Mining Top-K High Utility Itemsets

Cheng Wei Wu[1], Bai-En Shie[1], Philip S. Yu[2], Vincent S. Tseng[1]

[1]Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan, ROC

[2]Department of Computer Science, University of Illinois at Chicago, Chicago, Illinois, USA

{silvemoonfox, brianshie}@gmail.com, psyu@cs.uic.edu, tsengsm@mail.ncku.edu.tw

ABSTRACT

Mining high utility itemsets from databases is an emerging topic in data mining, which refers to the discovery of itemsets with utilities higher than a user-specified minimum utility threshold *min_util*. Although several studies have been carried out on this topic, setting an appropriate minimum utility threshold is a difficult problem for users. If *min_util* is set too low, too many high utility itemsets will be generated, which may cause the mining algorithms to become inefficient or even run out of memory. On the other hand, if *min_util* is set too high, no high utility itemset will be found. Setting appropriate minimum utility thresholds by trial and error is a tedious process for users. In this paper, we address this problem by proposing a new framework named *top-k high utility itemset mining*, where *k* is the desired number of high utility itemsets to be mined. An efficient algorithm named *TKU* (*Top-K Utility itemsets mining*) is proposed for mining such itemsets without setting *min_util*. Several features were designed in *TKU* to solve the new challenges raised in this problem, like the absence of anti-monotone property and the requirement of lossless results. Moreover, *TKU* incorporates several novel strategies for pruning the search space to achieve high efficiency. Results on real and synthetic datasets show that *TKU* has excellent performance and scalability.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications — Data Mining

General Terms: Algorithms, Performance

Keywords: Utility mining, high utility itemset, top-k pattern mining

1. INTRODUCTION

Frequent itemset mining (abbreviated as FIM) [1, 8] is a fundamental research topic in data mining. However, the traditional model of FIM may discover a large amount of frequent but low revenue itemsets and lose the information on valuable itemsets having low selling frequencies. Hence, FIM cannot satisfy the requirement of users who desire to discover itemsets with high utilities such as high profits. To address these issues, utility mining [2, 3, 6, 11, 12, 13, 18, 19, 20, 21, 23, 24, 25] emerges as an important topic in data mining. In utility mining, each item has a weight (e.g. unit profit) and can appear more than once in each transaction (e.g. purchase quantity). The utility of an itemset represents its importance, which can be measured in terms of weight, profit, cost, quantity or other information depending on the user preference. An itemset is called a *high utility itemset* (abbreviated as HUI) if its utility is no less than a user-specified *minimum utility threshold*. Utility mining is an important task and has a wide range of applications such as website click stream analysis [2, 11, 18, 20, 24], cross-marketing in retail stores [6, 12, 13, 19, 21, 23, 25] and biomedical applications [3].

Although this framework is essential to many applications, mining high utility itemsets is not an easy task because the *downward closure property* [1] does not hold. To facilitate the task of high utility itemset mining, most approaches [2, 11, 12, 21] utilize the *TWU* model and *TWDC* property to prune the search space. In this model, an itemset is called *HTWUI* if its *TWU* is no less than *min_util*, where the *TWU* of an itemset represents the upper bound of its utility. The TWDC property states that for any itemset that is not an HTWUI, all its supersets are low utility itemsets. The TWU-model consists of two phases named *phase I* and *phase II*. In *phase I*, all the HTWUIs are found. In phase II, the exact utilities of HTWUIs are calculated by scanning the database.

Although many studies have devoted to HUI mining, it is difficult for users to choose an appropriate minimum utility threshold in practice. Depending on the threshold, the output size can be very small or very large. Besides, the choice of the threshold also greatly influences the performance of the algorithms. If the threshold is set too low, too many high utility itemsets will be presented to the users. It is difficult for the users to comprehend the results. A large number of high utility itemsets also causes the mining algorithms to become inefficient or even run out of memory, because the more high utility itemsets the algorithms generate, the more resources they consume. On the contrary, if the threshold is set too high, no high utility itemset will be found. In this case, users need to try different thresholds by guessing and re-executing the algorithms over and over until being satisfied with the results. This process is both inconvenient and time-consuming.

We illustrate the problem of setting the minimum utility threshold with a real shopping transaction database named *Chainsto*re. Figure 1 shows the runtime and the number of high utility itemsets in Chainstore dataset of the state-of-the-art utility mining algorithm UP-Growth [19]. As it can be seen, the choice of *min_util* has a major impact on the output size even if it is just changed slightly. For example, consider the case of a user who is interested in finding the top 1000 itemsets that contribute the highest profits in the Chainstore dataset. If the user does not possess the background knowledge about the database for setting *min_util* (he needs to make a guess to choose the threshold), he has only a very small chance of selecting a *min_util* that will satisfy his requirements (he would need to set *min_util* between 0.02% and 0.03%). Moreover, if the threshold is set below 0.02%, the algorithm can take up to one hour before terminating on a typical desktop computer.
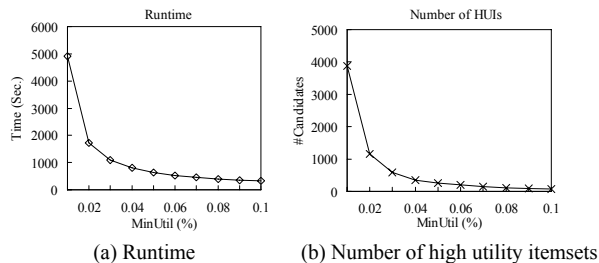
Figure 1. Runtime and number of high utility itemsets in Chainstore dataset under varied minimum utility thresholds

A similar problem occurring in FIM is how to determine an appropriate *minimum support threshold* to mine enough but not too many itemsets for the users. To precisely control the output size and discover the most frequent patterns without setting the threshold, a good solution is to change the task of mining frequent patterns to the task of mining the top-*k* frequent patterns [4, 5, 7, 9, 10, 14, 16, 17, 22]. The idea is to let the users specify *k*, i.e., the number of desired patterns, instead of specifying the minimum support threshold. Setting *k* is more intuitive than setting the threshold because *k* represents the number of itemsets that the user wants to find whereas choosing the threshold depends solely on database's characteristics, which are often unknown to users.

Although using a parameter *k* instead of a threshold would also be desirable in utility mining, developing an efficient algorithm for mining top-*k* high utility itemsets is not an easy task. It poses four major challenges as discussed below.

First, the utility of an itemset is neither monotone nor anti-monotone. In other words, the utility of an itemset may be equal to, higher or lower than that of its supersets and subsets. Therefore, many techniques [5, 7, 9, 10, 14, 16, 17, 22] developed in top-*k* frequent pattern mining that rely on anti-monotonicity to prune the search space cannot be directly applied to top-*k* high utility itemset mining.

The second challenge is how to incorporate the concept of top-*k* pattern mining with the TWU-model. Although the TWU-model is widely used in utility mining, it is difficult to adapt this model to top-*k* high utility itemset mining because the exact utilities of itemsets are unknown in phase I. When an HTWUI is generated in phase I, we cannot guarantee that its utility is higher than other HTWUIs and it is a top-*k* high utility itemset before performing phase II. To guarantee that all the top-*k* high utility itemsets can be captured in the set of HTWUIs, a naive approach is to run the algorithm with *min_util* = 0. However, this approach may encounter the large search space problem.

The third challenge is that *min_util* is not given in advance in top-*k* high utility itemset mining. In the traditional high utility itemset mining, the search space can be efficiently pruned by the algorithms with a given *min_util*. However, in the scenario of top-*k* high utility mining, the threshold is not provided. Therefore, the minimum utility threshold is initially set to 0. The mining task has to gradually raise the threshold to prune the search space. Thus the challenge is to design an algorithm that can raise the threshold as high as possible and make the number of candidates produced in phase I as small as possible.

The last challenge is how to effectively raise the threshold without missing any top-*k* high utility itemsets. A good algorithm is one that can effectively raise the threshold during the mining process. However, if an incorrect method for raising the threshold is used, it may result in some top-*k* high utility itemsets being pruned. Thus, how to effectively raise the threshold without missing any top-*k* high utility itemsets is a crucial challenge for this work.

In this paper, we address all of the above challenges by proposing an efficient algorithm named *TKU* for *Top-K U*tility itemset mining. This work has three major contributions.

First, we propose a novel framework for mining top-*k* high utility itemsets. An algorithm named *TKU* is proposed for efficiently mining the complete set of top-*k* high utility itemsets in the database without specifying *min_util* threshold.

Second, five new strategies are proposed for effectively raising the threshold at different stage of the mining process. The first four strategies effectively raise the threshold during the mining process to prune the search space and reduce the number of candidates in phase I. The last strategy effectively reduces the number of candidates that need to be checked in phase II. It improves the runtime of phase II and the overall performance.

Third, we conducted different kinds of experiments with real datasets. The results show that the performance of the proposed algorithm TKU is close to that of the optimal case of the state-of-the-art utility mining algorithm UP-Growth [19]. Moreover, it is over 100 times faster than the compared baseline algorithm.

The remainder of this paper is organized as follows. In Section 2, we introduce the background for utility mining and top-*k* pattern mining. Section 3 presents the proposed methods. Experiments are shown in Section 4 and conclusions are given in Section 5.

Table 1. An example database

TID	Transaction	TU
T_1	(A,1) (C,1) (D,1)	8
T_2	(A,2) (C,6) (E,2) (G,5)	27
T_3	(A,1) (B,2) (C,1) (D,6) (E,1) (F,5)	30
T_4	(B,4) (C,3) (D,3) (E,1)	20
T_5	(B,2) (C,2) (E,1) (G,2)	11

Table 2. Profit table

Item	A	B	C	D	E	F	G
Profit	5	2	1	2	3	1	1

2. BACKGROUND

This section introduces the preliminaries related to utility mining, and then defines the problem statement of top-*k* high utility itemset mining. We adopt the notations used in [19]. For more details about high utility itemsets, readers can refer to [19].

2.1 Problem definition

Given a finite set of distinct *items* $I = \{i_1, i_2, \ldots, i_m\}$. Each item $i_j \in I$ is associated with a positive number $p(i_j, D)$, called its *external utility*. A *transactional database* $D = \{T_1, T_2, \ldots, T_n\}$ is a set of transactions, where each transaction $T_c \in D$, $(1 \le c \le n)$ is a subset of *I* and has an unique identifier *c*, called *Tid*. In transaction T_c, each item i_j is associated with a positive number $q(i_j, T_c)$, called its *internal utility* in T_c. An itemset $X = \{i_1, i_2, \ldots, i_l\}$ is a set of *l* distinct items, where $i_j \in I$, $1 \le j \le l$, and *l* is the length of *X*. A *l*-itemset is an itemset of length *l*. An itemset *X* is said to be *contained* in a transaction T_c if $X \subseteq T_c$.

Definition 1. The *support count* of an itemset *X* is the number of transactions containing *X* in *D* and denoted as *SC(X)*. The *support of X* is defined as the ratio of *SC(X)* to |*D*|.

Definition 2. The *utility of an item* i_p *in a transaction* T_c is denoted as $u(i_j, T_c)$ and defined as $p(i_j, D) \times q(i_j, T_c)$.

Definition 3. The *utility of an itemset* X *in a transaction* T_c is denoted and defined as $u(X, T_c) = \sum_{i_j \in X} u(i_j, T_c)$.

Definition 4. The *utility of an itemset* X *in* D is denoted and defined as $u(X) = \sum_{X \subseteq T_c \wedge T_c \in D} u(X, T_c)$.

Definition 5. An itemset X is called *high utility itemset* if $u(X)$ is no less than a user-specified *minimum utility threshold min_util*.

Definition 6. Let *min_util* be the minimum utility threshold, the complete set of high utility itemsets in D is denoted as $f_H(D, min\_util)$. The goal of high utility itemset mining is to discover $f_H(D, min\_util)$.

Example 1. Let Table 1 be an example database containing five transactions. Each row in Table 1 represents a transaction, in which each letter represents an item and has a purchase quantity (internal utility). The unit profit of each item is shown in Table 2 (external utility). Suppose *min_util* is set to 30, the set of high utility itemsets in Table 1 is {{BD}:30, {ACE}:31, {BCD}:34, {BCE}:31, {BDE}:36, {BCDE}:40, {ABCDEF}:30}, where the number beside each itemset is its utility.

Note that the utility constraint is neither *monotone* nor *anti-monotone*. In other words, the utility of an itemset may be equal to, higher or lower than that of its supersets and subsets. Therefore, we cannot directly use the anti-monotone property (also known as downward closure property) to prune the search space. To facilitate the mining task, Liu et al. introduced the concept of *transaction-weighted downward closure* [12], which is based on the following definitions.

Definition 7. The *transaction utility* of a transaction T_R is denoted as $TU(T_R)$ and defined as $u(T_R, T_R)$.

Definition 8. The *transaction-weighted utilization* of an itemset X is the sum of the transaction utilities of all the transactions containing X, which is denoted as $TWU(X)$ and defined as $TWU(X) = \sum_{X \subseteq T_R \wedge T_R \in D} TU(T_R)$.

Definition 9. An itemset X is a *high transaction-weighted utilization itemset* (abbreviated as HTWUI) if $TWU(X) \geq min\_util$.

Property 1. (TWDC property) The *transaction-weighted downward closure* property states that for any itemset X that is not a HTWUI, all its supersets are low utility itemsets [12].

Definition 10. (Top-k high utility itemset) An itemset X is called a *top-k high utility itemset* in a database D if there are less than k itemsets whose utilities are larger than $u(X)$ in $f_H(D, 0)$.

Property 2. Let H be the complete set of top-k high utility itemsets in D. H may contain less than k high utility itemsets when $|f_H(D, 0)| \leq k$. Besides, H may contain more than k high utility itemsets when some itemsets have the same utility.

Definition 11. (Optimal minimum utility threshold) Let H be the complete set of top-k high utility itemsets in D. A minimum utility threshold δ^* is called *optimal minimum utility threshold* if there does not exist another threshold δ such that $\delta \geq \delta^*$ and $|f_H(D, \delta)| \geq k$. If $|H| \geq k$, $\delta^* = min\{u(X)| X \in H\}$.

Problem Statement. Given a transaction database D and the desired number of high utility itemsets k, the problem of finding the complete set of top-k high utility itemsets in D is to discover k itemsets with the highest utilities in D. An equivalent problem

statement is to discover all the itemsets whose utilities are no less than δ^* in D.

Example 2. Suppose the desired number of high utility itemset k is set to 3, the top-3 high utility itemsets in Table 1 is $H =$ {{BCDE}:40, {BDE}:36, {BCD}:34}. The optimal minimum utility threshold δ^* to retrieve H is equal to $min\{40, 36, 34\} = 34$.

2.2 Related work

2.2.1 High Utility Itemset Mining

Many studies have been proposed for mining HUIs, including *Two-Phase* [12], *IHUP* [2], *IIDS* [13] and *UP-Growth* [19]. *Two-Phase* and *IHUP* utilize *transaction-weighted downward closure* property to find high utility itemsets. They consist of two phases. In phase I, they find all HTWUIs from the database. In phase II, high utility itemsets are identified from the set of HTWUIs by scanning the original database. Although these methods capture the complete set of HUIs, they may generate too many candidates in phase I, i.e. HTWUIs, which degrades the performance of phase II and the overall performance (in terms of time and space). To reduce the number of candidates in phase I, various methods have been proposed (e.g. [13, 19]). Recently, Tseng et al. proposed *UP-Growth* [19] with four effective strategies *DGU*, *DGN*, *DLU* and *DLN*, for mining HUIs. Experiments showed that the number of candidates generated by UP-Growth in phase I can be order of magnitudes smaller than that of HTWUIs. To the best of our knowledge, UP-Growth is the state-of-the-art method for mining high utility itemsets. Although many studies addressed the topic of mining high utility itemset from transaction databases, few of them showed the flexibility of mining top-k high utility itemsets. Although the concept of top-k high utility itemset mining was first introduced in [3], the definition of high utility itemset in [3] is different from [2, 13, 15, 20] and our work.

2.2.2 Top-k Frequent Itemset Mining

In frequent pattern mining, several top-k pattern mining algorithms have been proposed [4, 5, 7, 9, 10, 14, 16, 17, 22]. Most of them (e.g. [5, 9, 10, 22]) follow a same general process for finding top-k patterns, although they also have several differences. We describe this general process below and then highlight the challenges for top-k high utility itemset mining.

The general process for mining top-k patterns from a database is the following. Initially, a top-k pattern mining algorithm sets minimum support threshold *minsup* to 0 to ensure that all the top-k patterns will be found. Then, the algorithm starts searching for patterns by using a search strategy. As soon as a pattern is found, it is added to a list of patterns L ordered by the support of patterns. The list L is used to maintain the top-k patterns found until now. Once k patterns are found, the value of *minsup* is raised to the support of the least interesting pattern in L. Raising *minsup* is used to prune the search space when searching for more patterns. Thereafter, each time a pattern is found that meets the minimum support threshold, the pattern is inserted into L, the patterns in L not respecting the threshold anymore are removed from L, and the threshold is raised to the support of the least frequent patterns in L. The algorithm continues searching for more patterns until no pattern is found by the search strategy.

What distinguish each top-k pattern mining algorithm are the data structures and search strategies to discover patterns. Top-k pattern mining algorithm needs to use appropriate data structure and search strategies to be efficient in both memory and execution

time. Besides, the efficiency of a top-*k* algorithm depends largely on how fast it can raise the minimum interestingness criterion (*minsup)* to prune the search space. To raise the threshold quickly, it is desirable that a top-*k* pattern mining algorithm uses a search strategy that will find the most interesting patterns as early as possible. Although several efficient top-*k* pattern mining algorithms [5, 9, 10, 22] have been designed based on this idea, it is not possible to simply adapt this idea to HUI mining. The reason is that the HUI mining is performed in two phases and that the exact utility of itemsets is only known during phase II. Therefore, mining the top-*k* HTWUIs during phase I would not necessarily result in finding the top-*k* HUIs in phase II. Another challenge is how to integrate effective strategies for raising *min-_util* given that the exact utility is only known in phase II.

By the above literature reviews, although there are many studies about utility mining and top-*k* pattern mining, fewer of them focus on the integration of mining top-*k* high utility itemsets. This paper addresses this topic to find top-*k* high utility itemsets.

Table 3. Items and their TWUs

Item	A	B	C	D	E	F	G
TWU	65	61	96	58	88	30	38

Table 4. Reorganized transactions and their RTUs

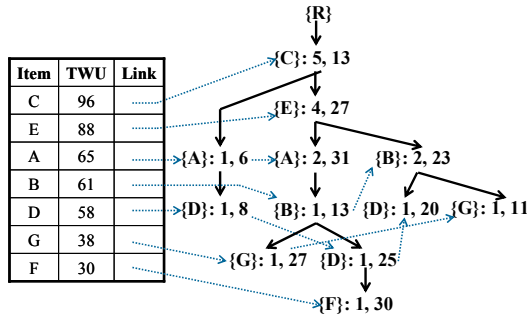| TID	Reorganized transaction	RTU
T_1'	(C,1) (A,1) (D,1)	8
T_2'	(C,6) (E,2) (A,2)	27
T_3'	(C,1) (E,1) (A,1) (B,2) (D,6)	30
T_4'	(C,3) (E,1) (B,4) (D,3)	20
T_5'	(C,2) (E,1) (B,2)	11

Figure 2. An UP-Tree when *min_util* = 0.

3. MINING TOP-K HIGH UTILITY ITEMSETS

In this section, we propose an efficient algorithm named *TKU* (*mining Top-K Utility itemsets*) for discovering top-*k* high utility itemsets without specifying *min_util*. We first present a baseline named *TKU_Base* approach and then introduce effective strategies to enhance its performance.

3.1 The baseline approach

The baseline approach TKU$_{Base}$ takes *k* as parameter and outputs the *k* itemsets with the highest utilities. It is an extension of UP-Growth, the current best method for mining high utility itemsets, and it adopts the idea of *UP-Tree* [19] to maintain the information of transactions and top-*k* high utility itemsets. The framework of TKU$_{Base}$ consists of three parts: (1) construction of *UP-Tree*, (2) generation of potential top-*k* high utility itemsets (abbreviated as *PKHUIs*) from the UP-Tree, and (3) identifying top-*k* high utility itemsets from the set of *PKHUIs*.

3.1.1 UP-Tree Structure
In this subsection, we briefly introduce the structure of UP-Tree. For the details about the UP-Tree, readers can refer to [19].

In UP-Tree, each node *N* consists of the following elements: *N.name* is the item name of *N*; *N.count* is the support count of *N*; *N.nu* is the node utility of *N*; *N.parent* records the parent node of *N*; *N.hlink* is a node link which points to a node whose item name is the same as *N.name*. *Header table* is employed to facilitate the traversal of UP-Tree. In the header table, each entry is composed of an *item name*, an *estimate utility value*, and a *link*. The link points to the last occurrence of the node having the same item name as the entry in the UP-Tree. The nodes whose item names are the same can be traversed efficiently by following the links in header table and the nodes in UP-Tree.

3.1.2 Construction of UP-Tree
A UP-Tree can be constructed with only two scans of the original database. In the first scan, the transaction utility of each transaction and TWU of each single item are computed. Thus, items and their TWUs are obtained. Subsequently, items are inserted into the header table in descending order of their TWUs. During the second database scan, transactions are reorganized and then inserted into the UP-Tree. Initially, the tree is created with a root *R*. When a transaction is retrieved, items in the transaction are sorted in descending order of TWU. A transaction after the above reorganization is called *reorganized transaction* and its transaction utility is called *RTU* (*reorganized transaction utility*). The RTU of a reorganized transaction T_d' is denoted as $RTU(T_d')$. When a reorganized transaction $T_d' = \{i_1, i_2, ..., i_m\}$ ($i_j \in I, 1 \leq j \leq m$) is retrieved, TKU$_{Base}$ applies the strategy *DGN* (*Discarding Global Node* utilities) [19] and calls the function ***Insert_Reorganized_Transaction***(R, i_1) to insert t_d'.

The function ***Insert_Reorganized_Transaction***(N, i_x) takes a node *N* in the UP-Tree and an item i_x ($i_x \in T_d'$, $1 \leq x \leq m$) in the reorganized transaction T_d' as inputs. The function is performed as follows:

Line 1: If *N* has a child *S* such that *S.item* = i_x, then increment *S.count* by 1; otherwise, create a new child node *S* with *S.item* = i_x, *S.count* = 1, *S.parent* = *N* and *S.nu* = 0.

Line 2: Increase *S.nu* by $(RTU(T_d') - \sum_{p=(x+1)}^{m} u(i_p, T_d'))$, where $i_p \in T_d'$ and $1 \leq p \leq m$.

Line 3: Call ***Insert_Reorganized_Transaction***(S, i_{x+1}) if $p \neq m$.

After inserting all reorganized transactions, the construction of the UP-Tree is completed. Figure 2 shows an UP-Tree for Table 1 when *min_util* = 0.

3.1.3 Generating PKHUIs from the UP-Tree
The proposed algorithm uses an internal variable named *border minimum utility threshold* (denoted as *border_min_util*) which is initially set to 0 and raised dynamically after a sufficient number of itemsets with higher utilities has been captured during the generation of PKHUIs. The development of the proposed method is based on the following definitions and lemmas.

Lemma 1. Let $P = <X_1, X_2,..., X_m>$ be a set of itemsets ($m \geq k$), where X_i is the *i*-th itemset in *P* and $u(X_i) \geq u(X_j), \forall \ i < j$. (In other words, X_i is the itemset with the *i*-th highest utility in *P*). For any itemset *Y*, if $u(Y) < u(X_k)$, *Y* is not a top-*k* high utility itemset.

Rationale. According to Definition 10, if there exist k itemsets whose utilities are higher than the utility of Y, Y is not a top-k high utility itemset.

Lemma 2. Let $P=\langle X_1, X_2,..., X_m\rangle$ be a set of itemsets $(m \geq k)$, where X_i is the i-th itemset in P and $u(X_i) \geq u(X_j), \forall\ i < j$. If $\delta_P = u(X_k)$, $f_H(D, \delta^*) \subseteq f_H(D, \delta_P)$.
Rationale. Let H be the complete set of top-k high utility itemsets. If $|H| \geq k$, $\delta^* = min\{u(X)|\ X \in H\}$ (by Definition 11). Because $\delta^* = min\{u(X)|\ X \in H\} \geq min\{u(X_i)|\ X_i \in P, 1 \leq i \leq k\} = u(X_k) = \delta_P$, $\delta^* \geq \delta_P$ and $f_H(D, \delta^*) \subseteq f_H(D, \delta_P)$).

Example 3. Suppose $k = 4$ and *border_min_util* $= 0$ initially. Let P be the set of 1-items in D. Then P = {{A}:20, {D}:20, {B}:16, {E}:15, {C}:13, {G}:7, {F}:5}, where the number beside each item is its exact utility. By Lemma 1, items {C}, {G}, {F} are unpromising to be the top-4 high utility itemsets. Therefore *border_min_util* can be raised to 15, the 4th highest utility value in P, and no top-k high utility itemset will be missed.

After raising *border_min_util*, the algorithm performs the *UP-Growth* search procedure with *min_util = border_min_util* to generate PKHUIs. Although Lemma 1 provides a way to raise *border_min_util*, it cannot be applied during the generation of PKHUIs in phase I. This is because the exact utilities of the PKHUIs are unknown during phase I. One of the solutions to this problem is to use *lower bound of the utility* of PKHUI to raise the *border_min_util*. A lower bound of the utility of an itemset can be estimated by the following definitions.

Definition 12. The *minimum item utility* of an item a is denoted as $miu(a)$ and defined as the value $u(a, T_r)$ for which $\neg \exists\ T_s \in D$ such that $u(a, T_s) < u(a, T_r)$.

Definition 13. The *minimum item utility of an itemset* $X=\{a_1, a_2,..., a_m\}$ is defined as $MIU(X) = \sum_{i=1}^{m} miu\ (a_i) \times SC(X)$.

Lemma 3. Let $C = \langle X_1, X_2,..., X_m\rangle$ be a set of itemsets $(m \geq k)$, where X_i is the i-th itemset in C and $MIU(X_i) \geq MIU(X_j), \forall\ i < j$. For any itemset Y, if $TWU(Y) < \delta_C = min\{MIU(X_i)\ |\ X_i \in\ C, 1 \leq i \leq k\}$, Y is not a top-k high utility itemset.
Rationale. According to Definition 8, $u(Y) \leq TWU(Y)$. If $TWU(Y) < \delta_C$, $u(Y) < \delta_C$. Besides, $u(Y) < MIU(X_i) \leq u(X_i)$, $X_i \in\ C, 1 \leq i \leq k$. According to Definition 10, if there exist k itemsets whose utilities are higher than the utility of Y, Y is not a top-k high utility itemset.

Lemma 4. Let $C = \langle X_1, X_2,..., X_m\rangle$ be a set of itemsets $(m \geq k)$, where X_i is the i-th itemset in C and $MIU(X_i) \geq MIU(X_j), \forall\ i < j$. If $\delta_C = MIU(X_k), f_H(D, \delta^*) \subseteq f_H(D, \delta_C)$.
Rationale. Let H be the complete set of top-k high utility itemsets. If $|H| \geq k$, $\delta^* = min\{u(X)|\ X \in H\}$ (by Definition 10). Because $\delta^* = min\{u(X)|\ X \in H\} \geq min\{u(X_i)|\ X_i \in\ C, 1 \leq i \leq k\} \geq min\{MIU(X_i)\ |\ X_i \in\ C, 1 \leq i \leq k\} = MIU(X_k)$, we have $\delta^* \geq \delta_C$ and $f_H(D, \delta^*) \subseteq f_H(D, \delta_C)$.

Lemma 5. For any itemset X, if $TWU(X) < border\_min\_util \leq \delta^*$, X and all its supersets are not top-k high utility itemsets.

Definition 14. The *maximum item utility* of an item a is denoted as $mau(a)$ and defined as the value $u(a, T_r)$ for which $\neg \exists\ T_s \in D$ such that $u(a, T_s) > u(a, T_r)$.

Table 5. Items and their mius and maus

Item	A	B	C	D	E	F	G
miu	5	4	1	2	3	5	2
mau	5	8	3	6	6	5	5

Definition 15. The *maximum utility of an itemset* $X=\{a_1, a_2,..., a_m\}$ is defined as $MAU(X) = \sum_{i=1}^{m} mau(a_i) \times SC(X)$.

Lemma 6. For any itemset X, if $MAU(X) < border\_min\_util < \delta^*$, X is not a top-k high utility itemset.
Rationale. According to Definition 15, we have $u(X) \leq MAU(X)$. If $MAU(X) < border\_min\_util$, $u(X) < border\_min\_util$. According to Definition 10, X is not a top-k high utility itemset.

Lemma 7. For any itemset X, the relationships between $MAU(X)$ $TWU(X)$, $u(X)$ and $MIU(X)$ is $MIU(X) \leq u(X) \leq min\{MAU(X), TWU(X)\}$.

Definition 16. An itemset is called a *PKHUI* (*P*otential top-*K* *H*igh *U*tility *I*temset) if its estimated utility (i.e., TWU) and MAU are no less than the *border_min_util*.

Based on the above lemmas and definitions, we have the following ideas to raise *border_min_util* during the generation of PKHUIs. As soon as a candidate X is found by the UP-Growth search procedure, we check whether its estimated utility (i.e, $TWU(X)$) is higher than *border_min_util*. If $TWU(X) < border\_min\_util$, X and all its supersets are not top-k high utility itemsets (Lemma 5). Otherwise, we check whether its *MAU* is higher than *border_min_util*. If $MAU(X) < border\_min\_util$, X is not a top-k high utility itemset (Lemma 6). Otherwise, X is considered as a candidate for phase II and it is outputted with its estimated utility value according to Lemma 7. If X is a valid PKHUI and $MIU(X) \geq border\_min\_util$, $MIU(X)$ can be used to raise the *border_min_util* (Lemma 3). To efficiently update *border_min_util*, we use a min-heap structure L to maintain the k highest MIUs of the PKHUIs until now. Once k MIUs are found, *border_min_util* is raised to the k-th *MIU* in L according to Lemma 3. Each time a PKHUI X is found and its *MIU* is higher than *border_min_util*, X is added into L and the lowest *MIU* in L is removed. After that, *border_min_util* is raised to the k-th *MIU* in L. The algorithm continues searching for more PKHUIs until no candidate is found by the UP-Growth search procedure. Figure 3 gives the pseudo code for the above processes.

```
If(TWU(X) ≥ border_min_util and MAU(X) ≥ border_min_util)
{   Output X and min{TWU(X), MAU(X)}
    If (MIU(X) ≥ border_min_util)
    { Add X to L and raise border_min_util by MIU(X)}
}
else
    {   X is not a valid PKHUI    }
```

Figure 3. The pseudo code for the strategy *MC*

Strategy 1. Raising the threshold by *M*UI of *C*andidate (MC)
For any newly mined PKHUI X, if its *MIU*, *TWU* and *MAU* are no less than the current *border_min_util*, then it is safe to use *MIU(X)* to raise *border_min_util*.

3.1.4 Identifying top-k HUIs from PKHUIs
In this part, we propose a basic method for identifying top-k high utility itemsets from the set of PKHUIs. Exact utilities of PKHUIs are identified and top-k high utility itemsets are examined by scanning the original database. Main method of this part is similar to that of phase II in [12, 19]. However, in previous work [12, 19], all candidates should be checked. Therefore, we only check the candidate itemset X whose estimated utility is larger than or equal to the *border_min_util* finally reached after phase I, i.e., $min(TWU(X), MAU(X)) \geq border\_min\_util$.

3.2 Effective strategies

In this subsection, we introduce four effective strategies to effectively raise *border_min_util* during different stage of the mining process.

3.2.1 Pre-evaluation Step

Although TKU$_{Base}$ provides a way to mine top-k HUIs, *border_min_util* is set to 0 before the construction of the UP-Tree. This results in the construction of a full UP-Tree in memory, which degrades the performance of the mining task. If we could raise *border_min_util* before the construction of the UP-Tree and prune unpromising items in the transactions, the number of nodes maintained in memory could be reduced and the mining algorithm could achieve better performance. To solve this problem, we propose a strategy named *PE* (*Pre-E*valuation) to raise *border_min_util* during the first scan of the database. A structure named *pre-evaluation matrix* (*PEM*) is used to store lower bounds for the utility of certain 2-itemsets. Each entry in *PEM* is denoted as *PEM*[x][y] and corresponds to a lower bound of $u(xy)$, where x, $y \in I$. Initially, each value in the matrix is set to 0. When a transaction $T_d = \{i_1, i_2, …, i_m\}$ ($i_j \in I$, $1 \leq j \leq m$) is retrieved during the first scan of the database, the utility of the itemset $\{i_1 i_j\}$ ($1 < j \leq m$) in T_d is added to the value of the corresponding entry of *PEM*[i_1][i_j]. For example, when $T_1 = \{(A,1), (C,1), (D,1)\}$ is retrieved, the corresponding entries *PEM*[A][C], *PEM*[A][D] are accumulated with $u(\{AC\}, T_1) = 6$ and $u(\{AD\}, T_1) = 7$. After scanning the database, *border_min_util* is set to the k-th highest value in *PEM*. Figure 4 shows the value of each entry in *PEM* after scanning Table 1. When $k = 4$, the 4th highest value in *PEM* is 18. Therefore, *border_min_util* can be raised to 18.

Strategy 2. Pre-Evaluation (PE) PE is applied during the first scan of the database. When a transaction $T_d = \{i_1, i_2, …, i_m\}$ ($i_j \in I$, $1 < j \leq m$) is retrieved, the utility of $u(i_1 i_j, T_d)$ is added to the corresponding entry *PEM*[i_1][i_j] in the pre-evaluation matrix, $1 < j \leq m$. Then, *border_min_util* can be raised to the k-th highest values in *PEM*. The space complexity is $O(|I|/2)$, where $|I|$ is the number of distinct items in the database.

Notice that in TKU$_{Base}$, the strategy DGU in [19] cannot be applied, because *border_min_util* is 0 before the construction of UP-Tree. However, if we raise *border_min_util* at pre-evaluation step, the strategy DGU can be applied to prune those items whose TWUs are less than *border_min_util*, which further reduces the size of UP-Tree and the number of candidates produced in phase I.

3.2.2 Raising the threshold by node utilities

The next proposed strategy is called *NU* (Raising the threshold by *N*ode *U*tilities), which is applied during the construction of UP-Tree. The strategy *NU* is developed based on the following lemmas.

Lemma 8. Let $PA = \{N_1, N_2,…, N_m, R\}$ be a path from a node N_1 to the root R in the UP-Tree and i_j be an item in N_j, $1 \leq j \leq m$. The node utility of N_1 is a lower bound for the utility of the itemset $\{i_1, i_2…, i_m\}$, $1 \leq j \leq m$.

Rationale. The UP-Tree is constructed by applying the strategy DGN [19]. According to the rationale described in [19], the utility of the itemset $\{i_1, i_2…, i_m\}$ is guaranteed to be higher than the node utility of N_1. Therefore, $N_1.nu \leq u(\{i_1, i_2…, i_m\})$.

Lemma 9. Let $S = <N_1, N_2,…, N_m>$ be an ordered set of nodes in UP-Tree ($m \geq k$), where N_i is the i-th node in S and $N_i.nu \geq N_j.nu, \forall\ i < j$. If $\delta_{NU} = N_k.nu$, then $f_H(D, \delta^*) \subseteq f_H(D, \delta_{NU})$.

	B	C	D	E	F	G
A	9	28	24	24	10	15
B		17	14	18	0	6
C			0	0	0	0
D				0	0	0
E					0	0
F						0

Figure 4. Pre-evaluation matrix

Rationale. Each node N_j to the root R represents an unique itemset X_j, $1 \leq j \leq m$ (Lemma 8). Let $S' = <X_1, X_2,..., X_m>$ be an ordered set of itemsets, where $N_j.nu \leq u(X_j)$ and $1 \leq j \leq m$. If $|H| \geq k$, then $\delta^* = min\{u(X)| X \in H\}$ (Definition 10). Because $min\{u(X)| X \in H\} \geq min\{u(X_j)| X_j \in S, 1 \leq j \leq k\} \geq min\{N_j.nu\ | N_j \in S', 1 \leq j \leq k\}$, we have $\delta^* \geq \delta_{NU}$ and $f_H(D, \delta^*) \subseteq f_H(D, \delta_{NU})$.

By Lemma 8 and 9, if there are more than k nodes in the UP-Tree during its construction and this value is higher than the current *border_min_util*, we can raise *border_min_util* to the k-th highest node utility in the UP-Tree. For example, suppose $k = 4$, when the first reorganized transaction $T_1' = \{(C,1), (A,1), (D,1)\}$ is inserted into the UP-Tee, the nodes $\{C\}$, $\{A\}$ and $\{D\}$ are created with node utilities 1, 6 and 8, which represent lower bounds for the utilities of itemsets $\{C\}$, $\{AC\}$ and $\{DAC\}$. When the second reorganized transaction is inserted into the tree, there are more than four nodes in the UP-Tree. Therefore, we can apply Lemma 9 to raise *border_min_util* to the *4*-th highest node utility in the UP-Tree.

Strategy 3. Raising the threshold by Node Utilities (NU) *NU* is applied during the construction of the UP-Tree (the second scan of the database). If there are more than k nodes in the current UP-Tree and k-th highest node utility is no less than the current *border_min_util*, *border_min_util* can be raised to the k-th highest node utility in the current UP-Tree.

3.2.3 Raising the threshold by MIU of Descendents

The third strategy that we propose is called *MD* (Raising the threshold by *MIU* of *D*escendents). It is applied after the construction of the UP-Tree and before the generation of PKHUIs. For each node N_α under the root in the UP-Tree, we traverse the sub-tree under N_α once to calculate the MIU of $N_\alpha N_\beta$ for each descendent node N_β of N_α. If there are more than k such values, *border_min_util* can be raised to the k-th highest value. For example, consider the UP-Tree in Figure 2 and suppose $k = 4$. The node under the root is $\{C\}$. We traverse the sub-tree under the node $\{C\}$ once and calculate the MIUs of its descendents. For the descendent $\{A\}$, the total support count of $\{A\}$ in the sub-tree of $\{C\}$ is $(1 + 2) = 3$. Therefore, the MIU of $\{AC\}$ is $(miu(\{A\}) + miu(\{C\})) \times SC(\{AC\}) = (5 + 4) \times 3 = 27$.

Strategy 4. Raising the threshold by *MIU* of *D*escendents (MD) *MD* is applied after the construction of UP-Tree and before the generation of PKHUIs. For each node N_α under the root in the UP-Tree, the support count of $N_\alpha N_\beta$ is calculated by traversing every its descendent node N_β. For each pair $N_\alpha N_\beta$, we calculate the MIU of $N_\alpha N_\beta$. If there are more than k MIUs larger than *border_min_util*, the *border_min_util* can be raised to k-th highest value.

Table 6. MIUs of descendents

Descendent	E	A	B	D	G	F
MIU	16	18	15	9	3	1

3.2.4 Raising the threshold during Phase II

In this part, top-*k* high utility itemsets are identified by checking the real utilities of PKHUIs in the database. The purpose of this part is the same as the basic method (in Section 3.1.4). Although the basic method can skip checking some candidates, the number of checked PKHUIs is still too large. Scanning database for checking the large amount of PKHUIs is very time-consuming.

In view of this, we propose an additional strategy for cooperating with the candidate skipping mechanism in Section 3.1.4. There are two main steps in this strategy. First, the candidates are sorted by the descendent order of estimated utilities, i.e., $min(TWU(X), MAU(X))$. Thus, the candidates with larger estimated utility values will be first checked; in other words, those having lower values will be checked later.

After *k* PKHUIs whose exact utilities are larger than *border_min_util* are found, a mechanism for raising *border_min_util* is applied. If the exact utility of a new HUI *Y* is larger than *border_min_util*, *Y* and *u(Y)* is inserted into a *top-k HUI list E* (All HUIs in *E* are ordered by their exact utilities), and the HUI with the lowest utility value is removed from *E*. Then *border_min_util* is raised to the utility of *k*-th HUI in *E*. If the estimated utility of the current candidate *Z*, i.e., $min(TWU(Z), MAU(Z))$, is less than the new border minimum utility threshold, all of the remaining candidates do not need to be checked. This is because that their upper bounds of exact utilities are not larger than *border_min_util*. Finally, *E* is the set of top-*k* high utility itemsets of the database. By this mechanism, the candidates with lower estimated utility values may not be checked since the *border_min_util* is raised. The I/O cost and execution time for phase II can be further reduced. This technique works well especially when *k* is small.

Strategy 5. Sorting candidates & raising threshold by the exact utility of candidates. (SE) *SE* is applied during phase II of TKU. Let CI be the set of candidates produced in phase I. Candidates in *CI* are sorted in the descendent order of their estimated utilities. Next, if there are more than *k* HUIs whose exact utilities are larger than *border_min_util*, *border_min_util* can be raised to the *k*-th highest exact utility. For any candidate *Z*, if $min(TWU(Z), MAU(Z))$ is less than *border_min_util*, *Z* and the remaining candidates do not need to be checked anymore.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of the proposed algorithm. Experiments were performed on computer with a 3.40 GHz Intel Core Processor with 4 gigabyte memory, and running on Windows 7. All of the algorithms are implemented in Java. Different types of real world datasets were used in the experiments. *Foodmart*, a sparse dataset, was acquired from Microsoft foodmart 2000 database [27]; *Mushroom*, a dense dataset, was obtained from the FIMI Repository [26]; *Chainstore*, a large dataset, was obtained from NU-MineBench 2.0 [15]. The two datasets Foodmart and Chainstore already contain unit profits and purchased quantities. For Mushroom dataset, unit profits for items are generated between 1 and 1000 by using a log-normal distribution and quantities of items are generated randomly between 1 and 5, as the settings of [19]. Table 7 shows the characteristics of the datasets used in the experiments. To evaluate the performance of the proposed strategies, we prepared three versions of TKU and gave them the names TKU, TKU_{noSE} and TKU_{Base} as shown in Table 8. These three versions are compared with the state-of-the-art utility mining algorithm UP-Growth [19].

Table 7. Datasets' characteristics

Dataset	#Transactions	Avg. length	#Items	Type
Foodmart	4,141	4.4	1,559	Sparse
Mushroom	8,124	23.0	119	Dense
Chainstore	1,112,949	7.2	46,086	Sparse Large

Table 8. Strategies used by the algorithms

Algorithm	Phase I				Phase II
	PE	NU	MD	MC	SE
TKU	Y	Y	Y	Y	Y
TKU_{noSE}	Y	Y	Y	Y	
TKU_{Base}				Y	Y

Because UP-Growth is not developed for mining top-*k* HUIs, it cannot be compared directly with TKU. To compare their performance, we considered the scenario where the users choose the optimal parameters for UP-Growth to produce the same amount of patterns as TKU (denoted as $UP_{Optimal}$ in the following experiments).

We first show the performance of the algorithms on the Foodmart dataset. The results are shown in Figure 5 and Table 9. In Figure 5 (a), it can be observed that the runtime for phase I of TKU approaches that of $UP_{Optimal}$. On the contrary, the performance of TKU_{Base} is the worst among all the algorithms. Its runtime is about 100 times slower than that of TKU. The reason is shown in Figure 5 (b). This figure shows the thresholds that the TKU and TKU_{Base} reached after phase I. Since UP-Growth does not raise the thresholds during the mining process, we show its initial thresholds (the optimal thresholds). In this figure, it can be observed that the thresholds reached by TKU are closer to the optimal thresholds than those of TKU_{Base}. On the other hand, TKU_{Base} does not apply the strategies PE, NU and MD. Therefore, it constructs a full UP-Tree with *min_util* = 0. Since raising the threshold for TKU_{Base} strictly depends on the MC strategy, it cannot be raised effectively. Thus its search space is the largest and its runtime is the longest.

The ineffectiveness of raising the threshold for TKU_{Base} also influences the number of candidates generated in phase I. The number of candidates generates by each algorithm is shown in Table 9. In this table, it can be observed that the number of candidates for TKU_{Base} is over 1000 times larger than TKU when *k* is less than 1000. The reason is that the strategies PE, NU and MD effectively raise the threshold at different stages of the mining process. Thus the number of patterns generated by TKU is much smaller than that of TKU_{Base}.

The runtime of each algorithm for phase II is shown in Figure 5 (c). Because each candidate needs to be checked in phase II and TKU_{Base} has the largest number of candidates, its performance for phase II is the worst. The performance of TKU_{noSE} is worse than TKU because the latter uses the strategy SE, which reduces the number of candidates need to be checked in phase II. Overall runtime of the algorithms is shown in Figure 5 (d). We can see that the runtime of TKU is over 100 times faster than TKU_{Base}, and only about twice less than that of $UP_{Optimal}$. Therefore, it can be concluded that TKU is an efficient algorithm since it can exactly find the top-*k* HUIs within a reasonable time.
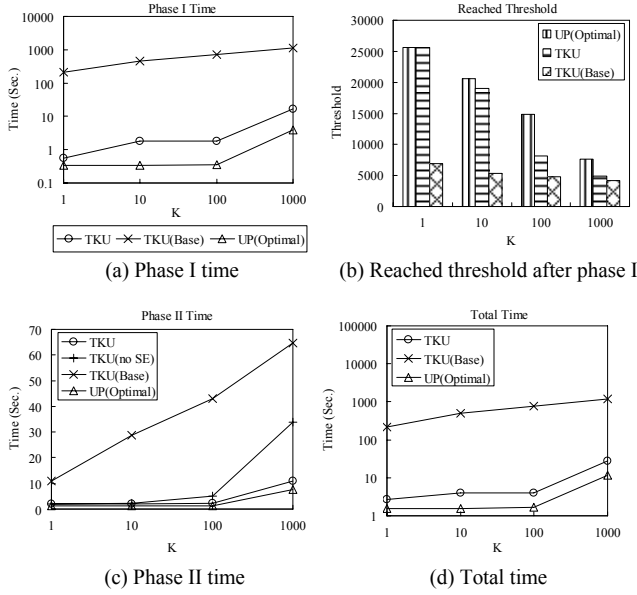
(a) Phase I time

(b) Reached threshold after phase I

(c) Phase II time

(d) Total time

Figure 5. Performance of the algorithms on Foodmart

Table 9. Number of candidates after Phase I

K	TKU	TKU$_{Base}$	Reduction ratio
1	1,379	2,466,459	1788.59
10	1,503	2,494,446	1659.65
100	2,456	2,537,225	1033.07
1,000	39,289	2,585,300	65.80

(a) Phase I time

(b) Reached threshold after phase I
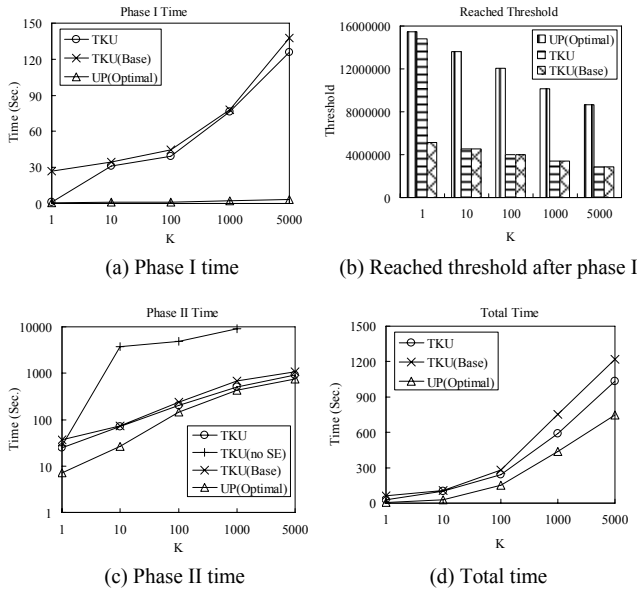

(c) Phase II time

(d) Total time

Figure 6. Performance of the algorithms on Mushroom

Table 10. Number of candidates after Phase I

K	TKU	TKU$_{Base}$	Reduction ratio
1	427	508,462	1190.78
10	597,301	713,793	1.20
100	803,377	920,040	1.14
1,000	1,540,583	1,657,403	1.08
5,000	2,594,337	2,711,248	1.05

Next, we show the performance on the Mushroom dataset. The results are shown in Figure 6 and Table 10. Figure 6 (a) shows the runtime for phase I of the algorithms. It can be observed that the runtime for phase I of TKU is close to that of TKU$_{Base}$. This is because Mushroom is a dense dataset. The estimated utility values, i.e., TWU values, of itemsets are much larger than their exact utilities. Thus the thresholds cannot be raised effectively in phase I. The thresholds reached by the algorithms are shown in Figure 6 (b). It can be seen that if *k* is larger than 1, the threshold reached by TKU is close to TKU$_{Base}$.

Table 10 shows the number of candidates generated by the algorithms during phase I. In this table, it can be seen that when *k* is larger than 1, the reduction ratio is slightly larger than 1. The reduction ratio decreases when *k* increases. Figure 6 (c) shows the runtime for phase II of the algorithms. The runtime for phase II of TKU$_{noSE}$ is the worst among the algorithms. This is because, without the SE strategy, TKU$_{noSE}$ needs to check all the candidates to determine which itemsets are top-*k* HUIs. When *k* is set to 5,000, the runtime of TKU$_{noSE}$ is too long to be executed (over 10,000 seconds). Finally, Figure 6 (d) shows the total runtime of the algorithms. We can conclude that although TKU is not as efficient as for the Foodmart dataset, it is still more efficient than TKU$_{Base}$.

Finally, we show the performance of the algorithms on Chainstore, a large dataset with over 1 million transactions. Because the runtime of TKU$_{Base}$ for this dataset is too long to be executed (over 20 hours when *k* = 1), we instead use UP-Growth with a low minimum utility threshold (0.01%) as the baseline (denoted as UP$_{Low}$ in the following experiments). The number of HUIs generated with *min_util* = 0.01% is about 3800. The results are shown in Figure 7 and Table 11. Figure 7 (a) shows the runtime for phase I of the algorithms. Since the threshold of UP$_{Low}$ is fixed, its runtime remains the same. It can be seen that the runtime of TKU is worse than UP$_{Low}$ when *k* is larger than 200. The reasons are that TKU needs to perform more computation for the strategies and it raises the threshold by the strategies step by step.

Figure 7 (b) shows the runtime for phase II of the algorithms. Although the runtime for phase I of TKU is slightly worse than UP$_{Low}$, the runtime for phase II of TKU is much faster than that of UP$_{Low}$. The total runtimes (the sum of the runtimes of phase I and phase II) are shown in Figure 7 (c). TKU is much faster than UP$_{Low}$. Generally, overall runtime of TKU is close to UP$_{Optimal}$. This is because UP$_{Low}$ needs to check all candidates in phase II; on the other hand, TKU only needs to check some of them because it uses the SE strategy.

Figure 7 (d) shows the number of candidates checked in phase II by each algorithm. It can be observed that although TKU generates much more candidates in phase I, the number of candidates that need to be checked by TKU is close to UP$_{optimal}$ in phase II. This is because using the SE strategy, TKU avoids checking some candidates that do not need to be checked. In contrary, since all candidates are checked by TKU$_{noSE}$, its performance is worse than TKU and UP$_{Optimal}$.

Finally, we show the threshold changes after applying the strategies in phase I. The results are shown in Table 11. In this table, it can be observed that the thresholds are raised higher when the strategies are applied. On the other hand, it can also be seen that the thresholds decrease when *k* increases. This is reasonable because the larger *k* is, the lower the thresholds are.

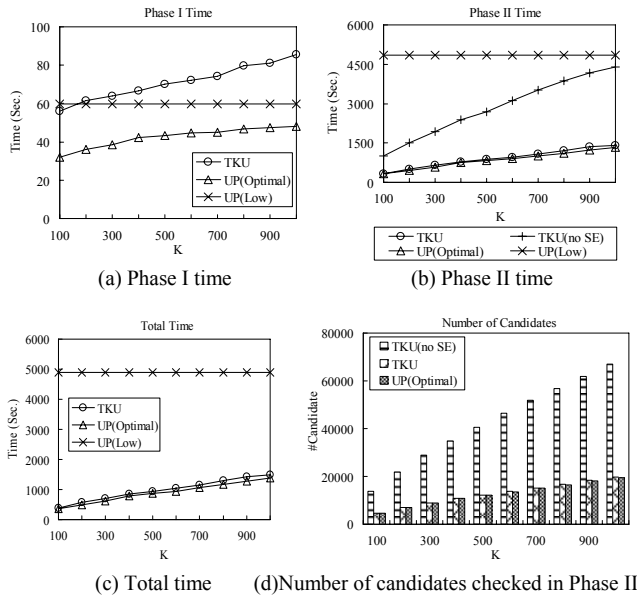Overall, Table 11 shows the effectiveness of all proposed strategies in phase I.

(a) Phase I time   (b) Phase II time

(c) Total time   (d)Number of candidates checked in Phase II

Figure 7. Performance of the algorithms on Chainstore

Table 11. Reached thresholds after each step in Phase I

K	PE	NU	MD	MC
100	2254.35	7509.38	7509.38	7509.38
200	1578.67	3929.84	3929.84	5324.16
300	1307.68	2728.92	2804.48	4346.16
400	1116.25	2158.83	2382.52	3803.76
500	988.49	1739.42	2050.96	3438.30
600	899.52	1457.85	1820.28	3145.21
700	826.38	1270.27	1650.04	2899.80
800	758.63	1117.47	1515.24	2734.67
900	712.97	1015.57	1412.70	2588.88
1,000	677.37	915.22	1334.82	2469.60

In general, the experimental results show that TKU outperforms TKU$_{Base}$ and UP$_{Low}$. Moreover, the performance of TKU is close to UP$_{Optimal}$. The reasons are listed as follows. First, strategies PE, NU, MD and MC in phase I effectively raise the threshold step by step in phase I. Thus the number of candidates that need to be checked in phase II is less and the search space in phase I is successfully reduced. Second, the SE strategy in phase II effectively reduces the number of candidates that need to be checked in phase II. Therefore, TKU is shown to be efficient with a performance that is close to that of the optimal case of the state-of-the-art utility mining algorithm UP-Growth.

5. CONCLUSION

In this paper, we have proposed an efficient algorithm named *TKU* for mining top-*k* high utility itemsets from transaction databases. TKU guarantees there is no pattern missing during the mining process. We develop four strategies for phase I to raise the border minimum utility threshold and reduce the search space and number of generated candidates. Moreover, a strategy is designed for phase II to decrease the number of checked candidates. The mining performance is enhanced significantly since both the search space and the number of candidates are effectively reduced by the proposed strategies. In the experiments, different types of real datasets are used to evaluate the performance of our algorithm. The experimental results show that TKU outperforms the baseline algorithms substantially and the performance of TKU is close to the optimal case of the state-of-the-art utility mining algorithm.

REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l Conf. on Very Large Data Bases*, pp. 487-499, 1994.

[2] C. F. Ahmed, S. K. Tanbeer, B.-S. Jeong and Y.-K. Lee. Efficient Tree Structures for High-utility Pattern Mining in Incremental Databases. In *IEEE Transactions on Knowledge and Data Engineering*, Vol. 21, Issue 12, pp. 1708-1721, 2009.

[3] R. Chan, Q. Yang and Y. Shen. Mining high-utility itemsets. In *Proc. of Third IEEE Int'l Conf. on Data Mining*, pp. 19-26, Nov., 2003.

[4] Y. L. Cheung, A. W. Fu, Mining frequent itemsets without support threshold: with and without item constraints. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16, No. 6, pp. 1052-1069, 2004.

[5] K. Chuang, J. Huang, M. Chen, Mining Top-K Frequent Patterns in the Presence of the Memory Constraint, *The VLDB Journal*, Vol. 17, pp. 1321-1344, 2008.

[6] A. Erwin, R. P. Gopalan and N. R. Achuthan. Efficient Mining of High-utility Itemsets from Large Datasets. In *PAKDD 2008, LNAI 5012*, pp. 554-561, 2008.

[7] A. W. Fu, R. W. Kwong and J. Tang, Mining N-Most Interesting Itemsets, In *Proc. of ISMIS'00*, 2000.

[8] J. Han, J. Pei and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of the ACM-SIGMOD Int'l Conf. on Management of Data*, pp. 1-12, 2000.

[9] J. Han, J. Wang, Y. Lu and P. Tzvetkov, "Mining Top-k Frequent Closed Patterns without Minimum Support," *In Proc. of ICDM*, 2002.

[10] Y. Hirate, E. Iwahashi and H. Yamana, TF^{2}P-Growth: An Efficient Algorithm for Mining Frequent patterns without any Thresholds, In *Proc. of ICDM* 2004.

[11] H.-F. Li, H.-Y. Huang, Y.-C. Chen, Y.-J. Liu, S.-Y. Lee. Fast and Memory Efficient Mining of High Utility Itemsets in Data Streams. In *Proc. of the 8th IEEE Int'l Conf. on Data Mining*, pp. 881-886, 2008.

[12] Y. Liu, W. Liao, and A. Choudhary. A fast high-utility itemsets mining algorithm. In *Proc. of the Utility-Based Data Mining Workshop*, 2005.

[13] Y.-C. Li, J.-S. Yeh and C.-C. Chang. Isolated Items Discarding Strategy for Discovering High-utility Itemsets. In *Data & Knowledge Engineering*, Vol. 64, Issue 1, pp. 198-217, 2008.

[14] S. Ngan, T. Lam, R. C. Wong and A. W. Fu, Mining N-most Interesting Itemsets without Support Threshold by the COFI-Tree, Int. J. Business Intelligence & Data Mining, Vol. 1, No. 1, pp. 88-106, 2005.

[15] J. Pisharath, Y. Liu, B. Ozisikyilmaz, R. Narayanan, W. K. Liao, A. Choudhary and G. Memik, NU-MineBench version 2.0 dataset and technical report, http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html

[16] T. M. Quang, S. Oyanagi, and K. Yamazaki, ExMiner: An Efficient Algorithm for Mining Top-K Frequent Patterns, *ADMA 2006, LNAI 4093*, pp. 436 – 447, 2006.

[17] L. Shen, H. Shen, P. Pritchard and R. Topor, Finding the N Largest Itemsets, in Proc. *Int'l Conf. on Data Mining*, pp. 211-222, 1998.

[18] B.-E. Shie, V. S. Tseng, and P. S. Yu. Online Mining of Temporal Maximal Utility Itemsets from Data Streams. In *Proc. of the 25th Annual ACM Symposium on Applied Computing* (*ACM SAC 2010*), 2010.

[19] V. S. Tseng, C.-W. Wu, B.-E. Shie, and P. S. Yu. UP-Growth: an efficient algorithm for high utility itemset mining. In *Proc. of Int'l Conf. on ACM SIGKDD*, pp. 253–262, 2010.

[20] V. S. Tseng, C. J. Chu, and T. Liang. Efficient mining of temporal high-utility itemsets from data streams. In *ACM KDD Workshop on Utility-Based Data Mining Workshop*, 2006.

[21] B. Vo, H. Nguyen, T. B. Ho, and B. Le. Parallel Method for Mining High-utility Itemsets from Vertically Partitioned Distributed Databases. In *KES 2009, Part I, LNAI 5711*, pp. 251-260, 2009.

[22] J. Wang and J. Han, TFP: An Efficient Algorithm for Mining Top-K Frequent Closed Itemsets, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, No. 5, pp. 652-664, May 2005.

[23] H. Yao, H. J. Hamilton, L. Geng, A unified framework for utility-based measures for mining itemsets. In *Proc. of ACM SIGKDD 2nd Workshop on Utility-Based Data Mining*, pp. 28-37, 2006.

[24] J.-S. Yeh, C.-Y. Chang and Y.-T. Wang. Efficient Algorithms for Incremental Utility Mining. In *Proc. of the 2nd Int'l Conf. on Ubiquitous information management and communication*, pp. 212-217, 2008.

[25] S.-J. Yen and Y.-S. Lee. Mining High-utility Quantitative Association Rules. In *Proc. of 9th Int'l Conf. on Data Warehousing and Knowledge Discovery* (*DaWaK'2007*), *Lecture Notes in Computer Science (LNCS) 4654*, pp. 283-292, 2007.

[26] Frequent itemset mining implementations repository, http://fimi.cs.helsinki.fi/

[27] FoodMart2000, Microsoft Developer Network (MSDN), http://msdn.microsoft.com/en-us/library/aa217032(v=sql.80).asp