Efficiently Updating the Discovered High Average-Utility Itemsets with Transaction Insertion

Jerry Chun-Wei Lin[1,2], Shifeng Ren[2], Philippe Fournier-Viger[3]
Jeng-Shyan Pan[1], and Tzung-Pei Hong[4,5]
[1]Fujian Provincial Key Laboatory of Big Data Mining and Applications
Fujian University of Technology, Fujian, China
[2]School of Computer Science and Technology
Harbin Institute of Technology Shenzhen Graduate School, Shenzhen, China
[3]School of Natural Sciences and Humanities
Harbin Institute of Technology Shenzhen Graduate School, Shenzhen, China
[4]Department of Computer Science and Engineering
National University of Kaohsiung, Kaohsiung, Taiwan
[5]Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan
jerrylin@ieee.org; renshifeng@stmail.hitsz.edu.cn; philfv@hitsz.edu.cn
jengshengpan@fjut.edu.cn; tphong@nuk.edu.tw

Abstract

High-utility itemset mining (HUIM) is an extension of frequent-itemset mining (FIM) but considers the unit profit and quantity of items to discover the set of high-utility itemsets (HUIs). Traditionally, the utility of an itemset is the summation of the utilities of the itemset in all the transactions regardless of its length. This approach is, however, inappropriate in real-world applications since the utility of the itemset increases along with the number of items within it. High average-utility itemset mining (HAUIM) was designed to provide more reasonable utility measure by taking the size of the itemset into account. Several algorithms were respectively designed to efficiently and effectively mine the set of high average-utility itemsets (HAUIs). Those algorithms can only handle, however, the static database and unsuitable for the dynamic environment since the size of data is frequently changed in real-life situations. In this paper, an incremental high-average utility pattern mining (IHAUPM) algorithm is presented to handle the incremental database with transaction insertion. The well-known fast updated (FUP) concept in the FIM is modified to adopt the designed algorithm, thus efficiently updating the discovered HAUIs. Based on the designed model for HAUIM with transaction insertion, the proposed IHAUPM algorithm can easily only handle the inserted transactions. Experiments are carried on six datasets and the results showed that the designed algorithm has better performance than the state-of-the-art algorithms performing in the batch manner.

Keywords: high average-utility itemsets; FUP; dynamic; insertion; incremental.

1 Introduction

Data mining or knowledge discovery can be referred to discover useful, potential, and implicit information from a very large database [1, 2, 13, 38]. The discovered information can thus be used to aid managers or decision makers for making the significant or efficient strategies. Based on different requirements in various applications, varied types of knowledge are respectively presented in different ways. Apriori algorithm [1] was the first algorithm to mine the correlations between the purchase itemsets, which can be referred as association-rule mining (ARM). It uses the level-wise approach to generate-and-test candidates at each level for finding the relationships among the items. The set of frequent itemsets (FIs) is first discovered by the Apriori algorithm based on the pre-defined minimum support threshold. After that, the set of FIs is then combined together, forming the set of association rules (ARs) based on the pre-defined minimum confidence threshold. This approach requires the amounts of computation and easily causes the memory leakage if the size of transactions in the database is too long, the frequent pattern (FP)-growth [13] algorithm was then further designed to mine the FIs based on its designed FP-tree structure. Several algorithms were widely studied to enhance the mining performance of the Apriroi-based [2, 6, 36, 38] and tree-based [3, 8, 13] algorithms.

Since the size of database is dynamically changed in real-life situations, such as some transactions could be inserted [7] into the original one; the discovered information is necessary to be updated since some new information may be arisen but some information may be lost. Most algorithms of FIM or ARM can only, however, handle the static database. When the data is changed in the database, the algorithms must be performed in the batch manner, which indicates that the already discovered information is useless and updated database is required to be scanned and examined to mine the required information. To handle the situation of transaction insertion, Cheung et al. proposed the Fast UPdated (FUP) [7] algorithm to adopt the pruning techniques used in in the DHP (Direct Hashing and Pruning) algorithm [37] for updating the discovered information. It divides the original databases and new inserted transactions into four parts and each part is performed by its own algorithm to update the required information. Since some unpromising itemsets in the FUP concept can be eliminated, the updating performance can thus be significantly improved.

In real-world situations, the frequent itemsets in FIM or ARM may only contribute a small portion to the overall profit, while non-frequent ones may contribute a large portion to the profit. High-utility itemset mining (HUIM) [31, 42, 43] was thus proposed to concern more factors such as unit profit and quantity of the items to reveal the high-utility itemsets (HUIs), which is an extension of FIM and widely applied in real-world situations. The utility of an itemset is to sum up the utilities of the itemset in all transactions regardless of its length [32, 42]. It is based on the measure of local transaction utility (quantity) and external utility (unit profit of items) to evaluate whether an itemset is a HUI. Liu et al. [31] then proposed the transaction-weighted utilization (TWU)-model to overestimate the upper value of the designed high-transaction-weighted utilization itemsets (HTWUIs), thus maintaining

the transaction-weighted downward closure (TWDC) property for mining the complete HUIs. To speed up mining performance, Lin et al. then presented a high utility pattern (HUP)-tree algorithm [21] for mining the set of HUIs. Several algorithms were also extensively developed to mine the HUIs [5, 17, 39, 43].

In HUIM, the itemset within more number of items may increase its utility regardless of its length. Thus, it is unfair to judge whether the itemset is a HUI with different lengths according to the same user-specified minimum threshold. To solve this limitation, Hong et al. proposed a high average-utility itemset mining (HAUIM) [15] to alleviate the effect of the length of the itemset and identify a better utility effect by considering the size of the itemset than the original utility standard. The average-utility of an itemset is defined as the total utility of an itemset divided by its number of items within it. An itemset is called a high average-utility itemset (HAUI) if its average utility is no less than pre-defined minimum high average-utility count. Thus, HAUIM is a totally different issue compared to the traditional HUIM since the new downward closure property, upper-bound model, pruning strategy, as well as the mining procedure are required to be developed. Lin et al. presented a high average-utility pattern (HAUP)-tree algorithm [21] to mine the set of HAUIs based on the efficient tree structure. The projection-based algorithm called PAI [19] was also designed to speed up mining performance. Lin et al. [30] then designed a average-utility (AU)-list structure to efficiently mine the HAUIs from a static database, which is the state-of-the-art algorithm for mining HAUIs.

Since the previous works of HAUIM can only handle the static database, but in real-life situations, database size is dynamically changed; for example, some transactions are frequently inserted into the original database. As we mentioned above, some information may be thus arisen, but some information may be lost in the updated progress. Although the batch processing of the updated database can solve this problem, it causes, however, multiple database scans and the already discovered information become invalid and useless. In this paper, we propose an incremental algorithm to efficiently update the discovered HAUIs when some transactions are frequently inserted into the original database. Major contributions are summarized as follows.

- We modified FUP concept [7] used in the FIM and ARM for updating the set of the discovered HAUIs. We have also maintained the correctness and completeness of the updated HAUIs to ensure that all the required information is completely discovered and updated.

- We adopted the efficient HAUP-tree structure to maintain the necessary information for later mining progress. An incremental IHAUPM algorithm is developed to divide the original database and new transactions into four cases for maintenance. The itemsets of each case can be correctly maintained and processed.

- The designed algorithm is sometimes unnecessary to multiply rescan the original database for updating the discovered HAUIs compared to the existing state-of-the-art algorithms

running in the batch manner.

The rest of the paper is organized as follows. Related work is reviewed in Section 2. The preliminaries and problem statement of the designed algorithm are stated in Section 3. The developed models and designed algorithm are described in Section 4. An illustrated example is provided in Section 5. Experiments are conducted and evaluated in Section 6. The conclusion and future work are finally given in Section 7.

2 Related Work

In this session, the related works of incremental concept, the high-utility itemset mining (HUIM), and the high average-utility itemset mining (HAUIM) are respectively reviewed and discussed.

2.1 Incremental Concept

Data mining techniques are used to extract meaningful and implicit patterns or rules from a very large dataset, and the most common approach is called association-rule mining (ARM). Agrawal and Srikant first proposed the Apriori algorithm [1] to mine association rules (ARs) from a set of transactions. It applies the level-wise approach to first mine the frequent itemsets (FIs) based on the minimum support threshold and performs the combinational mechanism from the mined FIs to retrieve the ARs based on the minimum confidence threshold. Since the Apriori algorithm applies the level-wise approach, thus the huge amounts of runtime and memory usage are necessary. To solve this problem, Han et al. proposed the Frequent-Pattern-tree structure (FP-tree) structure [13] and FP-growth algorithm for efficiently mining FIs without generation of candidate itemsets. To further improve the efficiency of FP-growth, Grahne and Zhu [12] proposed a novel FP-array technique to greatly reduce traversing time of the FP-tree especially in the sparse datasets. Another NFP-tree structure [8] was also designed to reduce the number of tree nodes, as well as the size of Header_Table by keeping two counts of each tree node. Thus, fewer tree nodes are traversed in the tree structure to mine the FIs.

The above algorithms can only process the static database. In real-life situations, transactions usually grow over time and the mined information must be re-evaluated. For example in ARM, some new ARs may be generated and some old ones may become invalid when new transactions are inserted into the original database. The conventional algorithms are processed in the batch manner to re-process the entirely updated database whether the transactions are inserted, deleted or modified from the original database. This procedure requires the amounts of computational cost and waste the discovered knowledge. Cheung et al. thus proposed the Fast UPdated (FUP) concept [7] to effectively handle the incremental database with transaction insertion. The FUP concept can be used to divide the original database and newly inserted transactions into four cases and each case is performed by its own procedure to maintain and update the discovered information. After all items in four cases are respectively performed,

the up-to-date information is thus correctly maintained. The FUP concept can be concluded with four cases for handling the transaction insertion, shown in Figure 1.

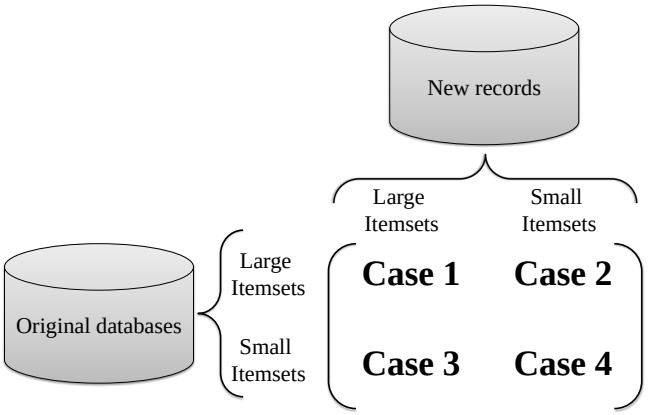

Figure 1: The FUP concept with transaction insertion.

In Figure 1, it can be seen that the itemsets in Case 1 are frequent both in the original database and in the new transactions; those itemsets will still be frequent after the database is updated. Similarly, the itemsets in Case 4 will still remain small after the database is updated. Thus, the results in Cases 1 and 4 will not affect the final frequent itemsets. The itemsets in Cases 2 and 3 may, however, be removed from the existing frequent itemsets or some frequent itemsets may be arisen.

Based on the FUP concept and the FP-tree structure, Hong et al. presented an incremental algorithm [14] for fast updating the frequent patterns. Although the FP-tree structure can be easily maintained in the memory, this algorithm can only be used handle the binary database. However, several realistic factors such as weight, interestingness, or unit profit of items can seriously influence the discovered patterns. It is thus a critical issue to develop more realistic algorithm to handle the incremental database.

2.2 High Utility Itemset Mining

High utility itemset mining (HAUIM) [5, 17, 39, 42, 43], an extension of high utility itemset mining (HUIM), is based on the measurement of local transaction utility and external utility. The local utility can be considered as the quantity of the items and the external utility can be considered as the unit profit of the items in the database. The utility of an itemset is to sum up the utilities of the items in the whole database. An itemset is concerned as the high-utility itemset (HUI) if its utility is no less than the minimum high-utility threshold.

Since the measure of HUIM is totally different than the measure used in the ARM, the downward closure (DC) property cannot be directly applied into the HUIM. In order to discover the complete and correct high utility itemsets, Liu et al. [32] first proposed a transaction-weighted downward closure property (TWDC) of itemsets to limit the search space and then designed the Two-Phase algorithm to level-wisely mine the HUIs. This approach requires

to generate-and-test candidates level-by-level, which suffers the problem of multiple database scans. Besides, all the potential itemsets at each level are necessary kept for finding the actual HUIs. Lin et al. then presented the high-utility pattern (HUP)-tree structure [21] to keep the necessary information in the main memory for further mining process. Tseng et al. then presented a compact UP-tree structure, the UP-Growth and UP-Growth+ algorithms and five pruning strategies for efficiently mining the HUIs [39]. Although the UP-Growth+ outperforms the Two-Phase algorithm, the memory leakage is still a problem in a large scale dataset. Liu and Qu then designed a novel list structure and developed a HUI-Miner algorithm [33] to mine the HUIs. The HUI-Miner is unnecessary to generate the amounts of candidates but still can discover the complete set of HUIs. Liu et al. then designed an algorithm [35] to mine the next itemsets using the suffix itemset for later combination. The reverse enumeration tree was designed as the search space and a liner CAUL data structure was used to keep the necessary information to facilitate the mining performance based on the designed d2HUP algorithm. Krishnamoorthy [17] proposed several pruning strategies to reduce the cost for constructing the utility-list structure to improve the performance of HUI-Miner. Similar as the mining limitation of FIs or ARs, it necessitates to frequently update the discovered information in the dynamic database. Ahmed et al. [4] then designed an incremental and interactive algorithms for mining HUIs. Lin et al. also presented an incremental algorithm [20] to update the discovered HUIs by employing the Two-Phase model and the FUP concept. Yun and Ryang designed a novel HUPID-Tree tree [44] and the HUPID-Growth algorithm to mine the HUIs. The over-estimated utility of the items is reduced and each node in the tree structure keeps more information for later mining process. A Header_Table keeps not only the utility value of each item but also its frequency.

Traditional algorithms of HUIM have to cope the exponential problem of the search space, Lin et al. then designed the PSO-based algorithm [28] to reduce the search space for mining the HUIs in a limit of time. To find the first k HUIs as the interesting patterns, Tseng et al. designed a top-k [40] approach for mining the HUIs. Duong et al. [10] then presented three pruning strategies to speed up the mining performance of [40]. In real-life applications, the HUIM can be applied in different situations. For example, in the basket analysis, the profit of items may be changed during the sales promotion. Lin et al. [25] then presented an one-phase HUI-DMiner algorithm and two pruning strategies to handle this issue for mining the precisely HUIs. Considering the positive and negative unit profit of the products during the promotion season [9], Lin et al. presented the FHN algorithm [24] and developed several pruning strategies to efficiently mine the HUIs. Based on the uncertainty constraint, for example, in the environment of wireless sensor network, the received data may consist of the inaccurate or uncertain information. Lin et al. then designed the algorithm [22] to mine the HUIs over the uncertain database by considering the excepted support model. The above algorithms consider that all items use a single minimum utility threshold to identify HUIs, however, it is inadequate and unfair to find the information with a single threshold since

all items are different and importance of each item should be concerned. Lin et al. then presented an algorithm to mine the HUIs under varied minimum high-utility thresholds [27]. This approach finds more reasonable HUIs than that of the traditional approaches.

The issue of privacy-preserving data mining (PPDM) has also emerging as a critical topic in recent decades, which can also be extended to the HUIM since some profitable products are considered as the confidential information. Lin et al. then surveyed the works of PPUM [23] and designed the algorithm [26] to hide the confidential HUIs and minimize three side effects in the sanitization process. Other issues related to HUIM are also developed in progress [29, 41, 18].

2.3 High Average-Utility Itemset Mining

In the traditional HUIM, the utility of an itemset is the summation of the utilities of the itemset in all the transactions regardless of its length. Thus, the utility of an itemset in a transaction increases along with the increasing of its length. This process will result that a longer itemset comes up with higher utility. Using the same minimum high utility threshold to identify the itemsets with different lengths is thus unfair. The high average-utility itemset mining (HAUIM) [15] was designed to alleviate the effect of the length of itemsets and identify really good utility itemsets. The average-utility measure was designed to reveal a better utility effect of combining several items than the original utility measure. The average-utility of an itemset is defined by calculating the total utility of an itemset divided by the number of items within the itemset. Several algorithms were extensively studied to mine the high average-utility itemsets (HAUIs). The Apriori-like TPAU [15] was developed to level-wisely mine the HAUIs under the generate-and-test approach. This approach requires the amounts of computation to find the HAUIs. Lan et al. then provided a projection-based approach [19] to tight the upper-bound of the promising candidates to find the HAUIs.

To speed up mining performance, Lin et al. proposed a high average-utility pattern (HAUP)-tree algorithm [21] for integrating the TPAU and FP-tree structure to firstly construct a condensed tree structure then efficiently mining HAUIs. To improve the mining performance, the HAUI-Miner [30] algorithm was presented to adopt the HUI-Miner structure for efficiently discovering the HAUIs without candidate generation. It uses the average-utility (AU)-list to keep the necessary information for the later join operation and mining progress. Although the above approaches can be used to efficiently mine the HAUIs, all of them fail to efficiently update the discovered HAUIs when the size of database is frequently changed. For example, when the transactions are frequently inserted into the original database, the above approaches have to process the entire database in the batch manner, which indicates that the updated database is scanned again to mine the updated information even though the size of inserted transactions is quite small. Besides, the computational cost is high since the set of the discovered HAUIs becomes old information; some HAUIs become invalid and some itemsets may arise as the new HAUIs. It is thus an important issue to efficiently updated the discovered information especially in the big data era under the dynamic environment.

3 Preliminaries and Problem Statement

3.1 Preliminaries

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a finite set of m distinct items. A quantitative database is a set of transactions $D = \{T_1, T_2, \ldots, T_n\}$, where each transaction $T_q \in D$ $(1 \leq q \leq m)$ is a subset of I and has a unique identifier q, called its *TID*. Besides, each item i_j in a transaction T_q has a purchase quantity denoted as $q(i_j, T_q)$. A profit table *PTable* indicates the unit profit value of each item in the database as $PTable = \{p(i_1), p(i_2), \ldots, p(i_m)\}$, where profit values are positive integers. A set of k distinct items $X = \{i_1, i_2, \ldots, i_k\}$ such that $X \subseteq I$ is said to be a k-itemset, where k is the length of the itemset. An itemset X is said to be contained in a transaction T_q if $X \subseteq T_q$. A minimum high average-utility threshold δ is set according to the user's preference (a positive integer). A quantitative database is shown in Table 1, which will be used as running example for the rest of this paper. This database contains five transactions and seven distinct items, denoted with letters from (a) to (g). The profit table indicates the unit profit of each item appearing in the database, and is shown in Table 2.

Table 1: A quantitative database.

TID	**Items with their quantities**
T_1	a:5, b:2, c:3, d:2
T_2	b:5, d:4, e:2, g:12
T_3	a:1, c:5, f:4
T_4	c:2, d:3, e:3
T_5	a:3, d:2, f:6

Table 2: A profit table.

Item	a	b	c	d	e	f	g
Profit	4	6	5	9	12	8	3

Definition 1. The average-utility of an item i_j in a transaction T_q is denoted as $au(i_j, T_q)$, and defined as:

$$au(i_j, T_q) = \frac{q(i_j, T_q) \times p(i_j)}{1}, \tag{1}$$

where $q(i_j, T_q)$ is the quantity of i_j in T_q, and $p(i_j)$ is the unit profit value of i_j.

For example, the average-utility of items (b), (c), and (d) in T_1 is respectively calculated as $au(b, T_1)(=\frac{2\times6}{1})(= 12)$, $au(c, T_1)(=\frac{3\times5}{1})(= 15)$, and $au(d, T_1)(=\frac{2\times9}{1})(= 18)$.

Definition 2. The average-utility of a k-itemset X in a transaction T_q is denoted as $au(X, T_q)$, and defined as:

$$au(X, T_q) = \frac{\sum\limits_{i_j \subseteq X \wedge X \subseteq T_q} q(i_j, T_q) \times p(i_j)}{|X|} = \frac{\sum\limits_{i_j \subseteq X \wedge X \subseteq T_q} q(i_j, T_q) \times p(i_j)}{k}, \tag{2}$$

where k is the number of items in X.

For example, the average-utility of itemsets (ac) and (acd) in T_1 are respectively calculated as $au(ac) = \frac{5\times4+3\times5}{2} = (17.5)$ and $au(acd) = \frac{5\times4+3\times5+2\times9}{3}$ $(= 17.6)$.

Definition 3. The average-utility of an itemset X in D is denoted as $au(X)$, and is defined as:

$$au(X) = \sum_{X \subseteq T_q \wedge T_q \in D} au(X, T_q). \tag{3}$$

For example, the average-utility of an itemset (ac) in the database depicted in Table 1 is calculated as $au(ac) = au(ac, T_1) + au(ac, T_3) = \frac{35}{2} + \frac{29}{2} (= 32)$.

Definition 4. The transaction utility of a transaction T_q is denoted as $tu(T_q)$, and defined as:

$$tu(T_q) = \sum_{i_j \subseteq T_q} u(i_j, T_q). \tag{4}$$

For example, the utilities of transactions in Table 1 are respectively calculated as $tu(T_1) = 20 + 12 + 15 + 18 (= 65)$, $tu(T_2)$ $(= 126)$, $tu(T_3) (= 61)$, $tu(T_4) (= 73)$, and $tu(T_5) (= 78)$.

Definition 5. The total utility of a database D is denoted as TU^D, and defined as the sum of all transaction utilities, that is:

$$TU^D = \sum_{T_q \in D} tu(T_q). \tag{5}$$

For example, the total utility in the running example of Table 1 is calculated as $TU^D = 65 + 126 + 61 + 73 + 78 (= 403)$.

To maintain the downward closure (DC) property, the average-utility upper bound ($auub$) property [15] was designed to over-estimate the utility of the promising itemsets, thus holding the DC property of HAUIM.

Definition 6 (transaction-maximum utility)**.** The transaction-maximum utility of a transaction T_q denoted as $tmu(Tq)$ and defined as:

$$tmu(T_q) = max\{u(i_j) | i_j \subseteq X \wedge X \subseteq T_q\}. \tag{6}$$

For example in Table 1, the transaction-maximum utilities of transactions T_1 to T_5 are respectively calculated as $tmu(T_1) (= 20)$, $tmu(T_2) (= 36)$, $tmu(T_3) (= 32)$, $tmu(T_4) (= 36)$ and $tmu(T_5) (= 48)$.

Definition 7 (Average-utility upper-bound, $auub$ property)**.** The average utility upper bound of an itemset X is denoted as $auub(X)$ and defined as:

$$auub(X) = \sum_{X \subseteq T_q \wedge T_q \in D} tmu(T_q), \tag{7}$$

where $tmu(T_q)$ is the maximum utility of transaction T_q such that $i_j \subseteq X \wedge X \subseteq T_q$.

For example in Table 1, the *auub* values of (a) and (ac) are respectively calculated as $auub(a) = (20 + 32 + 48)(= 100)$ and $auub(ac)$ $(= 20 + 32)(= 52)$.

Based on the *auub* property, the DC property can be held, thus reducing the size of unpromising candidates. Thus, the set of high average-utility upper bound itemsets (HAUUBIs) can be found to ensure the **completeness** and **correctness** of the *auub* property.

Definition 8 (High average-utility upper bound itemset, $HAUUBI$)**.** An itemset X is a HAUUBI if it satisfies the condition as:

$$HAUUBI \leftarrow \{X | auub(X) \geq TU^D \times \delta\} \tag{8}$$

Assume that the minimum average-utility threshold is set as 14.4%. For example in Table 1, (a) is a HAUUBI since its $auub(a)(= 100 > 403 \times 14.4\% = 58.023)$ but $auub(ac)$ is not a HAUUBI since $auub(ac)(= 52 < 58.023)$.

3.2 Problem Statement

Assume that the newly inserted database named (d) is inserted into the original database (D), and its given example is shown in Table 3. The unit profit of the items of Table 3 is the same as the original database shown in Table 2.

Table 3: Newly inserted transactions.

TID	**Items**
T_6	a:4, b:6, c:4, d:2
T_7	a:5, c:3, e:1, f:2, g:6

The problem of incremental HAUIM with transaction insertion is to efficiently update the set of the discovered HAUIs. Thus, an itemset X is a HAUI in the updated database $(D + d)$ if its utility is no less than the updated minimum average-utility count. The set of HAUIs is thus formally defined as:

$$HAUIs \leftarrow \{X | au(X) \geq (TU^D + TU^d) \times \delta\}, \tag{9}$$

where TU^d is the total utility in the inserted database (d), and δ is the minimum high average-utility threshold, which can be specified by user's preference.

4 Proposed Incremental Algorithm

In the past, Lin et al. proposed a HAUP-growth mining algorithm to mine th HAUIs from the constructed HAUP-tree [21]. Each node in the HAUP-tree keeps the quantities of the

prefix items of the processed node in the branch, which can speed up mining performance without building the enormous conditional trees as the FP-growth does. The HAUP-growth algorithm is designed, however, for mining the HAUIs in the batch manner. Thus, when some transactions are inserted into the original database, the HAUP-growth algorithm needs to process the updated database again to mine the updated information since some rules may thus be arisen but some rules may be lost in the updated database. Several existing algorithms can only handle the static databases. In this paper, we present an incremental high-average-utility pattern mining (IHAUPM) algorithm to handle the updated databases for mining the HAUIs with transaction insertion. The designed IHAUPM algorithm has two phases. In the first phase, the built HAUP-tree from the original database is maintained and updated by the inserted transactions. This process divides the itemsets into four cases according to the modified FUP concept used in the FIM but the designed algorithm considers the *auub* property of high average-utility upper bound itemsets (HAUUBIs) for the incremental process. Each case is performed by the designed procedure to maintain the HAUP-tree. Based on the FUP concept and *auub* property of HAUUBIs, the four cases is presented in Figure 2.

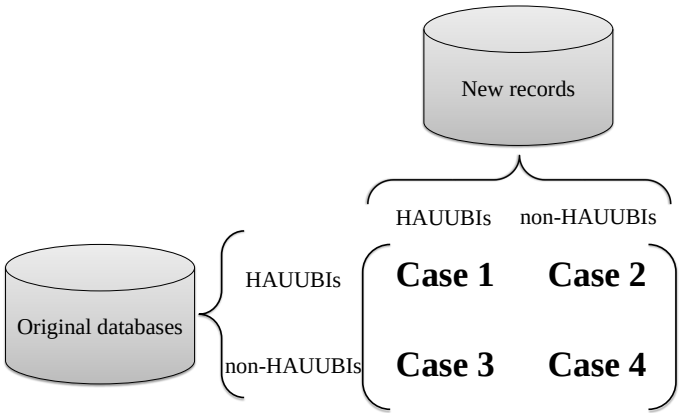

Figure 2: Proposed four cases model of incremental HAUIM.

The four cases model of the incremental HUIM can be described below.

- **Case 1:** An itemset is a HAUUBI both in the original database and in the inserted transactions. For an itemset X in case 1, we can obtain that:

$$auub(X)^U \geq (TU^D + TU^d) \times \delta. \tag{10}$$

- **Case 2:** An itemset is a HAUUBI in the original database, but is a non-HAUUBI in the inserted transactions. For an itemset X in case 2, we can obtain two situations without rescanning the original database D as:

$$\begin{cases} auub(X)^U \geq (TU^D + TU^d) \times \delta & \text{if } auub(X)^d \geq TU^d \times \delta; \\ auub(X)^U < (TU^D + TU^d) \times \delta & \text{if } auu(X)^d < TU^d \times \delta. \end{cases} \tag{11}$$

- **Case 3:** An itemset is a non-HAUUBI in the original database, but is a HAUUBI in the inserted transactions. For an itemset X in case 3, we can obtain two situations by rescanning original database D to get the $auub(X)^D$ as:

$$\begin{cases} auub(X)^D + auub(X)^d = auub(X)^U \geq (TU^D + TU^d) \times \delta; \\ auub(X)^D + auub(X)^d = auub(X)^U < (TU^D + TU^d) \times \delta. \end{cases} \tag{12}$$

- **Case 4:** An itemset is a non-HAUUBI both in the original database and in the inserted transactions. For an itemset X in case 4, we can obtain that:

$$auub(X)^U < (TU^D + TU^d) \times \delta. \tag{13}$$

From the above four cases, a summary of the final results in four cases is given in Table 4.

Table 4: Four cases of the incremental HAUIM.

Case: Original - New	**Results**
Case 1: HAUUBI - HAUUBI	Always frequent
Case 2: HAUUBI - non-HAUUBI	Decided from the existing information
Case 3: non-HAUUBI - HAUUBI	Decided by rescanning the original database
Case 4: non-HAUUBI - non-HAUUBI	Always non-HAUUBI

To ensure the **correctness** and **completeness** of the discovered HAUIs by the designed IHAUPM, the lemmas and proofs of four cases are given below.

Lemma 1 (Itemsets in Case 1)**.** An itemset X remains as the HAUUBI in the updated database (U) if its $auub(X)$ is no less than the updated minimum average-utility count both in D(= original database) and d(= inserted transactions).

Proof. Since $auub(X)^D \geq TU^D \times \delta$ and $auub(X)^d \geq TU^d \times \delta$, thus $auub(X)^U = auub(X)^D + auub(X)^d \geq TU^D \times \delta + TU^d \times \delta = (TU^D + TU^d) \times \delta \Rightarrow auub(X)^U \geq (TU^D + TU^d) \times \delta$.

□

Corollary 1. *An itemset X still remains a HAUUBI in the updated database if it belongs to **Case 1**; the itemsets in **Case 1** do not affect the number of the final updated itemsets, but affect the actual average-utilities of them.*

Lemma 2 (Itemsets in Case 2)**.** If an itemset X is a HAUUBI (appearing in HAUI-tree) in D but small in the inserted transactions d, two possible situations exist as: 1) it still remains a HAUUBI in U(= updated database) or 2) it becomes a non-HAUUBI in U since its $auub(X)$ is less than the updated minimum average-utility count.

Proof. Since $auub(X)^D \geq TU^D \times \delta$, the itemset X in d has two conditions as $auub(X)^d \geq TU^d \times \delta$ or $auub(X)^d < TU^d \times \delta$, thus:

(1) If $auub(X)^d \geq TU^d \times \delta$, we can obtain that $auub(X)^U = auub(X)^D + auub(X)^d \geq (TU^D + TU^d) \times \delta$.
(2) If $auu(X)^d < TU^d \times \delta$, we can obtain that $auub(X)^U = auub(X)^D + auub(X)^d < (TU^D + TU^d) \times \delta$.

□

Thus, for the itemsets in case 2, some itemsets still remain as the HAUUBI but some itemsets may become the non-HAUUBI in the updated database. For the itemsets in case 2, it is easier to maintain the updated information of the itemsets since those itemsets exist in the *Header_Table* of the HAUP-tree. Besides, after one database scan of the inserted transactions, the *auub* values of all itemsets in new transactions are discovered.

Corollary 2. *An itemset X may remain as a HAUUBI or become a non-HAUUBI in the updated database if it belongs to **Case 2**; the items in **Case 2** may be removed, and thus affect the final rules.*

Lemma 3 (Itemsets in Case 3)**.** If an itemset X is a small itemset (not existing in the *Header_Table* of HAUP-tree) but its $auub(X)^d$ in d is no less than the minimum high average-utility count of d, two possible conditions may exist when the database is updated as: 1) it becomes a HAUUBI or 2) it remains the non-HAUUBI in U.

Proof. If an itemset X does not exist in the *index_table* of HAUP-tree, the $auub(X)^D < TU^D \times \delta$.
Since $auub(X)^d \geq TU^d \times \delta$, D is then scanned to find the $auub(X)^D$.
For the updated database U, we can obtain that $auub(X)^U = auub(X)^D + auub(X)^d \geq auub(X)^D + TU^d \times \delta$. Thus, $auub(X)^U \geq (TU^D + TU^d) \times \delta$ or $auub(X)^U < (TU^D + TU^d) \times \delta$. □

Corollary 3. *An itemset X may become a HAUUBI or remain as a non-HAUUBI in the updated database if it belongs to **Case 3**; the itemsets in **Case 3** affect the final rules since some itemsets may become the HAUIs.*

The itemsets in Case 4 are unnecessary processed since they will never be the HAUUBIs, neither as the HAUIs. The lemma is given below.

Lemma 4 (Itemsets in Case 4)**.** An itemset X remains a non-HAUUBI in U if it is a non-HAUUBI both in D and d.

Proof. An itemset X is a non-HAUUBI in both D and d if its $auub(X)^D < TU^D \times \delta$ and $auub(X)^d < TU^d \times \delta$. Thus,
$auub(X)^U = auub(X)^D + auub(X)^d < TU^D \times \delta + TU^d \times \delta \ (= TU^D + TU^d) \times \delta$. □

Corollary 4. *An itemset X remains a non-HAUUBI in the updated database if it belongs to **Case 4**. The itemsets in **Case 4** do not affect the final results.*

According to the above lemmas and corollaries, the **correctness** and **completeness** of the proposed algorithm is obtained. In the designed model, the necessary information (the set of HAUUBIs) from the original database is kept in the main memory as a compressed tree structure for later mining progress. In real-world situation, the size of newly inserted transactions is quite small compared with the original database. Thus, it is easily to scan the newly inserted transactions to obtain the required information for later updating progress. When the transactions are inserted into the original database, we only need to scan the inserted parts to maintain and update the discovered information in the main memory. Although the original database is sometimes required to be scanned again while some itemsets belong to Case 3, this situation is rarely happened since the size difference is quite huge between the original database and newly inserted transactions. Thus, the multiple database scan can be successfully avoided. With the proposed algorithm for handling three cases (Case 4 can be directly ignored), the HAUP-tree can be efficiently and correctly updated and the required HAUIs can be successfully obtained by utilizing HAUP-growth method. The reason is that the *auub* property is used as the upper bound to maintain the downward closure property of the HAUUBIs. Thus, if an item/set is not a HAUUBI, its superset will not be the HAUUBI, neither as the HAUI. Therefore, the results of the proposed IHAUPM algorithm are correct and complete. Details of the designed incremental approach are described in Algorithm 1.

First, the new inserted database is scanned to find the transaction-maximum utility (tmu) of each transaction (Lines 1 to 2), and the *auub* of each item in d is then calculated (Lines 3 to 4). After that, the total utility of d is then calculated, as well as the summed up total utility of the updated database (Lines 5 to 6). The items in four cases are respectively processed. Notice that the itemsets in case 4 can be ignored since they cannot be the HAUIs according to our proofs. If the updated *auub* of each itemset is greater than or equal to the updated minimum high average-utility count, the itemset belong to case 1; the *auub* of each itemset in the *Header_Table* of the HAUP-tree is then updated, and the satisfied itemset is then put into the set of *insert_set* for later processing (Lines 9 to 12). If the updated *auub* of the itemset is less than the updated minimum high average-utility count, the itemset is directly removed from the HAUP-tree, and the child nodes of the processed node in the HAUP-tree are directly linked to the parent node of it (Lines 14 to 16). Otherwise, if the *auub* of the itemset in d is no less than the minimum high average-utility count and the processed itemset does not exist in the HAUP-tree, the original database is scanned again to find the actual *auub* of the itemet and put it in the set of *rescan_set* for updating the corresponding branch in the HAUP-tree (Lines 18 to 21). After that, the itemsets in the *rescan_set* are sorted by their *auub-descending* order (Line 22) and they are respectively inserted into the end of Header_Table (Line 23). The **UPdateBranch** function is then performed on two different sets (Lines 24 to 25) to update the HAUP-tree. Details are shown in Algorithm 2.

For the processed itemsets in the set (*rescan_set* or *insert_set*), if the itemset exists in the branch of the HAUP-tree (Lines 3 to 5), the *auub* value of the node is then updated by addition

Algorithm 1: Incremental high-average utility pattern mining (IHAUPM)

Input: D, a quantitative database; *PTable*, a profit table; the total utility TU^D in D; δ, the minimum average-utility threshold; *htree*, the built HAUP-tree from D; d, a set of inserted transactions.

Output: The updated HAUP-tree.

```
1  for each T_q ∈ d do
2      calculate tmu(T_q);
3  for each i_j in d do
4      calculate auub(i_j);
5  calculate total utility in d as TU^d;
6  calculate updated total utility TU^U = TU^D + TU^d;
7  for each i_j ∈ index_table of htree do
8      for each i_j in d do
9          if auub(i_j)^D + auub(i_j)^d ≥ TU^U × δ then
10             auub(i_j)^U = auub(i_j)^D + auub(i_j)^d;
11             update auub(i_j) ∈ index_table as auub(i_j)^U;
12             insert_set ← insert_set ∪ i_j;
13         else if auub(i_j)^D + auub(i_j)^d < TU^U × δ then
14             parent_node(i_j).child← child_node(i_j).parent;
15             remove i_j from htree;
16             remove i_j from index_table;
17         else
18             if auub(i_j)^d ≥ TU^d × δ ∧ i_j ∉ index_table then
19                 rescan D to obtain auub(i_j)^D;
20                 if auub(i_j)^D + auub(i_j)^d ≥ TU^U × δ then
21                     rescan_set ← rescan_set ∪ i_j;
22 sort rescan_set in current auub-descending order;
23 insert items in rescan_set to the end of index_table;
24 UPdateBranch(rescan_set);
25 UPdateBranch(insert_set);
```

Algorithm 2: UPdateBranch(set)

Input: the itemsets in the *set*.
Output: The updated *htree*.

```
1  for each i_j in the set do
2      for each i_j ⊆ T_q ∧ T_q ∈ D || T_q ∈ d do
3          if i_j ∈ htree.branch ∧i_j ∈ insert_set then
4              node(i_j).auub+ = tmu(T_q);
5              node(i_j).quan_Ary += T_q.quantity[1..j];
6          else
7              if i_j ∈ insert_set ∧i_j ∈ rescan_set then
8                  find the i_j.position in T_q ∈ D;
9                  node(i_j).auub = tmu(T_q);
10                 node(i_j).quan_Ary = T_q.quantity[1..j];
```

(Line 4), as well as the attached quantity array of the node (Line 5); otherwise, the position of the itemset in the transaction is then found and inserted into its corresponding position of the branch in the HAUP-tree (Line 8). The *auub* value of the node is then added into the node (Line 9), and its quantity array is then corresponding updated (Line 10).

In the second phase, the updated HAUP-tree is then processed by the constructed *Header_Table* to respectively mine the HAUIs using the HAUP-growth approach [21]. This is a trivial task since the complete information is kept in the HAUP-tree structure. After that, the set of HAUIs is then maintained and updated.

5 An Illustrated Example

In this section, an example is given to demonstrate the proposed IHAUPM algorithm. Suppose that the original quantitative database and its profit table were respectively shown in Tables 1 and 2. The minimum high average-utility threshold is initially set as 14.4%, which can be defined by users' preference. The HAUP-tree was first built from Table 1 to keep the necessary information for later mining process. From the given example, the constructed HAUP-tree is then shown in Figure 3, and the total utility of Table 1 was calculated as 403.

We also assume that some transactions are then inserted into the original database, and the inserted transactions were shown in Table 3. In this example, the total utility of the inserted database was calculated as 171. Thus, in the updated database, the updated total utility is (403 + 171)(= 574), and the minimum high average-utility count is then calculated as (574× 14.4%)(= 82.656).

First, the *auub* values of all items in Table 3 are respectively calculated and the results are shown in Table 5.

Based on the modified FUP model, the *auub* of each item in the inserted transactions is

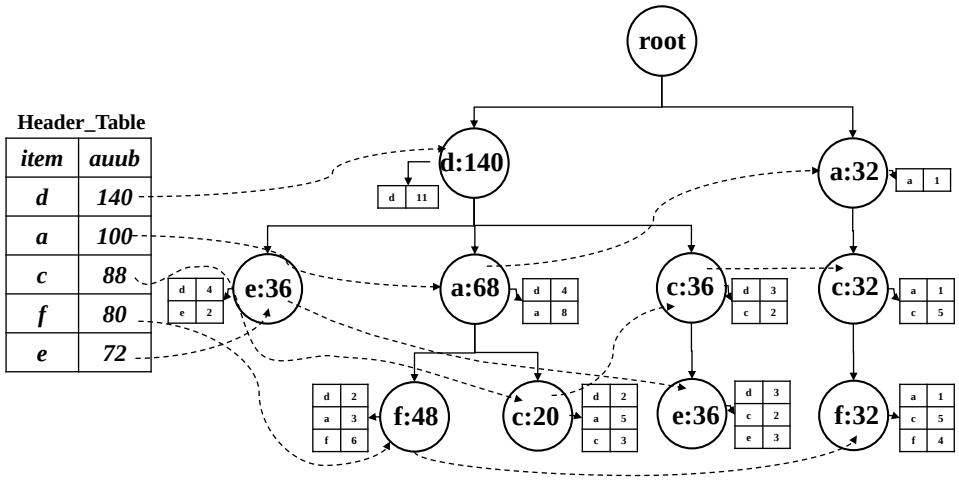

Figure 3: Constructed HAUP-tree.

Table 5: The *auub* values of all items in the inserted database.

Item	a	b	c	d	e	f	g
auub	56	36	56	36	20	20	20

checked against the minimum high average-utility count in the inserted transactions. In this example, the items (a), (c) and (d) belong to case 1; the items (e) and (f) belong to case 2, and the item (b) belongs to case 3. The remaining item (g) belongs to case 4 and can be directly ignored since it certainly cannot be a HAUI.

First, the items in case 1 are respectively processed to update their *auub* values in the *Header_Table* of HAUP-tree. For example, the *auub* value of (a) in the HAUP-tree was $auub(a)^D(= 100)$, and its *auub* value in the inserted database was calculated as $auub(a)^d(= 56)$; the updated *auub* value of (a) is summed up as $auub(a)^U(= 100 + 56)$ $(= 156 > 82.656)$. The item (a) is then put into the set of *insert_set* and its *auub* value in the *Header_Table* of HAUP-tree is then updated. This procedure is then processed for the items (c) and (d). After that, the *insert_set* $= \{a, c, d\}$. The items in case 2 are then processed. In this example, the *auub* values of (e) and (f) are respectively calculated and the results are $auub(e)^U(= 72 + 20)(= 92)$ and $auub(f)^U(= 80 + 20)(= 100)$. In this example, they remains as the potential HAUIs and the *auub* values of them in the *Header_Table* are respectively updated; (e) and (f) are both put into the set of *insert_set*. After that, the items in case 3 are then processed. In this example, only an item (b) exists in case 3; the original database is required to be scanned to find the *auub* value of (b) in the original database, and in this example, the $auub(b)^U(= 92)$, which is larger than the minimum high average-utility count; (b) is then put into the two sets of *insert_set* and *rescan_set*. Thus, *insert_set* $=\{a, b, c, d, e, f\}$, and *rescan_set* $= \{b\}$.

After that, the initially built HAUP-tree is updated based on the items in the *rescan_set*. In this example, it can be found that an item (b) appears in transaction T_1 and T_2 of Table 1; the corresponding branches of (b) in T_1 and T_2 are extracted. The corresponding branches

indicate that the items appear in the set of *insert_set* and corresponding to the items in the set of *rescan_set* in the original database, which can be used to update the HAUP-tree of the items in the *rescan_set* since those items did not exist in the HAUP-tree. The results are then shown in Table 6.

Table 6: Corresponding branches in the original database.

Tid	**Items with their quantities**	**Corresponding branch**
T_1	a:5, b:2, c:3, d:2	a:5, $\boldsymbol{b}$:2, c:3, d:2
T_2	b:5, d:4, e:2, g:12	$\boldsymbol{b}$:5, d:4, e:2

From the discovered corresponding branches of (b) shown in Table 6, the HAUP-tree is then updated and the item (b) is then inserted to the HAUP-tree according to its correct position in the branches. Besides, the attached array of the items (nodes) in the HAUP-tree is then updated. Since the items (a), (c), (d), and (e) were already inserted into the HAUP-tree except the item (b), it is unnecessary to inserted those items in the HAUP-tree.

After that, the items in *insert_set* are processed for handling the newly inserted transactions. For the inserted transactions, the corresponding branches of the items in *insert_set* are then found and the results are shown in Table 7.

Table 7: Corresponding branches in the inserted transactions.

Tid	**Items with their quantities**	**Corresponding branch**
T_6	a:4, b:6, c:4, d:2	a:4, b:6, c:4, d:2
T_7	a:5, c:3, e:1, f:2, g:6	a:5, c:3, e:1, f:2

Since the corresponding branches of Table 7 are from the newly inserted transactions; those branches are respectively inserted into the HAUP-tree following the construction phase of HAUP-tree. The final updated HAUP-tree is then shown in Figure 4.

After that, the HAUP-growth [21] is then processed to find the set of HAUIs. Notice that the HAUIs can be thus discovered while all the insertion operations are completely processed. From Figure 4, the discovered HAUIs are $\{a$:156, b:92, c:144, d:176, e:92, f:100$\}$.

6 Experimental Results

In this section, the performance of the proposed IHAUPM algorithm is compared with the state-of-the-art PAI [19], TPAU [15], HAUI-tree [35], HAUI-Miner [33] and the HAUP-growth [21] algorithms on several datasets. All algorithms were implemented in Java and experiments were carried on a computer having an Intel(R) Core(TM) i7-4790 3.60GHz processor with 8 GB of main memory, running the 64 bit Microsoft Windows 7 operating system. Experiments were conducted on six datasets, including four real-world datasets [11] and two synthetic dataset generated using the IBM Quest Synthetic Data Generator [16]. A simulation model [31] was developed to generate quantities (internal utilities) and unit profit values (external utilities)

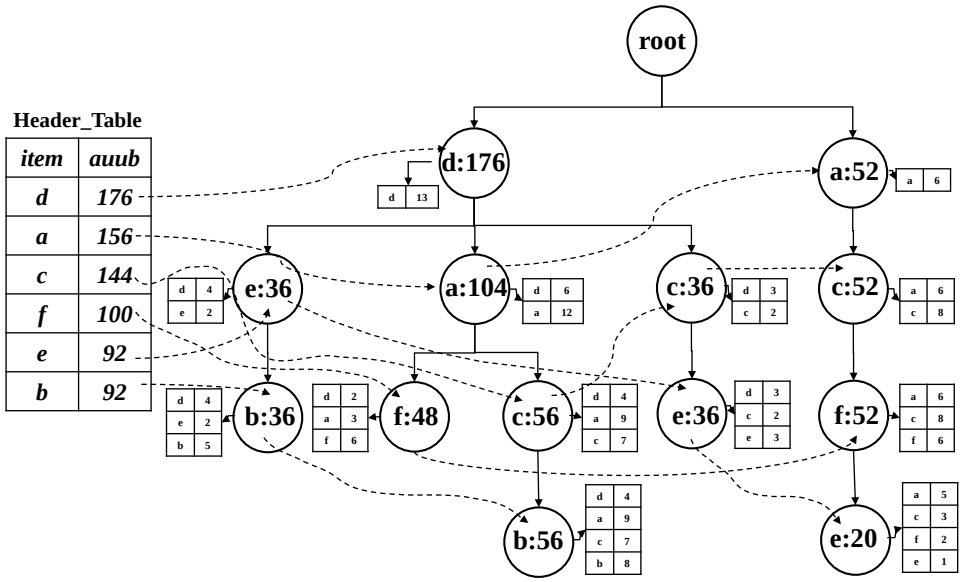

Figure 4: The final updated HAUP-tree.

of items in transactions for all datasets. External utilities have been generated in the [0.01, 10] interval using a log-normal distribution, and internal utilities have been randomly chosen in the [1,5] interval. Parameters used to describe the six datasets are shown in Table 8, and the characteristics of these datasets are shown in Table 9. In the experiments, the minimum average-utility threshold is denoted as **TH**, and the insertion ratio of the new transactions is denoted as **IR**. The **IR** is used to set the size of the newly inserted transactions compared to the original database. For example, if the **IR** is set as 5% and the size of the original dataset has 100 transactions, the number of newly inserted transactions is thus set as 5. To simulate the dynamic situation in real-world applications, the newly inserted transactions are randomly generated five times and then respectively inserted into the original dataset.

Table 8: Parameters of used datasets.

#|**D**|	Total number of transactions
#|**I**|	Number of distinct items
AveLen	Average length of transactions
MaxLen	Maximal length of transactions
Type	Database type (sparse or dense)

To assess the performance of the compared algorithms, the runtime, memory usage, number of determined candidates, and scalability were respectively analyzed. Results are reported below.

Table 9: Characteristics of used datasets.

Dataset	#|D|	#|I|	AvgLen	MaxLen	Type
foodmart	21,556	1,559	4	11	sparse
kosarak	990,002	41,270	8	2,498	sparse
retail	88,162	16,407	10	76	sparse
mushroom	8,125	120	23	23	dense
T10I4D100K	100,000	1,000	10	29	sparse
T40I10D100K	100,000	1,000	39.6	77	dense

6.1 Runtime

In this section, the runtime for mining HAUIs are compared under various minimum high average-utility thresholds on six datasets. Notice that the runtime is the sum up of I/O time, interval time and mining time. Results are shown in Figure 5.

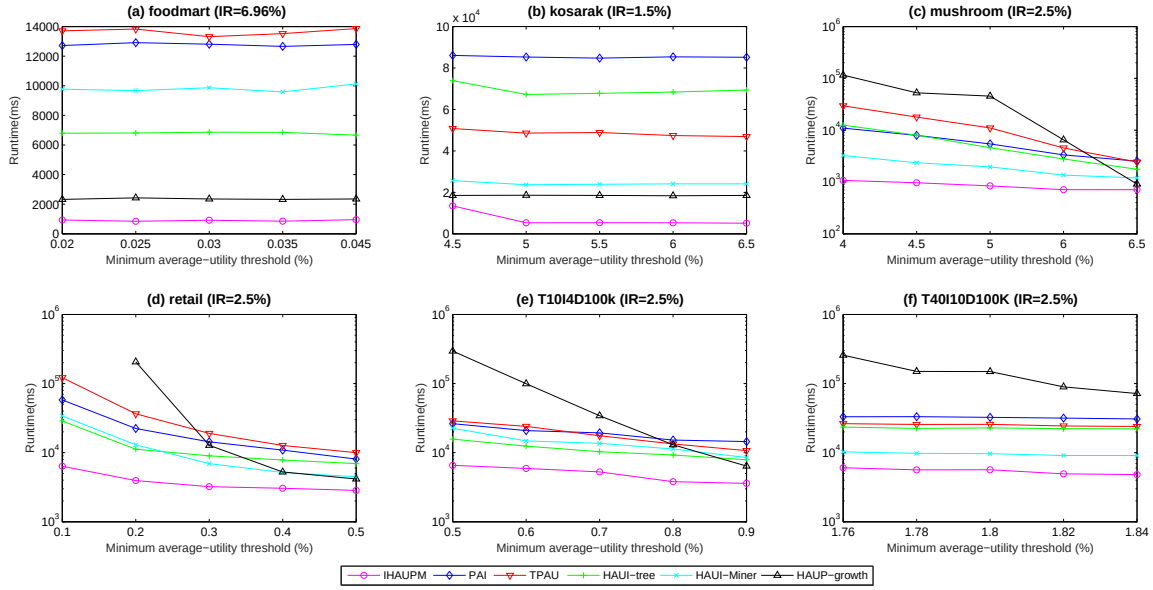

Figure 5: Runtime under various minimum high average-utility thresholds.

From Figure 5, it can be seen that the runtime decreases along with the increasing of minimum high average-utility thresholds. This is reasonable since when the minimum high average-utility threshold is set higher, fewer HAUIs are then discovered; thus runtime of the compared algorithms decreased. Moreover, it is obvious to see that the proposed IHAUPM algorithm outperforms the other algorithms under various minimum high average-utility thresholds. For example in Figure 5(a), the proposed IHAUPM has one order of magnitude faster than the TPAU and PAI algorithms and 10 times faster than the HAUI-Miner algorithm, 6 times faster than the HAUI-tree algorithm. This is reasonable since the TPAU and PAI algorithms apply the Two-Phase model to level-wisely discover the potential candidates for deriving the HAUIs. Although the HAUI-Miner is the state-of-the-art algorithm for mining HAUIs, it still requires

to handle the updated dataset in the batch manner, which means that the discovered information is thus ignored and the partial parts of the original dataset are scanned with multiple times. In addition, the runtime of the compared algorithms is then evaluated under various insertion ratios and the results are then shown in Figure 6.

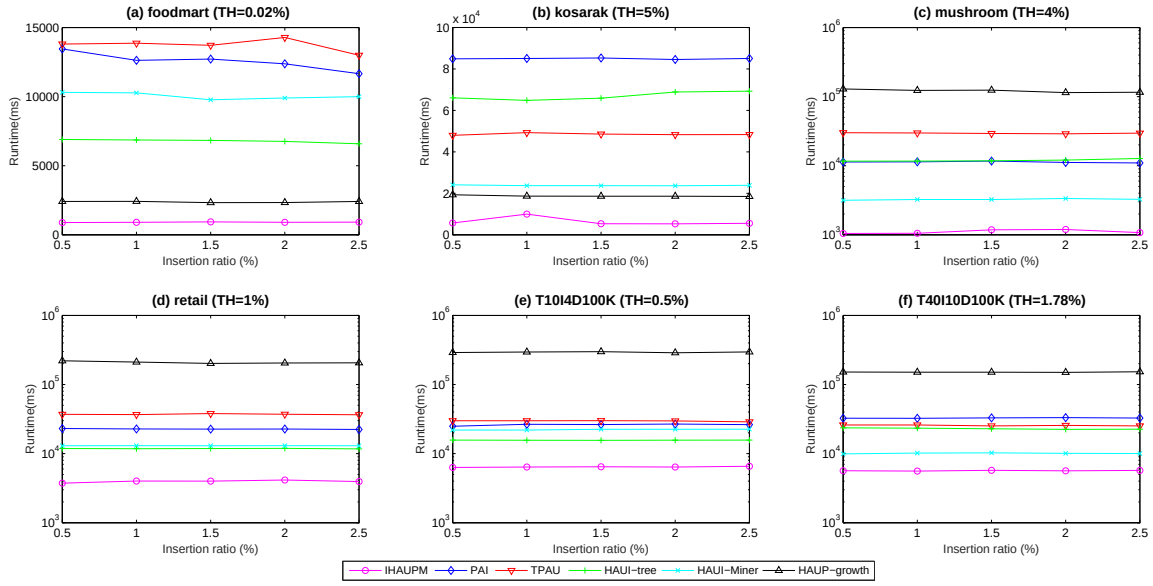

Figure 6: Runtime under various insertion ratios.

As shown in Figure 6, it can be concluded that the proposed IHAUPM performs well under various insertion ratios and has one or two orders of magnitude faster than the other algorithms. For example in Figure 6(f), the designed IHAUPM performs better than the state-of-the-art HAUI-Miner algorithm. The reason is that IHAUPM is designed for particularly handling the incremental database whereas the HAUI-Miner is performed on the batch manner and the database is required to be scanned with multiple times. Overall, the designed IHAUPM performs well than the other approaches.

6.2 Memory Usage

In this section, the memory usage for deriving the HAUIs from the updated dataset is compared. Results are shown in Figure 7.

It can be observed in Figure 7 that the designed IHAUPM algorithm has the least memory usage than the others under various minimum high average-utility thresholds on six datasets. For example in Figure 7(a), the memory usage of the designed algorithm is about 7 to 8 times less than PAI, TPAU, HAUI-tree and HAUP-growth and 1 times less than the state-of-the-art HAUI-Miner. The reason is that the PAI and TPAU are performed by the Two-Phase model, thus a large number of candidates is kept in the memory. The HAUI-Miner has no valid pruning strategies and it sometimes generates a large number of unpromising candidates, which leads to massive memory usage. The designed IHAUPM algorithm has, however, to maintain the

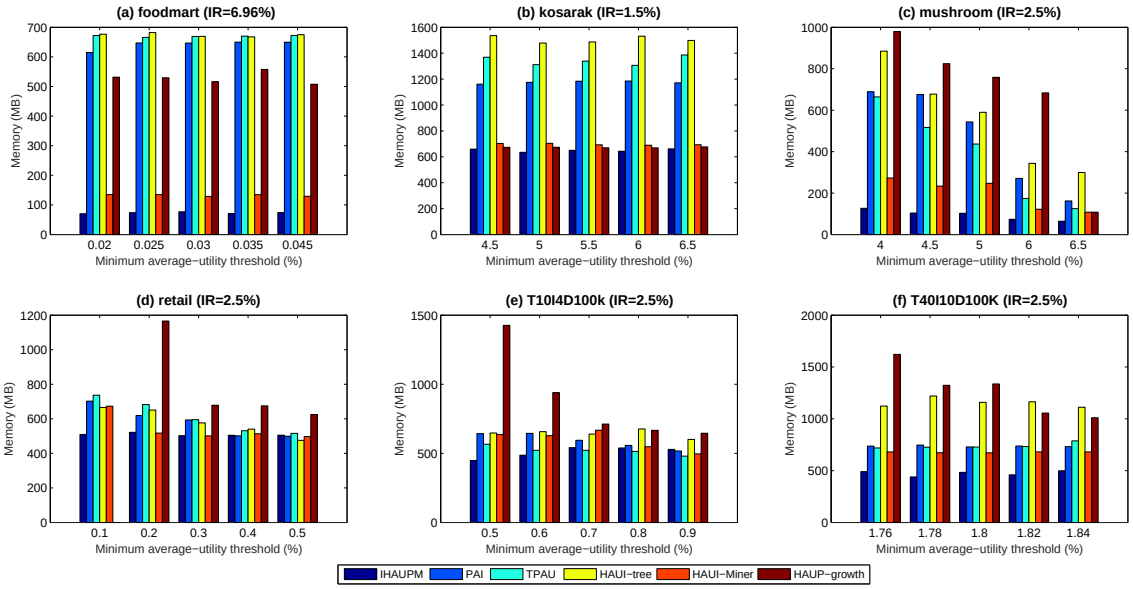

Figure 7: Memory usage under various minimum average-utility thresholds.

incremental parts of the inserted transactions, and the several unpromising candidates can be early pruned by the designed algorithm. From Figures 7(c), 7(d), 7(e), and 7(f), it can be found that the HAUP-growth needs a large number of memory usage. Although the HAUP-growth is efficient in mining process but it necessitates to keep the quantity information of the items in a branch, thus requiring more memory usage. In particular, when the minimum high average-utility threshold is set lower such as 0.1%, the HAUP-growth sometimes has the problem of memory leakage, which can be found in Figure 7(d). The memory usage of compared algorithms under various insertion ratios are then shown in Figure 8.

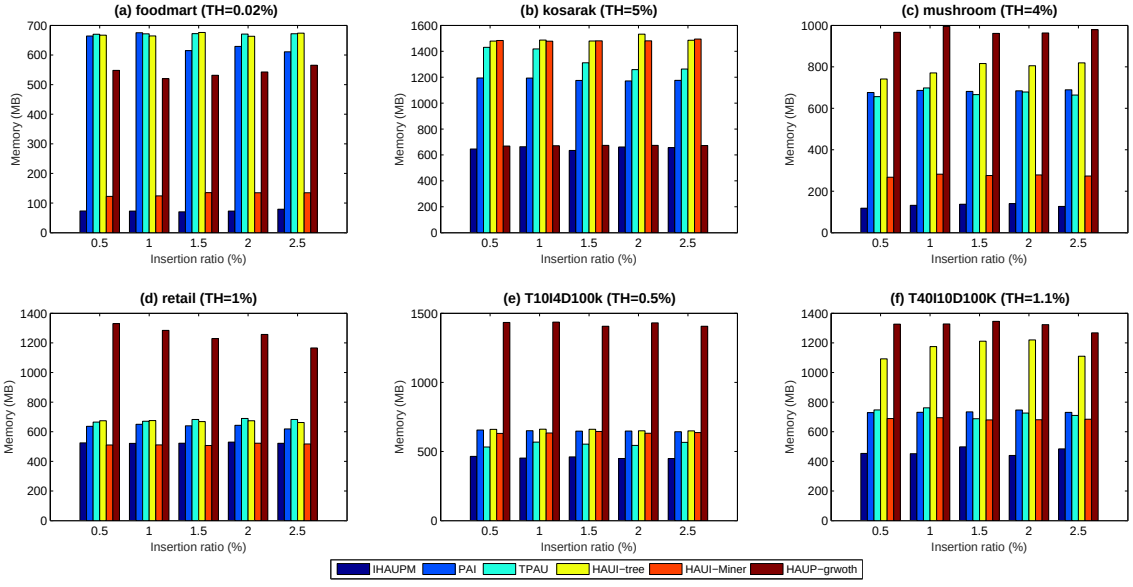

Figure 8: Memory usage under various insertion ratios.

It can be observed in Figure 8 that the designed IHAUPM algorithm outperforms the other algorithms under various insertion ratios. The HUAP-growth, PAI and TPAU almost have the most memory usage on six datasets. Although the HAUI-Miner has relatively low memory usage, the designed IHAUPM algorithm still performs well than HAUI-Miner. It also can be found that the HAUI-Miner is sometimes not efficiency since it is not designed for incremental database. Overall, the designed IHAUPM algorithm for handling the incremental database with transaction insertion performs well among the compared algorithms.

6.3 Number of Candidates

In this section, the number of candidates for finding the HAUIs under the various minimum high average-utility thresholds is compared. Results are shown in Figure 9.

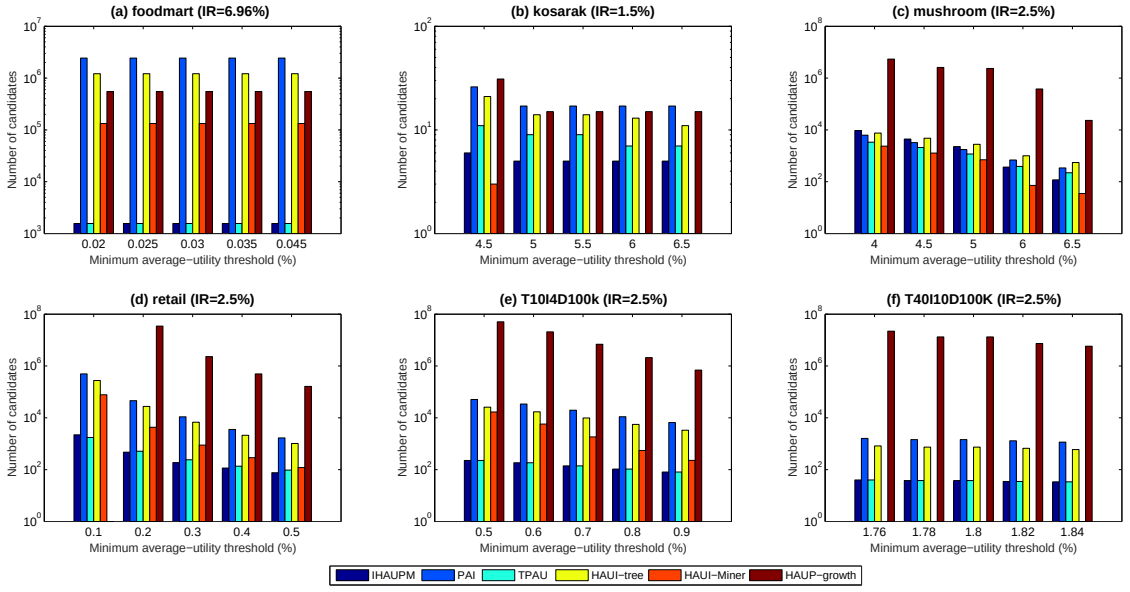

Figure 9: Number of candidates under various minimum high average-utility thresholds.

It can be observed in Figure 9 that the number of candidates generated by the designed algorithm is much less than the PAI, HAUI-tree and HAUP-growth algorithms. For example in Figure 9(a), the number of candidates generated by the designed algorithm is about 2 to 3 orders of magnitude less than PAI, HAUI-tree and HAUP-growth and the same as the TPAU. In Figures 9(b) and 9(c), the number of candidates of the designed algorithm is less than TPAU. The reason is that PAI, HAUI-tree and HAUP-growth has no valid pruning strategies to reduce the unpromising candidates for the whole updated dataset but the designed IHAUPM algorithm handles only the incremental parts; the size of the incremental dataset is much less than that of the updated one. Although the number of candidates generated by TPAU and HAUI-Miner is sometimes the same as or slightly more than designed algorithm, the TPAU suffers the inefficiency mining performance since it mines the HAUIs in the level-wise manner and the HAUI-Miner is designed for the static dataset but not for the incremental one. Besides,

the algorithms are also compared in terms of number of candidates under different insertion ratios and the results are shown in Figure 10.

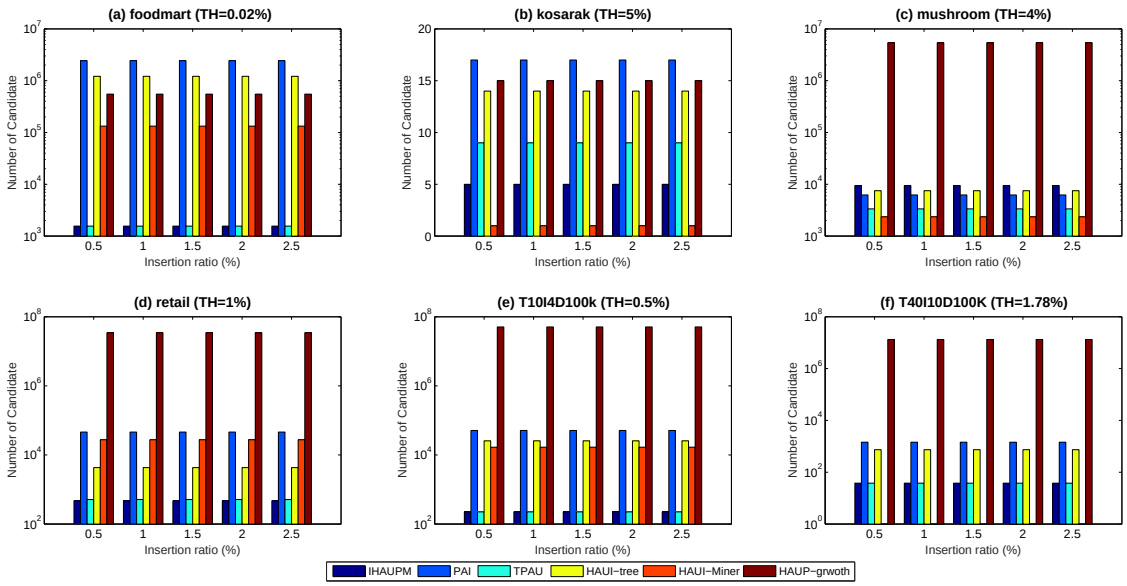

Figure 10: Number of candidates under various insertion ratios.

It can be observed in Figure 10 that the designed IHAUPM algorithm almost has the least number of candidates on six datasets under various insertion ratios, while the HAUP-growth and PAI almost have the most number of candidates. For example in Figure 10(a), the PAI has the two or three orders of magnitude more than the designed algorithm, and in Figure 10(c), the number of candidate of HAUP-growth is about 8 to 9 times more than others. Also, the number of candidates remains steady for all compared algorithms. It is reasonable since the size of the inserted transactions is much less than the original database; the designed IHAUPM algorithm only handles the small parts but the others need to maintain the whole updated database even though only few transactions are inserted into the database.

6.4 Scalability

In this section, the scalability of the algorithms were compared on a real-world kosarak dataset [11] and a series of synthetic dataset T10I4N4KD|X|K generated by IBM Generator [16] using the same parameter other than dataset size. |X| represents the dataset size, varying from 100K to 500K transactions with 100K increment in our experiments. To test the scalability of the proposed algorithm on real-world datasets, the original dataset is divided into 5 datasets kosarak|X|K, where the the dataset size is vary from 198K to 990K, with increments of 198K. The **TH** is set as 3.5% for the synthetic dataset and 4.5% for the kosarak dataset, and the **IR** is set as 5% for the synthetic datasets and 2.5% for the kosarak dataset. The results are respectively shown in Figures 11 and 12.

It can be observed in Figures 11 and 12 that the runtime, memory usage, and number of

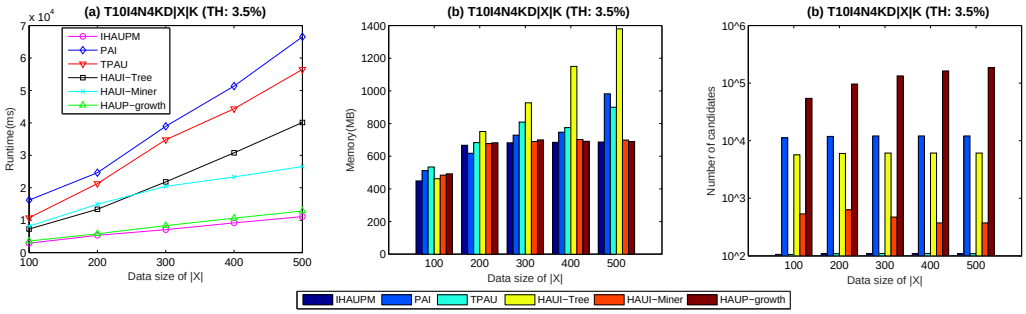

Figure 11: Scalability under various synthetic dataset sizes.

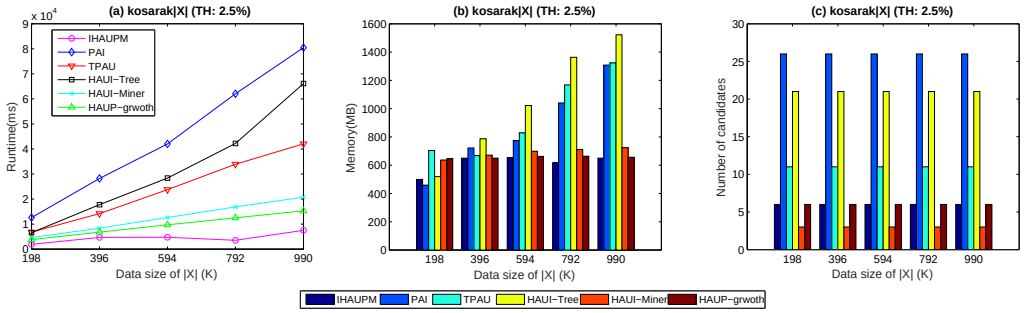

Figure 12: Scalability under various kosarak dataset sizes.

candidates increase along with the increasing of dataset size. This is reasonable since when the dataset size increases, the number of HAUIs increases, as well as the runtime, memory usage and number of candidates. The designed IHAUPM algorithm performs well than the other algorithms on both synthetic datasets and real-world datasets, and the PAI and TPAU algorithm have worse results of runtime. This is reasonable since they are performed in the level-wise manner to mine the HAUIs. The HAUI-tree requires the most memory usage since it keeps a lot of information in the tree structure. The HAUP-growth needs the most number of candidates since the potential and complete HAUIs are maintained by the HAUP-tree structure. Overall, the designed IHAUPM algorithm outperforms the others in terms of runtime, memory usage and the number of candidates. The effectiveness and efficiency of the designed algorithm is thus quite acceptable for real-world applications.

From the experiments, we can also see that the proposed algorithm performs than other algorithms in terms of runtime, memory usage and scalability in the incremental environment. This is reasonable since the designed algorithm is unnecessary to rescan the entire database with multiple times but only focusing on the incremental parts. For the batch-manner algorithms, the database is multiply rescanned whenever updated records come. Besides, the presented HAUP-tree keeps the sufficient information for later updating with only few additional memory usages to update the HAUIs. Thus, the proposed algorithm scales well whether in the synthetic or real-world datasets.

7 Conclusion and Future Work

High average utility mining (HAUIM) has attracted a lot of attention since it provides a fair measure than the HUIM to evaluate the discovered patterns. However, most algorithms of HAUIM are designed for handling the static database. Thus, when transactions are inserted into the original database, the updated database is required to be re-mined and the previous discovered information becomes useless.

To solve this problem, in this paper, we present an incremental high average-utility pattern mining (IHAUPM) algorithm for handling the transaction insertion in the dynamic databases. The FUP concept is thus modified and extended to handle original database and new transactions, and divide them into 4 cases. The compact HAUP-tree structure is used to maintain the 1-HAUUBIs, which can guarantee the correctness and completeness of the discovered HAUIs. The proof of each case in the designed algorithm is also provided. Based on the designed model, it rarely scans the original database except handling the itemsets in case 3. Theoretically, the designed algorithm can greatly reduce the computational cost while the transactions are frequently inserted into the original database. Thus, the designed algorithm can solve the realistic problem in the engineering, as well as provide the real-time decision-making system. Experimental results show that the proposed IHAUPM algorithm performs well than the state-of-the-art algorithms for maintaining the HAUIs.

Since transaction deletion and transaction modification also happen in real-life situations, it is necessary to design the efficient algorithms for updating the discovered information in the dynamic databases. Besides, a more compact data structure should be further designed to keep the necessary information especially in the limited memory.

Acknowledgment

This research was partially supported by National Natural Science Foundation of China (NSFC) under grant No.61503092 and and by the Tencent Project under grant CCF-Tencent IAGR20160115.

References

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," *The International Conference on Very Large Databases*, pp. 487-499, 1994.

[2] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," *ACM SIGMOD International Conference on Management of Data*, pp. 207-216, 1993.

[3] R. Agrawal, C. Aggarwal, and V. Prasad, "A tree projection algorithm for generation of frequent itemsets," *Journal of Parallel and Distributed Computing*, vol. 61, pp. 350-371, 2001.

[4] C. F. Ahmed, S. K. Tanbeer, B. S. Jeong, and Y. K. Lee, "Efficient tree structures for high utility pattern mining in incremental databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 12, pp. 17081721, 2009.

[5] R. Chan, Q. Yang, and Y. D. Shen, "Mining high utility itemsets," *IEEE International Conference on Data Mining*, pp. 19-26, 2003.

[6] C. Chang and C. Lin, "Perfect hashing schemes for mining association rules," *The Computer Journal*, vol. 48, pp. 168-179, 2005.

[7] D. Cheung, J. Han, V. Ng, and C. Wong, "Maintenance of discovered association rules in large databases: an incremental updating approach," *IEEE International Conference on Data Engineering*, pp. 106-114, 1996.

[8] C. Chi and K. Lain, "A new fp-tree algorithm for mining frequent itemsets," *The Content Computing, Advanced Workshop on Content Computing*, pp. 266-277, 2004.

[9] C. J. Chu, V. S. Tseng, and T. Liang, "An efficient algorithm for mining high utility itemsets with negative item values in large databases," *Applied Mathematics and Computation*, vol. 215(2), pp. 767778, 2009.

[10] Q. Duong, B. Liao, P. Fournier-Viger, T. L. Dama, "An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies," *Knowledge-Based Systems*, vol. 104, pp. 106-122, 2015.

[11] P. Fournier-Viger, J. C. W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam, "The SPMF Open-Source Data Mining Library Version 2 and beyond," *The European Conference on Machine Learning and Principles and Practice of Knowledge Discovery*, 2016.

[12] G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using fp-trees," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, pp. 1347-1362, 2005.

[13] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: a frequent-pattern tree approach," *Data Mining and Knowledge Discovery*, vol. 8, pp. 53-87, 2004.

[14] T. P. Hong, C. W. Lin, and Y. L. Wu, Incrementally fast updated frequent pattern trees, *Expert Systems with Applications*, vol. 34, pp. 2424-2435, 2008.

[15] T. P. Hong, C. H. Lee, and S. L. Wang, "Effective utility mining with the measure of average utility," *Expert Systems with Application*, vol. 38(7), pp.8259-8265, 2011.

[16] IBM Quest Data Mining Project, Quest Synthetic Data Generation Code, `http://www.almaden.ibm.com/cs/quest/syndata.html`, 1996.

[17] S. Krishnamoorthy, "Pruning strategies for mining high utility itemsets," *Expert Systems with Applications*, vol. 42(5), pp. 2371-2381, 2015.

[18] D. Kim and U. Yun, "Mining high utility itemsets based on the time decaying model," *Intelligent Data Analysis*, vol. 20(5), pp. 1157-1180, 2016.

[19] G. C. Lan, T. P. Hong, and V. S. Tseng, "Efficiently mining high average-utility itemsets with an improved upper-bound strategy," *International Journal of Information Technology & Decision Making*, vol. 11(5), 1009-103, 2012.

[20] C. W. Lin, G. C. Lan, and T. P. Hong, "An incremental mining algorithm for high utility itemsets," *Expert Systems with Applications*, vol. 39(8), pp. 7173-7180, 2012.

[21] C. W. Lin, T. P. Hong and W. H. Lu, "An effective tree structure for mining high utility itemsets," *Expert Systems with Applications*, vol. 38(6), pp. 7419-7424, 2011.

[22] J. C. W. Lin, W. Gan , P. Fournier-Viger, T. P. Hong, V. S. Tseng, "Efficient algorithms for mining high-utility itemsets in uncertain databases," *Knowledge-Based Systems*, vol. 96, pp. 171-187, 2015.

[23] J. C. W. Lin, W. Gan, P. Fournier-Viger, L. Yang, Q. Liu, S. Lukas, M. Voznak, "High utility-itemset mining and privacy-preserving utility mining," *Perspectives in Science*, vol. 7, pp. 74-80, 2015.

[24] J. C. W. Lin, P. Fournier-Viger, W. Gan, "FHN: An efficient algorithm for mining high-utility itemsets with negative unit profits," *Knowledge-Based Systems*, vol. 111, pp. 283-298, 2016.

[25] J. C. W. Lin, W. Gan, P. Fournier-Viger, T. P. Hong, and V. S. Tseng, "Fast algorithms for mining high-utility itemsets with various discount strategies," *Advanced Engineering Informatics*, vol. 30(2), pp. 109-126, 2016.

[26] J. C. W. Lin, W. T. Wu, P. Fournier-Viger, G. Lin, J. Zhan, and M. Voznak, "Fast algorithms for hiding sensitive high-utility itemsets in privacy-preserving utility mining," *Engineering Applications of Artificial Intelligence*, vol. 55, pp. 269-284, 2016.

[27] J. C. W. Lin, W. Gan , P. Fournier-Viger, T. P. Hong, and J. Zhan, "Efficient mining of high-utility itemsets using multiple minimum utility thresholds," *Knowledge-Based Systems*, vol. 113, pp. 100-115, 2016.

[28] J. C. W. Lin, L. Yang, P. Fournier-Viger, J. M. T. Wu, T. P. Hong, Leon S. L. Wang, and J. Zhan, "Mining high-utility itemsets based on particle swarm optimization," *Engineering Applications of Artificial Intelligence*, vol. 5, pp. 320-330, 2016.

[29] Y. C. Lin, C. W. Wu, and V. S. Tseng, "Mining high utility itemsets in big gata," *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 649-661, 2016.

[30] J. C. W. Lin, T. Li, P. Fournier-Viger, T.P. Hong, J. Zhan, M. Voznak, "An efficient algorithm to mine high average-utility itemsets," *Advanced Engineering Informatics*, vol. 30 pp. 233-243, 2016.

[31] Y. Liu, W. K. Liao, and A. Choudhary, "A two-phase algorithm for fast discovery of high utility itemsets," *Lecture Notes in Computer Science*, vol. 3518, pp. 689-695, 2005.

[32] Y. Liu, W. Liao, and A. Choudhary, "A fast high utility itemsets mining algorithm," *The International Workshop on Utility-Based Data Mining*, pp. 90-99, 2005.

[33] M. Liu, J. Qu, "Mining high utility itemsets without candidate generation categories," *International conference on Information and knowledge management*, pp. 55-64, 2012.

[34] T. Lu, B. Vo, H. T. Nguyen, and T. P. Hong, "A new method for mining high average utility itemsets," *The IFIP International Conference on Computer Information Systems and Industrial Management*, pp. 33-42, 2014.

[35] J. Liu, K. Wang, B. Fung, "Mining high utility patterns in one phase without generating candidates," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, pp. 1245-1257, 2016.

[36] J. S. Park, M. Chen, and P. Yu, "An effective hash-based algorithm for mining association rules," *ACM SIGMOD International Conference on Management of Data*, pp. 175-186, 1995.

[37] J. S. Park, M. S. Chen, P. S. Yu, "Using a hash-based method with transaction trimming for mining association rules," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9(5), pp. 812-825, 1997.

[38] R. Srikant and R. Agrawal, "Mining generalized association rules," *The International Conference on Very Large Data Bases*, pp. 407-419, 1995.

[39] V. S. Tseng, B. E. Shie, C. W. Wu, and P. S. Yu, "Efficient algorithms for mining high utility itemsets from transactional databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25(8), pp. 1772-1786, 2013.

[40] V. S. Tseng, C. W. Wu, P. Fournier-Viger, and P. S. Yu, "Efficient algorithms for mining Top-K high utility itemsets," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, pp. 54-67, 2016.

[41] J. M. T. Wu, J. Zhang, and J. C. W. Lin, "Mining of high-utility itemsets by ACO algorithm," *The 3rd Multidisciplinary International Social Networks Conference on SocialInformatics*, Article No. 44, 2016.

[42] H. Yao, H. Hamilton, and C. Butz, "A foundational approach to mining itemset utilities from databases," *SIAM International Conference on Data Mining*, pp. 211-225, 2004.

[43] H. Yao, H. J. Hamilton, L. Geng, "A unified framework for utility based measures for mining itemsets," *The International Workshop on Utility-Based Data Mining*, pp. 28-37, 2006.

[44] U. Yun and H. Ryang, "Incremental high utility pattern mining with static and dynamic databases," *Applied Intelligence*, vol. 42(2), pp. 323-352, 2015.