ARTICLE IN PRESS

Information Sciences xxx (xxxx) xxx

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Efficient top-k high utility itemset mining on massive data

Xixian Han *, Xianmin Liu, Jianzhong Li, Hong Gao

School of Computer Science and Technology, Harbin Institute of Technology, China

ARTICLE INFO

Article history:
Received 29 January 2020
Received in revised form 19 July 2020
Accepted 8 August 2020
Available online xxxx

Keywords:
Massive data
Top-*k* high utility itemsets
Prefix-based partitioning
Full suffix utility

ABSTRACT

In practical applications, top-*k* high utility itemset mining (top-*k* HUIM) is an interesting operation to find the *k* itemsets with the highest utilities. It is analyzed that, the existing algorithms only can deal with the small and medium-sized data, and their performance degrades significantly on massive data. This paper presents a novel top-*k* HUIM algorithm PTM which can mine top-*k* high utility itemsets on massive data efficiently. PTM maintains the transaction database as a set of prefix-based partitions, each of which can fit in the memory and keeps the transactions with the same prefix item. The utility of an itemset can be computed by the transactions in one particular partition. PTM processes the prefix-based partitions in the order of average transaction utility to raise utility threshold faster. By the concise assistant data structures, PTM can skip majority of the partitions directly. For the partitions to be processed, PTM utilizes depth-first search on the set enumeration tree of the promising items to find the required results. The full-suffix-utility-based subtree pruning rule is devised to reduce the exploration space of set enumeration tree effectively. The extensive experimental results show that PTM can discover the top-*k* high utility itemsets on massive data efficiently.

1. Introduction

High utility itemset mining (HUIM) [9] is an important research topic in a variety of practical data mining applications, including bio-informatics [29], mobile commerce environment planning [31], web click stream analysis [33] and cross marketing [26]. HUIM usually can be considered as a generalization of a traditional frequent itemset mining (FIM) [1,13,22], which finds the sets of items appearing together frequently (called frequent itemsets) in the transaction database. Typically, each transaction contains the purchased quantities of the items, and each item has a unit profit. FIM only concerns the presence and absence of items, but neglects other information of the transaction database. Consequently, FIM may find the uninteresting frequent itemsets which generate low profits.

More often than not, instead of the sets of items with high frequencies, the users prefer to find the sets of items which can generate the high utilities (e.g. high profits). Given a minimum utility threshold, HUIM aims to find the itemsets with the high utilities. To be specific, the utility of an itemset is a function of the internal utility (e.g. the quantity of an item in the transaction) and the external utility (e.g. unit profit of an item) of the constituent items, and it measures the total profit generated by the itemset in the transaction database. The itemset is called high utility itemset (HUI) if its utility is no less than the utility threshold, otherwise it is a low utility itemset. However, the choice of the proper minimum utility threshold in HUIM is not an easy task, since it depends on the underlying data characteristics and distribution, which often are

* Corresponding author.
E-mail addresses: hanxx@hit.edu.cn (X. Han), liuxianmin@hit.edu.cn (X. Liu), lijzh@hit.edu.cn (J. Li), honggao@hit.edu.cn (H. Gao).

https://doi.org/10.1016/j.ins.2020.08.028
0020-0255/

unknown to the users. The proper minimum utility threshold significantly affects the performance of HUIM algorithms and the comprehension of the discovered results, because the output size can vary greatly according to the specified utility threshold. This can be stated from two aspects.

- If the utility threshold is set too low, a large number of HUIs will be generated, the users will be overwhelmed and it is difficult for the users to find the really interesting itemsets. Besides, the performance of HUIM algorithm will be degraded seriously, because discovering too many HUIs requires much computation resource.
- If the utility threshold is set too high, too few (or even no) HUIs will be generated. The users cannot find any interesting itemsets at all.

One way to solve the problems mentioned above is to adopt a trial-and-error process, which repeats the HUIM algorithms by the different minimum utility thresholds until the satisfied results are obtained. Obviously, this option is highly inefficient.

Inspired by the top-*k* interesting itemset mining model [4,11], top-*k* high utility itemset mining (top-*k* HUIM) is introduced to address the issue practically. Top-*k* HUIM only needs to specify the required number *k* of the high utility itemsets instead of the minimum utility threshold. Evidently, using the parameter *k* alone, top-*k* HUIM can control the size of the discovered results precisely, and help the users to easily find the useful itemsets which can generate the highest profits. Due to the practical importance, top-*k* HUIM attracts the researchers and emerges as a new hot research topic recently. The existing top-*k* HUIM algorithms can be divided into two broad categories: two-phase algorithms and one-phase algorithms. Two-phase algorithms [25,30] compute the top-*k* HUIs in two phases. In phase 1, the UP-Tree is constructed and the potential top-*k* HUIs are generated. In phase 2, the actual top-*k* HUIs are discovered by computing the utilities of the candidates in another database scan. One-phase algorithms [7,15,19,27] compute the utilities of the itemsets directly and discover the top-*k* HUIs in single phase. It is analyzed in this paper that the existing algorithms only consider the small and medium-sized data, which can be held entirely in memory. On massive data, the existing algorithms need to generate a prohibitively large number of candidates and incur a rather high execution cost. In a word, the existing top-*k* HUIM algorithms cannot handle massive data well.

This paper first proposes a baseline algorithm *BA*, which can mine top-*k* HUIs on massive data by apriori-like execution mode. It is found that BA will incur a relatively high computation cost and I/O cost. In order to efficiently process top-*k* HUIM, this paper proposes a novel PTM algorithm (*P*refix-partitioning-based *T*op-*k* high utility itemset *M*ining), which utilizes the prefix-based partitioning to find top-*k* high utility itemsets on massive data efficiently. In this paper, the items have an imposed order, and the items in any itemset are arranged in the specified order. The prefix-based partitions are pre-constructed by splitting the transaction database into partitions, each of which can fit in the memory and contains the transactions with the same prefix item. By the prefix-based partitioning, the computation of the utility of an itemset can be performed in one partition, and does not involve the other partitions. PTM processes the partitions in the selection order of average transaction utility, which can raise the border utility threshold more quickly. During the execution, PTM maintains the *k* itemsets with the highest utilities found currently. When all the partitions are processed completely, PTM can report the required results directly. By the pre-constructed concise data structures, PTM can skip majority of the partitions, which cannot generate any top-*k* HUIs. This saves the computation cost and I/O cost effectively. For the partitions to be processed actually, the enumeration tree of the promising items is grown in depth-first order to perform the in-memory processing quickly. This paper devises the full-suffix-utility-based subtree pruning rule to speed up the in-memory processing further. The extensive experiments are conducted on synthetic and real data sets. The experimental results show that PTM can mine top-*k* HUIs on massive data efficiently.

The main contributions of this paper are listed as follows:

- This paper proposes a novel prefix-partitioning-based PTM algorithm to mine top-*k* high utility itemsets on massive data efficiently.
- The prefix-based partitioning strategy is presented in this paper, which splits the transaction database into the prefix-based partitions.
- The set enumeration tree in depth-first growing mode is presented to process the partitions. The full-suffix-utility-based subtree pruning rule is devised to reduce the exploration space of the set enumeration tree.
- The extensive experimental results show that PTM can process top-*k* HUIM on massive data efficiently.

The rest of the paper is organized as follows. The related works are reviewed in Section 2, followed by the preliminaries in Section 3. Then Section 4 develops the baseline algorithm. Section 5 and Section 6 introduce the prefix-based partitioning and PTM algorithm, respectively. Section 7 evaluates the performance of PTM. Finally, Section 8 draws the conclusion of this paper.

2. Related works

This section reviews the literature on the HUIM algorithms (Section 2.1) and and the top-*k* HUIM algorithms (Section 2.2).

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

2.1. High utility itemset mining

The existing HUIM algorithms can be divided into two types: two-phase algorithms and one-phase algorithms.

2.1.1. Two-phase algorithms

Liu et al. [21] devise Two-Phase algorithm to compute high utility itemsets. In phase 1, the transaction-weighted utilization model is defined to estimate the upper-bound on the utility of an itemset, which satisfies the transaction-weighted downward closure property. The processing of phase 1 is executed in the level-wise fashion. At each level, only the combinations of high transaction-weighted utilization itemsets can be added into the candidate set. Since the transaction-weighted utilization of the itemset is an over-estimation of its actual utility, Two-Phase performs another database scan in phase 2 to discover the true high utility itemsets from the candidate set.

Li et al. [16] propose the isolated items discarding strategy (IIDS) to improve the level-wise HUIM methods. An item is called the isolated item in the bth pass if it is not contained in any candidate high utility b-itemset, and it will also not appear in any candidate itemsets whose lengths are greater than b. This IIDS strategy can be exploited to reduce the number of candidates, and improve the performance of the level-wise HUIM algorithms.

Ahmed et al. [3] notice that the current HUIM algorithms do not take any advantages from their previous design. The efficient tree structure IHUP is proposed to mine high utility patterns in incremental databases with "build once mine many" property. In this way, they can use the previous data structures and mining results to avoid the unnecessary calculations. IHUP tree requires adjustment only for the last added/deleted/modified transactions.

For the problem of multiple-pass scan in level-wise algorithms, Tseng et al. [28] develop UP-Tree-based UP-Growth algorithm to discover the high utility itemsets, which can be generated from the Up-Tree efficiently by only two passes of the scan on the database. The UP-Tree is constructed compactly with the revised transactions, and the potential high utility itemsets (PHUIs) can be discovered by applying FP-Growth algorithm on UP-Tree. Four strategies (DGU, DGN, DLU, DLN) are utilized to reduce the estimated utilities of candidates by discarding the utilities of the unpromising items. Since the number of PHUIs is often much smaller than that of candidates with transaction-weighted utility estimation, the phase 2 in UP-Growth is much more efficient than the previous algorithms.

2.1.2. One-phase algorithms

Liu et al. [20] develop HUI-Miner algorithm to mine high utility itemsets without generating candidates. A novel structure, utility list, is proposed and utilized in HUI-Miner. The utility list of an itemset keeps the utility information and heuristic information for pruning. HUI-Miner can discover the high utility itemsets from the utility lists in the similar way as Eclat [32].

Krishnamoorthy [14] proposes HUP-Miner to improve the state-of-the-art algorithms. Different from the utility list structure, HUP-Miner introduces the partitioned utility list structure. By the proposed data structures, two novel pruning strategies, *partitioned utility pruning* and *lookahead utility pruning*, are employed to reduce the over-estimation of the actual utilities of the considered itemsets and then reduce the search space significantly.

As observed in [10], although HUI-Miner can perform a single phase to compute the high utility itemsets by the utility list structures, its execution cost can be high since a costly join operation has to be executed to generate an itemset and evaluate the utility. FHM algorithm, which is devised in [10], utilizes a novel pruning strategy EUCP based on item co-occurrences to prune the itemsets without the join operation.

Liu et al. [18] propose d2HUP algorithm for high utility pattern mining with the itemset share framework. The algorithm explores reverse set enumeration tree in depth-first fashion to discover high utility patterns in a single phase directly. The search space can be reduced by the pruning technique based on tighter upper bounds on utilities. Besides, based on closure property and singleton property, the lookahead strategy is utilized to find the high utility patterns earlier.

Peng et al. [24] propose mHUIMIner (modified HUI-Miner) to mine high utility itemsets on sparse datasets quickly. The IHUP-tree structure is integrated to HUI-Miner to avoid the unnecessary utility list construction. Although the algorithm does not have a complex pruning strategy, it performs quite well on sparse dataset.

Zida et al. [34] devise a novel one-phase EFIM algorithm for HUIM. The algorithm is based on the principle that, in the search space, all operations for each itemset should be performed in linear time and space. High-utility database projection and high-utility transaction merging are introduced to reduce the cost of database space. Two new upper bounds of the utilities of the itemsets, *revised sub-tree utility* and *local utility*, are included in EFIM to prune the search space efficiently.

In utility-list-based HUIM algorithms, creating and maintaining utility lists can be time consuming, and require a huge amount of memory. In order to address the limitation, Duong et al. [6] propose ULB-Miner algorithm to mine high utility itemsets efficiently by a novel utility-list buffer structure. The utility-list buffer structure can be used to store and retrieve utilities efficiently, and re-use the memory during the mining process.

2.2. Top-k high utility itemset mining

The existing top-k HUIM algorithms can be categorized generally as two types: two-phase algorithms and one-phase algorithms.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

2.2.1. Two-phase algorithms

Two-phase algorithms usually consist of two phases. In phase 1, the candidate top-*k* HUIs are acquired by level-wise fashion or the tree-based structures. In phase 2, the actual top-*k* HUIs are found by another database scan.

Wu et al. [30] propose TKU algorithm to mine top-*k* HUIs without setting the minimum utility threshold. In phase 1, TKU first constructs UP-Tree by two scans on the transaction database to maintain the information of the transactions. An internal variable *border minimum utility threshold* is used in TKU, which is set to 0 initially and gradually raised during the generation of potential top-*k* high utility itemsets (PKHUIs). The UP-Growth search procedure [28] is performed to generate PKHUIs from the UP-Tree with the border minimum utility threshold. In phase 2, TKU computes the exact utilities of PKHUIs by another database scan, and identifies the actual top-*k* HUIs. Four strategies (*PE, NU, MD* and *MC*) are used to raise the threshold during the mining process in phase 1 and prune the search space. The fifth strategy, *SE*, is applied in phase 2 to raise the utility threshold and reduce the number of candidates that need to be checked.

Since TKU will generate a huge number of candidate patterns, Ryang et al. [25] develop REPT algorithm for mining top-*k* HUIs with highly decreased candidates. The overall process of REPT is similar to that of TKU. REPT first constructs the global tree by two scans on database, generates candidate top-*k* HUIs from the tree, and identifies the results from the candidates by another database scan. The main improvement of REPT is to raise the minimum utility threshold effectively in order to reduce the size of candidates significantly. In the first database scan of the global tree construction, two strategies, *PUD* and *RIU*, are applied to increase the current minimum utility threshold. In the second scan of the tree construction, another two strategies, *RSD* and *NU*, are employed to increase the threshold again. REPT pre-computes the exact utilities of itemsets with the length of 1 or 2, which are utilized to raise the threshold effectively in the construction of global tree construction. The potential top-*k* HUIs can be generated from the global tree based on UP-Growth procedure. The *MC* strategy can be used to increase the threshold further in this process. In phase 2, REPT identifies the top-*k* HUIs from the candidates with another strategy *SEP*, which retrieves the PKHUIs in the descending order of the estimated utilities.

2.2.2. One-phase algorithms

Different from two-phase algorithms, one-phase algorithms do not generate candidates, but calculate the utilities of the itemsets directly. In this way, the top-*k* HUIs can be determined in one phase.

Tseng et al. [27] propose TKO algorithm to find the top-*k* HUIs in one phase. TKO adopts search procedure of HUI-Miner, and utilizes vertical data representation structure (utility list) [20] to store the utility information of the itemsets. When an itemset is generated, its utility is calculated by its utility list directly. Initially, each item is associated with a utility list, which is constructed by scanning the database twice and represents the information of the item in the involved transactions. A novel strategy *RUC* is combined with the HUI-Miner search procedure to raise the minimum utility threshold quickly. Four strategies (*PE*, *DGU*, *RUZ* and *EPB*) are provided to improve the performance of TKO by raising the initial utility threshold and reducing the estimated utility values of the itemsets.

Duong et al. [7] propose the utility-list-based kHMC algorithm to discover the high utility itemsets in one phase. Two novel strategies, *EUCPT* and *TEP*, are devised to prune the search space. The EUCPT strategy uses the item co-occurrence information to eliminate a large number of join operations. The TEP strategy utilizes a novel upper-bound on the utilities of the itemsets to reduce the search space. An intersection procedure is introduced in kHMC to construct utility lists with a low complexity. Besides, the early abandoning strategy is integrated in kHMC algorithm to abandon the construction of the utility lists whose corresponding itemsets are not top-*k* HUIs. In order to raise the internal minimum utility threshold effectively, several strategies (*RIU*, *COV* and *CUD*) are exploited to initialize and dynamically adjust the internal minimum utility threshold.

Krishnamoorthy [15] argues that the strategies for raising the minimum utility threshold values in the state-of-the-art algorithms are not effective for dense database, especially during the early stages of the mining process. A new utility-list-based method THUI is presented in [15] to efficiently mine top-*k* HUIs in one phase. Four threshold raising strategies (*RIU, RUC, LIU-E, LIU-LB*) are proposed in THUI to effectively raise the minimum utility threshold. The last two strategies are the newly introduced in THUI with LIU structure. Given the specified ordering of items, the LIU (leaf itemset utility) structure is a triangular matrix to keep the utility information of the itemsets containing a contiguous set of items in a compact form, which is argued to be more effective in raising threshold values than the previous structures.

Liu et al. [19] present a novel algorithm TONUP to directly grow top-*n* HUIs in one phase, which extends d2HUP [18] with the top-*n* interesting pattern model. The patterns in TONUP are enumerated as a prefix extension of another patterns. As more patterns are enumerated, the patterns with the first *n* greatest utilities are shortlisted. The border utility threshold can be raised quickly to prune the search space. TONUP improves d2HUP in several aspects, including an improved version iCAUL of the data structure CAUL and two strategies. The first strategy *AutoMaterial* uses a self-adjusting threshold to trade off between the pseudo projection and the materialization projection. The second strategy *DynaDescend* reorders the items in the descending order of the local utility upper bounds to quickly raise the border threshold and improve the pruning operation. Besides, more opportunistic strategies (*ExactBorder*, *SuffixTree* and *OppoShift*) are also proposed to implement the full-strength TONUP. The ExactBorder strategy uses the exact utilities of the enumerated patterns to raise the border threshold. The SuffixTree strategy employs a suffix tree to keep the shortlisted patterns, which is more efficient than the previous data structures. The OppoShift strategy opportunistically shifts to a two-round approach to deal with the extremely long enumerated patterns.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

2.2.3. Discussion

For the existing top-k HUIM algorithms, whether two-phase algorithms or one-phase algorithms, their main goals are to raise the border utility threshold as quickly as possible. The initial border utility threshold is set to 0 in top-k HUIM algorithm generally. And a greater threshold in the execution of the algorithm can help to reduce the exploration space and prune the candidate top-k HUIs effectively. However, the major problem of the existing algorithms is that, *they only consider the small and medium-sized transaction databases, which can be held entirely in memory*. On massive data, where the memory cannot keep the data entirely, the existing algorithms cannot identify the top-k HUIs directly. In this case, an extension should be utilized for them to deal with the transaction database, which first retrieves a part of the transaction database subject to memory constraint, computes the local top-k HUIs in the part, and discovers the final results from the local results. According to the analysis in Section 3.2, the extension will generate a prohibitively large number of candidates and degrade the performance severely. As such, this paper aims to devise a novel algorithm, which can deal with the massive data and mine the top-k HUIs efficiently.

3. Preliminaries

3.1. Problem definition

Given a transaction database T of n transactions, each of which is a subset of the universe of items $\mathscr{I} = \{i_1, i_2, \ldots, i_d\}$. The itemset is a subset of $\mathscr{I}$ and a b-itemset is an itemset with b items. Each item i in $\mathscr{I}$ has a positive value, indicating its importance or unit profit, which is called its *external utility* $eu(i)$. $\forall t \in T$, t is associated with a unique identifier *TID*, and each item i in t also has a positive value, indicating its quantity in t, which is called its *internal utility* $iu(i,t)$. The frequently used symbols in this paper are listed in Table 1.

Definition 3.1 (*The utility of an item in a transaction*). The utility $u(i,t)$ of an item i in a given transaction t is defined as $u(i,t) = eu(i) \times iu(i,t)$.

The utility of the item i in transaction t represents the profit generated by the sale of the item i in this transaction.

Definition 3.2 (*The utility of the itemset in a transaction*). Given the itemset X contained in a transaction t, the utility $u(X,t)$ of X in t is defined as $u(X,t) = \sum_{i \in X} u(i,t)$.

The utility of the itemset X in a transaction t represents the profit generated by the sale of X in t. Of course, if the transaction t does not contain the itemset X, the utility of X in t is 0.

Definition 3.3 (*The utility of the itemset in the transaction database*). Given the itemset X, its utility $u(X)$ in transaction database T is defined as $u(X) = \sum_{t \in g(X,T)} u(X,t)$, where $g(X,T)$ represents the set of transactions in T containing X.

The utility of the itemset X in the transaction database T represents the total profit generated by the sale of X in T.

Definition 3.4 (*Top-k high utility itemset mining*). Given the transaction database T and the specified number k, top-k high utility itemset mining (top-k HUIM) is to discover k itemsets with the greatest utilities, denoted by **R**. That is, $\forall X_1 \in R, \forall X_2 \in (\mathscr{P}(\mathscr{I}) - \varnothing) - R, u(X_1) \geqslant u(X_2)$, where $(\mathscr{P}(\mathscr{I}) - \varnothing)$ represents the power set of $\mathscr{I}$ excluding empty set.

Example 3.1. In this paper, we use a running example, as depicted in Figs. 1 and 2, to illustrate the algorithm execution. There are seven items $\{i_1, i_2, \ldots, i_7\}$ in the running example. By setting k to 3, the top-3 HUIs in the running example are listed in Fig. 3 with their utilities. For the itemset $\{i_3, i_6, i_7\}$, it is contained in $T(1), T(2), T(4), T(5), T(7), T(8)$, where $T(tid)$ represents the transaction in T with *TID tid*, $u(\{i_3, i_6, i_7\}) = 474$.

3.2. The analysis of the existing algorithms

This part analyzes the behavior of the existing top-k HUIM algorithms on massive transaction database T.

Since the memory cannot hold T entirely, the extension (denoted by *EA* in this part) of the existing algorithms should be developed, which reads a partition of the database subject to memory constraint each time, processes the partition and computes the top-k HUIs from the local results. The transaction database T can be divided logically into a number of non-overlapping partitions $T_1, T_2, \ldots, T_m, \forall 1 \leqslant a \neq b \leqslant m, T_a \cap T_b = \varnothing, \bigcup_{1 \leqslant a \leqslant m} T_a = T$. Each partition can be kept entirely in memory.

At first glance, EA can compute the local top-k HUIs (LR_a) from each partition T_a by the existing algorithms, and then discover the top-k HUIs from the candidates in $LR_1, LR_2, \ldots, LR_m$ by another scan on T to calculate their exact utilities. Nevertheless, as described below, for computing the correct top-k HUIs, it is not enough to just acquire the local top-k HUIs from each partition.

Given the itemset X, its utility $u(X) = \sum_{t \in g(X,T)} u(X,t)$. Since T can be divided logically into $T_1, T_2, \ldots, T_m$, the local utility $u(X, T_a)$ of X in $T_a (1 \leqslant a \leqslant m)$ is $u(X, T_a) = \sum_{t \in g(X,T_a)} u(X,t)$. Obviously, $g(X,T) = \bigcup_{1 \leqslant a \leqslant m} g(X, T_a)$ and $g(X, T_a) \cap g(X, T_b) = \varnothing$

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

Table 1
Summary of symbols.

Symbol	Meaning
T	the transaction table
d	the number of the distinct items
$\mathscr{I}$	the universe of the items $\{i_1, i_2, \ldots, i_d\}$
k	the specified number of HUIs
R	the top-k HUIs in T
$g(X,T)$	the transactions in T containing the itemset X
MH	min–heap to maintain the itemsets with k greatest utilities
δ	the border utility threshold
P_a	The prefix-based partition whose transactions begin with the item i_a
ps_a	The promising item set for P_a

TID	$Transactions\ in\ T$
1	$(i_3,3),(i_5,3),(i_6,6),(i_7,6)$
2	$(i_2,3),(i_3,5),(i_5,3),(i_6,4),(i_7,2)$
3	$(i_1,5),(i_2,2),(i_5,2),(i_6,2),(i_7,3)$
4	$(i_2,2),(i_3,4),(i_6,6),(i_7,3)$
5	$(i_1,6),(i_2,4),(i_3,2),(i_5,4),(i_6,3),(i_7,1)$
6	$(i_1,1),(i_4,4)$
7	$(i_1,5),(i_2,1),(i_3,1),(i_4,6),(i_5,3),(i_6,4),(i_7,5)$
8	$(i_2,5),(i_3,5),(i_5,5),(i_6,1),(i_7,4)$
9	$(i_2,1),(i_3,3),(i_4,6),(i_5,1),(i_6,5)$
10	$(i_1,1),(i_2,3),(i_6,1),(i_7,5)$

Fig. 1. The illustration of transaction database in the running example.

$item$	i_1	i_2	i_3	i_4	i_5	i_6	i_7
eu	5	8	9	7	4	7	6

Fig. 2. The illustration of external utilities in the running example.

$top\ k\ HUIs$	$utility$
$\{i_2, i_3, i_6\}$	469
$\{i_3,\ i_6,\ i_7\}$	474
$\{i_2,\ i_3,\ i_6,\ i_7\}$	489

Fig. 3. The illustration of top-k HUIs in the running example.

$(\forall 1 \leqslant a \neq b \leqslant m)$, so $u(X) = \sum_{1 \leqslant a \leqslant m} u(X, T_a)$. This means that, for any itemset, its utility is the sum of its local utilities in the m partitions. Given an itemset X, even though $u(X, T_a)$ $(\forall 1 \leqslant a \leqslant m)$ does not appear in the k greatest local utilities in T_a, X still can be a top-k HUI. We can consider a particular example here. Given two itemsets X and Y, the local utilities of Y are among the top-k greatest local utilities in all m partitions except for T_a, while all the local utilities of X are not among the top-k greatest local utilities. It is still possible that $u(X) > u(Y)$, as long as $u(X, T_a)$ is much larger than $u(Y, T_a)$, to be specific, $u(X, T_a) - u(Y, T_a) > \sum_{1 \leqslant b \neq a \leqslant m} \{u(Y, T_b) - u(X, T_b)\}$.

To some extent, the aggregation of local utilities for EA to discover the top-k HUIs is similar to the execution in the traditional top-k query [23], which utilizes the sorted lists of the involved attributes to compute the top-k results. In EA, the sorted list for the partition T_a consists of potentially up to $(2^d - 1)$ itemsets. Of course, many of the itemsets with the local utilities 0 do not appear in the sorted list. As analyzed in [12], given the element number N of the single sorted list, the number m of the involved sorted lists and the result size k, sorted-list-based methods are estimated to retrieve up to $N \times (k/N)^{1/m}$ elements in each relevant sorted list to make the top-k results contained in the retrieved elements under the assumption of the uniform and independent distribution. This means that only acquiring local top-k HUIs in each partition is far from enough to discover the top-k HUIs. In actual execution, EA even needs to acquire all the possible itemsets in each partition in order to avoid missing the possible top-k HUIs and guarantee the completeness of the results. This will generate a significantly large number of local candidates and incur a prohibitively high execution cost. In short, it is rather difficult for EA (if not impossible) to discover top-k HUIs on massive transaction database.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

Example 3.2. Suppose that the memory can maintain four transactions at a time. The transaction database in the running example can be divided into three partitions T_1, T_2 and T_3, as illustrated in Fig. 4. The possible itemsets generated in each partition are depicted also, which are arranged in the descending order of local utilities. If an itemset is not the subset of any transaction in the partition, its utility is 0 and it does not appear in the possible itemsets actually. By setting k to 3, only computing the local top-3 HUIs in each partition are not enough to cover the actual top-3 HUIs, $\{i_2, i_3, i_6\}$ will be missed obviously.

4. Baseline algorithm

Motivated by the limitation of EA, this section first devises a baseline algorithm *BA*, which utilizes the apriori-like level-wise execution mode to discover the top-k HUIs on massive transaction database T.

Before providing the details of BA, we first give some definitions used in this paper.

Definition 4.1 (*Transaction utility*). Given the transaction $t \in T$, its transaction utility $u(t,t)$ is defined as the sum of the utilities of the items in t, i.e. $u(t,t) = \sum_{i \in t} u(i,t)$.

Definition 4.2 (*Transaction-weighted utilization of an itemset*). Given an itemset X, its transaction-weighted utilization $twu(X)$ is defined as the sum of the transaction utilities of the transactions in T containing X, i.e. $twu(X) = \sum_{t \in g(X,T)} u(t,t)$, where $g(X,T)$ represents the set of transactions in T containing X.

Initially, at the first pass, BA maintains all single items $\{i_1, i_2, \ldots, i_d\}$ in C_1 (current candidate set). Then, BA performs a scan on T to compute the utility and the transaction-weighted utilization for each item.

Throughout the paper, the min–heap *MH*, which is initialized to empty, is used to keep the k itemsets with the greatest utilities. *Let δ represent the border utility threshold in this paper, which is the kth greatest utility of the itemsets found currently*. If *MH* contains k itemsets, $\delta = MH.min$, otherwise $\delta = 0$.

Given any 1-itemset X_1 in C_1, if $twu(X_1) < \delta, X_1$ can be removed from C_1, since X_1 and all its supersets are definitely not top-k HUIs by the *transaction-weighted downward closure property (TWDC property)* [21]. The remaining 1-itemsets in C_1 are used to generate the 2-itemsets, which are maintained in C_2. If C_2 contains no itemsets, BA terminates and reports the itemsets in *MH* as the required results. Otherwise, at the second pass, BA performs the second scan on T to calculate the utilities and transaction-weighted utilizations of 2-itemsets in C_2, updates *MH* and the value of δ, removes the 2-itemsets in C_2 by TWDC property, and generates the candidate 3-itemsets C_3 by the remaining 2-itemsets in C_2. BA enters the third pass. The process continues iteratively until the current candidate set is empty. Then BA terminates and reports the itemsets in *MH* as top-k HUIs. The candidate generation in BA is the same as that in Apriori algorithm [2].

The advantage of BA is that it can perform the top-k HUIM on massive transaction database, because BA does not need to maintain T in memory entirely but only retrieves one transaction every time. However, BA will generate many candidates in each pass, which incurs a relatively high computation cost to calculate the utilities of the candidates. Besides, BA needs to perform several passes of scan on T, which incurs a relatively high I/O cost on massive transaction database.

It is considered in this paper that, an efficient top-k HUIM algorithm on massive transaction database should have the following three characteristics: (i) it generates the complete set of top-k HUIs, (ii) it guarantees that the memory can keep the current working data set entirely, (iii) it reduces the processing cost as much as possible. In the following sections, this paper proposes a novel algorithm, which satisfies the three characteristics, to perform the top-k HUIM on massive transaction database efficiently.

5. Prefix-based partitioning

In this paper, we assume that *there is an imposed order for the items, and the items in any itemset are arranged in the order*. Given the universe of items $\mathscr{I} = \{i_1, i_2, \ldots, i_d\}$, without loss of generality, the imposed order is defined as $i_1 \prec i_2 \prec \ldots \prec i_d$. All possible itemsets G can be divided into up to d *non-overlapping and complete* groups $GR_1, GR_2, \ldots, GR_d$, where $GR_a (1 \leqslant a \leqslant d)$ contains the itemsets beginning with the item i_a. As a result, $\forall 1 \leqslant a \neq b \leqslant d, GR_a \cap GR_b = \varnothing, \bigcup_{1 \leqslant a \leqslant d} GR_a = G$. Obviously, all itemsets can be generated in groups. That is to find $GR_1, GR_2, \ldots, GR_d$ one by one. In the process of generating $GR_a (1 \leqslant a \leqslant d)$, we only need to involve the corresponding subset of the transactions in T.

In this paper, we devise a *prefix-based partitioning strategy*, which splits the transaction database T into d prefix-based partitions $P_1, P_2, \ldots, P_d$. The partition $P_a (1 \leqslant a \leqslant d)$ keeps the transactions with the prefix item i_a, which are generated from the transactions in T. $\forall t \in T$, *the transactions in the involved prefix-based partitions contain the suffix of t*.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

TID	*Transactions in* T_1
1	$(i_3,3),(i_5,3),(i_6,6),(i_7,6)$
2	$(i_2,3),(i_3,5),(i_5,3),(i_6,4),(i_7,2)$
3	$(i_1,5),(i_2,2),(i_5,2),(i_6,2),(i_7,3)$
4	$(i_2,2),(i_3,4),(i_6,6),(i_7,3)$

TID	*Transactions in* T_2
5	$(i_1,6),(i_2,4),(i_3,2),(i_5,4),(i_6,3),(i_7,1)$
6	$(i_1,1),(i_4,4)$
7	$(i_1,5),(i_2,1),(i_3,1),(i_4,6),(i_5,3),(i_6,4),(i_7,5)$
8	$(i_2,5),(i_3,5),(i_5,5),(i_6,1),(i_7,4)$

TID	*Transactions in* T_3
9	$(i_2,1),(i_3,3),(i_4,6),(i_5,1),(i_6,5)$
10	$(i_1,1),(i_2,3),(i_6,1),(i_7,5)$

	possible itemsets in T_1	*local utility*
1	$\{i_3,i_6,i_7\}$	286
2	$\{i_2,i_3,i_6,i_7\}$	221
3	$\{i_3,i_6\}$	220
4	$\{i_3,i_5,i_6,i_7\}$	214
5	$\{i_6,i_7\}$	210
6	$\{i_2,i_3,i_6\}$	191
...	...	...
47	$\{i_1\}$	25

	possible itemsets in T_2	*local utility*
1	$\{i_2,i_3,i_5,i_6,i_7\}$	316
2	$\{i_2,i_3,i_6,i_7\}$	268
...	...	...
10	$\{i_2,i_3,i_6\}$	208
...	...	...
17	$\{i_3,i_6,i_7\}$	188
...	...	...
127	$\{i_5\}$	48

	possible itemsets in T_3	*local utility*
1	$\{i_2,i_3,i_4,i_5,i_6,i_7\}$	116
...	...	...
14	$\{i_2,i_3,i_6\}$	70
...	...	...
43	$\{i_5\}$	4
	$\{i_2,i_3,i_6,i_7\}$	0
	$\{i_3,i_6,i_7\}$	0

Fig. 4. The illustration of local results in the partitions of the running example.

5.1. The partitioning process

A sequential scan is performed on T. $\forall t \in T$, assume that the transaction t contains h distinct items, i.e. $t = \{t[1], t[2], \ldots, t[h]\}$, where $t[b]$ $(1 \leqslant b \leqslant h)$ represents the bth item in t and $t[1] \prec t[2] \prec \ldots \prec t[h]$. $\forall 1 \leqslant b \leqslant h$, let $t[b]$ be the ath item i_a of $\mathscr{I}$ (i.e. $t[b] = i_a$), $\{t[b], t[b+1], \ldots, t[h]\}$ will be outputted into P_a. When the sequential scan is over, the d prefix-based partitions are obtained. The pseudo-code of prefix-based partitioning is shown in Algorithm 1.

Algorithm 1 PrefixbasedPartition(T, $\mathscr{I}$)

bf Input: T is the transaction database, $\mathscr{I}$ is the universe of the items.
Output: Prefix-based partitions $P_1, P_2, \ldots, P_d$
1: **while** T has more transactions **do**
2: retrieve the next transaction t of T, $t = \{t[1], t[2], \ldots, t[h]\}$
3: **for** $b = 1$ to h **do**
4: Let $t[b]$ be the ath item in $\mathscr{I}$ $(t[b] = i_a)$
Write $\{t[b], t[b+1], \ldots, t[h]\}$ to partition P_a.
5: **end for**
6: **end while**

Example 5.1. The prefix-based partitions of the running example are illustrated in Fig. 5. The *TID* field in Fig. 5 represents the original transaction identifiers in the transaction database T. Let $T(tid)$ be the transaction in T with TID tid. Because $T(3), T(5), T(6), T(7)$ and $T(10)$ are five transactions which contain i_1, the partition P_1 keeps the five transactions. Besides, since $T(6), T(7)$ and $T(9)$ contain i_4, the suffixes of the four transactions beginning with i_4 are kept in P_4. The other partitions can be generated similarly.

It should be noted that the prefix-based partitioning is performed only once before top-k HUIM is executed. By the prefix-based partitioning, the partitions can be maintained easily for the newly coming transactions. That is to retrieve the new transactions and output the relevant suffixes at the end of the corresponding partitions.

With the partitioning strategy, this paper devises a novel PTM algorithm (**P**refix-partitioning-based **T**op-k high utility itemset **M**ining) to discover the top-k high utility itemsets on massive transaction database efficiently.

5.2. Discussion

Usually, the value of d is relatively large. For example, in an average supermarket, the number of the products can be in the thousands or even greater. In this paper, it is reasonable to assume that each partition $P_a (1 \leqslant a \leqslant d)$ can be held entirely in the memory. For the particular cases that $\exists a (1 \leqslant a \leqslant d)$, P_a cannot fit in memory, the splitting operation can be performed similarly on P_a to generate more partitions, each of which keeps the transactions with the same prefix 2-itemset. For the sake of clarity, we consider that each partition with the same prefix single-item can be held entirely in the memory throughout the paper.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

TID	Transactions in P_1
3	$(i_1,5),(i_2,2),(i_5,2),(i_6,2),(i_7,3)$
5	$(i_1,6),(i_2,4),(i_3,2),(i_5,4),(i_6,3),(i_7,1)$
6	$(i_1,1),(i_4,4)$
7	$(i_1,5),(i_2,1),(i_3,1),(i_4,6),(i_5,3),(i_6,4),(i_7,5)$
10	$(i_1,1),(i_2,3),(i_6,1),(i_7,5)$

TID	Transactions in P_2
2	$(i_2,3),(i_3,5),(i_5,3),(i_6,4),(i_7,2)$
3	$(i_2,2),(i_5,2),(i_6,2),(i_7,3)$
4	$(i_2,2),(i_3,4),(i_6,6),(i_7,3)$
5	$(i_2,4),(i_3,2),(i_5,4),(i_6,3),(i_7,1)$
7	$(i_2,1),(i_3,1),(i_4,6),(i_5,3),(i_6,4),(i_7,5)$
8	$(i_2,5),(i_3,5),(i_5,5),(i_6,1),(i_7,4)$
9	$(i_2,1),(i_3,3),(i_4,6),(i_5,1),(i_6,5)$
10	$(i_2,3),(i_6,1),(i_7,5)$

TID	Transactions in P_3
1	$(i_3,3),(i_5,3),(i_6,6),(i_7,6)$
2	$(i_3,5),(i_5,3),(i_6,4),(i_7,2)$
4	$(i_3,4),(i_6,6),(i_7,3)$
5	$(i_3,2),(i_5,4),(i_6,3),(i_7,1)$
7	$(i_3,1),(i_4,6),(i_5,3),(i_6,4),(i_7,5)$
8	$(i_3,5),(i_5,5),(i_6,1),(i_7,4)$
9	$(i_3,3),(i_4,6),(i_5,1),(i_6,5)$

TID	Transactions in P_4
6	$(i_4,4)$
7	$(i_4,6),(i_5,3),(i_6,4),(i_7,5)$
9	$(i_4,6),(i_5,1),(i_6,5)$

TID	Transactions in P_5
1	$(i_5,3),(i_6,6),(i_7,6)$
2	$(i_5,3),(i_6,4),(i_7,2)$
3	$(i_5,2),(i_6,2),(i_7,3)$
5	$(i_5,4),(i_6,3),(i_7,1)$
7	$(i_5,3),(i_6,4),(i_7,5)$
8	$(i_5,5),(i_6,1),(i_7,4)$
9	$(i_5,1),(i_6,5)$

TID	Transactions in P_6
1	$(i_6,6),(i_7,6)$
2	$(i_6,4),(i_7,2)$
3	$(i_6,2),(i_7,3)$
4	$(i_6,6),(i_7,3)$
5	$(i_6,3),(i_7,1)$
7	$(i_6,4),(i_7,5)$
8	$(i_6,1),(i_7,4)$
9	$(i_6,5)$
10	$(i_1,1),(i_2,3),(i_6,1),(i_7,5)$

TID	Transactions in P_7
1	$(i_7,6)$
2	$(i_7,2)$
3	$(i_7,3)$
4	$(i_7,3)$
5	$(i_7,1)$
7	$(i_7,5)$
8	$(i_7,4)$
10	$(i_7,5)$

Fig. 5. The illustration of prefix-based partitioning in the running example.

Some required data structures are pre-constructed to speed up the execution of PTM. For the partition $P_a(1 \leqslant a \leqslant d)$, the data structure UIP_a (*U*tility *I*nformation in *P*artition) is constructed. UIP_a is an array with $(d-a+1)$ elements, because the transactions in P_a begin with the item i_a. Each element in UIP_a contains four parts $(fitem, sitem, iutil, twu)$, which are described below.

- For the first element $UIP_a[1], UIP_a[1].fitem = i_a, UIP_a[1].sitem = i_a, UIP_a[1].iutil$ and $UIP_a[1].twu$ are the utility and the transaction-weighted utilization of $\{i_a\}$ in P_a, respectively.
- For the element $UIP_a[b]$ $(\forall\ b, 2 \leqslant b \leqslant d-a+1)$, $UIP_a[b].fitem = i_a, UIP_a[b].sitem = i_{a+(b-1)}, UIP_a[b].iutil$ and $UIP_a[b].twu$ are the utility and the transaction-weighted utilization of the 2-itemset $\{i_a, i_{a+(b-1)}\}$ in P_a, respectively.

The data structure UIP_a can be generated by a single scan on P_a and *the elements in UIP_a are sorted in the descending order of twu*. Because the size of $UIP_a(1 \leqslant a \leqslant d)$ usually is small and the value of d is relatively large, $UIP_1, \ldots, UIP_d$ can be stored continuously as a single file *UIP* in order to reduce the management overhead of the file system. For convenience of locating the beginning position of UIP_a in *UIP*, the data structure $UOM(mtwu, offs)$ (*U*IP information of *O*ffset and *M*aximum twu) with d elements is preconstructed, where $UOM[a].offs$ and $UOM[a].mtwu$ represent the offset of first element of UIP_a in *UIP* and the maximum twu value among UIP_a, respectively. PTM also stores a copy of *UIP* (denoted by UIP_{sor}) in the descending order of *iutil*.

Example 5.2. The UIP structure and UOM structure of the running example are visualized in Fig. 6. $\forall 1 \leqslant a \leqslant 7, UIP_a$ contains the information of P_a. As an example, $UIP_1[2].fitem = i_1$ and $UIP_1[2].sitem = i_2$, the utility of the 2-itemset $\{i_1, i_2\}$ in P_1 is 165, and its corresponding transaction-weighted utilization in P_1 is 424. $UIP_1, UIP_2, \ldots$, and UIP_7 are stored continuously as a single file *UIP*. In the paper, the data type of item is *integer* (4 *bytes*) and the data type of utility is *long* (8 *bytes*), each element in *UIP* occupies 24 bytes. The UOM structure is built on UIP structure. The UIP_{sor} structure, which is a sorted version of *UIP* in the descending order of *iutil* field, is illustrated in Fig. 6 also.

6. PTM algorithm

This section introduces PTM algorithm detailedly. The overview of the algorithm is first provided in Section 6.1 to help understand PTM better, followed by the description of the basic process in Section 6.2. The selection order of the partitions is discussed in Section 6.3, and the processing of the single partition is devised in Section 6.4.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

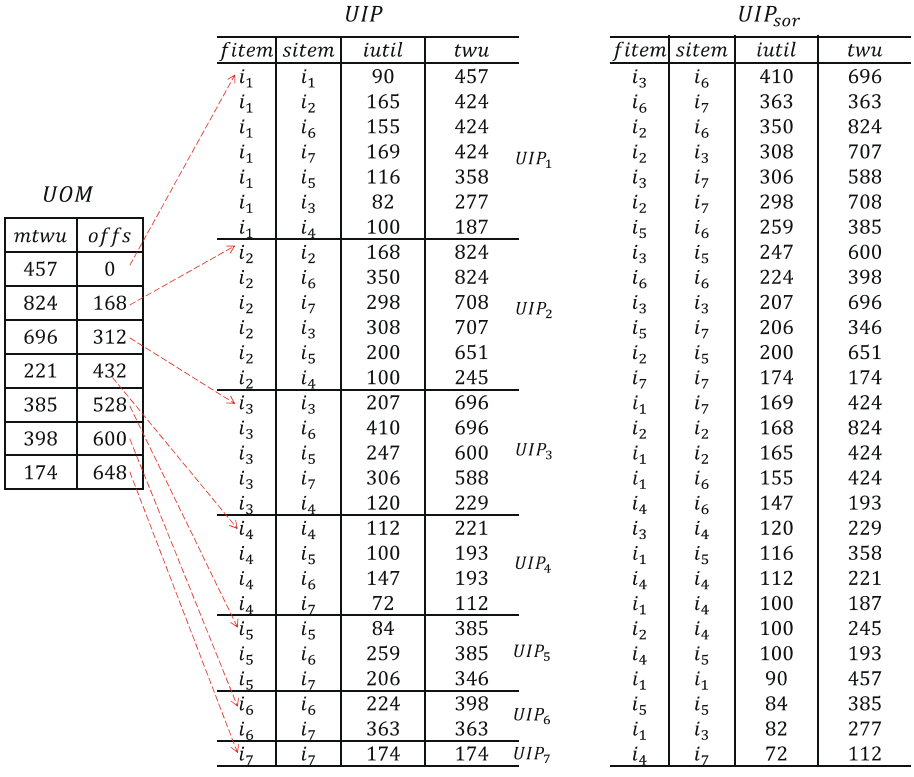


Fig. 6. The illustration of UIP structure in the running example.

6.1. The algorithm overview

Due to the prefix-based partitioning strategy, given the itemset $X = \{X[1], X[2], \ldots, X[h]\}$ and $X[1] = i_a$, the utility computation of X only needs the transactions in P_a and does not involve the transactions in other $(d-1)$ partitions. PTM processes the partitions one by one, maintains the candidate top-k HUIs found currently, and updates the border utility threshold δ. During the execution of PTM, some partitions can be skipped directly by the utility restriction of δ. In this way, the computational cost and I/O cost for PTM can be reduced correspondingly. When all the partitions are processed or skipped, PTM can return the top-k HUIs.

6.2. Basic description

Let the border utility threshold of the actual top-k HUIs be the *optimal utility threshold*. Evidently, if PTM can determine the optimal utility threshold quickly, the exploration space can be pruned significantly and the efficiency of PTM should be improved greatly. The problem here is that, unless the completion of the execution, it is rather difficult to acquire the optimal utility threshold in the first place. The best we can do is to raise the border utility threshold during the execution as quickly as possible.

6.2.1. Initialization of δ

At the beginning of the execution, PTM first computes the initial value of δ. Since the elements in UIP_{sor} are arranged in the descending order of *iutil* field, PTM retrieves k elements from UIP_{sor}. The min–heap MH, which is initialized to be empty, is utilized to maintain the k itemsets (1-itemsets or 2-itemsets) in UIP_{sor} with the greatest *iutil* values, and δ is set to $MH.min$, i.e. the minimum *iutil* value of the elements in MH. If there are less than k elements in UIP_{sor}, PTM only retrieves all of its elements and sets $\delta = 0$.

Example 6.1. As shown in Fig. 6, the initialization of δ in the running example is to retrieve 3 elements in $UIP_{sor}, MH = \{\{i_3, i_6\} : 410, \{i_6, i_7\} : 363, \{i_2, i_6\} : 350\}$, and the initial value of δ in the running example is set to 350.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

Given the set of partitions $P_1, P_2, \ldots, P_d$, PTM selects one partition at a time to process it. Let $P_a (1 \leqslant a \leqslant d)$ be the current partition to be selected. Before retrieving the transactions in P_a, PTM first checks whether $UOM[a].mtwu$ is smaller than δ. The data structure UOM is small enough and can be stored in memory at the very start. As proved in Theorem 6.1, if $UOM[a].mtwu < \delta, P_a$ cannot generate any top-k HUIs and PTM skips P_a directly. Then PTM selects the next partition.

Theorem 6.1. *Given the partition P_a and its corresponding UOM element $UOM[a]$, if $UOM[a].mtwu < \delta$, PTM can skip P_a directly.*

Proof 6.1. According to its construction method, UIP_a maintains the twu value of $\{i_a\}$ in P_a. Since all transactions in P_a contain the item i_a, the twu value of $\{i_a\}$ is the maximum among all twu values of the itemsets in P_a. $UOM[a].mtwu$ is set to the maximum twu value of UIP_a elements, i.e. $UOM[a].mtwu$ is equal to the twu value of $\{i_a\}$ in P_a. Hence, if $UOM[a].mtwu < \delta$, all possible itemsets in P_a have a lower utility than δ by TWDC property. In this case, the processing of P_a cannot generate any top-k HUIs, and PTM can skip P_a directly. Q.E.D.

Definition 6.1 (*Promising item*). Given the border utility threshold δ, an item $i \in \mathscr{I}$ is promising in partition P_a, if $twu(i, P_a) \geqslant \delta$, where $twu(i, P_a)$ is the transaction-weighted utilization of $\{i\}$ in P_a. Otherwise, it is unpromising.

If $UOM[a].mtwu \geqslant \delta$, it is possible for P_a to generate top-k HUIs, and PTM retrieves the elements of UIP_a sequentially (by locating the beginning position of $UOM[a].offs$ in UIP). Let uip be the element retrieved currently in UIP_a. If $uip.twu \geqslant \delta$, PTM maintains $uip.sitem$ into the set ps_a and retrieves the next element in UIP_a. The retrieval continues until the twu value of the currently retrieved UIP_a element is less than δ. According to Definition 6.1, PTM actually finds promising items in P_a. If $|ps_a| \leqslant 2$, PTM still does not need to retrieve transactions in P_a, because the possible top-k high utility 1-itemsets and 2-itemsets have been maintained in MH during the initialization of δ. Otherwise, PTM performs the processing of the partition P_a, updates MH and δ, as described in Section 6.4.

When all partitions have been processed or skipped, PTM terminates and the itemsets in MH are the required top-k HUIs. The pseudo-code of PTM algorithm is provided in Algorithm 2.

Algorithm 2 PTM Algorithm($P_1, P_2, \ldots, P_d$, k)
Input: $P_1, P_2, \ldots, P_d$ are the prefix-based partitions, k is the specified result size.
Output: The top-k high utility itemsets.
//The initialization of δ
1: MH is a min–heap and initialized to be empty **do**
2: Retrieve first k elements in UIP_{sor} into MH
3: $\delta = MH.min$ if $MH.size == k$, 0 otherwise
//The processing of the partitions
4: **while** there are remaining partitions **do**
5: Let P_a be the next selected partition
//If P_a cannot generate any top-k HUIs, skip it
6: **if** $UOM[a].mtwu < \delta$ **then**
7: **continue**
8: **end if**
9: Maintain the promising items of P_a in ps_a
//If the number of the promising items is no more than 2, skip P_a
10: **if** $|ps_a| \leqslant 2$ **then**
11: **continue**
12: **end if**
13: ProcessSinglePartition(P_a, ps_a, MH, δ)
14: **end while**
15: **return** the k itemsets in MH

6.3. The selection order of partitions

The simplest and most direct selection order of partitions is to select $P_1, P_2, \ldots, P_d$ sequentially. In the light of the description above, the border utility threshold is very important to the performance of top-k HUIM algorithm. As the execution continues, the quicker the border utility threshold reaches to the optimal utility threshold, the more candidates can be pruned and the more partitions can be skipped in PTM. Therefore, the selection order of the partitions is considered in this part.

The guiding idea for the selection order is to first retrieve the partitions that can increase the border utility threshold considerably by processing them. Naturally, the selection operation should not impose much cost on the overall execution, and the selection should be made by the information we already have or can easily acquired.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

The choice used in PTM is to retrieve the partitions in the order of *average transaction utility*, i.e. the ratio of the sum of the transaction utility over the transaction number in each partition. The intuition behind the choice is that, if the transaction has a high utility, it is likely to generate some itemsets with relatively high utilities. The average transaction utility of each partition can be pre-computed and then be used during the execution of PTM.

Example 6.2. In the running example, the average transaction utilities of the partitions are listed in Fig. 7. According to the results, the selection order of PTM is first P_2, next P_3, then P_1, P_4, P_5, P_6, and last P_7. Fig. 8 compares the effect of the different selection orders. In the selection order of average transaction utility, PTM can skip five partitions directly. Comparatively, PTM with sequential selection order can skip four partitions. In Fig. 8, "δ *before*" and "δ *after*" represent the value of δ before processing (or skipping) the partition and after processing (or skipping) the partition, respectively.

6.4. The processing of the single partition

The processing of the single partition certainly can be performed in the same way as the processing in BA (devised in Section 4). But, the level-wise execution needs to scan the partition several times. In this paper, we propose a new method to process single partition in PTM. Let P_a be the partition to be processed currently.

6.4.1. Unpromising item filtering

In order to process P_a, PTM loads all transactions in P_a into memory. For the currently retrieved transaction t, PTM discards the unpromising items in t directly since they do not contribute to the discovery of top-k HUIs. This can reduce the memory space further and save the computational cost for the following operation.

Example 6.3. In this part, we take P_2 to discuss the processing of the single partition. As illustrated in Fig. 9, given $ps_2 = \{i_2, i_3, i_5, i_6, i_7\}$ in the running example, the unpromising item i_4 is removed from the fifth transaction and the seventh transaction in P_2.

Given the promising item set ps_a, the set enumeration tree $\mathscr{E}$ is grown in depth-first order to discover the itemsets in P_a, whose utilities are not less than border utility threshold δ. Note that $\mathscr{E}$ also is dependent on the specified ordering of the items. Each node in $\mathscr{E}$ has six fields $(item, pat, iutil, child, tset, fsu)$, where *item* represents the item labelled in the node. Given a node e in $\mathscr{E}$, the items on the path from root to e constitute the itemset $e.pat$. The field $e.iutil$ is the utility of $e.pat$ in P_a. The field $e.child$ keeps the child nodes of e. The field $e.tset$ represents the subset of transactions in P_a which contain $e.pat$. The field $e.fsu$ is the *full suffix utility* of $e.pat$ in $e.tset$, which is defined formally in Definition 6.3. According to Theorem 6.2, if $e.fsu < \delta$, e does not need to be explored further in the depth-first search. The subtree pruning rule is devised as below.

> **Subtree pruning rule:** In the depth-first search of $\mathscr{E}$, given the currently explored node e, if $e.fsu < \delta$, the subtree rooted in e can be pruned entirely.

Definition 6.2. [The full suffix utility of an itemset in a transaction] The full suffix utility $fsu(X, t)$ of an itemset X in a transaction t, which contains X, is defined as $fsu(X, t) = \sum_{i \in X} u(i, t) + \sum_{lpos(X)+1 \leqslant b \leqslant size(t)} u(t[b], t)$, where $lpos(X)$ is the position of the last item of X in t, $size(t)$ is the number of items in t and $t[b]$ is the b*th* item in t.

Definition 6.3. [The full suffix utility of an itemset in a transaction set] The full suffix utility $fsu(X, tset)$ of an itemset X in a transaction set $tset$, each of which contains X, is defined as $fsu(X, tset) = \sum_{t \in tset} fsu(X, t)$.

Theorem 6.2. *During the depth-first search on $\mathscr{E}$, let e_1 be the node considered currently, if $e_1.fsu < \delta$, for any node e_2 in the subtree rooted in e_1, we have $e_2.iutil < \delta$.*

Proof 6.2. Given that e_2 is a descendant node of e_1 in $\mathscr{E}$, $e_1.pat \subset e_2.pat$, and $e_1.tset \supseteq e_2.tset$, since any transaction which contains $e_2.pat$ definitely contains $e_1.pat$. According to the definitions of utility and full suffix utility, we have,

partition	P_1	P_2	P_3	P_4	P_5	P_6	P_7
average transaction utility	91.4	103	99.4	73.7	55	44.2	21.8

Fig. 7. The illustration of average transaction utilities of the partitions.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

the retrieval of the partition in the order of average transaction utility

order	*mtwu*	*δ before*	*operation*	*promising items*	*δ after*
initialization	–	0	–	–	350
P_2	824	350	*retrieval*	$\{i_2, i_3, i_5, i_6, i_7\}$	445
P_3	696	445	*retrieval*	$\{i_3, i_5, i_6, i_7\}$	469
P_1	457	469	*skip*	–	469
P_4	221	469	*skip*	–	469
P_5	385	469	*skip*	–	469
P_6	398	469	*skip*	–	469
P_7	174	469	*skip*	–	469

the retrieval of the partition in the sequential order

order	*mtwu*	*δ before*	*operation*	*promising items*	*δ after*
initialization	–	0	–	–	350
P_1	457	350	*retrieval*	$\{i_1, i_2, i_5, i_6, i_7\}$	350
P_2	824	350	*retrieval*	$\{i_2, i_3, i_5, i_6, i_7\}$	445
P_3	696	445	*retrieval*	$\{i_3, i_5, i_6, i_7\}$	469
P_4	221	469	*skip*	–	469
P_5	385	469	*skip*	–	469
P_6	398	469	*skip*	–	469
P_7	174	469	*skip*	–	469

Fig. 8. The illustration of selection order in the running example.

	Transactions in P_2 before filtering	*transactions in P_2 after filtering*
1	$(i_2,3),(i_3,5),(i_5,3),(i_6,4),(i_7,2)$	$(i_2,3),(i_3,5),(i_5,3),(i_6,4),(i_7,2)$
2	$(i_2,2),(i_5,2),(i_6,2),(i_7,3)$	$(i_2,2),(i_5,2),(i_6,2),(i_7,3)$
3	$(i_2,2),(i_3,4),(i_6,6),(i_7,3)$	$(i_2,2),(i_3,4),(i_6,6),(i_7,3)$
4	$(i_2,4),(i_3,2),(i_5,4),(i_6,3),(i_7,1)$	$(i_2,4),(i_3,2),(i_5,4),(i_6,3),(i_7,1)$
5	$(i_2,1),(i_3,1),(i_4,6),(i_5,3),(i_6,4),(i_7,5)$	$(i_2,1),(i_3,1),(i_5,3),(i_6,4),(i_7,5)$
6	$(i_2,5),(i_3,5),(i_5,5),(i_6,1),(i_7,4)$	$(i_2,5),(i_3,5),(i_5,5),(i_6,1),(i_7,4)$
7	$(i_2,1),(i_3,3),(i_4,6),(i_5,1),(i_6,5)$	$(i_2,1),(i_3,3),(i_5,1),(i_6,5)$
8	$(i_2,3),(i_6,1),(i_7,5)$	$(i_2,3),(i_6,1),(i_7,5)$

Fig. 9. The illustration of unpromising item filtering in P_2.

$$(i)\quad e_2.iutil = \sum_{t\in e_2.tset}\sum_{i\in e_2.pat} u(i,t) = \sum_{t\in e_2.tset}\left[\sum_{i\in e_1.pat} u(i,t) + \sum_{j\in e_2.pat-e_1.pat} u(j,t)\right],$$

$$(ii)\quad e_1.fsu = fsu(e_1.pat, e_2.tset) + fsu(e_1.pat, e_2.tset - e_1.tset),$$

$$(iii)\quad fsu(e_1.pat, e_2.tset) = \sum_{t\in e_2.tset}\left[\sum_{i\in e_1.pat} u(i,t) + \sum_{lpos(e_1.pat)+1\leqslant b\leqslant size(t)} u(t[b],t)\right].$$

Given $t \in e_2.tset, \sum_{j\in e_2.pat-e_1.pat} u(j,t) \leqslant \sum_{lpos(e_1.pat)+1\leqslant b\leqslant size(t)} u(t[b],t)$. Therefore, $e_2.iutil \leqslant fsu(e_1.pat, e_2.tset) \leqslant e_1.fsu$. If $e_1.fsu < \delta, e_2.iutil < \delta$. Q.E.D.

6.4.2. Pseudo-copying

The transactions in P_a are kept in the memory as an array, $P_a[1], P_a[2], \ldots, P_a[|P_a|]$. Given the node e in $\mathscr{E}$, $e.tset$ does not store the relevant transactions physically, but only keeps the pointers of the transactions. This can reduce the required memory space significantly. For simplicity, when describing the algorithm execution, we still take the pointers of the transactions as the physical copies of the transactions.

When dealing with P_a, the root r of $\mathscr{E}$ is created first, with $r.item = i_a, r.pat = i_a$ and $r.tset = P_a$. The values of $r.iutil$ and $r.fsu$ can be computed correspondingly. $\forall j \in ps_a, j$ is one promising item in ps_a, let $idx(j, ps_a)$ be the positional index of the item j in ps_a. For example, the value of positional index of the first item in ps_a is 1.

6.4.3. The processing of depth-first search

Initially, let the currently explored node e be the root of $\mathscr{E}$, the depth-first search begins. If $e.fsu < \delta$, e does not need to be expanded further and the depth-first search continues to the next node. Otherwise, this means that it is possible to generate top-k HUIs in the subtree rooted in e, PTM needs to expand e to the next level. Of course, if $e.iutil \geqslant \delta, e.pat$ is a candidate

top-k HUIs, *we need to update MH and the value of* δ. Given the item $e.item$ labelled in e, the items used to expand e include $ps_a[idx(e.item, ps_a)+1], ps_a[idx(e.item, ps_a)+2], \ldots, ps_a[|ps_a|]$. In other words, e can have $(|ps_a| - idx(e.item, ps_a))$ child nodes. Let the bth child node of e be e_b, we have $e_b.item = ps_a[idx(e.item, ps_a)+b], e_b.pat = e.pat + e_b.item$ where "+" is a string concatenation operator, $e_b.tset$ is a subset of $e.tset$ which contains $e_b.item$ also. The values of $e_b.iutil$ and $e_b.fsu$ can be computed correspondingly. Then the first child node is selected in the depth-first search. This node can be processed similarly as mentioned above. When there is no more node to be explored in $\mathscr{E}$, the processing of P_a is terminated.

The pseudo-codes of the processing of single partition are provided in Algorithm 3 and Algorithm 4.

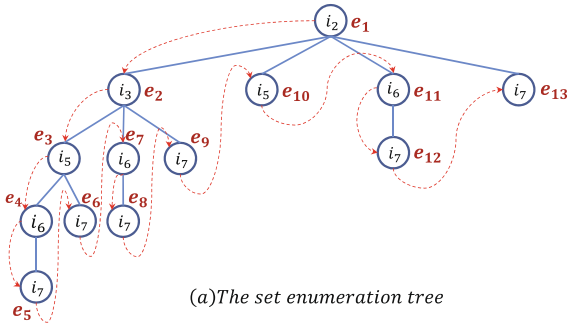
Algorithm 3: Process SinglePartition(P_a, ps_a, MH, δ)
Input: P_a is the current partition, ps_a is the promising item set, MH is min heap, δ is border utility threshold.
Output: The itemsets in P_a whose utilities is not less than δ.
//Unpromising item filtering
1: **while** P_a has more transactions **do**
2: Retrieve the next transaction t in P_a
3: Discard unpromising items of t not contained in ps_a
4: **end while**
//The depth-first search
5: Create root node r of $\mathscr{E}$, $r.item = i_a$, $r.pat = i_a$, $r.tset = P_a$ by pseudo-copy, compute $r.iutil$ and $r.fsu$ by $r.tset$
6: DepthFirstSearch(r, ps_a, MH, δ)

Algorithm 4: Depth First Search (e, ps_a, MH, δ)

Input e is the current node in $\mathscr{E}$, ps_a is the promising item set, MH is min heap, δ is border utility threshold.
Output: The itemsets in P_a whose utilities are not less than δ.
//Subtree pruning rule
1: **if** $e.fsu < \delta$ **then**
2: **return**
3: **end if**
//Check whether the current itemset is a candidate top-k HUIs, do not consider 1-itemsets and 2-itemsets
4: **if** $(e.iutil \geqslant \delta \wedge e.pat.size \geqslant 3)$ **then**
5: Update MH and δ
6: **end if**
//Expand the current node e to the next level, $idx(i, ps_a)$: positional index of the item i in ps_a
7: **for** $b = 1$ to $(|ps_a| - idx(e.item, ps_a))$ **do**
8: Create child node e_b of e, $e_b.item = ps_a[idx(e.item, ps_a)+b]$, $e_b.pat = e.pat + ps_a[idx(e.item, ps_a)+b]$
Add e_b to $e.child$
9: **end for**
10: **for** *each* $t \in e.tset$ **do**
11: Add t to $e_b.tset$ if t contains the item $ps_a[idx(e.item, ps_a)+b]$, $1 \leqslant b \leqslant (|ps_a| - idx(e.item, ps_a))$
12: **end for**
13: **for** *each* $e_b \in e.child$ **do**
14: Compute $e_b.iutil$ and $e_b.fsu$ by $e_b.tset$
15: DepthFirstSearch(e_b, ps_a, MH, δ)
16: **end for**

Example 6.4. The processing of P_2 in the running example is depicted in Fig. 10. According to the selection order of average transaction utility, P_2 is the first partition to be processed. Before growing the set enumeration tree $\mathscr{E}$ (depicted in Fig. 10(a), (c)) shows the current $MH = \{\{i_3, i_6\}, \{i_6, i_7\}, \{i_2, i_6\}\}$ with $\delta = 350$. Initially, the root node e_1 is constructed and visited, the values of the six fields of e_1 are listed in Fig. 10(b). Since $e_1.fsu(740) > \delta(350)$, the depth-first search continues on e_1. The first child e_2 of e_1 is visited, whose $e_2.fsu(623) > \delta(350)$. The depth-first search continues on e_2 and the first child e_3 of e_2 is visited now. Because $e_3.iutil(320) < \delta(350)$, MH does not change. The depth-first search continues on e_3 due to $e_3.fsu(511) > \delta(350)$ and the first child e_4 of e_3 is visited. Here, since $e_3.iutil(439) > \delta(350)$, PTM updates MH and δ. The similar depth-first search continues until all of the nodes rooted at e_1 are visited. Next the second child e_{10} of e_1 is visited. Because $e_{10}.fsu(423) < \delta(439)$, there is no need to expand e_{10} further. The depth-first search continues until all the required nodes are visited, and the processing of P_2 is over.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028



(a)The set enumeration tree

node	item	pat	iutil	child	tset	fsu
e_1	i_2	$\{i_2\}$	168	$\{e_2, e_{10}, e_{11}, e_{13}\}$	{1,2,3,4,5,6,7,8}	740
e_2	i_3	$\{i_2, i_3\}$	308	$\{e_3, e_7, e_9\}$	{1,3,4,5,6,7}	623
e_3	i_5	$\{i_2, i_3, i_5\}$	320	$\{e_4, e_6\}$	{1,4,5,6,7}	511
e_4	i_6	$\{i_2, i_3, i_5, i_6\}$	439	$\{e_5\}$	{1,4,5,6,7}	511
e_5	i_7	$\{i_2, i_3, i_5, i_6, i_7\}$	437	{ }	{1,4,5,6}	437
e_6	i_7	$\{i_2, i_3, i_5, i_7\}$	353	{ }	{1,4,5,6}	353
e_7	i_6	$\{i_2, i_3, i_6\}$	469	$\{e_8\}$	{1,3,4,5,6,7}	559
e_8	i_7	$\{i_2, i_3, i_6, i_7\}$	489	{ }	{1,3,4,5,6}	489
e_9	i_7	$\{i_2, i_3, i_7\}$	363	{ }	{1,3,4,5,6}	363
e_{10}	i_5	$\{i_2, i_5\}$	200	{ }	{1,2,4,5,6,7}	423
e_{11}	i_6	$\{i_2, i_6\}$	350	$\{e_{12}\}$	{1,2,3,4,5,6,7,8}	488
e_{12}	i_7	$\{i_2, i_6, i_7\}$	445	{ }	{1,2,3,4,5,6,8}	445
e_{13}	i_7	$\{i_2, i_7\}$	298	{ }	{1,2,3,4,5,6,8}	298

(b)The process of depth first search

operation	MH	δ
before visit	$\{i_3, i_6\}$: 410, $\{i_6, i_7\}$, : 363, $\{i_2, i_6\}$: 350	350
visit e_1 to e_3	$\{i_3, i_6\}$: 410, $\{i_6, i_7\}$, : 363, $\{i_2, i_6\}$: 350	350
visit e_4	$\{i_2, i_3, i_5, i_6\}$: 439, $\{i_3, i_6\}$: 410, $\{i_6, i_7\}$, : 363	363
visit e_5	$\{i_2, i_3, i_5, i_6\}$: 439, $\{i_2, i_3, i_5, i_6, i_7\}$: 437, $\{i_3, i_6\}$: 410	410
visit e_6	$\{i_2, i_3, i_5, i_6\}$: 439, $\{i_2, i_3, i_5, i_6, i_7\}$: 437, $\{i_3, i_6\}$: 410	410
visit e_7	$\{i_2, i_3, i_6\}$: 469, $\{i_2, i_3, i_5, i_6\}$: 439, $\{i_2, i_3, i_5, i_6, i_7\}$: 437	437
visit e_8	$\{i_2, i_3, i_6, i_7\}$: 489, , $\{i_2, i_3, i_6\}$: 469, $\{i_2, i_3, i_5, i_6\}$: 439	439
visit e_9 to e11	$\{i_2, i_3, i_6, i_7\}$: 489, , $\{i_2, i_3, i_6\}$: 469, $\{i_2, i_3, i_5, i_6\}$: 439	439
visit e_{12}	$\{i_2, i_3, i_6, i_7\}$: 489, , $\{i_2, i_3, i_6\}$: 469, $\{i_2, i_6, i_7\}$: 445	445
visit e_{13}	$\{i_2, i_3, i_6, i_7\}$: 489, , $\{i_2, i_3, i_6\}$: 469, $\{i_2, i_6, i_7\}$: 445	445

(c)The content of MH

Fig. 10. The illustration of processing P_2 in the running example.

7. Performance evaluation

7.1. Experimental settings

To evaluate the performance of PTM, we implement it in Java with jdk-8u112-windows-x64. The experiments are executed on DELL Precision T3431 Workstation (Intel(R) Core(TM) i7-9700 CPU @ 3.00 GHz (8 cores) + 32G memory +64bit windows 10). In the experiments, the performance of PTM is first evaluated against BA, the baseline algorithm, because the existing algorithms cannot be utilized directly to mine top-k HUIs on massive data, as analyzed in Section 3.2. Nevertheless, for understanding the performance of PTM more comprehensively, Section 7.8 provides the comparison with the existing algorithms (TKO, KHMC, THUI and TONUP) on the small and medium-sized data sets, which can be kept entirely in memory. In this paper, we consider the problem of top-k HUIM on massive data, which is stored in a standalone computer. On the one hand, the current standalone computer usually is equipped with the disks of TB capacity, which is large enough. On the other hand, a better top-k HUIM algorithm locally can improve the overall performance of the parallel and distributed top-k HUIM algorithms obviously. This is the reason why we do not include the parallel and distributed algorithms [5,17] in the experiments. In our future work, we will consider how to extend PTM to the parallel and distributed environment.

In the experiments, we evaluate the performance of PTM in terms of several aspects: transaction number (n), the specified result size (k), the number of items (d), the average transaction width (w). The experiments are executed on synthetic data set and real data set. The synthetic data set is generated by the data generator in [2], in which the number of maximal potential large itemsets is set to be double that of the number of items, and the average maximal potentially frequent itemset size is set to 4. The used parameter settings are listed in Table 2. For the items of each transaction in synthetic data set, the internal utilities are generated uniformly in the range of [1,10]. The external utilities of the items are generated in the range of

[0.01, 10] using a log-normal distribution as in [20]. The real data set used is OnlineRetail data set from UCI Machine Learning Repository.[1] The external utilities and the internal utilities for the items in the real data set are generated similarly as mentioned above.

7.2. The pre-construction of prefix-based partitioning

Given $d = 1500$ and $w = 10$, this part introduces the results of the pre-construction in PTM. The pre-construction time consists of two parts: splitting the table and generation of the structure *UIP*. In the default setting, i.e. $n = 50 \times 10^6$, the total pre-construction time is 1031.327 s. Naturally, when the number of n is greater, the pre-construction time is longer. However, it should be noted that the prefix-based partitioning only needs to be performed once. The later incoming transactions can be processed similarly and incrementally.

The prefix-based partitioning keeps more transactions than the original transaction database since a single transaction contains multiple items, but many of the transactions in the prefix-based partitions are shorter than their original counterparts. Given $n = 50 \times 10^6, d = 1500$ and $w = 10$, PTM maintains 9.32 times more transactions than the original transaction database, but the total number of items in the prefix-based partitioning is 6 times more than that in the original transaction database. It is true that the space requirement in PTM is larger. And yet, the disk is becoming much larger and much cheaper currently (for example, the Seagate BarraCuda 3.5 of 8 TB is at the price of US$149.99[2]), the main-stream computer usually has a sufficiently large storage capacity. Comparatively, it is generally recognized that the time is much more important than the space. In this paper, PTM algorithm trades an acceptably larger space overhead for a much greater speedup ratio. As verified in the following experiments, PTM can discover top-k HUIs on massive data efficiently.

7.3. Exp 1: the effect of transaction number

Given $k = 1000, d = 1500$ and $w = 10$, experiment 1 evaluates the performance of PTM with varying transaction numbers. As shown in Fig. 11(a), with a greater value of n, the execution times of BA and PTM both increase quickly, and PTM runs 48.271 times faster than BA. The significant performance advantage is due principally to the less I/O cost and much fewer candidates. Because of the level-wise execution mode, BA requires several full scans on the transaction database. As depicted in Fig. 11(b), PTM involves 4.634 time less I/O cost than BA. Furthermore, BA has to maintain much more candidates than PTM, which incurs a much higher computation cost. As illustrated in Fig. 11(c), PTM maintains 429.798 times fewer candidates than BA.

By the prefix-based partitioning strategy, PTM not only decreases the transaction number in each partition, but also shortens the average transaction width. Moreover, since the itemsets are grouped by the first item in the prefix-based partitioning, the transaction-weighted utilization of each itemset is lowered in each partition correspondingly. This in turn reduces the promising items involved in the processing of single partition. The number of the promising items in experiment 1 is reported in Fig. 11(h). For the partitions which are not skipped, we record the occurrence number of the size of the promising item sets. For example, at $n = 5M$, there are 26 partitions which have 3 promising items and 19 partitions which have 4 promising items. Although the number of the total items is 1500, the number of the promising items involved in PTM is less than 34 (except for one point at $n = 10 \times 10^6$). The much fewer number of promising items makes the more efficient processing of single partition and also makes PTM skip many partitions directly. As depicted in Fig. 11(d), given the 1500 partitions in total, PTM only needs to retrieve a fraction of them.

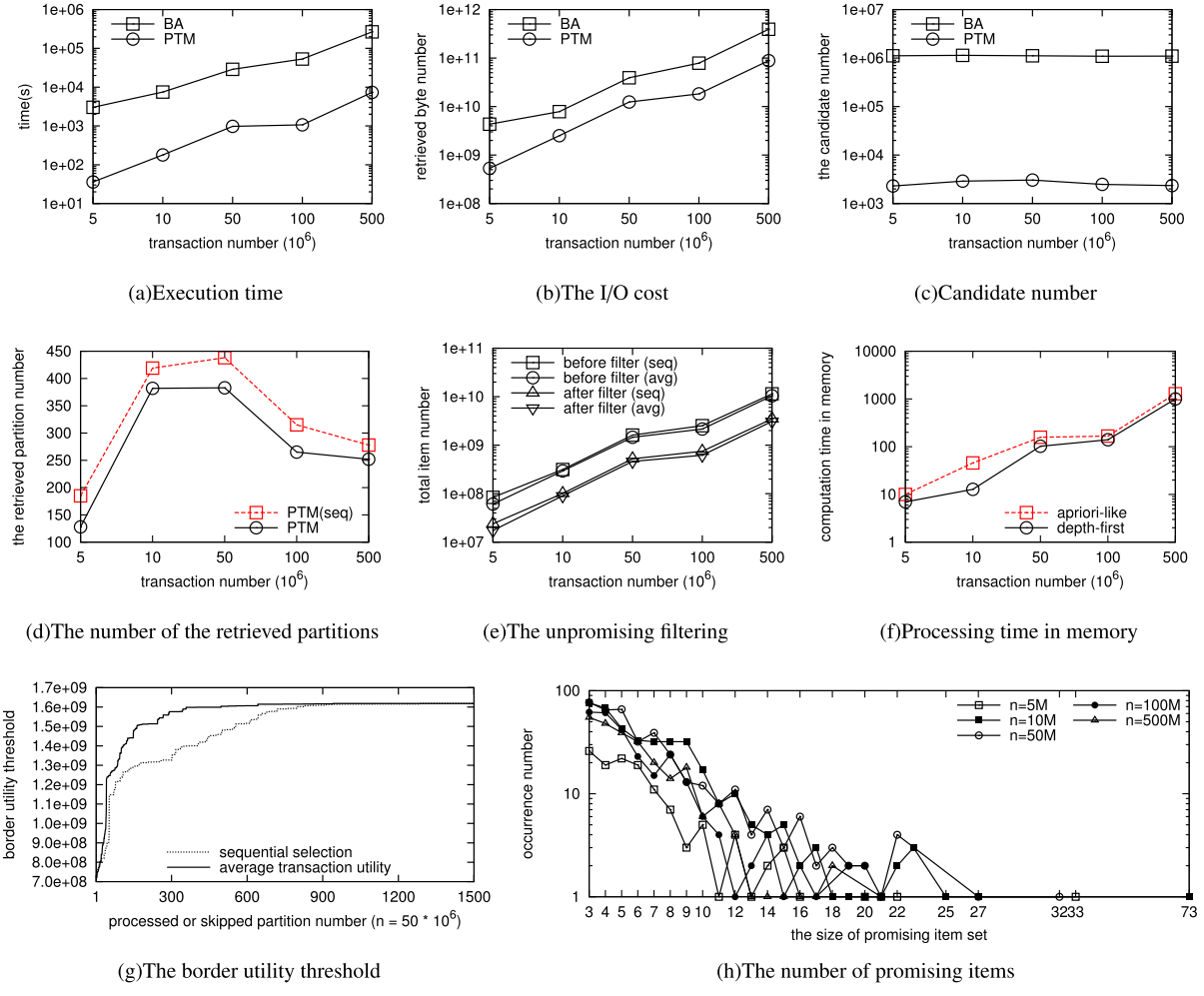
The effects of the adopted strategies in PTM, *selection order of average transaction utility*, *unpromising item filtering*, *depth-first search*, are illustrated also. Fig. 11(d) shows that, although the effect is not so significant, the selection order of average transaction utility can help to skip more partitions than sequential selection order. The effect of unpromising item filtering is illustrated in Fig. 11(e), in which "before filtering" and "after filtering" represent the total item number of the retrieved transactions in PTM before and after performing the filtering, respectively. On the one hand, unpromising item filtering can help to reduce the irrelevant items effectively. On the other hand, selection order of average transaction utility can filter more irrelevant items than the sequential selection order. This can be explained in Fig. 11(g), which shows that the border utility threshold with the selection order of average transaction utility increases faster than that with sequential selection order. Fig. 11(f) compares the depth-first search and the level-wise apriori-like processing. In order to make the comparison more reasonably, we start the apriori-like processing from the 3-itemsets directly, since the possible top-k HUIs of 1-itemsets and 2-itemsets have already been retrieved in the initialization of δ. As depicted in Fig. 11(f), the depth-first search has a better performance than the apriori-like processing. According to the results of the candidate number (Fig. 11) and the promising item size (Fig. 11(h)), the effect of subtree pruning rule can be reflected by pruning ratio, i.e. the one minus proportion of the explored nodes among the total nodes of the full set enumeration trees. It is found that a very small percentage of the nodes in the set enumeration tree needs to be visited in PTM, which verifies the good pruning effect of the subtree pruning rule.

[1] https://archive.ics.uci.edu/ml/datasets/Online+Retail.
[2] https://www.seagate.com/internal-hard-drives/hdd/barracuda/

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

Table 2
Parameter Settings.

Parameter	Used values
Transaction number(10^6) (syn)	5–500 (default: 50)
Result size (syn)	100–10000 (default: 1000)
The number of items (syn)	500–2500 (default: 1500)
The average transaction width (syn)	5–15 (default: 10)
Result size (real)	100–10000



(a)Execution time (b)The I/O cost (c)Candidate number

(d)The number of the retrieved partitions (e)The unpromising filtering (f)Processing time in memory

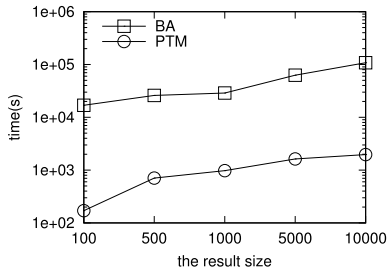
(g)The border utility threshold (h)The number of promising items

Fig. 11. The effect of transaction number.

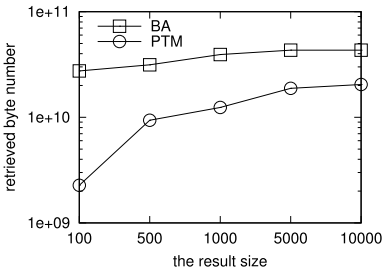
7.4. Exp 2: the effect of the result size

Given $n = 50 \times 10^6, d = 1500$ and $w = 10$, experiment 2 evaluates the performance of PTM with varying result sizes. As illustrated in Fig. 12(a), the execution times of both BA and PTM increase with a greater value of k. A larger value of k obviously leads to a lower border utility threshold, which, in return, affects the performance of the algorithm. In average, PTM runs 51.795 times faster than BA. The performance advantage of PTM comes from the less I/O cost and the fewer candidates. As illustrated in Fig. 12(b), PTM involves 4.6 times less I/O cost than BA, and the advantage gap between them decreases gradually with a greater value of k. In experiment 2, when k increases from 100 to 10000, the scan pass number of BA increases from 7 to 11, while the number of retrieved partitions in PTM rises from 56 to 866, which is depicted in Fig. 12(d). Comparatively, the different result sizes have a larger influence on PTM. As depicted in Fig. 12(c), PTM maintains 980.94 times fewer candidates than BA.

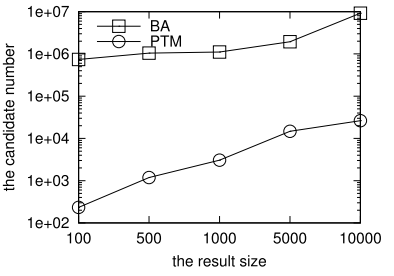
Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028



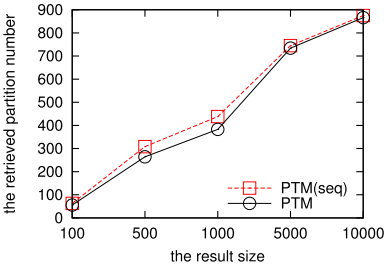
(a)Execution time



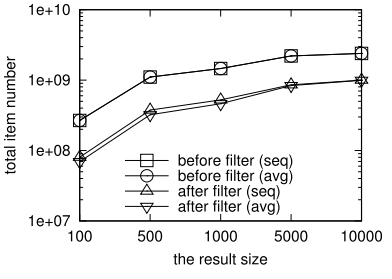
(b)The I/O cost



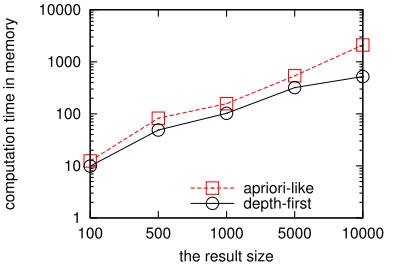
(c)Candidate number



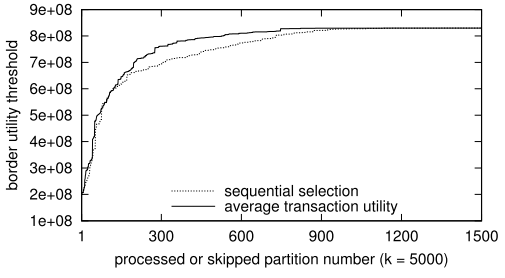
(d)The number of the retrieved partitions



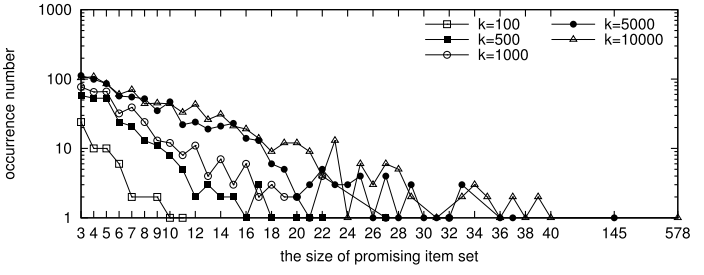
(e)The unpromising filtering



(f)Processing time in memory



(g)The utility threshold



(h)The number of promising items

Fig. 12. The effect of the result size.

As shown in Fig. 12(d), (e) and (g), the selection order of average transaction utility can skip more partitions, filter more irrelevant items and raise the border threshold more quickly, compared with the sequential selection. Fig. 12(f) shows that the depth-first search can be executed in memory faster than apriori-like processing, and the advantage gap is widened with a greater value of k.

The number of the promising items is illustrated in Fig. 12(h). Obviously, the overwhelming majority of the processed partitions have a rather small size of promising items. At $k \leqslant 1000$, the maximum size of the promising item set is 32, which is rather small compared with the 1500 items in total. At $k = 5000$ and 10000, the maximum sizes of the promising items are 145 and 578, respectively. This conforms with our analysis that a greater value of k leads to a lower border utility threshold and also a larger promising item set. However, among the partitions to be processed, the percentages of the partitions, whose sizes of promising item sets are not more than 32, are greater than 0.984 at $k = 5000$ and 0.958 at $k = 10000$. At $k = 5000$ and 10000, most of the partitions to be processed still have a small promising item set. According to the results of the candidates and the promising item numbers, the subtree pruning rule can reduce the exploration space of the set enumeration tree significantly.

7.5. Exp 3: the effect of the number of items

Given $n = 50 \times 10^6, k = 1000$ and $w = 10$, experiment 3 evaluates the performance of PTM with varying numbers of items. As illustrated in Fig. 13(a), although with some degree of variation, on the whole, the execution times of BA and PTM do not change significantly with the different values of d, and PTM runs 52.326 times faster than BA. As shown in Fig. 13(b), PTM involves 3.691 times less I/O cost than BA. With a greater value of d, the I/O cost of PTM decreases gradually, because the proportion of the partitions retrieved actually is reduced, as depicted in Fig. 13(d). Since the different values of d generate a different number of partitions, we report in Fig. 13(d) the percentage of the partitions actually processed among

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

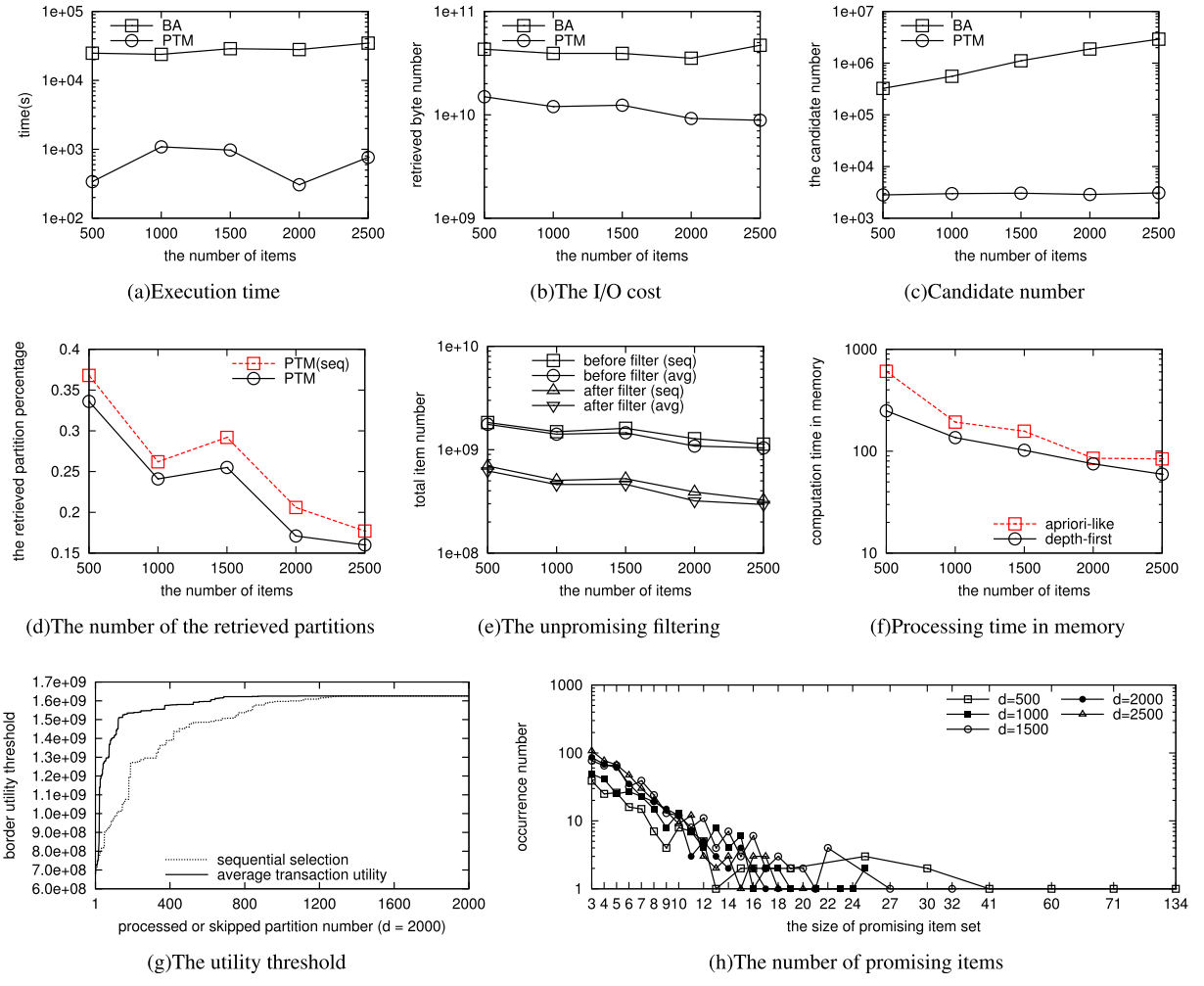


Fig. 13. The effect of the number of the items.

all partitions. The prefix-based partitioning strategy splits the transaction database into a set of partitions, each of which contains the transactions beginning with a certain item. Given a fixed number of transactions and a fixed average transaction width, a greater value of d means more partitions with fewer and shorter transactions overall. Of course, this makes a larger proportion of the partitions to be skipped given the border utility threshold. As illustrated in Fig. 13(c), PTM maintains 451.214 times fewer candidates than BA averagely, and the gap is widening with a greater value of d. This can be explained as follows. Given a greater value of d, the number of the possibly generated itemsets increases naturally. In terms of the execution mode, BA has to maintain more candidates. For PTM, the effect of more items can be offset by the unpromising item filtering, a larger proportion of partitions to be skipped and the subtree pruning rule.

The selection order of average transaction utility has a better effect than the sequential selection order, which is verified in Fig. 13(d), (e) and (g). And the depth-first search has a better performance than the apriori-like processing, as shown in Fig. 13(f). The number of the promising items is illustrated in Fig. 13(h). With few exceptions, the overwhelming majority of the processed partitions have the promising item sets of very small size. According to the results of the candidate number and the promising item number, subtree pruning rule can reach a very high pruning ratio (greater than 0.9985 in experiment 3).

7.6. Exp 4: the effect of the average transaction width

Given $n = 50 \times 10^6, k = 1000$ and $d = 1500$, experiment 4 evaluates the performance of PTM with varying transaction widths. As shown in Fig. 14(a), with a greater value of w, the execution times of BA and PTM both increase quickly, and averagely PTM runs 77.72 times faster than BA. The greater average transaction width makes each transaction keep more items. This enlarges the overall size of transaction database. Of course, the I/O cost of BA should increase with a greater value of w, just as illustrated in Fig. 14(b). Corresponding, the I/O cost of PTM has more substantial increase than that of BA, for the following reasons. Given a fixed number of d, a greater value of w means that the partitions in prefix-based partitioning can

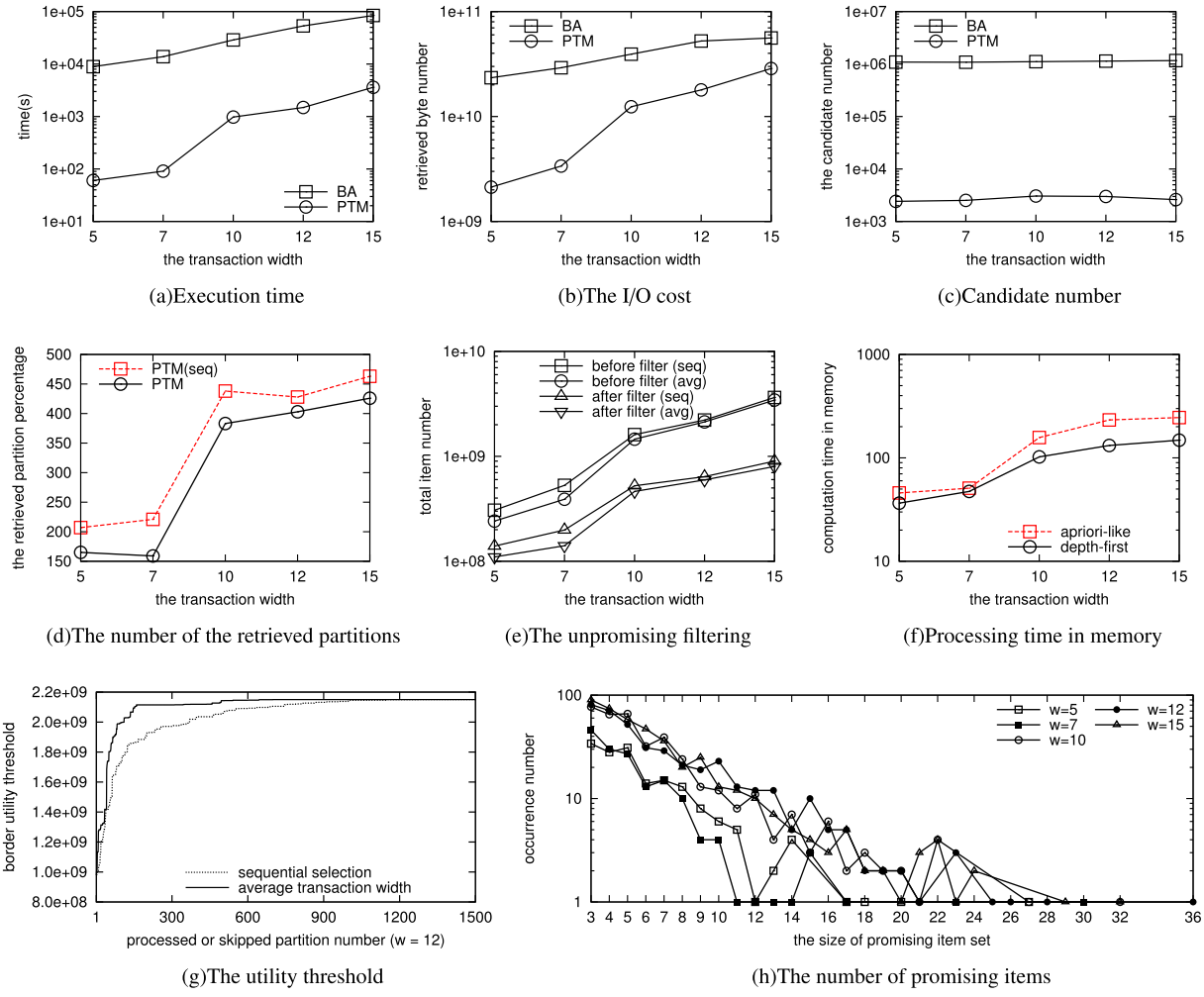


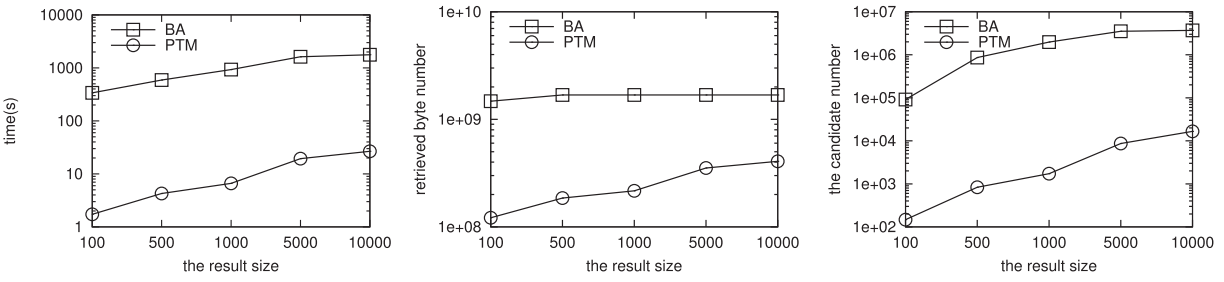
Fig. 14. The effect of the average transaction width.

maintain more transactions with greater widths. Thus, more partitions should be retrieved with a greater value of *w*, which is verified in Fig. 14(d). On the whole, PTM involves 5.544 times less I/O cost than BA. The number of the maintained candidates is reported in Fig. 14(c). The candidate number of each algorithm remains basically unchanged in experiment 4 and PTM maintains 414.918 times fewer candidates than BA. Because the number of the items is fixed in experiment 4 and BA adopts the apriori-like processing, the candidate number maintained by BA remains unchanged by and large. By the effective unpromising item filtering and the subtree pruning operation, the number of the candidates maintained by PTM does not vary much with the different values of *w*. The effect of selection order of average transaction utility is illustrated in Fig. 14(d), (e) and (g). Fig. 14(f) depicts that the depth-first search performs better than apriori-like processing. Fig. 14(h) illustrates that the number of the promising items involved in the processing of depth-first search is rather small, compared with the total number of the items.

7.7. Exp 5: real data – OnlineRetail Data Set

The real data used in experiment 5 is OnlineRetail data set. This is a transactional data set which contains all the transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail. The OnlineRetail data set contains 541909 transactions and 2603 items. In order to describe the performance of PTM on a relatively large data set, the transactions in the data set are copied ten times. The internal utilities of the transactions are generated independently. The experiment 5 evaluates the performance of PTM with varying result sizes. As depicted in Fig. 15(a), with a greater value of result sizes, the execution times of BA and PTM both increase obviously, and averagely PTM runs 125.072 times faster than BA. As shown in Fig. 15(b), PTM involves 7.587 times less I/O cost than BA. The number of the can-

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028



(a)Execution time (b)The I/O cost (c)Candidate number

Fig. 15. The effect of the real data.

didates maintained by each algorithm is illustrated in Fig. 15(c), and PTM maintains 685.084 times fewer candidates than BA. The explanation of the variation trend in experiment 5 can be described similarly as that in Section 7.4.

7.8. Exp 6: the comparison with the existing algorithms

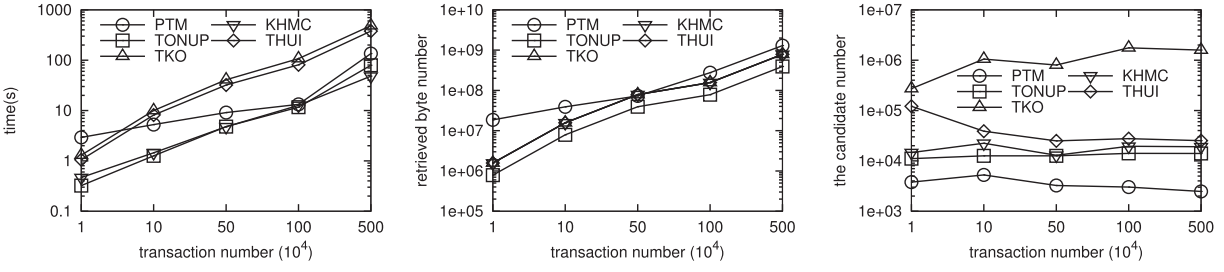
In order to evaluate the performance of PTM more comprehensively, we include in the experimental evaluation the existing top-k HUIM algorithms TKO [27], KHMC [7], THUI [15], TONUP [19] on the synthetic and real-life data of small and medium size.

- The first three algorithms (TKO, KHMC, THUI) are based on the utility list structure, in which each item(set) is associated with a utility list. After constructing the initial utility lists of items, the utility list of the candidate itemset (Pxy) can be acquired by merging the utility lists of Px and Py, where P is an itemset (could be empty), Px and Py are the itemsets by appending the items x and y to P respectively. TKO incorporates four strategies, PE, DGU, RUZ and EPB, to improve the efficiency. Besides utility list structure, KHMC utilizes EUCST structure to store item co-occurrence information and avoid the join operation. KHMC adopts RIU, CUD, COV, RUC, EA, TEP and EUCPT strategies to improve the efficiency. THUI introduces the LIU data structure to hold the utility information of a contiguous item set, and adopts RIU, LIU-E, LIU-LB, RUC and EA strategies.
- The fourth algorithm, TONUP, does not utilize the utility list structure, but adopts an opportunistic pattern growth approach, which treats the pattern enumeration as prefix extensions. Five opportunistic strategies, AutoMaterial, Dyna-Descend, ExactBorder, SuffixTree and OppoShift, are proposed in TONUP.

7.8.1. Synthetic data sets

For the synthetic data sets, the experiment in this part adopts a similar setting of the previous experiments (Experiment 1 to 4) with a smaller data set, which can be kept entirely in memory. In first four groups of experiment 6, the performance of PTM is evaluated on the synthetic data sets in terms of the transaction number (from 1×10^4 to 500×10^4), the result size (from 100 to 10000), the number of the items (from 500 to 2500) and the average transaction width (from 5 to 15).

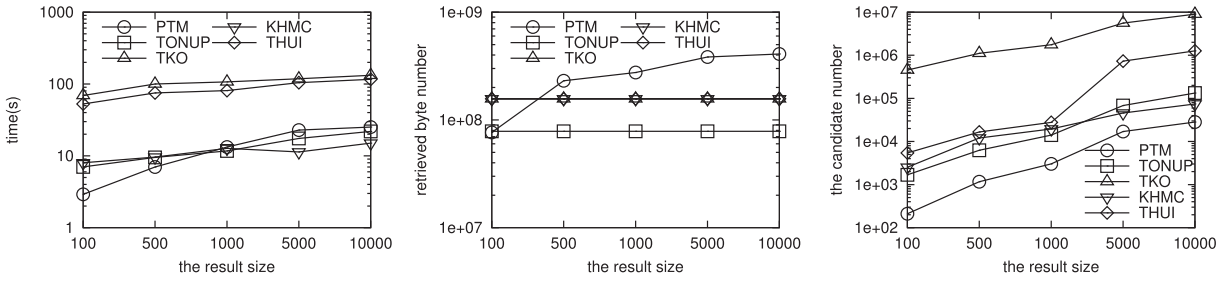
As illustrated in Fig. 16–19, on the small and medium-sized synthetic data set, the execution time of PTM is comparable to the existing algorithms, although PTM runs generally slower than the best of the existing algorithms. The I/O cost of PTM usually is higher than the existing algorithms. PTM adopts the prefix-based partitioning and has to retrieve the relevant partitions, while the existing algorithms only require single-pass or two-pass scan on the transactional database. By prefix-



(a)Execution time (b)The I/O cost (c)Candidate number

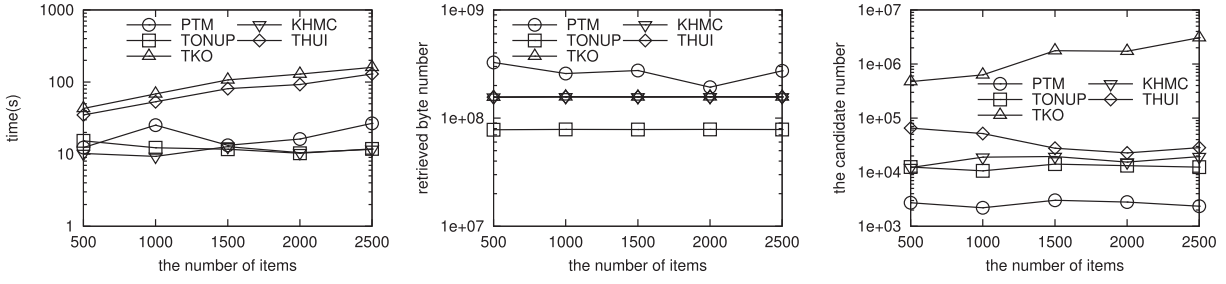
Fig. 16. The first group of experiment 6 (synthetic data of varying transaction numbers: $k = 1000, d = 1500, w = 10$).

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028



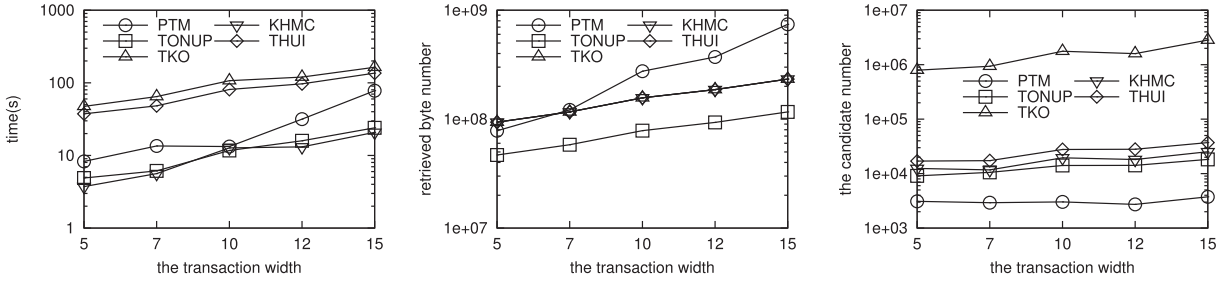
(a)Execution time (b)The I/O cost (c)Candidate number

Fig. 17. The second group of experiment 6 (synthetic data of varying result sizes: $n = 1M, d = 1500, w = 10$).



(a)Execution time (b)The I/O cost (c)Candidate number

Fig. 18. The third group of experiment 6 (synthetic data of varying numbers of items: $n = 1M, k = 1000, w = 10$).



(a)Execution time (b)The I/O cost (c)Candidate number

Fig. 19. The third group of experiment 6 (synthetic data of varying average transaction widths: $n = 1M, k = 1000, d = 1500$).

based partitioning, PTM can reduce the number of promising items significantly, and it maintains the least number of candidates among the first four groups of the experiment 6.

7.8.2. Real-life data sets

For the real-life data sets, chainstore, retail, foodmart and mushroom are obtained from the SPMF data mining library [8]. The characteristics of the real-life data sets are depicted in Table 3. In last four groups of experiment 6, the performance of PTM is evaluated on the real-life data sets with varying result sizes (1, 10, 100, 500 and 1000).

As illustrated in Figs. 20, 21 and 22, on the chainstore, retail and foodmart data sets, the execution time of PTM generally is much slower than that of the best existing algorithm. But its performance is still comparable to the other existing algorithms. The I/O cost of PTM increases quickly as more results are required, since it needs to retrieve more prefix-based partitions, while the existing algorithms only require one-pass or two-pass scan on the data. However, PTM still maintains the least number of the candidates compared to the existing algorithms, attributing to the prefix-based partitioning and subtree pruning.

It can be seen from Fig. 23 that, PTM runs much slower than all of the existing algorithms on mushroom data set, which is highly dense in nature. Here, the I/O cost of PTM is always much higher than that of the existing algorithms. Besides, as more results are required, the number of the maintained candidates in PTM increases quickly, and gradually becomes the largest in

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

Table 3
The characteristic of the real-life data sets.

Dataset	Transaction number	Item number	Average transaction width
chainstore	1112949	46086	7.2
retail	88162	16470	10.3
foodmart	4141	1559	4.5
mushroom	8124	119	23

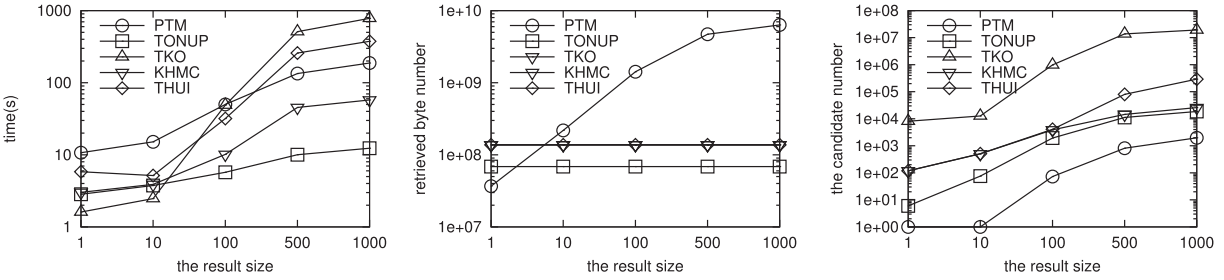


Fig. 20. The fifth group of experiment 6 (chainstore data: $n = 1112949, d = 46086, w = 7.2$).

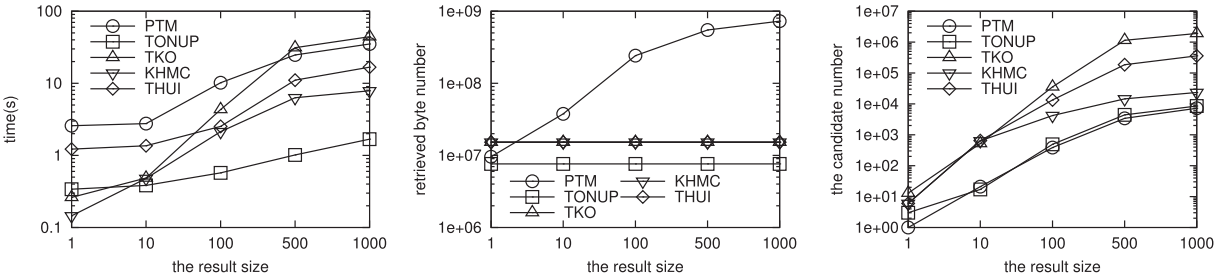


Fig. 21. The sixth group of experiment 6 (retail data: $n = 88162, d = 16470, w = 10.3$).

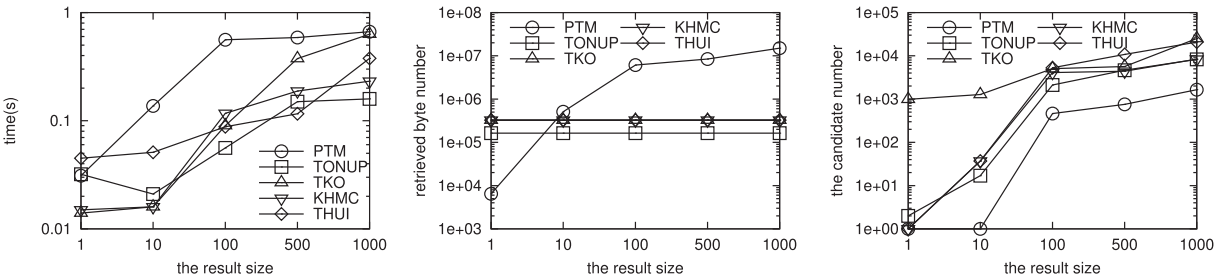


Fig. 22. The seventh group of experiment 6 (foodmart data: $n = 4141, d = 1559, w = 4.5$).

the eighth group of experiment 6. On mushroom data set, the top-k HUIM can find the top-k HUIs of the longer length. This not only increases the overall sizes of the prefix-based partitions, but also increases the number of the promising items during the execution of PTM. It is noted that THUI shows the best performance on dense data sets. The contiguous set of items can be acquired more likely on dense data sets.

7.8.3. Discussion

The experiment 6 compares the performance of PTM with the existing algorithms on the small and medium-sized data sets, including eight groups on synthetic and real-life data sets of different characteristics. In the most of the groups, PTM has

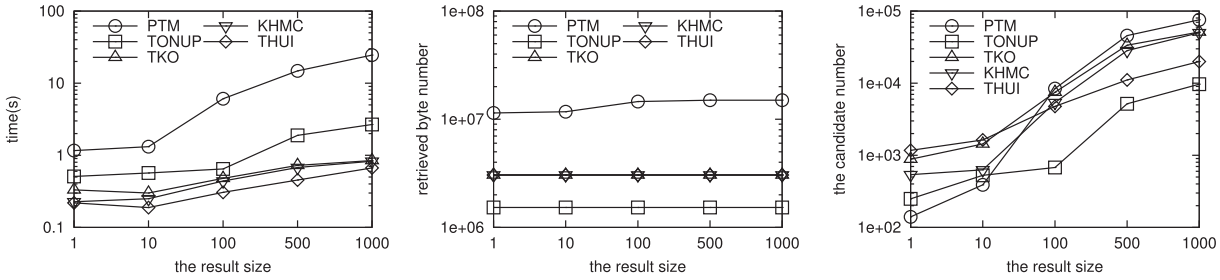


Fig. 23. The eighth group of experiment 6 (mushroom data: $n = 8124, d = 119, w = 23$).

a good enough performance, and it usually maintains the least number of the candidates compared to the existing algorithms. However, it still needs to be noted that, PTM is inferior to the existing algorithms on highly dense data sets (such as mushroom). In this case, one possible choice is that PTM selects the best existing algorithm to deal with the relevant prefix-based partitions. In our future work, we will consider how to improve PTM by extending the idea of prefix-based partitioning to the highly dense data sets. Nevertheless, just as analyzed in Section 3.2, the existing algorithms cannot deal with massive data directly. For PTM, it not only mines the top-k HUIs on massive data quickly, but also has a good enough performance on the small and mediums-sized data sets.

8. Conclusion

This paper considers the problem of top-k HUIM on massive data. Top-k HUIM is a hot research topic recently and many algorithms are proposed. However, it is analyzed that the existing algorithms only consider the data set of small and medium size, and their execution cost on massive data is prohibitively high. This paper first devises a baseline algorithm with the apriori-like level-wise execution mode to handle massive data. In order to solve the issues of the baseline algorithm, i.e. the multiple database scan and a large number of candidates, this paper presents a prefix-partitioning-based PTM algorithm to mine top-k HUIs on massive data efficiently. The transaction database is transformed into a set of prefix-based partitions, each of which shares the same prefix item. The prefix-based partitioning limits the range of the transactions for computing the utilities of the itemsets, and can speed up the mining operation significantly. PTM utilizes the selection order of average transaction utility to process the partitions, which can raise the border utility threshold quickly. By the assistant structure, most of the partitions, which cannot generate any top-k HUIs, can be skipped directly. For the partitions not skipped, this paper devises the efficient method to process them, which performs the depth-first search on the set enumeration tree of the promising items. The full-suffix-utility-based subtree pruning rule is exploited to reduce the exploration space. The extensive experiments are conducted on synthetic and real data sets. The experimental results show that PTM can discover top-k HUIs on massive data efficiently.

CRediT authorship contribution statement

Xixian Han: Conceptualization, Methodology, Software, Investigation, Writing - original draft. **Xianmin Liu:** Software, Investigation, Visualization. **Jianzhong Li:** Writing - review & editing. **Hong Gao:** Writing - review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by the NSFC under Grant Nos. 61872106, 61832003, 61632010, 61502121, National key research and development program of China under Grant No. 2016YFB1000703.

References

[1] Charu C. Aggarwal, Jiawei Han (Eds.), Frequent Pattern Mining, Springer, 2014.
[2] Rakesh Agrawal, Ramakrishnan Srikant, Fast algorithms for mining association rules in large databases, in: VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, 1994, pp. 487–499.
[3] Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer, Byeong-Soo Jeong, Young-Koo Lee. Efficient tree structures for high utility pattern mining in incremental databases. IEEE Trans. Knowl. Data Eng., 21(12):1708–1721, 2009.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028

[4] Yin-Ling Cheung and Ada Wai-Chee Fu, Ada Wai-Chee Fu, Mining frequent itemsets without support threshold: With and without item constraints, IEEE Trans. Knowl. Data Eng. 16 (9) (2004) 1052–1069.
[5] Youcef Djenouri, Djamel Djenouri, Asma Belhadi, Alberto Cano, Exploiting GPU and cluster parallelism in single scan frequent itemset mining, Inf. Sci. 496 (2019) 363–377.
[6] Quang-Huy Duong, Philippe Fournier-Viger, Heri Ramampiaro, Kjetil Nørvåg, Thu-Lan Dam, Efficient high utility itemset mining using buffered utility-lists, Appl. Intell. 48 (7) (2018) 1859–1877.
[7] Quang-Huy Duong, Bo Liao, Philippe Fournier-Viger, Thu-Lan Dam, An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies, Knowl.-Based Syst. 104 (2016) 106–122.
[8] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Zhihong Deng, Hoang Thanh Lam. The SPMF open-source data mining library version 2, in: Proceedings of 27th European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, ECML PKDD 2016, Part III, pages 36–40, 2016.
[9] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Roger Nkambou, Bay Vo, and Vincent S. Tseng, editors. High-Utility Pattern Mining: Theory, Algorithms and Applications. Springer, 2019.
[10] Philippe Fournier-Viger, Wu. Cheng-Wei, Souleymane Zida, Vincent S. Tseng, FHM: faster high-utility itemset mining using estimated utility co-occurrence pruning, in: Proceedings of 21st International Symposium Foundations of Intelligent Systems, ISMIS 2014, 2014, pp. 83–92.
[11] Fu. Ada Wai-Chee, Renfrew W.-W. Kwong, Jian Tang, Mining N-most interesting itemsets, in: Proceedings of the 12th International Symposium Foundations of Intelligent Systems, ISMIS 2000, 2000, pp. 59–67.
[12] Xixian Han, Jianzhong Li, Hong Gao, Efficient top-k retrieval on massive data, IEEE Trans. Knowl. Data Eng. 27 (10) (2015) 2687–2699.
[13] Xixian Han, Xianmin Liu, Jian Chen, Guojun Lai, Hong Gao, Jianzhong Li, Efficiently mining frequent itemsets on massive data, IEEE Access 7 (2019) 31409–31421.
[14] Srikumar Krishnamoorthy, Pruning strategies for mining high utility itemsets, Expert Syst. Appl. 42 (5) (2015) 2371–2381.
[15] Srikumar Krishnamoorthy, Mining top-k high utility itemsets with effective threshold raising strategies, Expert Syst. Appl. 117 (2019) 148–165.
[16] Yu-Chiang Li, Jieh-Shan Yeh, Chin-Chen Chang, Isolated items discarding strategy for discovering high utility itemsets, Data Knowl. Eng. 64 (1) (2008) 198–217.
[17] Chun-Han Lin, Cheng-Wei Wu, JianTao Huang, Vincent S. Tseng. Parallel mining of top-k high utility itemsets in spark in-memory computing architecture. In Proceedings of 23rd Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, PAKDD 2019, Part II, pages 253–265, 2019.
[18] Junqiang Liu, Ke Wang, Benjamin C.M. Fung, Mining high utility patterns in one phase without generating candidates, IEEE Trans. Knowl. Data Eng. 28 (5) (2016) 1245–1257.
[19] Junqiang Liu, Xingxing Zhang, Benjamin C.M. Fung, Jiuyong Li, Farkhund Iqbal, Opportunistic mining of top-n high utility patterns, Inf. Sci. 441 (2018) 171–186.
[20] Mengchi Liu, Jun-Feng Qu. Mining high utility itemsets without candidate generation. In Proceedings of 21st ACM International Conference on Information and Knowledge Management, CIKM'12, pages 55–64, 2012.
[21] Ying Liu, Wei-keng Liao, Alok N. Choudhary. A two-phase algorithm for fast discovery of high utility itemsets. In Proceedings of 9th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, PAKDD 2005, pages 689–695, 2005.
[22] José María Luna, Philippe Fournier-Viger, Sebastián Ventura. Frequent itemset mining: A 25 years review. Wiley Interdiscip. Rev. Data Min. Knowl. Discov., 9(6), 2019.
[23] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, David W. Cheung, Efficient top-k aggregation of ranked inputs, ACM Trans. Database Syst. 32(3):19 (2007).
[24] Alex Yuxuan Peng, Yun Sing Koh, Patricia Riddle. mhuiminer: A fast high utility itemset mining algorithm for sparse datasets, in: Proceedings of 21st Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, PAKDD 2017, Part II, pages 196–207, 2017.
[25] Heungmo Ryang, Unil Yun, Top-k high utility pattern mining with effective threshold raising strategies, Knowl.-Based Syst. 76 (2015) 109–126.
[26] Vincent S. Tseng, Bai-En Shie, Wu. Cheng-Wei, S.Yu. Philip, Efficient algorithms for mining high utility itemsets from transactional databases, IEEE Trans. Knowl. Data Eng. 25 (8) (2013) 1772–1786.
[27] Vincent S. Tseng, Wu. Cheng-Wei, Philippe Fournier-Viger, S.Yu. Philip, Efficient algorithms for mining top-k high utility itemsets, IEEE Trans. Knowl. Data Eng. 28 (1) (2016) 54–67.
[28] Vincent S. Tseng, Cheng-Wei Wu, Bai-En Shie, Philip S. Yu. Up-growth: an efficient algorithm for high utility itemset mining, in: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '10, pages 253–262, 2010.
[29] Jason Tsong-Li Wang, Mohammed Javeed Zaki, Hannu Toivonen, Dennis E. Shasha, editors. Data Mining in Bioinformatics. Springer, 2005.
[30] Cheng-Wei Wu, Bai-En Shie, Vincent S. Tseng, Philip S. Yu. Mining top-k high utility itemsets, in: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, pages 78–86, 2012.
[31] Ching-Huang Yun, Ming-Syan Chen, Mining mobile sequential patterns in a mobile commerce environment, IEEE Trans. Systems, Man, and Cybernetics, Part C 37 (2) (2007) 278–295.
[32] Mohammed Javeed Zaki, Scalable algorithms for association mining, IEEE Trans. Knowl. Data Eng. 12 (3) (2000) 372–390.
[33] Lin Zhou, Ying Liu, Jing Wang, Yong Shi. Utility-based web path traversal pattern mining. In Proceedings of the 7th IEEE International Conference on Data Mining (ICDM 2007), pages 373–380, 2007.
[34] Souleymane Zida, Philippe Fournier-Viger, Jerry Chun-Wei Lin, Cheng-Wei Wu, Vincent S. Tseng. EFIM: a fast and memory efficient algorithm for high-utility itemset mining. Knowl. Inf. Syst., 51(2):595–625, 2017.

Please cite this article as: X. Han, X. Liu, J. Li et al., Efficient top-k high utility itemset mining on massive data, Information Sciences, https://doi.org/10.1016/j.ins.2020.08.028