An Effective Approach for Mining *k*-item High Utility Itemsets from Incremental Databases

Nong Thi Hoa[1](✉) and Nguyen Van Tao[2]

[1] School of Computer Science, Duy Tan University, Da Nang, Vietnam
nongthihoa@duytan.edu.vn

[2] Thai Nguyen University of Information and Communication Technology, Thai Nguyen, Vietnam
nvtao@ictu.edu.vn

Abstract. Mining High Utility Itemset (HUI) from incremental database discovers itemsets making much profit from newest transactions. Therefore, mining HUIs from incremental database are important for planing business. Previous studies on mining exact HUIs consume both time and memory for computing. Therefore, fast algorithms for mining compact HUIs have proposed. However, studies on mining compact HUIs still take a long time and consume much memory because of considering all itemsets of items in a transaction. Moreover, decision making in business is more effective based on HUIs containing several items. In this paper, we propose a novel effective algorithm for mining k-item HUIs that meets the need of decision makers and overcomes the limits of mining compact HUIs. We present a simple list to store k-itemsets appearing during scanning database. This list stores items and utility of each itemset. Our approach perform two ways of database segmentation to mine all k-itemsets. For each way of database segmentation, we run the following algorithm. It consists of two main steps including segmenting the current database to form sub-partitions and mining k-itemsets from each sub-partition. k-item HUIs are extracted from the list based on the utility. The proposed algorithm obtain advantages including without candidate generation and without re-scanning when changing the threshold of utility. Experiments are conducted on dense benchmark databases. Results of experiments show that our algorithm is better than state-of-the-art methods.

Keywords: High utility itemset · HUI · Mining high utility itemset · Mining HUI · k-item HUI · Data mining

1 Introduction

Mining HUIs discovers itemsets whose utility overcome a given threshold. Mining HUIs from incremental databases updates HUIs from newest transactions. Therefore, mining HUIs from incremental databases is essential for activities of

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2021
Published by Springer Nature Switzerland AG 2021. All Rights Reserved
P. Cong Vinh and A. Rakib (Eds.): ICCASA 2021, LNICST 409, pp. 91–104, 2021.
https://doi.org/10.1007/978-3-030-93179-7_8

business. Algorithms of mining HUIs usually consume both time and memory for computing because of generating many itemsets and computing the utility of itemsets. Many studies on mining HUIs from incremental databases have developed to update newest HUIs. These studies can be divided into two categories including using a tree structure and pruning strategies [4–6], and using a list structure and properties of itemset utility [1–3]. In the first category, a tree structure is used to compress transactions in branchs of the tree. As a result, memory for computing decreases. Then, pruning strategies based on properties of HUIs helps remove parts of the tree that contain unpromising itemsets. Therefore, dropping time for tree traversal. In the second one, list structures store information of itemsets containing in transactions. Next, larger itemsets is generated from smaller itemsets based on the same of Transaction Identification. Utilities of larger itemsets is estimated from information of small itemsets generating them. Then, using properties of utility of itemsets to avoid generating larger unpromising itemsets. However, these studies still consume time and memory because there are too many itemsets in large databases.

Recently, several fast algorithms for discovering compact HUIs are proposed to decrease representation of HUIs. These studies use list structures to mined closed HUIs [7–10] and maximal HUIs [7,11]. Properties of compact HUIs is discovered to find compact HUIs more quickly. However, studies on compact HUIs still consume both time of CPU and memory because of processing all items in each transaction. Moreover, planing business is more effectively based on HUIs containing several items. HUIs with a few items is not enough information for making a good decision. HUIs including many items require decision makers to consider the importance of each item. Additionally, a transaction database contains a few of HUIs with many items. In this paper, we propose a novel effective algorithm for mining k-item HUIs that meets the need of business managers and decreases both time and memory by mining on each sub-partition of databases. We present a simple list structure to store the most important information of k-itemsets. This list is updated when scanning both the original database and incremental databases. Our algorithm consists of following steps. First, a vertical segmentation is performed for the current database to form sub-partitions. As a result, rows of each sub-partition contain k items. Next, k-itemsets in each sub-partition are mined and stored in a global list. Then, k-item HUIs is extracted from the global list based on utility of itemsets. Experiments are conducted on benchmark dense databases which have various different characteristics. We compare the performance of our algorithm to state-of-the-art algorithms. Experiment results show that the proposed algorithm significantly decreases both time and memory for computing, and it outperforms compared algorithms.

This paper is organized as follows. Next section is Related work. In Sect. 3, we present our approach for mining k-item HUIs. Section 4 shows experiment results and compares to other studies. Finally, conclusions are written in Sect. 5.

2 Related Works

Studies on mining HUIs focus on decreasing both time and memory for computing. We summarise studies on mining HUIs from incremental databases and

mining compact HUIs to present our motivation. Mining HUIs from incremental databases finds HUIs from newest transactions. Therefore, knowledge from databases is frequently updated to support better for decision makers. Y. Unil and et al. [1] presented an algorithm for mining HUPs with one database scan. They used set of utility lists to store information of candidate patterns. After the global data structure was constructed from the original database or updated from increased data, it was restructured according to a *twu* ascending order. Then, a algorithm mined from a utility list for a promising candidate pattern composed of an item with the smallest *twu* value. The mining process was performed based on HUI-Miner [14]. L. Judae and et al. [4] proposed an approach of pre-large concept to mine high utility patterns (HUPs). This method required only one scan as well as mined in dynamic environments. They used a tree structure (PIHUPL-tree) includes a header and a tail list. After the construction, the header was reordered by a *twu* descending. Mining HUIS was performed by tree traversal. R. Heungmo and Y Unil [5] proposed an algorithm for mining HUPs from data streams. A tree structure (SHU-Tree) was used to maintain information of transactions and HUPs in the current window. The proposed tree was restructured by updating information with overestimation utilities. Y. Unil and H. Ryang [6] proposed algorithm HUPID-Growth (HUPs in Incremental Databases Growth) for mining HUPs. Authors used a HUPID-Tree, and a restructuring method with a TIList (Tail-node Information List). This algorithm composed of three steps. In the first step, a global HUPID-Tree, TIList were constructed with a single database scan. Then, the tree was restructured by arranging nodes in a *twu* descending order. If new transactions were added to the original database, this tree and the TIList were updated. In the second step, candidate patterns were generated from the tree by the HUPID-Growth. All HUPs were identified from the extracted candidates in the last step. L. Chun-Wei and et al. [12,13] presented an algorithm based on pre-large concepts to update discovered HUIs. Itemsets were partitioned into three parts: large, pre-large, or small transaction-weighted utilization in the original database and in inserted transactions. Then, computing procedures were executed for each part. The downward closure property was applied to reduce the number of candidate itemsets. Only a small number of itemsets which were less than a threshold must be rescanned. L. Jerry Chun-Wei and et al. [3] proposed a memory-based incremental approach to build utility list structures for mining HUIs. An Estimated Utility Co-Occurrence Structure (EUCS) was applied to keep the relationship of 2-itemsets and eliminated the extension itemsets with lower utility. P. Fournier-Viger and et al. [2] proposed an algorithm EIHI (Efficient Incremental HUI miner) to maintain HUIs in dynamic databases. This algorithm scanned the database to calculate the *twu* of each item. A second database scan was performed to collect data for the EUCS structure. All itemsets were stored on a trie-like structure (HUI-trie). Each node represented an item and each itemset was represented by a path starting from the tree root and ending by an inner node or a leaf. Then, the depth-first search exploration of itemsets was performed to find HUIs.

Studies on mining compact HUIs have developed to find general HUIs such as closed HUIs, maximal HUIs. Therefore, both time of CPU and memory decrease

by avoiding to consider many itemsets. C. W. Wu and et al. [10] proposed the EU-List (Extended Utility-List) to maintain and calculate the utility of itemsets without scanning the original database. Next, a novel algorithm, CHUI-Miner (Closed HUI Miner), adopted the divide-and-conquer methodology to mine the complete set of close HUIs without producing candidates. For each closed itemset X, it used EU-List to calculate its utility to determine whether it was a closed HUI. Property of remaining utility was used to prune the search space. P. Fournier-Viger and et al. [9] presented EFIM-Closed (EFficient HUI Mining - Closed) based on the strict constraint of each itemset in the search space. EFIM-Closed proposed three strategies to discover close HUIs: closure jumping, forward closure checking, and backward closure checking. EFIM-Closed reduced the cost of database scans based on techniques: High-utility Database Projection and High utility Transaction Merging. Moreover, it used two new upper-bounds on the utility of itemsets named sub-tree utility and local utility to effectively prune the search space, and applied an efficient Fast Utility Counting technique to compute utility of itemsets. P. Fournier-Viger and et al. [7] proposed a novel algorithm named CHUI-Mine (Compact HUI Miner) to discover closed HUIs and maximal HUIs. It was a one-phase algorithm and discovered representations without generating candidates. Authors used the proposed PUDC (Pivot Utility Downward Closure) property to prune the search space. Moreover, an efficient algorithm, RHUI (Recover all HUIs from maximal patterns), was presented to recover all HUIs and their exact utilities from the set of maximal HUIs. This algorithm reused EU list in [10] during mining closed HUIs or maximal HUIs. N.T.T. Loan and et al. [11] presented optimized versions of CHUI-Mine [7] to mine maximal HUIs using P-Set, Estimated Utility Co-occurrence Pruning (EUCP), and First Utility Co-occurrence Structure (FUCS). P-set of itemset X listed identification of transactions containing X. EUCP pruned candidate itemsets based on the EUCS structure. FUCS was a structure to store sum of twu of itemsets containing two items. D. Thu-Lan and et al. [8] presented IncCHUI that mined close HUIs efficiently from incremental databases. They used an utility list structure that built and updated from one database scan. Next, pruning strategies was applied to increase the speed of construction of utility lists and eliminate candidates. Then, they suggested an hash based method to update or insert new closed sets. Previous studies show an idea approach for mining HUIs need obtain advantages including without generating candidates, without re-scanning database to compute the utility of itemsets. However, these studies still consume both time of CPU and memory because of processing all items in each transaction. Therefore, we propose a novel effective algorithm that obtains these advantages and overcome this limit.

3 Our Approach

Problem Statement: Given a DB, number of items of each itemset k, and an utility threshold $minutil$, mining k-item HUIs from DB discovers all k-itemsets whose utilities are greater than or equal to $minutil$.

Our idea is discovered from knowledge of process of making business decision, observation of recommendation systems, and properties of transaction databases. We see that decision makers plan business more effectively based on HUIs containing several items. HUIs with a few items are not enough information for decisions. HUIs including many items require decision makers to consider the importance of each items. Moreover, big recommendation systems usually show from 3 to 6 related items of the current item. Additionally, most HUIs in transaction databases contain several items and a few HUIs have many items. Therefore, we propose a fast effective algorithm for mining k-item HUIs with small number of items. The proposed algorithm is called inc-k-HUI-Miner (mining k-item HUIs from Incremental databases). We introduce a simple list to store k-itemsets containing in databases. This list is called Items-Utility list (IU list). Our algorithm perform on two ways of database segmentation. For each way of database segmentation, we run the following process. This process includes two main steps. In the first step, a vertical segmentation of the current database is performed to form sub-partitions (SPs) based on the value of k. Each row in a SP contains k items. In the second one, k-itemsets is listed from each SP and stores them in a global IU list. k-item HUIs are extracted from the global IU list based on the utility of itemsets. We explain our approach more detail in next subsections.

3.1 Structure of the IU List

We propose the IU list to store k-itemsets containing in the current database. Structure of this list includes *items, utility* of itemsets. Table 1 shows an example of the IU list.

Table 1. An example of the IU list with $k = 5$

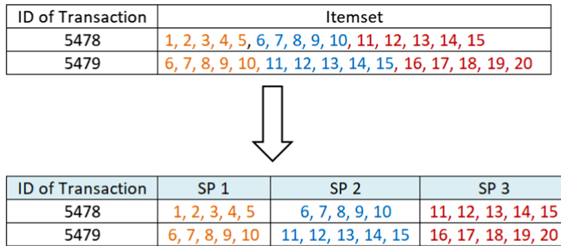
Items	Utility
1, 2, 3, 5, 6	10309
4, 5, 6, 7, 8	16778

The IU list is added or updated during browsing transactions. When a new itemset appears, a new row is added. If an itemset exists then updating its utility. Structure of the IU list is simple and only contains two important information for mining HUIs. Therefore, the IU list uses a small capacity of memory and helps to decrease time for searching existing itemsets.

3.2 The Proposed Algorithm for Mining *k*-item HUIs

Firstly, we perform a vertical segmentation for the current database. This database is divided into SPs whose rows contain k items. As a result, items in a transaction belong to many SPs. If number of items of the last SP is not

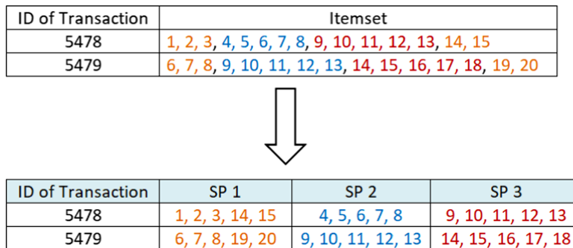
enough k items then adding default items. For example, adding item 0 to last SP. Database segmentation is done two times to find all k-item HUIs. In the first way for segmenting database, items in 1..k belong to SP 1 and items in k+1..2*k belong to SP 2, ... Do until the last item is assigned for the last SP. Similarly, the second way for segmenting database is done from $k/2$-th item if k is a even number. Meaning, $k/2..k/2 + k$ belong to SP 1 and $k/2 + k$+1..$k/2$ + 2 * k belong to SP 2. Do until the last item is assigned for the last SP. If k is a odd number then SP 1 start (k+1)/2-th. As a result, k-itemsets appearing between two adjacent SPs of the first way are mined by running the process listing k-itemsets with the second way. Assuming that we choose k = 5 for an example database. The first way for segmenting database is done in Fig. 1 and Fig. 2 shows the second one of segmenting database.



ID of Transaction	Itemset
5478	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
5479	6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

ID of Transaction	SP 1	SP 2	SP 3
5478	1, 2, 3, 4, 5	6, 7, 8, 9, 10	11, 12, 13, 14, 15
5479	6, 7, 8, 9, 10	11, 12, 13, 14, 15	16, 17, 18, 19, 20

Fig. 1. The first way for segmenting database with $k = 5$.



ID of Transaction	Itemset
5478	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
5479	6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

ID of Transaction	SP 1	SP 2	SP 3
5478	1, 2, 3, 14, 15	4, 5, 6, 7, 8	9, 10, 11, 12, 13
5479	6, 7, 8, 19, 20	9, 10, 11, 12, 13	14, 15, 16, 17, 18

Fig. 2. The second way for segmenting database with $k = 5$.

Users assign a value of k to meet the need of making decisions. Choosing a proper value of k helps improve the performance of mining k-item HUIs. We observe big recommendation systems in retail. We see that an item usually links to from 3 to 6 related items. Therefore, we recommend values of k that is from 4 to 7.

The top-down strategy is applied for exploring transactions in each SP. Our algorithm finds an k-itemset that appears in adjacent transactions. Assuming, a

k-itemset X appears in transactions from m to n. We formulate the utility of X in transactions from m to n and store in an available. Then, searching X from the IU list. If X exists then updating the utility of X. Otherwise, adding a new row to the IU list to store X. Repeat this process until browsing the last row of each SP. We only use a single IU list. As a result, an IU list stores itemsets appearing in all SPs of two ways of database segmentation.

When adding new transactions from an incremental database, the IU list is continuously added or updated. Meaning, we combine data of both the original database and incremental databases in an IU list. Therefore, inc-k-HUI-Miner still finds new HUIs without re-scanning the original database. Moreover, the proposed algorithm still outputs HUIs without re-scanning databases when changing *minutil*. Similarly, mining HUIs from a incremental database is done.

Process of mining k-item HUIs by inc-k-HUI-Miner is presented by Algorithm 1 and Algorithm 2. Algorithm 1 shows the method that are applied for a way of database segmentation. Algorithm 2 shows the performance of the proposed algorithm for mining k-item HUIs. Steps of Algorithm 1 is described as follow

Algorithm 1: Mining *k*-itemsets
Input: Given a way of database segmentation
DB - transaction database
P - a sub-partition, **L** - the IU list, **T** - a transaction
X - an itemset appearing in adjacent transactions
Y - an itemset appearing in the current row
Output: k-itemsets in the global IU list **L**.

```
1:  Divide DB into SPs .
2:  for each P in DB do
3:      Set X to the itemset in the first transaction of P
4:      for each T ∈ P do
5:        if X = Y then
6:          Update the utility of X
7:        end if
8:        if X ≠ Y then
9:          if X ∉ L then
10:           Add X to L
11:         else
12:           Update the utility of X in L.
13:         end if
14:       end if
15:       X=Y
```

Our approach for mining k-item HUIs is shown in Algorithm 2. Two ways for database segmentation is done to overlap SPs of the first way and SPs of the second one. A single IU list is used to store k-itemsets during mining k-item HUIs.

Algorithm 2: inc-k-HUI-Miner
Input: Given a database DB
$minutil$ - the threshold of utility
k - the number of items of an itemset
M_1,M_2: Two ways of database segmentation
L - the IU list that obtain from the Algorithm 1
Z - an itemset appearing in **L**
Output: k-item HUIs.
1: Mining k-itemsets with parameter k according to M_1.
2: Mining k-itemsets with parameter k according to M_2.
3: for each $\mathbf{Z} \in L$ do
4: if $\mathbf{Z}$.Utility $\geq minutil$ then
5: $\mathbf{Z}$ is a k-item HUI.
6: end if

3.3 Discussion

Our approach obtains following advantages

- Mine on a small part of databases to drop memory
- Avoid generation of candidate itemsets.
- Mine new HUIs without re-scanning databases when changing $minutil$.
- Propose a simple list to decrease memory and time for searching elements.
- Apply for both a original database and incremental databases.
- Be simple to understand and perform.

As a result, inc-k-HUI-Miner significantly decreases both time and memory for computing. We conduct experiments to prove the effectiveness of our algorithm.

4 Experiments

Experiments are conducted to measure influence of the utility threshold, influence of the insertion ratio, and scalability. Our approach is compared to state-of-the-art algorithms including EFIM-Closed [9], and CHUI-Miner, CHUI-MinerMax [10].

4.1 Experiment Setup

We performed experiments on dense benchmark databases having various characteristics. Three databases were used in experiments including *mushroom* (8124 transactions, 119 items), *chess* (3196 transactions, 75 items), *connect* (67557 transactions, 129 items). They were downloaded from FIMI Repository [15]. Most databases do not provide item utility and item count for each transaction. Like some previous studies [8], external utilities for items are generated between 0.01 and 10 by using a log-normal distribution and internal utilities for items are randomly generated from 1 to 10.

All algorithms were implemented in Java, where EFIM-Closed, CHUI-Miner, CHUI-MinerMax were obtained from SPMF [16]. We conducted experiments on a computer equipped with a 64 bit, Core i3 (2 GHz x 2 GHz) Intel Processor, 4 GB of main memory, and running Windows 10. Each data is an average of 5 experiment results. Time of CPU consists of time for inputting transactions and mining HUIs.

4.2 Performance of inc-k-HUI-Miner

Experiments measure influence of the utility threshold, influence of the insertion ratio, and scalability on *chess* and *connect* to evaluate the effectiveness of the proposed algorithm.

Results on *chess*
Influence of the Utility Threshold: We select values of *minutil* including 50, 100, 150, 200, 250 because *chess* is a small database. Value of k is greater, time and memory for computing is smaller. We choose $k = 5$ to obtain higher values of time and memory for computing. Number of 5-itemsets of *chess* is 260. Table 2 presents the influence of the utility threshold on *chess*. Data from Table 2 show the maximal time is 1409 (ms) and memory consumption is stable at 1.66 (MB).

Table 2. Influence of changing *minutil* on *chess*

minutil	#5-item HUI	Time of CPU (ms)	Memory (MB)
250	216	1388.8	1.65912
200	225	1409.6	1.65915
150	235	1360.4	1.65912
100	244	1359.0	1.65915
50	258	1386.6	1.65912

Influence of the Insertion Ratio: We setup *minutil* = 100 and $k = 5$ to run experiments. Selected insertion ratios are 10%, 15 %, 20%, and 25%. Data of each insertion ratio are different because we select 10% first transactions, 15% next ones, 20% next ones, and 25% next ones. Meaning, 70% first transactions are used. For *chess*, 300 transactions are 10% of transactions. Table 3 presents the influence of the insertion ratio on *chess*. Data in Table 3 show time of CPU increases from 83.3 (ms) to 286.7 (ms) and memory for computing is stable at 1.66 (MB).

Scalability Tests: We setup *minutil* = 100 and $k = 5$ to run experiments. Number of transactions are 20%, 40%, 60%, and 80%. For *chess*, selected transactions are from 1 to 600, 1200, 1800, 2400. Table 4 presents the scalability tests on *chess*. Data in Table 4 show values of time are from 140.7 (ms) to 879.7 (ms) and memory is about 1.66 (MB).

We see that the ratio of k-item HUIs and k-itemsets is high in all sub-tests. Memory of our algorithm is stable because it store k-itemsets of the current

Table 3. Influence of the insertion ratio on *chess* with $minutil = 100$

Insertion ratio	#5-item HUI	#5-itemset	Time of CPU (ms)	Memory (MB)
10%	86	93	83.3	1.65912
15%	102	113	125.0	1.65912
20%	137	149	171.7	1.65912
25%	185	201	286.7	1.65912

Table 4. Scalability tests on *chess* with $minutil = 100$

Scalability	#5-item HUI	#5-itemset	Time of CPU (ms)	Memory (MB)
20%	109	118	140.7	1.65912
40%	147	158	328.0	1.65912
60%	181	194	536.7	1.65912
80%	217	227	879.7	1.65912

database. Moreover, time and memory for computing are smaller many times than performance of modern computers.

Results on *connect*
Influence of the Utility Threshold: We select values of *minutil* including 100, 150, 200, 250, 300 on *connect*. We choose $k = 5$ and small values of *minutil* to obtain higher values of time and memory for computing. The newest transactions of *connect* are used to run experiments which is in from 50000 to 67557. With $k = 5$, number of 5-itemsets of *connect* is 691. Table 5 presents the influence of the utility threshold on *connect*. Data from Table 5 show the maximal time is 25187 (ms) and memory is stable at 26.85 (MB).

Table 5. Influence of changing *minutil* on *connect*

minutil	#5-item HUI	Time of CPU (ms)	Memory (MB)
300	564	25067.8	26.84734
250	579	25187.2	26.84730
200	590	25217.6	26.84741
150	620	24900.4	26.84730
100	645	25136.4	26.84742

Influence of the Insertion Ratio: We setup $minutil = 100$ and $k = 5$ to run experiments. Selected insertion ratios are 10%, 15 %, 20%, and 25%. 2000 transactions are 10% of *connect* (running from the 50000-th transaction). Table 6 presents the influence of the insertion ratio on *connect*. Data in Table 6 show time of CPU increases from 1453 (ms) to 5566.3 (ms) and memory is about 26.85 (MB).

Table 6. Influence of the insertion ratio on *connect* with $minutil$ = 100

Insertion ratio	#5-item HUI	#5-itemset	Time of CPU (ms)	Memory (MB)
10%	274	302	1453.0	26.84735
15%	294	336	2374.7	26.84735
20%	397	444	3967.7	26.84735
25%	481	536	5566.3	26.84742

Scalability Tests: We setup $minutil = 100$ and $k = 5$ to run experiments. Number of transactions are 20%, 40%, 60%, and 80%. For *connect*, they are 50001..54000, 50001..58000, 50001..62000, 50001..66000. Table 7 presents the scalability on *connect*. Data in Table 7 show values of time are from 3447.3 (ms) to 19604.3 (ms) and memory is about 26.85 (MB).

Table 7. Scalability tests on *connect* with $minutil$ = 100

Scalability	#5-item HUI	#5-itemset	Time of CPU (ms)	Memory (MB)
20%	337	376	3447.3	26.84735
40%	415	462	7586.7	26.84742
60%	546	611	13814.0	26.84742
80%	609	663	19604.3	26.84742

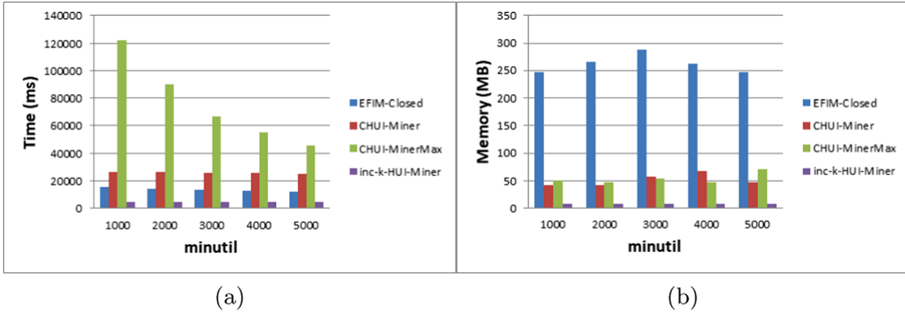
We also see that the ratio of k-item HUIs and k-itemsets is high and memory of the proposed algorithm is stable. Running on a computer equipped with a 64 bit, Core i3 Intel Processor, 4 GB of main memory, the maximal time on *connect* is 19.6 (s) and 26.85 (MB) is the highest memory. Time and memory of our approach are much smaller than performance of modern computers. Therefore, our approach is suitable for online analytical processing.

4.3 Comparing to the State-of-the-Art Methods

Values of $minutil$ for inc-k-HUI-Miner are selected as follow. Values of $minutil$ are selected to reach a high ratio of number of k-item HUIs and number of k-itemsets. As a result, time and memory of the proposed algorithm reach to the largest values. Experiments are conducted on *mushroom* to limit the running time.

Influence of Changing the $minutil$
We setup $k = 5$. $minutil$ for inc-k-HUI-Miner is lower 10 times than compared algorithms to obtain a large number of k-item HUIs. We select values of $minutil$ for EFIM-Closed, CHUI-Miner, and CHUI-MinerMax including 1000, 2000, 3000, 4000, 5000. Values of $minutil$ in our algorithm are 100, 200, 300, 400,

500 respectively. Figure 3 presents the time of CPU and memory consumption of *mushroom* when changing *minutil*. Figure 3(a) shows time of the proposed algorithm is lowest. EFIM-Closed and CHUI-Miner are the second and third one. The last one is CHUI-MinerMax. inc-k-HUI-Miner's time is lower 2.5 times than the second algorithm. Data from Fig. 3(b) show memory of our approach is smallest. It is lower 5 times than CHUI-Miner (the second one).

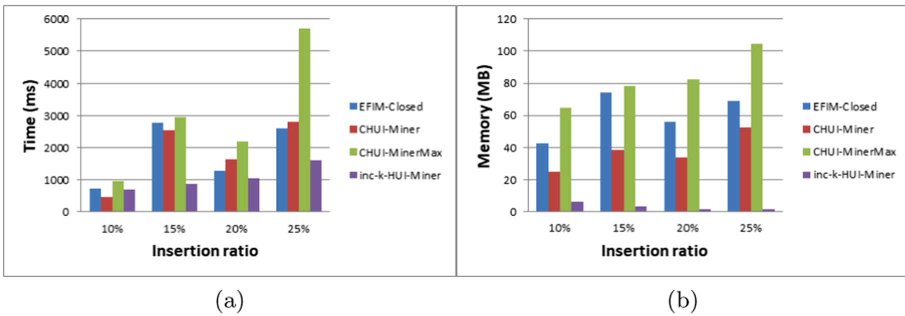


(a) (b)

Fig. 3. Results of *mushroom* when changing *minutil*

Influence of the Insertion Ratio

We setup *minutil* $= 100$ and $k = 5$ to run experiments. Selected insertion ratios are 10%, 15 %, 20%, and 25%. Data of each insertion ratio are different. For *mushroom*, 800 transactions are 10% of transactions. Figure 4 presents CPU's time and memory consumption of algorithms to measure the influence of the insertion ratio. Data from Fig. 4(a) show that inc-k-HUI-Miner is the best algorithm. The last one is CHUI-MinerMax. Similarly, our approach is better than compared methods in Fig. 4(b). Memory of the proposed algorithm is sharply lower than CHUI-Miner (the second one).

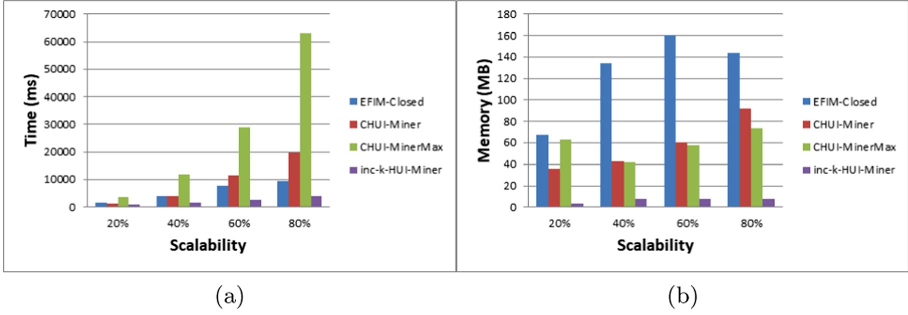


(a) (b)

Fig. 4. Results of *mushroom* when changing the insertion ratio

Scalability Tests

We setup *minutil* = 100 and $k = 5$ to run experiments. Number of transactions are 20%, 40%, 60%, and 80%. For *mushroom*, data are selected from the first transaction to 1600, 3200, 4800, 6400 respectively. Figure 5 presents CPU's time and memory consumption of algorithms on scalability tests. Data from Fig. 5 show that both time and memory of the proposed algorithm is better than compared methods. Especially, both time and memory of our algorithm sharply drop.



(a) (b)

Fig. 5. Results of *mushroom* for scalability tests

Results of experiments show that our approach is better than state-of-the-art methods. Both time and memory of the proposed are smaller many times than both hardware and performance of modern computers. Therefore, our algorithm is effective for applications running online analytic processing.

5 Conclusions

In this paper, we propose a novel effective algorithm for mining k-item HUIs that meets the need of business activities and improves the performance of mining HUIs. We use a simple list structure to store important information of k-itemsets during browsing transactions. The proposed algorithm perform two times of a process which finds k-itemsets from databases. This process includes two main steps. First, the current database is vertically segmented into sub-partitions. Then, k-itemsets are mined from each sub-partition. All k-itemsets are stored in a global list. k-item HUIs is extracted from the global list based utility of each itemset. Our approach obtains advantages including without candidate generation, without re-scanning database when changing the utility threshold. Experiments are conducted on dense benchmark databases including *mushroom*, *chess*, and *connect*. Experiment results show that our algorithm is better than state-of-the-art methods. Both time and memory of the proposed sharply decrease.

We will investigate to optimize the time for computing and conduct experiments on larger databases in the future.

References

1. Unil, Y., Ryang, H., Gangin, L., Fujita, H.: An efficient algorithm for mining high utility patterns from incremental databases with one database scan. Knowl.-Based Syst. **124**, 188–206 (2017)
2. Fournier-Viger, P., Jerry, L.C., Gueniche, T., Barhate, P.: Efficient incremental high utility itemset mining. In: Proceedings of 5th ASE International Conference on Big Data (2015)
3. Lin, J.C.-W., Gan, W., Hong, T.-P., Pan, J.-S.: Incrementally updating high-utility itemsets with transaction insertion. In: Luo, X., Yu, J.X., Li, Z. (eds.) ADMA 2014. LNCS (LNAI), vol. 8933, pp. 44–56. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14717-8_4
4. Judae, L., Unil Yun, Y., Gangin, L., Eunchul, Y.: Efficient incremental high utility pattern mining based on pre-large concept. Eng. Appl. Artif. Intell. **72**, 111–123 (2018)
5. Heungmo, R., Unil, Y.: High utility pattern mining over data streams with sliding window technique. Expert Syst. Appl. **57**, 214–231 (2016)
6. Yun, U., Ryang, H.: Incremental high utility pattern mining with static and dynamic databases. Appl. Intell. **42**(2), 323–352 (2014). https://doi.org/10.1007/s10489-014-0601-6
7. Wu, C.-W., Fournier-Viger, P., Gu, J.-Y., Tseng, V.S.: Mining compact high utility itemsets without candidate generation. In: Fournier-Viger, P., Lin, J.C.-W., Nkambou, R., Vo, B., Tseng, V.S. (eds.) High-Utility Pattern Mining. SBD, vol. 51, pp. 279–302. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-04921-8_11
8. Thu-Lan, D., Heri, R., Kjetil, N., Quang-Huy, D.: Towards efficiently mining closed high utility itemsets from incremental databases. Knowl.-Based Syst. **165**, 13–29 (2019)
9. Fournier-Viger, P., Zida, S., Lin, J.C.-W., Wu, C.-W., Tseng, V.S.: EFIM-closed: fast and memory efficient discovery of closed high-utility itemsets. In: MLDM 2016. LNCS (LNAI), vol. 9729, pp. 199–213. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41920-6_15
10. Wu, C.W., Fournier-Viger, P., Gu, J.Y., Tseng, V.S.: Mining closed+ high utility itemsets without candidate generation. In: Proceedings of Conference on Technologies and Applications of Artificial Intelligence (TAAI), pp. 187–194 (2015)
11. Loan, N.T.T., Bao, V.D., Trinh, N.D.D., Bay, V.: Mining maximal high utility itemsets on dynamic profit databases. Cybern. Syst. **51**(2), 140–160 (2020)
12. Chun-Wei, L., Wensheng, G., Tzung-Pei, H., Binbin, Z.: An incremental high-utility mining algorithm with transaction insertion. Sci. World J. (2015)
13. Lin, C.-W., Hong, T.-P., Lan, G.-C., Wong, J.-W., Lin, W.-Y.: Incrementally mining high utility patterns based on pre-large concept. Appl. Intell. **40**(2), 343–357 (2013). https://doi.org/10.1007/s10489-013-0467-z
14. Liu, J., Wang, K., Fung, B.: Direct discovery of high utility itemsets without candidate generation. In: International Conference on Data Mining, pp. 984–989 (2012)
15. Frequent Itemset Mining Dataset Repository (2012). http://fimi.ua.ac.be/
16. Fournier-Viger, P., Gomariz, A., Soltani, A., Lam, H., Gueniche, T.: Spmf: Open-source data mining platform (2014). http://www.philippe-fournier-viger.com/spmf