Mining Top-K Frequent Closed Patterns without Minimum Support*

Jiawei Han Jianyong Wang Ying Lu Petre Tzvetkov
University of Illinois at Urbana-Champaign, Illinois, U.S.A.
{hanj, wangj, yinglu, tzvetkov}@cs.uiuc.edu

ABSTRACT

In this paper, we propose a new mining task: mining top-k frequent closed patterns of length no less than min_ℓ, where k is the desired number of frequent closed patterns to be mined, and min_ℓ is the minimal length of each pattern. An efficient algorithm, called TFP, is developed for mining such patterns *without* minimum support. Two methods, *closed_node_count* and *descendant_sum* are proposed to effectively raise support threshold and prune FP-tree both *during* and *after* the construction of FP-tree. During the mining process, a novel *top-down and bottom-up combined* FP-tree mining strategy is developed to speed-up support-raising and closed frequent pattern discovering. In addition, a fast hash-based closed pattern verification scheme has been employed to check efficiently if a potential closed pattern is really closed.

Our performance study shows that in most cases, TFP outperforms CLOSET and CHARM, two efficient frequent closed pattern mining algorithms, even when both are running with the best tuned *min_support*. Furthermore, the method can be extended to generate association rules and to incorporate user-specified constraints. Thus we conclude that for frequent pattern mining, mining top-k frequent closed patterns without *min_support* is more preferable than the traditional *min_support*-based mining.

1. INTRODUCTION

As one of several essential data mining tasks, mining frequent patterns has been studied extensively in literature. From the implementation methodology point of view, recently developed frequent pattern mining algorithms can be categorized into three classes: (1) Apriori-based, horizontal formatting method, with Apriori [1] as its representative, (2) Apriori-based, vertical formatting method, such as CHARM [8], and (3) projection-based pattern growth method, which may explore some compressed data structure such as FP-tree, as in FP-growth [3].

The common *framework* is to use a *min_support* threshold to ensure the generation of the correct and complete set of frequent patterns, based on the popular Apriori property [1]: *every subpattern of a frequent pattern must be frequent* (also called the *downward closure property*). Unfortunately, this framework, though simple, leads to the following two problems that may hinder its popular use.

First, Setting *min_support* is quite subtle: *a too small threshold may lead to the generation of thousands of patterns, whereas a too big one may often generate no answers.* Our own experience at mining shopping transaction databases tells us this is by no means an easy task.

Second, frequent pattern mining often leads to the generation of a large number of patterns (and an even larger number of mined rules). And mining a long pattern may unavoidably generate an exponential number of subpatterns due to the downward closure property of the mining process.

The second problem has been noted and examined by researchers recently, proposing to mine (frequent) closed patterns [5, 7, 8] instead. Since a closed pattern is the pattern that covers all of its subpatterns with the same support, one just need to mine the set of closed patterns (often much smaller than the whole set of frequent patterns), without losing information. Therefore, mining closed patterns should be the default task for mining frequent patterns.

The above observations indicate that it is often preferable to change the task of *mining frequent patterns* to mining top-k frequent closed patterns of minimum length min_ℓ, where k is a user-desired number of frequent closed patterns to be mined (which is easy to specify or set default), *top-k* refers to the k most frequent closed patterns, and min_ℓ, the minimal length of closed patterns, is another parameter easy to set. Notice that without min_ℓ, the patterns found will be of length one (or their corresponding closed superpatterns) since a pattern can never occur more frequently than its corresponding shorter ones (i.e., subpatterns) in any database.

In this paper, we study the problem of mining top-k frequent closed patterns of minimal length min_ℓ efficiently without *min_support*, i.e., starting with $min\_support = 0$. Our study is focused on the FP-tree-based algorithm. An efficient algorithm, called TFP, is developed by taking advantage of a few interesting properties of top-k frequent closed patterns with minimum length min_ℓ, including (1) any transactions shorter than min_ℓ will not be included in the pattern mining, (2) *min_support* can be raised dynamically in the FP-tree construction, which will help pruning the tree before mining, and (3) the most promising tree branches can be mined first to raise *min_support* further, and the raised *min_support* is then used to effectively prune the remaining branches.

Performance study shows that TFP has surprisingly high performance. In most cases, it is better than two efficient frequent closed pattern mining algorithms, CLOSET and CHARM, with the best tuned *min_support*.

Moreover, association rules can be extracted by minor extension of the method, and constraints can also be incorpo-

* The work was supported in part by U.S. National Science Foundation, University of Illinois, and Microsoft Research.

rated into top-k closed pattern mining.

Therefore, we conclude that mining top-k frequent closed patterns without minimum support is more preferable (from both usability and efficiency points of view) than traditional *min_support*-based mining.

The remaining of the paper is organized as follows. In Section 2, the basic concept of top-k closed pattern mining is introduced, and the problem is analyzed with the related properties identified. Section 3 presents the algorithm for mining top-k closed patterns. A performance study of the algorithm is reported in Section 4. Extensions of the method are discussed in Section 5, and we conclude our study in Section 6.

2. PROBLEM DEFINITION

In this section, we first introduce the basic concepts of top-k closed patterns, then analyze the problems and present an interesting method for mining top-k closed patterns.

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of **items**. An **itemset** X is a non-empty subset of I. The **length of itemset** X is the number of items contained in X, and X is called an **l-itemset** if its length is l. A tuple $\langle tid, X\rangle$ is called a **transaction** where tid is a transaction identifier and X is an itemset. A **transaction database** TDB is a set of transactions. An itemset X is **contained** in transaction $\langle tid, Y\rangle$ if $X \subseteq Y$. Given a transaction database TDB, the **support** of an itemset X, denoted as $sup(X)$, is the number of transactions in TDB which contain X.

DEFINITION 1. **(top-k closed itemset)** *An itemset X is a* **closed itemset** *if there exists no itemset X' such that (1)$X \subset X'$, and (2) $\forall$ transaction T, $X \in T \rightarrow X' \in T$. A closed itemset X is a* **top-k closed itemset of minimal length** *min_ℓ if there exist*[1] *no more than $(k-1)$ closed itemsets of length at least min_ℓ whose support is higher than that of X.* ■

Our task is to mine top-k closed itemsets of minimal length min_ℓ efficiently in a large transaction database.

EXAMPLE 1 (**A transaction dataset example**). *Let Table 1 be our transaction database TDB. Suppose our task is to find top-4 frequent closed patterns with $min\_\ell = 4$.* ■

TID	Items	Ordered Items
T100	i, c, d, e, g, h	d, c, e, h, i, g
T200	m, a, p, c, e, d	d, c, e, a, p, m
T300	a, i, b, d, e, g	d, e, a, b, i, g
T400	b, a, d, h, c, n	d, c, a, h, b, n
T500	a, e, c	c, e, a
T600	n, a, c, d, e	d, c, e, a, n
T700	p, a, b, c, d, e, h	d, c, e, a, h, b, p

Table 1: A transaction database TDB.

Our first question is "*which mining methodology should be chosen from among the three choices:* Apriori, CHARM, *and* FP-growth*?*" Without min_support threshold, one can still use Apriori to mine all the l-itemsets level-by-level for l from 1 to min_ℓ. However, since one cannot use the downward closure property to prune *infrequent* l-itemsets for generation of $(l+1)$-itemset candidates, Apriori has to join all the length l itemsets to generate length $l+1$ candidates for all l from 1 to $min\_\ell - 1$. This is inefficient. CHARM loses its pruning power as well since it has to generate transaction id_list for every item, and find their intersected transaction id_list for every pair of such items since there is no itemset that can be pruned. Will the fate be the same for FP-growth? Since FP-growth uses a compressed data structure FP-tree to register TDB, all the possible itemsets of a transaction and their corresponding length information are preserved in the corresponding branch of the FP-tree. Moreover, FP-tree preserves the support information of the itemsets as well. Thus it is possible to utilize such information to speed up mining. Of the three possible methods, we will examine FP-growth in detail.

The question then becomes, "*how can we extend* FP-growth *for efficient top-k frequent closed pattern mining?*" We have the following ideas: (1) 0-*min_support* forces us to construct the "full FP-tree", however, with top-k in mind, one can capture sufficient higher support closed nodes in tree construction and dynamically raise *min_support* to prune the tree; and (2) in FP-tree mining, one can first mine the most promising subtrees so that high support patterns can be derived earlier, which can be used to prune low-support subtrees. In the following section, we will develop the method step-by-step.

3. MINING TOP-K FREQUENT CLOSED PATTERNS: METHOD DEVELOPMENT

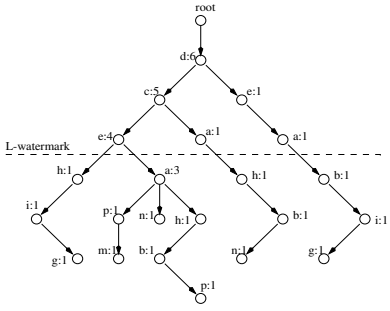
In this section, we perform step-by-step analysis to develop an efficient method for mining top-k frequent closed patterns.

3.1 Short transactions and l-counts

Before studying FP-tree construction, we notice,

REMARK 3.1 (**Short transactions**). *If a transaction T contains less than min_ℓ distinct items, none of the items in T can contribute to a pattern of minimum length min_ℓ.*

For the discussions below, we will consider only the transactions that satisfy the minimum length requirement.



a) The FP-tree constructed from TDB.

Item	count	l-count
d	6	0
c	5	0
e	5	0
a	5	3
h	3	3
b	3	3
i	2	2
g	2	2
n	2	2
p	2	2
m	1	1

b) Count and l-count of items.

Figure 1: FP-tree and its header table.

Let the occurrence frequency of each item be stored as *count* in the *(global) header table.* Here we introduce in

[1]Since there could be more than one itemset having the same support in a transaction database, to ensure the set mined is independent of the ordering of the items and transactions, our method will mine every closed itemset whose support is no less than the k-th frequent closed itemset.

the header table another counter, l(ow)-count, which records the total occurrences of an item at the level no higher than min_ℓ in the FP-tree, as shown in Figure 1 b).

REMARK 3.2 (**l-count**). *If the l-count of an item t is lower than min_support, t cannot be used to generate frequent itemset of length no less than min_ℓ.*

Rationale. *Based on the rules for generation of frequent itemset in* FP-tree *[3], only a node residing at the level min_ℓ or lower (i.e., deeper in the tree) may generate a prefix path no shorter than min_ℓ. Based on Remark 3.1, short prefix paths will not contribute to the generation of itemset with length greater or equal to min_ℓ. Thus only the nodes with l-count no lower than min_support may generate frequent itemset of length no less than min_ℓ.* ∎

People may wonder that our assumption is to start with $min\_support = 0$, how could we still use the notion of *min_support*? Notice that if we can find a good number (i.e., no less than k) of closed nodes with nontrivial support during the FP-tree construction or before tree mining, the *min_support* can be raised, which can be used to prune other items with low support.

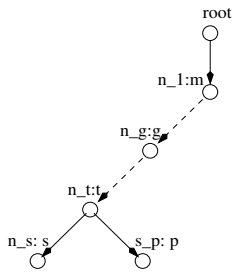
3.2 Raising *min_support* for pruning FP-tree

Since our goal is to mine top-k frequent closed nodes, in order to raise *min_support* effectively, one must ensure that the nodes taken into count are closed.

LEMMA 3.1 (**Closed node**). *At any time during the construction of an* FP-tree*, a node n_t is a closed node (representing a closed itemset) if it falls into one of the following three cases: (1) n_t has more than one child and n_t carries more count than the sum of its children, (2) n_t carries more count than its child, and (3) n_t is a leaf node.*

Rationale. *This can be easily derived from the definition of closed itemset and the rules for construction of* FP-tree *[3]. As shown in Figure 2 a), a node $(n_t : t)$ denotes an itemset $n_1, \ldots, n_t$ with support t. Any later transaction (or prefix-path) that contains exactly the same set of items will be represented by the same node in the tree with increased support. If n_t has more than one child and n_t carries more count than the sum of its children, then n_t cannot carry the same support as any of its children, and thus n_t must be a closed node. The same reason holds if n_t carries more count than its child. If n_t is a leaf node, the future insertion of branches will make the node either remain as a leaf node or carry more count than its children, thus n_t must be a closed node as well.* ∎

To raise *min_support* dynamically during the FP-tree construction, a simple data structure, called *closed_node_count* array, can be used to register the current count of each closed l-node with support node#, as illustrated in the left. The array is constructed as follows. Initially, all the count of each node# is initialized to 0 (or only the non-zero l-node is registered, depending on the implementation). Each closed l-node with support m in the FP-tree has one count in the *count* slot of node# m. During the construction of an FP-tree, suppose inserting one transaction into a branch makes the support of a closed l-node P increases from m to



a) Judging the closed node

l-node#	count
95	1
⋮	⋮
5	9
4	7
3	17
2	32
1	86

b) Count of closed l-nodes with support node#.

Figure 2: Closed node and closed node count array.

$m + 1$. Then the *count* corresponding to the node# $m + 1$ in the array is increased by one whereas that corresponding to the node# m is decreased by one.

Based on how the *closed_node_count* array is constructed, one can easily derive the following lemma.

LEMMA 3.2 (**Raise *min_support* using *closed_node***). *At any time during the construction of an* FP-tree*, the minimum support for mining top-k closed itemsets will be at least equal to the node# if the sum of the closed_node_count array from the top to node# is no less than k.* ∎

Besides using the *closed_node_count* array to raise minimum support, there is another method to raise *min_support* with FP-tree, called anchor-node descendant-sum, or simply descendant-sum, as described below. An anchor-node is a node at level $min\_\ell - 1$ of an FP-tree. It is called an *anchor-node* since it serves as an anchor to the (descendant) nodes at level min_ℓ and below. The method is described in the following example.

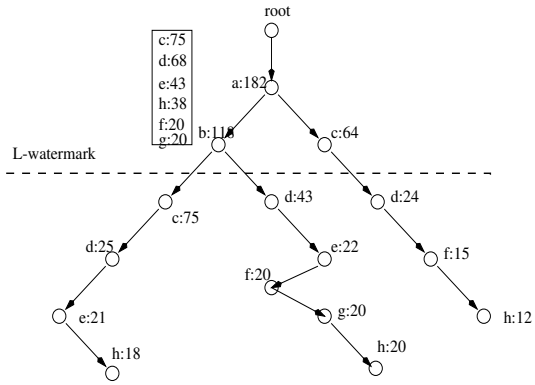


Figure 3: Calculate *descendant_sum* for an anchor node of an FP-tree**.**

EXAMPLE 2. *As shown in Figure 3, node b is an anchor node since it resides at level $min\_\ell - 1 = 2$. At this node, we collect the sum of the counts for each distinct itemset of b's descendants. For example, since b has two descendant d-nodes, $(d : 25)$ and $(d : 43)$, b's descendant_sum for d is $(d : 68)$ (which means that the support of itemset abd contributed from b's descendants is 68). From the* FP-tree *presented in Figure 3, it is easy to figure out that b's descendant_sum should be $\{(c : 75), (d : 68), (e : 43), (h : 38), (f : 20), (g : 20)\}$. Such summary information may raise min_support*

effectively. For example, min_support for top-3 closed nodes should be at least 43 based on b's descendant_sum. ∎

LEMMA 3.3 (*descendant_sum*). *Each distinct count in descendant_sum of an anchor node represents the minimum count of one distinct closed pattern that can be generated by the* FP-tree.

Rationale. *Let the path from the root of the* FP-tree *to an anchor node b be β. Let a descendant of b be d and its count be $count_d$. Then based on the method for* FP-tree *construction, there must exist an itemset $\beta \cdot d$ whose count is $count_d$. If $\beta \cdot d$ is a closed node, then it is the unique closed node in the* FP-tree *with count $count_d$; otherwise, there must exist a closed pattern which is its super-pattern with support $count_d$. Since another node in b's descendant_sum with the same support $count_d$ may share such a closed node with $\beta \cdot d$, and also another branch may contribute additional count to such a closed node, thus only a distinct count in descendant_sum of b may represent the minimum count of a distinct closed pattern generated by the* FP-tree. ∎

We have the following observations regarding the two support raising methods. First, the *closed_node_count* method is cheap (only one array) and is easy to implement, and it can be performed at any time during the tree insertion process. Second, comparing with *closed_node_count*, *descendant_sum* is more effective at raising *min_support*, but is more costly since there could be many $(min\_\ell - 1)$ level nodes in an FP-tree, and each such node will need a *descendant_sum* structure. Moreover, before fully scanning the database, one does not know which node may eventually have a very high count. Thus it is tricky to select the appropriate anchor nodes for support raising: too many anchor nodes may waste storage space, whereas too few nodes may not be able to register enough count information to raise *min_support* effectively. Computing *descendant_sum* structure for low count nodes could be a waste since it usually derives small *descendant_sum* and may not raise *min_support* effectively.

Based on the above analysis, our implementation explores both techniques but at different times: During the FP-tree construction, it keeps an *closed_node_count* array which raises *min_support*, dynamically prunes some infrequent nodes, and reduces the size of the FP-tree to be constructed. After scanning the database (i.e., the FP-tree is constructed), we traverse the subtree of the level $(min\_\ell - 1)$ node with the highest support to calculate *descendant_sum*. This will effectively raise *min_support*. If the so-raised *min_support* is still less than the highest support of the remaining level $(min\_\ell - 1)$ nodes, the remaining node with the highest support will be traversed, and this process continues. Based on our experiments, only a small number of nodes need to be so traversed (if k for top-k is less than 1000) in most cases.

3.3 Efficient mining of FP-tree for top-k patterns

The raise of *min_support* prunes the FP-tree and speeds up the mining. However, the overall critical performance gain comes from efficient FP-tree mining.

We have the following observations.

REMARK 3.3 (**Item skipping**). *If the count of an item in the global header table is less than min_support, then it is infrequent and its nodes should be removed from the* FP-tree. *Moreover, if the l-count of an item in the global header table is less than min_support, the item should not be used to generate any conditional* FP-tree. ∎

The TFP-mining with FP-tree is similar to FP-growth. However, there are a few subtle points.

1. "Top-down" ordering of the items in the global header table for the generation of conditional FP-trees.

The first subtlety is in what order the conditional FP-trees should be generated for top-k mining. Notice since FP-growth in [3] is to find the complete set of frequent patterns, its mining may start from any item in the header table. For top-k mining, our goal is to *find only the patterns with high support and raise the min_support as fast as possible to avoid unnecessary work*. Thus mining should start from the item that has the first non-zero l-count (which usually carries the highest l-count) in the header table, and walk down the header table entries to mine subsequent items (i.e., in the sorted_item_list order). This ordering is based on that items with higher l-count usually produce patterns with higher support. With this ordering, *min_support* can be raised faster and the top-k patterns can be discovered earlier. In addition, an item with l-count less than *min_support* do not have to generate conditional FP-tree for further mining (as stated in Remark 3.2). Thus, the faster the *min_support* can be raised, the more and earlier pruning can be done.

2. "Bottom-up" ordering of the items in a local header table for mining conditional FP-trees.

The second subtlety is how to mine conditional FP-trees. We have shown that the generation of conditional FP-trees should follow the order of the sorted_item_list, which can be viewed as top-down walking through the header table. However, it is often more beneficial to mine a conditional pattern tree in the "bottom-up" manner in the sense that we first mine the items that are located at the low end of a tree branch since it tends to produce the longest patterns first then followed by shorter ones. It is more efficient to first generate long closed patterns since the patterns containing only the subset items can be absorbed by them easily.

3. Efficient searching and maintaining of closed patterns using a pattern-tree structure.

The third subtle point is how to efficiently maintain the set of current frequent closed patterns and check whether a new pattern is a closed one.

During the mining process, a pattern-tree is used to keep the set of current frequent closed patterns. The structure of pattern-tree is similar to that of FP-tree. Recall that the items in a branch of the FP-tree are ordered in the support-decreasing order. This ordering is crucial for closed pattern verification (to be discussed below), thus we retain this item ordering in the patterns mined. The major difference between FP-tree and pattern-tree is that the former stores transactions in compressed form, whereas the latter stores potential closed frequent patterns.

The bottom-up mining of the conditional FP-trees generates patterns in such an order: for patterns that share prefixes, longer patterns are generated first. In addition, there is a total ordering over the patterns generated. This leads to our **closed frequent pattern verification scheme**, presented as follows.

Let $(i_1, \ldots, i_l, \ldots, i_j, \ldots, i_n)$ be the *sorted_item_list*, where i_l is the first non-zero l-count item, and i_j be the item whose

conditional FP-tree is currently being mined. Then the set of already mined closed patterns, S, can be split into two subsets: (1) S_{old}, obtained by mining the conditional trees corresponding to items from i_l to i_{j-1} (i.e., none of the itemsets contains item i_j), and (2) S_{i_j}, obtained so far by mining i_j's conditional tree (i.e., every itemset contains item i_j). Upon finding a new pattern p during the mining of i_j's conditional tree, we need to perform new pattern checking (checking against S_{i_j}) and old pattern checking (checking against S_{old}).

The new pattern checking is performed as follows. Since the mining of the conditional tree is in a bottom-up manner, just like CLOSET, we need to check whether (1) p is a subpattern of another pattern p_{i_j} in S_{i_j}, and (2) $supp(p) \equiv supp(p_{i_j})$. If the answer is no, i.e., p passes new pattern checking, p becomes a new closed pattern with respect to S. Note that because patterns in S_{old} do not contain item i_j, there is no need to check if p is a subpattern of the patterns in S_{old}.

The old pattern checking is performed as follows. Since the global FP-tree is mined in a top-down manner, pattern p may be a super-pattern of another pattern, p_{old}, in S_{old} with $supp(p) \equiv supp(p_{old})$. In this case, p_{old} cannot be a closed pattern since it is absorbed by p. Therefore, if p has passed both new and old pattern checking, it can be used to raise the support threshold. Otherwise, if p passes only the new pattern checking, then it is inserted into the pattern-tree, but it cannot be used to raise the support threshold.

The correctness of the above checking is shown in the following lemmas.

LEMMA 3.4 (**New pattern checking**). *If a pattern p cannot pass the new pattern checking, there must exist a pattern, p_{i_j}, in S_{i_j}, which must also contain item i_j with $supp(p_{i_j}) \equiv supp(p)$.*

Rationale. *This can be obtained directly from the new pattern checking method.* ∎

Let $prefix(p)$ be the prefix pattern of a pattern p (i.e., obtained by removing the last item i_j from p).

LEMMA 3.5 (**Old pattern checking**). *For old pattern checking, we only need to check if there exists a pattern $prefix(p)$ in S_{old} with $supp(prefix(p)) \equiv supp(p)$.*

Rationale. *Since a pattern in S_{old} does not contain item i_j, it cannot become a super-pattern of p. Thus we only need to check if it is a subpattern of p. In fact we only need to check if there is a pattern $prefix(p)$ in S_{old} with the same support as p. We can prove this by contradiction. Let us assume there is another subpattern of $prefix(p)$ that can be absorbed by p. If this is the case, according to our mining order, we know this subpattern must have been absorbed by $prefix(p)$ either via new pattern checking or old pattern checking.* ∎

LEMMA 3.6 (**Support raise**). *If a newly mined pattern p can pass both new pattern checking and old pattern checking, then it is safe to use p to raise min_support.*

Rationale. *From Lemma 3.5, there will be two possibilities for p. First, it is a real closed pattern, i.e., it will not be absorbed by any patterns later. Second, it will be absorbed by a later found pattern, and this pattern can only absorb pattern p. In this case, we will not use the later found pattern to raise support because it has already been used to raise support when we found pattern p (or p's precedents). Thus it is safe to use p to raise min_support.* ∎

To accelerate both new and old pattern checking, we introduce a two-level index header table into the pattern-tree structure. Notice that if a pattern can absorb (or be absorbed by) another pattern, the two patterns must have same support. Thus, our first index is based on the support of a pattern. In addition, for new pattern checking, we only need to check if pattern p can be absorbed by another pattern that also contains i_j; and for old pattern checking, we need to check if p can absorb $prefix(p)$ that ends with the second-last item of p. To speed up the checking, our second level indexing uses the last *item_ID* in a closed pattern as the index key. At each pattern-tree node, we also record the length of the pattern, in order to judge if the corresponding pattern needs to be checked.

The two-level index header table and the checking process are shown in the following example.

EXAMPLE 3 (**Closed pattern verification**). *Figure 4 shows a two-level indexing structure for verification of closed patterns. Based on the lemmas, we only need to index into the first structure based on the itemset support, and based on its matching of the last two items in the index structure to find whether the corresponding closed node is in the tree.* ∎

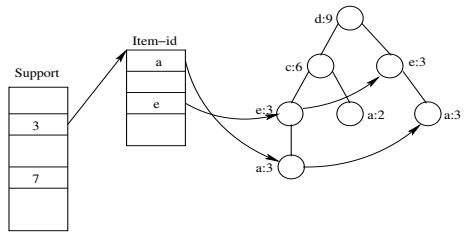


Figure 4: Two-level indexing for verification of closed patterns.

3.4 Algorithm

Now we summarize the entire mining process and present the mining algorithm.

ALGORITHM 1. *Mining top-k frequent closed itemsets with minimal length min_ℓ in a large transaction database.*

Input: *(1) A transaction database DB, (2) an integer k, i.e., the k most frequent closed itemsets to be mined, and (3) min_ℓ, the minimal length of the frequent closed itemsets.*

Output. *The set of frequent closed itemsets which satisfy the requirement.*

Method.

1. *Initially, min_support = 0;*
2. *Scan DB once*[2]*, Collect the occurrence frequency (count)*

[2] This scan can be replaced by a sampling process, which reduces one database scan but increases the chance that items may not be ordered very well due to biased sampling, which may hurt performance later. Thus such scan reduction may or may not improve the performance depending on the data characteristics and the ordering of transactions.

of every item in transactions and sort them in frequency descending order, which forms sorted_item_list *and the header of* FP-tree.

3. *Scan DB again, construct* FP-tree*, update the l-count in the header of the* FP-tree*, use closed node count array to raise min_support, and use this support to prune the tree. After scanning DB, traverse* FP-tree *with descendant_sum check, which raises min_support further, and the raised min_support is used to prune the tree.*

4. *Tree mining is performed by traversing down the header table, starting with the item with the first non-zero l-count and generate the conditional* FP-tree *for each item, as along as its l-count is no less than the current min_support. Each conditional* FP-tree *is mined in "bottom-up" (i.e., long to short) manner. Each mined closed pattern is inserted into a pattern-tree.*

5. *Output patterns from pattern-tree in the order of their support. Stop when it outputs k patterns.*

■

4. EXPERIMENTAL EVALUATION

In this section, we report our performance study of TFP over a variety of datasets.

In particular, we compare the efficiency of TFP with CHARM and CLOSET, two well known algorithms for mining frequent closed itemsets. To give the best possible credit to CHARM and CLOSET, our comparison is always based on assigning the best tuned *min_support* (which is difficult to obtain in practice) to the two algorithms so that they can generate the same top-k closed patterns for a user-specified k value (under a condition of min_ℓ). These optimal *min_support* are obtained by running TFP once under each experimental condition. This means that even if TFP has only comparable performance with those algorithms, it will still be far more useful than the latter due to its usability and the difficulty to speculate *min_support* without mining. In addition, we also study the scalability of TFP.

The experiments show that (1) the running time of TFP is shorter than CLOSET and CHARM in most cases when min_ℓ is long, and is comparable in other cases; and 2) TFP has nearly linear scalability.

4.1 Datasets

Both real and synthetic datasets are used in experiments, and they can be grouped into the following two categories.

1. Dense datasets that contain many long frequent closed patterns: 1) *pumsb* census data, which consists of 49,044 transactions, each with an average length of 74 items; 2) *connect-4* game state information data, which consists of 67,557 transactions, each with an average length of 43 items, and 3) *mushroom* characteristic data, which consists of 8,124 transactions, having an average length of 23 items. All these datasets are obtained from the UC-Irvine Machine Learning Database Repository.

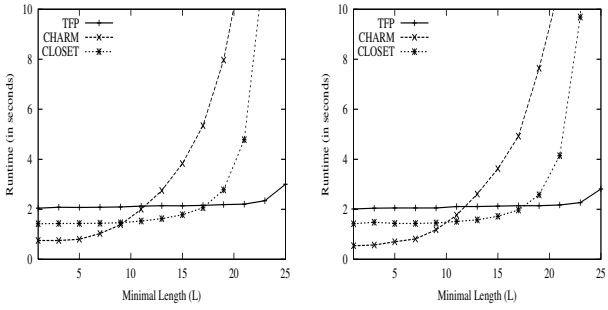
2. Sparse datasets: 1) *gazalle* click stream data, which consists of 59,601 transactions with an average length of 2.5 items, and contains many short (length below 10) and some very long closed patterns, (obtained from BlueMartini Software Inc.), and 2) *T10I4D100K* synthetic data from the IBM dataset generator, which consists of 100,000 transactions with an average length of 10 items, and with many closed frequent patterns having average length of 4.

4.2 Performance Results

All experiments were performed on a 1.7GHz Pentium-4 PC with 512MB of memory, running Windows 2000. The CHARM code was provided to us by its author. The CLOSET is an improved version that uses the same index-based closed node verification scheme as in TFP.

We compared the performance of TFP with CHARM and CLOSET on the 5 datasets by varying min_ℓ and K. In most cases K is selected to be either 100 or 500 which covers the range of typical K values. We also evaluated the scalability of TFP with respect to the size of database.

Dense Datasets: For the dense datasets with many long closed patterns, TFP performs consistently better than CHARM and CLOSET for longer min_ℓ.

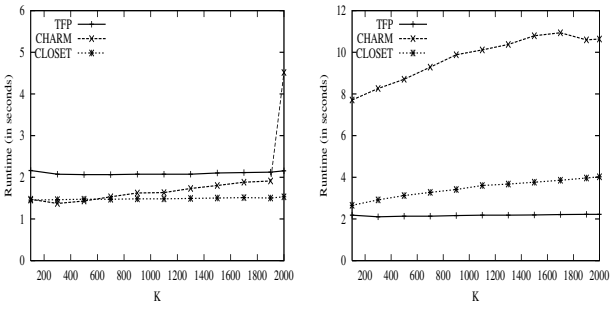


a) $K = 100$ b) $K = 500$

Figure 5: Performance on Connect-4 (I)

Figure 5 shows the running time of the three algorithms on the connect-4 dataset for K fixed at 100 and 500 respectively and min_ℓ ranging from 0 to 25. We observe that, TFP's running time for K at 100 and 500 remains stable over the range of min_ℓ. When min_ℓ reaches 11 or 12, TFP starts to outperform CHARM and the same for CLOSET when min_ℓ reaches 17 or 18. The reason is that for long patterns, the *min_support* is quite low. In this case, CHARM has to retain many short frequent patterns before forming the required longer patterns, and the FP-tree of CLOSET would also contain a large number of items that takes up much mining time. On the other hand, TFP is able to use the min_ℓ length restriction to cut many short frequent patterns early, thus reduce the total running time.

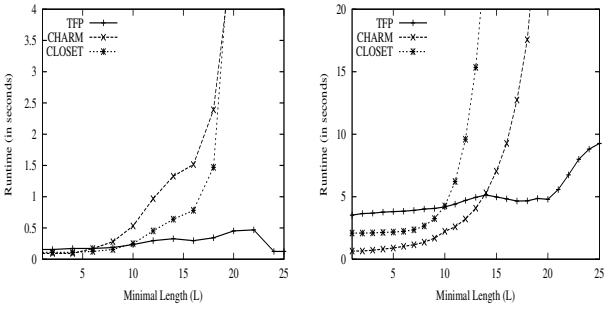
Figure 6 shows the running time of the three algorithms on the connect-4 dataset with min_ℓ set to 10 and 20 respectively and K ranging from 100 to 2000. For the connect-4 dataset, the average length of the frequent closed patterns is above 10, thus min_ℓ at 10 is considered to be a very low length restriction for this dataset. From a) we can see that even for very low length restriction such as 10, TFP's performance is comparable to that of CLOSET and CHARM when it runs without giving support threshold. For min_ℓ equal to 20, the running time for TFP is almost constant over the full range, and on average 5 times faster than CHARM and 2 to 3 times faster than CLOSET. We also noticed that, even for very low min_ℓ as K increases, the performance gap between TFP, CLOSET, and CHARM gets smaller.



a) $min\_\ell = 10$ b) $min\_\ell = 20$

Figure 6: Performance on Connect-4 (II)

Figure 7 shows the running time of the three algorithms on the mushroom and pumsb datasets with K set to 500 and min_ℓ ranges from 0 to 25. For the mushroom dataset, when min_ℓ is less than 6 all three algorithms have similar low running time. TFP keeps its low running time for the whole range of min_ℓ and starts to outperform CHARM when min_ℓ is as low as 6 and starts to outperform CLOSET when min_ℓ is equal to 8. Pumsb has very similar results as connect-4 and mushroom datasets.



a) Mushroom b) Pumsb

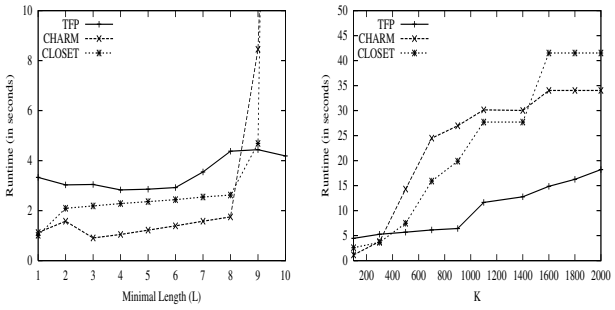
Figure 7: Performance on Mushroom and Pumsb

Sparse Dataset: Experiments show that TFP can efficiently mine sparse datasets without *min_support*. It has comparable performance with CHARM and CLOSET for low min_ℓ, and outperforms both on higher min_ℓ.

Figure 8a) shows the running times of TFP, CHARM, and CLOSET on T10I4D100K with K fixed at 100 and min_ℓ ranges from 1 to 10. Again, it demonstrates TFP's strength in dealing with long min_ℓ. At $min\_\ell = 8$, the performance of CHARM and CLOSET starts deteriorating, while TFP retains its good performance. Figure 8b) shows the performance on the same dataset but with min_ℓ fixed at 8 and varying K from 200 to 2000. The curves show that when K is above 400, the running times of CHARM and CLOSET are around 3 times slower than TFP.

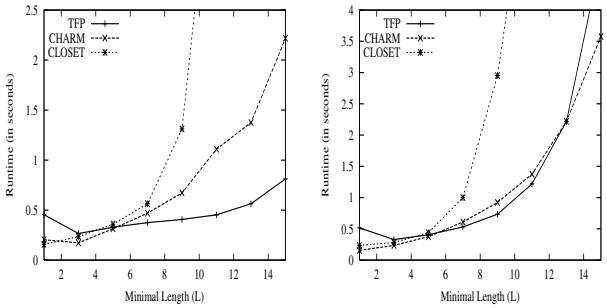
The experiments on the gazelle dataset are shown in Figure 9. For smaller K, TFP outperforms both CHARM and CLOSET for min_ℓ greater than or equal to 5. For $K = 500$, TFP continues to outperform CLOSET for min_ℓ greater than or equal to 5, and has similar performance as CHARM.

From this performance study, we conclude that TFP has good overall performance for both dense and sparse datasets. Its running time is nearly constant over a wide range of K and min_ℓ values for dense data. Unlike CHARM and CLOSETwhose performance deteriorates as min_ℓ increases,



a) $K = 100$ b) $L = 8$

Figure 8: Performance on T10I4D100K



a) $K = 100$ b) $K = 500$

Figure 9: Performance on Gazelle

TFP's running time stays low. The reason is inherent from the mining strategy of TFP, CHARM, and CLOSET. In most time, the support for long patterns is lower than that of short patterns. Thus even with the optimal support given, both CLOSET and CHARM are unable to prune short frequent patterns early, thus causing much time spent on mining useless patterns. On the other hand, TFP is able to use the min_ℓ length restriction to cut many short frequent patterns early, thus improves its running time instantly. In addition, TFP does not include any nodes that reside above min_ℓ level to participate in the mining process. As min_ℓ increases, more nodes reside above the min_ℓ level of the tree means that less conditional FP-trees need to be built, thus keeps the running time low.

Besides the good performance over long min_ℓ values, the performance of TFP over short min_ℓ values (even when $min\_\ell = 1$, i.e., no length constraint) is still comparable to that of CLOSET and CHARM. In such cases, the running times between the three do not differ much, and both CLOSET and CHARM were run with the optimal support threshold while TFP was not given any support threshold.

Scalability Test: Our performance tests showed that the running time of TFP increases linearly with increased dataset size.

5. DISCUSSION

In this section, we discuss the related work, how to generate association rules from the mined top-k frequent patterns, and how to push constraints into the mining process.

5.1 Related work

Recent studies have shown that closed patterns are more desirable [5] and efficient methods for mining closed pat-

terns, such as CLOSET [7] and CHARM [8], have been developed. However, these methods all require a user-specified support threshold. Our algorithm does not need the user to provide any minimum support and in most cases runs faster than two efficient algorithms, CHARM and CLOSET, which in turn outperform Apriori substantially [7, 8].

Fu, et al. [2] studied mining N most interesting itemsets for every length l, which is different from our work in several aspects: (1) they mine all the patterns instead of only the closed ones ; (2) they do not have minimum length constraints—since it mines patterns at all the lengths, some heuristics developed here cannot be applied; and (3) their philosophy and methodology of FP-tree modification are also different from ours. To the best of our knowledge, this is the first study on mining top-k frequent closed patterns with length constraint, therefore, we only compare our method with the two best known and well-performed closed pattern mining algorithms.

5.2 Generation of association rules

Although top-k frequent itemsets could be all that a user wants in some mining tasks, in some other cases, s/he wants to mine strong association rules from the mined top-k frequent itemsets. We examine how to do this efficiently.

Items in the short transactions, though not contributing to the support of a top-k itemset of length no less than min_ℓ, may contribute to the support of the items in it. Thus they need to be included in the computation which has minimal influence on the performance. To derive correct confidence, we have the following observations: (1) The support of every 1-itemset is derived at the start of mining. (2) The set of top-k closed itemsets may contain the items forming subset/superset relationships, and the rules involving such itemsets can be automatically derived. (3) For rules in other forms, one needs to use the derived top-k itemsets as probes and the known $min\_support$ as threshold, and perform probe constrained mining to find the support only related to those itemsets. (4) As an alternative to the above, one can set $min\_\ell = 2$, which will derive the patterns readily for all the combinations of association rules.

5.3 Pushing constraints into TFP mining

Constraint-based mining [4, 6] is essential to top-k mining since users may always want to put constraints on the data and rules to be mined. We examine how different kinds of constraints can be pushed into the top-k frequent closed pattern mining.

First, succinct and anti-monotone constraints can be pushed deep into the TFP-mining process. he succint constraints should be pushed deep to select only those itemsets before mining starts and the anti-monotonic constraint should be pushed into the iterative TFP-mining process in a similar way as FP-growth.

Second, for monotone constraints, the rule will also be similar to that in traditional frequent pattern mining, i.e, if an itemset mined so far (e.g., $abcd$) satisfies a constraint "$sum \geq 100$", adding more items (such as e) still satisfies it and thus the constraints checking can be avoided in further expansion.

Third, for convertible constraints, one can arrange items in an appropriate order so that the constraint can be transformed into an anti-monotone one and the anti-monotone constraint pushing can be applied.

Interested readers can easily prove such properties for top-k frequent closed pattern mining.

6. CONCLUSIONS

We have studied a practically interesting problem, mining top-k frequent closed patterns of length no less than min_ℓ, and proposed an efficient algorithm, TFP, with several optimizations: (1) using $closed\_node\_count$ and $descendant\_sum$ to raise $min\_support$ before tree mining, (2) exploring the top-down and bottom-up combined FP-tree mining to first mine the most promising parts of the tree in order to raise $min\_support$ and prune the unpromising tree branches, and (3) using a special indexing structure and a novel closed pattern verification scheme to perform efficient closed pattern verification. Our experiments and performance study show that TFP has high performance. In most cases, it outperforms two efficient frequent closed pattern mining algorithms, CLOSET and CHARM, even when they are running with the best tuned $min\_support$. Furthermore, the method can be extended to generate association rules and to incorporate user-specified constraints.

Based on this study, we conclude that mining top-k frequent closed patterns without $min\_support$ should be more preferable than the traditional $min\_support$-based mining for frequent pattern mining. More detailed study along this direction is needed, including further improvement of the performance and flexibility at mining top-k frequent closed patterns, as well as mining top-k frequent closed sequential patterns or structured patterns.

Acknowledgements. We are grateful to Dr. Mohammed Zaki for providing the code and data conversion package of CHARM and promptly answering many questions .

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. VLDB'94.

[2] A. W.-C. Fu, R. W.-W. Kwong, and J. Tang. Mining n-most interesting itemsets. ISMIS'00.

[3] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. SIGMOD'00.

[4] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. SIGMOD'98.

[5] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. ICDT'99.

[6] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. ICDE'01.

[7] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. DMKD'00.

[8] M. J. Zaki and C. J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. SDM'02.