Towards Efficiently Mining Closed High Utility Itemsets from Incremental Databases

Thu-Lan Dam, Heri Ramampiaro, Kjetil Nørvåg, Quang-Huy Duong

Department of Computer Science, Norwegian University of Science and Technology, Trondheim, Norway

Abstract

The set of closed high-utility itemsets (CHUIs) concisely represents the exact utility of all itemsets. Yet, it can be several orders of magnitude smaller than the set of all high-utility itemsets. Existing CHUI mining algorithms assume that databases are static, making them very expensive in the case of incremental data, since the whole dataset has to be processed for each batch of new transactions. To address this challenge, this paper presents the first approach, called IncCHUI, that mines CHUIs efficiently from incremental databases. In order to achieve this, we propose an incremental utility-list structure, which is built and updated with only one database scan. Further, we apply effective pruning strategies to increase the speed of construction of incremental utility-lists and eliminate candidates that are not updated. Finally, we suggest an efficient hash-based approach to update or insert new closed sets that are found. Our extensive experimental evaluation on both real-life and synthetic databases shows the efficiency, as well as the feasibility of our approach. It significantly outperforms previously proposed methods, that are mainly run in batch mode, in terms of speed, and it is scalable with respect to the number of transactions.

Keywords: High-utility itemset mining, Closed itemset mining, Incremental mining, Incremental utility list

1. Introduction

High Utility Itemset Mining (HUIM) is an extensively studied data mining task [1, 2], which extends Frequent Itemset Mining (FIM) [3] by considering the case where items can appear more than once in each transaction, and each item has a weight, e.g., unit profit. Therefore, HUIM can be used to discover itemsets having a high-utility, e.g., high profit. An itemset is a High-Utility Itemset (HUI) if its utility (or the profit that it yields in a database) is no less than a user-specified threshold. Recently, many efficient algorithms have been proposed to mine high-utility itemsets in static and incremental transaction databases [2, 4, 5, 6]. However, a drawback of traditional HUIM algorithms is that the set of HUIs can be very large, which depends on how the minimum utility parameter is specified by the user. In general, when a HUIM algorithm generates more HUIs, its execution time and memory consumption also greatly increase. In addition, the task of analyzing a large number of HUIs produced by a HUIM algorithm is difficult and time-consuming.

To address these issues, a compact and lossless representation of HUIs, named closed HUIs (CHUIs), was proposed by Tseng et al. [7]. This representation is inspired by the concept of frequent closed patterns [8, 9, 10], which was originally introduced in FIM for the mining of non-redundant (minimal) association rules to improve the performance in terms of memory usage and mining time [11, 12, 13]. In HUIM, an itemset is said to be a CHUI if (1) its utility is no less than the minimum utility threshold, and (2) it has no supersets that appear in the same transactions [7]. The set of CHUIs is interesting because it can be several orders of magnitude smaller than the set of all HUIs, and it allows deriving all HUIs without re-scanning the database. Moreover, the set of CHUIs also provides meaningful information to decision-makers since they are the largest HUIs that are common to groups of customers [7]. For example, let X be a CHUI, and Y be a non-CHUI, where $X \supset Y$, and $u(X) > u(Y)$. In market basket analysis, this means that no customer

Email addresses: lanfict@gmail.com (Thu-Lan Dam), heri@ntnu.no (Heri Ramampiaro), noervaag@ntnu.no (Kjetil Nørvåg), huydqyb@gmail.com (Quang-Huy Duong)

purchases Y without X. Thus, when a customer purchases Y, the retailer can recommend $X - Y$ to the customer to maximize the profit.

There are many efficient algorithms for mining CHUIs in the literature, such as CHUD (Closed$^+$ High Utility Itemset Discovery) [7], CHUI-Miner (Closed$^+$ High Utility Itemset mining) [14], CLS-Miner [15], and EFIM-Closed (EFficient high-utility Itemset Mining - Closed) [16]. However, common to these methods is that they assume that databases are static. To the best of our knowledge, there does not exist any method for maintaining CHUIs in dynamic databases. Meanwhile, in real-life applications such as online retail stores, new customer transactions are generated and added to the database all the time. As a result, previously found itemsets may be invalid, and new ones may appear. In this work, we address this need by introducing the task of incremental mining CHUIs, and proposing an algorithm named IncCHUI (Incremental Closed High-Utility Itemset miner), which introduces several ideas to efficiently maintain crucial information and CHUIs in dynamic databases. Specifically, we make the following main contributions:

- We introduce an incremental utility-list structure by adapting the traditional utility-list to store crucial information of all single items both in the original database and the added transactions. The mining process for the updated database performs searching only on itemsets that have appeared in the inserted transactions. We also introduce an efficient method to construct the incremental utility-lists of itemsets by pruning non-updated itemsets early. Further, we introduce a method to efficiently maintain the validity of these lists by restructuring it whenever a transaction is inserted. After each mining, we merge the utility list section of the original database and that of the added transactions together to form a single utility list. This list is, in turn, used to prepare for the next execution, with which the database will be updated.

- We introduce a novel algorithm named IncCHUI to mine CHUIs efficiently from incremental databases using the incremental utility-list structure. IncCHUI scans the original database or updated section only once to construct the lists of single items. To store the CHUIs discovered so far, we use a hash table called CHT. Next, for each CHUI P that is found when mining on the updated database, the algorithm first checks whether this itemset is already in the CHT. If yes, this means that P was previously found when applying the algorithm on the original database. In that case, we update its utility and support. Otherwise, P is inserted in the table CHT, since it is a new CHUI.

- We perform extensive experiments to evaluate the efficiency of the proposed algorithm on both real-life and synthetic databases which have various different characteristics. In these experiments, we compare the performance of IncCHUI against the state-of-the-art algorithms for mining CHUIs in static databases, run in batch mode. Interestingly, the results show that our algorithm is highly efficient, and it outperforms these state-of-the-art algorithms in terms of speed. Also, while the compared algorithms employ several complex pruning strategies and structures, our method does not.

The rest of this paper is organized as follows. In Section 2, we formally define the problem of incremental mining CHUIs and introduce its preliminaries. In Section 3, we briefly present the related works including HUIM from static and incremental databases, and the state-of-the-art methods for mining CHUIs in static databases. In Section 4, we present the IncCHUI algorithm as well as its components. In Section 5, we present and discuss the experimental results. Finally, in Section 6, we conclude our paper and outline the future work.

2. Preliminaries and problem definition

This section presents preliminaries related to high-utility itemset mining, and defines the problem of incremental closed high-utility itemset mining.

2.1. Problem definition

Let I=$\{i_1, i_2, \ldots, i_m\}$ be a set of m distinct items. Each item $i_j \in I$ is associated with a positive number $p(i_j)$, called the external utility of i_j. This number represents the relative importance of item i_j to the user (e.g., the unit profit of i_j). Let $\mathcal{D}$ be a transaction database containing a set of $n_\mathcal{D}$ transactions, $\mathcal{D}$ = $\{T_1, T_2, \ldots, T_{n_\mathcal{D}}\}$ such that $T_d \subseteq I (1 \leq d \leq n_\mathcal{D})$, and each transaction T_d has a unique identifier d called

Tid (Transaction id). Moreover, given a transaction T_d and an item i_j, let $q(i_j, T_d)$ be the internal utility (e.g., the purchase quantity) of item i_j in T_d, which is a positive integer. The utility of item i_j in transaction T_d is defined as the product of external and internal utilities of i_j in T_d, i.e., $u(i_j, T_d) = q(i_j, T_d) \times p(i_j)$.

An itemset X is a set of k distinct items $X = \{i_1, i_2, \ldots, i_k\}$, where $X \subseteq I$. Then, X is a k-itemset, and its length is k. From now on, for the sake of brevity, each itemset will be denoted by the concatenation of its items. For example, the itemset $\{x, y, z\}$ will be denoted as xyz. The union of two itemsets X and Y will be denoted as XY or $X \cup Y$.

	Tid	Transaction	TU
Original db $\mathcal{D}$	T_1	(a,1), (c,1), (d,1)	8
	T_2	(a,2), (c,6), (e,2), (g,5)	27
	T_3	(a,1), (b,2), (c,1), (d,6),(e,1),(f,5)	30
	T_4	(b,4), (c,3), (d,3), (e,1)	20
	T_5	(b,2), (c,2), (e,1), (g,2)	11
$\mathcal{N}$	T_6	(b,2), (d,5), (f,2)	16
	T_7	(a,1), (c,2), (d,1), (e,1)	12
		...	

Item	a	b	c	d	e	f	g
External utility	5	2	1	2	3	1	1

Figure 1: Example of incremental transaction database with items' quantities and profits.

Example 1. Fig. 1 shows an example a transaction database $\mathcal{D}$, which at the beginning contains five transactions (T_1, T_2, T_3, T_4, and T_5). Then, one or more transactions are inserted into the original database $\mathcal{D}$, i.e., T_6 and T_7 are added. The added section is named $\mathcal{N}$. In this example, the set of items is $I = \{a, b, c, d, e, f, g\}$. The lower table presents external utilities (e.g., unit profits) of these items. The items a, b, c, d, e, f, and g have external utilities of 5, 2, 1, 2, 3, 1, and 1, respectively. The itemset ac appears in transactions T_1, T_2, and T_3. The items a, c, e, and g respectively have purchase quantities of 2, 6, 2, and 5, in transaction T_2. The utility of item a in transaction T_1 is $u(a, T_1) = 1 \times 5 = 5$. Similarly, the utility of item c in transaction T_1 is $u(c, T_1) = 1 \times 1 = 1$.

Definition 1 (Utility of an itemset in a transaction)**.** The utility of an itemset X in a transaction T_d ($X \subseteq T_d$) is denoted as $u(X, T_d)$ and defined as $u(X, T_d) = \sum_{i \in X} u(i, T_d)$.

For example, in the database of Fig. 1, $u(ac, T_1) = 1 \times 5 + 1 \times 1 = 6$, and $u(ac, T_2) = 2 \times 5 + 6 \times 1 = 16$.

Definition 2 (Transaction utility)**.** The utility of a transaction T_d is denoted as $TU(T_d)$, and is calculated as $TU(T_d) = u(T_d, T_d)$.

For example, in the database of Fig. 1, $TU(T_1) = 8$, $TU(T_2) = 27$, $TU(T_3) = 30$, $TU(T_4) = 20$, and $TU(T_5) = 11$.

Definition 3 (Utility of an itemset)**.** The utility of an itemset X in a database $\mathcal{D}$ is defined as $u(X) = \sum_{X \subseteq T_d \wedge T_d \in \mathcal{D}} u(X, T_d)$.

For the original database in Fig. 1, $u(ac) = u(ac, T_1) + u(ac, T_2) + u(ac, T_3) = 6 + 16 + 6 = 28$. Let *minutil* be a user-specified minimum utility threshold such that $0 < minutil$. If the utility of an itemset X is no less than *minutil*, $u(X) \geqslant minutil$, then X is said to be a high-utility itemset (HUI). Otherwise, X is a low-utility itemset. The problem of HUIM concerns discovering all HUIs in a given transaction database.

Definition 4 (Support and Tid set of an itemset)**.** The Tid set (transaction id set) of an itemset X is the set of Tids of transactions containing X, and is denoted as $TidSet(X)$. The *support* of an itemset X is denoted as $sup(X)$, and defined as $sup(X) = |TidSet(X)|$.

Definition 5 (Closed high-utility itemset)**.** An itemset X is a *closed high-utility itemset* (CHUI) if there exists no proper superset $Y \supset X$ in the database such that $sup(X) = sup(Y)$ and its utility $u(X) \geq minutil$ [7, 14].

Given a user-specified minimum utility threshold *minutil*, the task of *closed high-utility itemset mining* is to discover all closed itemsets in the database having utilities that are no less than the *minutil* threshold [7, 14].

Example 2. Consider the original database of Fig. 1, and suppose that *minutil* is set to 30. The set of HUIs is $\mathcal{H} = \{bd{:}30,\ ace{:}31,\ bcd{:}34,\ bce{:}37,\ bde{:}36,\ bcde{:}40,$ and $abcdef{:}30\}$, where the number beside each itemset indicates its utility. Among those itemsets, the CHUIs are $\mathcal{CH} = \{ace{:}31,\ bce{:}37,\ bcde{:}40,$ and $abcdef{:}30\}$.

Problem statement. Let there be a database $\mathcal{D}$. A database $\mathcal{DN}$ is an update of database $\mathcal{D}$ if $\mathcal{DN} = \mathcal{D} \cup \mathcal{N}$, where $\mathcal{N}$ is a non-empty set of inserted transactions. Let *minutil*, $\mathcal{D}$ and $\mathcal{CH}$, respectively be a user-specified minimum utility threshold, a database, and the set of CHUIs found in $\mathcal{D}$. The problem of incremental closed high utility itemset mining is to find $\mathcal{CH}'$, which is the set of CHUIs in the updated database $\mathcal{DN}$, given *minutil*, $\mathcal{D}$, and $\mathcal{CH}$.

Example 3. Consider the original database $\mathcal{D}$ in Fig. 1, and $minutil = 30$. $\mathcal{DN}$ is the database $\mathcal{D}$ updated by inserting transactions $T_6 = (b,2)(d,5)(f,2)$, and $T_7 = (a,1)(c,2)(d,1)(e,1)$. In the updated database $\mathcal{DN}$, the CHUIs are $\mathcal{CH}'=\{ace{:}41,\ bce{:}37,\ bcde{:}40,\ abcdef{:}30,\ bdf{:}37,\ bd{:}44,\ ac{:}35,$ and $ce{:}32\}$. In this example, the task of incremental closed high-utility itemset mining is to find that utility (support as well) of the previously found CHUI *ace* increases from 31 to 41, and four new CHUIs (*bdf*, *bd*, *ac*, and *ce*) existing in $\mathcal{DN}$ without mining on $\mathcal{DN}$ from the beginning.

From this example, the previously found CHUIs (*bce*, *bcde* and *abcdef*) in the original database remain the same, while four new CHUIs are found as a result of the insertion of the new transactions T_6 and T_7. Further, there exist a scenario where the utility and support of CHUIs in $\mathcal{CH}$, such as with *ace*, are increased when inserting new transactions. Based on this observation, we introduce the following property.

Property 1. Let be an itemset $X \in \mathcal{CH}$ appearing in $\mathcal{D}$ but not appearing in $\mathcal{N}$, and $\mathcal{DN} = \mathcal{D} \cup \mathcal{N}$. Then, the utility and support of X in $\mathcal{DN}$ are the same as in the original database $\mathcal{D}$.

Proof. On the updated database $\mathcal{DN} = \mathcal{D} \cup \mathcal{N}$, the utility of itemset X is calculated by Definition 3 as follows, $u(X) = \sum_{X \subseteq T_d \wedge T_d \in \mathcal{DN}} u(X, T_d) = \sum_{X \subseteq T_d \wedge T_d \in \mathcal{D}} u(X, T_d) + \sum_{X \subseteq T_{d'} \wedge T_{d'} \in \mathcal{N}} u(X, T_{d'})$.

Let TD be the set of transaction identifiers containing X in $\mathcal{D}$, and let TN be the set of transaction identifiers containing X in $\mathcal{N}$. Let TDN be the set of transaction identifiers containing X in the updated database $\mathcal{DN}$. Hence, $TDN = TD \cup TN$, and the support of X according to Definition 4 is $sup(X) = |TidSet(X)| = |TDN(X)| = |TD(X)| + |TN(X)|$. Because X does not appear in $\mathcal{N}$, then $u(X) = \sum_{X \subseteq T_d \wedge T_d \in D} u(X, T_d)$, and $sup(X) = |TD(X)|$. □

To design an efficient algorithm to find $\mathcal{CH}'$, we should thus avoid exploring itemsets that do not appear in $\mathcal{N}$, thanks to Property 1.

In Frequent Itemset Mining (FIM), the downward closure property is employed for reducing the search space. However, this property does not hold with the utility measure in HUIM. In other words, an itemset may have a utility lower, equal or higher than the utility of its subsets. To restore this property, the transaction-weighted utilization (TWU) measure was introduced as an upper-bound on the utility [17], which is defined as follows.

Definition 6. The transaction-weighted utilization (TWU) [17] of an itemset X in a database $\mathcal{D}$ is denoted as TWU(X), and defined as $\text{TWU}(X) = \sum_{T_d \in \mathcal{D} \wedge X \subseteq T_d} TU(T_d)$.

For example, the TU of transactions T_1, T_2, and T_3 of the running example are respectively 8, 27, and 30. Hence, $\text{TWU}(a) = TU(T_1) + TU(T_2) + TU(T_3) = 8 + 27 + 30 = 65$. The following property of the TWU is commonly used to prune the search space in HUIM.

Property 2 (Pruning using the TWU [17])**.** Let there be an itemset X. If $\text{TWU}(X) < minutil$, then X and its supersets are low-utility itemsets.

Proof. The detail proof is provided in [17]. Intuitively, by Definition 6, TWU(X) is the sum of transaction utilities where X appears, TWU(X) must be no less than the utility of X, $u(X)$, and any of its supersets. Hence, for any itemset X, if TWU(X) is less than *minutil*, then X is a low-utility itemset as well as all its supersets, and it should be pruned. □

2.2. Utility-list structure

Many algorithms for mining high utility patterns use the aforementioned TWU measure (see Property 2) to prune the search space [17, 4, 7, 18]. These algorithms are mainly executed in two phases. In the first phase, they identify candidate high-utility patterns by calculating their TWUs. In the second phase, they scan the database and compute the exact utility of all candidates to filter out any low-utility patterns. The main drawback with using TWU is that TWU is a loose upper-bound, and thus numerous candidate patterns need to be considered to find the final set of results. To address this issue, several algorithms that mine high-utility patterns using a single phase have been proposed. They have been shown to be generally more efficient than the two-phase counterparts. To achieve this, one-phase algorithms employ the utility-list structure [19], and rely mainly on the concept of *remaining utility* to prune the search space.

Definition 7 (Utility-list [19])**.** Let $\succ$ be a total order on items from I. The *utility-list* of an itemset X in a database is denoted as *ul(X)*. It contains a tuple of the form $(tid, iutil, rutil)$ for each transaction T_{tid} containing X ($X \subseteq T_{tid}$). The *iutil* element of a tuple corresponding to a transaction T_{tid} stores the utility of X in T_{tid}. i.e., $u(X, T_{tid})$. The *rutil* element of a tuple stores the value $\sum_{i \in T_{tid} \wedge i \succ x \forall x \in X} u(i, T_{tid})$, and it is called the remaining utility.

Example 4. Consider the original database $\mathcal{D}$ in Fig. 1, the utility-list of item a is {(T_1, 5, 3)(T_2, 10, 17)(T_3, 5, 25)}. The utility-list of item e is {(T_2, 6, 5)(T_3, 3, 5)(T_4, 3, 0)}. The utility-list of itemset ae is {(T_2, 16, 5),(T_3, 8, 5)}.

As proposed in the HUI-Miner algorithm [19], the utility-list of any itemset can be obtained by intersecting the utility-lists of some of its subsets. Further, one phase algorithms prune the search space by utilizing the following property.

Property 3 (Pruning using the sum of iutil and rutil values [19])**.** Let there be an itemset X. Let the extensions of X be the itemsets obtained by appending an item y to X such that $y \succ i$, $\forall i \in X$. If the sum of *iutil* and *rutil* values in the utility-list of X is less than *minutil*, then X as well as all its transitive extensions are low-utility.

Proof. Let Y be the extension of X, $Y \supset X$, then $TidSet(Y) \subseteq TidSet(X)$. Let Y/X denote the set of all the items after X in Y. By Definition 1 we have $u(Y) = \sum_{T_d \in TidSet(Y)} u(Y, T_d)$, where $u(Y, T_d) = u(X, T_d) + u((Y/X), T_d) = u(X, T_d) + \sum_{y \in (Y/X)} u(y, T_d) \leq u(X, T_d) + \sum_{y \in (T_d - X)} u(y, T_d) = u(X, T_d) + ru(X, T_d)$. Therefore, $u(Y) \leq \sum_{T_d \in TidSet(Y)} u(X, T_d) + ru(X, T_d) \leq \sum_{T_d \in TidSet(X)} u(X, T_d) + ru(X, T_d)$. □

3. Related work

In this section, we briefly review studies related to high-utility itemset mining, closed high-utility itemset mining, and then we describe the differences between our method and the previous works.

3.1. Static high-utility itemset mining

The downward closure property in FIM does not hold with the utility measure in HUIM. Hence, to address this issue, Liu et al. [17] introduced the transaction-weighted utilization (TWU) measure as an upper-bound on the utility. Methods using TWU exploit the fact that TWU is anti-monotonic, making it suitable to reduce the search space in mining High Utility Items (HUIs), while ensuring that no HUI is missed. The basic assumption is that, if an itemset has a TWU lower than the *minutil* threshold, all its supersets can be ignored. Nevertheless, although this property is useful for reducing the search space, a problem is that the TWU is considered as a loose upper bound on the utility of itemsets. Hence, many more itemsets still need to be considered by algorithms relying on TWU to extract the set of HUIs. This, in turn, can result in long execution times and high memory usage.

Many algorithms to mine high utility patterns, including [17, 4, 7, 18], have been developed using the pruning strategy proposed by Liu et al. [17] to restrict the search space. Common to these methods are that they all are executed in two phases, with the first phase being used to identify candidate high-utility patterns, and the second phase to scan the database to compute the exact utility of all candidates, and filter out the patterns with low utility. The main issue with these algorithms is the high number of candidates that need to be considered to find the final set of results, due to a loose TWU upper bound as mentioned above. To partly address the issue with the two-phase algorithms, several approaches have been proposed to mine high-utility patterns in a single phase. Using a special utility-list structure [19] combined with an effective pruning strategy, one-phase algorithms have generally been shown to be more efficient than two-phase algorithms [20, 1, 21, 22, 2].

3.2. Dynamic high-utility itemsets mining

To cope with the challenges when working with dynamic databases, many methods have been introduced to mine HUIs in incremental databases [23, 4, 24, 5, 25, 26, 27, 28]. Unlike batch algorithms, incremental HUIM algorithms incrementally update and output HUIs, thus reducing the cost of discovering HUIs from scratch. These methods also extend the previous HUIM methods. Specifically, IUM [29] is the first algorithm for mining high temporal utility patterns from incremental transaction database which is based on the two-phase algorithm[17]. IHUP [4] proposed three tree structures (IHUP_{L}-tree, IHUP_{TF}-tree, and IHUP_{TWU}-tree) which were inspired from the FP-Growth algorithm [30] for incremental mining of HUIs when transactions are added. Although IHUP avoids drawbacks of the generate-and-test approach used by IUM, this method still needs expensive computational time to calculate actual HUIs from the set of candidates since IHUP generates a large number of candidates. PRE-HUI [31] is based on the pre-large concept [32] and the TWU model [17]. The pre-large concept is used to reduce the number of database scans that are required to update the results when new transactions are inserted. The original database is only rescanned when the number of inserted transactions is larger than the safety bound. However, PRE-HUI employs a level-wise approach to mine updated HUIs, and it suffers the same limitation as the tree-based methods. HUPID-Growth [5] introduced a new tree structure called HUPID-Tree, which is constructed through a single database scan. It also proposes a restructuring method with a new data structure called TIList (Tail-node Information List) to process incremental databases efficiently. In addition, HUPID-Growth introduces a strategy to reduce the overestimated utilities in conditional trees during the mining process. However, the performance of HUPID-Growth was not compared with its existing approaches such as IUM [29], and PRE-HUI [31].

As with HUIM methods, list-based methods without candidate generation have been proposed to mine HUIs from incremental databases. HUI-list-INS [33, 34] is the first algorithm of this category which is based on HUI-Miner algorithm [19]. It computes a TWU ascending order and constructs a global utility-list data structure based on this information through two database scans. Then, it mines high utility patterns from the global utility-lists without generating candidates. HUI-list-INS also employed the estimated utility co-occurrence structure [1] to speed up the incremental mining process. The experimental results showed that HUI-list-INS outperforms the previous methods. However, there is still room for improvement in the one-phase HUI-list-INS algorithm, such as developing more efficient pruning strategies. EIHI [24] reduces search space by reducing the number of local utility lists generated in the mining process. Moreover, it uses a trie-based structure named HUI-trie to insert and maintain information of HUIs, and to prune the search space during the updating process of utility-lists. However, this method requires additional operations such as creating new utility lists for new data, merging these lists into the utility-lists of the original database, and maintaining the rank order of single items according to the original database to maintain the HUI-trie. Moreover, inserting HUIs into the HUI-trie can be costly because the order of single items changes when new transactions are added, meanwhile the number of HUIs is large. Recently, a list-based method named LIHUP [26] was introduced which builds and updates its global lists by scanning the database only one time, meanwhile HUI-list-INS and EIHI require two database scans. However, LIHUP prunes the candidates by the remaining utility upper-bound [19], which is still loose and costly. And the experimental results have shown that LIHUP outperforms the tree-based methods, IHUP [4] and HUPID [5].

The most recent algorithm for incremental mining HUIs is PIHUP [27], which adopts the pre-large concept and improves the PRE-HUI algorithm. PIHUP introduces a new data structure called pattern tree. This tree is constructed by using mined large and pre-large patterns, where each node stores its actual utility

au and TWU values. The updated pattern tree is derived by using large, pre-large patterns stored in the pattern tree and the newly inserted data. PIHUP needs only one scan, and as a result, outperforms the PRE-HUI algorithm. However, this method still remains a candidate generation-and-test method, which can produce many useless candidates.

Hong et al. [35] introduced the concept of high average-utility itemset (HAUI), a concept that is slightly different from the traditional HUIM. Here, the average-utility of an itemset is calculated as the sum of item utilities divided by the length of itemset. Hence, the number of mined HAUIs is fewer than the number of HUIs under the same threshold. The problem of mining HAUIs from static and incremental databases has been studied extensively, and several efficient methods for addressing this problem have been proposed [25, 36].

In recent years, several algorithms have been proposed to mine HUIs in data streams, that are based on different assumptions with incremental HUIs mining. These algorithms assume that data arrive continuously at a very fast rate, with the amount of data being unknown or unbounded, and that it is impossible to store all the data permanently. The representative algorithms for mining HUIs in data streams include MHUI-BIT [37], MHUI-TID [37], SHU-Grow [38], and SHUPM [39]. MHUI-BIT and MHUI-TID are Apriori-based algorithms. They employ a level-wise generate-candidate-and-test approach to explore the search space of itemsets. The drawbacks of such an approach is that it requires to both perform numerous database scans and maintain a large number of candidates. SHU-Grow addresses this issue by employing a pattern growth approach. It utilizes the reducing global estimated utilities and reducing local estimated utilities techniques to decrease the overestimated utilities of itemsets. Moreover, SHU-Grow introduces a SHU-Tree structure to maintain information about the data and HUIs. Experimental results have shown that SHU-Grow outperforms the existing methods. However, SHU-GROW still requires a large amount of time to compute the actual utilities of candidate patterns in the second phase. Therefore, Yun et al. [39] recently proposed a new algorithm, called SHUPM, which does not generate candidate patterns. SHUPM [39] introduces a new list structure named SHUP-List to maintain the information of recent batches with a strategy that allows to efficiently update the remaining utilities in these lists. It also employs a new pruning technique to reduce the search space. Based on the results reported in [39], SHUPM is efficient in terms of runtime, memory usage, and scalability.

3.3. Static closed high-utility itemsets mining

As presented above, high utility itemset mining (HUIM) has been a major research, and many works have studied different areas of high utility mining [19, 20, 1, 21, 22, 2, 40]. However, the main drawback with HUIM is that the result sets of HUIs return by HUIM algorithms are often very large, which makes the process of analyzing these result sets a challenging task. To address this issue, many methods have been proposed to mine more concise, representative subsets of closed HUIs [7, 14, 16, 15]. These methods incorporate techniques from closed FIM with techniques used in HUIM in order to reduce the search space effectively, while ensuring that no CHUIs are missed. Closed$^+$ High Utility itemset Discovery (CHUD) [7] is the first algorithm for mining CHUIs. It is a two-phase and a depth-first search algorithm that extends the DCI_Closed algorithm [10], which is one of the fastest algorithms to mine frequent closed itemsets in a transaction database. Further, it adopts a data structure called transaction utility table (TU-Table) [17] to store the transaction utilities of all transactions. The TU-Table allows to efficiently compute the estimated utility of any itemset X, using its Tid set, without scanning the database. In addition, CHUD utilizes several efficient strategies to reduce the search space. Nevertheless, because CHUD is two-phase algorithm, it inherits the limitations of two-phase algorithms, in terms of performance degradation and memory consumption.

More recent algorithms have focused on mining CHUIs more efficiently as a one-phase approach [14, 16, 15]. An example of these algorithms is CHUI-Miner [14], which is also the first one-phase algorithm for mining of CHUIs. CHUI-Miner integrates techniques from closed itemset mining [11, 10] to discover closed patterns only. It utilizes a list structure called Efficient Utility list (EU-List) to store information about the utility of itemsets and adopts a divide-and-conquer approach to mine CHUIs without producing candidates. CHUI-Miner only prunes the search space using the aforementioned TWU measure [17] and the remaining utility upper-bounds [19]. Another approach is the EFIM-Closed (Closed EFficient high-utility Itemset Mining) algorithm [16], which is based on the constraint that all operations for each itemset in the search space should be performed in linear time and space. To achieve this, the algorithm proposed several efficient strategies to discover CHUIs, and effective techniques to reduce the cost of database scans.

Moreover, EFIM-Closed employs two upper-bounds on the utility of itemsets named sub-tree utility and local utility to effectively prune the search space. Although, EFIM-Closed is a highly efficient algorithm, it employs an expensive database sort operation to identify duplicate transactions, which, in turn, may degrade its performance when mining on large databases. Finally, CLS-Miner [15] was introduced to mine CHUIs more efficiently. Unlike Closed$^+$ High Utility itemset Discovery (CHUD), it is a one-phase algorithm relying on the utility-list structure to discover CHUIs. From this perspective, CLS-Miner is similar to CHUI-Miner, but there are some key differences. First, CLS-Miner applies search space pruning strategies, which are different from those of the CHUI-Miner. An important feature is that CLS-Miner's strategies can prune itemsets in the search space before their utility-lists are fully constructed, and thus greatly reduce the cost of mining CHUIs. Second, CLS-Miner introduces an efficient pre-check containing method to quickly determine if an itemset is a subset of another itemset. The authors used this method to optimize the operations of closure computations and subsumption checks typically performed by closed pattern mining algorithms. It is, however, important to note that these two operations are performed repeatedly in closed pattern mining algorithms. Hence, this pre-check method considerably reduces the time for discovering CHUIs of CLS-Miner.

3.4. Differences from previous works

Although closed high utility itemset mining has many applications, and the existing methods [7, 14, 16, 15] are efficient to mine CHUIs, they mainly assume that databases are static. There are several algorithms designed for maintaining HUIs in dynamic databases [4, 24, 33, 5, 26]. However, to the best of our knowledge, there does not exist any method for mining CHUIs in incrementally generated datasets. Meanwhile, databases are continuously growing in size in various real-life applications, and thus existing methods for static databases may no longer be suitable for processing or extracting useful information. That is, they do not perform well on dynamic databases. Motivated by this, in this work we introduce a new method for mining CHUIs from incremental databases in the literature.

The core novel ideas are the incremental utility-list structure and the maintenance mechanisms. Although using utility-list structure in an incremental way has been utilized in several existing methods, e.g., [33, 34, 26], our method is more advance and is different from these methods in several ways. First, the HUI-list-INS algorithm [33, 34] employs the traditional utility list structure, builds the utility lists of single item separately (DB.UL, the utility lists of the original database D, and db.UL, the utility lists of added transactions d). In contrast, with our utility-list structure, a utility-list has two parts, one part for the original database and another part for the added section. More importantly, our utility-list is built with only one database scan and remains valid whenever transactions are added; whereas, HUI-list-INS has to scan database two times to build its lists. Before the mining process, HUI-list-INS merges the utility lists with respect to the original database and the added section of the same single item. However, the two parts of our list are concatenated to form a single utility-list for the updated database after the mining process is finished, to prepare the algorithm for the next execution where new transactions are added. Second, LIHUP algorithm [26] is quite similar to the HUI-list-INS algorithm, but it only needs one database scan, and its list remains valid with new transactions being added. From this perspective, it is similar to our IncCHUI approach. However, LIHUP performs mining process like the previous approach, HUI-Miner[19], by using its utility lists, and does not apply any advanced pruning strategies. Third, in this work, we introduce a fast incremental utility-list construction procedure, while both HUI-list-INS and LIHUP use the old utility-list construction method, which is costly. Fourth, HUI-list-INS and LIHUP perform searching of HUIs in the whole database, while our method employs the utility-list part of the updated section to allow early pruning. This is because we can avoid exploring the itemsets that do not appear in the inserted transactions. Fifth and finally, we introduce an efficient hash-based approach to update or insert new CHUIs that are found during the mining process. Our approach is more efficient than the method that EIHI [24] employs, which requires maintaining the rank order of single items according to the original database to maintain the HUI-trie. Besides, inserting HUIs into the HUI-trie is costly because the order of single items changes when new transactions are added, whereas the number of HUIs is very large.

4. The IncCHUI method

The proposed IncCHUI algorithm is a single-phase approach that utilizes the incremental utility-list structure to mine CHUIs in incremental databases. IncCHUI is executed in three distinct stages. In the first

stage, it reads the original database or the newly added transactions, initializes the global lists of single items and closed hash table CHT if needed, then it sets up a total order on items. In stage 2, it sorts the global list according to the updated total order of items specified in the first stage, then performs a procedure to update this list to make it valid. In the third stage, a recursive search procedure is executed to efficiently mine CHUIs. The result CHUIs are stored in the closed table properly with its maintenance rule. The overall mining process of the proposed algorithm is shown in Fig. 2, meanwhile its components and structures are presented in detail in the following subsections.

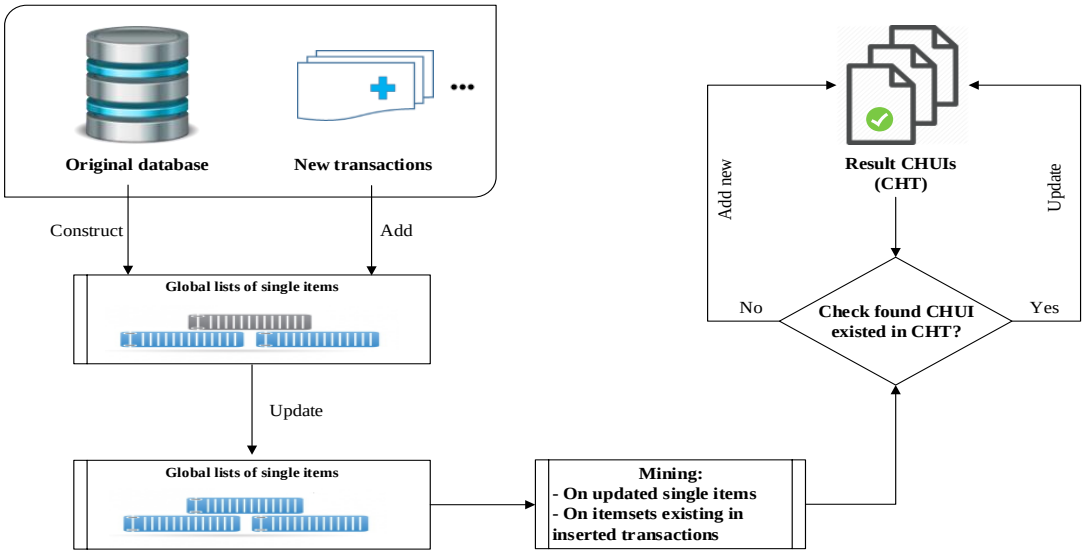


Figure 2: Overall process of the proposed method.

4.1. Incremental utility-list structure

To store crucial information of itemsets in the original database, as well as in the added transactions, we employ the traditional utility-list structure [19] in an incremental way. Its definition and properties are as follows.

Definition 8 (Incremental Utility-list Structure)**.** Let $\succ$ be a total order on items from I. Let TD be the set of transaction identifiers in the original database $\mathcal{D}$, and let TN be the set of transaction identifiers in a non empty set of transactions $\mathcal{N}$, which is appended to the original database to obtain the updated database $\mathcal{DN}$, $TDN = TD \cup TN$. The *incremental utility-list* of an itemset X in the database $\mathcal{DN}$ is denoted as $iUL(X)$, and it includes two traditional utility-lists storing information about X with respect to the original database $\mathcal{D}$ and the updated section $\mathcal{N}$, namely $ul_{\mathcal{D}}$ and $ul_{\mathcal{N}}$. Formally,

$iUL(X).ul_{\mathcal{D}} = \bigcup_{X \in TD_{tid}} (tid; u(X, TD_{tid}); \sum_{i \in X \wedge x \succ i} u(i, TD_{tid}))$, and
$iUL(X).ul_{\mathcal{N}} = \bigcup_{X \in TN_{tid'}} (tid'; u(X, TN_{tid'}); \sum_{i \in X \wedge x \succ i} u(i, TN_{tid'}))$.

Based on this definition, we have the following property to specify utility of one itemset regarding its incremental utility-list.

Property 4 (Sum of iutil values)**.** Let there be an itemset X. The utility of X, $u(X)$, on the updated database $\mathcal{DN}$ is equal to the sum of all the *iutil* values in its incremental utility-list $iUL(X)$, i.e., $u(X) = \sum iUL(X)ul_{\mathcal{D}}.iutil + \sum iUL(X)ul_{\mathcal{N}}.iutil$. If the sum is higher than or equal to the *minutil* threshold, it follows that X is a HUI. Otherwise, X is a low-utility itemset.

Example 5. Consider the example database presented in Fig. 1. The original database contains 5 transactions $(T_1, \ldots, T_5)$, then transactions T_6 and T_7 are added. The total order of items according to the ascending order of TWU values is $g \succ f \succ a \succ b \succ d \succ e \succ c$. Fig. 3 depicts the incremental utility-lists of single items for the example, where white parts represent tuples of the $ul_{\mathcal{D}}$ lists (the original transactions), and

grey parts represent tuples of the $ul_{\mathcal{N}}$ lists (the added transactions). In this figure, the utility-list of added transactions of g is empty ($iUL(g).ul_{\mathcal{N}} = \text{NULL}$), because the added transactions T_5 and T_6 do not contain item g. From this example, we introduce the following property to prune itemsets that are not in the added section.

g		
Tid	iutil	rutil
2	5	22
5	2	9

f		
Tid	iutil	rutil
3	5	25
6	2	14

a		
Tid	iutil	rutil
1	5	3
2	10	12
3	5	20
7	5	7

b		
Tid	iutil	rutil
3	10	10
4	8	12
5	4	5
6	4	10

d		
Tid	iutil	rutil
1	2	1
3	6	4
4	6	6
6	10	0
7	2	5

e		
Tid	iutil	rutil
2	6	6
3	3	1
4	3	3
5	3	2
7	3	2

c		
Tid	iutil	rutil
1	1	0
2	6	0
3	1	0
4	3	0
5	2	0
7	2	0

Figure 3: Incremental utility-lists of single items for the example database.

Property 5. For any itemset X, if the utility-list of X in $\mathcal{N}$, $iUL(X).ul_{\mathcal{N}}$, is empty, then the itemset X and all its extensions do not need to be explored.

This property is directly obtained from Property 1. Furthermore, from Property 3 [19], we have the following property to prune low-utility itemsets employing the increment utility-list structure.

Property 6 (Pruning using the increment utility-list)**.** Let $\succ$ be a total order on items from I. Let there be any itemset X, and the extensions of X be the itemsets obtained by appending an item y to X such that $y \succ i$, $\forall i \in X$. If the sum of *iutil* and *rutil* values in $iUL(X).ul_{\mathcal{D}}$ plus the sum of *iutil* and *rutil* values in $iUL(X).ul_{\mathcal{N}}$ is less than *minutil*, then X as well as all its transitive extensions are low-utility.

The incremental utility-list of any itemset xy can be obtained by intersecting the incremental utility-lists of its subsets $iUL(x)$ and $iUL(y)$ without scanning the database. The basic procedure for intersecting utility-lists was proposed in the HUI-Miner algorithm [19], and can be performed in linear time. This procedure is useful for constructing utility-lists, and it can be applied directly to utility-lists stored in the incremental utility-list structure. However, to construct the incremental utility-list of xy, we first intersect the utility-lists of x and y on updated section, $iUL(x).ul_{\mathcal{N}}$ and $iUL(y).ul_{\mathcal{N}}$, then we intersect the utility-lists of x and y on the original section, $iUL(x).ul_{\mathcal{D}}$ and $iUL(y).ul_{\mathcal{D}}$. Property 5 means that if the result of intersecting on the updated section is empty, then we avoid performing further intersection, and return a NULL result list. In conclusion, Property 5 enables early pruning, making it possible to provide a fast incremental utility-list construction procedure, as presented in Algorithm 1.

4.2. Maintenance mechanisms

In this section, we present in detail the maintenance mechanisms of the proposed method including how to maintain the incremental utility-lists of the global single items and the set of closed itemsets found so far.

4.2.1. Construction and update of the global list

To store crucial information of single items on both $\mathcal{D}$ and $\mathcal{N}$, we use the proposed incremental utility-list structure presented in Section 4.1. First, we construct a global data structure consisting of a set of incremental utility-lists, named *global list*, where each list stores information of one single item in the incremental database. However, different from the previous method by Liu and Qu [19], we scan the database only one time, and the *rutil* values are initialized as zero. When finished reading the transactions, we recalculate the *rutil* values after a global total order of single items is specified. The intuition behind these tasks is based on the following property.

Property 7. The *iutil* values in $ul_{\mathcal{D}}$ and $ul_{\mathcal{N}}$ of incremental utility-lists of items are not changed if the total order of items changed. However, the *rutil* values in these lists calculated based on the previous total order on items from I are invalid when inserting new transactions. Therefore, the *rutil* should be recomputed according to the change of the new total order.

Algorithm 1 Construct increment utility-list

Input: The increment utility-list of itemset Px, $iUL(Px)$, and the increment utility-list of itemset Py, $iUL(Py)$.
Output: The increment utility-list for new itemset Pxy, $iUL(Pxy)$.

1: $iUL(Pxy)$ = NULL
2: **for each** (tuple $exn \in iUL(Px).ul_{\mathcal{N}}$) **do**
3: Search tuple $eyn \in iUL(Py).ul_{\mathcal{N}}$ such that $eyn.tid = exn.tid$
4: **if** eyn is not NULL **then**
5: $exyn \longleftarrow (exn.tid;\ exn.iutil + eyn.iutil;\ eyn.rutil)$
6: $iUL(Pxy).ul_{\mathcal{N}} \longleftarrow iUL(Pxy).ul_{\mathcal{N}} \cup exyn$
7: **end if**
8: **end for**
9: **if** ($iUL(Pxy).ul_{\mathcal{N}}$ is empty) **then**
10: **return** NULL // by Property 5
11: **end if**
12: **for each** (tuple $exd \in iUL(Px).ul_{\mathcal{D}}$) **do**
13: Search tuple $eyd \in iUL(Py).ul_{\mathcal{D}}$ such that $eyd.tid = exd.tid$
14: **if** eyd is not NULL **then**
15: $exyd \longleftarrow (exd.tid;\ exd.iutil + eyd.iutil;\ eyd.rutil)$
16: $iUL(Pxy).ul_{\mathcal{D}} \longleftarrow iUL(Pxy).ul_{\mathcal{D}} \cup exyd$
17: **end if**
18: **end for**
19: **return** $iUL(Pxy)$

Proof. We can easily prove this property by referring to the definition of traditional utility-list [19] in Definition 7. Let $\succ$ be a total order on items from I, i.e., the ascending order of TWU values. According to Definition 7, utilities of items in the transaction stay the same even if the total order $\succ$ is changed. This is because the utility of each item is a multiplication of its purchase quantity (internal utility) and its profit (external utility) in a database, where the purchase quantity and profit are fixed values and do not depend on the total order. Meanwhile, each *rutil* value of an item in a tuple ($tid \in TDN$, $iutil \in \mathbb{R}$, $rutil \in \mathbb{R}$) is the sum of utilities of subsequent items after the item in a transaction. However, a set of subsequent items regarding the item is changed when the total order $\succ$ is updated. In addition, the total order $\succ$ of items varies when adding a non empty set of transactions $\mathcal{N}$ to $\mathcal{D}$, as the TWU values of single items may increase. □

Table 1: TWU values w.r.t the original database.

Item	a	b	c	d	e	f	g
TWU	65	61	96	58	88	30	38

Table 2: TWU values w.r.t the incremental database.

Item	a	b	c	d	e	f	g
TWU	77	77	108	86	100	46	38

Example 6. The original database $\mathcal{D}$ in Fig. 1 contains 5 transactions $T_1, \ldots, T_5$. The TWU values of single items in I are showed in Table 1. Hence, the total order of single items according to the ascending order of TWU values is $f \succ g \succ d \succ b \succ a \succ e \succ c$, and the global list of $\mathcal{D}$ is depicted in Fig. 4. After that, section $\mathcal{N}$ containing transactions T_6 = (b,2)(d,5)(f,2), and T_7 = (a,1)(c,2)(d,1)(e,1) is added to the original database. This makes the TWU values of single items change, for example TWU(f) increases from 30 to 46. The TWU values of single items are updated and presented in Table 2. Therefore, the previous total order of single items is no longer valid for the whole incremental database $\mathcal{DN}$, and changes to $g \succ f \succ a \succ b \succ d \succ e \succ c$. Thus, the utility-lists of single items for the original database are not valid, and should be recalculated properly according to the new total order.

What can be derived from this example is that after adding a set of non empty transactions $\mathcal{N}$ to the original database $\mathcal{D}$, we update the TWU information of items. The proposed algorithm utilizes the updated TWU to set up a new total order $\succ$ on items from I, i.e., TWU values ascending order. We then rearrange

f		
Tid	iutil	rutil
3	5	25

g		
Tid	iutil	rutil
2	5	22
5	2	9

d		
Tid	iutil	rutil
1	2	6
3	6	19
4	6	14

b		
Tid	iutil	rutil
3	10	9
4	8	6
5	4	5

a		
Tid	iutil	rutil
1	5	1
2	10	12
3	5	4

e		
Tid	iutil	rutil
2	6	6
3	3	1
4	3	3
5	3	2

c		
Tid	iutil	rutil
1	1	0
2	6	0
3	1	0
4	3	0
5	2	0

Figure 4: Incremental utility-lists of single items for the original database.

the global list according to this updated total order of items in I. After that, we traverse all incremental utility-lists in the global list starting from the one of item having the largest TWU value and updates the *rutil* values accordingly. We first update the *rutil* values of $ul_{\mathcal{N}}$, then that of $ul_{\mathcal{D}}$. While traversing, we use a temporary array to store and compute the sum of *iutil* values having the same $tid \in TDN$, the array index is the *tid* and the value store at each array entry is increased by that current *iutil* value. In other words, each array value stores the remaining utility of the corresponding transaction at current step. Before increasing this sum value, we assign the previous sum value to the *rutil* value of the tuple. When finishing the traversal, the temporary array will be deleted to save memory. Algorithm 2 shows the pseudocode of this restructure procedure. Note that, the global list for the original database is constructed when we consider performing the first time with $\mathcal{D}$ contains no transaction and adding transactions in $\mathcal{N}$. Finally, at the end of mining process, for each increment utility-list of single items in the global list, the utility-list part in $\mathcal{N}$, $ul_{\mathcal{N}}$, is concatenated to that in $\mathcal{D}$, $ul_{\mathcal{D}}$, to form a single utility-list for the updated database $\mathcal{DN}$. This is to prepare the algorithm for the next execution where the database $\mathcal{DN}$ will be updated.

Algorithm 2 Procedure Update Global List

Input: A set of increment utility-lists, GUL.

1: Initialize array RuA containing *rutil* values of items with its elements are zero
2: **for each** list $iUL(i') \in$ GUL while traversing from bottom to top **do**
3: **for each** tuple e of $iUL(i')$ **do**
4: $e.rutil$ = RuA[$e.tid$]
5: RuA[$e.tid$] = RuA[$e.tid$] + $e.iutil$
6: **end for**
7: **end for**
8: **delete** RuA

Example 7. Consider the global list of original database presented in Fig. 4. Table 2 presents the updated TWU values of single items when transactions T_6 and T_7 are added, and the new total order is $g \succ f \succ a \succ b \succ d \succ e \succ c$. Fig. 5 depicts the rearranged global list presenting in Fig. 4 according to this new total order, where the *rutil* values of the tuples for the added section are zero, and the *rutil* values of the tuples for the original section may be invalid according to the new total order and need recalculating. Then we traverse this global list and recalculate the *rutil* values as follows. Firstly, the traversal starts from the increment utility-list of item c, since TWU(c) is the largest, its *rutil* values are set to zero. The *iutil* values of this item are assigned to entries of the temporary array RuA with indexes are the transaction identifiers, i.e., RuA[1] = $u(c, T_1) = 1$, RuA[2] = $u(c, T_2) = 6$, etc. Fig. 5 also shows the results of this step. We next traverse the list of item e, which has five tuples with the Tids are 2, 3, 4, 5, and 7. The *rutil* values in the tuples of the list are assigned to the stored sums in RuA at the corresponding index in Tids, which respectively are 6, 1, 3, 2, and 2. In addition, the stored values increases by the *iutil* values in these tuples, i.e., RuA[2] = RuA[2] + $u(e, T_2) = 6 + 6 = 12$. Fig. 6 is the processing result for the incremental utility-lists of items c and e. Fig. 7 is the result after updating the lists of items c, e, and d. We apply the similar traversing and updating process for the remaining lists, and the final result is depicted in Fig. 3.

g		
Tid	iutil	rutil
2	5	22
5	2	9

f		
Tid	iutil	rutil
3	5	25
6	2	0

a		
Tid	iutil	rutil
1	5	1
2	10	12
3	5	4
7	5	0

b		
Tid	iutil	rutil
3	10	9
4	8	6
5	4	5
6	4	0

d		
Tid	iutil	rutil
1	2	6
3	6	19
4	6	14
6	10	0
7	2	0

e		
Tid	iutil	rutil
2	6	6
3	3	1
4	3	3
5	3	2
7	3	0

c		
Tid	iutil	rutil
1	1	0
2	6	0
3	1	0
4	3	0
5	2	0
7	2	0

Tid	1	2	3	4	5	6	7
RuA =	1	6	1	3	2	0	2

Figure 5: The reordered global list of Fig. 4 according to the new total order.

g		
Tid	iutil	rutil
2	5	22
5	2	9

f		
Tid	iutil	rutil
3	5	25
6	2	0

a		
Tid	iutil	rutil
1	5	1
2	10	12
3	5	4
7	5	0

b		
Tid	iutil	rutil
3	10	9
4	8	6
5	4	5
6	4	0

d		
Tid	iutil	rutil
1	2	6
3	6	19
4	6	14
6	10	0
7	2	0

e		
Tid	iutil	rutil
2	6	6
3	3	1
4	3	3
5	3	2
7	3	2

c		
Tid	iutil	rutil
1	1	0
2	6	0
3	1	0
4	3	0
5	2	0
7	2	0

Tid	1	2	3	4	5	6	7
RuA =	1	12	4	6	5	0	5

Figure 6: Update process for the list of e according to the new total order.

g		
Tid	iutil	rutil
2	5	22
5	2	9

f		
Tid	iutil	rutil
3	5	25
6	2	0

a		
Tid	iutil	rutil
1	5	1
2	10	12
3	5	4
7	5	0

b		
Tid	iutil	rutil
3	10	9
4	8	6
5	4	5
6	4	0

d		
Tid	iutil	rutil
1	2	1
3	6	4
4	6	6
6	10	0
7	2	5

e		
Tid	iutil	rutil
2	6	6
3	3	1
4	3	3
5	3	2
7	3	2

c		
Tid	iutil	rutil
1	1	0
2	6	0
3	1	0
4	3	0
5	2	0
7	2	0

Tid	1	2	3	4	5	6	7
RuA =	3	12	10	12	5	10	7

Figure 7: Update process for the list of d according to the new total order.

4.2.2. Maintaining the set of closed high-utility itemsets found so far

In this subsection, we first describe a property and then how the algorithm maintains the set of CHUIs when transactions are added.

Property 8. If itemset X is a CHUI in $\mathcal{D}$ before the addition of transactions in $\mathcal{N}$, then X remains a CHUI after transactions in $\mathcal{N}$ are added to the original database $\mathcal{D}$.

Proof. Let Y be a superset of X, i.e., $\forall Y, X \subset Y$, the support of X is thus greater than the support of Y, and the utility of X is no less than the *minutil* threshold because X is a closed high-utility itemset. If $X \notin \mathcal{N}$, then the support and the utility of X are not changed by Property 1, hence X is still a CHUI after adding transactions in $\mathcal{N}$ to the original database $\mathcal{D}$. Otherwise, if $X \in \mathcal{N}$, then the support and the utility of X calculated by Definition 3 will increase as a result of adding transactions in $\mathcal{N}$. In this case, the support of X still exceeds the support of its superset Y, and its utility is greater than the *minutil* threshold. Therefore, X remains a CHUI affter transactions in $\mathcal{N}$ are added to the original database $\mathcal{D}$. □

From Property 8, it can be derived that the set of closed high-utility itemsets $\mathcal{CH}'$ found in the updated database $\mathcal{DN}$ is always a superset of the set of closed high-utility itemsets $\mathcal{CH}$ found in the original database $\mathcal{D}$. Moreover, the utilities of itemsets in $\mathcal{CH}$ may only increase or stay the same in $\mathcal{CH}'$ as a result of the insertion of the new transactions, as well as its support. It is thus important to have a mechanism for storing itemsets in $\mathcal{CH}$ that be able to quickly update their utility and support later, when adding transactions. Hence, we store the closed itemsets found so far in a hash table, called the closed table CHT. The key of the hash table element is the actual itemset, and the value is its utility and support. Whenever a CHUI is found during the mining process on the added part, we first check whether it already was in the closed table CHT. If yes, we update its utility and support. Otherwise, it is a new one and it is put into the closed table CHT. When the algorithm terminates, all the CHUIs found so far are in the closed table CHT. If requested by a user, the algorithm returns this set of result CHUIs.

4.3. The IncCHUI algorithm

Algorithm 3 shows the details of our algorithm. The main steps of IncCHUI are the followings. First, IncCHUI initializes the global list GUL and the closed table CHT if it is the first time. The proposed algorithm scans each transaction T in the original database $\mathcal{D}$ or the new inserted $\mathcal{N}$ only one time to build the corresponding global increment utility-lists GUL (lines 3-11). For each item $i \in T$, if there does not exist its list in the global list GUL, the algorithm creates its list. Then, the algorithm adds a tuple regarding the transaction T to the list of item i (lines 8-9), where its *rutil* is zero, as well as calculates the TWU value of single item i. After reading all the transactions, IncCHUI establishes a total order $\succ$ on items, which is the order of ascending TWU values (line 14). Second, IncCHUI sorts the global list GUL according to the total order $\succ$ (line 15). Then, it calls the Update Global List procedure (Algorithm 2) to adjust all the *rutil* values of items (line 16). Third, if there is a mining request from the user, the Search-CHUI procedure (Algorithm 4) is performed on the set of potential single items (items appearing in added transactions and having TWU no less than *minutil*) to recursively search for closed high utility itemsets (line 19). Then, IncCHUI outputs the result in CHT (line 20). Forth and finally, IncCHUI merges the utility-lists in $\mathcal{N}$ into utility-lists in $\mathcal{D}$ of the increment utility-list of single items to form a single utility-list for $\mathcal{DN}$, preparing for the next execution where we add transactions to the database $\mathcal{DN}$ (lines 21-24).

As for searching of CHUIs, Algorithm 4 presents the details of our Search-CHUI procedure. This procedure takes the current itemset to be extended P, the two sets of items *PreSet(P)* and *PostSet(P)* as input parameters. The procedure computes all the closed high utility itemsets that strictly contain P by analyzing all the valid closed itemsets that are obtained by extending P with the items in its *PostSet*. The main steps of the Search-CHUI procedure are the followings. For each item x of $PostSet(P)$, the procedure creates an itemset $Px = P \cup x$, and builds its increment utility-list $iUL(Px)$ using Algorithm 1 (lines 2-3). If the sum of the *iutil* and *rutil* values in this list is no less than *minutil*, then the extensions of Px will be explored (line 4). Before exploring these extensions, the procedure IsSubsumedCheck(Px, $PreSet(P)$) is called to check whether Px is included in previously found closed itemsets. If yes, then the supersets of Px do not need to be explored (line 5). Otherwise, the search procedure tries to merge Px with each item $y \in PostSet(P)$, such that $y \succ x$ to form a larger itemset Pxy. The *postSetInner* variable is initialized to an empty set (line 6). Before constructing the utility-list of Pxy, the algorithm checks if item y appears in added section $\mathcal{N}$ (line 8). If this is true, then the procedure checks if y belongs to the closure of Px. If this condition is satisfied, the increment utility-list of Pxy is constructed (line 13). Next, if the sum of the *iutil* and *rutil* values in the $iUL(Pxy)$ is less than *minutil*, then this means that Pxy and its extensions are low utility itemsets. Hence, the search procedure stops adding items to Pxy to not generate its extensions (lines 14-16). Otherwise, y is added to *postSetInner* (line 18). If Pxy is, on the other hand, a potential CHUI and its utility is no less than *minutil*, the next step is to check whether Pxy has been discovered before in the closed table CHT. If this is the case, then the algorithm updates its utility and support. Otherwise, the algorithm puts it into CHT (lines 21-27). The Search-CHUI procedure is then recursively called to continue exploring the search space with Pxy and its two corresponding sets (line 29). Lastly, the item x is added to *PreSet(P)* (line 30). When the Search-CHUI algorithm terminates, all the CHUIs in the database have been obtained and enumerated.

The third stage is the most expensive stage. However, IncCHUI inherits the efficient search space browsing and closure computation techniques of DCI_CLOSED algorithm [10], which is, as already mentioned,

Algorithm 3 IncCHUI

Input: The original database $\mathcal{D}$, an incremented transaction data $\mathcal{N}$, and the *minutil* threshold.
Output: A complete set of closed high-utility itemsets.

```
1: Global variables: a global set of increment utility-list GUL, a closed high-utility itemset table CHT
2: Initialize GUL ⟵ ∅, CHT ⟵ ∅ for the first time
3: for each transaction T ∈ D or N {only scan one time} do
4:    for each item i ∈ T do
5:       if GUL.iUL(i) = NULL then
6:          Create iUL(i) in GUL
7:       end if
8:       Create a tuple for T in GUL.iUL(i) accordingly D or N
9:       Set iutil and rutil of the tuple to u(i,T) and zero
10:      Update TWU(i)
11:   end for
12: end for
13: Let I be the list of single items
14: Let ≻ be the ascending order of TWU values on items in I
15: Sort the increment utility-lists in GUL according to ≻
16: Call Update Global List(GUL) // Algorithm 2
17: if user request mining then
18:   Let I* be the list of items being updated and having TWU values no less than minutil// by Properties 1-2
19:   Call Search-CHUI (∅, ∅, the increment utility-list of items ∈ I*) //Algorithm 4
20:   Output result itemsets in CHT
21:   for each iUL ∈ GUL do
22:      Merge iUL.ul_N to iUL.ul_D
23:      iUL.ul_N = NULL
24:   end for
25: end if
```

one of the fastest algorithms for mining frequent closed itemsets, along with the proposed techniques presented above. It is here important to note that state-of-the-art algorithms, such as CHUI-Miner [14] and CLS-Miner [15], also adopt the techniques of DCI_CLOSED. Nevertheless, in contrast to DCI_CLOSED, IncCHUI prunes candidates that are not in the inserted transactions (Property 5), or low-utility (Property 6). Moreover, IncCHUI computes closure of itemsets directly. Hence, it avoids calculating the utility lists of low-utility or non-closed itemsets.

Proof of correctness and completness. First, using the *PostSet* set guarantees that the complete set of potential closed itemsets will be obtained, meanwhile using the *PreSet* set guarantees that all duplicate closed itemsets will be pruned by the procedure IsSubsumedCheck. For this reason, the Search-CHUI procedure can be said to be correct and complete in terms of finding closed itemsets and eliminating all non-closed itemsets [10]. Second, IncCHUI employs the incremental utility-list structure to maintain crucial information of items on both original database and updated section. Since Property 4 is correct with respect to calculating the utility of itemsets, IncCHUI can correctly calculate the utility of itemsets and identify only the high-utility itemsets. It is here important to note that the Update Global List procedure guarantees that the global list of itemsets is always valid, also when new transactions are inserted. Third and finally, the basic pruning strategies used in the proposed algorithm are Properties 5, and 6. These properties have been proven earlier to be correct with respect to prune only not-updated and low-utility candidates. In conclusion, we can conclude that the proposed IncCHUI algorithm is correct and complete for incremental mining all closed high-utility itemsets.

Example 8. This example shows how IncCHUI works in the incremented database $\mathcal{DN}$ of Fig. 1. At the beginning, the original database $\mathcal{D}$ contains five transactions, and the global list is constructed as shown in Fig. 4. Suppose that *minutil* is 30. According to Example 2, the set of CHUIs in $\mathcal{D}$ is $\mathcal{CH} = \{ace{:}31,$

Algorithm 4 Search-CHUI

Input : An itemset P, a set of pre-extensions of P ($PreSet(P)$), and a set of post-extensions of P ($PostSet(P)$).

```
1: for each item x ∈ PostSet(P) do
2:   Px ⟵ P ∪ x
3:   Construct iUL(Px) // Algorithm 1
4:   if iUL(Px) is not NULL and Px is potential high-utility {by Property 6} then
5:     if IsSubsumedCheck(Px, PreSet(P)) = False then
6:       postSetInner ⟵ ∅
7:       for each itemset y ∈ PostSet(P) and y ≻ x do
8:         if iUL(y).ul_N is NULL then
9:           Continue //by Property 5
10:        end if
11:        if TidSet(Px) ⊆ TidSet(y) then
12:          Pxy ⟵ Px ∪ y
13:          Construct iUL(Pxy) //Algorithm 1
14:          if iUL(Pxy) = NULL or Pxy is low-utility then
15:            Break
16:          end if
17:        else
18:          postSetInner ⟵ postSetInner ∪ y
19:        end if
20:      end for
21:      if Pxy is high-utility then
22:        if Pxy ∈ CHT then
23:          Update the support and utility of Pxy
24:        else
25:          Put Pxy into CHT
26:        end if
27:      end if
28:      PreSet(Pxy)⟵ PreSet(P), PostSet(Pxy) ⟵ postSetInner
29:      Call Search-CHUI(Pxy, PreSet(Pxy), PostSet(Pxy))
30:      PreSet(P)⟵ PreSet(P) ∪ x
31:    end if
32:  end if
33: end for

Function IsSubsumedCheck(Y, PreSet(Y))
34: for each (item J ∈ PreSet(Y)) do
35:   if (TidSet(Y) ⊆ TidSet(J)) then
36:     return True
37:   end if
38: end for
39: return False
```

bce:37, *bcde*:40, and *abcdef*:30}. Then, section $\mathcal{N}$ containing transactions T_6 and T_7 is added. IncCHUI performs the following steps to discover the set of CHUIs $\mathcal{CH}'$ in the $\mathcal{DN}$ database, where the closed table CHT contains itemsets *ace*, *bce*, *bcde*, and *abcdef*, and the global list of single items is depicted in Fig. 4.

Step 1. IncCHUI scans the two added transactions one time, then restructures the global list. The result global list of single items for the $\mathcal{DN}$ is shown in Fig. 3.

Step 2. The list of single promising items is $I^* = \{f, a, b, d, e, c\}$. Item g is eliminated since it is not in the

added transactions (by the proposed Property 1).

Step 3. The Search-CHUI procedure begins its recursive search for CHUIs first using item *f*, with parameters $PostSet = \{a, b, d, e, c\}$, and *PreSet* is still empty. This procedure finds CHUIs that are supersets of f by trying to append items from *PostSet* to *f*. Extensions of f are potential CHUIs since the sum *iutil* and *rutil* values of f is $(5 + 22 + 2 + 9) = 38$ (Fig. 3), which is no less than $minutil = 30$, and f is not subsumed by any items because the *PreSet* is empty. Hence, items from *PostSet* are appended to f to generate the closure of *f*. Before adding, pruning conditions are checked. Item a is first considered adding to generate itemset *fa*. $iUL(a).ul_{\mathcal{N}}$ is not empty, but $TidSet(f)= \{3, 6\} \nsubseteq TidSet(a)=\{1, 2, 3, 7\}$. Therefore, a is put into temporary variable *postSetInner*, which will be used in the inner recursive call later. The algorithm considers appending b to f to create the itemset *fb*. Its incremental utility-list is constructed by Algorithm 1, and $iUL(fb) = \{(T_3, 15, 15), (T_6, 6, 10)\}$. This itemset *fb* is not pruned, and it is a potential HUIs by Property 4, since $(15 + 15 + 6 + 10) = 46 > minutil = 30$. IncCHUI next considers appending d, e, c to *fb* in the same way. This results in the closed high utility itemset *fbd*, having the utility-list $iUL(fbd) = \{(T_3, 21, 9), (T_6, 16, 0)\}$. *fbd* is a new CHUI, because its utility (which is 37) is bigger than *minutil*, and *fbd* is not in the closed table CHT. IncCHUI puts *fbd* into the CHT. On the other hands, items e and c are inserted to *postSetInner* since $TidSet(fb) \nsubseteq TidSet(e)$, and $TidSet(fb) \nsubseteq TidSet(c)$.

Next, the search procedure enters the inner recursive to search for CHUIs which are supersets of *fbd* by extending it with a, e, and c (in $PostSet(fbd)= postSetInner$), and its preset is empty. However, when adding a to *fbd* in order to form itemset *fbda*, $iUL(fbda) =$ NULL calculated by Algorithm 1, since $iUL(fbda).ul_{\mathcal{N}} = \varnothing$. Therefore, *fbda* is pruned early by our Property 5. Similarly, *fbde* and *fbdc* are also pruned early. The inner recursively search finishes without finding any CHUIs. Finally, f is added to the set *PreSet*, and the loop with item f ends.

Step 4. IncCHUI searches for CHUIs starting with item a. This process is similar to Step 3, with $PreSet = \{f\}$ and $PostSet = \{b, d, e, c\}$. And IncCHUI finds two closed itemsets, the first is itemset ac with $iUL(ac) = \{(T_1, 6, 2), (T_2, 16, 6), (T_3, 6, 19), (T_7, 7, 5)\}$, and the second is itemset *ace* with $iUL(ace) = \{(T_2, 22, 0), (T_3, 9, 16), (T_7, 10, 2)\}$. ac is the new one and IncCHUI puts it into the CHT, while *ace* exists in the CHT, IncCHUI thus updates its utility and support to 41 and 3, respectively.

Step 5. The remaining items are processed in the same way. Finally, the set of closed high utility itemsets in $\mathcal{DN}$ is $\mathcal{CH}'$={*ace*:41, *bce*:37, *bcde*:40, *abcdef*:30, *bdf*:37, *bd*:44, *ac*:35, and *ce*:32}, where the number besides each itemset indicates its utility. At the end, IncCHUI concatenates the lists in $\mathcal{N}$ into $\mathcal{D}$ of the global list preparing for the next update.

4.4. Complexity analysis

In this section, we analyse the complexity of our method IncCHUI by first considering its two required key operations: global data structure construction/update, and pattern searching. Then, we discuss the worst-case complexity of IncCHUI, and compare with the complexity of the state-of-the-art CHUI-Miner, CLS-Miner, and EFIM-Closed algorithms.

1. **Global data structure construction/update**. Let $n_{\mathcal{D}}$ and $n_{\mathcal{N}}$ be the number of transactions in the original database $\mathcal{D}$ and the added part $\mathcal{N}$, respectively. Let w be the average transaction length, and m be the number of distinct items. The time required to read all the transactions in the original or the added section is $\mathcal{O}(n_{\mathcal{D}} \times w)$ or $\mathcal{O}(n_{\mathcal{N}} \times w)$, respectively. Building the initial incremental utility-list, and calculating the TWU of single items are performed within this reading step. Sorting the TWU values or the utility-list of single items requires $\mathcal{O}(mlog(m))$, in terms of time. Update Global List procedure takes $\mathcal{O}(m \times w)$ time. Hence, the time complexity of this phase is $\mathcal{O}(n_{\mathcal{D}} \times w + 2mlog(m) + m \times w)$ or $\mathcal{O}(n_{\mathcal{N}} \times w + 2mlog(m) + m \times w)$, which respectively is roughly $\mathcal{O}((n_{\mathcal{D}} + m) \times w)$ or $\mathcal{O}((n_{\mathcal{N}} + m) \times w)$, since the sort is only performed once time.

2. **Pattern searching**. This is the major operation performed by IncCHUI to discovery all CHUIs by recursively applying the Search-CHUI procedure. The complexity of this procedure is proportional to the number of times that the *IsSubsumedCheck* procedure is called, which is proportional to the

number of itemsets in the search space that are not pruned by the algorithm. In the worst case, no itemsets are pruned by the pruning properties, and the added transactions involve all items. Then the algorithm must consider $2^m - 1$ itemsets in the search space. Thus, the worst-case complexity is $\mathcal{O}(2^m - 1)$.

In conclusion, the overall worst case time complexity of the proposed algorithm IncCHUI (Algorithm 3) is $\mathcal{O}(2^m - 1)$, which is the same as CHUI-Miner [14] and CLS-Miner [15]. The time complexity of IncCHUI is roughly linear with the number of patterns that it visits in the search space. EFIM-Closed [16] is, on the other hand, different from InCHUI because of the following EFIM-Closed operations: (1) sorting transaction database in $\mathcal{O}(nwlog(nw))$, where $n = n_{\mathcal{D}}+n_{\mathcal{N}}$, (2) for each primary itemset α encountered during the depth-first search, EFIM-Closed performs database projection, transaction merging, backward/forward extension checking and upper-bound calculation. These three tasks are each carried out in $\mathcal{O}(nw)$. Thus, the time complexity of EFIM-Closed is also proportional with the number of itemsets in the search space.

Remark. The number of patterns in the search space is determined by the effectiveness of the pruning strategies that the algorithms employ. The search space is also much smaller than $2^m - 1$ itemsets since not all items co-occur in a database, or the added transactions may not involve all single items. Furthermore, IncCHUI is designed in a way such that it only needs to scan the database once. In contrast, the other algorithms scan the database twice and they must re-scan the whole database each time transactions are inserted, since they run in batch mode. This would be time consuming, especially if the database is large. In the experimental evaluation in the next section, we demonstrate that IncCHUI is more efficient than CHUI-Miner [14], CLS-Miner [15], and EFIM-Closed [16] algorithms.

5. Performance study

This section presents an extensive experimental evaluation to assess the efficiency of IncCHUI algorithm, including its performance when varying (*i*) *minutil* value, (*ii*) the number of transactions added (insertion rate), and (*iii*) databases' size (scalability tests). Because IncCHUI is the first algorithm for incremental mining CHUIs, the only way to evaluate its performance against the state-of-the-art algorithms for mining CHUIs is to compare it with the algorithms used in static databases and run in batch mode, which are CHUI-Miner [14], CLS-Miner [15], and EFIM-Closed [16]. CHUI-Miner and CLS-Miner employ the traditional utility-list structure, with the same exploring technique as the IncCHUI algorithm; whereas EFIM-Closed utilizes the database projection and transaction merging techniques.

5.1. Experiment setup

All the algorithms were implemented in Java, where CHUI-Miner[1] and EFIM-Closed[2] were obtained from the authors' pages. We carried out the experiments on a computer equipped with a 64 bit Core i5 2.4 GHz Intel Processor, 4GB of main memory, and running Windows 7 as operating system. We performed experiments on both real-life and synthetic datasets having various characteristics. Table 3 presents characteristics of these datasets, where #Trans, #Items, and #Avg indicate the number of transactions, the number of distinct items and the average transaction length, respectively. The Connect, Mushroom and Retail datasets were obtained from the FIMI Repository[3]. The Foodmart dataset is the Microsoft FoodMart 2000 database[4], and ChainStore[5] was obtained from the NUMineBench software distribution. OnlineRetail[6] is a recently transactional dataset which contains all the transactions occurring between 01/12/2010 and 09/12/2011 from a UK-based and registered non-store online retail. The company mainly sells unique all-occasion gifts. Lastly, T10I4D100K is a synthetic dataset generated using the IBM Quest dataset generator [41].

[1] http://bigdatalab.cs.nctu.edu.tw/software.php
[2] http://www.philippe-fournier-viger.com/spmf
[3] http://fimi.cs.helsinki.fi/data
[4] http://bigdatalab.cs.nctu.edu.tw/software.php
[5] http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html
[6] http://archive.ics.uci.edu/ml/datasets/online+retail

We selected these datasets because they include both dense, sparse, and large datasets, and thus represent well the main types of data seen in real-life applications. These datasets are also the most common used benchmark datasets in the high utility pattern mining literature. All datasets except Foodmart, OnlineRetail, and ChainStore do not include internal (purchase quantity) and external (profit unit) utility values. Thus, for these datasets, the internal and external utilities have been respectively generated randomly in the [1, 5] and [1, 10] intervals using a log-normal distribution, as in previous works [14, 16, 15].

Table 3: Details of the datasets.

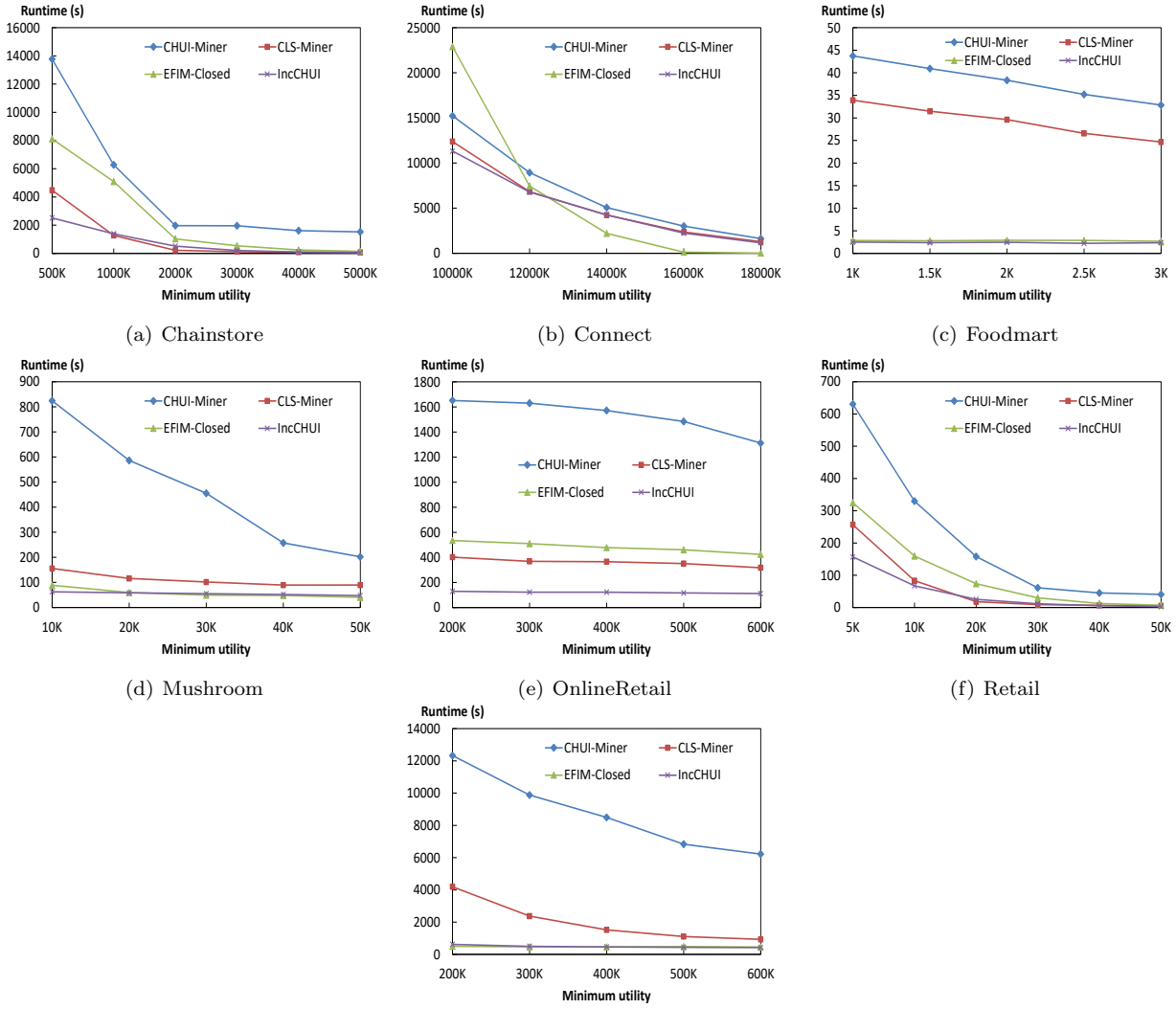
Dataset	#Trans	#Items	#Avg
Chainstore	1,112,949	46,086	7.3
Connect	67,557	129	43.0
Foodmart	4141	1559	4.4
Mushroom	8124	119	23.0
OnlineRetail	541,909	2603	4.37
Retail	88,162	16,470	10.3
T10I4D100K	100,000	870	10.1

5.2. Influence of the minutil threshold

We first performed experiment to evaluate the proposed algorithm under various *minutil* threshold values with a fixed number of inserted transactions (equivalently is the insertion ratio). Since these datasets have different number of transactions, and choosing a wide range of insertion ratios may provide a broader view on the performance of the algorithms, we adopted several insertion ratio values. In particular, the insertion ratio on Chainstore, Connect, Retail, and OnlineRetail is 10%, while for Foodmart and Mushroom it is 5%, and on the remaining synthetic dataset it is 1%. For example, the dataset T10I4D100K contains 100,000 transactions, and its insertion ratio is set as 1%. Hence, the compared algorithms are applied begin at the first 1000 transactions, increasing in increments of $100{,}000 \times 1\% = 1000$ transactions, to the whole dataset. In other words, the compared algorithms are applied 100 times on this dataset, where the input data size gradually increase from 1000 to 100,000 transactions with the step size is 1000 transactions. We ran all the compared algorithms on each of the datasets while gradually decreasing the *minutil* threshold, until a clear winner was observed. The execution times include the total time for reading the input database, discovering the patterns, and writing results to an output file. For each dataset, we also recorded the peak memory consumed by the algorithms when processing each incremented data part. For example, with the T10I4D100K dataset and the insertion rate equals to 1% presented above, we record the peak memory usages of the algorithms on 100 times running. Then we select and report the maximum values among those memory usages. Fig. 8 presents the accumulated run time comparison, while Table 4 compares the maximum memory consumption.

In Fig. 8, we can see that CHUI-Miner has the largest time consumption on all datasets except Connect, where EFIM-Closed consumes the largest time when the *minutil* value is smallest on this dataset. On the real datasets, InCHUI is the fastest, and there is a big gap between the execution times when the *minutil* value is small. In particular, on the Connect dataset, EFIM-Closed is faster than the remaining algorithms when the *minutil* value is large, however, it requires a big increasing amount of execution time when the *minutil* value is small. CLS-Miner is slightly slower than the proposed method. On the synthetic dataset T10I4D100K, IncCHUI is faster than both CHUI-Miner and CLS-Miner. It is also faster than EFIM-Closed when *minutil* value is larger than 300K. Furthermore, the run time of our method is almost the same when decreasing *minutil* threshold value on Foodmart, Mushroom, OnlineRetail, and T10I4D100K datasets. The reason can be explained as follows. Although CHUI-Miner, CLS-Miner and our algorithm, IncCHUI, employ similar utility-list structures, both CHUI-Miner and CLS-Miner use the traditional utility-list [19] requiring to scan the database two times to construct utility-lists of single items. This also applies to EFIM-Closed, even though it applies different techniques to reduce the cost of database scans [16]. As discussed in Section 3, these techniques require an expensive database sort operation to identify duplicate transactions, which would, in turn, degrade its performance on large datasets. In contrast, IncCHUI only scans the database one time.

Further, recall that the algorithms used in our comparative study were performed in batch mode. Hence, it is necessary to re-scan the database each time transactions are inserted into the original database. This, in itself, contributes to performance degradation. Moreover, the way the three previous algorithms build their required structures leads to mining on the incremental database from scratch, which is inefficient. On the other hand, IncCHUI efficiently maintains the incremental utility-lists of single items and the results found so far, and only performs searching on the updated data part. Overall, when varying the *minutil* value, our method is up to 24, 13, 4 times faster than CHUI-Miner, CLS-Miner, and EFIM-Closed, respectively.



(a) Chainstore (b) Connect (c) Foodmart

(d) Mushroom (e) OnlineRetail (f) Retail

(g) T10I4D100K

Figure 8: Accumulated runtime comparison when varying utility threshold.

Table 4 compares the peak memory usages of the four algorithms when the *minutil* threshold is set to the smallest values used in the previous experiment. The minimum usage values are typeset boldface. All the memory measurements were done using the standard Java API. As can be observed in the table, the memory consumption of the algorithms varies a lot. Our proposed method requires the least memory on five out of seven datasets. EFIM-Closed is more memory efficient on Foodmart dataset than the other algorithms, since this algorithm generates projected databases that are often very small in size, due to transaction merging, especially on small, short transaction datasets. CLS-Miner, on the other hand, needs to store other structures in memory for its pruning strategies, such as the Estimated Utility Co-occurrence Structure and the coverage

of items. Hence, it consumes more memory than the other algorithms, especially on the ChainStore dataset.

Table 4: Comparison of maximum memory consumption (MB) when varying utility threshold.

Dataset	CHUI-Miner	CLS-Miner	EFIM-Closed	IncCHUI
Chainstore	987.62	1949.00	**677.21**	851.33
Connect	368.57	768.4	702.38	**355.82**
Foodmart	156.5	207.23	**21.78**	59.7
Mushroom	279.48	260.36	200.40	**39.12**
OnlineRetail	780.26	661.10	653.48	**442.66**
Retail	704.63	719.23	594.04	**141.81**
T10I4D100K	712.38	637.35	316.42	**61.96**

To verify whether the IncCHUI algorithm obtains all the CHUIs, we kept record of the mining results produced by the four algorithms. All the algorithms gave the same results with respect to the same minimum utility threshold values and the same transactions. Moreover, we compared the number of CHUIs with the number of HUIs obtained by our previous method [2]. Tables 5 and 6 show the results at different minimum utility threshold and database size on the Mushroom and Foodmart datasets, respectively. In these tables, using the representation of CHUI produces a huge reduction in the number of result patterns, especially when the *minutil* threshold value is small.

Table 5: Number of extracted patterns on Mushroom.

Database size (%)	Minimum utility threshold									
	10K		20K		30K		40K		50K	
	#HUIs	#CHUIs	#HUIs	#CHUIs	#HUIs	#CHUIs	#HUIs	#CHUIs	#HUIs	#CHUIs
20	22390646	7996	3805495	2916	930745	1623	399437	807	147963	598
40	83178297	20527	21473415	9595	8646214	5487	4230218	3563	2336987	2447
60	141184897	37676	40017244	20777	17181448	13422	8818875	9668	5178501	7233
80	175259635	60088	48512583	34407	20349622	22773	10481875	16047	5983649	12038
100	229433266	78018	63423965	45891	26268454	30736	13327479	22034	7502602	16617

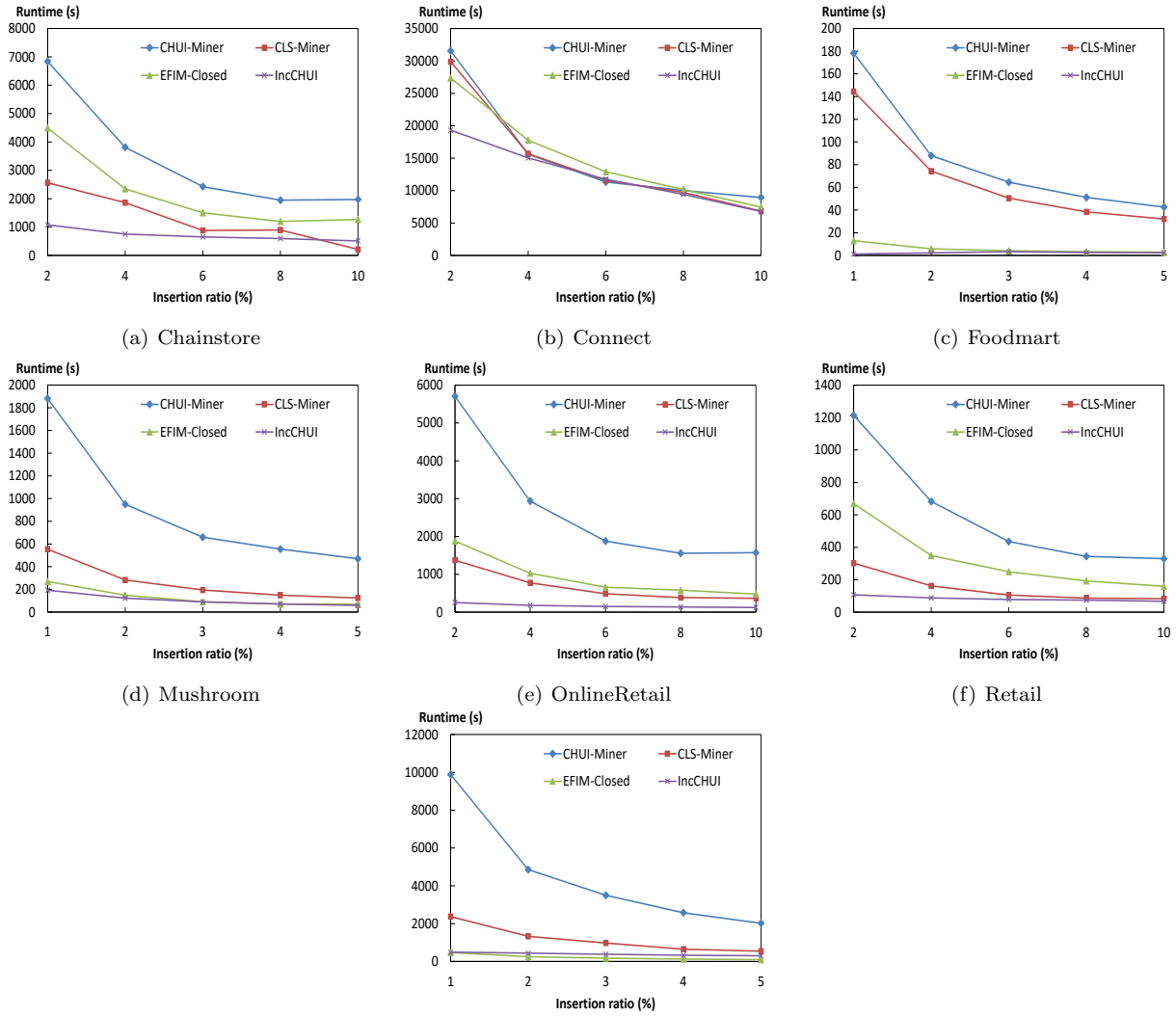
Table 6: Number of extracted patterns on Foodmart.

Database size (%)	Minimum utility threshold									
	1K		1.5K		2K		2.5K		3K	
	#HUIs	#CHUIs	#HUIs	#CHUIs	#HUIs	#CHUIs	#HUIs	#CHUIs	#HUIs	#CHUIs
20	40715	1642	33735	1319	25405	1028	17533	781	11095	583
40	88711	2970	76940	2630	62016	2200	47174	1821	34270	1471
60	125866	4105	108646	3756	86649	3279	64700	2791	45763	2306
80	182380	5280	159870	4883	130318	4294	100392	3691	74066	3099
100	219012	6454	191173	5986	154670	5273	117592	4552	85034	3804

5.3. Influence of the insertion ratio

We further performed experiments to evaluate the efficiency of the proposed IncCHUI algorithm when varying the number of transactions inserted (insertion ratio), while the *minutil* threshold values are fixed. Specifically, the *minutil* values were set to 2000K, 12000K, 1K, 20K, 400K, 10K, and 300K on Chainstore, Connect, Foodmart, Mushroom, OnlineRetail, Retail, and T10I4D100K, respectively. We ran all the compared algorithms on each of the datasets while gradually decreasing/increasing the insertion rate until a clear winner was observed.

Fig. 9 shows the accumulated runtime comparison for this evaluation. In this figure, we can observe that the proposed algorithm IncCHUI outperforms the other algorithms under various insertion ratios. The execution time of IncCHUI almost is constant when varying the number of added transactions on the Foodmart, OnlineRetail, Retail, and T10I4D100K datasets. On the remaining datasets, the execution time of IncCHUI increases slowly when decreasing insertion ratio. Meanwhile, the execution times of other compared algorithms largely change when the insertion ratio is smaller since they were run in batch mode. CHUI-Miner has the largest time consumption on all the datasets. On the synthetic dataset T10I4D100K, the execution time of IncCHUI is almost constant and more stable than that of EFIM-Closed. The three baseline algorithms have almost the same performance on Connect dataset, but IncCHUI is better than all three algorithms on this dataset. In general, when varying the number of inserted transactions, our method is up to 145, 117, 10 times faster than CHUI-Miner, CLS-Miner, and EFIM-Closed, respectively.



(a) Chainstore (b) Connect (c) Foodmart

(d) Mushroom (e) OnlineRetail (f) Retail

(g) T10I4D100K

Figure 9: Accumulated runtime comparison when varying insertion rate.

In Table 7, we compare the maximum memory usages of the four algorithms when insertion ratio was set to the smallest values used in this experiment. Again, the minimum usage values are typeset boldface. In this table, we can see that the memory consumptions of the algorithms vary depending on the dataset used. As before, the proposed method requires the least memory on all the datasets, except Foodmart

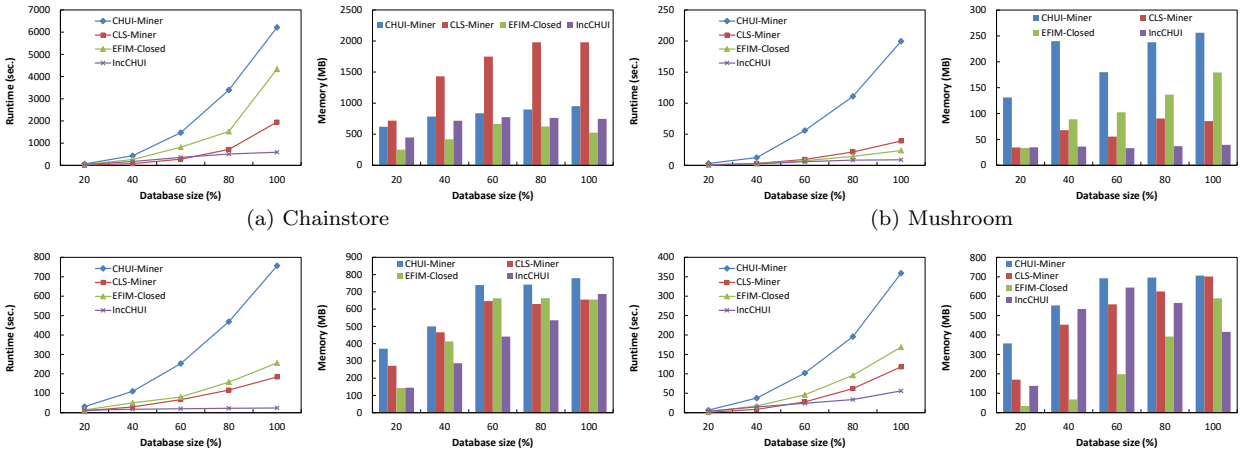
Table 7: Comparison of maximum memory consumption (MB) when varying insertion rate.

Dataset	CHUI-Miner	CLS-Miner	EFIM-Closed	IncCHUI
Chainstore	945.68	1225.50	**651.84**	848.00
Connect	372.03	695.54	693.44	**354.74**
Foodmart	173.72	261.04	**21.79**	59.70
Mushroom	291.10	260.36	227.91	**35.90**
OnlineRetail	781.89	657.90	649.25	**555.16**
Retail	704.85	719.23	643.87	**142.78**
T10I4D100K	710.67	634.98	656.32	**61.96**

and Chainstore. For the same reason as mentioned before, EFIM-Closed is much more memory efficient on Foodmart dataset than the remaining algorithms. CHUI-Miner, CLS-Miner, and IncCHUI are single phase and employ utility-list structure to store crucial information, but CLS-Miner needs to store other structures in memory for its pruning strategies. IncCHUI only maintains the set of closed itemsets found so far, and the global incremental utility-lists of single items.

5.4. Scalability tests

In the final set of experiments, we performed scalability tests in order to verify that the running time and memory consumption of our method do not grow exponentially. In addition, we study the impact of changes when increasing the number of transactions. We first use 20% of transactions in a database as the original database, then the incremental sections are added and processed. We executed the algorithms from the related work by employing the lowest *minutil* values as used in the previous experiments.



(a) Chainstore (b) Mushroom

(c) OnlineRetail (d) Retail

Figure 10: Scalability tests.

Fig. 10 shows the result of this evaluation. We note that the runtime for IncCHUI is the time to process each incremented part. Meanwhile, the run times for the remaining algorithms are the accumulated execution times, because these algorithms work with static databases and are run in batch mode. In this figure, we can see that the four algorithms require more execution time when the database size become larger, of which IncCHUI has the best scalability performance. The runtime performance of CHUI-Miner is the worst and it is much slower than the others. The reason is that CHUI-Miner only employs the basics overestimations (TWU and remain utility) without any other pruning strategies and complex structures, in contrast to CLS-Miner and EFIM-Closed. Moreover, since the related algorithms are run in batch mode and scan the database two times, this could contribute to the performance degradation.

In terms of memory usage, our experiments show (see Fig. 10) that the memory consumption of IncCHUI as function of the database size is almost constant on Chainstore and Mushroom, and increases linearly on

the OnlineRetail dataset. In general, for all the algorithms, the trend in memory consumption varies greatly depending on the dataset used, the structures employed by the algorithms, and the total number of generated candidates during the mining process. For example, Chainstore is a sparse and large dataset. On this dataset, CLS-Miner consumes more memory since it needs to store the structures for its pruning strategies, such as the coverage of items and the Estimated Utility Co-occurrence Structure. Note, however, that CLS-miner's runtime on this dataset is quite low. On the dense dataset Mushroom, the memory consumption of CLS-Miner is quite low, as a result of the effectiveness of its coverage pruning strategy. Nevertheless, IncCHUI has the overall least memory consumption on this dataset.

To summarize, IncCHUI shows the best scalability with respect to the sizes of the input databases compared to the other algorithms in our study. Also, the maximum memory used by IncCHUI for the complete mining process is small and reasonable. Finally, our evaluation confirms that both the runtime and the memory consumption of IncCHUI did not grow exponentially with the increased number of transactions.

6. Conclusion

In this work, we introduced a new approach, called IncCHUI, to efficiently discover and maintain closed high-utility itemsets (CHUIs) in incremental databases. To address the challenges with the dynamic nature of such databases, we developed a new algorithm applying several new important features. First, we proposed an incremental utility-list structure to store crucial information in itemsets. An important aspect of this list structure is that it is built and restructured by scanning the database only once, instead of scanning the whole database two times as other list-based methods do. Second, we suggested a new technique to mine updated items, and prune itemsets that are not present in the updated databases. Third, for efficiency, we proposed a method based on a hash table to maintain CHUIs. To evaluate our approach, we performed extensive experiments on both real-life and synthetic datasets, and compared it with the state-of-the-art methods. This evaluation confirmed the feasibility and efficiency of our method, which were achieved without the need to implement complicated pruning strategies or complex data structures.

Our future work includes studying more pruning strategies, and mining high-utility patterns in dynamic databases, with which old transactions normally become less important than newer ones, or have to be removed.

Acknowledgements

The basic ideas of this work were developed during the five months T-L. Dam spent as an ERCIM fellow at the Norwegian University of Science and Technology (NTNU), in 2017. This work was partly funded by the Norwegian Research Council through the ExiBiDa project, and the NTNU through the MUSED project.

References

[1] P. Fournier-Viger, J. C.-W. Lin, Q.-H. Duong, T.-L. Dam, FHM+: Faster High-Utility Itemset Mining Using Length Upper-Bound Reduction, in: Trends in Applied Knowledge-Based Systems and Data Science, Springer International Publishing, 2016, pp. 115–127.

[2] Q.-H. Duong, P. Fournier-Viger, H. Ramampiaro, K. Nørvåg, T.-L. Dam, Efficient high utility itemset mining using buffered utility-lists, Applied Intelligence 48 (2018) 1859–1877.

[3] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, VLDB (1994) 487–499.

[4] C. F. Ahmed, S. K. Tanbeer, B. S. Jeong, Y. K. Lee, Efficient tree structures for high utility pattern mining in incremental databases, IEEE Transactions on Knowledge and Data Engineering 21 (2009) 1708–1721.

[5] U. Yun, H. Ryang, Incremental high utility pattern mining with static and dynamic databases, Applied Intelligence 42 (2015) 323–352.

[6] A. Y. Peng, Y. S. Koh, P. Riddle, mHUIMiner: A fast high utility itemset mining algorithm for sparse datasets, in: Proceedings of PAKDD'2017, 2017, pp. 196–207.

[7] 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 Transactions on Knowledge and Data Engineering 27 (2015) 726–739.

[8] G. Grahne, J. Zhu, Fast algorithms for frequent itemset mining using FP-trees, IEEE Transactions on Knowledge and Data Engineering 17 (2005) 1347–1362.

[9] M. J. Zaki, C.-J. Hsiao, Efficient algorithms for mining closed itemsets and their lattice structure, IEEE Transactions on Knowledge and Data Engineering 17 (2005) 462–478.

[10] C. Lucchese, S. Orlando, R. Perego, Fast and memory efficient mining of frequent closed itemsets, IEEE Transactions on Knowledge and Data Engineering 18 (2006) 21–36.

[11] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Efficient mining of association rules using closed itemset lattices, Information Systems 24 (1999) 25 – 46.

[12] T. Le, B. Vo, An N-list-based algorithm for mining frequent closed patterns, Expert Systems with Applications 42 (2015) 6648 – 6657.

[13] L. T. T. Nguyen, T. Trinh, N. T. Nguyen, B. Vo, A method for mining top-rank-k frequent closed itemsets, Journal of Intelligent and Fuzzy Systems 32 (2017) 1297–1305.

[14] C. W. Wu, P. Fournier-Viger, J. Y. Gu, V. S. Tseng, Mining closed+ high utility itemsets without candidate generation, in: 2015 Conference on Technologies and Applications of Artificial Intelligence (TAAI), 2015, pp. 187–194.

[15] T.-L. Dam, K. Li, P. Fournier-Viger, Q.-H. Duong, CLS-Miner: efficient and effective closed high-utility itemset mining, Frontiers of Computer Science (2018) 1–25.

[16] P. Fournier-Viger, S. Zida, J. C.-W. Lin, C.-W. Wu, V. S. Tseng, EFIM-Closed: Fast and memory efficient discovery of closed high-utility itemsets, in: Proceedings of Machine Learning and Data Mining in Pattern Recognition (MLDM) 2016, 2016.

[17] Y. Liu, W.-k. Liao, A. Choudhary, A two-phase algorithm for fast discovery of high utility itemsets, in: Advances in Knowledge Discovery and Data Mining, 2005, pp. 689–695.

[18] V. S. Tseng, C.-W. Wu, P. Fournier-Viger, S. Y. Philip, Efficient algorithms for mining top-k high utility itemsets, IEEE Transactions on Knowledge and Data Engineering 28 (2016) 54–67.

[19] M. Liu, J. Qu, Mining high utility itemsets without candidate generation, in: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, CIKM '12, 2012, pp. 55–64.

[20] V. Tseng, B.-E. Shie, C.-W. Wu, P. Yu, Efficient algorithms for mining high utility itemsets from transactional databases, IEEE Transactions on Knowledge and Data Engineering 25 (2013) 1772–1786.

[21] Q.-H. Duong, B. Liao, P. Fournier-Viger, T.-L. Dam, An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies, Knowledge-Based Systems 104 (2016) 106–122.

[22] T.-L. Dam, K. Li, P. Fournier-Viger, Q.-H. Duong, An efficient algorithm for mining top-k on-shelf high utility itemsets, Knowledge and Information Systems 52 (2017) 621–655.

[23] C. Ahmed, S. Tanbeer, B. S. Jeong, Y. K. Lee, Efficient tree structures for high utility pattern mining in incremental databases, IEEE Transactions on Knowledge and Data Engineering 21 (2009) 1708–1721.

[24] P. Fournier-Viger, J. C.-W. Lin, T. Gueniche, P. Barhate, Efficient incremental high utility itemset mining, in: Proceedings of the ASE BigData & Social Informatics 2015, 2015, pp. 53:1–53:6.

[25] D. Kim, U. Yun, Efficient algorithm for mining high average-utility itemsets in incremental transaction databases, Applied Intelligence 47 (2017) 114–131.

[26] U. Yun, H. Ryang, G. Lee, H. Fujita, An efficient algorithm for mining high utility patterns from incremental databases with one database scan, Knowledge-Based Systems 124 (2017) 188 – 206.

[27] J. Lee, U. Yun, G. Lee, E. Yoon, Efficient incremental high utility pattern mining based on pre-large concept, Engineering Applications of Artificial Intelligence 72 (2018) 111 – 123.

[28] G. Wensheng, L. J. ChunWei, F. Philippe, C. HanChieh, H. TzungPei, F. Hamido, A survey of incremental highutility itemset mining, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 8 (2018) e1242.

[29] J.-S. Yeh, C.-Y. Chang, Y.-T. Wang, Efficient algorithms for incremental utility mining, in: Proceedings of the 2nd International Conference on Ubiquitous Information Management and Communication, ICUIMC '08, 2008, pp. 212–217.

[30] J. W. Han, J. Pei, Y. W. Yin, Mining frequent patterns without candidate generation: A frequent-pattern tree approach, Data Mining and Knowledge Discovery 8 (2004) 53–87.

[31] C.-W. Lin, T.-P. Hong, G.-C. Lan, J.-W. Wong, W.-Y. Lin, Incrementally mining high utility patterns based on pre-large concept, Applied Intelligence 40 (2014) 343–357.

[32] T.-P. Hong, C.-Y. Wang, Y.-H. Tao, A new incremental data mining algorithm using pre-large itemsets, Intelligent Data Analysis 5 (2001) 111–129.

[33] J. C.-W. Lin, W. Gan, T.-P. Hong, J.-S. Pan, Incrementally updating high-utility itemsets with transaction insertion, in: X. Luo, J. X. Yu, Z. Li (Eds.), Proceedings of Advanced Data Mining and Applications, 2014, pp. 44–56.

[34] J. C.-W. Lin, W. Gan, T.-P. Hong, B. Zhang, An incremental high-utility mining algorithm with transaction insertion, The Scientific World Journal 2015 (2015).

[35] T.-P. Hong, C.-H. Lee, S.-L. Wang, Effective utility mining with the measure of average utility, Expert Systems with Applications 38 (2011) 8259 – 8265.

[36] U. Yun, D. Kim, Mining of high average-utility itemsets using novel list structure and pruning strategy, Future Generation Computer Systems 68 (2017) 346 – 360.

[37] H.-F. Li, H.-Y. Huang, S.-Y. Lee, Fast and memory efficient mining of high-utility itemsets from data streams: with and without negative item profits, Knowledge and Information Systems 28 (2011) 495–522.

[38] H. Ryang, U. Yun, High utility pattern mining over data streams with sliding window technique, Expert Systems with Applications 57 (2016) 214 – 231.

[39] U. Yun, D. Kim, E. Yoon, H. Fujita, Damped window based high average utility pattern mining over data streams, Knowledge-Based Systems 144 (2018) 188–205.

[40] Q.-H. Duong, H. Ramampiaro, K. Nørvåg, P. Fournier-Viger, T.-L. Dam, High utility drift detection in quantitative data streams, Knowledge-Based Systems (2018).

[41] R. Agrawal, R. Srikant, Quest Synthetic Data Generator. Available at, (<http://www.almaden.ibm.com/cs/quest/syndata.html>), 1994.