Data Science and Engineering (2024) 9:177–203
https://doi.org/10.1007/s41019-024-00241-2

RESEARCH PAPER

Efficient Top-*k* Frequent Itemset Mining on Massive Data

Xiaolong Wan[1] · **Xixian Han**[1]

Received: 29 May 2023 / Revised: 22 October 2023 / Accepted: 30 December 2023 / Published online: 6 February 2024

Abstract
Top-*k* frequent itemset mining (top-*k* FIM) plays an important role in many practical applications. It reports the *k* itemsets with the highest supports. Rather than the subtle minimum support threshold specified in FIM, top-*k* FIM only needs the more understandable parameter of the result number. The existing algorithms require at least two passes of scan on the table, and incur high execution cost on massive data. This paper develops a prefix-partitioning-based PTF algorithm to mine top-*k* frequent itemsets efficiently, where each prefix-based partition keeps the transactions sharing the same prefix item. PTF can skip most of the partitions directly which cannot generate any top-*k* frequent itemsets. Vertical mining is developed to process the partitions of vertical representation with the high-support-first principle, and only a small fraction of the items are involved in the processing of the partitions. Two improvements are proposed to reduce execution cost further. Hybrid vertical storage mode maintains the prefix-based partitions adaptively and the candidate pruning reduces the number of the explored candidates. The extensive experimental results show that, on massive data, PTF can achieve up to 1348.53 times speedup ratio and involve up to 355.31 times less I/O cost compared with the state-of-the-art algorithms.

Keywords Top-*k* frequent itemset mining · PTF · Prefix-based partitioning · Hybrid vertical storage · Candidate pruning

1 Introduction

Frequent itemset mining (FIM) plays an important role in many practical applications, including data mining [2, 48, 49], medical diagnosis [32], spatio-temporal data analysis [9] and bioinformatics [41]. Given a set of items, each transaction in the table contains a subset of the items. FIM discovers the sets of items (called *frequent itemsets*) that co-occur frequently among huge exploration space, which increases exponentially with respect to the number of the items. For each itemset, its support is the number of transactions containing it. FIM specifies a minimum support threshold to restrict the acquired itemsets whose supports are no less than the threshold.

Due to its practical importance, FIM has received extensive attentions since firstly proposed in [4, 5], and many algorithms have been proposed [3, 29, 31, 40]. However, it is found that the users normally are difficult to set the proper minimum support threshold to acquire the satisfied results [58], especially when without the relevant knowledge.

- If the support threshold is set too high, very few (or even no) frequent itemsets can be discovered, and the users cannot find enough useful information for their decisions.
- If the support threshold is set too small, a significantly large number of frequent itemsets will be generated, which not only degrades the efficiency seriously, but also overwhelms the users by enormous size of the results.

In order to obtain the reasonable size of frequent itemsets, one option is to try the different support thresholds over a certain range, which obviously is not acceptable due to high execution cost. Although the maximal and closed frequent itemset mining algorithms [11, 28, 35, 52, 57] are proposed to identify condensed representatives of frequent itemsets, they still need to specify the support threshold and cannot control the number of the required results.

For a better applicability of FIM, especially for the common users who are lack of the relevant knowledge, a parameter-light solution is preferred to let the data speak for themselves [33, 54]. Top-*k* frequent itemset mining (Top-*k* FIM) only asks the users to specify the number (*k*) of the

✉ Xixian Han
hanxx@hit.edu.cn

[1] School of Computer Science and Technology, Harbin Institute of Technology, No. 92, Xidazhi Street, Harbin, Heilongjiang, China

required results directly, which is much comprehensible for the users, rather than the subtle support threshold.

1.1 Gap Analysis of Top-*k* FIM

This paper focuses on top-*k* FIM on massive data. Normally, massive data refer to data at massive scale. In this paper, we specify the scope of massive data as data set that cannot be maintained entirely on main memory when performing mining.

In order to understand the research question better, a gap analysis is performed first [12], which assesses the differences between the actual performance and expected performance. On massive data, the memory cannot hold the entire data set and it will take a relatively long time to just scan the data set sequentially once. Under the circumstance, an efficient algorithm is expected to compute top-*k* frequent itemsets on massive data quickly.

The existing top-*k* FIM algorithms adopt apriori-like execution mode [15, 23] and FP-tree-based execution mode [14, 45]. The former algorithms often need multiple scans on the table, which will incur a high I/O cost and a high computation cost for counting support on massive data. The latter algorithms have to construct the FP-tree first in memory by two passes of scan. Besides the relatively high I/O cost, it is unrealistic to assume that, on massive data, the complete FP-tree can be kept in the memory. In a word, the existing algorithms cannot deal with top-*k* FIM on massive data efficiently.

Considering that the number *k* of the results is relatively small, the solution with a high efficiency on massive data should involve the subset of the transactions and the subsets of items in the involved transactions, which are helpful for computing the top-*k* frequent itemsets. This is the research idea of the algorithm proposed in this paper.

1.2 Our Contributions

This paper develops the prefix-based partitioning strategy given the specified order of the items. Each partition contains the transactions beginning with a certain item. By the prefix-based partitions, this paper devises PTF algorithm to mine top-*k* frequent itemsets on massive data efficiently. PTF utilizes a min-heap *MH* of size *k* to maintain the *k* itemsets discovered so far with the highest supports. Before processing the prefix-based partitions sequentially, PTF initializes *MH* with the pre-constructed structures. Due to the prefix-based partitioning strategy, the itemsets beginning with a certain item can be generated separately in the corresponding partition. The partitions can be skipped directly, whose maximum supports of itemsets generated possibly are no larger than the minimum support of itemsets in *MH*. For the partitions to be processed, vertical mining is performed on the transactions of vertical representation by the high-support-first principle, and only a small fraction of the items are involved in the processing of the partitions. In this way, *PTF not only skips the entirely unnecessary partitions but also skips the unpromising items in the processed partitions*. Two improvements are utilized in PTF. One improvement introduces the *hybrid vertical storage mode* to reduce the involved I/O cost furthermore. The tid-sets of the items can be stored in the form of tid-lists, dif-lists or bit-vectors adaptively. The corresponding operations are devised to intersect the tid-sets of items in the different forms. Another improvement exploits pruning strategy to reduce the number of the considered candidates by anti-monotone property. The extensive experiments are conducted on synthetic and real data sets. The experimental results show that, compared with the existing algorithms, PTF can achieve up to three orders of magnitude speedup ratio and involve two orders of magnitude less I/O cost.

The contributions of this paper are listed as follows:

- This paper develops a novel prefix-partitioning-based PTF algorithm to mine top-*k* frequent itemsets on massive data efficiently, which only retrieves a small fraction of transaction table for computing the results.
- Hybrid vertical storage mode is proposed in this paper to keep the vertical representation of the items in one of three storage formats adaptively and reduce the involved I/O cost significantly.
- Vertical mining is performed on the processed partitions of vertical representation, where the itemsets are generated with the high-support-first principle to speed up the in-memory computation.
- The experimental results on synthetic and real-life data sets show that PTF outperforms the existing algorithms significantly.

The rest of the paper is organized as follows. We first survey the related works in Sect. 2, followed by preliminaries described in Sect. 3. The prefix-based partitioning is introduced in Sect. 4. In Sect. 5, we develop PTF algorithm, whose performance is evaluated in Sect. 6. The paper is concluded in Sect. 7.

2 Related Works

2.1 Frequent Itemset Mining

Given a support threshold *minsup*, FIM aims to discover the frequent itemsets among the universe of items, whose supports are greater or equal to *minsup*. By trading off in varying degrees the potential itemset number and the counting strategy, the existing FIM algorithms can be divided into

three categories: level-wise approach, pattern-growth-based approach and vertical-format-based approach.

Level-wise approach. Apriori algorithm [5] adopts a level-wise execution mode. It uses the downward closure property to prune the search space. The frequent b-itemsets in F_b are used to generate the candidates C_{b+1} of the frequent $(b+1)$-itemsets, where b-itemset is a set of items having b items. By one pass of scan, the supports of candidates in C_{b+1} are computed to find the frequent $(b+1)$-itemsets F_{b+1}. The process is repeated until F_{b+1} is empty. Apriori algorithm often needs multiple scans over the table and incurs a high I/O cost on massive data, whose execution cost is analyzed in [20]. Many researchers study how to improve and extend the Apriori for a better performance. The improvements include partitioning strategy [47], hashing technique [42], pre-computation [31] and distributed/parallel processing [7].

Pattern-growth-based approach. Han et al. [30] propose FP-growth algorithm to mine the frequent patterns. FP-tree is a compact prefix-based trie structure to compress the transaction table. FP-growth starts with a frequent length-1 pattern as initial suffix pattern, and the set of frequent items co-occurring with the suffix pattern is extracted as conditional-pattern base, which is then constructed as conditional FP-tree. With the current suffix pattern and the conditional FP-tree, if the conditional FP-tree is not empty, FP-growth performs mining recursively. Grahne et al. [24] devise FPGrowth* algorithm, which utilizes a special data structure FP-array to reduce the traversal time on FP-tree and speed up the FP-growth method significantly. Pei et al. [43] devise H-mine based on a novel hyper-linked data structure H-struct to efficiently mine databases with different data characteristics. H-mine has very limited and precisely predictable space overhead, and can be scaled up to large database by partitioning. Liu et al. [39] propose the AFOPT algorithm to mine the complete set of frequent patterns, which uses ascending frequency ordered prefix-tree to store the conditional databases, and the tree is traversed in top-down fashion.

Vertical-format-based approach. The level-wise approach and the pattern-growth-based approach discover the frequent itemsets by mining on the transaction table in the horizontal data format. Zaki [55] presents the ECLAT algorithm to mine the frequent itemsets on the vertical data format. Each itemset is associated with a TID list of transactions containing it. The frequent itemsets can be discovered by the intersection of the corresponding TID lists. According to Apriori property, the frequent $(b+1)$-itemsets can be generated by the previous generated frequent b-itemsets. There are many extensions or improvements for the ECLAT algorithm by some novel vertical representations, including Diffset [56], N-list [19], Nodeset [18], DiffNodeset [17] and NegNodeset [8].

FIM often generates a large number of frequent itemsets, which makes it difficult for the users to obtain knowledge from them easily. In order to limit size of the output, the condensed representation of the frequent itemsets usually is required, such as maximal frequent itemsets [11, 28, 52] and closed frequent itemsets [35, 57], which are much smaller than the frequent itemsets. However, the computation of the condensed representation still requires a specified *minsup*, which is tricky to find an appropriate value.

Besides, there also exist some approaches for mining frequent itemsets based on symbolic artificial intelligence [25], which integrate the constraints and return the required itemsets satisfying the constraints. For the cases that the hard constraints do not guarantee the best results, the solutions with soft constraints are devised to entail the relaxing constraints [10, 51]. The constraint programming can be embedded into itemset mining [26, 27]. The declarative constraint programming can be used to model and solve the constraint-based mining task. The constraint-based itemset mining algorithms mainly consider how to use declarative constraint programming to model and solve the issue of itemset mining. They provide another perspective for addressing itemset mining. However they do not give sufficient consideration to the efficient execution of algorithm. Comparatively, the main concern of this paper is how to design the algorithm and acquire the high execution efficiency.

2.2 Top-*k* Frequent Itemset Mining

Top-k FIM only needs to specify the number of the required results, a much more understandable and applicable parameter than the support threshold.

Given a upper bound u of itemset size, Fu et al. [23] propose the first algorithm *Itemset-Loop* for mining N-most interesting itemsets, the most frequent N b-itemsets for each b $(1 \le b \le u)$. Initially, *Itemset-Loop* sorts all 1-itemsets in the descending order of support, and reports the first N 1-itemsets to be the N-most interesting 1-itemsets P_1. The candidate b-itemsets C_b $(b \ge 2)$ are generated from potential $(b-1)$-itemsets in the way of the Apriori-gen algorithm [5]. P_b is the N-most interesting b-itemsets in C_b. Let $support_b$ be the minimum support value of the N-th b-itemset in P_b, and let $lastsupport_b$ be the support of the last b-itemset in P_b. If $lastsupport_1 \ge support_b$, P_1 should be augmented by adding 1-itemsets with supports no less than $support_b$. P_2 is updated correspondingly by including the 2-itemsets whose supports are no less than $support_b$. If $lastsupport_1 \le support_b$, $support_b$ will be compared with $lastsupport_2, \cdots, lastsupport_{b-1}$ successively with similar processing. *Itemset-Loop* iterates the process similarly until P_u is generated. *Itemset-iLoop* improves *Itemset-Loop* in the way that when generating P_b, the loop back operation first

examines $(b-1)$-itemsets rather than loop backing to augmentation of the potential 1-itemsets.

Cheung et al. [14] also consider the N-most interesting itemset mining. BOMO algorithm is first introduced which is based on FP-tree structure [30]. Initially, a complete FP-tree is built with all items in the database. Different from FP-growth algorithm, BOMO starts at the middle of the header table to achieve a good trade-off between the number of large conditional FP-trees and pruning power. From the middle position, BOMO first goes upward to the top and then from the (middle + 1) position down to the bottom of the header table. During the mining process, $result_b$ is used to maintain the current resulting set of the N-most interesting b-itemsets. The current support threshold δ can be kept for all the itemsets and be used to prune the conditional FP-trees. Since BOMO requires a complete FP-tree with all items, another algorithm LOOPBACK is devised to build a smaller initial FP-tree by an initial support threshold $\delta > 0$.

Chuang et al. [15] explore pure top-k frequent patterns in the presence of the memory constraint. Different from the N-most interesting itemset mining that retrieves the N most frequent b-itemsets ($1 \leq b \leq u$) for an upper bound u, the pure top-k frequent patterns refer to the k most frequent itemsets without any constraint of item-lengths. It is noted that FP-tree-based solutions cannot be memory-constraint since the size of FP-tree is proportional to the database size, and MTK is developed in [15] as a level-wise-based search algorithm to compute pure top-k frequent itemsets. In order to solve the problem of level-wise algorithms that generate a large number of candidates potentially, MTK adopts the δ-stair search strategy. Initially, k most frequent 1-itemsets and the hash table of all 2-itemsets in each transaction are maintained. The δ-stair search shares M_c candidates equally to δ different itemset lengths. The upward δ-stair search step and downward δ-stair search step are adaptively switched until the results are found. The δ-stair search combines the advantages of horizontal first search and vertical first search.

Pyun et al. [45] consider FP-tree-based top-k frequent patterns with combination reducing technique. During the mining process on FP-tree, single-paths with many items are often generated, and a large number of combinations from the items will be generated in the single-paths. In order to reduce the pattern combinations, CRM algorithm defines strict closed pattern, i.e., all sub-patterns have the same support. The composite pattern is defined as the combination of the strict closed patterns with a general pattern. Initially CRM builds a global FP-tree containing all information of transaction table. CRM first selects the middle item in the header table of the FP-tree, generates the conditional database and tree, performs the mining operation. Top-k list maintains the frequent patterns (or composite patterns) found in the mining steps. CRM then selects the items above and below the middle item, respectively, and processes them in the similar way as mentioned above. After the above mining step, CRM converts the composite patterns in top-k list into the corresponding general patterns to release actual frequent patterns. CRMN algorithm is devised also to utilize multiple top-k lists to mine N-itemset top-k patterns by a similar operation to CRM.

Top-rank-k frequent itemset mining [21] is very similar to top-k FIM, and it returns the frequent itemsets whose ranks are not larger than k. Abdelaal et al. [1] customize the three N-list-based or Nodeset-based FIM algorithms to develop the latest customized top-rank-k FIM algorithms (TK_FIN, TK_PrePost and TK_PrePost+). The dynamic minimum support threshold strategy is used to increase the running minimum support threshold gradually. The transactions are first scanned to generate the 1-itemsets, which are inserted into the table structure TabK with k entries. During the second scan, the transactions are retrieved and arranged in descending order of the supports of the items. The prefix tree is built on the retrieved transactions. The tree is scanned to generate the N-lists or Nodesets of 2-itemsets. The rest of execution is performed in level-wise fashion. There are also other researches and extensions about top-k FIM, such as top-k FIM under differential privacy conditions for preserving privacy [36, 38], top-k FIM over data stream [13], top-k frequent-regular closed patterns [6], top-k frequent closed itemsets [53], top-k closed sequential patterns [44, 50] and top-k maximal frequent itemsets [46].

In this paper, we focus on the pure top-k frequent itemset mining. The latest top-k FIM algorithms, including level-wise algorithm [15] and prefix-tree-based algorithm [45], require at least two passes of scan on $\mathcal{T}$. On massive data, they require a high execution cost to mine the top-k frequent itemsets.

3 Preliminaries

In this section, the problem of top-k ***f***requent ***i***temset ***m***ining (top-k FIM) is firstly defined in Sect. 3.1, and then problem analysis is provided in Sect. 3.2 to motivate an efficient algorithm. It is considered that an efficient algorithm should not retrieves all transactions but only involve the subsets of items in the required transactions useful for the mined results. The definitions of running minimum support and promising items are devised in Sect. 3.2 to determine the useful items when mining top-k FIM.

3.1 Problem Definition

Let $\mathcal{I} = \{x_1, x_2, \ldots, x_d\}$ be the set of all items, the itemset X is a subset of $\mathcal{I}$, i.e., $X \subseteq \mathcal{I}$, which is called b-itemsets if $|X| = b$. Let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ be the set of all transactions. $\forall t_i \in \mathcal{T}$, $t_i \subseteq \mathcal{I}$, and a unique transaction identifier (tid) is

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

Transaction table

TID	*Transactions*
1	x_2, x_6, x_{10}
2	x_1, x_2, x_9, x_{10}
3	x_3, x_4
4	x_2, x_{10}
5	$x_2, x_4, x_5, x_8, x_9, x_{10}$
6	x_2, x_9, x_{10}
7	$x_2, x_5, x_8, x_9, x_{10}$
8	$x_2, x_3, x_4, x_9, x_{10}$
9	x_2, x_9
10	x_2, x_4, x_{10}
11	x_7, x_{10}
12	x_2, x_8, x_9

Top 8 *frequent itemsets*

Itemset	*Support*
$\{x_4\}$	4
$\{x_9, x_{10}\}$	5
$\{x_2, x_9, x_{10}\}$	5
$\{x_2, x_9\}$	7
$\{x_9\}$	7
$\{x_2, x_{10}\}$	8
$\{x_{10}\}$	9
$\{x_2\}$	10

associated with t_i. For the itemset X, one of the most important property is its *support*, the number of the transactions in $\mathcal{T}$ containing X. Mathematically, the support $\sigma(X)$ of the itemset X can be stated as $\sigma(X) = |\{t_i | X \subseteq t_i, t_i \in \mathcal{T}\}|$.

The itemset X is called *frequent* if its support is no less than the user-specified threshold *minsup*. As defined in Definition 1, FIM discovers all the frequent itemsets in $\mathcal{T}$ given *minsup*.

Definition 1 (*Frequent itemset mining*) Given $\mathcal{T}$ and *minsup*, frequent itemset mining finds all itemsets FI (***F***requent ***I***temsets), whose supports are no less than *minsup*, i.e., $FI = \{X | X \subseteq \mathcal{I}, \sigma(X) \geq minsup\}$.

Due to the difficulty of choosing the appropriate value of *minsup*, top-*k* FIM is devised to find k most frequent itemsets without the specification of *minsup*, as defined in Definition 2. The number k of the required frequent itemsets is much easier to be understood compared with *minsup*. Let $\mathcal{P}_\mathcal{I}$ be the power set of $\mathcal{I}$ excluding the empty set.

Definition 2 (*Top-k frequent itemset mining*) Given $\mathcal{T}$ and k, top-k frequent itemset mining aims to find k most frequent itemsets FI_k, where $|FI_k| = k$ and $\forall X \in FI_k, \forall Y \in \mathcal{P}_\mathcal{I} - FI_k$, $\sigma(X) \geq \sigma(Y)$.

Example 1 As shown in Fig. 1, a running example is provided in this paper to illustrate the algorithm execution. There are 10 items $x_1, x_2, \ldots, x_{10}$ and 12 transactions $t_1, t_2, \ldots, t_{12}$ in the running example. We want top-8 FIM results. For example, given itemset $\{x_2, x_9, x_{10}\}$, its support is 5since it is contained in five transactions t_2, t_5, t_6, t_7, t_8.

Given $\mathcal{I}$, the number $|\mathcal{P}_\mathcal{I}|$ of the possible itemsets is $2^d - 1$, which increases exponentially with respect to d. Here $d = |\mathcal{I}|$ is the number of the items. Obviously, the brute-force approach to search each possible itemset is practically impossible. Thus, all the solutions to solve FIM utilize the *downward closure property* in Property 3.1 to reduce the search space significantly.

Property 3.1 (Downward closure property) *If an itemset X is frequent, then all the subsets of X are frequent.*

Property 3.2 (Anti-monotone property) *Given an itemset X, for any subset Y of X ($Y \subseteq X$), we have $\sigma(X) \leq \sigma(Y)$.*

Property 3.1 is very intuitive, $\forall X \subseteq t_i$, any subset of X is contained in t_i also. The downward closure property can be rephrased as the anti-monotone property (Property 3.2). The frequently used symbols in this paper are shown in Table 1.

3.2 Problem Analysis

On massive data, $\mathcal{T}$ usually maintains a large number of transactions, tens of millions or even billions. In such a size of data, it will take a relatively long time just to scan it sequentially once, not to mention the involvement of expensive operations. Even though the mainstream machine has a larger and larger capacity of main memory, it is reasonable to consider that all transactions of $\mathcal{T}$ cannot be maintained entirely in main memory.

The number k of the required results typically is small (for example 10,000), since too many frequent itemsets provide

Table 1 Summary of symbols

Symbol	Meaning
$\mathcal{I}$	the set of items $\{x_1, x_2, \ldots, x_d\}$
k	The number of the required results
FI_k	The k most frequent itemsets
MH	Min heap to keep the k most frequent itemsets so far
$rmsup$	The kth highest support of the itemsets so far
P_i	The prefix-based partition corresponding to x_i
AR_i	The array to maintain the promising items in P_i
QE	The priority queue to keep the candidates

item	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
support	1	10	2	4	2	1	1	3	7	9

four promising items given rmsup = 3

Fig. 2 The illustration of promising items

excessive information that the users cannot find the useful knowledge easily. Evidently, among all the possible itemsets of size $2^d - 1$, the selected k most frequent itemsets are much smaller, even negligible, to be more precise. Therefore, *the ideal case is to involve the subsets of items in the required transactions only* which is helpful to compute FI_k.

Normally, itemset mining is considered to be a rather expensive task. An efficient implementation will be much helpful for the users. In our opinion, three important factors should be considered for designing an efficient top-k FIM algorithm on massive data.

- Although *minsup* is not specified explicitly, it is used throughout the execution to discard the candidate itemsets by anti-monotone property. The existing algorithms set *minsup* equal to zero initially, and gradually raise *minsup* when more itemsets are generated. Some precomputation is required to raise the initial *minsup* properly at a low cost. This can improve the efficiency partly.
- On massive data, the I/O cost of a single sequential scan is already relatively high. In order to achieve high efficiency, a *sub-linear algorithm* is more desirable, which only involves a small fraction of the transactions. This requires designing the more resultful pruning strategy to reduce the number of the involved transactions.
- Besides the pruning of the whole transactions, it is also desirable to narrow down the used items in each involved transaction correspondingly.

Definition 3 (*Running minimum support*) During the execution of mining top-k frequent itemsets, the support of the kth most frequent itemsets generated so far is referred as the ***r***unning ***m***inimum ***sup***port (denoted by *rmsup*).

Definition 4 (*Promising item*) $\forall x_i \in \mathcal{I}$, if $\sigma(x_i) > rmsup$, x_i is a promising item. Otherwise, x_i is unpromising and any superset of x_i cannot be the top-k frequent itemsets.

Example 2 As depicted in Fig. 2, given $rmsup = 3$, the promising items in the running example are x_2, x_4, x_9, x_{10} since their supports are greater than 3.

In computing top-k frequent itemsets, we only need to consider the promising items, and discard others directly. This is also the natural result of downward closure property. Clearly, the fewer the number of promising items, the lower computation cost.

$\forall x_i, x_j \in \mathcal{I}(1 \le i \ne j \le d)$, $\sigma(x_i x_j) \le \min\{\sigma(x_i), \sigma(x_j)\}$ according to anti-monotone property. If we focus on the transactions containing x_i, a.k.a. the ***cov***er set $COV(\mathcal{T}, x_i) = \{t_j | x_i \in t_j, t_j \in \mathcal{T}(1 \le j \le n)\}$, every itemset generated from $COV(\mathcal{T}, x_i)$ can be requested to contain x_i, since x_i occurs in every transaction in $COV(\mathcal{T}, x_i)$. Any itemset containing $x_i x_j$, if it exists, can be discovered in $COV(\mathcal{T}, x_i)$. If $\sigma(x_i x_j) \le rmsup$, the computation on $COV(\mathcal{T}, x_i)$ to find top-k frequent itemsets, which contain x_i definitely, does not need to consider x_j. Since $\sigma(x_i x_j) \le \sigma(x_j)$, the much fewer promising items will be considered when processing $COV(\mathcal{T}, x_i)$ compared with those when processing $\mathcal{T}$.

4 Prefix-Based Partitioning

Based on the analysis mentioned in Sect. 3.2, all itemsets can be generated by the cover set of individual items with fewer promising items involved. This section firstly devises a prefix-based partitioning strategy in Sect. 4.1 to split transaction table and materialize the cover sets of the items without the generation of duplicate itemsets. The space consumption of the prefix-based partitioning in direct form is analyzed in Sect. 4.2, showing that prefix-based partitioning is a workable strategy in space. Some useful structures are described in Sect. 4.3, which compute the supports of individual items in prefixed-based partitions to help item-level and partition-level pruning.

4.1 Partitioning Strategy

Given $\mathcal{I} = \{x_1, x_2, \ldots, x_d\}$, assume that there exists a total order among the items. Without loss of generality, let $x_1 \prec x_2 \prec \ldots \prec x_d$, the items are arranged in alphabetical order. The items in any itemsets are arranged in this order.

The prefix-based partitioning splits the table into d partitions, in each of which the transactions *share the same prefix item*. $\forall t \in \mathcal{T}$ and t contains b items, i.e., $t = \{x_{1'}, x_{2'}, \ldots, x_{b'}\}$ and $x_{1'} \prec x_{2'} \prec \ldots \prec x_{b'}$. $\forall 1 \le j \le b$, the suffix items of t, $\{x_{j'}, \ldots, x_{b'}\}$, are outputted to the partition $P_{j'}$. The prefix-based partitions $P_1, P_2, \ldots, P_d$ can be acquired by a sequential scan on $\mathcal{T}$.

For prefix-based partition $P_i (1 \le i \le d)$, $\forall t \in \mathcal{T}$, if $x_i \in t$, the suffix items of t beginning with x_i will be maintained in P_i, otherwise, the items of t will not appear in P_i. In other words, P_i can be uniquely determined given $\mathcal{T}$.

Example 3 Figure 3 illustrates the prefix-based partitioning for transaction $t_2 = \{x_1, x_2, x_9, x_{10}\}$ in the running example.

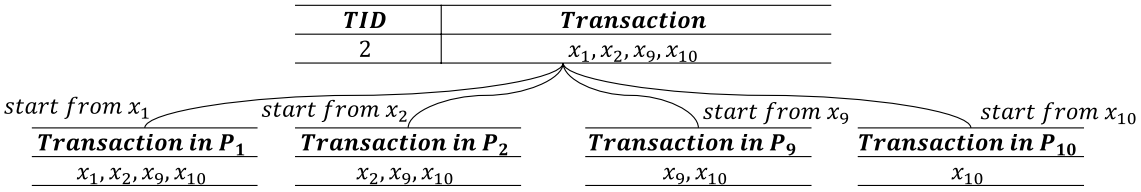


Fig. 3 The illustration of prefix-based transaction partitioning

TID in P_1	Transaction
1	x_1, x_2, x_9, x_{10}

TID in P_2	Transaction
1	x_2, x_6, x_{10}
2	x_2, x_9, x_{10}
3	x_2, x_{10}
4	$x_2, x_4, x_5, x_8, x_9, x_{10}$
5	x_2, x_9, x_{10}
6	$x_2, x_5, x_8, x_9, x_{10}$
7	$x_2, x_3, x_4, x_9, x_{10}$
8	x_2, x_9
9	x_2, x_4, x_{10}
10	x_2, x_8, x_9

TID in P_3	Transaction
1	x_3, x_4
2	x_3, x_4, x_9, x_{10}

TID in P_4	Transaction
1	x_4
2	$x_4, x_5, x_8, x_9, x_{10}$
3	x_4, x_9, x_{10}
4	x_4, x_{10}

TID in P_5	Transaction
1	x_5, x_8, x_9, x_{10}
2	x_5, x_8, x_9, x_{10}

TID in P_6	Transaction
1	x_6, x_{10}

TID in P_7	Transaction
1	x_7, x_{10}

TID in P_8	Transaction
1	x_8, x_9, x_{10}
2	x_8, x_9, x_{10}
3	x_8, x_9

TID in P_9	Transaction
1	x_9, x_{10}
2	x_9, x_{10}
3	x_9, x_{10}
4	x_9, x_{10}
5	x_9, x_{10}
6	x_9
7	x_9

TID in P_{10}	Transaction
1	x_{10}
2	x_{10}
3	x_{10}
4	x_{10}
5	x_{10}
6	x_{10}
7	x_{10}
8	x_{10}
9	x_{10}

Fig. 4 The illustration of prefix-based partitions in the running example

All prefix-based partitions are depicted in Fig. 4. There are 10 partitions since the running example has 10 items.

Theorem 1 *The prefix-based partitioning can generate complete and non-redundant itemsets.*

Proof Given the prefix-based partition $P_i (1 \le i \le d)$, the itemsets with the prefix item x_i can be generated in P_i according to the partitioning strategy. Let the possible itemsets generated in P_i be G_i, and any itemset possibly generated belongs to $\bigcup_{i=1}^{d} G_i$. Besides, $\forall P_i, P_j (1 \le i \ne j \le d)$, $G_i \cap G_j = \emptyset$, because they generate the itemsets with different prefix items. □

For any itemset X generated in P_i, *it is a mandatory requirement that* $x_i \in X$, because all transactions in P_i begin with x_i. Theorem 1 proves that the prefix-based partitioning can generate complete and non-redundant itemsets. The support of any itemset X, whose first item is x_i, can be computed by the transactions in P_i only. According to the analysis in Sect. 3.2, the number of the promising items when processing P_i can be fewer significantly compared with that when processing $\mathcal{T}$.

4.2 The Space Consumption in Direct Form

In this part, we first analyze the space consumption of the prefix-based partitioning. Given $\mathcal{T}$, let average transaction width (i.e. average item number in a transaction) be m, each item is represented as an integer value (4 bytes). $\forall t \in \mathcal{T}$ and $t = \{x_{1'}, x_{2'}, \ldots, x_{m'}\}$, $\forall 1 \le j \le m$, $(m - j + 1)$ items $\{x_{j'}, \ldots, x_{m'}\}$ are written into the partition $P_{j'}$. The total data size generated from t is $4 \times (1 + \ldots + m) = 4 \times \frac{m \times (m+1)}{2}$ bytes in the prefix-based partitioning, it is $\frac{m+1}{2}$ times larger compared with the size of t.

Transaction table usually can be classified as sparse datasets and dense datasets, in terms of average transaction width and the number of items [16].

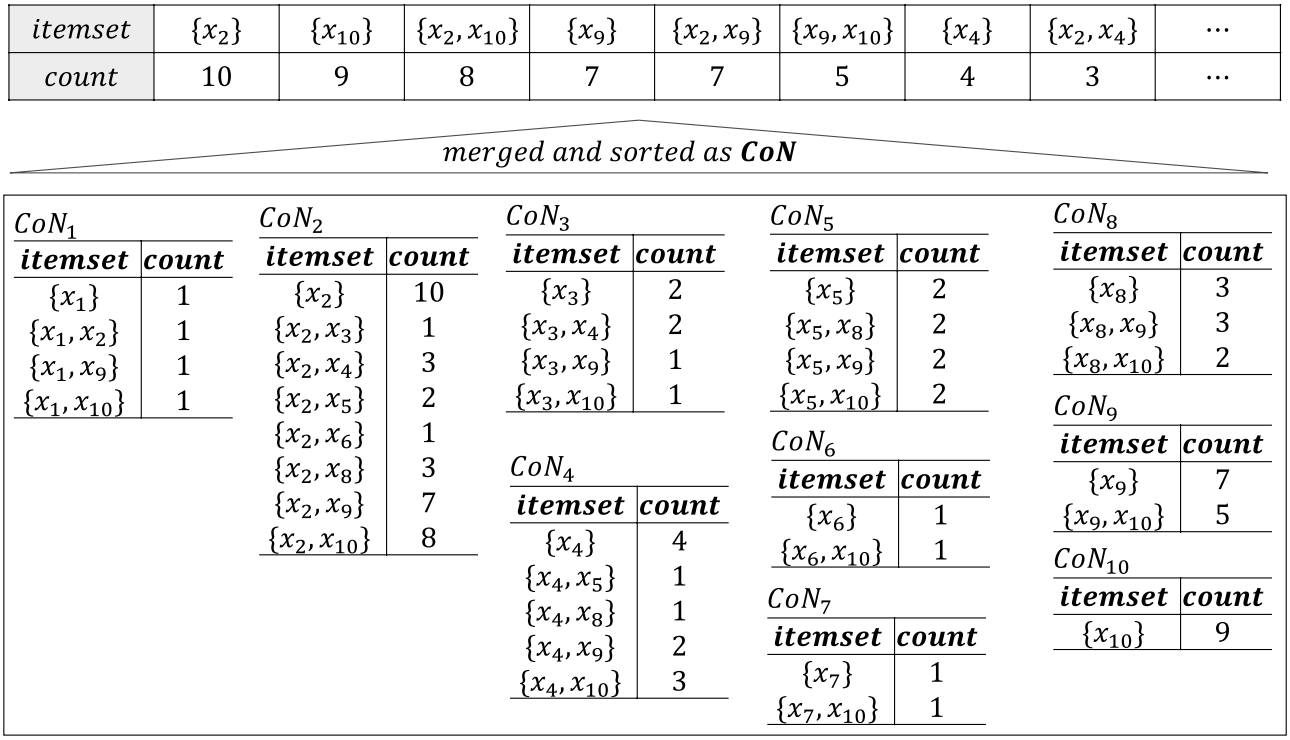


Fig. 5 The illustration of CoN structure in the running example

On *sparse dataset*, it usually has a small value of m and a large value of d. Many practical applications, including the most popular area *market basket analysis*, generate sparse data set. For example, the supermarket sells a variety of goods (thousands or even larger), but each customer usually purchases very few of them in a transaction. The average basket size of Walmart is 11,[1] chainstore data set and online retail data set have the average item counts of 7.23 and 4.37, respectively.[2] In the prefix-based partitioning, the several-fold larger space consumption of the original data will be incurred for the sparse datasets. In return, orders of magnitude speedup can be achieved for our solution compared with the existing algorithms. This can be treated as a classical implementation of the idea that sacrifices space to improve efficiency.

On *dense dataset*, there is a small value of d, and the value of m is relatively large. It is possible that the direct form of the prefix-based partitioning maintains one order of magnitude larger data than the original data. However, in this case, the value of d is small, some transactions in the prefix-based partitions are identical, especially when the values of possible items are small. *Duplicate merging* can be assisted in the prefix-based partitioning, the duplicate transactions can be merged to reduce the space.

Since our emphasis is to mine top-k frequent itemsets efficiently by the prefix-based partitioning, the direct form of the partitioning is utilized here to clarify the description.

4.3 Some Pre-constructed Structures

Given the prefix-based partition $P_i(1 \le i \le d)$, whose transactions share the same prefix item x_i, the ***c***o-***o***ccurrence ***n***umbers of the items in P_i with x_i are recorded in the structure CoN_i. CoN_i in the memory is maintained in the form of *hash table*. Let $CoN_i(x)$ be the co-occurrence number corresponding to the item x in CoN_i. $\forall t \in P_i$ and $t = \{x_i, y_2, \ldots, y_b\}(x_i \prec y_2 \prec \ldots \prec y_b)$, firstly $CoN_i(x_i)$ is increased by 1, and $\forall 2 \le j \le b$, $CoN_i(y_j)$ is increased by 1.

All elements in $CoN_1, \ldots, CoN_d$ are merged, let CoN(*itemset*, *count*) represent the merged structure. Specifically, for any element $(item = x, count = c)$ in $CoN_i(1 \le i \le d)$, if $(x = x_i)$, $(\{x_i\}, c)$ is added to CoN, otherwise $(\{x_i x\}, c)$ is added to CoN. The elements in CoN are sorted in the descending order of *count*. The structure CoN actually provides the supports of d 1-itemsets and $\binom{d}{2}$ 2-itemsets, which will be utilized for PTF algorithm. Note that any 1-itemsets or 2-itemsets which are not kept

[1] https://snapshot.numerator.com/retailer/walmart/.

[2] http://www.philippe-fournier-viger.com/spmf/.

explicitly in CoN have count 0, indicating that they do not occur in the transactions. Let $|CoN|$ denote the number of elements kept in CoN.

Example 4 Given 10 items in the running example, $CoN_1, \ldots, CoN_{10}$ are depicted in Fig. 5, which are merged and sorted in the descending order of count as CoN structure. In Fig. 5, only first 8 elements of CoN are listed, and other elements have count values no greater than 3.

5 PTF Algorithm

This section introduces PTF algorithm (***P***refix-partitioning-based ***T***op-k ***F***requent itemset mining) on massive data.

5.1 The Overview of the Algorithm

PTF processes the prefix-based partitions sequentially. The min heap for candidates and running minimum support $rmsup$, which are initialized by CoN structure, can be updated during its execution. By $rmsup$ and the maintained promising items, PTF can directly skip those partitions which cannot generate any top-k frequent itemsets. This reduces the I/O cost and the computation cost effectively. When all partitions are processed or skipped, PTF discovers the mined results. The main component of PTF is described in Fig. 6.

Algorithm 1 PTF algorithm

Input: The prefix-based partitions $P_1, \ldots, P_d$, the number k $(k > 0)$
Output: The top-k frequent itemsets
// The initialization of MH and $rmsup$
1: **if** $k \leq |CoN|$ **then**
2: Read the first k elements in CoN into MH
3: $rmsup \leftarrow MH.min$
4: **else**
5: Read all elements in CoN into MH
6: $rmsup \leftarrow 0$
// The maintenance of the promising items
7: **for** each element X in MH **do**
8: **if** $X = \{x_i\}$ **then**
9: insert x_i to AR_i
10: **if** $X = \{x_i x_j\}$ **then**
11: insert x_j to AR_i
// Processing the partitions sequentially
12: **for** each partition P_i **do**
13: **for** each item x in AR_i **do**
14: **if** $(x = x_i) \wedge (\sigma(x_i) \leq rmsup)$ **then**
15: Remove all elements in AR_i and **break** // goto Line 18
16: **if** $x = x_j (j > i)$ **then**
17: remove x when $\sigma(x_i x_j) \leq rmsup$
18: **if** $|AR_i| \leq 2$ **then**
19: **continue**
20: **else**
21: $ProcessSglPartition(P_i, AR_i, MH)$
22: **return** MH

5.2 Basic Implementation

During its execution, PTF uses a ***m***in ***h***eap MH of size k (initially empty) to maintain the k most frequent itemsets discovered so far. Let $MH.min$ be the kth largest support of the itemsets in MH. The pseudo-code of PTF is listed in Algorithm 1.

The initialization of rmsup. As listed in line 1 to line 6 of Algorithm 1, initially, PTF reads the first k elements in the structure CoN and inserts them into MH, since the elements in CoN are arranged in the descending order of their supports. If $|CoN| \geq k$, $rmsup = MH.min$, otherwise, $rmsup = 0$.

Note that all the itemsets in MH currently are 1-itemsets or 2-itemsets. Let $AR_1, AR_2, \ldots, AR_d$ correspond to d ***ar***rays. $\forall X \in MH$, if $X = \{x_i\}$ is a 1-itemset, x_i is inserted into AR_i, and if $X = \{x_i x_j\}$ is a 2-itemset, x_j is inserted into AR_i. The maintenance of the promising items is stated from line 7 to line 11 of Algorithm 1.

Example 5 The initialization of rmsup in the running example is illustrated in Fig. 7, and initially $rmsup = 3$ given $k = 8$. With current $rmsup$, the promising items for each partition are listed in Fig. 8.

After the initialization of $rmsup$, PTF deals with the partitions one by one (from line 12 to line 21 of Algorithm 1). $\forall 1 \leq i \leq d$, let P_i be the currently processed partition. PTF first iterates through the elements in AR_i. Due to anti-monotone property, $\sigma(x_i)$ is the maximum support that all itemsets generated in P_i can have. If $\sigma(x_i) \leq rmsup$, P_i cannot generate any top-k frequent itemsets, and there is no meaning of checking the remaining items in AR_i (line 14). Here, PTF clears all elements in AR_i and terminates the iteration (line 15). Otherwise, $\forall x_j \in AR_i - x_i$, if $\sigma(x_i x_j) \leq rmsup$, x_j is removed from AR_i (line 16 and line 17)). At the end of the iteration on AR_i, *the remaining elements in AR_i correspond to the promising items in P_i*. Actually, AR_i contains the $|AR_i|$ items in P_i with the largest supports. The items in AR_i are sorted in the specified order.

The case of skipping partition. Proved by Theorem 2, if $|AR_i| \leq 2$, PTF does not need to process P_i any more, it can skip P_i directly and proceed with the next partition (line 18 and line 19 of Algorithm 1).

Note that, in order to improve the readability of the paper, the theoretical contents in this section are slit from the algorithmic description, and the relevant theorems and their proofs are put in Sect. 5.5.

Example 6 As shown in Fig. 5, with the initialized $rmsup$ (3), PTF does not need to process $P_1, P_3, P_5, P_6, P_7, P_8$ since their maximum supports are no greater than current $rmsup$.

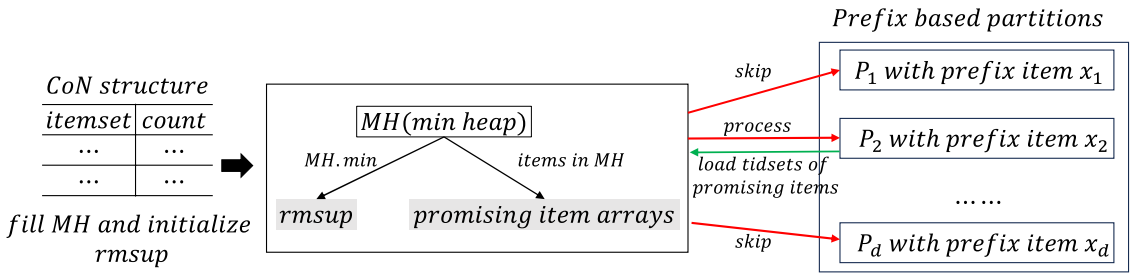


Fig. 6 The illustration of main component of PTF algorithm

The min heap ***MH*** *is implemented by an array, the first element is the one with the minimum support*

itemset	$\{x_2, x_4\}$	$\{x_4\}$	$\{x_9, x_{10}\}$	$\{x_2, x_9\}$	$\{x_9\}$	$\{x_2, x_{10}\}$	$\{x_{10}\}$	$\{x_2\}$
support	3	4	5	7	7	8	9	10

rmsup = 3 *initially*

Fig. 7 The illustration of initializing rmsup in the running example

The arrays of promising items

AR_1	AR_2	AR_3	AR_4	AR_5	AR_6	AR_7	AR_8	AR_9	AR_{10}
{}	$\{x_2, x_9, x_{10}\}$	{}	$\{x_4\}$	{}	{}	{}	{}	$\{x_9, x_{10}\}$	$\{x_{10}\}$

Fig. 8 The illustration of promising items after rmsup initialization in the running example

And PTF processes P_2, P_4, P_9, P_{10} sequentially. Firstly, P_2 is dealt with.

If there are at least three promising items in AR_i (i.e., $|AR_i| \geq 3$), PTF deals with P_i to discover the possible top-k frequent itemsets (line 21 of Algorithm 1). The pseudo-code of processing single partition is listed in Algorithm 2.

Since only the promising items in AR_i are useful for the mining process, PTF needs to retrieve the required data. Therefore, the transactions in P_i are kept in vertical storage mode. For an item x in P_i, the tid-set of x is a set of transaction identifiers of the transactions in P_i containing x. The tid-sets of the items (excluding x_i) are arranged *in the descending order of supports of the items*. The tid-set of x_i is not kept explicitly since all transactions in P_i contain x_i. Due to $x_i \in AR_i$, when processing P_i, PTF only retrieves *the tid-sets of the first* $(|AR_i| - 1)$ *items* in P_i. Here, the case of the items with the equal supports should be considered to guarantee that PTF obtains the tid-sets of items in AR_i.

Example 7 Figure 9 illustrates the vertical storage mode of the current P_2, the tid-sets of the items are arranged in the descending order of supports. Since the promising items of P_2 are x_2, x_9, x_{10} (Fig. 8), only the tid-sets of x_{10} and x_9 are retrieved.

PTF utilizes a ***h***ash ***t***able *HT* to maintain the tid-sets of the itemsets during the processing of P_i. At first, $(|AR_i| - 1)$ 2-itemsets, $x_i x_j$ $(x_j \in AR_i - x_i)$, and their tid-sets are kept in *HT* (line 2 and line 3 of Algorithm 2). Given the itemset X which is in *HT*, $HT(X)$ corresponds to the tid-sets of X. Besides, a priority ***q***u***e***ue *QE* is exploited to store the itemsets and explore them by the high-support-first principle (line 4 of Algorithm 2). The root element has the maximum support in *QE*. Let *QE.max* be the largest supports among itemsets in *QE*. At the beginning, $(|AR_i| - 1)$ 2-itemsets $x_i x_j$ $(x_j \in AR_i - x_i)$ are maintained in *QE*.

The overview of execution. From the itemsets in *QE*, PTF generates the longer itemsets and their corresponding tid-sets in the way of the *growth of the itemsets* described below until the *termination condition* is satisfied (from line 5 to line 14 of Algorithm 2).

$TID\ in\ P_2$	$Transaction$
1	x_2, x_6, x_{10}
2	x_2, x_9, x_{10}
3	x_2, x_{10}
4	$x_2, x_4, x_5, x_8, x_9, x_{10}$
5	x_2, x_9, x_{10}
6	$x_2, x_5, x_8, x_9, x_{10}$
7	$x_2, x_3, x_4, x_9, x_{10}$
8	x_2, x_9
9	x_2, x_4, x_{10}
10	x_2, x_8, x_9

vertical storage

$item\ in\ P_2$	$tidset$
x_2	$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$
x_{10}	$1, 2, 3, 4, 5, 6, 7, 9$
x_9	$2, 4, 5, 6, 7, 8, 10$
x_8	$4, 6, 10$
x_4	$4, 7, 9$
x_5	$4, 6$
x_3	7
x_6	1

arranged in the descending order of size of tidset

Fig. 9 The illustration of vertical storage mode of partition P_2. The tid-set of x_2 is not maintained explicitly

Algorithm 2 ProcessSglPartition

Input: Prefix-based partition P_i, promising item set AR_i, min heap MH
// Load the first $|AR_i| - 1$ items and their tidsets in P_i
1: Sort the elements in AR_i in the specified order
2: **for** $b \leftarrow 1$ to $|AR_i| - 1$ **do**
3: Load x_j and its corresponding tid-sets into HT ($HT.put(x_i x_j, tidset(x_j))$)
4: Insert $x_i x_j$ into max heap QE
// The termination condition
5: **while** $QE.max > rmsup$ **do**
6: Remove root element $X_{rt} = Xy_1$ from QE and y_1 is the ath element in AR_i.
7: **if** $|X_{rt}| \geq 3$ **then**
8: Update MH and $rmsup$
// Expand the current itemset
9: **foreach** $y_2 \in AR_i(a+1, \ldots, |AR_i|)$
10: **if** $HT(Xy_2) \neq null$ **then**
11: Generate Xy_1y_2 and its tid-sets by intersecting $HT(Xy_1)$ and $HT(Xy_2)$
12: **if** $\sigma(Xy_1y_2) > rmsup$ **then**
13: Put Xy_1y_2 and its tid-sets into HT
14: Insert Xy_2y_2 into QE
15: **return**

The growth of the itemsets. Every time PTF needs to explore more itemsets, the root element X_{rt} in QE is removed (line 6 of Algorithm 2). Within the loop from line 8 to line 14, we have $\sigma(X_{rt}) > rmsup$. Let X_{rt} be in the form of Xy_1, where X is the prefix items of X_{rt} without the last item y_1. If $|X_{rt}| \geq 3$, PTF first updates MH by removing its root element and inserting X_{rt} into it (line 8 of Algorithm 2). Then more itemsets will be generated by expanding X_{rt} with each of the items in AR_i after y_1 (from line 9 to line 14 of Algorithm 2). Let y_1 be the ath item in AR_i. $\forall y_2 \in AR_i(a+1, \ldots, |AR_i|)$, PTF first checks whether the itemset Xy_2 is kept in HT.

- If $HT(Xy_2) \neq null$, the tid-set $HT(Xy_1)$ of Xy_1 and the tid-set $HT(Xy_2)$ of Xy_2 are intersected to generate the tid-set of the itemset Xy_1y_2 (line 11). If $\sigma(Xy_1y_2) > rmsup$, PTF keeps Xy_1y_2 and its tid-sets in HT, and also inserts it in QE for the further exploration (line 13 and line 14 of Algorithm 2).
- Otherwise, $HT(Xy_2) = \text{null}$, as proved in Theorem 3, $\sigma(Xy_2) \leq rmsup$, PTF proceeds with the next item in AR_i to expand X_{rt}.

In a word, PTF explores the itemsets in the enumeration space by *hybrid search strategy*, combining depth-first search and level-wise search.

Termination condition. If the support of X_{rt} is no greater than $rmsup$, i.e., $QE.max \leq rmsup$, as proved in Theorem 4, the processing of P_i can terminate and PTF continues to deal with the next partition. The condition is checked in line 5 of Algorithm 2.

(1) *At first, the hashtable* **HT** *keeps tidsets of* $\{x_2, x_{10}\}$ *and* $\{x_2, x_9\}$.

key	**value**
$\{x_2, x_{10}\}$	1, 2, 3, 4, 5, 6, 7, 9
$\{x_2, x_9\}$	2, 4, 5, 6, 7, 8, 10

(2) *The max heap* **QE** *is filled with* $\{x_2, x_{10}\}$ *and* $\{x_2, x_9\}$

itemset	$\{x_2, x_{10}\}$	$\{x_2, x_9\}$
support	8	7

(3) *Remove the root* $\{x_2, x_{10}\}$ *of QE.*

itemset	$\{x_2, x_9\}$
support	7

$X_{rt} = \{x_2, x_{10}\}, \sigma(X_{rt}) = 8$

(4) *Remove the root* $\{x_2, x_9\}$ *of QE, and expand it.*

itemset	$X_{rt} = \{x_2, x_9\}, \sigma(X_{rt}) = 7$
support	X_{rt} *is expaned by adding* x_{10}*, since* $x_9 \prec x_{10}$ *and* $\{x_2, x_{10}\}$ *is maintained in HT*

(5) *Compute tidset of* $\{x_2, x_9, x_{10}\}$ *and add to QE.*

key	**value**
$\{x_2, x_{10}\}$	1, 2, 3, 4, 5, 6, 7, 9
$\{x_2, x_9\}$	2, 4, 5, 6, 7, 8, 10
$\{x_2, x_9, x_{10}\}$	2, 4, 5, 6, 7

itemset	$\{x_2, x_9, x_{10}\}$
support	5

(6) *Remove the root* $\{x_2, x_9, x_{10}\}$ *of QE, and updte MH, current rmsup* = 4.

itemset	$X_{rt} = \{x_2, x_9, x_{10}\}, \sigma(X_{rt}) = 5$
support	

Fig. 10 The illustration of vertical storage mode of partition P_2. The tid-set of x_2 is not maintained explicitly

Example 8 The processing of P_2 is depicted in Fig. 10. The tid-sets of $\{x_2, x_9\}$ and $\{x_2, x_{10}\}$ are maintained in hash table *HT*. The two 2-itemsets are filled into max heap *QE*. Current root element of *QE* is $\{x_2, x_{10}\}$, which is removed first. Since its size is less than 3 and x_{10} is the last promising item of P_2, $\{x_2, x_{10}\}$ cannot update *MH* and add larger itemsets to *QE*. Next, the current root element $\{x_2, x_9\}$ of *QE* is removed. $\{x_2, x_9\}$ can be extended by adding x_{10}. The tid-set of $\{x_2, x_9, x_{10}\}$ is generated by intersecting the tidsets of $\{x_2, x_9\}$ and $\{x_2, x_{10}\}$, and the support of $\{x_2, x_9, x_{10}\}$ is 5. Given the current rmsup 3, $\{x_2, x_9, x_{10}\}$ is added into *QE* and its tid-set is kept in *HT*. The current root $\{x_2, x_9, x_{10}\}$ of *QE* is removed, its size is larger than 2 and support is greater than *rmsup*. Min heap *MH* is updated by removing it root $\{x_2, x_4\}$ and inserting $\{x_2, x_9, x_{10}\}$. The current *rmsup* is updated to 4. $\{x_2, x_9, x_{10}\}$ cannot be extended since x_{10} is the last promising item of P_2. Then *QE* is empty and the processing of P_2 is complete. For the remaining partitions $P_3, \ldots, P_{10}$, the maximum support of possibly generated itemsets is less than 4, PTF directly skips them and reports the mined results.

5.3 Improvement 1: Hybrid Vertical Storage Mode

At dealing with P_i, AR_i corresponds to the $|AR_i|$ items with the largest supports, and PTF needs to retrieve the tid-sets of the first $(|AR_i| - 1)$ items into memory. Although this discards the irrelevant items directly, the required data volume is still considerable since the loaded items are the $(|AR_i| - 1)$ most frequent items (except x_i). Improvement 1 introduces hybrid vertical storage mode for the prefix-based partitions to reduce the involved I/O cost in the execution.

5.3.1 Storage Mode

$\forall 1 \le i \le d$, let n_i be the number of transactions in the partition P_i. Since the support of any itemset beginning with x_i can be computed by P_i only, $\forall 1 \le a \le n_i$, the ath transaction in P_i is assigned with a unique identifier (*tid*) a. The transactions in P_i are stored in the vertical format. That is, each item $x_j (i + 1 \le j \le d)$ is associated with a *tid-set*, representing the tids of the transactions containing it. Since all transactions in P_i share the prefix item x_i, we do not keep its tid-set. *The tid-sets for the items (excluding* x_i*) in* P_i *are arranged in the descending order of the supports of the items.*

For the item x_j, its tid-set can be maintained in one of the three storage mode (*tid-list*, *dif-list*, *bit-vector*) depending on three size values, $SZ_{tidlist} = 4 \times |L(x_j)|$, $SZ_{diflist} = 4 \times \big(n_i - |L(x_j)|\big)$ and $SZ_{bv} = \lceil \frac{n_i}{8} \rceil$.

- The first value is the size of tid-list $L(x_j)$, which is a list to keep the tids of the transactions containing x_j.
- The second value is the size of the dif-list $D(x_j)$, i.e. $\{1, 2, \ldots, n_i\} - L(x_j)$, which is a list to keep the tids of the transactions not containing x_j.
- The third value is the size of the bit-vector $BV(x_j)$ of n_i bits. The ath bit in $BV(x_j)$ is 1 if the ath transaction contains x_j, otherwise, the ath bit in $BV(x_j)$ is 0.

Hybrid storage mode selects the corresponding storage mode with the minimum size. Storage schema of tid-set of x_j is $\big(x_j, flag, DS, L(x_j)|D(x_j)|BV(x_j)\big)$, where x_j is the item, $flag(-1, 0, 1)$ indicates the tid-set is stored as tid-list $L(x_j)$,

item in P_i	***tidset***
x_i	1, … , 1000
…	…
x_{i+j_1}	1,2,3, … , 997
…	…
x_{i+j_2}	1,2,3, … , 500
…	…
x_{i+j_3}	1,2,3,4
…	…

The size of any bit vector is 125 *bytes.*

$L(x_{i+j_1}) = \{1,2,\dots,997\}$, 3988 *bytes.*

$D(x_{i+j_1}) = \{998,999,1000\}$, 12 *bytes.*

$L(x_{i+j_2}) = \{1,2,\dots,500\}$, 2000 *bytes.*

$D(x_{i+j_2}) = \{501,502,\dots,1000\}$, 2000 *bytes.*

$L(x_{i+j_3}) = \{1,2,3,4\}$, 16 *bytes.*

$D(x_{i+j_3}) = \{5,6,\dots,1000\}$, 3984 *bytes.*

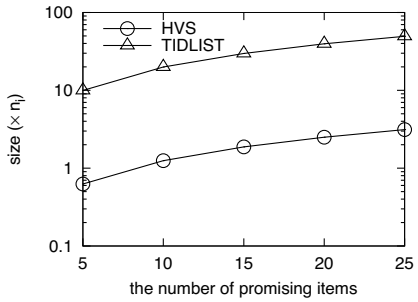
hybrid vertical storage

…	x_{i+j_1}	0	12	$D(x_{i+j_1})$	…	x_{i+j_2}	1	125	$BV(x_{i+j_2})$	…	x_{i+j_3}	−1	16	$L(x_{i+j_1})$	…

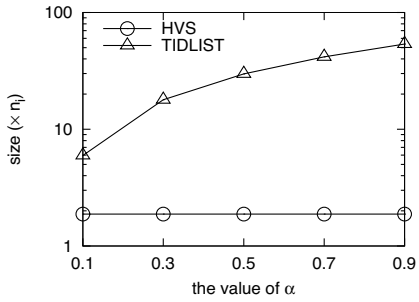
$D(.): flag = 0, dif\ list,$ $BV(.): flag = 1, bit\ vector,$ $L(.): flag = -1, tid\ list.$

Fig. 11 The illustration of hybrid vertical storage

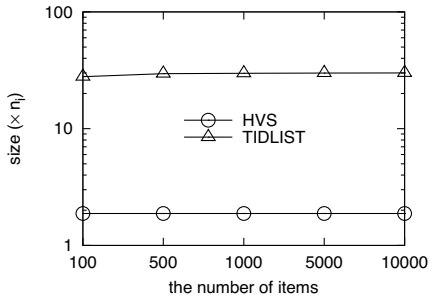
Fig. 12 The space comparison between hybrid vertical storage mode and complete tid-list format



(a)$\alpha = 0.5$ and $(d-i) = 1000$



(b)pitemnum $= 15$ and $(d-i) = 1000$



(c)pitemnum $= 15$ and $\alpha = 0.5$

dif-list $D(x_j)$ or bit-vector $BV(x_j)$, and DS is ***d***ata ***s***ize (in bytes) of tid-set in the given format.

Example 9 Figure 11 illustrates an example of hybrid vertical storage for partition P_i, which has 1000 transactions. For item x_{i+j_1} whose tid-set contains 997 tids, $D(x_{i+j_1})$ occupies the smallest space. For item x_{i+j_2} whose tid-set includes 500 tids, $BV(x_{i+j_2})$ is the structure with the smaller space compared with other two forms. For item x_{i+j_3} whose tid-set keeps 4 tids, the direct tid-list form is the best choice.

Next we analyze the effect of hybrid vertical storage mode. Given a partition P_i, there are $(d-i)$ tid-sets for the possible items $x_{i+1},\dots,x_d$. For the sake of discussion, suppose that the supports of $\sigma(x_i x_j)$ $(i+1 \le j \le d)$ are uniformly distributed over the range $[0, \alpha \times n_i]$ and $\sigma(x_i x_{i+1}) > \sigma(x_i x_{i+2}) > \dots > \sigma(x_i x_d)$, where $\alpha (0 < \alpha \le 1)$ is the parameter depending on the characteristics of the table. $\forall (i+1) \le j \le d$, $\sigma(x_i x_j) = \frac{d-j+1}{d-i} \times \alpha \times n_i$, $SZ_{tidlist} = 4 \times \frac{d-j+1}{d-i} \times \alpha \times n_i$, $SZ_{diflist} = 4 \times (1 - \frac{d-j+1}{d-i} \times \alpha) \times n_i$ and $SZ_{bv} = \lceil \frac{n_i}{8} \rceil$. If

$SZ_{bv} \leq \min(SZ_{tidlist}, SZ_{diflist})$, i.e. $\frac{1}{32} \leq \frac{d-j+1}{d-i} \times \alpha \leq \frac{31}{32}$, the tid-set of x_j is kept in bit-vector. If $SZ_{tidlist} \leq \min(SZ_{diflist}, SZ_{bv})$, i.e., $\frac{d-j+1}{d-i} \times \alpha \leq \frac{1}{32}$, the tid-set of x_j is kept in tid-list. If $SZ_{diflist} \leq \min(SZ_{tidlist}, SZ_{bv})$, i.e., $\frac{d-j+1}{d-i} \times \alpha \geq \frac{31}{32}$, the tid-set of x_j is kept in dif-list. According to the computation formula above, Fig. 12 compares the space consumption for loading the required tid-sets between hybrid vertical storage mode and complete tid-list format from the aspects of varying promising item numbers, α values and $(d-i)$ values, respectively. As depicted in Fig. 12, by utilizing hybrid vertical storage mode, PTF can reduce the I/O cost of loading the tid-sets of promising items by an order of magnitude, which also is verified in our experiments. It should be noted that, the results in Fig. 1 are not targeted to the specific type of data set, but they illustrate the general effect of the proposed storage mode under the above assumption. The actual effect in the real-life data sets of different characteristics is provided in the experimental section.

5.3.2 The Intersection of the Tid-Sets

Since the tid-sets of the items in P_i are kept in three data formats: tid-lists, dif-lists and bit-vectors, the intersection of the tid-sets should be processed in six different cases.

Case 1: *the intersection of tid-list and tid-list.*

The intersection in case 1 is simple. Given two itemsets Xy_1 and Xy_2, their tid-lists are $L(Xy_1)$ and $L(Xy_2)$, respectively. Since tid-list maintains the tids in the ascending order, the tid-list of Xy_1y_2 is $L(Xy_1) \cap L(Xy_2)$ which can be computed directly by binary merging. $\sigma(Xy_1y_2) = |L(Xy_1y_2)|$.

Case 2: *the intersection of dif-list and dif-list.*

Given two itemsets Xy_1 and Xy_2, their dif-lists are $D(Xy_1)$ and $D(Xy_2)$, respectively. $D(Xy_1) = \{1, 2, \ldots, n_i\} - L(Xy_1)$ and $D(Xy_2) = \{1, 2, \ldots, n_i\} - L(Xy_2)$. $\forall a_1 \in D(Xy_1)$, the transaction with tid a_1 does not contain Xy_1, and $\forall a_2 \in D(Xy_2)$, the transaction with tid a_2 does not contain Xy_2. Obviously, $\forall a \in D(Xy_1) \cup D(Xy_2)$, the transaction with tid a does not contain Xy_1 or Xy_2. The intersection of the tid-sets of Xy_1 and Xy_2 is $L(Xy_1) \cap L(Xy_2)$, which corresponds to the transactions contains both Xy_1 and Xy_2, we have:

$$L(Xy_1y_2) = L(Xy_1) \cap L(Xy_2) = \{1, 2, \ldots, n_i\} - D(Xy_1) \cup D(Xy_2)$$
$$D(Xy_1y_2) = \{1, 2, \ldots, n_i\} - L(Xy_1y_2) = D(Xy_1) \cup D(Xy_2)$$

$\sigma(Xy_1y_2) = n_i - |D(Xy_1y_2)|$. If $|D(Xy_1y_2)| > \frac{n_i}{2}$, the tid-set of Xy_1y_2 is transformed into the tid-list format.

Case 3: *the intersection of bit-vector and bit-vector.*

Given two itemsets Xy_1 and Xy_2, bit-vector of Xy_1 is $BV(Xy_1)$ and the bit-vector of Xy_2 is $BV(Xy_2)$. The intersection of the tid-sets of Xy_1 and Xy_2 is: $BV(Xy_1y_2) = BV(Xy_1)\&BV(Xy_2)$, where & is the bitwise AND operator. $\sigma(Xy_1y_2)$ is the number of bit 1 in $BV(Xy_1y_2)$. If $\lceil \frac{n_i}{8} \rceil > 4 \times \sigma(Xy_1y_2)$, $BV(Xy_1y_2)$ is converted into the tid-list form.

The number of bit 1 in bit-vector. A simple way to compute the number of bit 1 in $BV(Xy_1y_2)$ is checking whether the ath bit of $BV(Xy_1y_2)$ is 1 ($\forall 1 \leq a \leq n_i$) and accumulating the counts. But, a better option is table-driven and counting the number of bit 1 in each byte directly. A hash table with 256 key-value pairs is utilized to map each byte value to the number of bit 1 in it, which can be pre-computed. Each byte value in the bit-vector can be retrieved sequentially and its number of bit 1 can be obtained directly by the hash table.

Case 4: *the intersection of tid-list and dif-list.*

Given two itemsets Xy_1 and Xy_2, tid-list of Xy_1 is $L(Xy_1)$ and the dif-list of Xy_2 is $D(Xy_2)$. The intersection of the tid-sets of Xy_1 and Xy_2 is:

$$\begin{aligned} L(Xy_1y_2) &= L(Xy_1) \cap L(Xy_2) = L(Xy_1) \cap (\{1, 2, \ldots, n_i\} - D(Xy_2)) \\ &= L(Xy_1) - L(Xy_1) \cap D(Xy_2) = L(Xy_1) - D(Xy_2) \end{aligned}$$

The last equation can be proved in Theorem 5. The formula above is very intuitive. $L(Xy_1)$ keeps the tids of the transactions containing Xy_1 and $D(Xy_2)$ maintains the tids of the transactions not containing Xy_2. Of course, $L(Xy_1) - D(Xy_2)$ represents the tids of the transactions containing both Xy_1 and Xy_2, i.e., $L(Xy_1) \cap L(Xy_2)$. $\sigma(Xy_1y_2) = |L(Xy_1y_2)|$.

Case 5: *the intersection of tid-list and bit-vector.*

Given two itemsets Xy_1 and Xy_2, tid-list of Xy_1 is $L(Xy_1)$ and the bit-vector of Xy_2 is $BV(Xy_2)$. The intersection of the tid-sets of Xy_1 and Xy_2 is kept as a tid-list $L(Xy_1y_2)$. $\forall a \in L(Xy_1)$, if the ath bit of $BV(Xy_2)$ is 1, a is inserted into $L(Xy_1y_2)$. $\sigma(Xy_1y_2) = |L(Xy_1y_2)|$.

Case 6: *the intersection of dif-list and bit-vector.*

Given two itemsets Xy_1 and Xy_2, dif-list of Xy_1 is $D(Xy_1)$ and the bit-vector of Xy_2 is $BV(Xy_2)$. The intersection of the tid-sets of Xy_1 and Xy_2 is kept as a bit-vector $BV(Xy_1y_2)$, which firstly is a copy of $BV(Xy_2)$. $\forall a \in D(Xy_1)$, the ath transaction does not contain Xy_1, the ath bit of $BV(Xy_1y_2)$ is set to 0. $\sigma(Xy_1y_2)$ is the number of bit 1 in $BV(Xy_1y_2)$. If $\lceil \frac{n_i}{8} \rceil > 4 \times \sigma(Xy_1y_2)$, $BV(Xy_1y_2)$ is converted into the tid-list form.

5.4 Improvement 2: Candidate Pruning

For the root element $X_{rt} = Xy_1$ removed from QE, if $\sigma(Xy_1) > rmsup$, PTF expands it with the items in AR_i after y_1. Let y_1 be the ath item in AR_i. According to the basic implementation, $\forall y_2 \in AR_i(a+1, \ldots, |AR_i|)$, if the itemset Xy_2 is kept in HT, PTF generates the itemsets Xy_1y_2 by intersecting the tid-sets of Xy_1 and Xy_2. However, the intersection of two tid-sets is relatively expensive, and we want to reduce the computation cost effectively.

In the initialization of *rmsup*, if a 1-itemset or 2-itemset belongs to top-k frequent itemsets, it has been maintained in

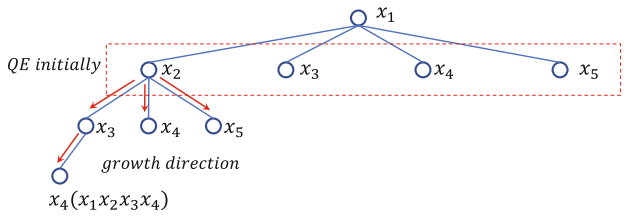


Fig. 13 The illustration of hybrid search strategy. For some partition, there are five promising items x_1, x_2, x_3, x_4, x_5. Suppose that $x_1x_2x_3x_4$ is the next itemset to be explored

MH. Before processing P_i, AR_i maintains the $|AR_i|$ items with the largest supports, satisfying the condition: $\forall x_j \in AR_i - x_i$, $\sigma(x_ix_j) > rmsup$.

Timeliness pruning. To merge Xy_1 and Xy_2, the former is the root element of *QE*, and the latter is found to be kept in *HT*. Of course, we have $\sigma(Xy_1) > rmsup$ currently. For Xy_2, it is kept in *HT* because $\sigma(Xy_2)$ is greater than *rmsup* of the time when Xy_2 is generated. Ever since Xy_2 is generated, the value of *rmsup* could be increased and be no less than $\sigma(Xy_2)$ now. Therefore, before intersecting the tid-sets of Xy_1 and Xy_2, PTF first checks whether $\sigma(Xy_2)$ is greater than the current *rmsup*. If $\sigma(Xy_2) \leq rmsup$, there is no need to merge Xy_1 and Xy_2.

Last-item pruning. For Xy_2 which is not pruned by timeliness pruning, we have $\sigma(Xy_2) > rmsup$. If Xy_1y_2 is a top-*k* frequent itemset, then all its subsets should belong to FI_k. Because $\sigma(Xy_1) > rmsup$ and $\sigma(Xy_2) > rmsup$, the support of any subset of Xy_1 and Xy_2 is greater than *rmsup* due to downward closure property. In another word, if the support of any itemset in $\mathcal{P}^-(Xy_1y_2) - \mathcal{P}(Xy_1) - \mathcal{P}(Xy_2)$, where $\mathcal{P}^-(Xy_1y_2)$ represents the powerset of Xy_1y_2 excluding empty set and Xy_1y_2 itself, is no greater than *rmsup*, we do not need to merge Xy_1 and Xy_2. However, due to the hybrid search strategy PTF adopts, the operation of checking all itemsets in $\mathcal{P}^-(Xy_1y_2) - \mathcal{P}(Xy_1) - \mathcal{P}(Xy_2)$ could go wrong, discarding the top-*k* frequent itemsets incorrectly. As illustrated in Fig. 13, without loss of generality, let the promising items be x_1, x_2, x_3, x_4, x_5 when processing partition P_1, and the hybrid search strategy can be depicted by a growing tree. The root of the tree is x_1 since all transactions begin with x_1. According to the execution of PTF dealing with single partition, the 2-itemsets x_1x_2, x_1x_3, x_1x_4 and x_1x_5 are maintained in max heap *QE* initially. This corresponds to one level extension for the tree. Let x_1x_2 be the root element of *QE* then, which is removed from *QE* and extended by the following promising items $\{x_3, x_4, x_5\}$, and the subtree rooted at x_2 extends one level with three leaves. We do not consider the pruning operation in this step temporarily, since x_2x_3, x_2x_4, x_2x_5 may be kept in *MH* in the initialization of *rmsup*. Let $x_1x_2x_3$ be the next root element of *QE* to be selected and we want to expand it by x_4, i.e., merging $x_1x_2x_3$ and $x_1x_2x_4$. Suppose now we perform the pruning by using of the constituent subsets. Because $x_1x_3x_4 \in \mathcal{P}^-(x_1x_2x_3x_4) - \mathcal{P}(x_1x_2x_3) - \mathcal{P}(x_1x_2x_4)$ and $x_1x_3x_4$ is not be generated yet, we may prune $x_1x_2x_3x_4$ by mistake. Therefore, when merging Xy_1 and Xy_2, PTF only checks whether y_1y_2 is contained in *MH*, the *k* itemsets with the largest supports discovered so far. This is because, in the initialization of *rmsup*, any 2-itemset, if it belongs to FI_k, is kept in *MH* already. If y_1y_2 is not contained in *MH*, $\sigma(y_1y_2) \leq rmsup$, we do not need to merge Xy_1 and Xy_2.

5.5 Analysis

This part includes the theorems and their proofs used in this section.

Theorem 2 *Given a prefix-based partition P_i, if $|AR_i| \leq 2$, the processing of P_i cannot generate any new top-k frequent itemsets.*

Proof All transactions in P_i share the same prefix item x_i, the maximum support of the itemsets that can be generated in P_i is no larger than that of $\{x_i\}$.

- If $|AR_i| = 0$, the support of $\{x_i\}$ is no larger than *rmsup*, supports of any itemsets generated in P_i are no larger than *rmsup*.
- If $|AR_i| = 1$ or 2, only 1-itemsets or 2-itemsets are needed to be checked in P_i. Considering that in the initialization of *rmsup*, *MH* has maintained the 1-itemsets and 2-itemsets that are possible to be top-*k* frequent itemsets. PTF still does not need to deal with P_i.

In short, if $|AR_i| \leq 2$, the processing of P_i cannot generate any new top-*k* frequent itemsets. □

Theorem 3 *When expanding the itemset Xy_1 with another item y_2 where $y_1 \prec y_2$, if $HT(Xy_2) = null$, $\sigma(Xy_2) \leq rmsup$.*

Proof First, note that AR_i maintains the promising items in P_i. $\forall x_j \in AR_i - x_i$, the 2-itemset x_ix_j and its tid-set are maintained in *HT*, and x_ix_j is kept in *QE*.

Before the growth of the itemsets, all itemsets in *QE* are 2-itemsets, and the first root element removed is 2-itemsets also. Let $Xy_1(X = x_i)$ be the current root element and y_2 be the item satisfying $y_1 \prec y_2$, if Xy_2 is not kept in *HT*, it means $\sigma(Xy_2) \leq rmsup$ because *HT* maintains all the 2-itemsets with prefix item x_i whose supports are greater than *rmsup*. After the expansion for Xy_1, *HT* and *QE* maintains the 3-itemsets beginning with x_iy_1 whose supports are greater than *rmsup*.

During the execution, let the current root element be Xy_1, where $|Xy_1| \geq 3$. According to the growth strategy, Xy_1 is generated by expanding X with y_1 in the previous execution. In that process, not only y_1, all the promising items in AR_i

Table 2 Parameter settings in experiments

Parameter	Used values
Transaction number (10^7) (syn)	1–100
Result number (syn)	1000–10,000
The number of items (syn)	500–2500
The average transaction width (syn)	10–30
Result size (real-life sparse data set)	1000–10,000
Result size (real-life dense data set)	1000–5000

Table 3 The characteristic of the real-life data sets

Dataset	#Trans	#Items	Avg.Tr.Width	Type
Chainstore	1,112,949	46,086	7.23	Sparse
OnlineRetail	540,455	2603	4.37	Sparse
Accidents	340,183	468	33.8	Dense
Connect	67,557	129	43	Dense
Pumsb	49,046	2113	74	Dense
Susy	5,000,000	179	19	Dense

after the last item of X are used to expand X also. *HT* and *QE* only keep the expanded itemsets whose supports are greater than *rmsup*. Let y_2 be the item to be expanded with Xy_1. If Xy_2 is not kept in *HT*, $\sigma(Xy_2) \leq rmsup$. □

Theorem 4 *If $QE.max \leq rmsup$, there is no need to process P_i further and the processing of P_i can terminate.*

Proof According to the property of max heap, X_{rt} has the maximum support among the itemsets in *QE*. For any itemset X_1 that is not generated so far, it is definitely the superset of some itemset X_2 in *QE* in terms of the growth strategy, and $\sigma(X_1) \leq \sigma(X_2)$ due to the anti-monotone property. Obviously, $\sigma(X_1) \leq \sigma(X_{rt})$. Therefore, if $QE.max \leq rmsup$, it means that the supports of all the itemsets in *QE* and the itemsets undiscovered so far are no greater than *rmsup*, they cannot be the top-k frequent itemsets. The processing of P_i can terminate. □

Theorem 5 *Given $L(Xy_1)$ and $D(Xy_2)$, we have $L(Xy_1) - L(Xy_1) \cap D(Xy_2) = L(Xy_1) - D(Xy_2)$.*

Proof (1) $\forall a \in L(Xy_1)$ and $a \in D(Xy_2)$, $a \in L(Xy_1) \cap D(Xy_2)$, we have $a \notin L(Xy_1) - L(Xy_1) \cap D(Xy_2)$ and $a \notin L(Xy_1) - D(Xy_2)$. (2) $\forall a \in L(Xy_1)$ and $a \notin D(Xy_2)$, we have $a \notin L(Xy_1) \cap D(Xy_2)$, $a \in L(Xy_1) - L(Xy_1) \cap D(Xy_2)$ and $a \in L(Xy_1) - D(Xy_2)$. (3) $\forall a \notin L(Xy_1)$, no matter whether $a \in D(Xy_2)$ or not, $a \notin L(Xy_1) - L(Xy_1) \cap D(Xy_2)$ and $a \notin L(Xy_1) - D(Xy_2)$. All possibilities are included and the equation is satisfied. □

6 Performance Evaluation

In this section, PTF is implemented in Java with jdk-15_windows-x64 to evaluate its performance. The experiments are executed on DELL Precision T3431 Workstation (Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz (8 cores) + 32 G memory + 64bit windows 10).

The performance of PTF is evaluated against the latest top-k FIM algorithms, FP-tree-based CRM [45] and level-wise-based MTK [15]. The performances of other algorithms [14] are reported to be inferior to the two algorithms. *In order to evaluate the performance better, the initialization of rmsup, which is proposed in PTF, is also adopted in CRM and MTK*. For CRM, a higher initial *rmsup* makes a much smaller FP-tree and a better efficiency. For MTK, the initialization *saves the first scan* in initial step, which reduces the I/O cost and computation cost. In the experiments, the performance is evaluated in synthetic data sets and real-life data sets. The used parameter settings are listed in Table 2. The parameter δ of MTK is set to 3 on synthetic data sets and to 2 on real-life data sets, the best choice of δ as stated in [15]. For the top-rank-k FIM algorithm, we do not evaluate it in the experiments. The reason is that, (1) its definition is not exactly same as the top-k FIM, and multiple itemsets can have the same rank, (2) its prefix-tree-based execution mode makes it hold the same performance issue as CRM, which will be illustrated in the experiments.

Synthetic data sets. The synthetic data sets are generated by the data generator in [5], in which the number of maximal potential large itemsets is set to be double that of the number of items, and the average size of maximal potentially frequent itemset is set to 6. On synthetic data sets, the performance of PTF is evaluated in terms of four aspects: transaction number (n), result number (k), the number of total items (d) and the average transaction width (m).

Real-life data sets. The six real-life data sets, Chainstore, OnlineRetail, Accidents, Connect, Pumsb and Susy are selected from the SPMF data mining library [22]. They hold different characteristics in several aspects and are used to evaluate the performance of PTF on real-life data more comprehensively. SPMF provides real-life data sets commonly used in the related research fields. Although the data sets available in SPMF are of small and medium size, they reflect the data characteristic in the real world. Table 3 shows the characteristics of the used real-life data sets, consisting of sparse type and dense type. The selected real-life data sets include the largest dense data set (Susy) and the largest sparse customer transaction data set (Chainstore) in SPMF. Some data sets of small size are also selected to compare PTF with the state-of-the-art algorithms, because the devised PTF focuses on massive data and its performance on small-size data sets is also intriguing.

Table 4 The space comparison of hybrid vertical storage, direct form, CRM and MTK on real-life data sets, $r_{hvs} = \frac{SIZE_{hybridvertical}}{SIZE_{transTable}}$ and $r_{drt} = \frac{SIZE_{directform}}{SIZE_{transTable}}$, $r_{crm|mtk} = \frac{SIZE_{transTable}}{SIZE_{transTable}}$, CRM and MTK are executed on the original transaction table, their space consumption is equal to the size of transaction table

ratio	Chain	Online	Accid	Conn	Pum	Susy
r_{hvs}	16.94	0.55	3.26	1.25	5.86	1.21
r_{drt}	9.31	3.11	18	22.47	37.99	10.25
$r_{crm\|mtk}$	1	1	1	1	1	1

The space consumption of prefix-based partitioning. The hybrid vertical storage selects the storage formats of the tid-sets of the items with the minimum sizes. On synthetic data sets, given $m = 15$, the prefix-based partitions in hybrid vertical storage mode incurs about 5 times larger space consumption than the original transaction table. Relatively, the prefix-based partitions in the direct form need about 9 times larger space consumption. On real-life data sets, the space consumption comparison is listed in Table 4. Normally, hybrid vertical storage can help reduce the space consumption effectively. On Chainstore data set, hybrid vertical storage mode generates a larger space consumption compared with the direct form, because this data set has many items and the tid-set of each item is stored in direct form with the extra data of 12 bytes. According to the characteristics of the available real data sets [22], the number of the total items in the vast majority of cases is not too high, and the space consumption of hybrid vertical storage mode can stay within a reasonable range. For the cases that the number of items is much high and the average transaction width is relatively small, the overhead of extra 12 bytes for each tid-set is obviously significant, and the space consumption of hybrid vertical storage mode may be even larger than that of direct form. *Note that, the comparison results mentioned above actually include the space consumption of CRM and MTK*. Since the two algorithms process the original data directly, the space consumption is equal to the size of original transaction table. The comparison results of hybrid vertical storage and direct form with transaction table are just the comparison of space consumption of PTF, CRM and MTK. In this case, PTF can maintain only a small number of the most frequent items rather than all items in the prefix-based partitions. As verified in the experiments, only a small fraction of the items are promising items when PTF processes the partitions.

6.1 The Effect of Transaction Number

Given $k = 3000$, $d = 1500$ and $m = 15$, experiment 1 evaluates the performance of PTF with varying transaction numbers. The execution times of PTF, MTK and CRM are shown in Fig. 14a. Note that we only report the experimental results of CRM at $n = 10^7$ in the experiments of synthetic data sets, because the constructed FP-tree exceeds the memory limit at the larger values of n. As illustrated in Fig. 14a, PTF runs 255.948 times faster than MTK averagely. The execution time of CRM (110.105 s) at $n = 10^7$ is very close to that of MTK (145.275 s). The speedup ratio of PTF over MTK increases with a greater value of n, from 41.842 at $n = 10^7$ to 477.172 at $n = 100 \times 10^7$.

The significant performance advantage comes from the less I/O cost, the fewer candidates and the more efficient computation. As depicted in Fig. 14, PTF only involves 73.102 times less I/O cost than MTK. The I/O cost of CRM at $n = 10^7$ is equal to that of MTK. In experiment 1, MTK only needs to perform two full table scans, indicating that PTF only retrieves less than $\frac{1}{36}$ data of the original transaction table. The candidate numbers of PTF and MTK are reported in Fig. 14c. The candidate number of MTK is the total number of candidates during its δ-stair search, and the candidate number of PTF is the total number of itemsets explored actually in vertical mining. It is reported that PTF has 2.318 times fewer candidates than MTK.

The effect of hybrid vertical storage is illustrated in Fig. 14d, which is compared with the size of the involved tid-sets in tid-list format. By hybrid vertical storage, the size of the retrieved data in PTF decreases by 10.739 times. The effect of candidate pruning is reported in Fig. 14e. The pruning ratio is computed by the number of pruned candidates over the sum of the pruned candidates and the explored candidates actually before the termination condition is satisfied. The pruning ratio is not significant, because the supports of all itemsets in QE initially are greater than $rmsup$. Each expanded itemset, only if its support exceeds $rmsup$, will be kept in QE for the further expansion. The numbers of the promising items for the partitions are depicted in Fig. 14f. For any partition, if the number of the promising items is no greater than 2, PTF skips the partition directly. Given $d = 1500$, the overwhelming majority of the partitions are skipped, and the processing of the rest of the partitions only involves much small number (≤ 20) of promising items in experiment 1. Comparatively, given the $rmsup$ value after initialization by CoN, a majority of the items are promising items (more than 1200 in experiment 1) when computing on the whole table.

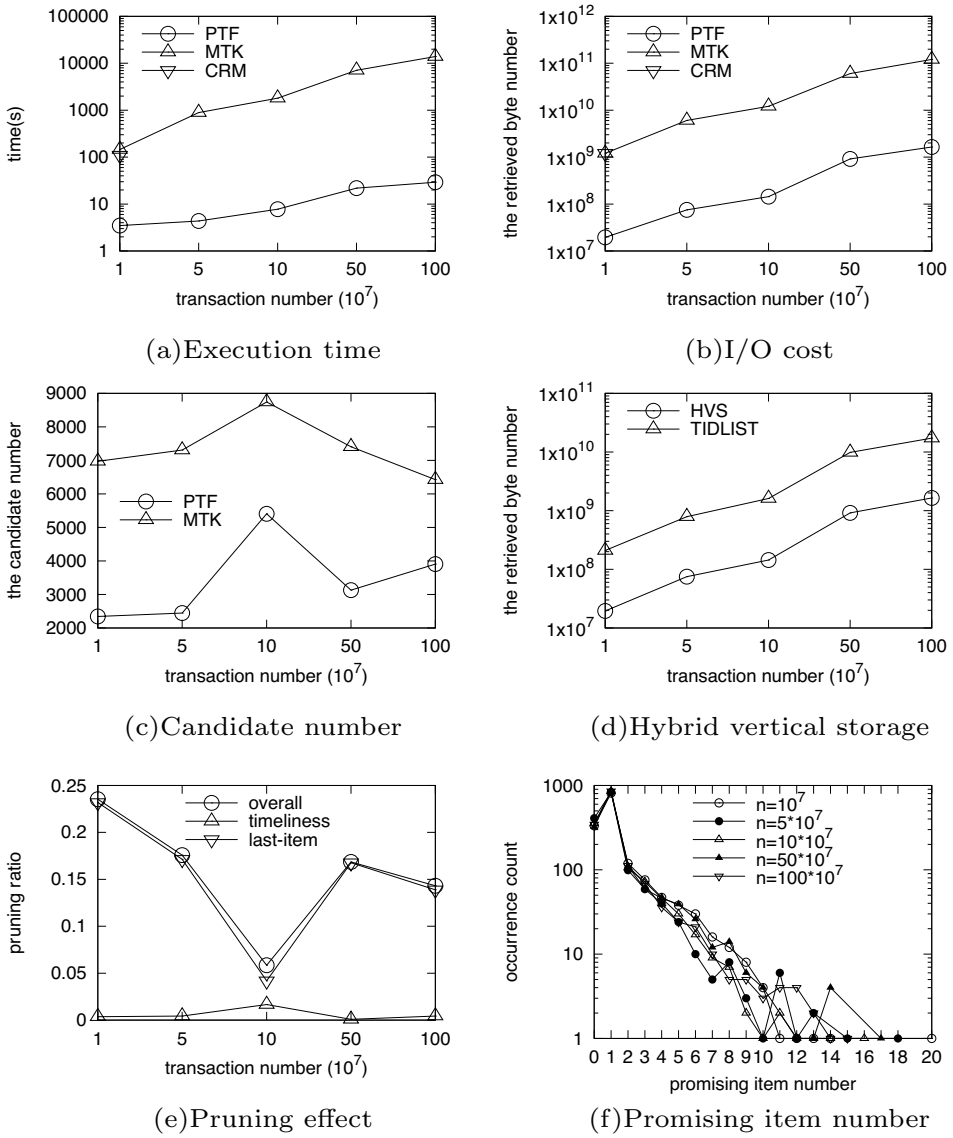


Fig. 14 The effect of transaction number

6.2 The Effect of Result Number

Given $n = 10 \times 10^7$, $d = 1500$ and $m = 15$, experiment 2 evaluates the performance of PTF with varying result numbers. A larger value of k returns more frequent itemsets, and it leads to a smaller value of *rmsup* naturally. The execution times of PTF and MTK should increase with a larger value of k. This also is reflected in Fig. 15a, which shows that PTF runs 653.728 times faster than MTK. Relatively speaking, the rising trend of execution time of MTK is faster than that of PTF. At $k = 10000$, the speedup ratio of PTF over MTK reaches to three orders of magnitude. For PTF, the quick increasing of execution time from $k = 1000$ to $k = 3000$ can be explained that, when $k = 1000$, the initialized *rmsup* by *CoN* is very high that only 23 partitions are processed actually. The similar variation trends are also reflected in the I/O cost (Fig. 15b) and the candidate number (Fig. 15c). On average, PTF involves 122.331 times less I/O cost than MTK, which performs one full table scan at $k = 1000$ and two full table scans at other values of k. PTF also maintains 2.319 fewer candidates than MTK. The effect of hybrid vertical storage is depicted in Fig. 15d, which shows that PTF in hybrid vertical storage involves 10.96 times less I/O cost than PTF in the direct tid-list storage. The effect of candidate pruning is reported in Fig. 15e. The numbers of the promising items for the partitions are illustrated in Fig. 15f, indicating that more partitions can be skipped directly at a smaller value of k.

6.3 The Effect of Item Number

Given $n = 10 \times 10^7$, $k = 3000$ and $m = 15$, experiment 3 evaluates the performance of PTF with varying item

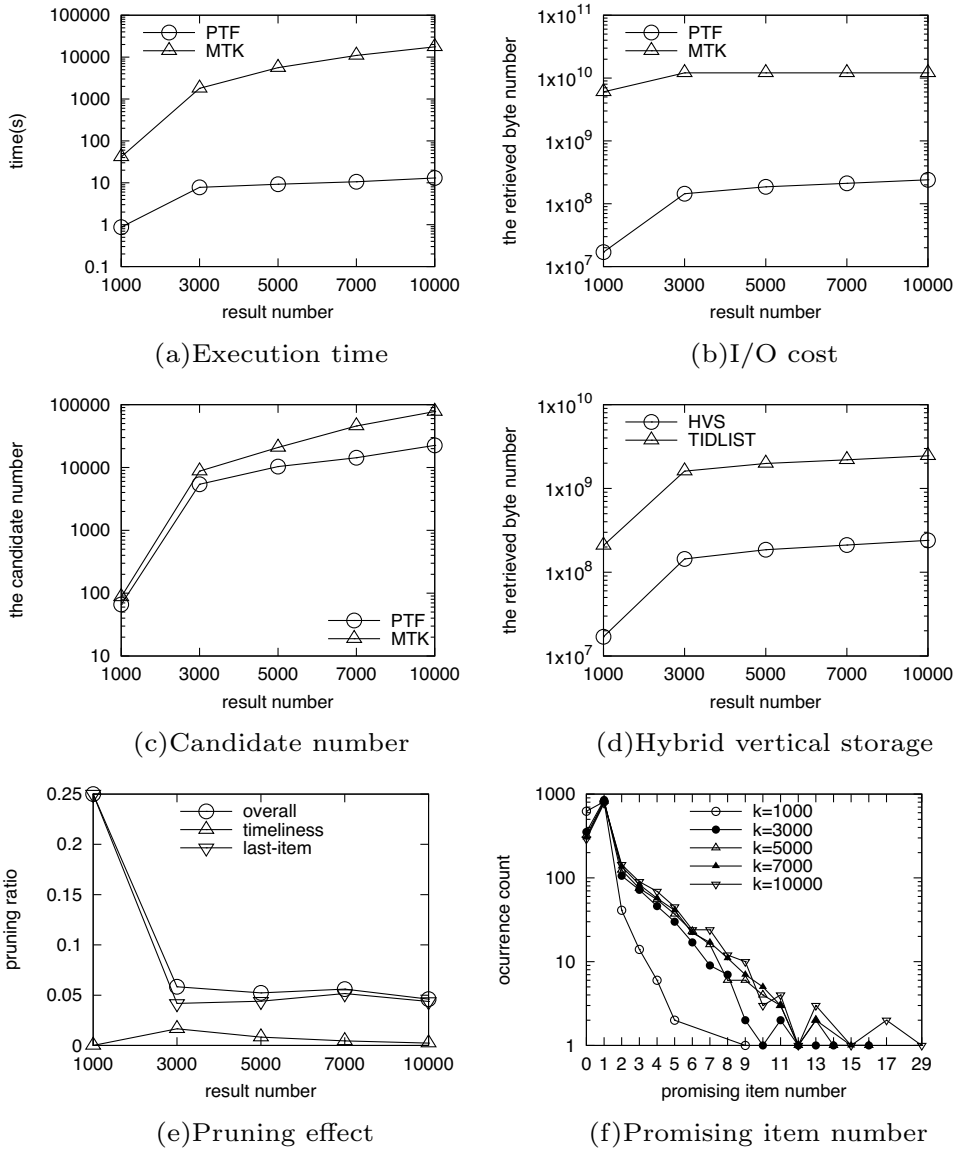


Fig. 15 The effect of result number

numbers. When the value of d rises from 500 to 2500, a fixed average transaction width means the synthetic data set becomes increasingly sparse. Therefore, the execution time and the candidate number are considered to be smaller with a greater value of d, which are reflected in Fig. 16a, c, respectively. In experiment 3, PTF runs 231.395 times faster than MTK and maintains 2.289 times fewer candidates than MTK.

With a greater value of d, the I/O cost of PTF keeps decreasing, as shown in Fig. 16b. Comparatively, the I/O cost of MTK first decreases from $d = 500$ to $d = 1000$, then remains unchanged in experiment 3. For PTF, the decline trend of I/O cost is due to the fewer promising items involved with a greater value of d, which is also depicted in Fig. 16f. For MTK, except for $d = 500$, it needs to perform two full table scans.

PTF in the hybrid vertical storage reduces the retrieved data volume by 10.376 times compared with that in the direct tid-list format, as illustrated in Fig. 16d. The effect of candidate pruning is reported in Fig. 16e.

6.4 The Effect of Average Transaction Width

Given $n = 10 \times 10^7$, $k = 3000$ and $d = 1500$, experiment 4 evaluates the performance of PTF with varying average transaction widths. As illustrated in Fig. 17a, with a greater value of m, the execution times of MTK and PTF both show the overall trend for increasing, and PTF runs 281.738 times faster than MTK. For one thing, the greater value of m makes the transaction table of a larger size. For another, the cost of counting supports will be higher with the greater value of m. As depicted in Fig. 17b, PTF involves 64.552 times

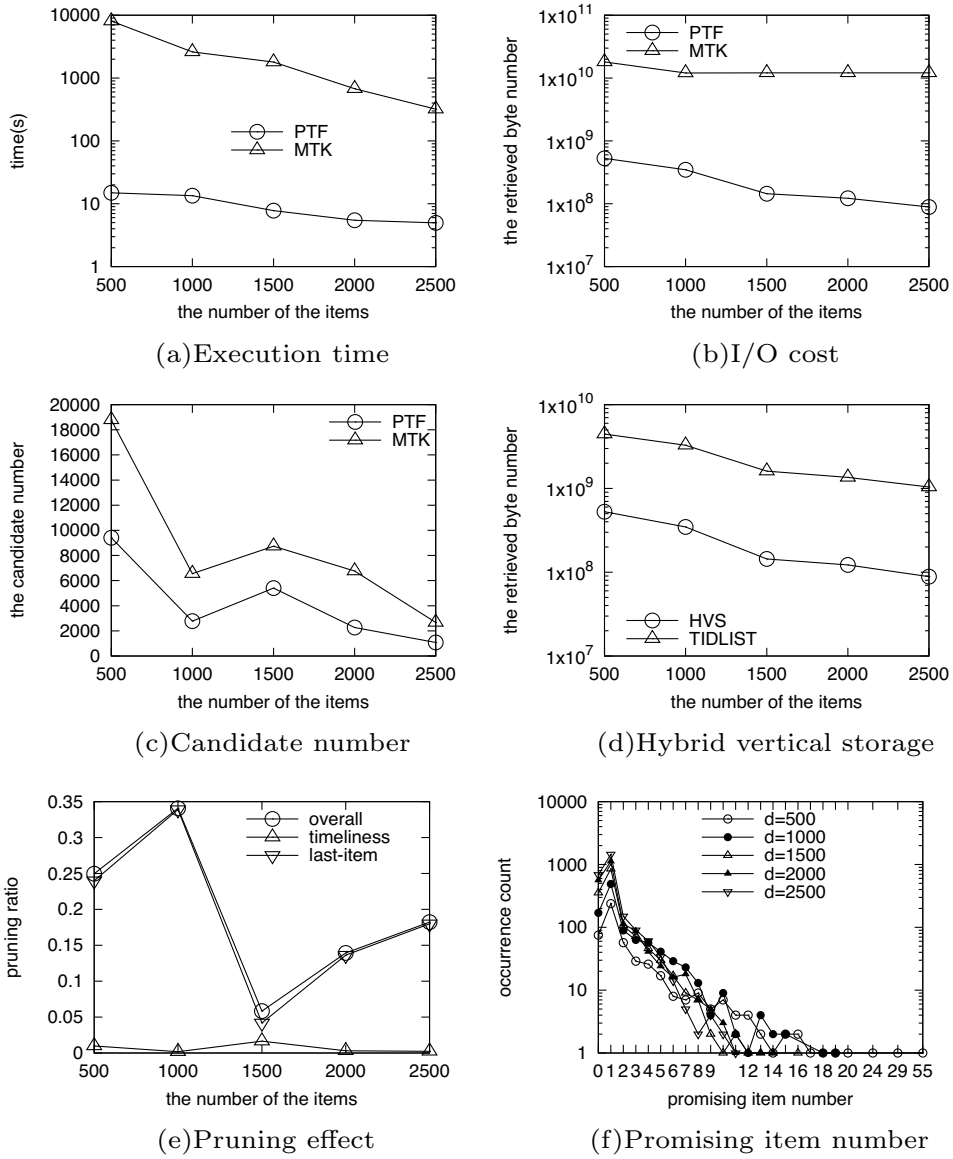


Fig. 16 The effect of item number

less I/O cost than MTK. Although the candidate numbers of PTF and MTK exhibit obvious variation, we can still find a relatively steady trend in overall given the other unchanged parameters. As shown in Fig. 17c, PTF maintains 2.606 times fewer candidates than MTK. As shown in Fig. 17d, the hybrid vertical storage helps PTF reduces I/O cost by an order of magnitude. The effect of candidate pruning is depicted in Fig. 17e. The numbers of the promising items are reported in Fig. 17f. Although there are 1500 items, PTF only involves a much small number of promising items when processing the partitions.

6.5 The Real-Life Data Sets

In experiment 5, the performance of PTF is evaluated on real-life data sets with varying result numbers. Since the used six real-life data sets are of small and medium size, the performance of FP-tree-based CRM is reported in experiment 5 also.

On the sparse data set, it normally has the relatively high number of items and short average transaction width. Therefore, MTK should perform well on sparse data sets of small and medium size, which is verified in Figs. 18a and 19a. MTK even is the best algorithm on Chainstore data when the result number is less than 10,000. CRM also makes good enough performance, and performs best on OnlineRetail data. The relatively small size of data does not liberate the full potential of PTF. For PTF, it has to retrieve the involved prefix-based partitions (349 partitions on chainstore data and 467 partitions on OnlineRetail data at $k = 10000$ as shown in Figs. 18d and 19d), the time to locate the partitions can be amortized on massive data, but the effect is more significant

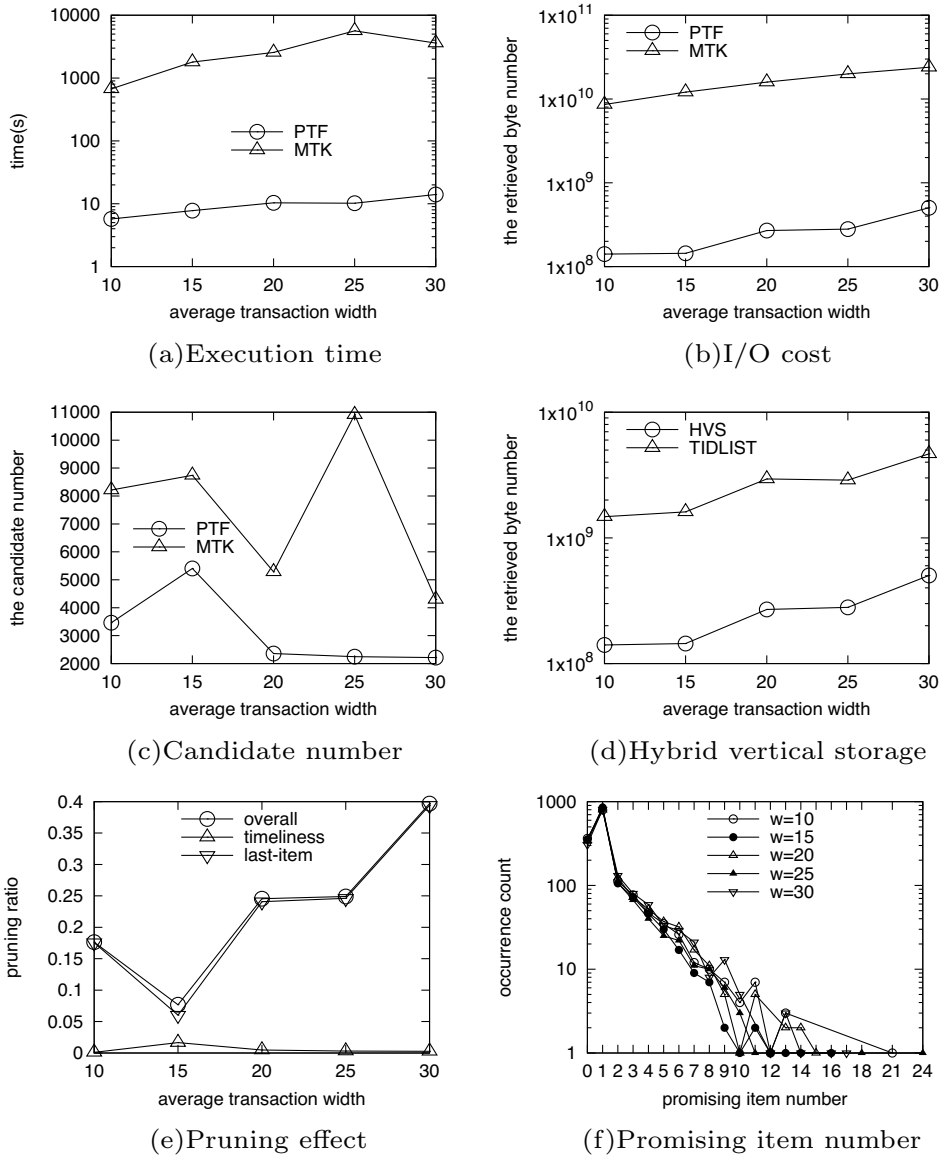
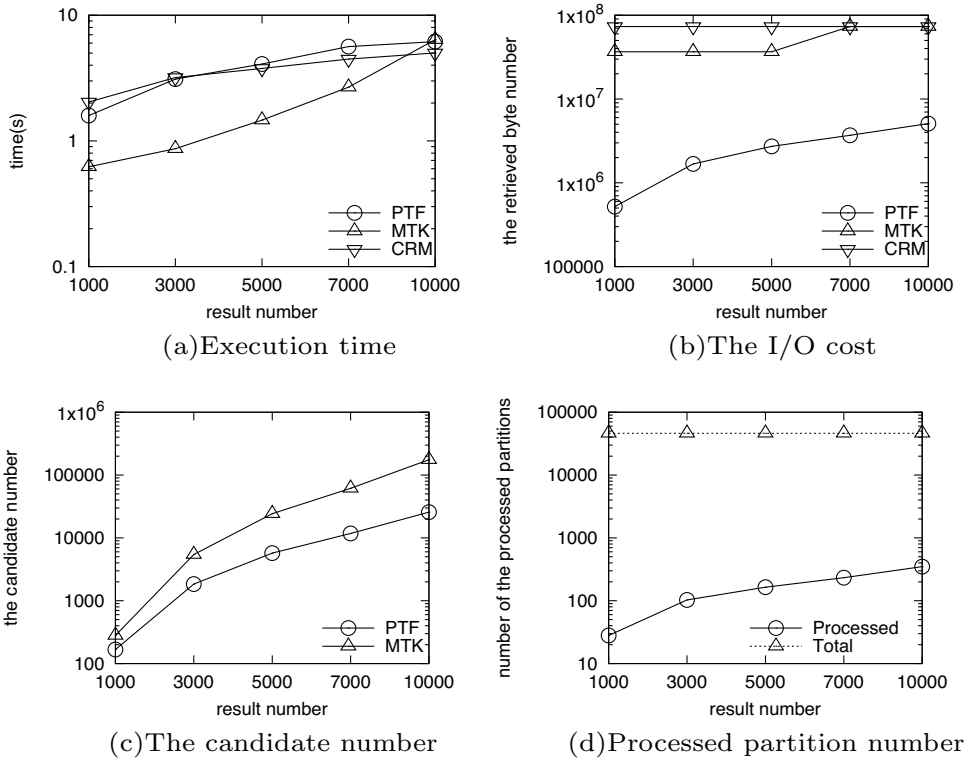


Fig. 17 The effect of transaction width

on small and medium-sized data. However, the performance of PTF still is acceptable and can be comparable with the other two algorithms. As shown in Figs. 18b and 19b, PTF involves an order of magnitude less I/O cost than MTK and CRM. Since CRM needs two full table scans and MTK requires one or two full table scans, PTF only retrieves a small fraction of the table. As depicted in Figs. 18c and 19c, PTF maintains 4.177 times fewer candidates and 3.186 times fewer candidates than MTK on chainstore data and OnlineRetail data, respectively.

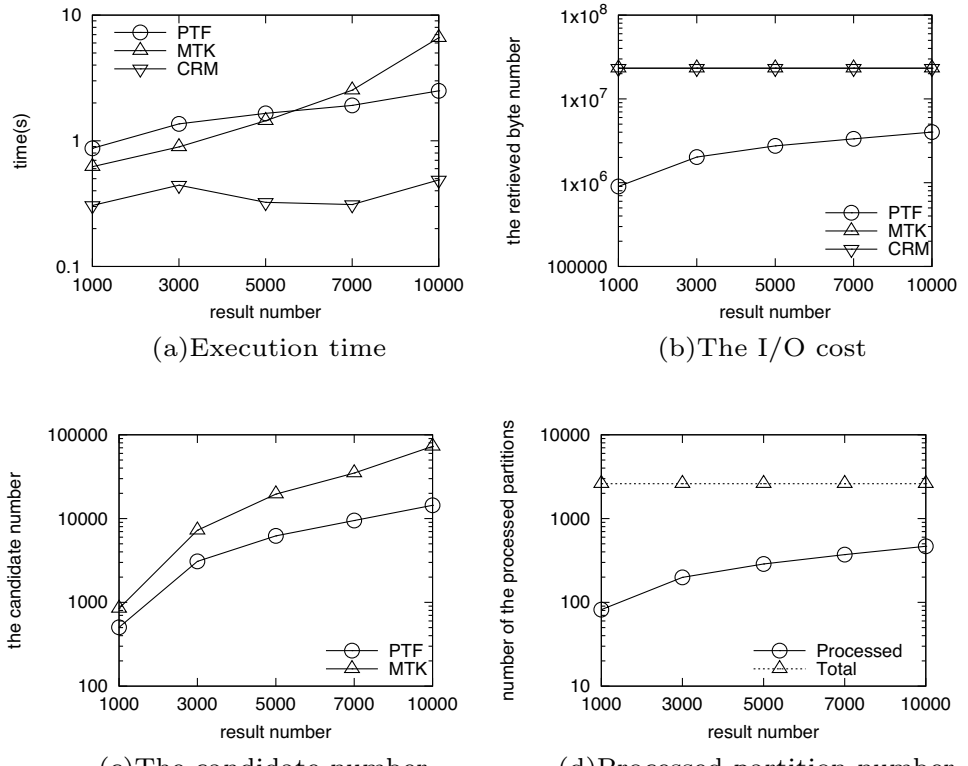
On the dense data set, it often has the relatively small number of items and long average transaction width. The performance of MTK is much poorer on dense data sets than that on sparse data sets. As shown in Figs. 20a, 21a, 22a and 23a, MTK is orders of magnitude slower than CRM and PTF, this can be explained that MTK has to maintain a large number of candidates as verified in Figs. 20c, 21c, 22c and 23c and the several full table scans for support counting. On Connect data and Pumsb data, CRM always shows the best performance, but the execution time of PTF is close to that of CRM. On Accidents data and Susy data, PTF even performs better than CRM in most cases. One overall advantage of PTF is that, it only involves a small fraction of table. As shown in Figs. 20b, 21b, 22b and 23b, the I/O cost of PTF is one order of magnitude less than that of the other algorithms. For one thing, much less data are retrieved, and for another, the computation cost for counting support is reduced correspondingly. The number of the processed partitions is depicted in Figs. 20d, 21d, 22d and 23d, a very small fraction of partitions have to be processed on dense data sets.



(a)Execution time (b)The I/O cost

(c)The candidate number (d)Processed partition number

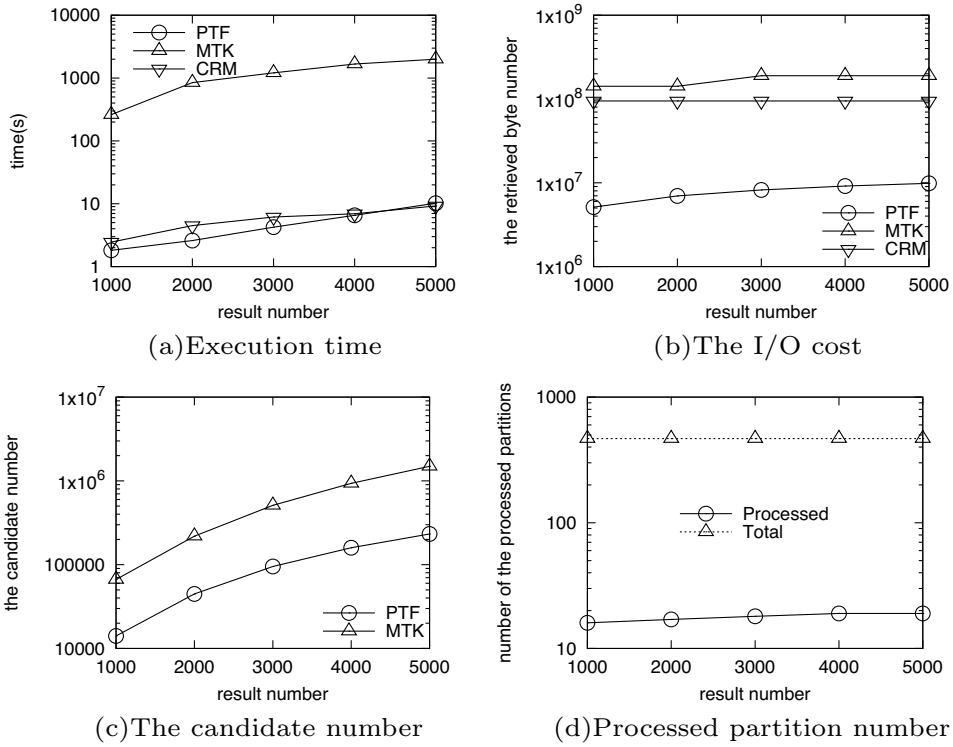
Fig. 18 The first group of experiment 5 (Chainstore, sparse data)



(a)Execution time (b)The I/O cost

(c)The candidate number (d)Processed partition number

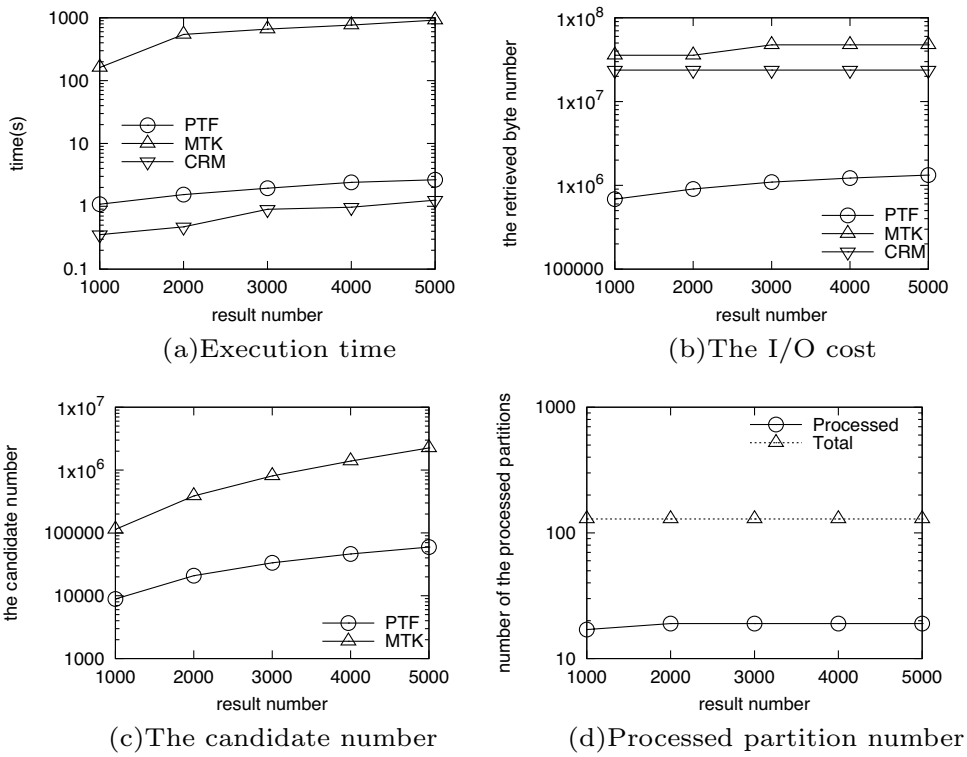
Fig. 19 The second group of experiment 5 (OnlineRetail, sparse data)



(a)Execution time (b)The I/O cost

(c)The candidate number (d)Processed partition number

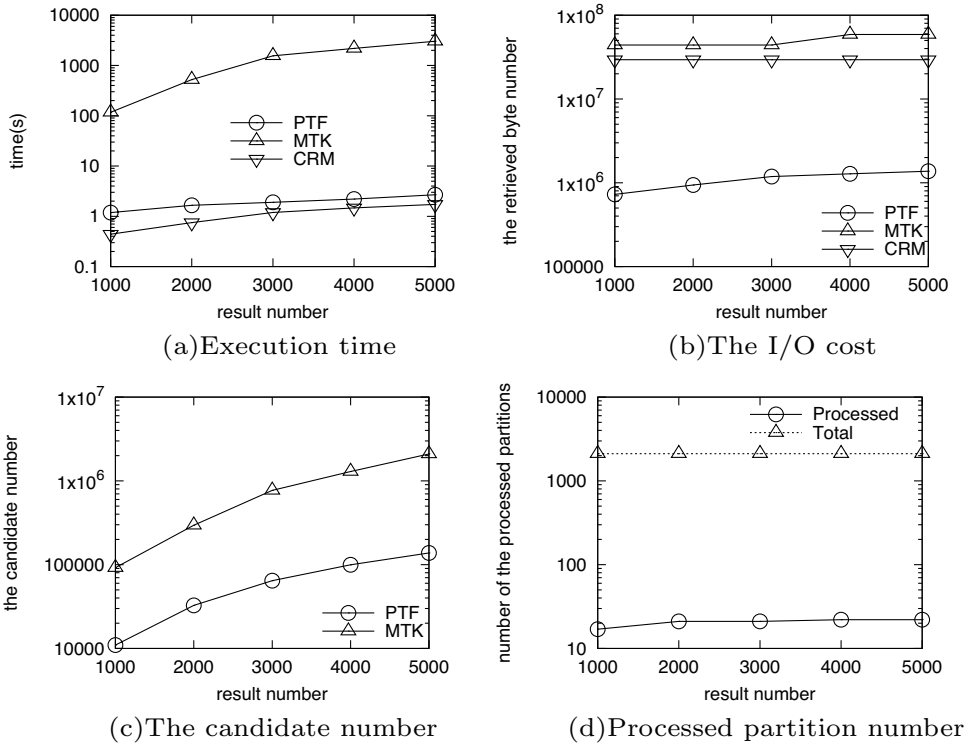
Fig. 20 The third group of experiment 5 (Accidents, dense data)



(a)Execution time (b)The I/O cost

(c)The candidate number (d)Processed partition number

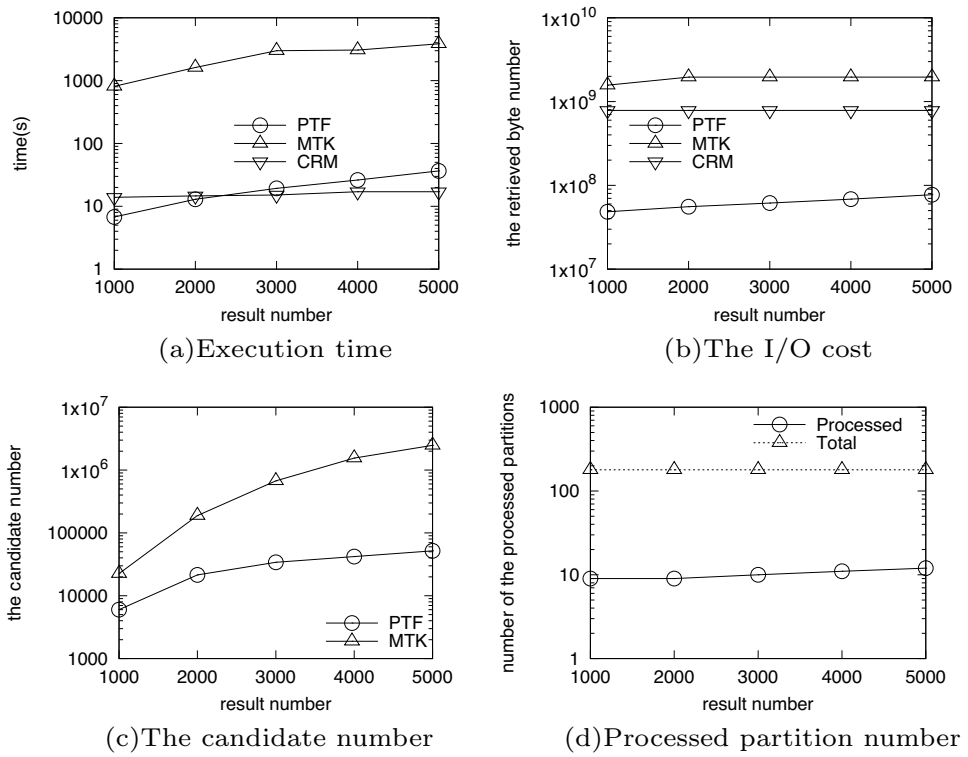
Fig. 21 The fourth group of experiment 5 (Connect, dense data)



(a)Execution time (b)The I/O cost

(c)The candidate number (d)Processed partition number

Fig. 22 The fifth group of experiment 5 (Pumsb, dense data)



(a)Execution time (b)The I/O cost

(c)The candidate number (d)Processed partition number

Fig. 23 The sixth group of experiment 5 (Susy, dense data)

6.6 Discussion

Through the experimental evaluation demonstrated above, we can understand the performance advantage of PTF more comprehensively.

On massive data, all transactions cannot be maintained entirely in main memory. The performance advantage of PTF is much significant over the state-of-the-art algorithms. On massive data, FP-growth-based CRM algorithm even cannot initialize the required FP-tree. For MTK, its level-wise execution makes multiple-pass scan on the transaction table. On massive data, the I/O cost of MTK is very high. Comparatively, PTF utilizes the prefix-based partitioning, hybrid vertical storage mode and effective pruning strategies. It shows a much high efficiency on massive data. PTF can achieve up to three orders of magnitude speedup compared with MTK.

Although PTF focuses on dealing with massive data, we still evaluate its performance on six real-life data sets of small and medium size. For one thing, the performance of PTF should be evaluated against real-life data. For another, it is still expected to assess PTF more comprehensively. By and large, compared with state-of-the-art algorithms, PTF on data sets of small and medium size does not perform well as it is executed on massive data. For example, PTF, MTK and CRM have similar performance on sparse data sets of small and medium size. And on dense data set, although PTF performs much better than MTK, its execution time is very close to CRM. This can be elaborated from two aspects. Firstly, the prefix-partitioning, helping PTF acquire a high execution efficiency on massive data, affects the execution of PTF on data sets of small and medium size to some extent. The costs of locating the involve partitions can be neglected when deal with massive data, while the costs account for considerable proportion on data set of small and medium size, since the overall execution time is limited. Secondly, the proposed improvement strategies for PTF still remain valid here and the execution time of PTF always can be comparable against the best state-of-the-art algorithms.

According to the scope of massive data specified in Sect. 1.1, in the experiments, we do not consider the performance evaluation of parallel and distributed environment [37]. It should be noted that a more efficient algorithm always is required to speed up the processing of mining problem on a larger scale of data. After all, we cannot *only rely on* the computer cluster to deal with massive data, since the commonly scale-out approach has its limit due to technology and monetary wall [34]. In the experiments, we generate the synthetic data set of up to 10^9 transactions to evaluate the performance of PTF. Besides, we select the real-life data sets from open-source SPMF data mining library, including two largest data sets of typical sparse and dense types for frequent itemset mining. As verified by the experimental results, the performance of PTF is evaluated in several aspects and PTF shows a satisfactory performance on massive data. In the future work, we will extend the idea of PTF algorithm to parallel and distributed environment.

In this part, we also discuss the issues of false negative and false positive for the results of PTF. A false negative is when a top-*k* frequent itemset gets rejected, and a false positive is when an unqualified itemset is considered to be a top-*k* frequent itemset. During the execution of PTF, apart from the candidate pruning strategy, PTF utilizes a min heap *MH* to maintain the *k* itemsets generated so far with the highest supports, computes the supports of the possible itemsets gradually, updates *MH* if necessary. When the execution terminates, the itemsets in *MH* naturally are the required results. Obviously, there is no false positive and no false negative now. With the candidate pruning strategy, the emphasis is whether PTF discards the top-*k* frequent itemsets incorrectly. With timeliness pruning, PTF utilizes the actual supports of the itemsets used for extension to determine whether the pruning is performed. It does not discard the top-*k* frequent itemsets. For last-item pruning, it uses the 2-itemsets formed by two last items to determine whether the pruning is executed. According to the anti-monotone property, it also does not discard the top-*k* frequent itemsets. In short, there are no any false negatives or any false positives in the results of PTF.

7 Conclusion

This paper considers the problem of top-*k* frequent itemset mining (top-*k* FIM) on massive data. It is analyzed that the existing algorithms focus on the data set of small and medium size. On massive data, they either need several full table scans or have to construct the assistant structure which cannot be kept completely in the memory, incurring high execution cost. This paper develops a novel PTF algorithm, which is based on prefix-partitioning strategy, to mine top-*k* frequent itemsets on massive data efficiently. The transactions in each prefix-based partition share the same prefix item and frequent itemsets with the prefix item can be computed in the partition separately. PTF processes the partitions sequentially, and the overwhelming majority of partitions, which cannot generate any top-*k* frequent itemsets, can be skipped directly. For each partition to be processed, vertical mining is performed on the vertical representation of the promising items only by high-support-first principle. The hybrid vertical storage mode is devised to maintain the tid-sets of the items adaptively and reduce the execution cost significantly. The candidate pruning operations, including timeliness pruning and

last-item pruning, are presented to reduce the number of the candidates. The extensive experimental results show that, compared with the existing algorithms, PTF can achieve up to three orders of magnitude speedup ratio.

Acknowledgements We appreciate anonymous reviewers for reviewing the manuscripts, despite their busy schedules, taking out time to read the manuscript and give the helpful feedback.

Author Contributions XW, Conceptualization, Software, Investigation, Writing—original draft preparation; XH, Methodology, Investigation, Writing—reviewing and editing, Funding acquisition.

Funding This study was supported by National Natural Science Foundation of China [Grant Number (U21A20513)] and Taishan Scholars Program of Shandong Province [Grant Number (tsqn202211091)]. Natural Science Foundation of Shandong Province (grant number [ZR2023QF059]), Foundation of Young Teachers of HIT (grant number [IDGA10002193]).

Data Availability All data, models, and code generated or used during the study appear in the submitted article.

Declarations

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article.

Ethical Approval This article does not contain any studies with human participants or animals performed by any of the authors. Results are gotten through simulation and tested number of times to take final value.

References

1. Abdelaal AA, Abed S, Alshayeji M, Al-laho M (2021) Customized frequent patterns mining algorithms for enhanced top-rank-k frequent pattern mining. Expert Syst Appl 169:114530
2. Aggarwal CC (2015) Data mining—the textbook. Springer, Berlin
3. Aggarwal CC, Han J (eds) (2014) Frequent pattern mining. Springer, Berlin
4. Agrawal R, Imielinski T, Swami AN (1993) Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD international conference on management of data, pp 207–216
5. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: Proceedings of 20th international conference on very large data bases, pp 487–499
6. Amphawan K, Lenca P (2015) Mining top-k frequent-regular closed patterns. Expert Syst Appl 42(21):7882–7894
7. Apiletti D, Baralis E, Cerquitelli T, Garza P, Pulvirenti F, Venturini L (2017) Frequent itemsets mining for big data: a comparative analysis. Big Data Res 9:67–83
8. Aryabarzan N, Minaei-Bidgoli B, Teshnehlab M (2018) negFIN: an efficient algorithm for fast mining frequent itemsets. Expert Syst Appl 105:129–143
9. Atluri G, Karpatne A, Kumar V (2018) Spatio-temporal data mining: a survey of problems and methods. ACM Comput Surv 51(4):83:1-83:41
10. Bistarelli S, Bonchi F (2007) Soft constraint based pattern mining. Data Knowl Eng 62(1):118–137
11. Burdick D, Calimlim M, Flannick J, Gehrke J, Yiu T (2005) MAFIA: a maximal frequent itemset algorithm. IEEE Trans Knowl Data Eng 17(11):1490–1504
12. Channon DF, Sammut-Bonnici T (2015) Wiley encyclopedia of management—vol 12 strategic management. Wiley, New York
13. Chen H (2014) Mining top-k frequent patterns over data streams sliding window. J Intell Inf Syst 42(1):111–131
14. Cheung Y-L, Fu AW-C (2004) Mining frequent itemsets without support threshold: With and without item constraints. IEEE Trans Knowl Data Eng 16(9):1052–1069
15. Chuang K-T, Huang J-L, Chen M-S (2008) Mining phtop-k frequent patterns in the presence of the memory constraint. VLDB J 17(5):1321–1344
16. Dawar S, Goyal V, Bera D (2017) A hybrid framework for mining high-utility itemsets in a sparse transaction database. Appl Intell 47(3):809–827
17. Deng Z-H (2016) DiffNodesets: an efficient structure for fast mining frequent itemsets. Appl Soft Comput 41:214–223
18. Deng Z-H, Lv S-L (2014) Fast mining frequent itemsets using nodesets. Expert Syst Appl 41(10):4505–4512
19. Deng Z-H, Wang Z, Jiang J-J (2012) A new algorithm for fast mining frequent itemsets using n-lists. Sci China Inf Sci 55(9):2008–2030
20. Dunkel B, Soparkar N (1999) Data organization and access for efficient data mining. In: Proceedings of the 15th international conference on data engineering, pp 522–529
21. Fang G-D, Deng Z-H (2008) VTK: vertical mining of top-rank-k frequent patterns. In: Ma J, Yin Y, Yu J, Zhou S (eds) Fifth international conference on fuzzy systems and knowledge discovery, FSKD 2008, 18–20 October 2008, Jinan, Shandong, China, Proceedings, vol 2. IEEE Computer Society, pp 620–624
22. Fournier-Viger, P, Lin JC-W, Gomariz A, Gueniche T et al (2016) The SPMF open-source data mining library version 2. In: Proceedings of 19th European conference on principles of data mining and knowledge discovery, Part III, volume 9853 of LNCS. Springer, Berlin, pp 36–40
23. Fu AW-C, Kwong RW-w, Tang J (2000) Mining phN-most interesting itemsets. In: Proceedings of the 12th international symposium on foundations of intelligent systems, volume 1932 of LNCS, pp 59–67. Springer, Berlin
24. Grahne G, Zhu J (2005) Fast algorithms for frequent itemset mining using fp-trees. IEEE Trans Knowl Data Eng 17(10):1347–1362
25. Grossi V, Romei A, Turini F (2017) Survey on using constraints in data mining. Data Min Knowl Discov 31(2):424–464
26. Guns T, Dries A, Nijssen S, Tack G, De Raedt L (2017) MiningZinc: a declarative framework for constraint-based mining. Artif Intell 244:6–29
27. Guns T, Nijssen S, De Raedt L (2011) Itemset mining: a constraint programming perspective. Artif Intell 175(12–13):1951–1983
28. Halim Z, Ali O, Khan MG (2021) On the efficient representation of datasets as graphs to mine maximal frequent itemsets. IEEE Trans Knowl Data Eng 33(4):1674–1691

29. Han J, Cheng H, Xin D, Yan X (2007) Frequent pattern mining: current status and future directions. Data Min Knowl Discov 15(1):55–86
30. Han J, Pei J, Yin Y, Mao R (2004) Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Min Knowl Discov 8(1):53–87
31. Han X, Liu X, Chen J, Lai G, Gao H, Li J (2019) Efficiently mining frequent itemsets on massive data. IEEE Access 7:31409–31421
32. Hirano S, Tsumoto S (2019) Mining frequent temporal patterns from medical data based on fuzzy ranged relations. In: 2019 IEEE international conference on big data (big data), pp 2654–2658. IEEE
33. Keogh EJ, Lonardi S, Ratanamahatana C (2004) Towards parameter-free data mining. In: Proceedings of the tenth ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 206–215
34. Kersten ML, Sidirourgos L (2017) A database system with amnesia. In: 8th Biennial conference on innovative data systems research, CIDR 2017, Chaminade, CA, USA, January 8–11, 2017, Online Proceedings. http://www.cidrdb.org
35. Le T, Vo B (2015) An n-list-based algorithm for mining frequent closed patterns. Expert Syst Appl 42(19):6648–6657
36. Lee J, Clifton CW (2014) Top-k frequent itemsets via differentially private FP-trees. In: The 20th ACM SIGKDD international conference on knowledge discovery and data mining, pp 931–940. ACM
37. Li H, Wang Y, Zhang D, Zhang M, Chang EY (2008) PFP: parallel FP-growth for query recommendation. In: Pu P, Bridge DG, Mobasher B, Ricci F (eds) Proceedings of the 2008 ACM conference on recommender systems, RecSys 2008, Lausanne, Switzerland, October 23–25, 2008. ACM, pp 107–114
38. Liang W, Chen H, Zhang J, Zhao D, Li C (2020) An effective scheme for top-k frequent itemset mining under differential privacy conditions. Sci China Inf Sci 63(5)
39. Liu G, Hongjun L, Lou W, Yabo X, Xu JY (2004) Efficient mining of frequent patterns using ascending frequency ordered prefix-tree. Data Min Knowl Discov 9(3):249–274
40. Luna JM, Fournier-Viger P, Ventura S (2019) Frequent itemset mining: a 25 years review. Wiley Interdiscip Rev Data Min Knowl Discov 9(6)
41. Naulaerts S, Meysman P, Bittremieux W, Trung-Nghia V et al (2015) A primer to frequent itemset mining for bioinformatics. Briefings Bioinform 16(2):216–231
42. Park JS, Chen M-S, Yu PS (1995) An effective hash based algorithm for mining association rules. In: Proceedings of the 1995 ACM SIGMOD international conference on management of data. ACM Press, pp 175–186
43. Pei J, Han J, Lu H, Nishio S, Tang S, Yang D (2001) H-mine: hyper-structure mining of frequent patterns in large databases. In: Proceedings of the 2001 IEEE international conference on data mining, pp 441–448
44. Pham T-T, Do T, Nguyen A, Vo B, Hong T-P (2020) An efficient method for mining top-k closed sequential patterns. IEEE Access 8:118156–118163
45. Pyun G, Yun U (2014) Mining top-k frequent patterns with combination reducing techniques. Appl Intell 41(1):76–98
46. Salam A, Khayal MSH (2012) Mining top-k frequent patterns without minimum support threshold. Knowl Inf Syst 30(1):57–86
47. Savasere A, Omiecinski E, Navathe SB (1995) An efficient algorithm for mining association rules in large databases. In: Proceedings of 21th international conference on very large data bases. Morgan Kaufmann, pp 432–444
48. Shah A, Halim Z (2019) On efficient mining of frequent itemsets from big uncertain databases. J Grid Comput 17(4):831–850
49. Tang B, Zeng J, Tang Q, Yang C, Shen Q, Hou U L, Yan X, Zeng D (2022) CheetahKG: a demonstration for core-based top-k frequent pattern discovery on knowledge graphs. In 38th IEEE International Conference on Data Engineering, ICDE 2022, pp 3134–3137. IEEE
50. Tzvetkov P, Yan X, Han J (2005) TSP: mining top-phk closed sequential patterns. Knowl Inf Syst 7(4):438–457
51. Ugarte W, Boizumault P, Loudni S, Crémilleux B, Lepailleur A (2015) Soft constraints for pattern mining. J Intell Inf Syst 44(2):193–221
52. Vo B, Pham S, Le T, Deng Z-H (2017) A novel approach for mining maximal frequent patterns. Expert Syst Appl 73:178–186
53. Wang J, Han J, Ying L, Tzvetkov P (2005) TFP: an efficient algorithm for mining top-k frequent closed itemsets. IEEE Trans Knowl Data Eng 17(5):652–664
54. Xu T, Xu A, Mango J, Liu P, Ma X, Zhang L (2022) Efficient processing of top-k frequent spatial keyword queries. Sci Rep 12:7352
55. Zaki MJ (2000) Scalable algorithms for association mining. IEEE Trans Knowl Data Eng 12(3):372–390
56. Zaki MJ, Gouda K (2003) Fast vertical mining using diffsets. In: Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 326–335
57. Zaki MJ, Hsiao C-J (2002) CHARM: an efficient algorithm for closed itemset mining. In: Proceedings of the Second SIAM international conference on data mining. SIAM, pp 457–473
58. Zhang S, Xindong W, Zhang C, Jingli L (2008) Computing the minimum-support for mining frequent patterns. Knowl Inf Syst 15(2):233–257