Appl Intell (2017) 47:1240–1255
DOI 10.1007/s10489-017-0939-7

Mining top-k high-utility itemsets from a data stream under sliding window model

Siddharth Dawar[1] · **Veronica Sharma**[1] · **Vikram Goyal**[1]

Published online: 8 June 2017

Abstract High-utility itemset mining has gained significant attention in the past few years. It aims to find sets of items i.e. itemsets from a database with utility no less than a user defined threshold. The notion of utility provides more flexibility to an analyst to mine relevant itemsets. Nowadays, a continuous and unbounded stream of data is generated from web-clicks, transaction flow from retail stores, sensor networks, etc. Mining high-utility itemsets from a data stream is a challenging task as the incoming stream of data has to be processed on the fly with time and storage memory constraints. The number of high-utility itemsets depends on the user-defined threshold. A large number of itemsets can be generated at very low threshold values and vice versa. It can be a tedious task to set a threshold value to get a reasonable number of itemsets. Top-k high-utility itemset mining was coined to address this issue. k is the number of high-utility itemsets in the result set as defined by the user. In this paper, we propose a data structure and an efficient algorithm for mining top-k high-utility itemsets from a data stream. The algorithm has a single phase that does not generate any candidates, unlike many algorithms that work in two phases, i.e., candidate generation followed by candidates verification. We conduct extensive experiments on several real and synthetic datasets. Experimental results demonstrate that our proposed algorithm performs 20 to 80 times better on sparse datasets and 300 to 700 times on dense datasets than the state-of-the-art algorithm in terms of computation time. Furthermore, our proposed algorithm requires less memory compared to the state-of-the-art algorithm.

Keywords Data mining · Pattern mining · Utility mining · Data streams · Top-k high utility mining

1 Introduction

High-utility itemset mining [5, 9, 12, 19] aims to find itemsets from a transaction database with utility no less than a user-defined threshold. A transaction database consists of a set of retail store transactions, where each transaction contains a set of items. Each item in a transaction has an internal utility associated with it. For example in a retail store application scenario, the internal utility can be the number of items purchased by a customer. The relative importance of items present in the database is defined by an external utility. For the application of retail stores, it can be profit earned by the store on an item. The utility for an item in a transaction is defined as the product of its internal and external utility. The utility of an itemset (a set of items) can be defined as the sum of the utility of each item present in the itemset. High-utility itemset mining has been extensively applied in the retail domain for market-basket analysis [10, 27]. In the retail store domain, the High-utility itemset mining would discover the set of items which generate high profits for a retail store. Such information can be used for better business decision making like increasing the inventory of high-utility products or laying out a plan for items in the store.

✉ Vikram Goyal
vikram@iiitd.ac.in

Siddharth Dawar
siddharthd@iiitd.ac.in

Veronica Sharma
veronica1428@iiitd.ac.in

[1] Department of Computer Science, Indraprastha Institute of Information Technology, Delhi, India

The notion of utility can be defined according to the application domain. For example, consider a document collection as a transaction database, where every document is modeled as a transaction such that words within a document are items. The internal utility associated with every word is the frequency of the word in a document, known as term frequency(TF). The external utility of a word is its inverse document frequency(IDF) defined as the reciprocal of the number of documents that contain a particular word. High-utility itemset mining can be used to find a meaningful set of words with a high TF-IDF [24] score. Such set of words can be used for query recommendation by the search engine to its user. Utility mining has been studied extensively in contexts like sequence database [34, 35], episodes [22, 31], data streams [4], etc.

Data streams model the scenario where we have a continuous, unbounded stream of data. Data streams are generated from various sources like click-stream data from the web, sensor networks, telecommunication networks, RFID tags [1], etc. The challenge in mining patterns from data streams is that we can not store and process the complete stream due to time and memory constraints. Retail giants like Walmart generate a continuous stream of user purchased products in real time. High-utility itemset mining can be applied to mine interesting patterns from a data stream to get relevant information quickly for taking suitable actions. For example, high-utility products can be recommended to new customers. The store can also use such patterns to improve the sales by applying different marketing strategies by either giving discounts or through shelf-space management [2]. High-utility itemset mining can also be used for real-time load balancing in wireless networks. Consider the scenario in a corporate environment, where multiple WiFi access points are deployed. These access points are connected to a switch through a communication medium like a microwave or a LAN cable. Each access point serves a set of users during a certain period. Consider the following model of a transaction database, where the access points are defined as items, and internal utility associated with an access point represents the load in terms of the number of people connected to it. Each transaction is modeled as the usage of access points for a single day. The external utility associated with each access point is the cost of transferring data to the server. The utility of an access point can be defined as the product of its load and cost. The set of highly loaded access points can be identified in real time. Such information can be used to reduce the load by redistributing it or improving connectivity with the server.

A couple of algorithms [4, 37] have been proposed in the literature for mining high-utility itemsets from a data stream. An inherent problem with pattern mining algorithms is the generation of a large number of patterns. The number of patterns depends on the value of defined threshold. Low threshold results in the generation of a large number of patterns and a high threshold generate very few patterns. It is difficult for a user to define an appropriate threshold. In order to relieve the user from this tedious task, the notion of *top-k* high-utility itemsets was coined. Zihayat et al. [37] proposed a tree data structure called *HUDS-tree* and an algorithm called *T-HUDS* for mining top-k high-utility itemsets from a data stream under the sliding window model. In the sliding window model, a window is composed of a fixed number of batches. Each batch contains a fixed number of transactions. When a new batch arrives, the oldest batch is removed, and the window slides to include the new batch. The aim is to mine top-k high-utility itemsets from each sliding window. HUDS-tree is constructed by inserting transactions into the tree, and each node maintains batch wise overestimated utility values. T-HUDS algorithm is a two-phase algorithm. It generates candidate high-utility itemsets in the first phase and verifies them to find exact top-k high-utility itemsets in the next phase. However, the time spent in verifying the patterns depends on the number of candidates generated and reduces the efficiency of the algorithm. In this paper, we propose a data structure and an algorithm which mines top-k high-utility itemsets without generating any candidates. Our proposed algorithm mines the complete set of top-k high-utility itemsets from every sliding window without missing any itemsets. We also give a proof of correctness for our algorithm.

Our contributions can be summarized as follows:

1. We design a data structure called *iList* to mine high-utility itemsets from a data stream. The data structure is efficient in terms of insertion and deletion of batches.
2. We propose an efficient algorithm that uses *iList* to mine top-k high-utility itemsets assuming the sliding window model.
3. We conduct extensive experiments on real and synthetic datasets to demonstrate the effectiveness of our proposed algorithm compared to the state-of-the-art algorithms.

Our paper is organized as follows. The relevant literature and necessary background information are discussed in Section 2. Our proposed data structure and the algorithm are presented in Section 3 and experimental results are presented in Section 4. Finally, Section 5 concludes the paper.

2 Background

2.1 Preliminaries and problem statement

The problem of mining high-utility itemsets is defined as follows. We are given a set of m items $I = \{i_1, i_2, ..., i_m\}$,

where each item i_p has a positive quantity q_i and profit $pr(i_p)$ associated with it. A set of k distinct items put together is called an itemset $I = \{i_1, i_2, ..., i_k\}$ of length k. A transaction database $D = \{T_1, T_2, ..., T_n\}$ is a set of n transactions, where every transaction is an itemset.

Definition 1 (Utility of an item in a transaction) The utility of an item i in a transaction T is denoted as $u(i, T)$ and defined as the product of its quantity in T and profit i.e., $u(i, T) = q_i * pr(i)$.

Definition 2 (Utility of an itemset in a transaction) The utility of an itemset X contained in a transaction T is denoted as $u(X, T)$ and defined by $\sum_{i \in X} u(i, T)$.

For example, consider the example transaction database (Table 1) and the profit table (Table 2). The utility of $\{A\}$ in T_3 is $u(\{A\}, T_3) = 1 \times 5 = 5$ and $u(\{A, B\}, T_3) = u(A, T_3) + u(B, T_3) = 5 + 4 = 9$. If an itemset is not contained in a transaction then its utility for that transaction is taken as zero.

Definition 3 (Utility of an itemset in database) The utility of an itemset X in database D is denoted as $u(X)$ and defined as $\sum_{T \in D} u(X, T)$.

For example, the utility of itemset $u(\{A, B\}) = u(\{A, B\}, T_3) + u(\{A, B\}, T_4) + u(\{A, B\}), T_5) = (1 \times 5 + 2 \times 2) + (1 \times 5 + 4 \times 2) + (1 \times 5 + 2 \times 2) = 9 + 13 + 9 = 31$.

Definition 4 (High-utility itemset) An itemset is called a high-utility itemset if its utility is no less than a given minimum user-defined threshold denoted by min_util.

Consider min_util to be 30. Itemset $\{AB\}$ is a high-utility itemset as its utility is 31. Frequent itemset mining satisfies the anti-monotonicity property i.e. the supersets of an infrequent itemset are infrequent. The algorithms designed for frequent itemset mining use this property to prune the search space while enumerating frequent itemsets.

Table 2 Profit table of items

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

However, high-utility itemset mining does not satisfy this property. The superset of a low utility itemset can have utility greater than the minimum user-defined threshold. Researchers came up with the concept of transaction weighted utility (TWU), which satisfies the anti-monotonic property and helps to prune the search space.

Definition 5 (Transaction utility) The transaction utility of a transaction T is denoted by $TU(T)$ and defined as $\sum_{\forall i \in T} u(i, T)$.

For example, the transaction utility of every transaction in our example database is shown in Table 1.

Definition 6 (TWU of an itemset) TWU of an itemset X is the sum of the transaction utilities of all the transactions containing X, which is denoted as $TWU(X)$ and defined as $\sum_{\substack{X \subseteq T, \\ T \in D}} TU(T)$.

For example, $TWU(\{A\}) = TU(T_1) + TU(T_2) + TU(T_3) + TU(T_4) + TU(T_5) = 129$.

Definition 7 (High TWU itemset) An itemset X is called a high transaction weighted utility itemset, if $TWU(X)$ is no less than min_util.

If $min\_util = 60$, $\{A\}$ is a high TWU itemset. However, if $min\_util = 130$, $\{A\}$ and none of its supersets can be a high-utility itemset.

Our goal is to mine top-k high-utility itemsets over data streams, where k is defined by a user. Data stream models the scenario where a stream of transactions arrives continuously in real time and processed in batches. A **batch**

Table 1 Transaction database

			Tid	Transaction	Transaction utility
SW2	SW1	B1	T_1	$(A:1)\,(C:1)\,(D:1)$	8
			T_2	$(A:2)\,(C:6)\,(E:2)\,(G:5)$	27
		B2	T_3	$(A:1)\,(B:2)\,(C:1)\,(D:6)\,(E:1)\,(F:5)$	50
			T_4	$(A:1)(B:4)\,(C:3)\,(D:3)\,(E:2)$	28
		B3	T_5	$(A:1)\,(B:2)\,(C:2)\,(E:1)\,(G:2)$	16
			T_6	$(B:2)\,(C:2)\,(E:1)\,(G:2)$	11

B_i contains a fixed number of transactions i.e. $B_i = \{T_1, T_2, ..., T_r\}$. A **sliding window** consists of m recent batches, where m is called the size of window, denoted as $winSize$. A sliding window can be represented as $SW_i = \{B_i, B_{i+1}, ..., B_m\}$. Consider our example database as a data stream, where each batch is composed of two transactions and a sliding window contains two batches. The first two batches form the first window i.e. $SW_1 = \{B_1, B_2\}$. Upon arrival of batch B_3, the window slides to remove batch B_1 and include B_3. The sliding window and batches are shown in our example database (Table 1).

Problem statement For each sliding window SW_i in a data stream, the problem is to find the top-k high-utility itemsets in SW_i, ranked in descending order of their utility, where k is a user-defined parameter.

2.2 Related work

In this subsection, we discuss the relevant literature on itemset mining. A table summarizing the related work is shown in Table 3. Frequent itemset mining [7, 8, 13, 15, 20, 33] has been studied extensively in past two decades. Given a transaction database D, where each transaction represents the items purchased together by a customer, frequent itemset mining aims to find the set of items with a frequency no less than a minimum user-defined threshold. The naive approach is to enumerate the complete search space to find the frequent itemsets. The number of frequent itemsets in a transaction database with m distinct items can be 2^m-1. The search space needs to be explored smartly by using algorithmic techniques. Agrawal et al. [2] proposed an algorithm called *Apriori* which explores the search space in a breadth-first or level-wise manner. Initially, the algorithm scans the database to find the set of frequent items. The algorithm generates candidate itemsets of length k by joining two frequent $k-1$ length itemsets. A database scan is performed to find the frequent itemsets at any level. If length of the longest frequent itemset is k, *Apriori* will scan the database k times. The problem with *Apriori* is the generation of a large number of candidates and multiple scans of the database. Han et al. [13] proposed a new data structure called *FP-tree* and a recursive depth-first search algorithm *FP-Growth* which mines frequent itemsets without generating any candidates within two database scans only. The transactions in the database are inserted to form a *FP-tree*. The root node of the tree is a special node which represents the null item. Every non-root node in an FP-tree stores the following information: name, frequency, link to the child nodes and a link to the parent node. The name field of a node n is an identifier of the item and frequency stores the number of transactions which contains item n. A path from the root to a node n in the tree represents the transactions which contain the itemset represented by nodes along that path. A header table is also constructed for faster traversal of the *FP-tree*. A header table consists of two fields: itemname, a pointer to linked-list which contains pointers to all

Table 3 An overview of itemset mining algorithms

Algorithm	Author	Year	Data structure	Candidate generation	Mining	Database
Apriori	Agrawal et al.	1994	None	Yes	Frequent itemsets	Transactions
Eclat	Zaki et al.	1997	Tid-list	No	Frequent itemsets	Transactions
FP-Growth	Han et al.	2000	FP-tree	No	Frequent itemsets	Transactions
Frequent	Chang et al.	2003	Summary	No	Frequent itemsets	Data stream
Moment	Chi et al.	2004	Closed enumeration tree	No	Frequent closed itemsets	Data stream
MFI-TransSW	Li et al.	2009	Bit sequence	No	Frequent itemsets	Data stream
TOPSIL-Miner	Yang et al.	2010	TOPSIL-tree	No	Top-k frequent itemsets	Data stream
Apriori closed	Yang et al.	2003	None	Yes	Top-k high-utility closed itemsets	Transactions
Mining	Ahmed et al.	2009	IHUP-tree	Yes	High-utility itemsets	Transactions
UP-Growth	Tseng et al.	2010	UP-tree	Yes	High-utility itemsets	Transactions
HUI-Miner	Liu et al.	2012	Utility-list	No	High-utility itemsets	Transactions
TKU, TKO	Tseng et al.	2016	UP-tree, Utility-list	No	Top-k high-utility itemsets	Transactions
THUI-Mine	Tseng et al.	2006	None	Yes	High-utility itemsets	Data stream
MHUI-Bit, MHUI-Tid	Li et al.	2008	LexTree	Yes	High-utility itemsets	Data stream
GUIDE	Shi et al.	2012	MUI-tree	Yes	Maximal high-utility itemsets	Data stream
HUPMS	Ahmed et al.	2012	HUS-tree	Yes	High-utility itemsets	Data stream
T-HUDS	Zihayat et al.	2014	HUDS-tree	Yes	Top-k high-utility itemsets	Data stream
SHU-Grow	Ryang et al.	2016	SHU-tree	Yes	High-utility itemsets	Data stream

the nodes with the same name as that of item-name. The algorithm starts with an empty prefix, and the header table is processed in a bottom-up manner. The linked-list associated with an item i is scanned to identify the paths which contain item i in FP-tree. Such paths are identified and inserted to form a local *FP-tree* for the item i. The set of all frequent itemsets with item i as their prefix are mined from the local FP-tree. After the *FP-tree* of item i is completely processed, the algorithm proceeds with the next item from the header table. The recursive generation of local FP-trees is a costly operation when the database contains long transactions. Zaki et al. [36] proposed an inverted-list structure called tid-list and an algorithm *Eclat* which generates frequent itemsets by intersecting the tid-lists associated with it. A tid-list associated with an itemset contains the transaction identifiers which contains that itemset. The cardinality of the tid-list is the frequency of an itemset in the database. The advantage of *Eclat* algorithm is the use of simple intersection operations for computing the support of an itemset. Several algorithms [7, 8, 15, 16, 32, 33] have been proposed to extract frequent itemsets from a data stream assuming different models like landmark, sliding window, time fading, etc. Landmark model assumes a fixed point in time called landmark and mines patterns from it to the current time point. Time fading is similar to landmark model but gives more importance to recent data compared to historical using a decay factor. Sliding window model stores a current window of the most recent data and mines itemsets from it. The window is updated on the arrival of new data. Frequent itemset mining assumes the binary presence or absence of items in a transaction. In real life, it has been observed the items can be purchased in multiple quantities and have different importance. The frequent itemset mining algorithms can not be applied directly in scenarios where both internal and external utility of items are considered.

The problem of high-utility itemset mining [6] was coined to address such scenarios. Liu et al. [19] proposed a two-phase algorithm for mining high-utility itemsets which finds potential high-utility itemsets in the first phase and verifies them in the next phase by scanning the database again. Ahmed et al. [3] were the first to propose a tree-based two-phase recursive algorithm for mining high-utility itemsets. The authors defined a data structure similar to FP-tree called *IHUP-tree*, where each node stores the item id, support, and *TWU* values. Their proposed algorithm generates candidate high-utility itemsets in the first phase using TWU information stored at each node in the tree. Another database scan is performed to find the exact high-utility itemsets. A problem with algorithms based on *IHUP*-tree is the generation of a large number of candidates in the first phase. In order to reduce the number of candidates, Tseng et al. [27] proposed another FP-tree like data structure called UP-tree, which stores node utility at each node of the tree. Node utility is a better upper bound estimate compared to *TWU*. Liu and Qu [18] proposed an inverted-list data structure called utility-list and an algorithm called *HUI-Miner* to mine high-utility itemsets in a single phase only without generating any candidates. The utility-list associated with an itemset X stores the following information: Tid, exact utility, remaining utility. Tid is the identifier of the transaction which contains X. The exact utility field stores the utility of X in the transaction with identifier Tid. The remaining utility is a heuristic information used to prune the supersets of itemset X. However, a typical problem with inverted-list based algorithms is the cost associated with intersecting lists to generate the list of larger itemsets. Some strategies [10, 14] were proposed to address this issue. The number of interesting patterns generated by mining algorithms depends on the user-defined threshold. It is difficult to select a threshold for a pattern mining algorithm to return a reasonable number of patterns which can be analyzed for business decision making. A couple of algorithms [29, 30] have also been proposed to mine top-k high-utility itemsets from a static transaction database.

Some algorithms have been proposed to mine high-utility itemsets from a data stream. Tseng et al. [26] proposed the first algorithm for mining high-utility temporal itemsets from a data stream. The authors proposed an algorithm called *THUI-Mine* based on the two-phase algorithm [19] for mining high-utility itemsets assuming the sliding window model. Li et al. [17] proposed two representations for an item (bit-vectors, tid-lists) and a summary tree data structure to mine high-utility itemsets from a transaction sensitive sliding window. Their proposed tree structure stores high TWU itemsets of length one and two. The candidates of larger length are generated in a level-wise apriori fashion. Shie et al. [25] proposed a novel framework called *GUIDE* for mining maximal high-utility itemsets assuming the landmark, sliding window and time fading models. Ahmed et al. [4] were the first to propose a tree data structure called *HUS-tree* and an algorithm for mining high-utility itemsets from a data stream under the sliding window model. The proposed tree structure was similar to UP-tree [28] and each node maintained batch-wise *TWU* and support count. Zihayat and An [37] proposed another data structure called *HUDS-tree* and an algorithm for mining top-k high-utility itemsets from data stream under the sliding window model. Instead of storing *TWU* with each node, *HUDS-tree* uses a closer upper bound estimate called prefix utility and prunes the search space better than *HUS-tree*. The authors also proposed several strategies to increase the minimum threshold from zero before starting the mining process to decrease the number of candidates. Recently, Ryang and Yun [23] proposed techniques to reduce the number of candidates generated by *HUS-tree* and another two-phase algorithm for mining high-utility patterns

Table 4 TWU of items in SW_1

Item	A	B	C	D	E	F	G
TWU	113	78	113	86	105	50	27

from a data stream assuming the sliding window model. In the present study, we propose an inverted-list data structure and an algorithm which mines top-k high-utility itemsets from a data stream without generating any candidates.

3 Vert_top-k DS: mining top-k high-utility itemsets over data streams

We now present our inverted list data structure called *iList* which captures the utility information associated with an itemset across windows. We introduce some terminologies [18], before describing our data structure.

Definition 8 (Reorganized transaction) A transaction T' is called as a reorganized transaction, if the items present in T' are sorted in ascending order of their TWU value.

Definition 9 (Set of items after itemset X in T') Given an itemset X and a transaction T' with $X \subseteq T'$, the set of all items after X in T' is denoted as T'/X.

Definition 10 (Remaining utility of an itemset X in transaction T') The remaining utility of itemset X in T', denoted as $RU(X, T')$, is the sum of utilities of all the items in T'/X.

For example, the *TWU* of items in the first sliding window SW_1 is shown in Table 4. The ordering of items in ascending order of *TWU* is $(G < F < B < D < E < A < C)$. Here, G is the item with the least *TWU* and item C has the highest *TWU*. Consider the first transaction from our example database shown in Table 1. In T_1, item A appears with utility 5. Item C appears after A in the first transaction according to the ordering defined above. Therefore, the remaining utility(RU) of item A in the first transaction is equal to the utility of C in T_1 i.e. 1.

Table 5 iList of item B in SW_1

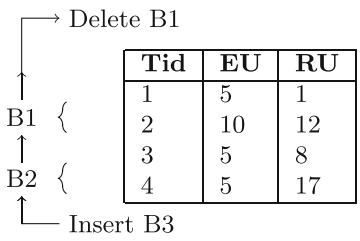


Table 6 iList of item B in SW_1

	Tid	EU	RU
B2	3	4	4
	4	8	9

3.1 iList data structure

iList maintains a FIFO queue of batches. Each batch contains a list of $\langle Tid, EU, RU \rangle$ tuples. *Tid* stores the transaction identifier which contains an itemset. *EU* is the exact utility of an itemset. *RU* is the remaining utility of an itemset. Whenever a new batch of transactions arrives, it is added to the end of the queue. Once the size of the queue exceeds the window size, the first batch is removed from the front, and the new batch is inserted from the rear. The advantage of *iList* data structure is that it performs insertion and deletion of batches quickly. For example, the *iList* for item {A} and {B} for the first sliding window SW_1 is shown in Tables 5 and 6 respectively. Note that our *iList* data structure is an adaptation of utility-list [18] proposed for static transaction database scenarios.

The *iList* data structure is constructed by scanning the sliding window twice. In the first scan, *TWU* of items is computed. During the second scan, items in each transaction are sorted according to ascending order of *TWU* and *iList* for each item is created. The *iList* for an itemset composed of two items is created by intersecting the *iList* of individual items. The first step is to find the batches which contain both items. As an example, we want to construct the *iList* of itemset $\{AB\}$ from *iLists* A and B respectively. The *iLists* of A and B have batch B_2 in common. Once the list of common batches are identified, transactions which contains both items are identified. For example, items A and B appear together in transaction T_3 and T_4 respectively. The *iList* for itemset $\{AB\}$ is shown in Table 7. The process for constructing the *iList* of k-itemset is similar to the above process.

3.2 Proposed algorithm

In this subsection, we propose an algorithm for mining top-k high-utility itemsets as presented in Algorithm 1. Our algorithm called Vert_top_k_DS takes as input: batch of

Table 7 iList of itemset AB in SW_1

	Tid	EU	RU
B2	3	9	4
	4	13	9

transactions, k, $winSize$, $iLists$, and returns the set of top-k high-utility itemsets.

Algorithm 1 Vert_top_k_DS algorithm

Input: Batch B_i, k, $winSize$, $iLists$. ▷ iLists the list of iList of distinct items present in B_i
Output: Complete set of top-k high-utility itemsets.

1: **if** $iLists$ is empty **then** ▷ Processing first batch of the first sliding window
2: $top\_k\_thres = 0$ ▷ top_k_thres will store utility of top-k high-utility itemset
3: Scan the transactions in current batch B_i to compute the *TWU* of items.
4: **else**
5: Update the *TWU* of items.
6: Update $iLists$ i.e. reuse $iLists$ from previous window.
7: **end if**
8: Add batch B_i to current sliding window SW_i.
9: **if** number of batches $\geq winSize$ **then**
10: top_k_hui = Compute_top_k_window(k, $winSize$, top_k_thres, $iLists$, SW_i)
11: **end if**
12: $top\_k\_thres = \min\ \{ u_{SW_i - B_i}(itemset) | itemset \in top\_k\_hui \}$
13: return top_k_hui
14:
15: **function** COMPUTE_TOP_K_WINDOW(k, $winSize$, top_k_thres, $iLists$, SW_i)
16: **if** $iLists$ is empty **then**
17: Create $iLists$ of items.
18: **end if**
19: **for** each entry i in $iLists$ **do**
20: **if** $u(i) \geq top\_k\_thres$ **then** ▷ Pre-insertion of distinct items in top_k_hui
21: Insert item $(i, u(i))$ into top_k_hui
22: **end if**
23: **end for**
24: top_k_thres = max{top_k_thres, $utility\_k^{th}\_itemset$}
25: Call Vert_Miner({}, $iLists$, top_k_thres) ▷ Start the mining process
26: **end function**
27:
28: **function** VERT_MINER($I.iList$, Ext_I, top_k_thres)
29: **for** each iList Ix in Ext_I **do**
30: **if** (**then**$Ix.sumEU \geq top\_k\_thres$ and Ix is not an itemset of length one))
31: Update top_k_hui with itemset $(Ix, u(Ix))$ ▷ Add itemset Ix to top-k buffer
32: **end if**
33: **if** (**then** $Ix.sumEU + Ix.sumRU \geq top\_k\_thres$)) ▷ Estimate the utility of supersets of Ix
34: Ext_Ix=∅
35: **for** each itemset Iy in Ext_I such that y comes after x **do**
36: Ixy=$Ix \cup Iy$
37: $Ixy.iList$=Construct(I, Ix, Iy)
38: Ext_Ix=Extensions of $Ix \cup Ixy$
39: **end for**
40: Vert_Miner (Ix, Ext_Ix, top_k_thres)
41: **end if**
42: **end for**
43: **end function**

Upon receiving a batch of transactions, the algorithm checks if the *iList* of items is empty (i.e. we are processing the first batch). If empty, the algorithm scans the transactions to compute the *TWU* of the distinct items present in

the current batch. Else, *TWU* of items is updated for the current batch. Our algorithm maintains the batch-wise *TWU* of items so that *TWU* of items can be updated easily upon arrival of a new batch and removal of the oldest batch. If *iLists* is not empty, *iLists* of items from the previous window is updated. The information stored for the previous batch is removed and ⟨*Tid*, *EU*, *RU*⟩ tuples for the new batch will be added. Once the window becomes full i.e. number of batches in the window equals the window size, Compute_top_k_window() method is invoked. The variable top_k_thres stores the exact utility of the current top-k high-utility itemset discovered during execution of our algorithm. For the first window, top_k_thres is set to zero. The method Compute_top_k_window() mines top-k high-utility itemsets from the current sliding window SW_i.

Now, we will discuss the method Compute_top_ k_ window in detail. If the *iLists* is empty, the items in every transaction are sorted in ascending order of *TWU* values. The *iLists* of individual items present in SW_i is created. Once the *iLists* are created, the mining process can begin. However, for top-k mining algorithms, it is beneficial to raise the minimum threshold from zero before starting the mining process. The algorithm inserts the individual items along with their utility values in the top-k buffer, and the buffer is sorted in decreasing order of utility value (Line 21). The $min\_top\_k\_thres$ value is set accordingly as shown in Line 24. After raising the threshold, the method $Vert\_Miner$ is invoked.

Since the individual items are already inserted in the top-k buffer along with their utility values, they are not added again to avoid duplicate insertion (Line 30). $Vert\_Miner$ generates the *iList* of pairs by intersecting the lists of single items. After the *iList* of an itemset is generated, the algorithm tries to insert the itemset in the top-k buffer. An itemset is inserted only if its exact utility is no less the current top_k_thres. An itemset and its supersets are pruned, if the sum of exact utility and remaining utility is less than top_k_thres. Vert_Miner proceeds in a similar manner and returns the complete set of top-k high-utility itemsets in the current sliding window. $Vert\_Miner$ returns the complete set of top-k high-utility itemsets from the sliding window SW_i.

In the sliding window model, the last $\{winSize - 1\}$ batches remain common between two consecutive windows. In order to compute a better threshold for the next sliding window, we implemented the threshold raising strategy [37]. In this strategy, the utility of the top-k itemsets is computed in the common batches between two windows. The top_k_thres is set as shown in Line 12 of our algorithm. As discussed above, our algorithm uses the sum of EU and RU value associated with an itemset to prune its supersets. The naive approach is to scan the *iList* once to compute the sum. As an implementation trick, we store the sum of EU and RU values for each batch of transactions for every itemset X denoted by X.iList.B.sumEU and X.iList.B.sumRU respectively.

Next, we illustrate the working of our algorithm with an example. Let us consider the database shown in Table 1. Let k be 3 and winSize be 2. The first window consists of batches B_1 and B_2. The *TWU* of items in the first window is shown in Table 4. Items in each transaction are sorted according to ascending order of *TWU* values. The ordering of items is $(G < F < B < D < E < A < C)$ for the first sliding window. Upon getting the complete sliding window, our algorithm constructs the *iLists* of single items. Single items are pre-inserted in the top-k buffer. The top-k buffer after pre-insertion is shown below.

$$top\_k\_hui = \{\{F\} : 25,\ \{A\} : 25,\ \{D\} : 20\}$$

The top_k_thres value is set to 20. The top-k high-utility itemsets in the first sliding window are shown below.

$$top\_k\_hui = \{\{ABCDE\} : 53,\ \{ABCDEF\} : 50,\ \{ABDEF\} : 49\}$$

The utility of itemsets {ABCDE}, {ABCDEF} and {ABDEF} in batch B_2 is computed and top_k_thres is set for the next sliding window as shown below.

$$top\_k\_hui = \{\{ABCDE\} : 53,\ \{ABCDEF\} : 50,\ \{ABDEF\} : 49\}$$

Upon arrival of the next batch B_3, the next sliding window SW_2 is constructed. The *TWU* of items in sliding window SW_2 is shown in Table 8. The algorithm proceeds by mining top-k high-utility itemsets from the second window. The algorithm terminates when there are no incoming batches.

Claim 1 Given the iList of an itemset X and the current sliding window SW_i, if

$$\sum_{B \in SW_i} X.iList.B.sumEU + X.iList.B.sumRU < top\_k\_thres$$

X and its supersets can not be top-k high-utility itemsets for window SW_i.

Proof An itemset X and its supersets are pruned only if,

$X.sumEU + X.sumRU < top\_k\_thres.$

Table 8 TWU (SW_2) table

Item	A	B	C	D	E	F	G
TWU	94	105	105	78	105	50	27

Table 9 Characteristics of real and synthetic datasets

Dataset	#trans	#items	Avg. length	Batch size	Window size	Type
ChainStore	11,12,949	46,086	7.2	1,00,000	6	Sparse
Retail	88,162	16,470	10.3	10,000	3	Sparse
Accidents	3,40,183	468	33.8	50,000	3	Dense
Connect	67,557	129	43	10,000	3	Dense
Mushroom	8,416	119	23	1,000	3	Dense
Retail_3x	2,64,486	16,470	10.3	88,162	2	Sparse
Mushroom_3x	25,248	119	23	8,416	2	Dense

It is trivial to prove that,

$$X.sumEU + X.sumRU = \sum_{B \in SW_i} X.iList.B.sumEU + X.iList.B.sumRU$$

Let us consider a superset of X denoted as X'.

$$For\ \forall t \supseteq X',\ u(X', t) \leq u(X, t) + \sum_{i \in t/X} u(i, t).$$

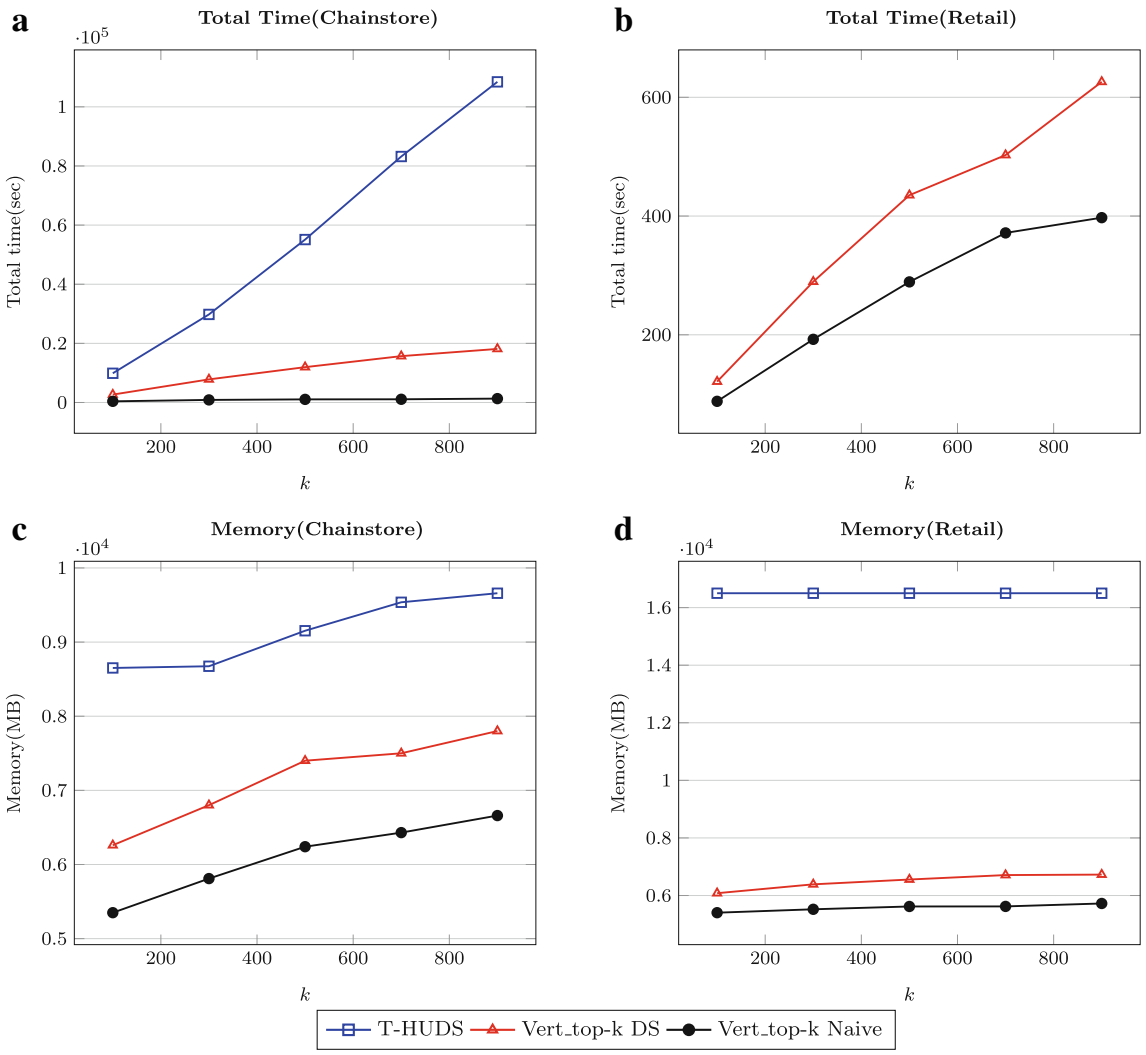


Fig. 1 Performance evaluation on sparse datasets. T-HUDS does not complete its execution for the Retail dataset and hence its plot is not reported

Table 10 Number of intermediate itemsets generated by Vert_top-k DS and Vert_top-k Naive for sparse datasets

Dataset	k	Vert_top-k DS	Vert_top-k Naive
Chainstore	100	5,49,57,268	1,79,07,454
	300	23,62,53,650	7,19,15,147
	500	39,77,92,596	10,33,67,820
	700	54,06,08,869	13,64,87,344
	900	63,58,89,688	16,00,35,955
Retail	100	2,40,37,137	1,94,92,763
	300	9,89,43,831	7,98,96,037
	500	15,65,68,444	13,24,61,788
	700	22,27,43,759	19,11,08,181
	900	27,78,74,288	22,88,71,899

$$Since\ X'.tids \subseteq X.tids, \therefore u(X') \leq \sum_{id(t) in X.tids} u(X,t) + RU(X,t) < top\_k\_thres.$$

$\therefore X$ and its supersets can not be top-k high-utility itemsets. □

Theorem 1 *Algorithm Vert_top_k_DS returns the complete set of top-k high utility itemsets.*

Proof We will prove that top_k_thres is never above the utility of the k^{th} itemset in a sliding window SW_i. We update top_k_thres at two places; during pre-insertion and execution of Vert_Miner. Since we are computing exact utility of itemsets and insertion operation is performed on when the utility of itemset X is no less than the utility of k^{th} itemset present in the buffer, it is guaranteed that there will be no false positives.

According to Claim 1, for an itemset X if,

$$\sum_{B \in SW_i} X.iList.B.sumEU + X.iList.B.sumRU < top\_k\_thres$$

then X and its supersets can not be top-k high-utility itemsets. $\therefore$ our algorithm will return the complete set of top-k high-utility itemsets. □

3.3 Another variant: Vert_top-k naive

It has been observed that vertical mining algorithms give their best performance when transactions are sorted in ascending order of *TWU* values [18]. In Vert_top-k Naive variant, *iLists* of items are built from scratch for each window. Whenever a new window arrives, it is scanned to compute the *TWU* of items and items in each transaction are sorted according to ascending order of TWU values. In this variant, there is no reuse of *iLists* of single items from the previous windows. **Note** that top_k_thres computed from a sliding window SW_i is reused for the next sliding window SW_{i+1}. We compare the performance of Vert_top-k Naive with Vert_top-k DS in Section 4.

4 Experiments and results

In this section, we compare the performance of our proposed algorithms against the state-of-the-art algorithm T-HUDS. We conduct experiments on an Intel Xeon(R) CPU=26500@2.00 GHz processor with 16 GB free RAM and has JDK 1.6 installed on a Windows 8.1 operating system. The algorithms were implemented in Java. The datasets used for our experiments are listed in Table 9. The datasets have different characteristics in terms of the number of transactions(#trans), number of distinct items(#items) in the database, and average transaction length(Avg. length). All real datasets except ChainStore were obtained from FiMi repository [11]. ChainStore dataset was obtained from NU-Minebench 2.0 repository [21]. Only the ChainStore dataset has quantity and profit associated with each item in the database. For other datasets, quantity values were randomly generated between 1 to 10 and profit values follows a log-normal distribution in the range 1 to 10 like previous work [37]. The batch size is chosen such that each dataset has approximately ten batches like chosen by previous work [37]. The algorithms are compared on the following parameters: total time taken to mine top-k high-utility itemsets from all windows, and average memory consumed during the execution of the algorithms. We conduct the following set of experiments:

1. Comparison against T-HUDS for varying k.
2. Comparison against T-HUDS for varying window size.
3. Comparison against T-HUDS for different database size.
4. Study the effect of varying distribution on the performance of our algorithm.

4.1 Varying k

In this subsection, we compare the performance of our proposed algorithm against T-HUDS for various sparse and dense datasets. The parameter k is varied from 100 to 900. Experimental results for sparse datasets are shown in Fig. 1. The result on ChainStore dataset shows that our proposed variants perform 20 to 80 times faster than T-HUDS as it does not generate any candidates. T-HUDS algorithm ran out of memory on Retail dataset after running

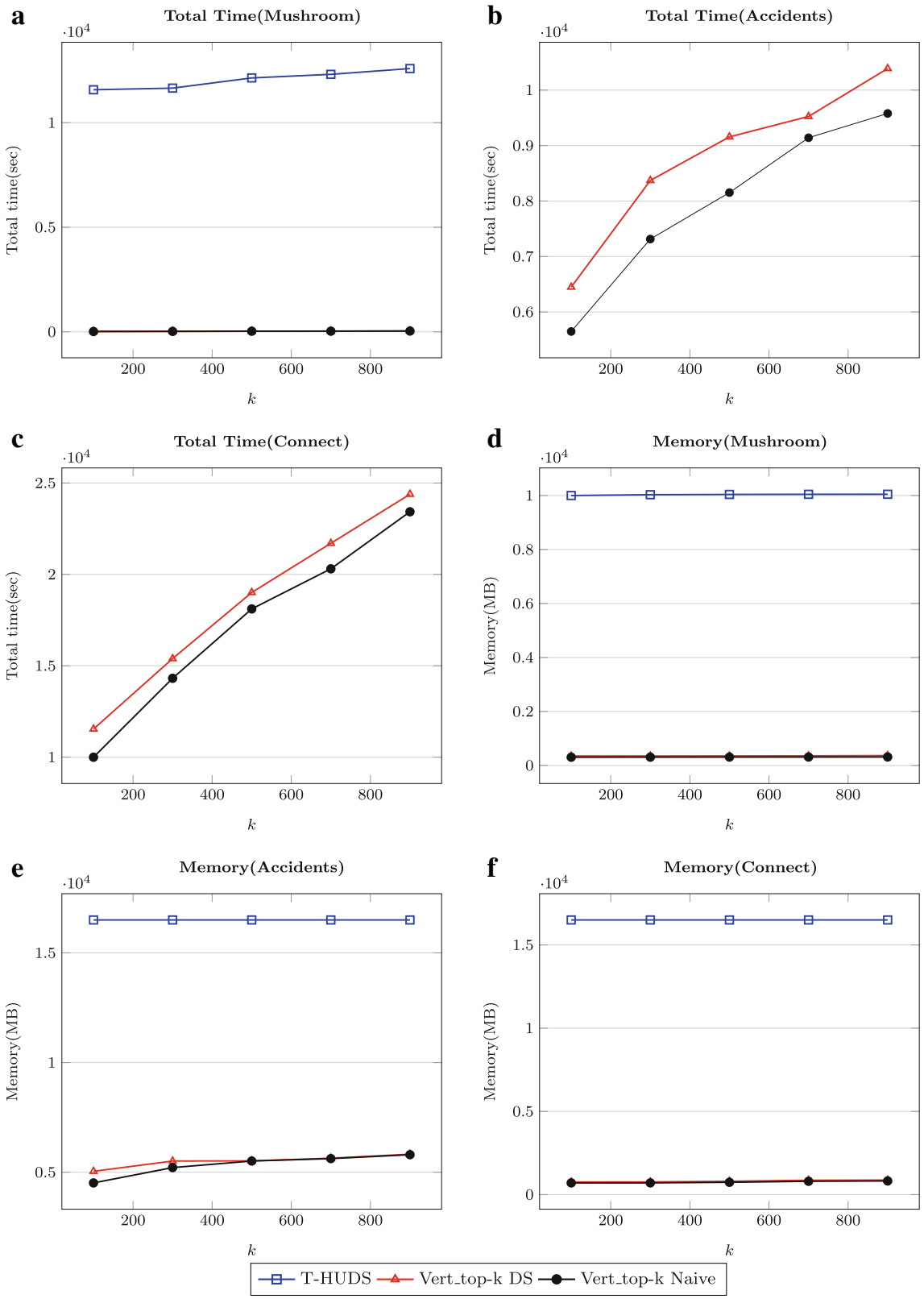


Fig. 2 Performance evaluation on dense datasets.T-HUDS does not complete its execution for the Accidents and Connect dataset and hence its plot is not reported

Table 11 Number of intermediate itemsets generated by Vert_top-k DS and Vert_top-k Naive for dense datasets

Dataset	k	Vert_top-k DS	Vert_top-k Naive
Accidents	100	34,62,426	13,19,399
	300	82,85,458	42,86,676
	500	83,04,026	66,66,111
	700	85,04,771	84,33,778
	900	99,72,508	99,40,390
Connect	100	1,51,06,166	74,79,944
	300	1,67,14,255	1,54,62,548
	500	2,03,73,328	2,00,00,730
	700	2,47,65,852	2,32,05,367
	900	2,86,72,982	2,71,78,615
Mushroom	100	1,11,417	94,188
	300	1,95,197	1,82,074
	500	2,54,847	2,50,610
	700	3,05,703	3,04,713
	900	3,55,748	3,55,663

for few hours. Hence, we compare our proposed variants only for the retail dataset. The results show that Vert_top-k Naive performs better than Vert_top-k DS on both datasets. Vert_top-k Naive constructs the *iLists* from scratch from each window, and thus uses the best possible ordering (*TWU* ascending order) for each window. Liu and Qu [18] observed that algorithms based on inverted-lists perform better when the transactions are sorted in ascending order of *TWU* values, compared to lexicographic and TWU descending ordering. We observe that Vert_top-k Naive performs better than Vert_top-k DS on sparse datasets. The number of intermediate itemsets generated by Vert_top-k DS and Vert_top-k Naive for sparse datasets is shown in Table 10.

Table 12 Number of candidates generated by T-HUDS for ChainStore dataset

Window size	Number of candidates
2	9,73,093
4	8,38,590
6	6,46,312
8	4,55,094
10	2,67,754

Our results verify the hypothesis that Vert_top-k Naive generates fewer intermediate itemsets compared to Vert_top-k DS. The results also demonstrate that our proposed variants consume less memory than T-HUDS.

Experimental results for dense datasets are shown in Fig. 2. Except for Mushroom, T-HUDS ran out of memory for other datasets. Our proposed algorithm performed 300 to 700 times better than T-HUDS for Mushroom dataset as it does not generate any candidates. Table 11 shows the number of intermediate itemsets generated by our proposed variants on dense datasets. As can be observed, the difference in the number of intermediate itemsets generated is less for dense datasets compared to sparse datasets. Our proposed variants consume less memory than T-HUDS on dense datasets. T-HUDS consumes a huge amount of memory as a lot of recursive trees are generated during the candidate generation phase. The algorithm T-HUDS tries to raise the threshold value by using some lower-bound estimates on the utility of itemsets during the first phase. Our algorithm can raise the threshold quickly as it can compute the exact utility of itemsets in a single phase only. The results also show that the time taken by the algorithms increases with k.

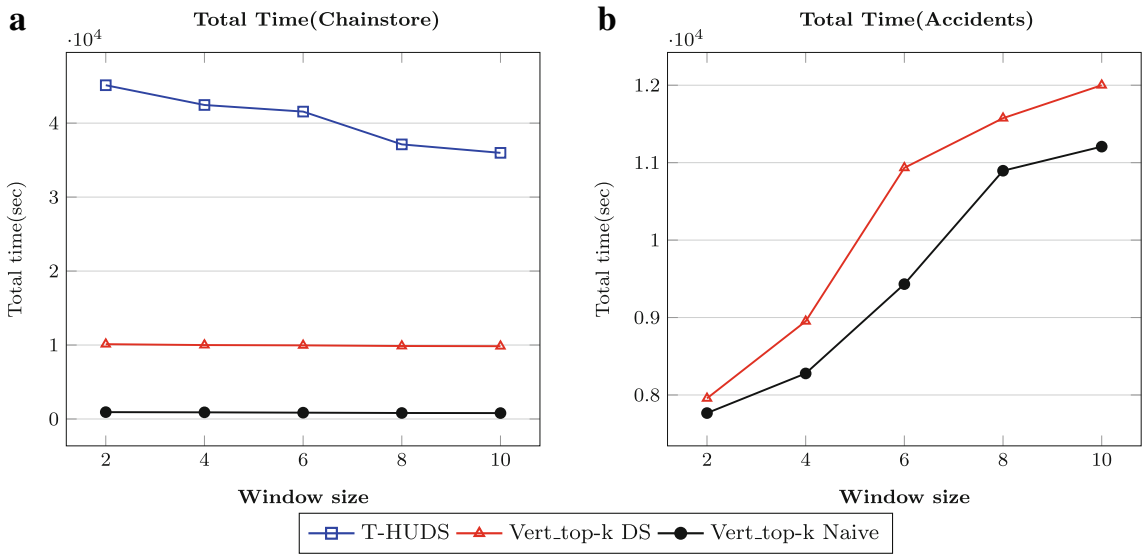


Fig. 3 Effect of varying window size

Table 13 Number of intermediate itemsets generated by our variants for varying window size

Dataset	Window size	Vert_top-k DS	Vert_top-k Naive
ChainStore	2	23,36,05,737	14,87,15,642
	4	57,72,55,173	10,65,49,551
	6	49,77,92,596	10,33,67,820
	8	39,77,92,596	10,30,96,199
	10	29,77,92,596	10,28,43,411
Accidents	2	65,56,169	65,15,303
	4	65,31,135	68,30,037
	6	72,49,794	72,45,200
	8	16,34,16,395	74,50,020
	10	16,34,16,395	74,50,020

4.2 Varying window size (k = 500)

In this subsection, we study the effect of varying number of batches on the performance of algorithms. We fix k to 500 for our experiments. The results for ChainStore and Accidents datasets are shown in Fig. 3. The total time taken by T-HUDS decreases with increase in the window size. As observed from Table 12, the number of candidate high-utility itemsets generated by T-HUDS decreases with increase in the window size. The result shows that the time taken by our algorithms increases with the number of batches.

As can be observed from Table 13, the number of intermediate itemsets generated by our proposed variants decreases with increase in window size for ChainStore which is a sparse dataset. However, the total time and number of intermediate itemsets increase for the Accidents dataset. Dense datasets like Accidents tend to generate a large number of long high-utility itemsets. We believe that the difference in trends of the number of intermediate itemsets is due to the different characteristics of ChainStore and Accidents only.

4.3 Scalability

In this subsection, we observe the effect of scalability on the performance of different algorithms. k was fixed to 500 for these set of experiments. We choose ChainStore and Accidents datasets as they are the largest datasets for the sparse and dense category. Experiments were conducted by varying the number of transactions present in the database. The number of transactions varies from 50 percent to 100 percent of the whole database. The results in Fig. 4 show that the running time increases with increase in the number of transactions. The running time of T-HUDS algorithm increases sharply for a sparse dataset like ChainStore. T-HUDS ran out of memory on Accident dataset for different proportion of transactions. The running time of our proposed variants grows steadily on sparse datasets and increases linearly for dense datasets.

4.4 Experiments to evaluate the effect of distribution

In this subsection, we study the effect of changing distribution on the performance of our proposed variants and results are shown in Fig. 5. The dataset Retail_3x was generated by taking the transactions from the retail dataset and replicating it two times. The replication of transactions was done so that *TWU* distribution of items remains the same across windows. Mushroom_3x was generated from the Mushroom dataset. The transactions were also replicated two times for this dataset. The results show that Vert_top-k DS performs better than Vert_top-k Naive by few seconds

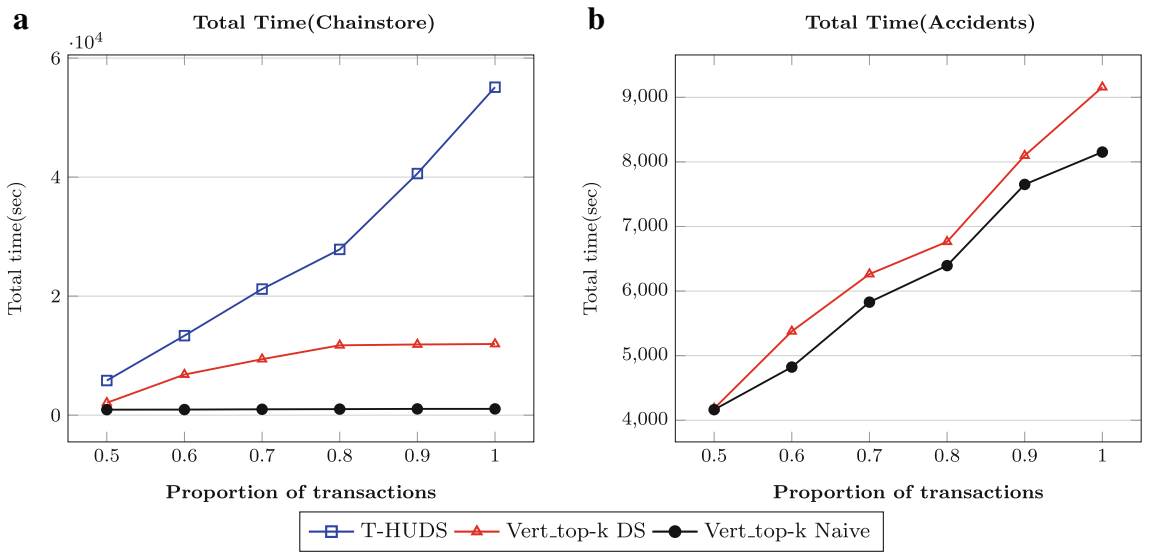


Fig. 4 Effect of scalability

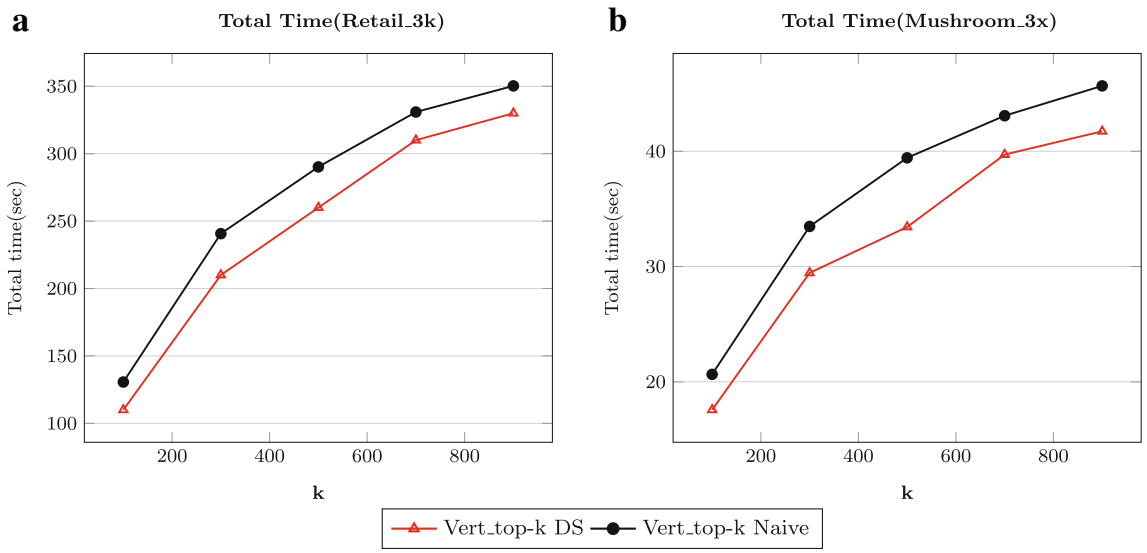


Fig. 5 Effect of distribution on performance of our proposed variants

for both datasets as the *TWU* distribution does not change across windows. Vert_top-k DS gains the benefit of re-using inverted lists from the previous windows and performs better as compared to Vert_top-k Naive.

5 Conclusions and future work

In this paper, we proposed a data structure and an algorithm for mining top-k high-utility itemsets from a data stream. Our proposed algorithm uses an inverted-list data structure and does not generate candidates during the mining process. Experimental results on various real and synthetic datasets demonstrated the superior performance of the proposed algorithm compared to the state-of-the-art algorithms. The naive approach of applying a vertical mining algorithm on each window without using inverted lists from previous windows performs better as it uses the best possible ordering of items. We also observed that our proposed algorithm performs better than the naive approach for datasets where the ordering of items does not change across windows. In the future, we would like to adapt our algorithm for other data stream models as well as scale it for big data using technologies like Apache Storm, Apache Spark, etc. We will also work on improving the efficiency of our proposed algorithm by reducing the number of intersections of inverted lists and using transaction merging technique to reduce the size of the database.

Acknowledgments This work was supported in parts by Infosys Centre for Artificial Intelligence, Indraprastha Institute of Information Technology Delhi (IIIT-Delhi), and Visvesvaraya Ph.D scheme for Electronics and IT.

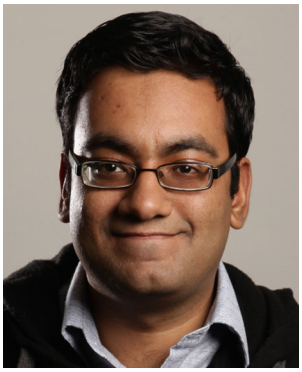
Compliance with Ethical Standards

Conflict of interests The authors declare that they have no conflict of interest.

References

1. Aggarwal CC (2013) Managing and mining sensor data. Springer Science & Business Media
2. Agrawal R, Srikant R et al (1994) Fast algorithms for mining association rules. In: Proceedings of the 20th international conference on very large data bases, VLDB, vol 1215, pp 487–499
3. Ahmed CF, Tanbeer SK, Jeong BS, Lee YK (2009) Efficient tree structures for high utility pattern mining in incremental databases. IEEE Trans Knowl Data Eng 21(12):1708–1721. doi:10.1109/TKDE.2009.46
4. Ahmed CF, Tanbeer SK, Jeong BS, Choi HJ (2012) Interactive mining of high utility patterns over data streams. Expert Syst Appl 39(15):11,979–11,991. doi:10.1016/j.eswa.2012.03.062. http://www.sciencedirect.com/science/article/pii/S0957417412005854
5. Bansal R, Dawar S, Goyal V (2015) An efficient algorithm for mining high-utility itemsets with discount notion, Springer International Publishing, pp 84–98. doi:10.1007/978-3-319-27057-9_6
6. Chan R, Yang Q, Shen YD (2003) Mining high utility itemsets. In: Third IEEE international conference on data mining, pp 19–26. doi:10.1109/ICDM.2003.1250893
7. Chang JH, Lee WS (2003) Finding recent frequent itemsets adaptively over online data streams. In: Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining, ACM, New York, NY, USA, KDD '03, pp 487–492. doi:10.1145/956750.956807
8. Chi Y, Wang H, Yu PS, Muntz RR (2004) Moment: maintaining closed frequent itemsets over a stream sliding window. In: Fourth IEEE international conference on data mining, 2004. ICDM '04, pp 59–66. doi:10.1109/ICDM.2004.10084
9. Dawar S, Goyal V (2014) Up-hist tree: an efficient data structure for mining high utility patterns from transaction databases. In: Proceedings of the 19th international database engineering & applications symposium, ACM, New York, NY, USA, IDEAS '15, pp 56–61. doi:10.1145/2790755.2790771

10. Fournier-Viger P, Wu CW, Zida S, Tseng VS (2014) FHM: Faster High-utility itemset mining using estimated utility co-occurrence pruning, Springer International Publishing, pp 83–92. doi:10.1007/978-3-319-08326-1_9
11. Goethals B, Zaki M (2012) The fimi repository
12. Goyal V, Dawar S, Sureka A (2015) High utility rare itemset mining over transaction databases, Springer International Publishing, pp 27–40. doi:10.1007/978-3-319-16313-0_3
13. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceedings of the 2000 ACM SIGMOD international conference on management of data, ACM, New York, NY, USA, SIGMOD '00, pp 1–12, doi:10.1145/342009.335372
14. Krishnamoorthy S (2015) Pruning strategies for mining high utility itemsets. Expert Syst Appl 42(5):2371–2381. doi:10.1016/j.eswa.2014.11.001. http://www.sciencedirect.com/science/article/pii/S0957417414006848
15. Leung CKS, Jiang F (2011) Frequent itemset mining of uncertain data streams using the damped window model. In: Proceedings of the 2011 ACM symposium on applied computing, ACM, New York, NY, USA, SAC '11, pp 950–955. doi:10.1145/1982185.1982393
16. Li HF, Lee SY (2009) Mining frequent itemsets over data streams using efficient window sliding techniques. Expert Syst Appl 36(2, Part 1):1466–1477. doi:10.1016/j.eswa.2007.11.061. http://www.sciencedirect.com/science/article/pii/S0957417407006057
17. Li HF, Huang HY, Chen YC, Liu YJ, Lee SY (2008) Fast and memory efficient mining of high utility itemsets in data streams. In: 2008 eighth IEEE international conference on data mining, pp 881–886. doi:10.1109/ICDM.2008.107
18. Liu M, Qu J (2012) Mining high utility itemsets without candidate generation. In: Proceedings of the 21st ACM international conference on information and knowledge management, ACM, New York, NY, USA, CIKM '12, pp 55–64. doi:10.1145/2396761.2396773
19. Liu Y, Liao Wk, Choudhary A (2005) A two-phase algorithm for fast discovery of high utility itemsets. Springer, Berlin, Heidelberg, pp 689–695. doi:10.1007/11430919_79
20. Pei J, Han J, Mao R et al (2000) Closet: an efficient algorithm for mining frequent closed itemsets. In: ACM SIGMOD workshop on research issues in data mining and knowledge discovery, vol 4, pp 21–30
21. Pisharath J, Liu Y, Wk Liao, Choudhary A, Memik G, Parhi J (2005) Nu-minebench 2.0. Department of Electrical and Computer Engineering, Northwestern University, Tech Rep
22. Rathore S, Dawar S, Goyal V, Patel D (2016) Top-k high utility episode mining from a complex event sequence. In: Proceedings of the 21st international conference on management of data, computer society of India
23. Ryang H, Yun U (2016) High utility pattern mining over data streams with sliding window technique. Expert Syst Appl 57:214–231. doi:10.1016/j.eswa.2016.03.001. http://www.sciencedirect.com/science/article/pii/S0957417416300902
24. Salton G, Buckley C (1988) Term-weighting approaches in automatic text retrieval. Inf Process Manag 24(5):513–523. doi:10.1016/0306-4573(88)90021-0. http://www.sciencedirect.com/science/article/pii/0306457388900210
25. Shie BE, Yu PS, Tseng VS (2012) Efficient algorithms for mining maximal high utility itemsets from data streams with different models. Expert Syst Appl 39(17):12,947–12,960. doi:10.1016/j.eswa.2012.05.035. http://www.sciencedirect.com/science/article/pii/S095741741200749X
26. Tseng VS, Chu CJ, Liang T (2006) Efficient mining of temporal high utility itemsets from data streams. In: Second international workshop on utility-based data mining, Citeseer, vol 18
27. Tseng VS, Wu CW, Shie BE, Yu PS (2010) 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, ACM, New York, NY, USA, KDD '10, pp 253–262. doi:10.1145/1835804.1835839
28. Tseng VS, Shie BE, Wu CW, Yu PS (2013) Efficient algorithms for mining high utility itemsets from transactional databases. IEEE Trans Knowl Data Eng 25(8):1772–1786. doi:10.1109/TKDE.2012.59
29. Tseng VS, Wu CW, Fournier-Viger P, Yu PS (2016) Efficient algorithms for mining top-k high utility itemsets. IEEE Trans Knowl Data Eng 28(1):54–67. doi:10.1109/TKDE.2015.2458860
30. Wu CW, Shie BE, Tseng VS, Yu PS (2012) Mining top-k high utility itemsets. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining, ACM, New York, NY, USA, KDD '12, pp 78–86. doi:10.1145/2339530.2339546
31. Wu CW, Lin YF, Yu PS, Tseng VS (2013) Mining high utility episodes in complex event sequences. In: Proceedings of the 19th ACM SIGKDD international conference on knowledge discovery and data mining, ACM, New York, NY, USA, KDD '13, pp 536–544. doi:10.1145/2487575.2487654
32. Yang B, Huang H (2010) Topsil-miner: an efficient algorithm for mining top-k significant itemsets over data streams. Knowl Inf Syst 23(2):225–242. doi:10.1007/s10115-009-0211-5
33. Yen SJ, Lee YS, Wu CW, Lin CL (2009) An efficient algorithm for maintaining frequent closed itemsets over data stream. Springer, Berlin, Heidelberg, pp 767–776. doi:10.1007/978-3-642-02568-6_78
34. Yin J, Zheng Z, Cao L (2012) Uspan: an efficient algorithm for mining high utility sequential patterns. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining, ACM, New York, NY, USA, KDD '12, pp 660–668. doi:10.1145/2339530.2339636
35. Yin J, Zheng Z, Cao L, Song Y, Wei W (2013) Efficiently mining top-k high utility sequential patterns. In: 2013 IEEE 13th international conference on data mining, pp 1259–1264. doi:10.1109/ICDM.2013.148
36. Zaki MJ, Parthasarathy S, Ogihara M, Li W et al (1997) New algorithms for fast discovery of association rules. In: KDD, vol 97, pp 283–286
37. Zihayat M, An A (2014) Mining top-k high utility patterns over data streams. Inf Sci 285:138–161. doi:10.1016/j.ins.2014.01.045. http://www.sciencedirect.com/science/article/pii/S0020025514000814



Siddharth Dawar received his B.Tech degree in Information Technology in 2012 from Guru Gobind Singh Indraprastha University, Delhi, India and his M.Tech degree in Information Security from Indraprastha Institute of Information Technology, Delhi, India in 2014. Since 2014 he is a Ph.D scholar at Indraprastha Institute of Information Technology, (IIIT-Delhi), Delhi, India. His research interests include data mining, machine learning and information security.

Veronica Sharma recieved her B.Tech in Computer Science and Engineering in 2013 at Indira Gandhi Institute of Technology (IGIT), Delhi, India and her M.Tech Degree in Computer Science from Indraprastha Institute of Technology (IIIT-Delhi), Delhi, India in 2016. Her research interests include Data Mining and Machine Learning. Other areas of interest includes exploring the recent trends and algorithms in Neural Networks and Deep Learning.

Vikram Goyal received his M.Tech. in Information Systems in 2003 at Netaji Subhash Institute of Technology, Delhi, India and his Ph.D degree in Computer Science from the Department of Computer Science and Engineering at IIT Delhi, India in 2009. Since 2009 he is a faculty at Indraprastha Institute of Information Technology, (IIIT-Delhi), New Delhi, India. His research interests include Knowledge Engineering and Data Privacy. Other areas of interest are data mining, big data analytics & information security. He has completed a couple of projects with DST India and Deity, India on the problems related to Privacy in Location-based services and Digitized Document Fraud Detection, respectively.