proposed to mine incremental utility pattern mining efficiently.
- Pruning techniques considering the list structures are presented.
- Significant performance improvements are shown with various experiments.

ARTICLE INFO

Article history:
Received 2 July 2018
Received in revised form 19 October 2018
Accepted 12 December 2018
Available online 10 January 2019

Keywords:
Data mining
High utility patterns
Incremental mining
Indexed lists
Utility mining

ABSTRACT

Since traditional frequent pattern mining approaches assume that all the items in binary databases have the same importance regardless of their own features, they have difficulty in satisfying requirements of real world applications such as finding patterns with high profits. High utility pattern mining was proposed to deal with such an issue, and various relevant works have been researched. There have been demands for efficient solutions to find interesting knowledge from specific environments in which data accumulates continuously with the passage of time such as social network service, wireless network sensor data, etc. Although several algorithms have been devised to mine high utility patterns from incremental databases, they still have performance limitations in the process of generating a large number of candidate patterns and identifying actually useful results from the found candidates. In order to solve the problems, we propose a new algorithm for mining high utility patterns from incremental databases. The newly proposed data structures in a list form and mining techniques allow our approach to extract high utility patterns without generating any candidates. In addition, we suggest restructuring and pruning techniques that can process incremental data more efficiently. Experimental results on various real and synthetic datasets demonstrate that the proposed algorithm outperforms state-of-the-art methods in terms of runtime, memory, and scalability.

© 2019 Elsevier B.V. All rights reserved.

1. Introduction

As one of various data mining techniques [1,2], pattern mining has played an important role in finding useful knowledge hidden from large amounts of data [3–10]. As a fundamental research topic of pattern mining, frequent pattern mining identifies all of the patterns that appear more frequently than a user-defined threshold from a given binary database [3,11]. A binary database is composed of a number of transactions, each of which has multiple items. Then, each item is denoted as 0 (absence) or 1 (presence). Traditional approaches assume that all the items in such a binary type of data have the same importance regardless of their own characteristics. Hence, they mine patterns simply considering item frequencies with various real-world features ignored. However, each item in real-world databases has its own importance as well as quantity information that may exceed 1. In this regard, frequent pattern mining does not satisfy the need to find patterns with high profits in real-world applications such as market analysis. High utility pattern mining has been researched as an important topic in the pattern mining area in order to deal with such problems [12–15]. High utility pattern mining ([16,17]; Yun, Ryang, & Ryu, 2016) considers non-binary features and the distinct importance of items to the mining process, which has made it possible to extract more useful patterns, called high utility patterns, satisfying a user-given threshold.

In recent years, large-scale data have been generated continually in various real-world applications such as web data, wireless network sensor data, and social network service. As a result, this trend has increased the importance of data analysis. Previous

* Corresponding author.
E-mail addresses: yunei@sejong.ac.kr (U. Yun), hjnam@sju.ac.kr (H. Nam), ganginlee@sju.ac.kr (G. Lee), ecyoon@konkuk.ac.kr (E. Yoon).

https://doi.org/10.1016/j.future.2018.12.029
0167-739X/© 2019 Elsevier B.V. All rights reserved.

techniques for static data analysis have trouble in processing such incremental data because they have to conduct their own mining processes from scratch in order to reflect additional data [5,6, 18–21]. To overcome this problem, various incremental pattern mining methods have been proposed [22–24]. Incremental can be applied in a wide variety of fields. In this field, the erasable itemset [25], Top-K itemset [26], uncertain itemset [27,28] and high utility itemset [29]. In this paper, we explain the incremental high utility pattern mining combining high utility and incremental. Incremental high utility pattern mining [14,23,30,31] is an approach proposed to mine high utility patterns from non-binary databases in which the data are continually accumulated. The corresponding algorithms satisfy the Anti-monotone property using their own overestimation techniques in order to minimize search space for mining interesting patterns without any pattern loss [32–35] (Anti-monotone property: if an item or pattern is found to be invalid, all of its supersets become meaningless). Although recent approaches can mine high utility patterns from incremental databases without working from scratch whenever new data are entered and without missing any valid pattern, they suffer from significant computational overhead problems in checking the validity of candidate patterns. Motivated by the problems, we propose an efficient algorithm for mining high utility patterns from incremental databases with indexed list structures without generating any candidates.

The main contributions of this paper are as follows: (1) proposing efficient list-based data structures and techniques for mining high utility patterns from incremental databases without any candidate generation; (2) devising a restructuring technique that can efficiently reflect incremental data to the data structures within a single database scan; (3) demonstrating that the proposed algorithm outperforms previous state-of-the-art approaches through performance evaluation tests on various real and synthetic datasets.

The remainder of this paper is as follows. Section 2 introduces previous interesting works related to the proposed method. Section 3 describes details of the proposed algorithm including the newly suggested data structures, techniques for pattern pruning, data structure restructuring, and recursive pattern mining, and empirical examples for them. Section 4 reports the experimental results of the proposed method and previous state-of-the-art ones including their in-depth analyses. Finally, Section 5 concludes this paper and introduces our future works.

2. Related work

In this paper, we introduce influential studies related to our approach. We first explain the basic concept of fundamental pattern mining methods focusing on binary databases, called frequent pattern mining. After that, we describe static approaches for high utility pattern mining on non-binary databases and dynamic methods on incremental non-binary databases.

2.1. Frequent pattern mining

Frequent pattern mining is a series of processes for finding all of the patterns that appear more frequently than a user-given threshold from a given binary database (called frequent patterns). Apriori [3] and FP-Growth [11] are well-known methods in this field. Apriori is a level-wise method using a candidate generate-and-test manner. Therefore, the algorithm extracts a large number of candidate patterns although meaningless patterns are pre-pruned from the Anti-monotone property [3]. Assuming that k is the length of the longest pattern among the ones generated from a given database, Apriori processes 1 to k level tasks, i.e., it may have to scan the database at least k times. FP-Growth was proposed to overcome the limitations of Apriori. The algorithm uses a tree data structure, FP-Tree, which is constructed in two database scans. Once the tree structure is built, FP-Growth mines frequent patterns based on a divide-and-conquer manner without any additional database scan and candidate generation. However, since they have been designed to deal with binary data types only, their usefulness continues to decline.

2.2. High utility pattern mining from static databases

High utility pattern mining was proposed to solve the limitations of traditional frequent pattern mining approaches. However, the support-based Anti-monotone property used in frequent pattern mining does not guarantee complete results of high utility patterns from non-binary databases. That is, the property causes pattern loss problems. In order to overcome this issue, Two-Phase [32], which is the first high utility pattern mining algorithm based on Apriori, employs an overestimated factor for satisfying the Anti-monotone property in high utility pattern mining, called Transaction Weighted Utilization (TWU). TWU-based Anti-monotone property is called Transaction Weighted Downward Closure (TWDC) [32], which means that, if a pattern has lower TWU than a given minimum utility threshold, any of its super patterns is invalid. After Two-Phase, various advanced approaches have been proposed, such as Apriori-based methods, FUM and DCG+ [36], and FP-Growth-based methods, UP-Growth [37] and UP-Growth+ [38]. Although they have different data structures and mining techniques from one another, their procedures are mainly divided into two parts: (1) Phase I — generates candidate patterns with TWU values higher than or equal to the threshold; (2) Phase II — identifies actual high utility patterns from the candidates through additional database scanning processes. In recent years, many studies on high utility mining have been published. EFIM-Closed [39] and FDHUP[40] are tree-based high utility pattern mining algorithm. Also, list-based approaches such as HUI-Miner [41], FHM (Fournier-Viger 2014), and IMHUP [42] have been devised to conduct high utility pattern mining operations without generating candidate patterns. However, their effectiveness is limited to static data.

2.3. Incremental high utility pattern mining

In recent years, more and more data have been being generated in various application fields. Hence, the features and volumes of data continue to change with the passage of time. Since static approaches have to conduct their own mining operations from scratch whenever new data are inputted, they suffer from fatal computational overheads. Therefore, in order to guarantee reasonable algorithm efficiency with respect to such incremental stream data, we need to process newly inputted data only without additional database scans and reflect them into the previously processed without any error. FUP-HU [30] is an Apriori-based incremental high utility pattern mining algorithm, which is the result of applying the concept of FUP, incremental frequent pattern mining method [43], into Two-Phase. This method classifies patterns within an original database into two types, Large and Small patterns, by comparing their TWU values with a given threshold. Large patterns have TWU values higher than or equal to the threshold while Small ones have lower values. After that, FUP-HU also divides patterns generated from additional databases (incremental data) into the two types and then it mines high utility patterns in a more efficient way by considering the changes in pattern types between the original and additional data. PRE-HUI [44,45] is a variation of FUP-HU, which uses two types of thresholds, Upper threshold and Lower threshold, and employs a new concept for predicting the changes of pattern states on incremental data, named Pre-large concept [44,45]. The Upper threshold plays the same role as the

Table 1
Example of a non-binary database and item profits.

	TID	Transaction	TU	Item	Profit
Original DB	T_1	(A, 2) (B, 1) (D, 1) (F, 1)	13	A	2
	T_2	(A, 1) (B, 2) (E, 2)	18	B	5
	T_3	(B, 1) (C, 2) (F, 2)	11	C	1
	T_4	(A, 2) (D, 1) (E, 2) (G, 1)	17	D	2
	T_5	(B, 2) (C, 1) (D, 1) (E, 2)	19	E	3
	T_6	(D, 2) (E, 1)	7	F	2
db_1^+	T_7	(C, 1) (E, 2) (F, 1)	9	G	5
	T_8	(A, 1) (B, 1) (C, 2) (D, 1)	11		
db_2^+	T_9	(B, 1) (D, 2) (E, 1)	12		
	T_{10}	(A, 3) (F, 1) (G, 2)	18		

minimum utility threshold and the Lower threshold is a smaller value than the Upper threshold. Pre-large patterns mean patterns of which the TWU values are between the thresholds. The PRE-HUI algorithm guarantees better performance than FUP-HU, but it is not an exact approach because pattern losses can be caused depending on how the Lower threshold is set.

An FP-Growth-based method, IHUP [14] constructs its own tree structure in a single database scan and extracts candidates for high utility patterns. After that, the algorithm identifies actual high utility patterns from the candidates through an additional database scan. Whenever new incremental data are inputted, the method processes them, updates the previously constructed tree structure, and conducts its own mining operations. HUPID [23] is an advanced tree-based approach, which employs its own data structures, HUPID-Tree and TList. Based on a divide-and-conquer manner, the method overcomes the drawbacks of Apriori-based algorithms. In addition, its special techniques for reducing over-estimated utility values have made a significant contribution to improving algorithm performance by generating a smaller number of candidate patterns. However, such a method still suffers from significant computational overheads because useless candidate patterns are still generated although the number may be further reduced compared to the other algorithms. On the other hand, the proposed method does not need to worry about this problem because it can mine a complete set of high utility patterns on incremental data without any candidate generation.

3. Indexed list based incremental high utility pattern mining without candidate generation

In this paper, we propose a new algorithm for mining high utility patterns from incremental databases without generating any candidate, called Indexed list based Incremental High Utility Pattern Mining (IIHUM). Our method includes new data structures and techniques for pattern pruning, data structure restructuring, and recursive pattern mining for incremental data processing. We also provide important preliminaries for high utility pattern mining and empirical examples for the proposed techniques.

3.1. Preliminaries

In utility pattern mining, a non-binary database with n transactions can be denoted as $D = \{T_1, T_2, \ldots, T_n\}$, which includes a set of m distinct items, $I = \{i_1, i_2, \ldots, i_m\}$. Each transaction T_d ($1 \le d \le n$) has a unique identifier, TID, and consists of multiple items belonging to I ($T_d \subseteq I$). Each item i_p ($1 \le p \le m$) has its own importance, called external utility and denoted $eu(i_p)$, and a non-binary quantity value for T_d, called internal utility and denoted as $iu(i_p, T_d)$. The utility of item i_p in T_d is calculated by multiplying its internal and external utility values, denoted as $u(i_p, T_d) = eu(i_p) \times iu(i_p, T_d)$. A pattern, P, which can be generated from D, is denoted as $P = \{i_1, i_2, \ldots, i_k\}$ ($P \subseteq I$ and $1 \le k \le m$). P can also be expressed as k-itemset, which means a set of items with length k.

Definition 1 (*Utility of a Pattern in a Transaction*)**.** Given a pattern, P, in a transaction, T_d, Utility of P in T_d is denoted and calculated as $u(P, T_d) = \sum u(i_p, T_d)$, where $i_p \in P$ and $P \subseteq T_d$.

Definition 2 (*Utility of a Pattern in a Database*)**.** Given a pattern, P, in a database, D, Utility of P in D is denoted and calculated as follows: $u(P) = \sum u(P, T_d)$, where $P \subseteq T_d$ and $T_d \in D$.

Definition 3 (*Transaction Utility of a Transaction*)**.** Given a transaction, T_d, its utility, $tu(T_d)$, is calculated as follows: $tu(T_d) = \sum u(i_p, T_d)$, where $i_p \in T_d$.

For example, in the original database of Fig. 1, 2-length pattern AB appears in the two transactions T_1 and T_2, where the utility of the pattern for each transaction is $u(AB, T_1) = u(A, T_1) + u(B, T_1) = 4 + 5 = 9$ and $u(AB, T_2) = u(A, T_2) + u(B, T_2) = 2 + 10 = 12$. Therefore, the utility of AB becomes $u(AB) = u(AB, T_1) + u(AB, T_2) = 9 + 12 = 21$. Transaction utility values of T_1 and T_2 are calculated as $tu(T_1) = u(A, T_1) + u(B, T_1) + u(D, T_1) + u(F, T_1) = 4+5+2+2 = 13$ and $tu(T_2) = u(A, T_2) + u(B, T_2) + u(E, T_2) = 2 + 10 + 6 = 18$.

Let $u(D) = \sum tu(T_d)$, where $T_d \in D$ be the total utility of D (the sum of all tu values) and δ be a user-given threshold percent value. Then, the corresponding minimum utility threshold is denoted as $minutil = u(D) \times \delta$. For a pattern, P, If u(P) is not smaller than minutil, P is called a high utility pattern; otherwise, a low utility pattern. Then, high utility pattern mining can be defined as a series of processes for finding all of the patterns of which the utility values satisfy minutil. In high utility pattern mining, the TWDC model [32] is typically used to maintain the Anti-monotone property (also called downward closure property) [3,11]. In the model, each pattern P has an overestimated utility value, TWU, which is denoted as $twu(P) = \sum tu(T_d)$, where $P \subseteq T_d$ and $T_d \in D$. If twu(P) is not smaller than or equal to minutil, P becomes a transaction-weighted high utility pattern (also called potential high utility pattern or candidate high utility pattern). For example, $u(D) = tu(T_1) + tu(T_2) + tu(T_3) + tu(T_4) + tu(T_5) + tu(T_6) = 13+18+11+17+19+7 = 85$ in the original database of Table 1. If a user-given threshold percent value, δ, is 0.3 (30%), $minutil = u(D) \times \delta = 85 \times 0.3 = 25.5$. In the case of pattern AB, although AB is not a high utility pattern because $u(AB) = 21 < minutil$, it becomes a transaction-weighted high utility pattern since $twu(AB) = tu(T_1) + tu(T_2) = 13+18 = 31 > minutil$. That is, AB is not considered a valid result, but it is not pruned and the subsequent works for finding its super patterns are conducted.

3.2. Overall process of mining high utility patterns without candidate generation from incremental databases

Fig. 1 shows the overall architecture of the proposed algorithm. It shows the data that flows in real time. The method first scans a given original database once to construct our global list structures. Thereafter, the constructed list structures are restructured according to a TWU ascending order and their index information is properly set again. If new transaction data are entered, the proposed method reflects them into the previously constructed data structures without re-scanning the previously processed data. In this process, TWU values of the items composing the global structures are calculated again, and the updated data structures are restructured according to the changed TWU ascending order. After the data structures are restructured, the user gives a threshold percent value to request a result of high utility patterns for the current incremental database. Then, the algorithm recursively generates conditional list data structures from the global data structures. After all of the recursive works are finished, the user can directly receive a complete set of high utility patterns without any candidate check.

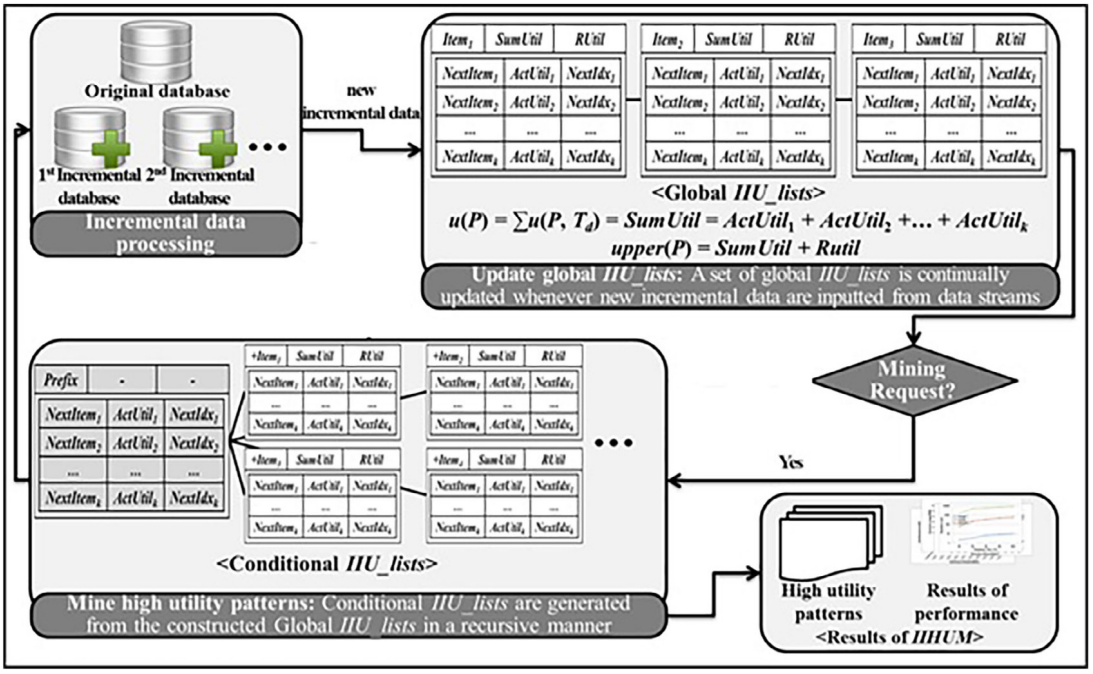


Fig. 1. Overall architecture of the proposed algorithm.

3.3. Constructing global indexed utility lists with original database

The previously proposed approach for incremental high utility pattern mining performs tree-based data processing works and overestimation-based pattern mining operations. In other words, since the approach generates a number of candidates through its overestimation method, it has to perform additional works for identifying whether each candidate is an actually meaningful result. We already know that such candidate checking works, called Phase II, are time-consuming tasks [34,35,46,47] (The literature says that the tree-based approaches have been found to consume large amounts of time and space resources in Phase II). Moreover, the number of generated candidate patterns can be increased exponentially according to the threshold settings, which means that the previous approaches are not suitable for dealing with large-scale incremental data. For this reason, we propose new efficient data structures for mining high utility patterns from incremental data without generating any candidate patterns.

The data in a given incremental database are refined and stored in the global data structures of our framework, called Incremental Indexed Utility List (IIU-List). A global IIU-List is assigned to each item within the current database, $\{i_p\}$, which is denoted as IIU-List$_{\{ip\}}$. Each IIU-List IU-List$_{\{ip\}}$ is composed of a number of entries for storing the information of the transactions with i_p. Fig. 2 presents the general form of a global IIU-List for an item. For each entry (an entry is related to one transaction with the current item), NextItem points to the item label next to the current item in the corresponding transaction; ActUtil is an actual utility value of the current item in the transaction (the product of the internal and external utility values); NextIdx indicates which entry in the IIU-List corresponding to the next item the current item appears in. SumUtil is the sum of all the ActUtil values in the current IIU-List, which is equal to the actual utility of the current item. RUtil indicates the maximum utility value that can be added to SumUtil when the current item is expanded to the maximum. The details of them are explained with proper definitions and examples in the subsequent section.

Item	*SumUtil*	*RUtil*
$NextItem_1$	$ActUtil_1$	$NextIdx_1$
$NextItem_2$	$ActUtil_2$	$NextIdx_2$
$NextItem_3$	$ActUtil_3$	$NextIdx_3$
...	...	...
$NextItem_k$	$ActUtil_k$	$NextIdx_k$

Fig. 2. General design of global IIU-List.

Definition 4 (*Remaining Utility of a Pattern*)**.** Let T_d be a transaction with pattern P and T_d/P be a set of the items appearing after P in T_d. Then, the remaining utility of P in T_d is denoted as ru(P, T_d) = $\sum$u(i_p, T_d), where $i_p \in T_d/P$. ru is employed to perform pattern pruning works through the Anti-monotone property based on an overestimation manner, which is different from the TWU method used in the previous tree-based approach. Its details are described in the subsequent section.

For a given database, D, IIHUM constructs a set of global IIU-Lists in the following manner. The method sorts a transaction of D, T_d = $\{i_1, i_2, \ldots, i_l\}$, in a canonical order such as a lexicographic order (there is no problem if the transaction is not sorted because our restructuring tasks are performed after the insertion process). Note that we assume the order of the items in the example database is the lexicographic order to give the readers more intuitive contents. Let $T_d' = \{i_1', i_2', \ldots, i_l'\}$ be the sorted transaction. Then, the proposed algorithm inserts each item to the set of global IIU-Lists starting from the last item, i_l'. That is, if an IIU-List for i_l', IIU-List$_{\{il'\}}$, does not exist in the global data structure, a new list for the item is created and a new entry for i_l' in T_d' is inserted into the list. Note that the initial state of the global data structure is empty. The reason why we process the items of the transaction in a reverse order is that NextItem and NextIdx can be set more easily without additional calculation works. In other words, since the first processed item, i_l', is the last item of the transaction, i.e., $T_d'/\{i_l'\} = \phi$, NextItem and NextIdx are set to NULL. ActUtil is set to u(i_l', T_d') and added to SumUtil of the list (SumUtil = SumUtil + u(i_l', T_d')).

After that, IIHUM generates the next item, i_{l-1}'. As in the case of the previous item, a new IIU-List for i_{l-1}', IIU-List$_{\{il-1'\}}$, is constructed if the list does not exist in the global data structure. Thereafter, a new entry for i_{l-1}' in T_d' is added to the list. Then, its ActUtil is updated with u(i_{l-1}', T_d') and SumUtil of the list increases by u(i_{l-1}', T_d'). Since i_{l-1}' is not the last item, its NextItem and NextIdx are set to the information of the previously processed item, i_l', instead of setting them to NULL. That is, NextItem of i_{l-1}' becomes the label (name) of i_l', and NextIdx of i_{l-1}' becomes the index number of the entry for i_l'. The remaining items from i_{l-2}' to i_1' are also processed in the same way. By doing so, we can track how the items are linked to each other and what utility value each item has. If every item in the transaction is processed, we increase the TWU values of the corresponding items, {twu(i_1'), twu(i_2'), …, twu(i_l')}, by tu(T_d'). After the insertion process is finished with respect to the current incremental database, we can obtain a complete set of global IIU-Lists (not yet restructured) and the TWU information of the items composing the current database. Note that the TWU information is not employed for calculating overestimated utility values of patterns like the previous tree-based approach. Instead, we use the information to restructure the global lists, so that the proposed algorithm can mine high utility patterns more efficiently (see the next section for more details). After the construction process, RUtil of each IIU-List is initialized to 0 and then computed during the restructuring process. The RUtil calculation process can be conducted simultaneously with the restructuring process with a few additional operations. Fig. 3 shows the set of global IIU-Lists constructed from the original database of Table 1.

Example 1. Consider constructing a set of global IIU-Lists from the original database in Table 1. Then, IIHUM inserts the data from the first transaction T_1 = {A, B, D, F} to the last one T_6 = {D, E}. Since the initial state of the global data structure is empty, the algorithm generates an IIU-List for F, IIU-List$_{\{F\}}$, and inserts a new entry into the list, which is the first entry in the list. Therefore, its index number is assigned to 1 and used to connect the entry of another IIU-List for the next item. Since F is the last item in T_1 ($T_1/\{F\} = \phi$), NextItem and NextIdx of the corresponding entry are set to NULL, respectively. ActUtil and SumUtil of the entry become u(F, T_1') = 2 and 0 + 2 = 2, respectively. In the case of the next item D, the algorithm creates a new IIU-List for D, IIU-List$_{\{D\}}$, and an entry is added to the list. The current item D is not the last item. Hence, its entry information does not have NULL; instead, its NextItem and NextIdx are set to F and 1, respectively. Its ActUtil and SumUtil are u(D, T_1) = 2 and 0 + 2 = 2, respectively. The remaining two items B and A are processed in the same way. Since tu(T_1') = 13, twu(A), twu(B), twu(D), and twu(F) are updated as 13. In the second transaction, T_2 = {A, B, E}, there is no IIU-List for the last item E. Therefore, IIU-List$_{\{E\}}$ is generated and its first entry for E is created, where its ActUtil and SumUtil become u(E, T_2) = 6 and 0 + 6 = 0, respectively. Since $T_2/\{E\} = \phi$, NextItem and NextIdx of the entry become NULL. Meanwhile, IIU-List for the next item B already exists. Hence, the second entry is created to the list instead of generating a new list, where its ActUtil, SumUtil, NextItem, and NextIdx are set to u(B, T_2) = 10, 5 + 10 = 15, E, and 1, respectively. In addition, the TWU values of A, B, and E are increased by tu(T_2) = 18.

Note that the sequence of the constructed IIU-Lists is not important because the entries of the lists have their own sequence information through the connections among them.

3.4. Restructuring global indexed utility lists

If the global IIU-Lists are completely constructed through a single scan of the current incremental database, TWU values for all the items are also calculated at the same time. Then, IIHUM sorts the set of the constructed global list structures in a TWU ascending order and updates the index information of the lists again. These works are called IIU-List restructuring processes. Note that the main purpose of the restructuring tasks is eventually to modify the link information of the entries within the lists. Hence, the sequence among the lists is not important. Instead, we only have to remember the order in which the lists are referenced. However, in this paper, we illustrate the re-arranged global IIU-Lists according to the TWU ascending order for the convenience of the readers. The proposed method also sets the RUtil values of the lists during the restructuring processes. Table 2 shows the TWU values of the items composing the original part of the example database.

Table 2
TWU values of the items in the original part of the example database.

ITEM	G	F	C	A	D	E	B
TWU	17	24	30	48	56	61	61

In order to restructure the global lists, the algorithm processes the global list structures sorted in the previous order, $i_1' < i_2' < \ldots < i_m'$, from the first list IIU-List$_{\{i1'\}}$ to the last one IIU-List$_{\{im'\}}$ one by one. First, for the selected list, IIU-List$_{\{ip'\}}$(1≤p≤m), our method processes its entries in sequence, where the other entries connected to the selected entry are found using its index information. Let $E_{\{ip',Td'\},1}$ be an entry at the start point and E_set = {$E_{\{ip',Td'\},1}$, $E_{\{ip',Td'\},2}$, …, $E_{\{ip',Td'\},o}$} be a set of o entries found from $E_{\{ip',Td'\},1}$ (including $E_{\{ip',Td'\},1}$). In the beginning, RUtil values of the global lists are initialized to 0. Then, the algorithm first re-arranges the global lists in the obtained TWU ascending order (recall that the rearrangement of the global lists is provided for the convenience of the readers. In the actual implementation, their actual locations do not need to be changed. All we have to do is modify the entry information within the global lists and set the visit sequence of the lists). After that, the method selects the first entry in the IIU-List corresponding to the first sequence of the previous order ($E_{\{ip',Td'\},1}$). Then, we can also determine E_set = {$E_{\{ip',Td'\},1}$, $E_{\{ip',Td'\},2}$, …, $E_{\{ip',Td'\},o}$}, which is sorted in the TWU ascending order. Let E′_set = {$E'_{\{ip',Td'\},1}$, $E'_{\{ip',Td'\},2}$, …, $E'_{\{ip',Td'\},o}$} be the state of E_set after sorting it in the TWU ascending order. Starting from the first entry of E′_set, entry modification works are performed. From E′_set, we can obtain the sum of the actual utility values for all of its entries, which is a temporary value to calculate RUtil values of the lists. The algorithm visits the location of $E_{\{ip',Td'\},i}$ (1≤i≤o) that $E'_{\{ip',Td'\},1}$ points to and updates its NextItem and NextIdx as the name and index number of $E_{\{ip',Td'\},2}$ (the next entry in E′_set). At the same time, the sum decreases by ActUtil of $E'_{\{ip',Td'\},1}$ and RUtil of the IIU-List with $E'_{\{ip',Td'\},1}$ increases by the decreased sum. The algorithm performs the same operations for the next entry of E′_set, $E'_{\{ip',Td'\},2}$. In the case of the last entry, $E'_{\{ip',Td'\},o}$, NextItem and NextIdx of the entry that $E'_{\{ip',Td'\},0}$ points to are set to NULL because there is no further entry connected after $E'_{\{ip',Td'\},o}$. The other entries in the global list structures are processed in the same way, and any entry once processed is not considered again. If the above operations are performed with respect to every entry of the global list structures, we can obtain a set of completely restructured global IIU-Lists.

Example 2. Fig. 4 shows the processes of restructuring the global IIU-Lists in Fig. 3. In Fig. 4(a), the entry data corresponding to the first transaction in the original database do not satisfy the current TWU ascending order. Therefore, in order to restructure the entries, we determine their index information and sort the entries in the TWU ascending order. After sorting them, A, B, D, and F are changed to F, A, D, and B. The sum of the ActUtil values in the entries is 13 (= 4 + 5 + 2 + 2). We visit the first entry of the list for F, where its NextItem and NextIdx are modified to A

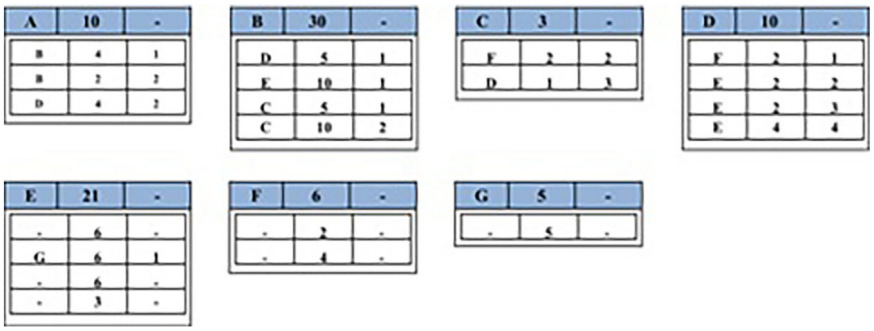


Fig. 3. Global IIU-Lists constructed from the original database of Table 1.

and 1, respectively. RUtil of the list increases by 11 (= 13 – 2). In the same manner, the first entries of IIU-Lists for A, D, and B are processed as shown in the figure. Since there is no further entry connected after the last entry in the list for B, NextItem and NextIdx of the entry become NULL, respectively. Fig. 4(b) presents how to conduct the restructuring works for the entry data corresponding to the second transaction. If all the entries are processed in this way, we can obtain a set of completely restructured global IIU-Lists as shown in Fig. (c).

The reason why we restructure the global IIU-Lists in the TWU ascending order is that the re-arranged data structures can allow the proposed algorithm to mine high utility patterns from incremental data in a more efficient way.

The previous tree-based incremental approach requires two times of data structure scans for its own tree restructuring processes. Meanwhile, the proposed data structure can be restructured in a single scan.

3.5. Updating global indexed utility lists with incremented data

If new transactions are inputted to the current incremental database, IIHUM reads the additional part, not the entire data, and reflects them to the previously constructed global IIU-Lists. Our algorithm first sorts each new transaction $T_{new} = \{i_1, i_2, \ldots, i_l\}$ in the current TWU ascending order. Let $T_{new}' = \{i_1', i_2', \ldots, i_l'\}$ be the sorted T_{new}. Then, the algorithm reflects the items of T_{new}' starting from the last item i_l'. In the process of the insertion, if IIU-List$_{\{il'\}}$does not exist in the set of the global data structures, it is newly constructed. A new entry for i_l', $E_{\{il',Tnew'\}}$, is inserted into IIU-List$_{\{il'\}}$, which becomes the last entry. Since $T_{new}'/\{i_l'\} = \phi$, NextItem and NextIdx of the entry become NULL, respectively. Its ActUtil value becomes u(i_l', T_{new}') and SumUtil of the list increases by u(i_l', T_{new}'). Since there is no item processed previously, the corresponding list maintains the same state of RUtil. The remaining items from i_{l-1}' to i_1' are processed in the same manner. Unlike the case of i_l', their NextItem and NextIdx information is set to proper values to connect their own next entries belonging to T_{new}'. In addition, the RUtil values of the lists including the items are increased by the sum of the ActUtil values for the items before the current item. That is, RUtil of the list for i_2' increases by the sum of the ActUtils for i_3' to i_l'. During the process, the TWU values of the items, {twu(i_1'), twu(i_2'), …, twu(i_l')}, are increased by tu(T_{new}'). If all the new transactions are reflected in the global list structures, IIHUM initializes the RUtil values of the lists as 0 (if the TWU ascending order is not broken after the insertion process, they are not initialized). The reason why they are set to 0 again is that reflecting new data can change the sequence of the items. Such changes can also influence the results of RUtil calculations. For example, the items after i_1 in T_{new}, $T_{new}/\{i_1\} = \{i_2, i_3, \ldots, i_l\}$, can be different from those after i_1' in T_{new}', $T_{new}'/\{i_1\}$. Accordingly, we re-calculate exact RUtil information of the lists during the restructuring process. RUtil of an IIU-List is an important value for calculating the minimum overestimated utility value of the item or pattern corresponding to the list. If RUtil is set too high, pruning effect becomes weak; meanwhile, if it is set to low, pattern losses can occur. That is, by calculating exact RUtil values, we can reduce the search space for mining high utility patterns without any pattern loss. After initializing the RUtil values, the proposed method calculates the TWU ascending order for the current data, restructures the global lists according to the order, and updates new RUtil values for the lists.

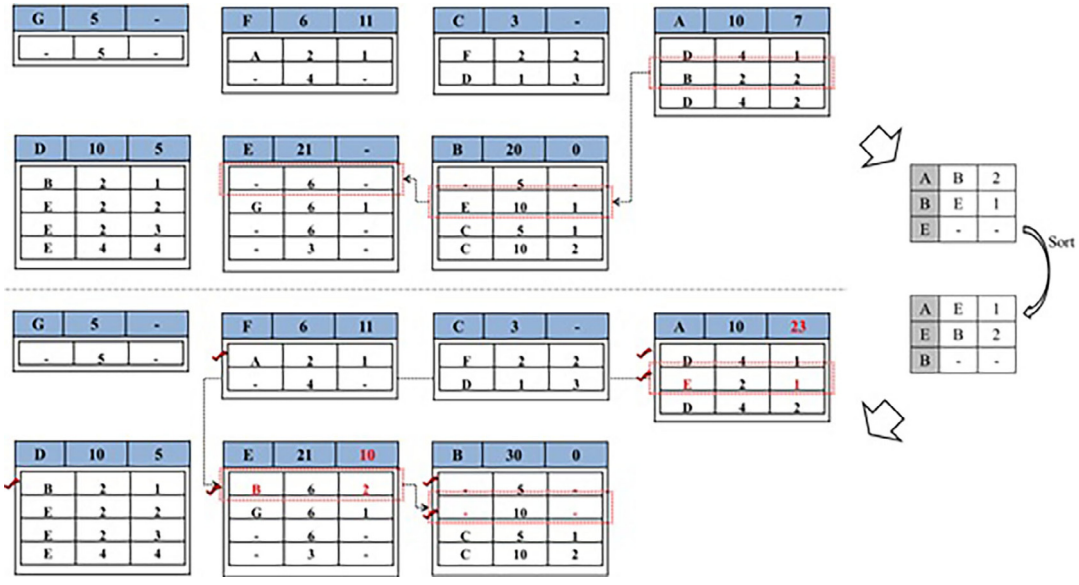
Example 3. Consider updating the global IIU-Lists previously restructured in Fig. 4 with the new data belonging to db$_1$+. The original sequence of the first transaction in db1+ (T_7) is C, E, and F, which is sorted in the previous TWU ascending order. Then, the result is F, C, and E. According to the changed sequence, the items are inserted as shown in Fig. 5(a). In this process, SumUtil and RUtil values of the IIU-Lists corresponding to the items are updated. Recall that the sorted items are inserted in the reverse order for efficient update tasks. That is, item E is first processed. SumUtil and RUtil of the IIU-List for E are 21 and 20, respectively. ActUtil and the remaining utility of E are 6 and 0 (because it is the last item). Therefore, SumUtil of the list is updated as 27 while its RUtil is unchanged. In the case of item C, those of the list for C are changed from 3 to 4 and from 23 to 29, respectively. NextItem and NextIdx of the newly inserted entry for C become E and 5 (we can fulfill the entry information without additional operations because the information can be obtained in the process of the previous item E). SumUtil and RUtil of the list for F are 6 and 18. Since the actual and remaining utility values of F in T_7 are 2 and 7, respectively, SumUtil and RUtil of the list become 8 and 25. Fig. 5(b) presents the state of the global IIU-Lists after processing the second transaction, T_7. Because there is no change in the TWU order before and after db$_1$+ is entered, the restructuring works are not conducted.

The reason why we focus on calculating exact RUtil values is that they are employed to compute upper bounds of IIU-Lists that decide whether the corresponding patterns are valid or not.

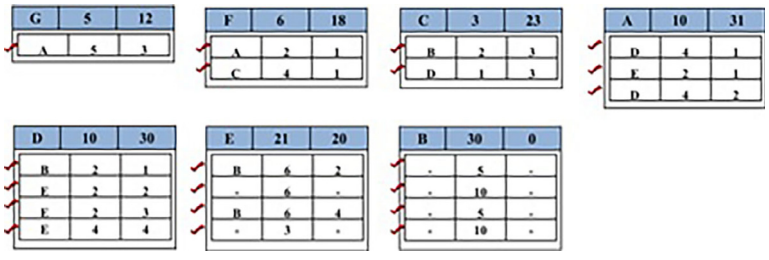
Definition 5 (*Upper Utility Bound of a Pattern*)**.** Given an IIU-List, l, its upper bound, upper(l) is denoted as upper(l) = SumUtil(l) + RUtil(l) (for pattern P corresponding to l, we can also denote the upper bound as upper(P) = SumUtil(P) + RUtil(P)). If upper(l) is smaller than the given threshold, l and every IIU-List generated from l become useless results. Then, we can prune them without any pattern loss.

The reason why we perform the restructuring works whenever new incremental data are entered is that we have to always reflect the exact remaining utility information to the IIU-Lists. If the IIU-List restructuring tasks are conducted without updating the remaining utility values, fatal pattern losses can occur.

(a) After restructuring the entries of the global IIU-Lists corresponding to the first transaction



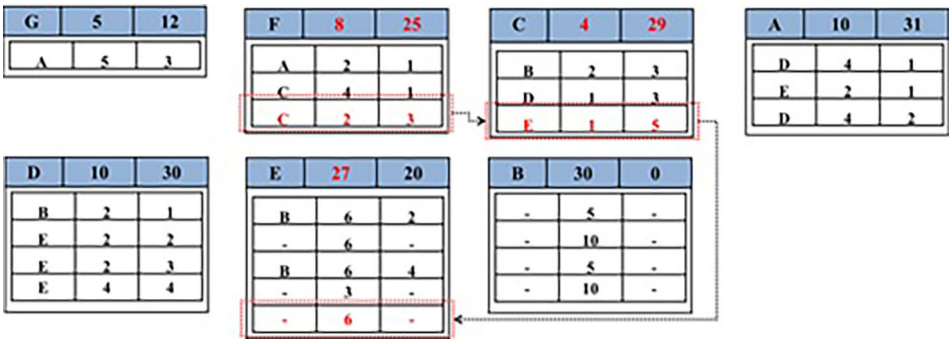
(b) After restructuring the entries of the global IIU-Lists corresponding to the second transaction



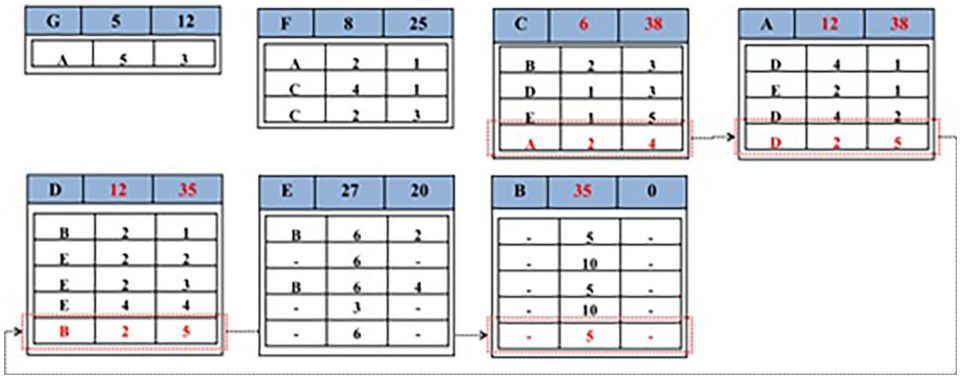
(c) After finishing the restructuring processes

Fig. 4. Result of restructuring the global IIU-Lists in Fig. 3.

Lemma 1. *Let $L = \{l_1, l_2, \ldots, l_k\}$ be a set of IIU-Lists and $L' = \{l'_1, l'_2, \ldots, l'_k\}$ be a set of IIU-Lists restructured without RUtil updates. Then, for each element of L' and all the possible conditional lists generated from L', it does not guarantee that the upper bound is always larger than or equal to minutil.*



(a) After processing the first transaction in $db_1+ (T_7)$



(b) After processing the second transaction in $db_1+ (T_8)$

Fig. 5. Result of inserting new transactions into the global tree structure in Fig. 4.

Proof. Recall that the upper bound of an IIU-List is the result of considering the actual utility values of the items appearing after the item corresponding to the list. In other words, for $l_x(1 \leq x \leq k)$, upper($_{lx}$) = SumUtil($_{lx}$) + RUtil(l_x), where SumUtil is a value that is not affected by the list restructuring works. Meanwhile, since RUtil is calculated from the utility values for the items that appear after the item corresponding to lx, it may vary depending on the order of the items. L has been sorted in a certain order such as a lexicographic order or a previous TWU ascending order. Hence, upper(l_x) has reflected the utility values of all the items after lx. Let $S_{lx} = \{s_{lx1}, s_{lx2}, \ldots, s_{lxi}\}$ be a set of IIU-Lists that can be expanded from l_x. Then, for each s_{lxh} $(1 \leq h \leq i)$, we can guarantee that u(s_{lxh}) $\leq$ upper(lx). On the other hand, each l′ in L′ has been updated with the index information based on the current TWU ascending order, while its RUtil is a value calculated on the basis of the previous order. Let $S_{l'x} = \{s_{l'x1}, s_{l'x2}, \ldots, s_{l'xi}\}$ be a set of IIU-Lists that can be expanded from l′$_x$. Then, there can be any $s_{l'x}$ such that u($s_{l'x}$) > upper(l′$_x$). For this reason, pruning l′$_x$ without updating RUtils can cause fatal pattern loss problems. ■

3.6. Mining high utility patterns from global indexed utility lists

If a mining request occurs after the current global IIU-Lists are restructured, IIHUM finds all of the high utility patterns satisfying a given threshold from the data structures. As data progressively increases, low utility patterns can become high utility patterns, and vice versa. Moreover, the minimum utility threshold can be changed by the requests of the users. For this reason, in order to mine high utility patterns without any pattern loss in such a dynamic environment, we need to maintain all the items composing the current incremental database by storing them in our global data structures regardless of their TWU values. Meanwhile, the traditional approaches designed to process static data calculate TWU values of items and their order in the first database scan and remove items with lower TWUs than the given threshold in the process of constructing their own data structures in the second database scan. Therefore, they are not suitable for incremental pattern mining because they have to perform the mining operations from scratch in order to extract a complete set of high utility patterns from incremental data.

Assuming that the TWU ascending order of the current incremental database is $i_1 < i_2 < \ldots < i_m$, the proposed method conducts the following processes starting from IIU-List$_{\{i1\}}$. Recall that, in the actual implementation, we do not need to re-arrange the actual locations of the global lists because their entries already have the index values that can track the items according to the TWU ascending order. All we have to do is access the global lists in the sequence of the order.

IIHUM first compares upper(i_1) to minutil and then proceeds with its pattern expansion processes if upper(i_1) $\geq$ minutil. Otherwise, i_1 is omitted in the current mining process. If the item is not pruned, SumUtil(i_1) is compared to minutil again. If SumUtil(i_1) $\geq$ minutil, i_1 is extracted as a high utility pattern.

Lemma 2. *Let $P = \{i_1, i_2, \ldots, i_k\}$ be a pattern that can be obtained from a given database, DB, and $P' = \{i_1, i_2, \ldots, i_k, i_{k+1}, \ldots, i_{k+x}\}$ be a superset of P ($P' \in DB$; $1 \leq x$; $k + x \leq$ length of the longest transaction in DB). Then, upper(P) is always higher than or equal to u(P′).*

Proof. Let L_P be an IIU-List for P. Then, SumUtil(L_P) indicates the result of the utility value calculated from the transactions where the items of P, i_1, i_2, …, i_k, appear at the same time. That is,

SumUtil(L_P) = u(P). P′ that can be extended from P has a length between k + 1 and k + x. Hence, its actual utility, u(P′), is influenced by the characteristics of the additional items, i_{k+1}, …, i_{k+x}, such as their frequencies, quantity values, and weights. Since the actual utility factor does not satisfy the Anti-monotone property, u(P′) may be larger or smaller than u(P). Meanwhile, upper(P) = SumUtil(P) + RUtil(P) = u(P) + RUtil(P), where RUtil (P) is the result of adding the actual utility values of every item that can appear after P in the pattern extension process from P. That is, upper(P) is the maximum value of the actual utility that P′ can have. For this reason, if upper(P) < minutil, u(P′) < minutil because u(P) ≤ upper(P). Consequently, by pruning the IIU-List for P in this case, we can omit a number of operations that cause unnecessary results, IIU-Lists for any super patterns of P. ■

For this reason, the proposed algorithm first computes the upper bound value before determining whether to generate further IIU-Lists from i_1. If upper(i_1) < minutil, we do not need to assign the search space for finding any superset that can be generated from i_1. In other words, no conditional IIU-List for the item needs to be constructed. On the other hand, if upper(i_1) ≥ minutil, the algorithm proceeds with the next works for generating conditional IIU-Lists for 2-itemsets of which the prefix is i_1. Starting from each entry of IU-List$_{\{i1\}}$, IIHUM searches for the entries connected by the index information.

In other words, let l(i_k) be an IIU-List for item $i_k \in$ DB and $ES_{l(ik)}$ = {$e_{ik,1}$, $e_{ik,2}$, …, $e_{ik,n}$} be a set of the entries in l(i_k). Then, the algorithm tracks each element of $ES_{l(ik)}$, $e_{ik,\ m}$ ($1 \le m \le n$), in the following manner. In the beginning, IIHUM generates a temporary IIU-List for prefix i_k (denoted as $l_{base(ik)}$) in order to maintain a set of IIU-Lists for 2-itemsets with the prefix as shown in Fig. 6 (denoted as $LS_{(ik)} = \{l_{p1}, l_{p2}, \ldots, l_{pq}\}$. After that, for each e_{ik}, m, the algorithm visits the entries connected to it one by one. Let $ES_{eik,m} = \{e_{ik,m,1}, e_{ik,m,2}, \ldots, e_{ik,m,o}\}$ be a set of the entries connected to $e_{ik,m}$. Then, for each of the entries, $e_{ik,m,r}$ ($1 \le r \le o$), the method finds whether or not there is the IIU-List for the 2-itemset with the item of the current entry. If so, a new entry is created as the currently last entry into the list. Otherwise, the algorithm generates a new IIU-List for the 2-itemset and adds a new entry to the list as the first entry. Then, the item name and index information of the new entry is updated to $l_{base(ik)}$. ActUtil of the new entry becomes the sum of ActUtil($e_{ik,m}$) and ActUtil($e_{ik,m,r}$), where the sum is also added to SumUtil of the new IIU-List for the 2-itemset. After that, the next $e_{ik,m,r}$ is processed. In the case of the last $e_{ik,m,r}$, the new entry of the IIU-List for the 2-itemset generated by $e_{ik,m,r}$ sets its NextItem and NextIdx information to NULL, respectively, because there are no more items to process. Such works are repeated with respect to every $e_{ik,\ m}$. Then, we can obtain the complete result of $LS_{(ik)}$.

Example 4. Consider mining 2-length high utility patterns from the global IIU-Lists shown in Fig. 5(b) when minutil is 21 (= 105 × 0.2). In the figure, the upper bound of G, upper(G), is 17 (= 5 + 12). Therefore, all of the subsequent operations for G are omitted. Since upper(F) = 33 > minutil, the algorithm conducts the pattern expansion processes to construct conditional IIU-Lists for the 2-itemsets generated when F is set to the prefix. Fig. 6 shows the results of the conditional lists. The list at the left side of the figure is a base-list for prefix F, denoted as _F. the others at the right side are the IIU-Lists for the 2-itemsets, which are denoted as +*i* (i: the newly added item). For example, +*C* means pattern FC, which is the result of adding C to the prefix F. Tracking the indexes of the IIU-List for F in Fig. 5(b) in sequence, the algorithm constructs the conditional lists. The index information obtained from the first entry of the list is as follows: F{A;2;1} → A{D;4;1} → D{B;2;1} →2 B{-;5;-}. After reading F{A;2;1}, the method inserts a new entry into the list for _F. A and 2 are inputted to NextItem and ActUtil of this entry, respectively; meanwhile, its NextIdx is set to the value of +NextItem, i.e., the entry number of the list for +*A*. In this case, NextIdx becomes 1. After reading A{D;4;1}, the entry information of the list for +*A* is updated. NextItem and NextIdx are set in the same manner. Meanwhile, in this entry, we insert 6, which is the sum of the utility in A and that of the prefix. At the same time, SumUtil of the list for +*A* is updated. The actual utility information of A is temporarily maintained to compute RUtil values of the lists for the 2-itemsets. If B{-;5;-} is processed, the RUtil information of the lists is updated. The result of accumulating the utility values of the entries is 11 (= 4 + 2 + 5), where 7 (the value excluding the utility of A) is updated as the RUtil value of +*A*. If the above works are repeated with respect to every entry for F in the global IIU-Lists, we can obtain a complete set of the conditional lists as shown in Fig. 6. Since the upper bounds of all the lists do not satisfy minutil, pattern growth operations for F are finished.

There is an additional consideration when we construct conditional IIU-Lists for patterns with k lengths (k > 2). In the case of constructing a conditional IIU-List for a 2-itemset, ActUtil of an entry in this list is calculated by adding ActUtil of the entry in the list for the current prefix and that in the list for the newly added item. Meanwhile, in the case of creating a conditional IIU-List for k-itemsets, if ActUtil of its entry is computed in the same manner, the result becomes larger than the actual value because the utility value corresponding to the prefix is redundantly added to the result. In other words, the list for a k-itemset is generated from two IIU-Lists for k – 1-itemsets, where the utility of the prefix is already reflected in each of the two lists when we create them. Therefore, if ActUtil values of the entries in these two lists are directly added to ActUtil of the entry in the list for the k-itemset, the utility of the prefix is added twice. To solve the problem, we subtract this value from the sum of the ActUtil values. By doing so, we can determine exact results of ActUtil values in the lists for k-itemsets without any additional operations.

Example 5. Fig. 7 shows how to construct conditional IIU-Lists for longer patterns from the lists for the 2-itemsets in Fig. 6. Note that minutil is set to 10.5 (= 105 × 0.1) to show the expansion processes. Since the upper bound of +*C* (= FC) is larger than minutil, pattern growth works are performed based on the list. First, the information of +*C* is inserted to _FC as in the construction case of the lists for 2-itemsets. After that, tracking the index information of +*C*, the algorithm generates IIU-Lists for the items that can participate in expanding _FC. As mentioned earlier, we subtract the overlapped utility value of the prefix in the process of calculating ActUtil of each entry in the constructed lists. That is, when the list for +*B* is constructed from _FC, ActUtil of the first entry is calculated as follows: 6 + 9 – 4 = 11. In the IIU-Lists for the 2-itemsets, +*D* and +*E* are not selected as the prefix for the next mining operations because their upper bounds do not satisfy the given threshold. However, it does not mean that the entries within them are excluded in the other mining operations.

By recursively generating conditional IIU-Lists from the constructed global lists, we can obtain a complete set of high utility patterns from a given incremental database without extracting any candidate.

If new incremental data are entered, the proposed algorithm reflects them to the previously constructed global IIU-Lists, restructures the updated data structures, performs its recursive pattern expansion works, and provides the results to the user.

3.7. Algorithm for mining high utility patterns from incremental databases

In this paper, we have proposed a method that can more efficiently mine high utility patterns from incremental databases

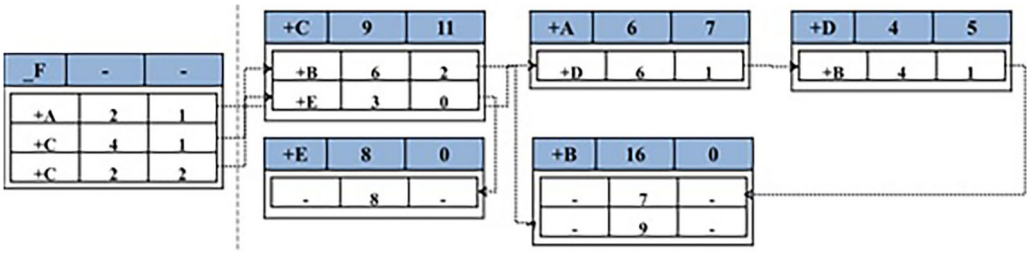


Fig. 6. Conditional IIU-Lists for the 2-itemsets expanded from F.

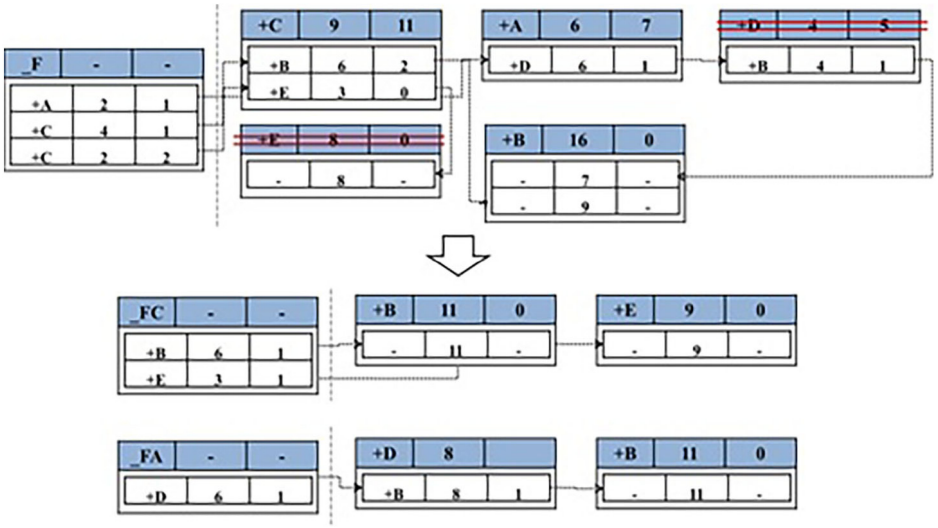


Fig. 7. Conditional IIU-Lists for the 3-itemsets constructed from the IIU-Lists for the 2-itemsets in Fig. 6.

through the newly proposed data structures and mining techniques. In this section, we explain how the proposed data structures and techniques are performed in our algorithm through the following overall procedure shown in Fig. 8.

The proposed algorithm deals with a number of databases because it processes incremental data as shown in the input parameters. After initializing a set of global IIU-Lists, the algorithm process each database (lines 01–21). Since there is no TWU ascending order previously calculated before processing the original (first) database, a canonical order such as a lexicographic order is applied to sort transactions of the database (line 04). Once the database is processed, we can obtain the corresponding TWU ascending order, which is used when processing the next incremental database. For each transaction, the algorithm calculates and updates TWU values of the items (line 06). After that, new IIU-Lists are created if necessary, and new entries are generated and updated (lines 07–12). Then, the method computes a new TWU ascending order for the currently constructed L and conducts its restructuring works (lines 14–15). Thereafter, if the user requests to mine patterns, IIHUM recursively constructs conditional IIU-Lists to mine valid high utility patterns (lines 17–20).

If the restructuring function is called, the algorithm initializes RUtils of all the elements in L in order to re-calculate their exact values after modifying the index information of the lists (line 01). For each element in L, the algorithm conducts the restructuring works as follows. For each entry of the current IIU-List, the method searches for all the entries connected to the current entry, which are stored to a temporary entry set (line 04). Such operations are easily performed through the NextItem and NextIdx information. The collected entries are sorted in the newly obtained TWU ascending order. Then, the algorithm modifies the NextItem and NextIdx information of the corresponding entries based on the newly obtained order (lines 06–09). In this process, entries already processed are ignored to prevent duplication of operations (lines 07–08). If we finish the restructuring works for every element connected to a certain entry, the RUtil values of the IIU-Lists corresponding to them are updated (line 11). After the above operations are performed with respect to all the entries in L, we can obtain the completely restructured L.

If a mining request occurs, function Mine is called. After a set of conditional IIU-Lists are initialized, the algorithm performs the following operations with respect to the received parameters. For each element in L, the method first checks whether its upper bound satisfy minutil or not (line 03). If this condition is not satisfied, all of the operations related to the corresponding IIU-List are pruned; otherwise, the method checks again whether its SumUtil (actual utility) satisfies minutil (line 04). Items or patterns satisfying this condition become actual high utility patterns (line 05). However, just because the current item or pattern satisfies the second condition does not mean that it can be pruned permanently. The reason is that valid high utility patterns may be generated from such an item or pattern. After the prefix update work (line 06), the algorithm generates a base-list for the prefix (line 07). Thereafter, for each entry in the currently selected IIU-List, the method updates its entry information to the base-list (line 09). Then, for each entry connected to the current one, the method 1) generates a new conditional IIU-List for the entry if necessary (lines 11–12), 2) creates a new entry into the list, and 3) inserts necessary information (lines 14–17). If the length of the currently expanded pattern is 3 or longer, the duplicated utility value is excluded in the process of calculating ActUtil of the new entry (line 15). If the above works are finished with respect to the selected entry, the algorithm updates RUtil values of the corresponding IIU-Lists. When all the entries of the currently selected IIU-List in L

Table 3
Features of the real datasets.

Dataset	Num. of Trans.	Num. of items	Avg. Trans. Size	Data size
Accidents	340,183	468	33.8	33.8 MB
Chain-store	1,112,949	46,086	43.0	8.82 MB
Connect	67,557	129	8.1	31.40 MB
Retail	88,162	16,470	10.306	3.97 MB
Mushroom	8,124	120	23	0.545 MB
Kosarak	990,002	41,270	8.1	31.4 MB

Table 4
Characteristics of synthetic datasets.

Dataset	Num. of Trans.	Num. of items	Avg. Trans. Size	Data size
T10I4DxK	100,000 – 1,000,000	1,000 (fixed)	10 (fixed)	3.83 MB – 38.30 MB
$Tx_1Lx_2Nx_3$	100,000 (fixed)	10,000 – 40,000	10 – 40	4.81 MB – 21.90 MB

are processed, we can obtain a complete set of conditional IIU-Lists that can be generated from the list. If the number of elements in the set is 2 or more, the algorithm recursively calls the Mine function in order to find high utility patterns with longer lengths (lines 21–22). If there is only one element, the method compares its actual utility value to minutil and determines whether to extract the corresponding pattern or not (lines 23–25). If all the recursive works are terminated, we can obtain a complete set of high utility patterns from the given incremental database.

4. Performance evaluation

4.1. Experimental environment and datasets

In this section, we conduct various experiments to compare the proposed algorithm, IIHUM, with the state-of-the-art methods, FUP-HU [30], HUPID [23], and LIHUP [42]. All the algorithms including IIHUM have been implemented in the C/C++ language for our performance evaluation tests, which were performed on 4.0 GHz CPU, 32 GB RAM, and Windows 10 OS. We use two types of benchmark datasets, real datasets for runtime and memory performance evaluation and synthetic datasets for scalability tests.

Table 4 presents the features of the real datasets used to evaluate runtime and memory usage performance of the algorithms. Accidents, Connect, Retail, Mushroom, and Kosarak are available at the FIMI Repository (http://fimi.cs.Helsinki.fi); Chain-store is available at NU-MineBench 2.0 [48]. Chain-store, Retail, and Kosarak are sparse datasets composed of short-length transactions and include a large number of distinct items. On the other hand, Accidents, Connect, and Mushroom are dense datasets, which have relatively long-length transactions and a small number of items. In the table for the characteristics of the real datasets, Num. of Trans., Avg. Trans. Size, and Num. of Items mean the number of transactions, the average length of the transactions, and the number of distinct items in each dataset, respectively. These real datasets are famous benchmark datasets widely employed in the pattern mining area.

We use two groups of synthetic datasets to evaluate the scalability of the algorithms. The datasets are available at the IBM data generator [3]. Table 4 shows the characteristics of the synthetic datasets. Since the datasets in Tables 3 and 4 except for Chain-store do not have their own internal and external utility information, we have set the external utility values according to the log normal distribution between 0.01 and 10 and internal utility values randomly from 1 to 10 as in the previous literature [12,13,23,30].

In order to conduct the performance evaluation for the high utility pattern mining algorithms on incremental data processing environments, we divide each of the employed datasets into 5 parts, where the first part becomes an original database and the others become incremental databases. In the beginning, each algorithm reads the original database, constructs its own data structure, performs the restructuring process, and mines high utility patterns. In the same manner, the algorithm processes each of the incremental databases. When all the tasks are finished, the performance results of the methods are evaluated.

4.2. Performance comparison on real dense datasets

We first perform experiments for performance comparisons among FUP-HU, HUPID, LIHUP, and our IIHUM using the real dense datasets, Accidents, Connect, and Mushroom. These experiments consist of runtime and memory tests on the settings of increasing data sizes and changing threshold.

4.2.1. Runtime and memory tests on increasing data size

Figs. 9, 10, and 11 present the runtime results of the Accidents, Connect, and Mushroom datasets on increasing data sizes, where the threshold settings are 30% , 85%, and 20%, respectively. As shown in the figures, the algorithms require more runtime resources as the size of a given dataset becomes larger and more data are accumulated. The results of the Mushroom dataset in Fig. 11 show that all algorithms tend to decrease as the data size increases in some sections. This seems to be a feature of the Mushroom dataset. However, the proposed method guarantees the fastest runtime speed in every case. As an Apriori-based algorithm, FUP-HU shows the most inefficient runtime performance among the compared methods. Although HUPID is faster than FUP-HU, there is still a significant performance gap between it and ours. The reason why the competitors have worse performance than the proposed algorithm is that they have to generate candidate patterns in the process of extracting high utility patterns. They are not able to mine valid patterns directly because of their own pattern mining mechanism. Instead, they first extract candidate patterns based on the overestimation methods and then conduct verification works to find valid patterns from the candidates. A series of the works for extracting candidate patterns is called Phase I, and that for mining high utility patterns after Phase I is called Phase II. The time spent in Phase 2 is known to be longer than the time consumed in Phase 1 in many cases [23,30,37,38]. Moreover, the larger the size of a given database is and the lower the threshold is, the more runtime resources such algorithms require. The list-based LIHUP is much faster than FUP-HU and HUPID, but slower than our IIHUM. LIHUP and IIHUM can mine the high utility pattern without generating candidate patterns in the incremental database. However, LIHUP is not as good as our indexed list because it only uses a list structure that is less efficient than our indexed list. For this reason, our method can provide more efficient runtime performance compared to the competitors. In the Connect datasets in Fig. 10, when the data size is 20% or later, the results of FUP-HU cannot be expressed partially. Because the algorithm requires a runtime that is too long for the Connect dataset, such as 20,000 s for a data size of 20%. The proposed algorithm also guarantees the best performance in this dataset as shown in the figure. As the data size becomes larger, HUPID and LIHUP show a larger runtime increase than that of IIHUM. Meanwhile, our method presents more stable runtime performance. For example, the runtime increasing degrees of HUPID, LIHUP and IIHUM are approximately 4.9, 2.9 and 2.6 times, respectively, when the database size is increased from 20% to 100%.

Figs. 12–14 show the results of memory usage for the Accidents, Connect, and Mushroom datasets, where some of the memory results of FUP-HU in Connect cannot be expressed partially from the figure because of the aforementioned reason. Since FUP-HU is

Input: a set of databases, $DB = [db_1, db_2, \ldots, db_k, db_{k+1}, \ldots)$; a minimum utility threshold δ;
// db_1: an original database; the others; incremental databases
Output: a set of high utility patterns, R

Main Procedure
01. Initialize a set of global *IIU-Lists*, L;
02. For each database, db_x in DB, do
03. For each transaction, T_k, in dbx, do
04. Sort T_k in the current order; // canonical order in the initial state or previous *TWU* ascending order in the other cases
05. For each item, i_i, in T_k, do // processed from the last item of T_k
06. Update *TWU* of i_i;
07. If *IIU-List* for i_i, l_{il}, exists in L, then
08. Create a new entry and insert *NextItem*, *ActUtil*, and *NextIdx* for i_i;
09. Update *SumUtil* of l_{il};
10. Else
11. Generate a new *IIU-List* for i_i, l_{il};
12. Create a new entry and insert *NextItem*, *ActUtil*, and *NextIdx* for i_i;
13. End for
14. Calculate new *TWU* ascending order for L;
15. Call *Restructure*(L);
16. End for
17. If there is a mining request from User, then
18. Initialize R;
19. Calculate *minutil* from sum of *TU*s and δ;
20. Call *Mine*(R, L);
21. End for

Procedure *Restructure*(R, L)
01. Set *RUtil* of every element of L to 0; //initializing *RUtil*
02. For each *IIU-List*, l_k, in L, do
03. For each entry, e_x, in l_k, do
04. A temporary set of entries, $T \leftarrow$ all the entries connected to e_x;
05. Sort T in the *TWU* ascending order of L;
06. For each entry, t_y, in T, do //processed from the last entry of T
07. If t_y is an entry processed already, then
08. Go to line 5 and process the next entry of T;
09. Update *NextItem* and *NextIdx* of the actual entry t_y points to with those of the previous entry in T; //null at the first time
10. End for
11. Update *RUtil* values of the *IIU-List*s related to T;
12. End for
13. End for

Procedure *Mine*(R, L)
01. Initialize a set of conditional *IIU-List*s, L';
02. For each *IIU-List*, l_k, in L, do
03. If l_k.*SumUtil* + l_k.*RUtil* ≥ *minutil*, then
04. If l_k.*SumUtil* ≥ *minutil*, then
05. $R \leftarrow R \cup l_k$.*pattern*; //actual high utility pattern;
06. Add l_k.*pattern* to the current prefix, *Pref*;
07. Insert a new *IIU-List* for *Pref*, l'_{Pref}, into L';
08. For each entry, e_x, in l_k, do
09. Insert e_x.*NextItem* and e_x.*ActUtil* into a new entry of l'_{Pref};
10. For each entry linked to e_x, e'_x, do
11. If *IIU-List* for e'_x does not exist in L', then
12. Create a new *IIU-List* for e'_x, $l'_{e'x}$;
13. Insert a new entry, e'', into $l'_{e'x}$;
14. $l'_{e'x}$.*NextItem* $\leftarrow e'_x$.*nextItem*;
15. $l'_{e'x}$.*ActUtil* $\leftarrow e'_x$.*ActUtil* + l'_{Pref}.*ActUtil*; //if the pattern length is longer than 2, the overlapped utility value is subtracted
16. Add $l'_{e'x}$.*ActUtil* to $l'_{e'x}$.*SumUtil*;
17. Set *NextIdx* of the previous entry to the current entry index of $l'_{e'x}$;
18. End for
19. Update *RUtil* values of the conditional *IIU-List*s related to each e'_x;
20. End for
21. If L' has 2 or more elements, then
22. Call *Mine*(R, L') //recursive call
23. Else if L' has a single element, then
24. If l'.*SumUtil* ≥ *minutil*, then
25. $R \leftarrow R \cup l'$.*pattern*; //actual high utility pattern;
26. End for

Fig. 8. Overall procedure of the proposed algorithm.

an Apriori-based algorithm, it seems to have good memory performance at relatively high minutil settings as shown in Fig. 12. However, its memory efficiency is sharply decreased when the number of the generated candidate patterns is explosively increased as the threshold becomes lower. HUPID shows the worst memory efficiency since the method has to maintain complicated tree structures during the mining process and spend large memory space in verifying a large number of candidates although the number is smaller than that of FUP-HU. On the other hand, the proposed algorithm and LIHUP does not require any additional memory for such candidate checks because it does not extract any candidate pattern. Moreover, since our method employs the indexed list data structures and LIHUP employs the list data structures simpler than the previous tree structures, it can achieve more efficient memory performance than that of HUPID. As shown in Fig. 13, the memory efficiency gap between HUPID and IIHUM is not large

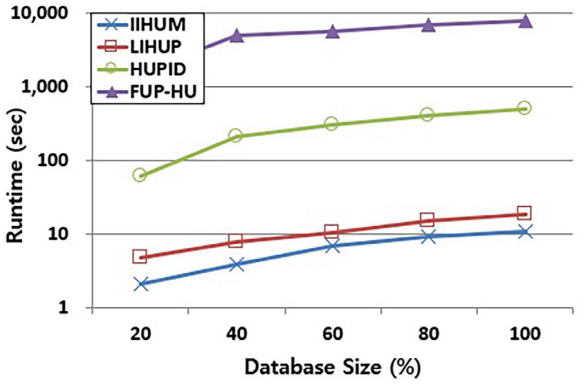


Fig. 9. Runtime test on Accidents (minutil = 30%).

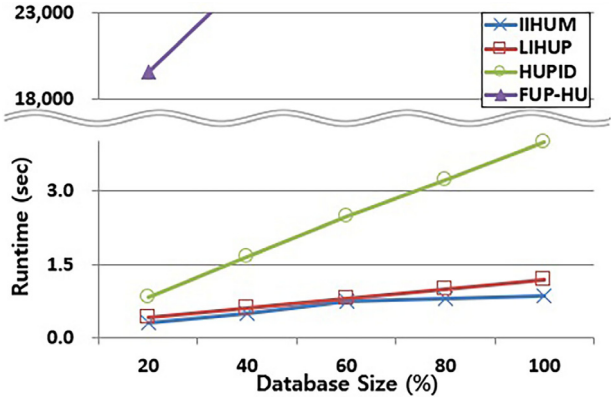


Fig. 10. Runtime test on Connect (minutil = 85%).

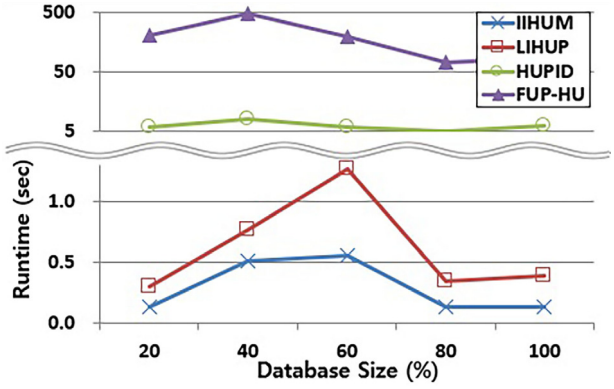


Fig. 11. Runtime test on Mushroom (minutil = 20%).

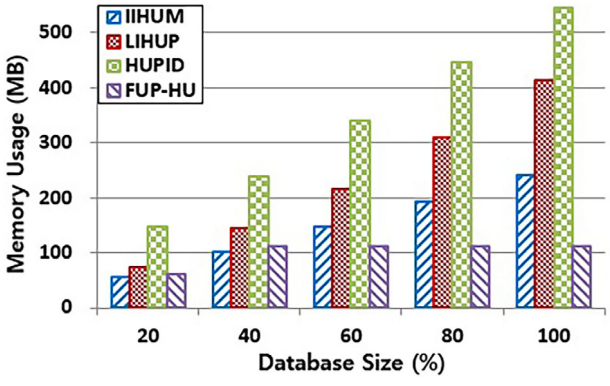


Fig. 12. Memory usage test on Accidents (minutil = 30%).

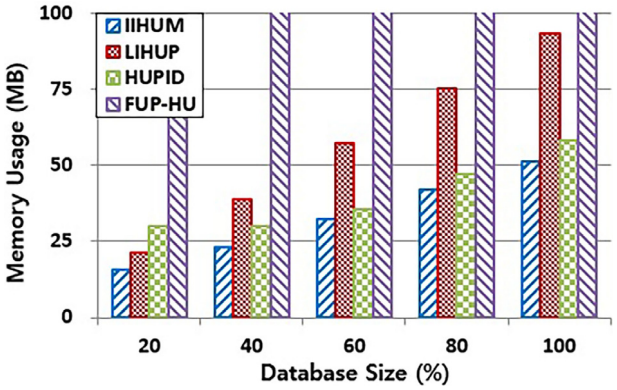


Fig. 13. Memory usage test on Connect (minutil = 85%).

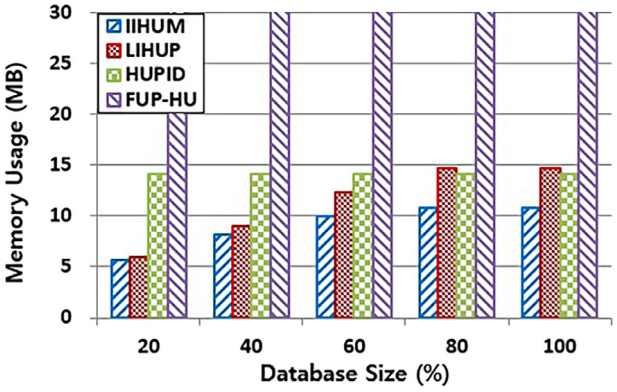


Fig. 14. Memory usage test on Mushroom (minutil = 20%).

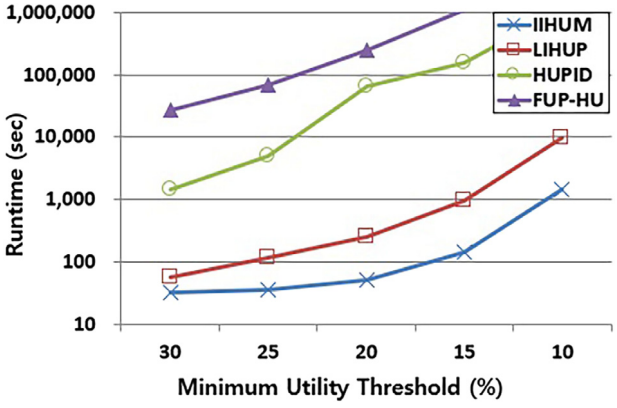


Fig. 15. Runtime test on Accidents.

except for FUP-HU and LIHUP. And the result of Fig. 14 shows that the memory efficiency gap of the three algorithms except FUP-HU is not large. Connect datasets and Mushroom datasets are very dense and smaller than Accidents datasets. Nevertheless, IIHUM guarantees the most efficient memory performance as shown in the figure since the method constructs its own data structures with less memory and does not extract any candidate.

4.2.2. Runtime and memory tests on changing threshold settings

Figs. 15, 16, and 17 present the experimental results of runtime and memory usage on changing threshold settings, where FUP-HU cannot be expressed partially from the graphs because of its too slow runtime speed and excessive memory usage. Although HUPID operates normally at relatively high threshold settings, the

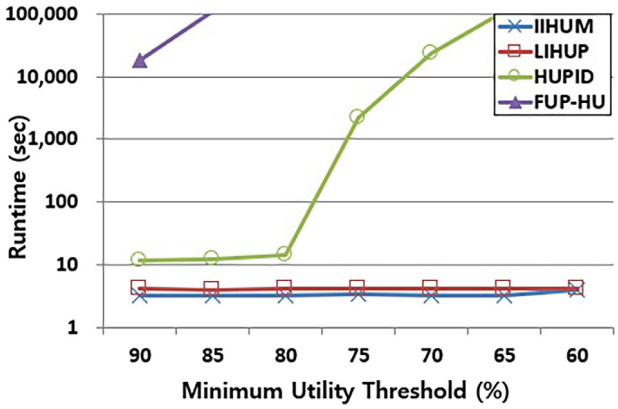


Fig. 16. Runtime test on Connect.

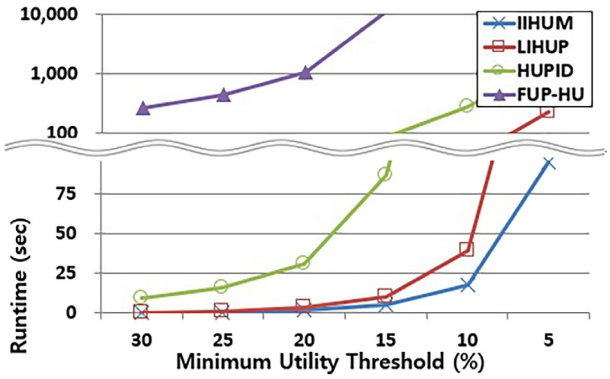


Fig. 17. Runtime test on Mushroom.

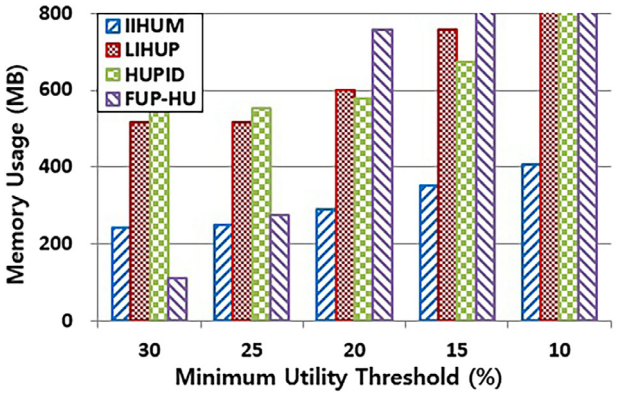


Fig. 18. Memory usage test on Accidents.

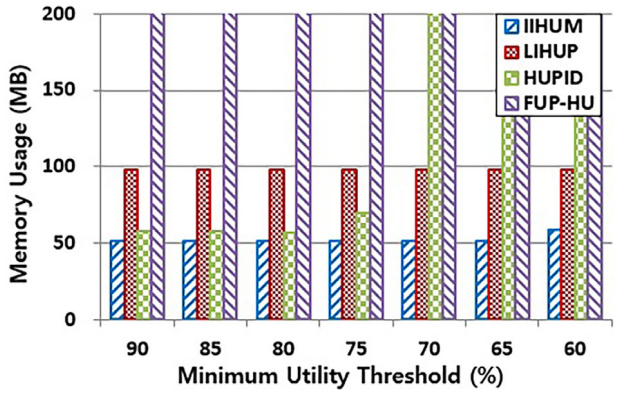


Fig. 19. Memory usage test on Connect.

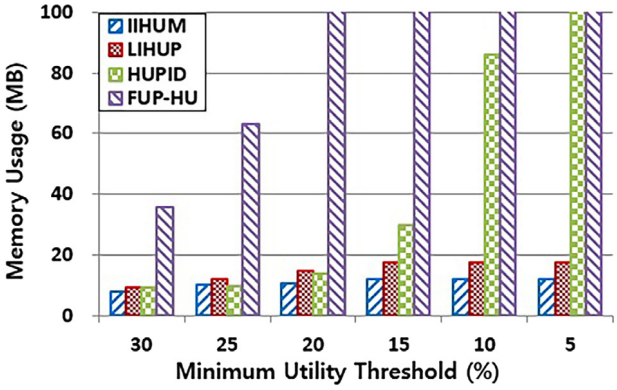


Fig. 20. Memory usage test on Mushroom.

method fails to mine high utility patterns as minutil becomes lower because it requires excessive runtime and memory resources As can be seen in Fig. 15, while the minutil changes from 30% to 15%, the runtime of LIHUP and IIHUM increases slowly. On the other hand, the runtime of HUPID increases dramatically. While minutil was changed from 30% to 20%, the runtime of FUP-HU, HUPID, LIHUP and our IIHUM increased by about 9.4, 44.49, 4.49, and 1.5 times, respectively. As you can see the other figures, we can see that our algorithm is good for run time performance as the threshold increases. The reason why HUPID fails to operate normally at relatively low threshold settings is that the algorithm has to construct a large number of local tree structures with complicated forms during the recursive process and generate many candidate patterns. Meanwhile, the proposed algorithm generates more efficient list data structures in during the recursive process and extracts no candidate patterns, which allows our approach to mine actual high utility patterns more quickly and efficiently.

Figs. 18, 19, and 20 present the experimental results of memory usage on changing threshold settings, where FUP-HU and HUPID cannot be expressed partially from the graphs because of its excessive memory usage. Since both LIHUP and IIHUM are incremental approaches, they consume stable memory space to maintain their own data structures regardless of threshold settings although their memory usage is different from each other. Therefore, the difference in their memory usage on changing threshold settings is mainly affected by a series of the recursive pattern expansion processes and the candidate pattern generation and verification tasks. In Figs. 18 and 20, the memory usage of the FUP-HU sharply increases as the threshold decreases. LIHUP and HUPID show a relatively small memory usage change from the threshold of 30% to 20%, and when the threshold is decreased, the memory usage change rapidly increases in Fig. 18. Such a tendency also similarly occurs in the results of Mushroom in Fig. 20. Meanwhile, because the proposed method does not perform any operations for the candidate generation and verification works, it can guarantee more stable memory consumption. We can determine from the above experiments that the proposed algorithm can mine high utility patterns most efficiently in almost all cases.

4.3. Performance comparison on real sparse datasets

In this section, we provide the results of the performance evaluation tests for the sparse datasets, Chain-store, Retail, and Kosarak.

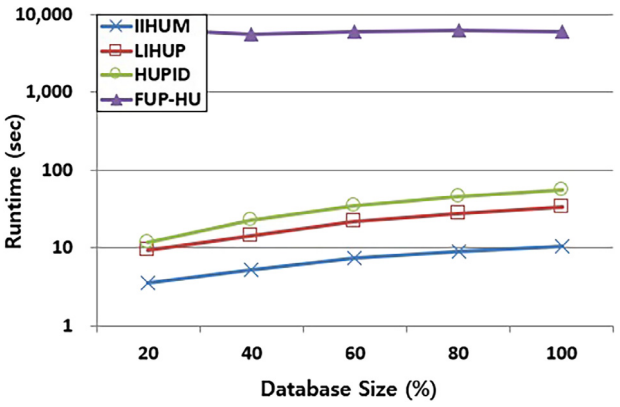


Fig. 21. Runtime test on Chain-store (minutil = 0.2%).

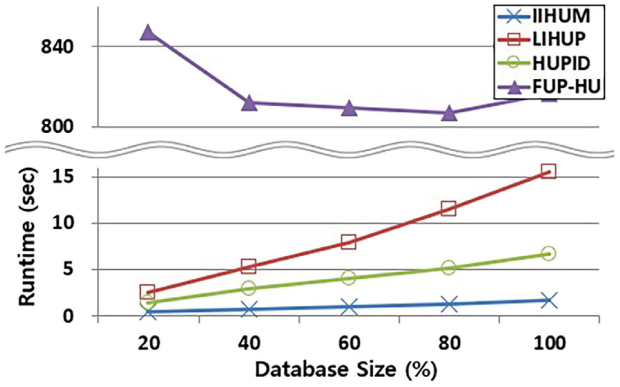


Fig. 22. Runtime test on Retail (minutil = 0.2%).

4.3.1. Runtime and memory tests on increasing data size

Figs. 21–23 show the runtime results of the sparse datasets, Chain-store, Retail, and Kosarak, where minutil has been set to 0.2%, 0.2%, and 2%. As shown in the figures, except for the FUP-HU algorithm, other algorithms require more runtime resources as the size of a given dataset becomes larger and more data are accumulated. Chain-store and Kosarak are sparse dataset larger than Retail. The proposed method shows a more stable runtime increase than that of HUPID and LIHUP for the gradually increasing data size. Although HUPID and LIHUP can mine high utility patterns for the dataset, its runtime efficiency lags behind that of ours. In addition, the larger the data size is, the worse the efficiency becomes. FUP-HU has the worst performance in every case; meanwhile, the others guarantee much better performance. Especially, we can determine that IIHUM shows the best runtime performance in all the cases because of its own data structures and mining techniques.

Figs. 24, 25, and 26 present the memory usage results of the Chain-store, Retail, and Kosarak datasets on increasing data sizes, where the threshold settings are 0.2%, 0.2%, and 2%, respectively. The proposed algorithm guarantees faster runtime performance as well as more efficient memory usage as shown in the figure. Because Chain-store and Kosarak are sparse dataset, it is hard for HUPID to have the node sharing effect which is an advantage of the tree structure. Hence, as the data size becomes larger, the memory usage of the algorithm is increased more than that of the proposed method. In the case of the Retail dataset, FUP-HU mines high utility patterns but its memory efficiency is the worst because this Apriori-based approach spends huge memory space in generating an enormous number of candidate patterns and verifying them. The figures show that LIHUP has less memory usage than FUP-HU and HUPID, but more memory usage than IIHUM in almost all cases.

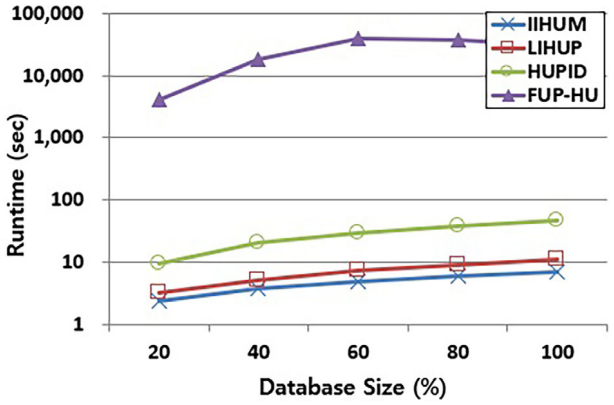


Fig. 23. Runtime test on Kosarak (minutil = 2%).

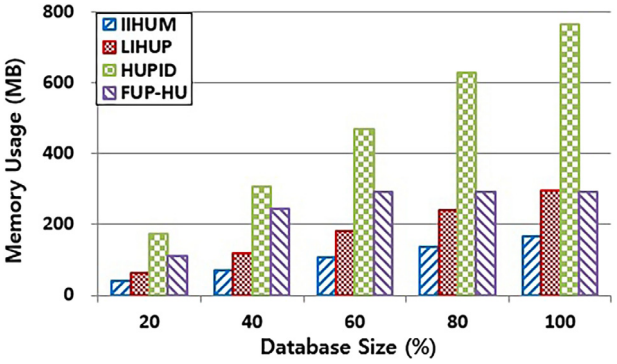


Fig. 24. Memory usage test on Chain-store (minutil = 0.2%).

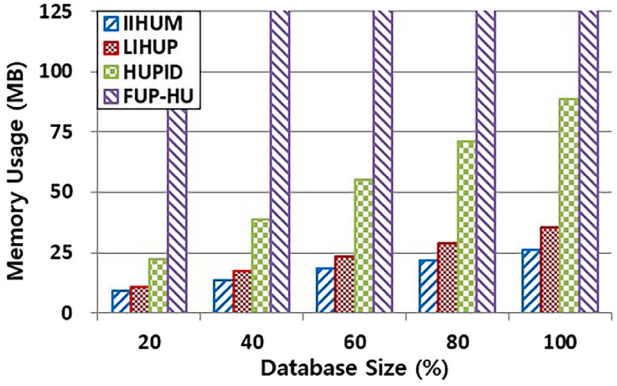


Fig. 25. Memory usage test on Retail (minutil = 0.2%).

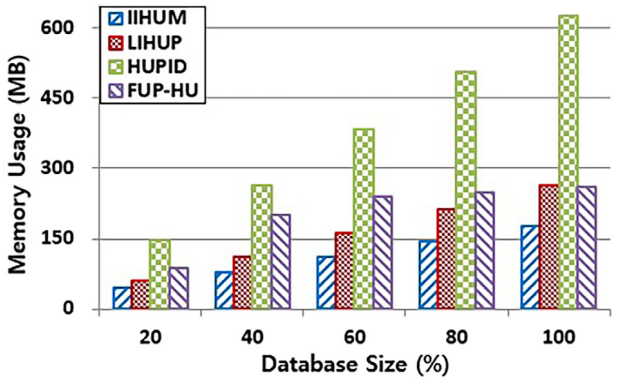


Fig. 26. Memory usage test on Kosarak (minutil = 2%).

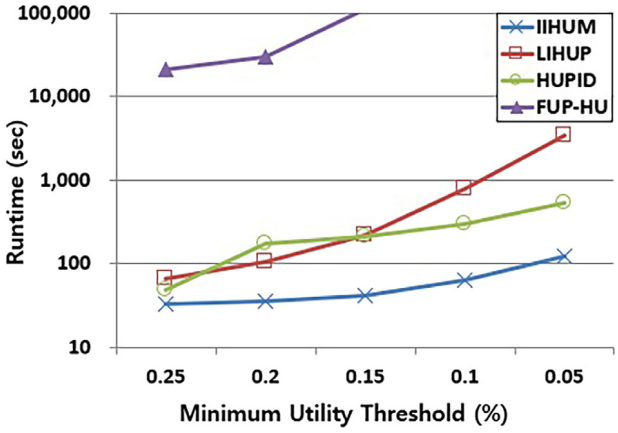


Fig. 27. Runtime test on Chain-store.

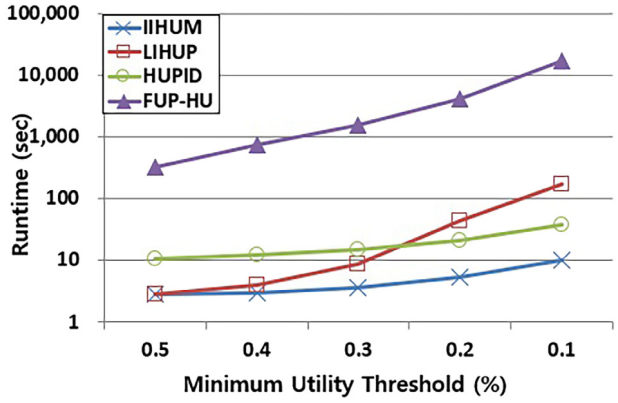


Fig. 28. Runtime test on Retail.

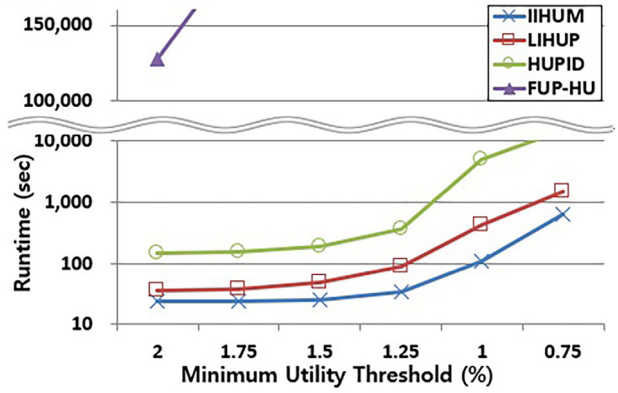


Fig. 29. Runtime test on Kosarak.

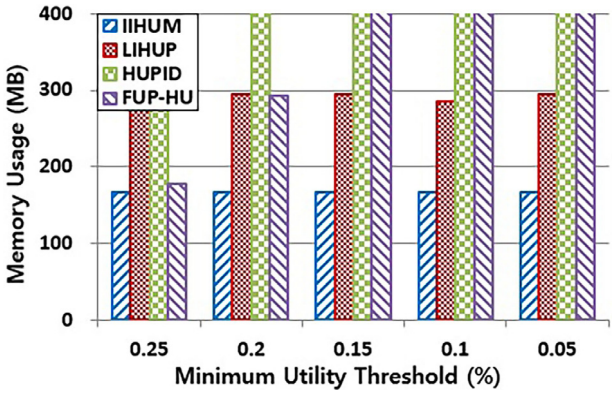


Fig. 30. Memory usage test on Chain-store.

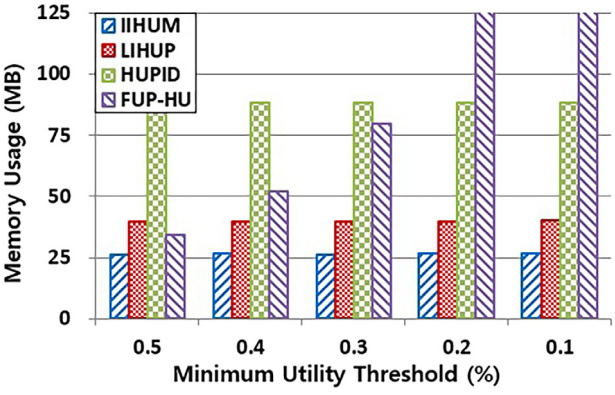


Fig. 31. Memory usage test on Retail.

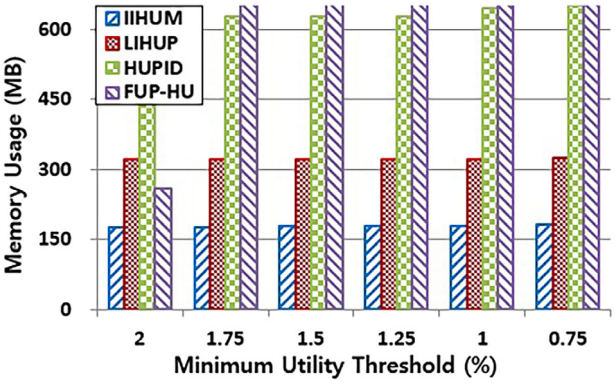


Fig. 32. Memory usage test on Kosarak.

On the other hand, by using the proposed simple but efficient data structure and mining final results of high utility patterns without any candidate generation, IIHUM guarantees the most efficient memory performance.

4.3.2. Runtime and memory tests on changing threshold settings

Figs. 27, 28, and 29 present the experimental results of runtime on changing threshold. In the Chain-store datasets in Fig. 10, when the threshold is 0.2% or later, the results of FUP-HU are excluded. Because the algorithm requires a runtime that is too long for the Chain-store dataset, such as 200,000 s. Although the method succeeds in operating for the Retail dataset, it has significantly poor runtime performance compared to the other three algorithms. In the Kosarak dataset in Fig. 29, the performance of the FUP-HU was not good, and only the result of the threshold of 2% (approximately 130,000 s) could be shown. As shown in Figs. 27, 28, and 29, the proposed algorithm guarantees the best performance in every case. HUPID has good performance with respect to the Retail dataset with relatively small data size, but we can observe that its runtime efficiency is sharply decreased in the case of Kosarak with large data size. When the threshold is changed from 2% to 1.25%, HUPID, LIHUP, and our IIHUM are not significantly different in Fig. 29. However, the runtime increases sharply when the threshold changes from 1.25% to 1%. The difference in runtime is that HUPID is the largest at 13.8 times and IIHUM is the smallest at 3.26 times.

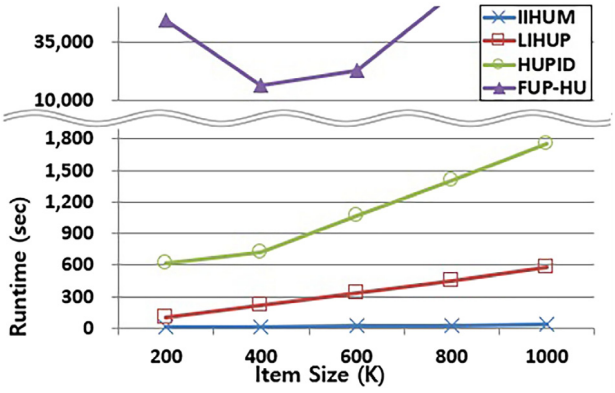


Fig. 33. Runtime scalability on T10.I4.DxK.

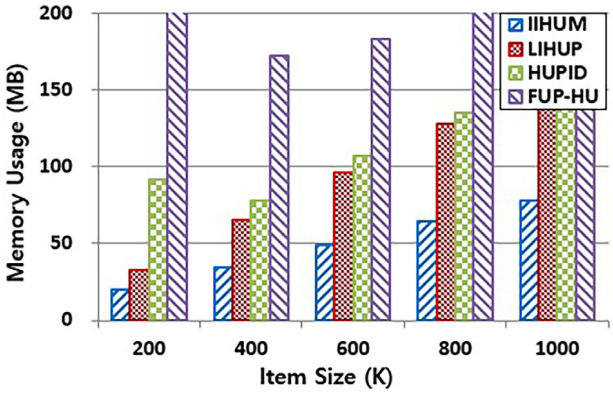


Fig. 34. Memory scalability on T10.I4.DxK.

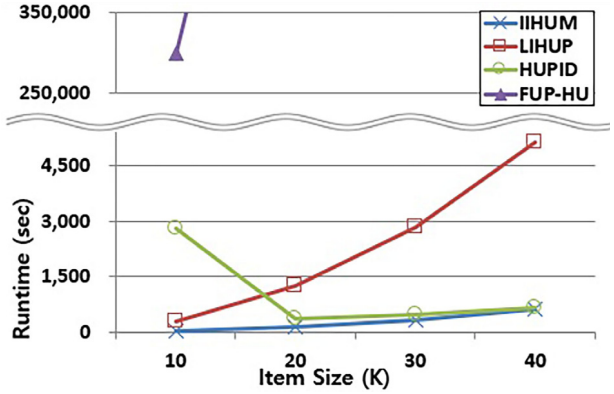


Fig. 35. Runtime scalability on Ta.Nb.Lc.

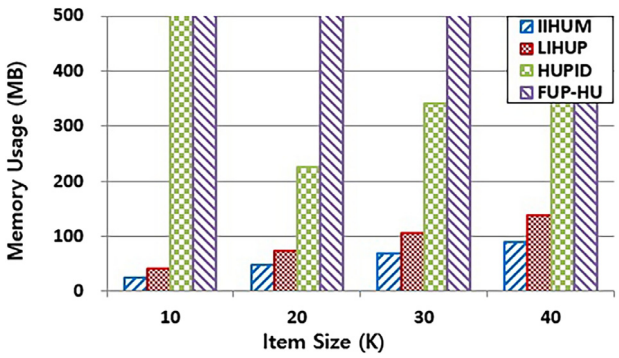


Fig. 36. Memory scalability on Ta.Nb.Lc.

Figs. 30, 31, and 32 present the experimental results of memory usage on changing threshold settings, where FUP-HU and HUPID cannot be expressed partially with result value from the graphs because of its excessive memory usage. In Figs. 30, 31, and 32, HUPID, LIHUP, and IIHUM show almost constant memory consumption regardless of the threshold settings. Meanwhile, the memory usage of FUP-HU is gradually increased according to the decrease of the threshold. Such observations indicate that the construction and maintenance of their main data structures occupy most of the memory usage in HUPID, LIHUP and IIHUM. On the other hand, their recursive pattern mining processes (including the candidate verification works in the case of HUPID) have little effect on the overall memory usage. However, since FUP-HU has to generate and verify an enormous number of high utility candidate patterns in this case, it consumes more and more memory as shown in the figure. Meanwhile, because the proposed method does not perform any operations for the candidate generation and verification works, it can guarantee more stable memory consumption. We can determine from the above experiments that the proposed algorithm can mine high utility patterns most efficiently in almost all cases.

4.4. Scalability test

We evaluate the scalability of the algorithms using the groups of the synthetic datasets shown in Table 4, T10.I4.DxK, and Ta.Nb.Lc. Figs. 33 and 34 are the experimental results of the scalability for the first group, T10.I4.DxK, where minutil has been set to 0.01% and x changes from 200 to 1000. The number of the other attributes in T10.I4.DxK is fixed. In the experiment, we used a smaller file, about 3/4 the size of each dataset, to get the results for all the algorithms. Nevertheless, FUP-HU cannot be expressed partially with result value from the results of the experiments because of its performance limitations. As the data size (i.e., the number of transactions) increases, the runtime and memory usage of the three algorithms HUPID, LIHUP, and IIHUM are also increased. The FUP-HU has very high runtime and memory usage when the item size is 200k, but it gets smaller at 400k and the runtime and memory usage increase as the data size increases from 400k. As data size increases, HUPID increases dramatically from 400k compared to other algorithms. On the other hand, we can observe that the proposed algorithm guarantees very stable runtime scalability while the increasing rate of the runtime in LIHUP is much larger than that of IIHUM. Such a tendency similarly appears in the memory scalability test. The reason why the memory usage in HUPID is larger when K is 200 than when it is 400 is that the algorithm generates a larger number of candidate patterns when K is 200 because of the utility values which have been set randomly. Meanwhile, IIHUM does not have any effect because it generates no candidates during the mining process. Therefore, we can see that our algorithm IIHUM shows better runtime efficiency and memory efficiency than other algorithms.

Figs. 35 and 36 show the runtime and memory scalability tests for the second synthetic dataset group, Ta.Nb.Lc, where minutil has been set to 0.018% and the ranges of a, b, and c are 10 to 40, 10 000 to 40 000, and 1000 to 4000, respectively. As in the case of the previous scalability test, the results of FUP-HU cannot be expressed partially with result value from the graphs because of its performance limitations. HUPID shows the worst performance when the item size is 10 since the algorithm generates a larger number of candidate patterns compared to the other settings. In other words, the algorithm performance of HUPID is strongly influenced depending on the state of given datasets, which leads to unstable performance as shown in the figures. LIHUP increases

runtime dramatically as item size grows. When the item size is 10K, the runtime efficiency is much better than HUPID, but when the item size is 20K, the runtime efficiency drops sharply. Also, its memory efficiency is better than HUPID and less than IIHUM in terms of memory usage compared to runtime. On the other hand, our method guarantees the most stable runtime and memory scalability on the datasets with an increasing number of attributes because the algorithm is not related to any negative effects caused by the candidate pattern generation and verification tasks.

From the various experimental results above, we can determine that the proposed algorithm can extract high utility patterns most efficiently in the incremental data mining environments.

5. Conclusions

In this paper, we proposed a new efficient algorithm for mining high utility patterns on incremental non-binary databases. Based on the newly designed indexed-list data structures, the proposed method achieved the performance improvement of incremental high utility pattern mining by preventing candidate patterns from being extracted. In addition, we suggested a restructuring technique for dealing with incremental data more efficiently in our data structures. Through the new data structures and various mining techniques, we solved the problems of fatal performance degradation caused by the previous state-of-the-art incremental approaches. The experimental results from the various real and synthetic benchmark datasets proved that the proposed algorithm can mine high utility patterns more efficiently from incremental non-binary databases compared to the previous approaches. The incremental data processing techniques proposed in this paper can also be applied to different types of stream pattern mining areas such as sliding window and damped window model-based data mining environments. We are scheduled to research such advanced works in our future work.

Acknowledgments

This research was supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology, Republic of Korea (NRF No. 20152062051 and NRF No. 20189083109).

References

[1] K. Sabo, R. Scitovski, An approach to cluster separability in a partition, Inform. Sci. 305 (2015) 208–218.
[2] J. Sá Jr., Shape classification using line segment statistics, Inform. Sci. 305 (2015) 349–356.
[3] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings of the 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile, 1994, pp. 487–499.
[4] Q. Huynh-Thi-Le, T. Le, B. Vo, H.B. Le, An efficient and effective algorithm for mining top-rank-k frequent patterns, Expert Syst. Appl. 42 (1) (2015) 156–164.
[5] G.-C. Lan, T.-P. Hong, V.S. Tseng, S.-L. Wang, Applying the maximum utility measure in high utility sequential pattern mining, Expert Syst. Appl. 41 (11) (2014) 5071–5081.
[6] G.C. Lan, T.P. Hong, V.S. Tseng, An efficient projection-based indexing approach for mining high utility itemsets, Knowl. Inf. Syst. 38 (1) (2014) 85–107.
[7] C.W. Wu, Y.F. Lin, P.S. Yu, V.S. Tseng, Mining high utility episodes in complex event sequences, in: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, IL, USA, 2013, pp. 536–544.
[8] W. Ismail, M. Hassan, H. Alsalamah, Mining of productive periodic-frequent patterns for IoT data analytics, Future Gener. Comput. Syst. 88 (1) (2018) 512–523.
[9] S. Tanbeer, M. Hassan, A. Almogren, M. Zuair, Scalable regular pattern mining in evolving body sensor data, Future Gener. Comput. Syst. 75 (1) (2017) 172–186.
[10] C. Zou, H. Deng, J. Wan, Mining and updating association rules based on fuzzy concept lattice, 82(1), 698-706, 2018.
[11] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA, 2000, pp. 1–12.
[12] C.F. Ahmed, S.K. Tanbeer, B.S. Jeong, H.J. Choi, Interactive mining of high utility patterns over data streams, Expert Syst. Appl. 39 (15) (2012) 11979–11991.
[13] C.F. Ahmed, S.K. Tanbeer, B.S. Jeong, Y.K. Lee, H.J. Choi, Single-pass incremental and interactive mining for weighted frequent patterns, Expert Syst. Appl. 39 (9) (2012) 7976–7994.
[14] C.F. Ahmed, S.K. Tanbeer, B.S. Jeong, Y.K. Lee, Efficient tree structures for high utility pattern mining in incremental databases, IEEE Trans. Knowl. Data Eng. 21 (12) (2009) 1708–1721.
[15] D. Kim, U. Yun, Mining high utility itemsets based on the time decaying model, Intell. Data Anal. 20 (5) (2016) 1157–1180.
[16] J. Liu, K. Wang, B.C.M. Fung, Direct discovery of high utility itemsets without candidate generation, in: Proceedings of the 12th IEEE International Conference on Data Mining, Brussels, Belgium, 2012, pp. 984–989.
[17] H. Ryang, U. Yun, High utility pattern mining over data streams with sliding window technique, Expert Syst. Appl. 57 (2016) 214–231.
[18] P. Fournier-Viger, C.W. Wu, S. Zida, V.S. Tseng, FHM: faster high-utility itemset mining using estimated utility co-occurrence pruning, in: Proceedings of the 21st International Symposium on Methodologies for Intelligent Systems, Roskilde, Denmark, 2014, pp. 83–92.
[19] G. Lee, U. Yun, H. Ryang, D. Kim, Approximate maximal frequent pattern mining with weight conditions and error tolerance, Int. J. Pattern Recognit. Artif. Intell. 30 (6) (2016) 1–42.
[20] G. Lee, U. Yun, A new efficient approach for mining uncertain frequent patterns using minimum data structure without false positives, Future Gener. Comput. Syst. 68 (2017) 89–110.
[21] G. Lee, U. Yun, K. Ryu, Mining frequent weighted itemsets without storing transaction IDs and generating candidates, Internat. J. Uncertain. Fuzziness Knowledge-Based Systems 25 (1) (2017) 111–144.
[22] L. Troiano, G. Scibelli, Mining frequent itemsets in data streams within a time horizon, Data Knowl. Eng. 89 (2014) 21–37.
[23] U. Yun, H. Ryang, Incremental high utility pattern mining with static and dynamic databases, Appl. Intell. 42 (2) (2015) 323–352.
[24] G. Lee, U. Yun, Single-pass based efficient erasable pattern mining using list data structure on dynamic incremental databases, Future Gener. Comput. Syst. 80 (1) (2018) 12–28.
[25] T.P. Hong, K.-Y. Lin, C.-W. Lin, B. Vo, An incremental mining algorithm for erasable itemsets, in: IEEE International Conference on Innovations in Intelligent Systems and Applications (INISTA), 2017, pp. 286–289.
[26] M. Zihayat, A. An, Mining top-k high utility patterns over data streams, Inform. Sci. 285 (20) (2014) 138–161.
[27] H. Liu, K. Zhou, P. Zhao, S. Yao, Mining frequent itemsets over uncertain data streams, Int. J. High Perform. Comput. Netw. (IJHPCN) 11 (4) (2018) 312–321.
[28] H. Li, N. Zhang, J. Zhu, Y. Wang, H. Cao, Probabilistic frequent itemset mining over uncertain data streams, Expert Syst. Appl. 112 (2018) 274–287.
[29] L. Feng, L. Wang, B. Jin, UT-Tree: efficient mining of high utility itemsets from data streams, Intell. Data Anal. 17 (4) (2013) 585–602.
[30] C.-W. Lin, G.-C. Lan, T.-P. Hong, An incremental mining algorithm for high utility itemsets, Expert Syst. Appl. 39 (8) (2012) 7173–7180.
[31] W. Gan, C.-W. Lin, P. Fournier-Viger, H.-C. Chao, T.-P. Hong, A survey of incremental high-utility itemset mining, Wiley Interdisc. Rew.: Data Min. Knowl. Discovery 8 (2) (2018).
[32] Y. Liu, W.K. Liao, A.N. Choudhary, A two-phase algorithm for fast discovery of high utility itemsets, in: Proceedings of the 9th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, Hanoi, Vietnam, 2005, pp. 689–695.
[33] H. Ryang, U. Yun, Indexed list-based high utility pattern mining with utility upper-bound reduction and pattern combination techniques, Knowl. Inf. Syst. 51 (2) (2017) 627–659.
[34] J. Yin, Z. Zheng, L. Cao, Y. Song, W. Wei, Efficiently mining top-k high utility sequential patterns, in: Proceedings of the 13th IEEE International Conference on Data Mining, 2013, pp. 1259–1264.
[35] U. Yun, H. Ryang, K. Ryu, High utility itemset mining with techniques for reducing overestimated utilities and pruning candidates, Expert Syst. Appl. 41 (8) (2014) 3861–3878.
[36] Y.C. Li, J.S. Yeh, C.C. Chang, Isolated items discarding strategy for discovering high utility itemsets, Data Knowl. Eng. 61 (1) (2008) 198–217.
[37] V.S. Tseng, C.W. Wu, B.E. Shie, P.S. Yu, UP-Growth: an efficient algorithm for high utility itemset mining, in: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, 2010, pp. 253–262.
[38] V.S. Tseng, B.E. Shie, C.W. Wu, P.S. Yu, Efficient algorithms for mining high utility itemsets from transactional databases, IEEE Trans. Knowl. Data Eng. 25 (8) (2013) 1772–1786.
[39] P. Fournier-Viger, S. Zida, C.-W. Lin, C.-W. Wu, V.S. Tseng, EFIM-closed: fast and memory efficient discovery of closed high-utility itemsets, in: International Conference on Machine Learning and Data Mining in Pattern Recognition, New York, NY, USA, 2016, pp. 199–213.

[40] C.-W. Lin, W. Gan, P. Fournier-Viger, T.-P. Hong, H.-C. Chao, FDHUP: Fast algorithm for mining discriminative high utility patterns, Knowl. Inf. Syst. 51 (3) (2017) 873–909.
[41] M. Liu, J.F. Qu, Mining high utility itemsets without candidate generation, in: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, Maui, HI, USA, 2012, pp. 55–64.
[42] H. Ryang, U. Yun, G. Lee, H. Fujita, An efficient algorithm for mining high utility patterns from incremental databases with one database scan, Knowl. Inf. Syst. 124 (2017) 188–206.
[43] D.W.-L. Cheung, J. Han, V.T.Y. Ng, C.Y. Wong, Maintenance of discovered association rules in large databases: An incremental updating technique, in: Proc. of the 12th International Conf. on Data Engineering, 1996, pp. 106–114.
[44] C.-W. Lin, T.-P. Hong, W. Gan, H.-Y. Chen, S.-T. Li, Incrementally updating the discovered sequential patterns based on pre-large concept, Intell. Data Anal. 19 (5) (2015) 1071–1089.
[45] C.-W. Lin, T.-P. Hong, G.-C. Lan, J.-W. Wong, W.-Y. Lin, Efficient updating of discovered high-utility itemsets for transaction deletion in dynamic databases, Adv. Eng. Inform. 29 (1) (2015) 16–27.
[46] W. Song, Y. Liu, J. Li, Mining high utility itemsets by dynamically pruning the tree structure, Appl. Intell. 40 (1) (2014) 29–43.
[47] V.S. Tseng, C.-W. Wu, P. Fournier-Viger, P.S. Yu, Efficient algorithms for mining the concise and lossless representation of high utility itemsets, IEEE Trans. Knowl. Data Eng. 27 (3) (2015) 726–739.
[48] J. Pisharath, Y. Liu, B. Ozisikyilmaz, R. Narayanan, W.K. Liao, A. Choudhary, G. Memik, NU-MineBench version 2.0 dataset and technical report. URI: http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html, 2005.

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. He was with the Multimedia Laboratory, Korea Telecom, from 1997 to 2002. After receiving the Ph.D. degree, he was a Postdoctoral Associate for almost one year in the Computer Science Department, Texas A&M University. After that, he was a Senior Researcher with the Electronics and Telecommunications Research Institute. In March 2007, he joined the School of Electrical and Computer Engineering, Chungbuk National University, South Korea. Since August 2013, he has been with the Department of Computer Engineering, Sejong University, Seoul, South Korea. His research interests include data mining, information retrieval, database systems, artificial intelligence, and digital libraries.