TKQ: Top-K Quantitative High Utility Itemset Mining

Mourad Nouioua[1,5], Philippe Fournier-Viger[1][0000−0002−7680−9899], Wensheng Gan[2], Youxi Wu[3], Jerry Chun-Wei Lin[4], and Farid Nouioua[5]

[1] Harbin Institute of Technology (Shenzhen), Shenzhen, China
[2] Jinan University, Guangzhou, China
[3] Hebei University of Technology, Tianjin, China
[4] Western Norway University of Applied Sciences (HVL), Bergen, Norway
[5] University of Bordj Bou Arreridj, Bordj Bou Arreridj, Algeria
mouradnouioua@gmail.com,philfv@hit.edu.cn,wsgan001@gmail.com, wuc567@163.com,jerrylin@ieee.org, faridnouioua@gmail.com

Abstract. High utility itemset mining is a well-studied data mining task for analyzing customer transactions. It consists of finding the sets of items purchased together that yield a profit that is greater than a *minutil* threshold, set by the user. To find more precise patterns with purchase quantities, that task was recently generalized as high utility quantitative itemset mining. But an important drawback of current algorithms is that finding an appropriate *minutil* value is not intuitive and can greatly influence the output. A too small *minutil* value may lead to very long runtimes and finding millions of patterns, while a too high value, may result in missing many important patterns. To address this issue, this paper redefines the task as top-k quantitative high utility itemset mining and proposes a novel algorithm named TKQ (**T**op **K** **Q**uantitative itemset miner), which let the user directly specify the number k of patterns to be found. The algorithm includes three strategies to improve its performance. Experiments on benchmark datasets show that TKQ has excellent performance.

Keywords: High utility itemset mining · Quantitative itemsets · Top-k Pattern mining.

1 Introduction

Nowadays, a large amount of data is collected about customer purchases in online stores, brick and mortar stores. To analyze the behavior of customers and gain insights into their habits, a popular data mining task is high utility itemset mining (HUIM) [3, 6, 15]. The objective of that task is to enumerate all the high utility itemsets. A high utility itemset (HUI) is a set of items (products) that yield a profit that is greater or equal to a parameter called the minimum utility threshold (*minutil*). Finding HUIs in transactions is useful to understand customer habits. For instance, it can reveal that customers buying $\{chocolate, cake\}$

together is highly profitable. However, a major problem is that HUIs do not provide information about the purchase quantities of items (e.g. how many chocolate bars a customer may purchase with how many cakes).

To give more detailed information about HUIs to users, HUIM was recently generalized as the task of high utility quantitative itemset mining (HUQIM) [8, 7, 11, 14]. The goal is to find patterns called high utility quantitative itemsets (HUQIs) that are profitable and also include the quantity information. For example, a HUQI is *"chocolate:2, cake:4-6"*, which indicates that buying 2 chocolate bars with 4 to 6 cakes yields a high profit. HUQIs are more informative than HUIs. The added information about quantities can be useful for marketing [7, 8, 11]. But finding HUQIs is much more difficult than mining HUIs. The reason is that not only different items must be combined together to form HUQIs but also different quantities may be considered.

Several algorithms have been developed to mine HUQIs such as HUQA [14], VHUQI [8], HUQI-Miner [7], and FHUQI-Miner [11]. These algorithms adopt various data structures and strategies to reduce the search space and find all HUQIs. However, an important drawback of those algorithms is that finding an appropriate *minutil* value is not intuitive and can greatly influence the output. If the user chooses a too small *minutil* value, then an algorithm may run for hours and find millions of patterns. But if *minutil* is too high, few patterns may be discovered. Hence, users typically set this parameter by trial and error, which is time-consuming.

To address this issue, this paper redefines the task of HUQIM as top-k HUQIM and proposes a novel algorithm named TKQ (**T**op **K** **Q**uantitative itemset miner), which let the user directly specify the desired number k of patterns to be found without having to define *minutil* before starting the mining process. The algorithm then finds exactly k patterns, and relies on three strategies to improve its performance. Experiments on benchmark datasets show that TKQ has excellent performance and can be a suitable alternative to find HUQIs giving only the desired number of HUQIs k.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 presents preliminaries and the problem definition. Section 4 introduces the designed TKQ algorithm. Section 5 reports results from the experimental evaluation. Finally, Section 6 presents the conclusion.

2 Related Work

Research on pattern mining algorithms has started almost three decades ago by initially focusing on finding patterns that appear frequently in data such as frequent itemsets [4]. To efficiently find frequent itemsets in transactions, the property that the support is anti-monotonic is widely used. It says that the support (occurrence frequency) of an itemset cannot be greater than that of its supersets.

Though, frequent itemset mining (FIM) is useful, frequent itemsets are not always the most important for users. To find patterns that meet other interestingness criteria such as the profit, FIM was generalized as the problem of high utility itemset mining (HUIM) [5, 10, 12] where both quantities and unit profits are taken into account. HUIM is harder than FIM because the utility function for selecting patterns is neither monotonic nor anti-monotonic. As a solution, HUIM algorithms apply various upper bounds on the utility that are anti-monotonic to reduce the search space such as the TWU [10] and the remaining utility upper bound [9].

To address the problem that HUIs do not provide information about quantities, HUIM was extended as HUQIM. HUQA is the first HUQIM algorithm [14]. It introduced the concept of candidate quantitative itemsets as well as a search space pruning property based on a k-support bound measure. Then, a more efficient algorithm named VHUQI was designed [8], it relies on a vertical database representation to reduce the cost of database scans. Thereafter, to further reduce the search space, the HUQI-Miner algorithm was designed [7], which relies on both a *remaining utility* and a TWU-based upper bound to eliminate low utility quantitative itemsets (LUQIs). Recently, an improved version of this algorithm named FHUQI-Miner was proposed [11]. Besides, additional search space pruning strategies were introduced to reduce the search space. FHUQI-Miner is the state-of-the-art HUQIM algorithm [11].

3 Preliminaries and Problem Definition

This section first introduces the problem of HUQIM, and then the novel problem of top-k HUQIM is presented.

Let there be a finite set of N distinct items (products), $I = \{I_1, I_2, \ldots, l_N\}$. A positive number p_i is assigned to each item $i \in I$ called *external utility*, which represents its unit profit. A *quantitative transaction database* is a finite set of transactions, $D = \{T_1, T_2, \ldots, T_M\}$. Each transaction $T_d \in D$ $(1 \leq d \leq M)$ has a unique identifier d and it contains a set of exact Q-items, $T_d = \{x_1, x_2, \ldots, x_k\}$. An *exact Q-item* x is a pair (i, q) indicating that q units of item $i \in I$ have been purchased. For example, Table 1 shows a transaction database with four transactions (T_1 to T_4) and four items (A to D). The transaction T_2 contains two exact Q-items $(B, 4)$ and $(C, 3)$, which indicates that a customer has purchased four units of item B and three units of C. Table 2 gives the external utility values of items. For example, $p_A = 3$ indicates that a profit (external utility) of 3$ per unit is gained by selling product A. The transaction database presented in Table 1 will be used as a running example in the rest of this paper.

HUQIM contains two kinds of Q-items, exact Q-items and range Q-items. Range Q-items do not exist explicitly in the database but they are obtained using the combination process. A *range Q-item* is a triple (i, l, u) indicating that an item i is purchased with a quantity that is at least l and no more than u. The interval size of (i, l, u) is calculated as $u - l + 1$ and it is called a *Q-interval*. For instance, $(B, 4, 6)$ is a range Q-item with a Q-interval of size 3. Note that, any exact Q-item, (i, q), can be expressed as a range Q-item as (i, q, q). A set X of Q-items is called a *Q-itemset*. A Q-itemset with k items is called a *k-Q-itemset*.

Table 1: A transaction database

T_{id}	Transaction
T_1	(A,2) (B,5) (C,2) (D,1)
T_2	(B,4) (C,3)
T_3	(A,2) (C,2)
T_4	(A,2) (B,6) (D,1)

Table 2: External utility of items

Item	A	B	C	D
Profit	3	1	2	2

A Q-itemset containing at least one Q-item with a Q-interval size greater than 1 is called a *range Q-itemset*. For example, $[(B,5)(C,5,7)]$ is a range 2-Q-itemset.

A range Q-item $y = (j,l,u)$ *contains an exact Q-item* $x = (i,q)$ if $i = j$ and $q \in [l,u]$. Furthermore, y *contains a range Q-item* $z = (j',l',u')$ if $j' = j$, $l \leq l'$ and $u \geq u'$. As example, the range Q-item $(B,2,4)$ contains the exact Q-item $(B,3)$ and the range Q-item $(B,2,3)$.

An exact Q-item x *occurs in a transaction* T_d if $x \in T_d$. A range Q-item y *occurs in a transaction* T_d if a Q-item from T_d is contained in y. For instance, (A,2) occurs in T_1, and (B,4,5) occurs in T_1 and T_2.

A Q-itemset X *occurs in a transaction* T_d if for all $x \in X$, x occurs in T_d. The set of transactions where a Q-itemset X appears is called its *occurrence-set* and is denoted as $OCC(X)$. The occurrence frequency or support of a Q-itemset X is defined as $SC(X) = |OCC(X)|$. For instance, the occurrence-set of $X = [(A,2),(C,2)]$ is $OCC(X) = \{T_1, T_3\}$. Thus, $SC(X) = 2$.

The goal of HUQIM is to find all Q-itemsets that have a high utility (e.g. yield a high profit) [8]. The *utility of an exact Q-item* $x = (i,q)$ in a transaction T_d is defined and denoted as $u(x,T_d) = p_i \times q$. The *utility of a range Q-item* $x = (i,l,u)$ in a transaction T_d is defined as $u(x,T_d) = \sum_{j=l}^{u} u((i,j),T_d)$. The *utility of a Q-itemset X in a transaction* T_d is defined as $u(X,T_d) = \sum_{x \in X} u(x,T_d)$. For instance, $u((B,4),T_2) = 1 \times 4 = 4$, $u((B,3,4),T_2) = u((B,3),T_2) + u((B,4),T_2) = 0 + 4 = 4$ and $u([(A,2)(B,6)],T_4) = 6 + 6 = 12$.

The utility of a Q-itemset X in a database D is denoted and defined as $u(X) = \sum_{T_d \in OCC(X)} u(X,\ T_d)$ [8]. For instance, $u([(B,4,5)(C,\ 2,3)]) = u([(B,4,5)$ $(C,2,3)],\ T_1)$ $+u([(B,4,5)\ (C,2,3)],T_2) = 9 + 10$=19 and $u([(B,5)\ (D,1)]) = u([(B,5)(D,1)]$,$T_1)$ =5 + 2=7.

The *utility of a transaction* T_d is $TU(T_d) = \sum_{y \in T_d} u(y,T_d)$, that is the sum of the utility of its Q-items. The total utility of a database D is defined as $\sigma = \sum_{T_d \in D} TU(T_d)$, that is the sum of the utility of its transactions. For instance $TU(T_3) = u((A,2),T_2) + u((C,2),T_2) = 6 + 4 = 10$ and $\sigma = TU(T_1) + TU(T_2) + TU(T_3) + TU(T_4) = 17 + 10 + 10 + 14 = 51$.

The task of **High Utility Quantitative Itemset Mining (HUQIM)** aims to find all the *high utility quantitative itemsets* (HUQIs) in a database D given some user-defined *minimum utility threshold* ($0 \leq minutil \leq \sigma$). A Q-itemset X is a HUQI if $u(X) \geq minutil$. Otherwise, X is a *low utility quantitative itemset* (LUQI) [8]. For example, if $minutil = 13$, the Q-itemset $[(A,2)(B,5)(C,2)]$ is a HUQI because $u([(A,2)(B,5)(C,2)]) = 15 \geq 13$.

HUQIM is a hard problem for three reasons: First, the utility function is neither *anti-monotonic* nor *monotonic*, i.e, a Q-itemset may have a utility that is greater, smaller or equal to that of its supersets [8]. Hence, the utility cannot be directly used to reduce the search space and other strategies must be used. Second, one item in HUIM can form several Q-items in HUQIM. For example, the item B in Table 1 has three different quantities. Thus, we have three distinct Q-items $(B, 4)$, $(B, 5)$ and $(B, 6)$. Third, from each itemset, many range Q-itemsets may be formed which makes the search space larger than the case of HUIM.

HUQIM algorithms adopt two main operations, namely join operation and merge operation [7, 11]. The join operation starts from small itemsets and recursively adds items to these itemsets to find larger itemsets. The merge operation is used to find range Q-itemsets which are produced by combining a set of Q-items that have consecutive quantities. For example, Q-items $(B, 5)$ and $(C, 2)$ can be joined to obtain the 2-Q-itemset $[(B, 5)(C, 2)]$ while Q-items $(B, 4)$ and $(B, 5)$ can be merged to obtain the range Q-itemset $[(B, 4, 5)]$. It is important to note that in HUQIM, some pairs of LUQIs may be combined to obtain a range HUQI. But the combination process is limited with a parameter called $(qrc > 0)$ (Quantitative Related Coefficient). The qrc controls Q-intervals of produced range Q-itemsets to avoid obtaining range Q-itemsets having large Q-intervals. More precisely, two Q-itemsets $X = [(x_1, l_1, u_1), (x_2, l_2, u_2), \ldots, (x_k, l_k, u_k)]$ and $Y = [(y_1, l'_1, u'_1), (y_2, l'_2, u'_2), \ldots, (y_k, l'_k, u'_k)]$ can be combined to generate a range Q-itemset $Z = [(i_1, l_1, u_1), (i_2, l_2, u_2), \ldots, (i_k, l_k, u'_k)]$ if four conditions are satisfied [7]:
(1) X and Y are candidate Q-itemsets, i.e. $\frac{minutil}{qrc} \leq u(X) \leq minutil$ and $\frac{minutil}{qrc} \leq u(Y) \leq minutil$.
(2) X and Y have the same prefix. i.e, the first $(k-1)$ Q-items in X and Y are the same. Formally, $\forall (1 \leq i \leq k-1)$: $x_i = y_j$, $l_i = l'_j$ and $u_i = u'_j$.
(3) For the last Q-item, $x_k = y_k$ and $l'_k = (u_k + 1)$.
(4) The Q-interval of the last Q-item to be generated should be less than or equal to qrc, i.e, $u'_k - l_k \leq qrc$.

The combination of Q-itemsets can be done using three methods, named *Combine_ Min*, *Combine_Max* and *Combine_All* [2, 7]. Due to the space limitation, only the *Combine_All* method is described. The *Combine_All* method aims to find all possible range HUQIs that can be made by combining candidate Q-itemsets, or candidate Q-itemsets with range Q-itemsets [7]. For instance, consider a set of candidate Q-itemsets $\{[(A, 2)(C, 4)], [(A, 2)(C, 5)], [(A, 2)(C, 6)]\}$. The *Combine_All* method will generate three range Q-itemsets: $[(A, 2)(C, 4, 5)]$, $[(A, 2)(C, 5, 6)]$ and $[(A, 2)(C, 4, 6)]$. Then, the method will keep only HUQIs from the set of generated Q-itemsets. To see more details about the join and combining operations, the reader is refered to [7, 11].

Though several HUQIM algorithms have been designed, setting the *minutil* parameter is difficult and time-consuming for users. Setting it too low may result in too many patterns and very long runtimes, while setting it too high may result in too few patterns. To address this issue, the task of top-K HUQIM is proposed.

Top-k High Utility Quantitative Itemset Mining (Top-k HUQIM). Top-k HUQIM consists of finding a set γ of k Q-itemsets such that there are no other Q-itemsets not in γ that have a higher utility. For example, the top-3 HUQIs found in the database of the running example are: $[(A,2)(C,2)]$, $[(A,2)]$ and $[(A,2)(B,5)(C,2)(D,1)]$ with utilities 20, 18 and 17, respectively.

4 The TKQ algorithm

To efficiently perform the task of top-k HUQIM, a novel algorithm is designed, called TKQ (Top-K Quantitative itemset miner).

In contrast with traditional HUQIM where the user has to set the *minutil* threshold before starting the mining process, the *minutil* value is unknown in top-k HUQIM. Thus, *minutil* is initially set to zero. However, it is impracticable to start the itemset search with $minutil = 0$ because this may lead to consider many Q-itemsets that are not promising which will negatively affect the performance of the algorithm. To overcome this problem, it is desirable to apply some effective *threshold raising strategies* to raise *minutil* to higher values without missing any top-k HUQIs. For this purpose, TKQ adopts three effective threshold raising strategies. The first two raising strategies, named RIU and CUD, are adapted from strategies used for top-k HUIM [13, 1] so as to be used for top-k HUQIM. These strategies are applied before starting a depth-first search. The third raising strategy is used during the depth-first search to raise *minutil* based on the current list of top-k HUQIs. In the following, the Q-item utility-list structure is first introduced and then the pruning strategies used by TKQ to reduce the search space. Finally, the TKQ algorithm is described, integrating the three raising strategies, to find HUQIs.

Q-items utility-lists. As for FHUQI-Miner, TKQ is also a utility-list based algorithm. A utility-list is a structure that is used to represent each pattern (Q-itemset) of the mining problem. The utility-list allows to quickly calculate the utility of its associated Q-itemset without requiring to scan the database [9, 11].

Let there be a database D and a total order relation $\prec$ on distinct Q-items in D, the utility-list of a Q-itemset X, denoted as $UL(X)$, is composed of a set of tuples that have the form $(T_d, Eutil, Rutil)$. Each tuple stores the utility information of a Q-itemset X in a transaction T_d such as $T_d \in OCC(X)$. d is the identifier of transaction T_d, $Eutil$ is the utility of X in T_d, i.e, $Eutil(X, T_d) = u(X, T_d)$, and $Rutil$ is the remaining utility of X in transaction T_d. Formally, $Rutil(X, T_d) = \sum_{i \in T_d/X} u(i, T_d)$ where T_d/X contains the set of all Q-items that come after Q-items of X according to the order $\prec$. In addition to these tuples, $UL(X)$ have also two additional records, named $SumEutil$ and $SumRutil$. $SumEutil$ is the sum of all $Eutil$ values and it represents the exact utility of X in D, i.e, $u(X)$. Whereas, $SumRutil$ gives the sum of all $Rutil$ values which can be used to prune unpromising Q-itemsets and their extensions during the mining process using the remaining utility pruning strategy.

For the running example, let's assume that the order $\prec$ of Q-items is: $(A,2) \prec (B,4) \prec (B,5) \prec (B,6) \prec (C,2) \prec (C,3) \prec (D,1)$. The utility-list of Q-itemset

$[(A,2)(C,2)]$ has two tuples $(T_1,10,2)$ and $(T_3,10,0)$ with $SumEutil = 20$ and $SumRutil = 2$. Utility-lists allow to easily calculate the utilities of different Q-itemsets without scanning the database. However, it is impracticable to perform an exhaustive search by considering all possible Q-itemsets, especially when the number of Q-items is large. Therefore, it is necessary to apply some pruning strategies during the search for patterns to eliminate unpromising Q-itemsets with all their transitive extensions that are unpromising as well.

Pruning Strategies. TKQ adopts three pruning strategies to efficiently mine HUQIs:

TWU Pruning Strategy. The transaction weighted utility of a Q-itemset X, denoted as $TWU(X)$, is the sum of all transaction utilities where X appears. Formally, $TWU(X) = \sum_{T_d \in OCC(X)} TU(T_d)$.

The TWU measure is anti-monotonic and it has two important properties: First, the TWU is an upper bound on the utility of Q-itemsets and their supersets. More precisely, $\forall X \in D$, $TWU(X) \geq u(X)$. Second, for two Q-itemsets X and Y, if $X \subseteq Y$ then $TWU(X) \geq TWU(Y)$. Based on these properties, the TWU is used to prune unpromising Q-itemsets using property 1.

Property 1 (TWU Pruning Strategy). Given a Q-itemset X, the TWU pruning strategy states that if $TWU(X) < \frac{minutil}{qrc}$, then the Q-itemset X is not promising. Thus, X can be eliminated with all its transitive extensions. Otherwise, X is a promising Q-itemset. Back to our example, $TWU([(B,4)(C,3)])$=$u(T_2)$= 10. If $minutil = 30$ and $qrc = 2$, then $TWU([(B,4)(C,3)]) < \frac{30}{2}$. Thus, $[(B,4)(C,3)]$ can be pruned with all its extensions because they are not promising.

Remaining utility pruning strategy. The second pruning strategy adopted by TKQ is based on the remaining utility of Q-itemsets ($SumRutil$) stored in their utility-lists.

Property 2 (SumRutil Pruning Strategy). The SumRutil pruning strategy states that for a Q-itemset X, if $SumEutil(X) + SumRutil(X) < minutil$ then Q-itemset X and all its transitive extensions are LUQIs. For instance, $SumEutil$ $([(A,2)(C,2)])$=20 and $SumRutil([(A,2)(C,2)])$=2. If $minutil = 30$, then $[(A,2)$ $(C,2)]$ is pruned with all its extensions.

Co-occurrence pruning strategy. In addition to the above pruning strategies, TKQ also adopts another pruning strategy that allows to eliminate LUQIs with their transitive extensions without even the need to construct their utility-lists. The co-occurrence pruning strategy is based on *the TQCS structure*. The *TQCS structure (TWU of Q-items Co-occurrence based Structure)* is constructed during the second database's scan and it contains all pairs of Q-items that have co-occured in the database with their utility information. More precisely, in TKQ, the TQCS structure is a set of tuples that have the form (a,b,c,d) where a and b are two Q-items that co-occurred in the database, and c is the TWU of the Q-itemset $[ab]$ that resulted from their concatenation. Formally, $c = TWU([ab])$ and d is the exact utility of Q-itemset $[ab]$, i.e, $d = u([ab])$. Based on the TQCS, there exist two pruning strategies, namely *Exact Q-items Co-occurrence Pruning Strategy (EQCPS)* and *Range Q-items Co-occurrence Pruning Strategy (RQCPS)*. EQCPS is used to prune unpromising exact Q-itemsets while RQCPS is used for pruning unpromising range Q-itemsets. The reader is referred to [11] for a detailed explanation of the TQCS structure and its corresponding pruning strategies with illustrative examples.

The Algorithm. This section describes the main steps of TKQ to find top-k HUQIs including the raising strategies that are used. The TKQ algorithm (Algorithm 1) takes four parameters as input: (1) The transaction database D, (2) The user-selected number of patterns k, (3) The combining method CM and (4) The quantitative related coefficient qrc. TKQ outputs the top-k HUQIs in D. TKQ starts by scanning the database to calculate the TWU and the exact utility of each Q-item in D (line 1). Then, TKQ applies the first raising strategy based on the exact utilities of Q-items using the RIU raising strategy (line 2).

Algorithm 1: The TKQ algorithm

```
Input  : $D$: The quantitative transaction database, $k$: The desired number of patterns,
         $CM$: The combining method ($Combine\_Min$, $Combine\_Max$ or $Combine\_All$),
         $qrc$: The quantitative related coefficient.
Output: The top-k HUQIs.
1  First database scan to calculate the $TWU$ and utility of each Q-item;
2  Raise $minutil$ to the $k^{th}$ largest utility value; // RIU raising strategy
3  Create initial set of promising Q-items $P^*$ such that $\forall x \in P^* : TWU(x) \geq \frac{minutil}{qrc}$ and
     discard unpromising Q-itemsets; // TWU pruning strategy
4  Second database scan to create utility-lists of promising Q-items $ULs(P^*)$ and build the
     TQCS structure;
5  Raise $minutil$ to the $k^{th}$ largest utility value in TQCS; // CUD strategy
6  Create a priority queue to store the top-k Q-itemsets;
7  foreach $x \in P^*$ do
8      if $UL(x).SumEutil \geq minutil$ then
9          Q=Update_Queue(Q,x);
10         $H = H \cup x$;
11     else
12         if $UL(x).SumEutil + UL(x).SumRutil \geq minutil$ then $E = E \cup x$ ;
13         if $\frac{minutil}{qrc} \leq UL(x).SumEutil \leq minutil$ then $C = C \cup x$ ;
14 Discover High Utility range Q-itemsets ($HR$) using $CM$ and $C$;
15 foreach $x \in HR$ do
16     Q=Update_Queue(Q,x);
17 $QI_s \leftarrow sort(H \cup E \cup HR)$;
18 Recursive_Mining_Search($\emptyset$, $QI_s$, $UL_s(QI_s)$, $P^*$, $qrc$, $CM$, $k$,$Q$,$minutil$);
```

Real Items Utilities Threshold Raising Strategy (RIU). The RIU raising strategy is based on the utility calculation of all Q-items in the database. Let $q = \{q_1, q_2, \ldots, q_n\}$ be a set of n Q-items in the database D, and $L_1 = \{u(q_1), u(q_2), \ldots, u(q_n)\}$ be the list of utilities of Q-items in q. If $n \geq k$, then $minutil$ can be raised to the k^{th} largest value in L_1. In the running example, if $k = 5$, then $minutil$ is raised to the 5^{th} largest utility which is $u(B, 5) = 5$.

Based on the raised $minutil$ and using the TWU pruning strategy (property 1), TKQ identifies promising Q-items and remove unpromising Q-items to reduce the search space (line 3). Then, TKQ scans again the database to construct the utility-lists of promising Q-items and to build the TQCS structure (line 4) which will be used not only to reduce the search space during the recursive search for patterns, but also to raise again the $minutil$ threshold to a higher value using the CUD raising strategy (line 5).

Co-occurrence Threshold Raising Strategy (CUD). The CUD raising strategy uses the utilities of 2-Q-itemsets stored in TQCS to raise *minutil*. Note that, utilities of 2-Q-itemsets are easily calculated during the second database scan. Formally, suppose that the TQCS structure contains h tuples of the form (a_i, b_i, c_i, d_i) where $1 \leq i \leq h$. From these tuples, the list $L_2 = \{d_1, d_2, \ldots, d_k, \ldots, d_h\}$ is created which contains the utilities of all 2-Q-itemsets that co-occurred in the database. The utilities of these patterns are sorted in descending order. If $h \geq k$ and if the k^{th} largest utility in L_2, denoted as d_k, is larger than the current *minutil* then *minutil* is raised directly to d_k. Continuing with the previous example, the CUD raising strategy raises *minutil* to $u[(B,4)(C,3)] = 10$ which is the 5^{th} largest utility in L_2.

After applying the CUD raising strategy, TKQ creates a priority queue Q that is used to store the top-k HUQIs (line 6). Moreover, TKQ checks the utility of each promising Q-item. If the utility of the current Q-item is not less than *minutil*, this Q-item is inserted in Q using the *Update_Queue* procedure which will be presented later (lines 8-10). Otherwise, TKQ performs two tests to put this Q-item either in the set of candidate Q-items (C) or the set of Q-items to be explored (E) (lines 11-13). If the candidate set C is not empty, TKQ will perform the combination process (line 14). At this point, TKQ calls the recursive mining search procedure (line 18) which is illustrated in Algorithm 2.

Algorithm 2: *Recursive_Mining_Search*

```
Input : P: The prefix Q-itemset, $QI_s$: The Q-itemsets list, $UL_s(QI_s)$: Utility lists of
        Q-itemsets, $P^*$: The list of promising Q-itemsets, qrc: The quantitative related
        coefficient, CM: The combining method, k: The desired number of patterns, Q:
        A priority queue of k Q-itemsets sorted by ascending utility, minutil: The
        internal minimum utility threshold
Output: The set of HUQIs with respect to prefix P.
1  foreach $[Px]$ such that $x \in QI_s$ do
2      $QI_s \leftarrow \emptyset$; $P^* \leftarrow \emptyset$;
3      foreach $[Py]$ such that $y \in P^*$ and $y \succ x$ do
4          Apply EQCPS or RQCPS pruning strategies.
5          $Z \leftarrow [Pxy]$; $UL(Z) = Construct(x, y, P)$;
6          if $U \geq \frac{minutil}{qrc}$ then
7              $P^* = P^* \cup Z$;
8              if $UL(Z).SumEutil \geq minutil$ then
9                  Q=Update_Queue(Q,x);
10                 $E = E \cup Z$;
11             else
12                 if $UL(Z).SumEutil + UL(Z).SumRutil \geq minutil$ then $E = E \cup Z$;
13                 if $\frac{minutil}{qrc} \leq UL(Z).SumEutil \leq minutil$ then $C = C \cup Z$;
14     Discover High Utility range Q-itemsets $HR$ using $CM$ and $C$;
15     foreach $x \in HR$ do
16         Q=Update_Queue(Q,x);
17     $QI_s \leftarrow (H \cup E \cup HR)$;
18     Recursive_Mining_Search($Px$,$QI_s$,$UL_s(QI_s)$,$P^*$,$qrc$,$CM$,$k$,$Q$,$minutil$ );
```

The TKQ recursive search algorithm is a depth-first search algorithm which is similar to that used in FHUQI-Miner [11] with changes to find the top-k HUQIs. Note that, the reason for choosing FHUQI-Miner as a basis for TKQ is that FHUQI-Miner is the most efficient HUQIM algorithm [11]. The TKQ recursive search procedure has two more parameters which are the desired number of patterns k and the priority queue Q that will contain the top-k HUQIs. TKQ differs from FHUQI-Miner in that *minutil* in TKQ is not fixed. In fact, TKQ gradually raises the *minutil* value based on the HUQIs found during the depth-first search (line 9 and line 16). This raising process is done using the third raising strategy (*Update_Queue*) which is illustrated in Algorithm 3.

Algorithm 3: *Update_Queue*

```
Input  : $Q$: A priority queue of k Q-itemsets sorted by ascending utility, $x$: A Q-itemset
         to be inserted in $Q$.
Output: $Q$ with $x$ inserted.
1 Insert $x$ into Q;
2 if $|Q| > k$ then
3     Raise $minutil$ to the $k^{th}$ largest value in $Q$;
4     Remove all Q-itemsets having a utility less than $minutil$;
```

Besides, TKQ maintains the queue Q of the k Q-itemsets that have the highest utility until now. More precisely, once a new HUQI is found, it is inserted in Q (line 1). Then, *minutil* is raised to the new k^{th} highest value in Q (line 3). Finally, each Q-itemset in Q having a utility less than *minutil* is deleted from Q (line 4). As more HUQIs are found, the internal *minutil* threshold is raised based on Q, which allows to reduce the search space. When no more patterns can be generated, the top-k HUQIs are returned to the user.

5 Experiments

Experiments have been carried out to evaluate the performance of TKQ. They were done on a workstation having an Intel(R) i7-8700 processor, 16 GB RAM, and the Windows 7 operating system. The performance of TKQ was compared with FHUQI-Miner, and both were implemented in Java.

Datasets. Four benchmark datasets were utilized, which were obtained from the SPMF data mining library [2]. These datasets are often used in the HUQIM literature to evaluate algorithms, and have various characteristics. *Foodmart* is a sparse dataset with 4,141 transactions and 1,559 distinct items. *Retail* is a sparse dataset having 88,162 transactions and 16,470 items. *Mushroom* is a dense dataset with 8,416 transactions and 128 items. Lastly, *BMS* is a sparse dataset with 7,751 transaction and 23,340 items. The first dataset has real utility values, while the three others have synthetic values [11].

Influence of the RIU and CUD strategies on the runtime. A first experiment was done to evaluate the benefits of using the designed raising strategies in TKQ. TKQ was compared with two modified versions named TKQ-RIU and TKQ-CUD, where only the RIU and CUD strategies were used, respectively. The runtimes of the three algorithms for k values ranging from 10 to 5,000 are shown in Fig. 1. It can be observed that, generally TKQ is faster than TKQ-RIU and TKQ-CUD on the four datasets. For Retail, it can be clearly seen that, TKQ is faster than the two other algorithms especially when k is increased. Moreover, it can be seen also from Fig. 1 that, for small k values, the runtime of TKQ and

TKQ-RIU is quite similar. However, as k is increased, TKQ becomes faster than both the TKQ-RIU and TKQ-CUD algorithms.

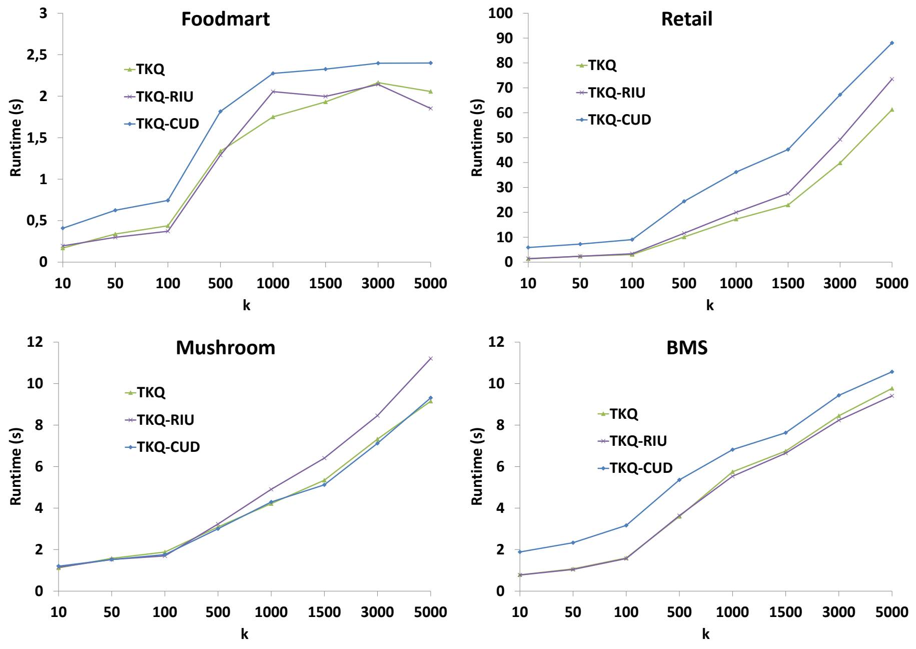

Fig. 1: Influence of the CUD and RIU strategies on runtime

To gain more insights into the effectiveness of each strategy, Fig. 2 shows how much each strategy raised the *minutil* threshold as a percentage of the optimal *minutil* value. It is found that for small k values, RIU can raise *minutil* more than CUD. However, for larger k values, CUD can raise *minutil* more than RIU. Hence, it is desirable to jointly use the two strategies to raise *minutil*.

Runtime comparison with FHUQI-Miner for the optimal *minutil* threshold. In a second experiment, TKQ's performance was compared with the state-of-the-art FHUQI-Miner HUQIM algorithm. To be able to compare both algorithms, FHUQI-Miner was set with an optimal *minutil* value that results in obtaining the same k patterns as TKQ. It should be noted that, this comparison is unfair as TKQ has to start searching from *minutil* $= 0$ while FHUQI-Miner directly knows the optimal *minutil* value and use this information to reduce the search space. However, this comparison is still interesting to see how close the performance of TKQ can be to that of FHUQI-Miner.

Tables 3 and 4 compare the runtimes of TKQ and FHUQI-Miner for different k values on the four datasets. FHUQI-Miner is generally faster than TKQ. This was expected as FHUQI-Miner directly uses the optimal *minutil* threshold. However, it can be found that the performance of TKQ is very close to that of

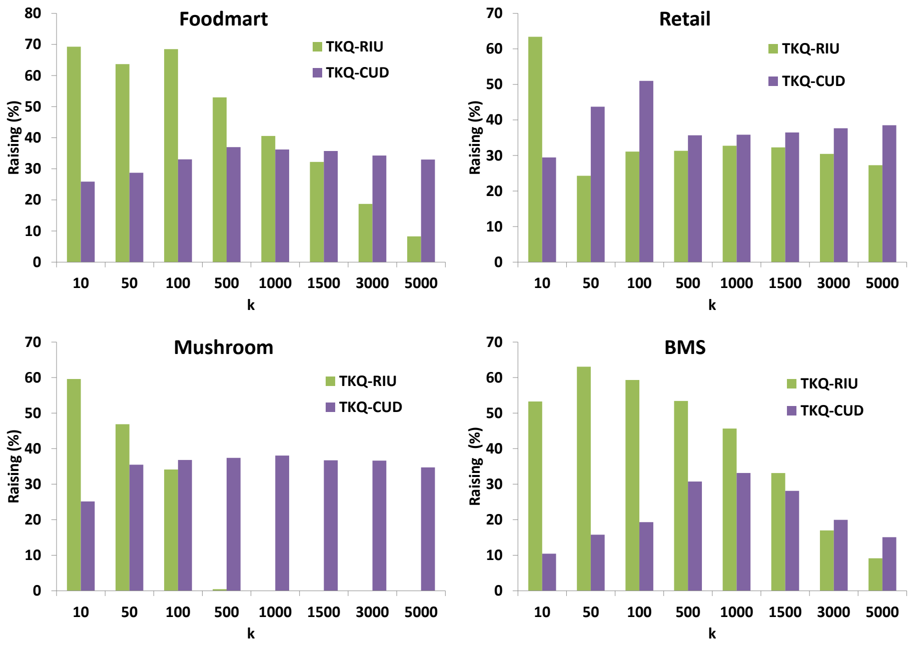

Fig. 2: Comparison of RIU and CUD's effectiveness to raise the *minutil* threshold

FHUQI-Miner. This indicates that, TKQ is efficient as the added cost for using the k parameter is generally small. Moreover, this is a good result considering that in real-life, the user is unlikely to set the optimal *minutil* threshold for FHUQI-Miner to obtain a given number of patterns as it will be demonstrated in the next experiment. From Table 3, it can be seen that the results of both algorithms on the Foodmart dataset are very close. For Retail, the difference between FHUQI-Miner and TKQ increases for large values of k. This result is expected since the task of top-k HUQIM becomes harder as k is increased, especially for large datasets such as Retail. However, it is worth noticing that, this difference is small and the performance of the two algorithms remains close. The results in Table 4 for the Mushroom and BMS datasets confirm again that the performance of TKQ is very close to the performance of FHUQI-Miner.

Benefits of using TKQ instead of FHUQI-Miner. The third experiment is performed to illustrate the benefits of using TKQ instead of FHUQI-Miner. Generally, if a user wants to find k patterns using FHUQI-Miner, the user must guess a *minutil* threshold value to find k HUQIs. But the user does not know beforehand how many patterns (HUQIs) will be found for a given *minutil* value. If the user sets *minutil* lower than the optimal *minutil* value, a very large number of patterns will be found, and runtime may be very long. Conversely, if the user sets *minutil* higher than the optimal *minutil* value, none or few patterns may be found. This experiment is designed to assess how likely a user

Table 3: Comparison of TKQ with FHUQI-Miner on Foodmart and Retail

k	**Foodmart**			**Retail**		
	minutil	**TKQ**	**FHUQI-Miner**	**minutil**	**TKQ**	**FHUQI-Miner**
10	26908	0.159	0.146	9781812	1.337	1.298
50	22725	0.236	0.237	2589300	2.485	1.549
100	22051	0.268	0.187	1603476	2.788	1.964
500	19843	0.644	0.358	445120	9.726	6.625
1000	18767	1.185	0.24	299264	14.869	10.791
1500	18015	0.945	0.255	230398	19.276	14.442
3000	16546	1.324	0.315	148950	32.713	25.82
5000	15422	1.222	0.348	117991	43.737	35.572

Table 4: Comparison of TKQ with FHUQI-Miner on BMS and Mushroom

k	**BMS**			**Mushroom**		
	minutil	**TKQ**	**FHUQI-Miner**	**minutil**	**TKQ**	**FHUQI-Miner**
10	1227600	0.75	0.713	3345705	1.073	1.045
50	735335	1.089	0.995	1645875	1.695	1.675
100	491940	1.375	1.257	1254574	2.104	2.066
500	246822	3.152	2.324	779980	3.196	2.821
1000	246723	3.898	2.291	617934	3.72	3.345
1500	246678	4.282	2.671	524151	4.078	4.045
3000	246606	5.328	2.585	389360	5.344	4.738
5000	246554	5.845	2.502	307956	6.206	5.559

is to select the optimal *minutil* value using FHUQI-Miner to get exactly the desired number of patterns k. To evaluate this, TKQ was run for different values of k on the four datasets, and the optimal *minutil* values were recorded. Based on the recorded *minutil* values, the average size between the *minutil* values for each pair of consecutive k values $[k_1, k_2]$ was calculated. Given that the user can choose *minutil* from the range $[0, \textit{total utility}]$, the interval size is the percentage of the difference between optimal *minutil* values for obtaining k_1 and k_2 versus the total utility of the database (σ). The average sizes for the four datasets are presented in Table 5. Note that, due to the space limitation, only the interval sizes are shown without giving the optimal *minutil* for each value of k.

Table 5: Probability to find HUQIs between $k1$ and $k2$

k_1-k_2	**Interval size(%)**			
	Foodmart	Retail	BMS	Mushroom
10-50	0,015	0,515	0,316	0,588
50-100	0,002	0,070	0,072	0,135
100-500	0,008	0,082	0,088	0,164
500-1000	0,003	0,010	0,030	0,056
1000-1500	0,002	0,004	0,017	0,032
1500-3000	0,005	0,005	0,025	0,046
3000-5000	0,004	0,002	0,015	0,028

It can be seen that the interval sizes are very small which shows the difficulty of selecting the optimal *minutil*. For example, the optimal *minutil* to find 50 HUQI for Foodmart is 22725 while the optimal *minutil* for 100 HUQI is 22051. The difference between these thresholds is thus 674, that is only 0.002% of the whole range of selection which is from 0 to the total utility (σ), i.e, $[0, 27027840]$. This means that using FHUQI-Miner, to obtain top-k HUQIs where $k \in [50, 100]$, the user should select *minutil* in a very small interval that represents only 0.002% from the whole range of selection. Thus, the selection of an optimal threshold for FHUQI-Miner is unlikely. It can be seen also that for some datasets such as Mushroom and Retail, the interval size decreases as k_1 and k_2 are increased.

From these results, it is concluded that, although FHUQI-Miner is slightly faster than TKQ, it is strongly recommended to use TKQ instead of FHUQI-Miner to obtain a desired number of HUQIs. Using TKQ allows the user to avoid spending time on fine-tuning the *minutil* threshold to find just enough patterns. And the runtime difference between TKQ and FHUQI-Miner is very small considering that users may need to run FHUQI-Miner multiple times to find an optimal *minutil* value.

6 Conclusion

To address the difficulty of setting the *minutil* threshold in HUQIM, this paper has redefined the task as top-k HUQIM. An algorithm named TKQ was designed to efficiently discover the top-k HUQIs. The algorithm adopts a depth-first search and integrates several pruning and raising strategies to improve its performance. An experimental evaluation with several datasets has shown that TKQ is efficient and has a performance that is generally close to the state-of-the-art HUQIM algorithm knowing that TKQ addresses a more difficult problem.

In future work, other optimizations will be developed. Moreover, it is planned to develop a distributed version of TKQ to mine patterns in very large datasets.

References

1. Duong, Q.H., Liao, B., Fournier-Viger, P., Dam, T.L.: An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies. Knowledge-Based Systems **104**, 106–122 (2016)
2. Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu, C.W., Tseng, V.S.: Spmf: a java open-source pattern mining library. The Journal of Machine Learning Research **15**(1), 3389–3393 (2014)
3. Fournier-Viger, P., Lin, J.C.W., Truong-Chi, T., Nkambou, R.: A survey of high utility itemset mining. In: High-Utility Pattern Mining, pp. 1–45. Springer (2019)
4. Fournier-Viger, P., Lin, J.C.W., Vo, B., Chi, T.T., Zhang, J., Le, H.B.: A survey of itemset mining. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery **7**(4), e1207 (2017)
5. Fournier-Viger, P., Wu, C.W., Zida, S., Tseng, V.S.: Fhm: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In: International symposium on methodologies for intelligent systems. pp. 83–92. Springer (2014)
6. Gan, W., Lin, J.C.W., Fournier-Viger, P., Chao, H.C., Tseng, V.S., Yu, P.S.: A survey of utility-oriented pattern mining. arXiv preprint arXiv:1805.10511 (2018)
7. Li, C.H., Wu, C.W., Huang, J., Tseng, V.S.: An efficient algorithm for mining high utility quantitative itemsets. In: 2019 International Conference on Data Mining Workshops (ICDMW). pp. 1005–1012. IEEE (2019)
8. Li, C.H., Wu, C.W., Tseng, V.S.: Efficient vertical mining of high utility quantitative itemsets. In: 2014 IEEE International Conference on Granular Computing (GrC). pp. 155–160. IEEE (2014)
9. Liu, M., Qu, J.: Mining high utility itemsets without candidate generation. In: Proceedings of the 21st ACM international conference on Information and knowledge management. pp. 55–64 (2012)
10. Liu, Y., Liao, W.k., Choudhary, A.: A two-phase algorithm for fast discovery of high utility itemsets. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining. pp. 689–695. Springer (2005)
11. Nouioua, M., Fournier-Viger, P., Lin, J.C.W., Gan, W.: Fhuqi-miner: Fast high utility quantitative itemset mining. Applied Intelligence (2021)
12. Peng, A.Y., Koh, Y.S., Riddle, P.: mhuiminer: A fast high utility itemset mining algorithm for sparse datasets. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining. pp. 196–207. Springer (2017)
13. Ryang, H., Yun, U.: Top-k high utility pattern mining with effective threshold raising strategies. Knowledge-Based Systems **76**, 109–126 (2015)
14. Yen, S.J., Lee, Y.S.: Mining high utility quantitative association rules. In: International Conference on Data Warehousing and Knowledge Discovery. pp. 283–292. Springer (2007)
15. Zhang, C., Han, M., Sun, R., Du, S., Shen, M.: A survey of key technologies for high utility patterns mining. IEEE Access **8**, 55798–55814 (2020)