TFP: An Efficient Algorithm for Mining Top-K Frequent Closed Itemsets

Jianyong Wang, Jiawei Han, *Senior Member*, *IEEE*, Ying Lu, and Petre Tzvetkov

Abstract—Frequent itemset mining has been studied extensively in literature. Most previous studies require the specification of a *min_support* threshold and aim at mining a complete set of frequent itemsets satisfying *min_support*. However, in practice, it is difficult for users to provide an **appropriate** *min_support* threshold. In addition, a complete set of frequent itemsets is much less compact than a set of frequent closed itemsets. In this paper, we propose an alternative mining task: **mining top-*k* frequent closed itemsets of length no less than** *min_l*, where k is the desired number of frequent closed itemsets to be mined, and *min_l* is the minimal length of each itemset. An efficient algorithm, called *TFP*, is developed for mining such itemsets *without mins_support*. Starting at *min_support* = 0 and by making use of the length constraint and the properties of top-*k* frequent closed itemsets, *min_support* can be raised effectively and *FP-Tree* can be pruned dynamically both during and after the construction of the tree using our two proposed methods: the *closed node count* and *descendant_sum*. Moreover, mining is further speeded up by employing a top-down and bottom-up combined *FP-Tree* traversing strategy, a set of search space pruning methods, a fast 2-level hash-indexed result tree, and a novel closed itemset verification scheme. Our extensive performance study shows that *TFP* has high performance and linear scalability in terms of the database size.

Index Terms—Data mining, frequent itemset, association rules, mining methods and algorithms.

1 Introduction

Frequent itemset mining algorithms can be categorized into three classes: 1) Apriori-based, horizontal formatting method, with Apriori [1] as its representative, 2) projection-based, horizontal formatting, pattern growth method, which may explore some compressed data structure such as *FP-tree*, as in *FP-growth* [14], and 3) vertical formatting method, such as *CHARM* [27]. The common *framework* among these methods is to use a *min_support* threshold to ensure the generation of the correct and complete set of frequent itemsets, based on the popular Apriori property [1]: *Every subpattern of a frequent pattern must be frequent* (also called the *downward closure property*). This framework, though simple, leads to the following two problems that may hinder its popular use.

First, to come up with an appropriate *min_support* threshold, one needs to have detailed knowledge about the mining query and the task-specific data, and be able to estimate, without mining, how many itemsets will be generated with a particular threshold. Setting *min_support* is quite subtle: *A too small threshold may lead to the generation of thousands of itemsets, whereas a too big one may often generate no answers*. Our own experience at mining shopping transaction databases tells us that this is by no means an easy task. Finding a good threshold by trial and error is so tedious that a miner may naturally ask: "*I am only interested in finding top-100 frequent itemsets. Is there an easy way to do it without knowing min_support?*"

Second, frequent itemset mining often leads to the generation of a large number of itemsets (and an even larger number of mined rules). Unfortunately, mining a long itemset may unavoidably generate an exponential number of subitemsets due to the downward closure property. Obviously, such a combinatorial explosion problem must be fixed before frequent itemset mining becomes attractive.

The second problem has been noted and examined by researchers recently, proposing to mine (frequent) closed itemsets [22], [23], [7], [3], [27], [25], [8] instead. Since a closed itemset is the itemset that covers all of its subitemsets with the same support, one just needs to mine the set of closed itemsets (often much smaller than the whole set of frequent itemsets), without losing information. For example, a frequent itemset "$a_1, a_2, \ldots, a_{100}$" contains $2^{100} - 1$ frequent subitemsets. By mining closed itemsets, one just needs to generate one itemset "$a_1, a_2, \ldots, a_{100}$" if all of its subitemsets have the same support. Thus, mining closed itemsets should be the default task for mining frequent itemsets.

To solve the first problem, we propose changing the task of *mining frequent itemsets satisfying the min_support* threshold to mining top-*k* frequent closed itemsets of minimum length *min_l*, where k is a user-desired number of frequent closed itemsets to be mined (which is easy to specify), *top-k* refers to the k most frequent closed itemsets, and *min_l* (where $min_l \geq 0$) is the minimal length of closed itemsets. At the first glance, this sounds like changing user's burden from specifying *min_support* to specifying *min_l*. However, there are some fundamental differences. First, *min_l* is usually a small number (such as 2 to 20), which is easy to be specified by users with their applications in mind, whereas *min_support* is often data-dependent and may need to be adjusted over a wide range. Second, the *min_l* constraint

- *J. Wang is with the Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, China. E-mail: jianyong@mail.tsinghua.edu.cn.*
- *J. Han, Y. Liu, and P. Tzetkov are with the Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., 2132 Siebel Center for Science, Urbana, IL 61801-2302. E-mail: {hanj, yinglu, tzvetkov}@uiuc.edu.*

Manuscript received 23 June 2003; revised 23 Apr. 2004; accepted 20 Oct. 2004; published online 17 Mar. 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0103-0603.

gives us freedom to mine only long itemsets without first finding numerous shorter ones, or mine only the itemsets of a specific length. Third, even if one does not want to specify the *min_l* constraint, our top-*k* mining still provides a viable tool for efficient mining without specifying the minimum support constraint by setting *min_l* to 0. And finally, top-*k* frequent itemsets, though not necessarily being the top-*k* quality itemsets (which are usually application-dependent), may be used for further mining of high quality ones.

In this paper, we develop such an efficient algorithm, called *TFP*, that takes advantage of a few interesting properties of top-*k* closed itemsets with minimum length *min_l*, including:

1. Any transactions shorter than *min_l* will not be included in the itemset mining.
2. *min_support* can be raised dynamically in the *FP-Tree* construction, which will help pruning the tree before mining.
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.
4. A set of search space pruning methods and an efficient itemset closure checking scheme are proposed to speed up the closed itemset mining.

Our performance study shows that *TFP* has surprisingly high performance, in most cases, even better than two efficient frequent closed itemset mining algorithms, *CHARM* and *CLOSET+*, with the best tuned *min_support*.

The remainder of the paper is organized as follows: In Section 2, the concept of *top-k closed itemset mining* is introduced, with the problem analyzed and related properties identified. Section 3 presents the algorithm for mining top-*k* closed itemsets. A performance study of the algorithm is reported in Section 4. The related work is discussed in Section 5, and the study is concluded in Section 6.

2 Problem Definition

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of **items**. An **itemset** X is a nonempty 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[1] of an itemset X, denoted as $sup(X)$, is the number of transactions in TDB which contain X.

Definition 1. *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 frequent closed itemset of minimal length** min_l if there exist[2] no more than $(k-1)$ closed itemsets of length at least min_l whose support is higher than that of X.*

1. For convenience of discussion, *support* is defined here as *absolute* occurrence frequency. Notice it is defined in some literature as the relative one, i.e., the occurrence frequency versus the total number of transactions in the transaction database.

2. 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.

TABLE 1
A Transaction Database TDB

TID	Items	Ordered Items
100	d, a, e, g	a, d, e, g
200	b, a	a, b
300	h, a, c, b, e	a, b, c, e, h
400	a, b, c, d	a, b, c, d
500	a, c, b, f, d, i	a, b, c, d, f, i
600	b, a, c	a, b, c
700	f, a, b, i, c, d	a, b, c, d, f, i
800	a, e, c, b	a, b, c, e
900	j	j

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

Example 1. The first two columns in Table 1 show the transaction database TDB in our running example. Suppose our task is to find top-4 frequent closed itemsets with *min_l* = 2. We can find and sort the list of items in support descending order. The sorted item list is called the *sorted_item_list*. In this example, the *sorted_item_list* is $\langle a:8, b:7, c:6, d:4, e:3, f:2, i:2, g:1, h:1, j:1\rangle$. The items in each transaction are sorted according to the *sorted_item_list* order and shown in the third column of Table 1. The set of the top-4 frequent closed itemsets with a minimum length 2 in TDB are

$$\{ab:7, abc:6, ad:4, abcd:3, ae:3\}.$$

Previous studies [14], [25] show that the prefix-tree based algorithms like *FP-growth* are efficient in mining frequent itemsets. "*How can we extend FP-growth for efficient mining of top-k frequent closed itemsets?*" 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 number of higher support closed itemsets during tree construction and dynamically raise *min_support* to prune the tree. 2) One can first mine the most promising subtrees so that high support itemsets can be derived early, which can be used to prune low-support subtrees. 3) We should also design some effective search space pruning methods and efficient itemset closure checking scheme to speed up the mining of closed itemsets. In the following section, we develop the method step by step.

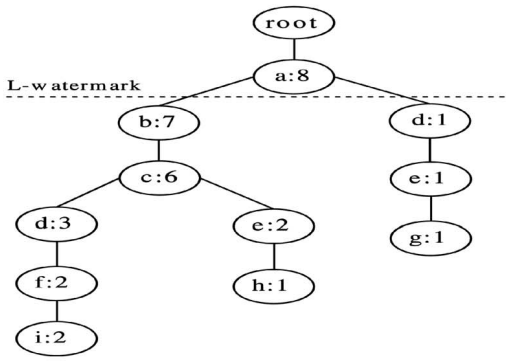
3 Development of Efficient Mining Method

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

3.1 Short Transactions and *l*-counts

It is easy to see that *a transaction containing less than min_l distinct items cannot contribute to the support of an itemset of minimum length min_l*. Thus, the first pruning method is to consider only the transactions that satisfy the minimum length requirement.

Example 2. The *FP-tree* for top-*k* itemsets with *min_l* length is constructed as follows: Scan the TDB once and collect the support (occurrence frequency) for each distinct item,



(a)

Item	support	ℓ-count
a	8	0
b	7	7
c	6	6
d	4	4
e	3	3
f	2	2
i	2	2
g	1	1
h	1	1

(b)

Fig. 1. *FP-Tree* and its header table. (a) The *FP-Tree* constructed from TDB. (b) Support and *l*-count of items.

excluding its counts in short transactions. We get the following *sorted_item_list*:

$$\langle a:8, b:7, c:6, d:4, e:3, f:2, i:2, g:1, h:1\rangle.$$

An *FP-tree* can be constructed using this list as follows [14]: 1) Items in each transaction are arranged according to the order of sorted_item_list and inserted into the *FP-tree* as a branch. 2) If the branch contains a prefix path shared with some existing branch in the tree, the support of each corresponding node in the branch is increased by 1. 3) The remaining (suffix) path is inserted into the tree with the support of each node initiated to 1. The complete *FP-tree* so constructed is in Fig. 1a.

Let the occurrence frequency of each item be stored as *support* in the *(global) header table*. As shown in Fig. 1b, we introduce in 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*.

Remark 3.1 (*l*-count). If the ℓ-count of an item p is lower than *min_support*, p cannot be used as a starting point (i.e., the prefix item) to generate frequent itemset of length no less than *min_l* under the *FP-growth* mining paradigm.

Rationale. Based on the rules for generation of frequent itemset in *FP-growth* [14], 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*. Since short prefix paths will not contribute to the generation of itemset with length greater than or equal to *min_l*, only the items with ℓ-count no less than *min_support* may be used as a starting point to generate frequent itemsets of length no less than *min_l*. However, this does not mean that an item with ℓ-count lower than *min_support* contributes nothing to the set of frequent itemsets. For example, in Fig. 1a, item a cannot be used as a prefix item to generate any frequent itemsets no shorter than *min_l*, however, it can be included in some frequent itemsets like $ba:7$ (It can be mined with prefix item b).

People may wonder that since we start with *min_support* = 0, how we could still use the notion of *min_support*. Notice that if we can find a good number (i.e., no less than k) of closed itemsets with nontrivial support during the *FP-tree* construction or before tree mining, the *min_support* can be raised, which can be used to prune infrequent items from the *FP-tree*.

3.2 Raising *min_support* for Pruning *FP-tree*

Since our goal is to mine top-k frequent closed itemsets, in order to raise *min_support* effectively, one must ensure that the itemsets which are used to raise minimum support must be closed.

Definition 2. *At any time during the construction of an FP-tree, a node n_t is a **closed node** if its support is more than the sum of the supports of its children.*

Let us see our running example. In Fig. 2a, the nodes in the dotted ellipses all fall into the case in Definition 2, as a result, they are all closed nodes.

Property 3.1 (Invariance of a closed node). *Once an FP-tree node n_t becomes a closed node, it will stay a closed node when transactions are added to the FP-tree.*

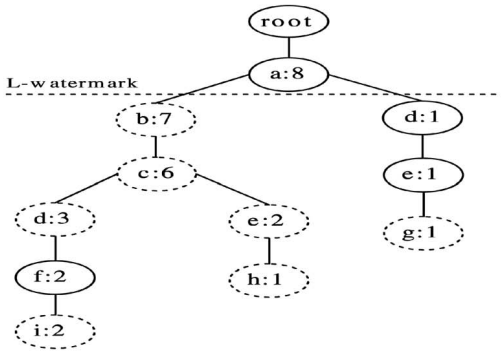
Proof. According to the rules for construction of *FP-tree* [14], any transaction that increments the support of a child of a closed node also increments the support of the closed node itself equally and, hence, its support stays more than the sum of the supports of its children. □

Property 3.2 (Relationship to a closed itemset). *A closed node n_t represents a distinct closed itemset whose support is no less than the support of this closed node.*

Proof. Let the closed node be n_t with a support τ. It is obvious that the prefix path of $n_1 \rightarrow n_2 \cdots \rightarrow n_t$ in the *FP-tree* is unique to node n_t. In the following, we will prove that itemset $S_{n_t} = \{n_1, \ldots, n_t\}$ is a closed itemset.

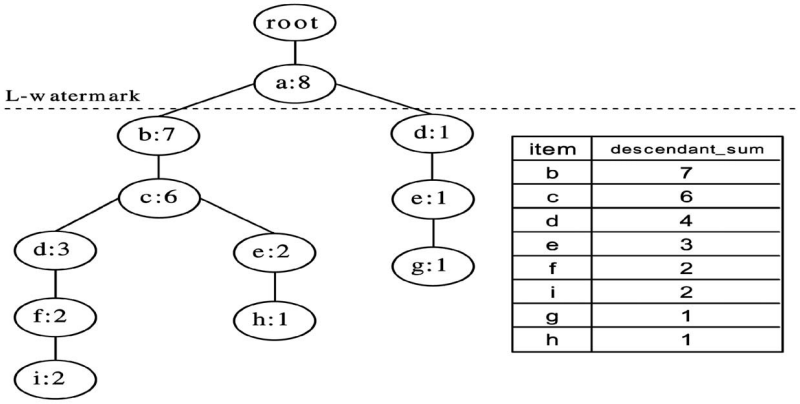
Assume S_{n_t} is not closed, that is, there exists another item i' which does not belong to the set of items $\{n_1, \ldots, n_t\}$ but can be used to extend S_{n_t} and $sup(S_{n_t}) \equiv sup(S_{n_t} \cup \{i'\})$. This means item i' should always occur together with S_{n_t}. Because S_{n_t} contains all the items in node n_t's prefix path, item i' can only be located in the descendant nodes of n_t and the sum of the supports of these descendant nodes labelled with i' should be τ as well, but this contradicts with the definition of a closed node. Thus, itemset S_{n_t} must be closed.

Also, because itemset S_{n_t} can be contained in some other *FP-tree* branches, the final support of itemset S_{n_t} should be no less than τ. □



count	# closed nodes
7	1
6	1
5	0
4	0
3	1
2	2
1	2

Fig. 2. Closed code and closed node count array. (a) Closed nodes in TDB and (b) closed node count array.



item	descendant_sum
b	7
c	6
d	4
e	3
f	2
i	2
g	1
h	1

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

To dynamically raise *min_support* during the *FP-Tree* construction, a simple data structure, called closed_node_count array, can be used to register the current number of the closed nodes under the L-watermark (i.e., at the level no higher than *min_l* in the *FP-tree*) with respect to (abbreviated w.r.t.) a certain count. The array is constructed as follows: Initially, each number of the closed nodes w.r.t. a certain count is initialized to 0. Upon getting a new closed node, the number of closed nodes w.r.t. the count of this newly found closed node will be increased by one. According to Property 3.1, the insertion of a new transaction into the *FP-tree* may increase the count of a closed node from τ to $\tau + 1$. If this happens, we will decrease the number of closed nodes w.r.t. count τ by 1 and, in the meantime, increase the number of closed nodes w.r.t. count $\tau + 1$ by 1. Fig. 2b shows the final status of the closed_node_count array in our running example.

Based on the Properties 3.1 and 3.2 of a closed node and how the *closed_node_count* array is constructed, one can easily derive the following lemma.

Lemma 3.1 (Support raising with *closed_node_count*). *At any time during the construction of an FP-tree, the minimum support for mining top-k frequent closed itemsets will be no less than the corresponding count S if the sum of the number of the closed nodes in closed_node_count array from the top to count S is no less than k.*

In our running example, we can raise the minimum support for mining top-4 frequent closed itemsets to 2 according to Fig. 2b.

Besides using the *closed_node_count* array to raise the minimum support, there is another support raising method with *FP-tree*, called anchor-node descendant-sum, or simply descendant-sum, as described below. An anchor-node is a node at level $min\_l - 1$ 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.

Example 3. As shown in Fig. 3, node $a : 8$ is an anchor node since it resides at level $min\_l - 1 = 1$. At this node, we collect the sum of the supports for each distinct itemset of node $a : 8$'s descendants. For example, since anchor node $a : 8$ has two descendant *d*-nodes, $d : 3$ and $d : 1$, $a : 8$'s *descendant_sum* for d is $d : 4$ (which means that the support of itemset ad contributed from $a : 8$'s descendants is four). From the *FP-tree* presented in Fig. 3, it is easy to figure out that $a : 8$'s *descendant_sum* should be $\{(b : 7), (c : 6), (d : 4), (e : 3), (f : 2), (i : 2), (g : 1), (h : 1)\}$. Such summary information may raise *min_support* effectively. For example, *min_support* for the top-4 frequent closed itemsets should be at least 3 based on $a : 8$'s *descendant_sum*.

Notice that if *descendant_sum* contains two or more identical support value, only one of them will contribute to the support of closed itemsets because it is possible that the nodes with identical support can be merged to form one longer closed itemset. For example, since $af : 2$ and $ai : 2$ are actually subitemsets of $abcdfi : 2$, they can only be counted as one closed itemset with support= 2. Thus, by examining only the *descendant_sum* of one

anchor node $a:8$, we can derive that *min_support* of the top-5 frequent closed itemsets should be at least 2, while *min_support* of the top-6 frequent closed itemsets should be at least 1.

Lemma 3.2 (*descendant_sum*). *Each distinct support in descendant_sum of an anchor node represents the minimum support of one distinct closed itemset.*

Proof. Let the path from the root of the *FP-tree* to an anchor node b be β and the set of items in β be S_β . Let $count_i$ be a distinct count in *descendant_sum* of the anchor node b and the list of descendant nodes whose descendant_sum count equals $count_i$ be $S_i = \langle i_1, i_2, \ldots, i_k \rangle$ $(k \geq 1)$, which are sorted according to the *sorted_item_list*. Assume the subset of items in S_i which co-occur with i_1 $count_i$ times under the anchor node b is S_{i_1}, then $S_\beta \cup S_{i_1}$ forms a new itemset and has a support $count_i$ w.r.t. anchor node b. Similar to the proof of Property 3.2, we can easily prove that $S_\beta \cup S_{i_1}$ is a closed itemset with a support no less than $count_i$ (Due to limited space, the proof is left to the interested readers). Because no two such itemsets (i.e., $S_\beta \cup S_{i_1}$) are identical, each itemset, $S_\beta \cup S_{i_1}$, represents a distinct closed itemset. □

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 $(min\_l - 1)$ 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 support. 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 support information to raise *min_support* effectively. Computing *descendant_sum* structure for low support 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 a *closed_node_count* array which raises *min_support*, dynamically prunes some infrequent items and their corresponding *FP-tree* nodes, and reduces the size of the *FP-tree* to be constructed. Note some closed nodes may disappear, but this does not affect the correctness of the support raising method because the support of a removed closed node must be lower than the current minimum support and contributes nothing to the current minimum support. After scanning the database (i.e., the *FP-tree* is constructed), we traverse the subtree of the level $(min\_l - 1)$ 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 $(min\_l - 1)$ nodes, the remaining node with the highest support will be traversed, and this process continues until there is no remaining level $(min\_l - 1)$ node that has a support higher than the current *min_support*. Based on our experiments, only a small number of nodes need to be so traversed if k for top-*k* is not very large. Note the worst case of this heuristic is to traverse all the level $(min_l - 1)$ nodes whose supports are no smaller than 2, thus the overhead is no greater than scanning the whole *FP-tree* once. Compared to the whole mining process, it is marginal and can be neglected in many cases.

3.3 Efficient Mining of *FP-Tree* for top-*k* Itemsets

The raise of *min_support* effectively prunes the *FP-tree* and speeds up the mining. However, efficient mining strategy, search space pruning methods, and itemset closure checking scheme are also critical to the overall performance.

3.3.1 Mining Strategy

The *FP-growth* algorithm adopts a totally bottom-up *FP-tree* searching order to mine the whole set of frequent itemsets given a user-specified support threshold. However, this mining strategy may not be good for mining the top-*k* most frequent closed itemsets. There are two subtle points for the *TFP* algorithm.

1. "Top-down" ordering of the items in the global header table for the generation of conditional *FP-tree*s, where a *conditional FP-tree* of an item p is the *FP-tree* constructed with the set of p's prefix paths in the *FP-tree* [14]. The first subtlety is in what order the conditional *FP-Tree*s should be generated for top-*k* mining. For top-*k* mining, our goal is to *find only the itemsets 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 ℓ-count that is no smaller than the current minimum support 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 itemsets with higher support. With this ordering, *min_support* can be raised faster and the top-*k* itemsets can be discovered earlier. In addition, an item with ℓ-count less than *min_support* does not have to generate conditional *FP-tree* for further mining (as stated in Remark 3.1). Thus, the faster the *min_support* can be raised, the earlier pruning can be done.
2. "Bottom-up" ordering of the items in a local header table for mining conditional *FP-tree*s. The second subtlety is how to mine conditional *FP-tree*s. We have shown that the generation of conditional *FP-tree*s 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 *FP-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 itemsets first then followed by shorter ones. It is more efficient to first generate long closed itemsets since the itemsets containing only the subset items can be absorbed by them easily. More importantly, as we will see in

Section 3.3.3, an efficient itemset closure checking scheme can be easily designed based on the combination of the "top-down" ordering of the items in the global header table and the "bottom-up" ordering of the items in the local header tables.

3.3.2 Search Space Pruning Methods

To accelerate the top-k frequent closed itemset mining, several search space pruning techniques which have never been used in several previous studies have been adopted, including the *item merging* [23], [10], [27] and the *prefix-itemset skipping* [5], [23], [10].

For any prefix itemset X, after its conditional database (i.e., projected *FP-tree*) has been built, we can find its local frequent items by scanning it once, where a *conditional database* of an itemset X is the database consisting of the set of X's prefix paths in the *FP-tree* and its *local frequent items* are the set of frequent items of the conditional database (see [14] for details). If some local frequent items have the same support as their prefix itemset X, we can use Remark 3.2 to prune search space.

Remark 3.2 (Item merging). For any prefix itemset X and its local frequent item set S, assume S_X is the set of items in S with the same support as X. The items in S_X should be merged with X to obtain a new prefix X' with local frequent item set $S' = (S - S_X)$, that is, items in S_X can be safely removed from the local frequent item list of X'.

Proof. Since every item in S_X has the same support as X, any subset of S_X will appear in every transaction X appears. Thus, every itemset, X'', obtained by growing X with a proper subset of S_X will have the same support as itemset $X' = (X \cup S_X)$. Because X' is a proper superset of X'', X'' must be nonclosed, and any frequent closed itemset with a prefix itemset X'' can be obtained directly by growing X', which means we can safely prune the items in S_X from the local frequent item set of X'. □

Before extending a prefix itemset X to generate frequent closed itemsets, we should first check if there is another already found frequent closed itemset Y, which is a proper superset of X with the same support. If that is the case, we should avoid growing X based on Remark 3.3.

Remark 3.3 (Prefix-itemset skipping). At any time for a certain prefix itemset X, if there is an already found frequent closed itemset Y, and $(X \subset Y) \wedge (sup(X) = sup(Y))$ holds, there is no hope to generate frequent closed itemsets with prefix X.

Proof. Because $(X \subset Y) \wedge (sup(X) = sup(Y))$ holds, X is not a frequent closed itemset, and X and Y appear in the same set of transactions. For any frequent closed itemset Z grown from X, if we use itemset $(Z - X)$ to grow Y, $((Y \cup (Z - X)) \supseteq Z) \wedge (sup(Y \cup (Z - X)) = sup(Z))$ must hold and based on the mining strategy described in Section 3.3.1, we know any closed itemset with prefix Y must have been mined before we mine the closed itemsets with prefix X. That is, any new itemset grown from X with $(Z - X)$ is not closed. □

3.3.3 Itemset Closure Checking Scheme

Because we are only interested in mining the top-k frequent closed itemsets, it is important to efficiently maintain the set of already mined frequent closed itemset candidates and assure that every final top-k frequent closed itemset is really closed.

During the mining process, a pattern-tree is used to keep the set of current frequent closed itemset candidates. 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 itemset verification (to be discussed below), thus we retain this item ordering in the itemsets mined. The major difference between *FP-tree* and pattern-tree is that the former stores transactions in compressed form, whereas the latter stores potential frequent closed itemsets.

The bottom-up mining of the conditional *FP-tree*s generates itemsets in such an order: For itemsets that share prefixes, longer itemsets are generated first. In addition, there is a total ordering over the itemsets generated. This leads to our development of the **frequent closed itemset verification scheme**, 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 nonzero l-count item and i_j be the item whose conditional *FP-tree* is currently being mined. Then, the set of already mined closed itemsets, 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 itemset p during the mining of i_j's conditional tree, we need to perform new itemset checking (checking against S_{i_j}) and old itemset checking (checking against S_{old}).

The new itemset checking is performed as follows: Since the mining of the conditional tree is in a bottom-up manner and the *item merging* pruning technique has been applied, just like *CLOSET* [23], we need to check whether 1) p is a subitemset of another itemset 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 itemset checking, p becomes a new closed itemset with respect to S. Note that because itemsets in S_{old} do not contain item i_j, there is no need to check if p is a subitemset of the itemsets in S_{old}.

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

Let $prefix(p)$ be the prefix itemset of an itemset p (i.e., obtained by removing the last item i_j from p). The correctness of the above checking is shown in the following lemmas.

Lemma 3.3 (Old itemset checking). *For old itemset checking, we only need to check if there exists an itemset $prefix(p)$ in S_{old} with $supp(prefix(p)) \equiv supp(p)$.*

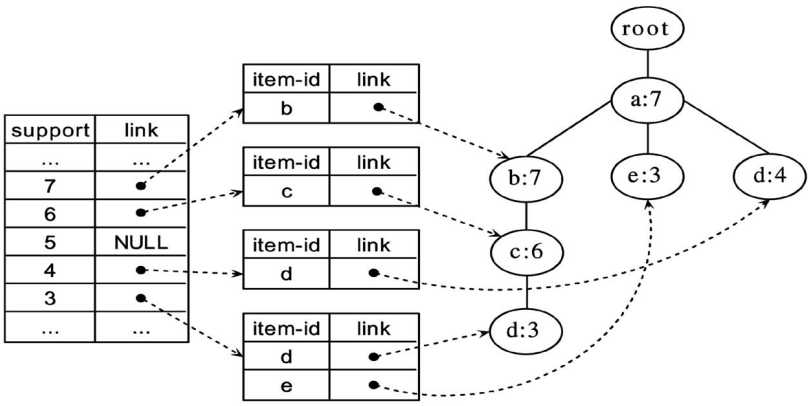


Fig. 4. Two-level indexing for verification of closed itemsets.

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

Lemma 3.4 (Support raise). *If a newly mined itemset p can pass both new itemset checking and old itemset checking, then it is safe to use p to raise min_support.*

Proof. From Lemma 3.3, there will be two possibilities for p. First, it is a real closed itemset, i.e., it will not be absorbed by any itemsets later. Second, it will be absorbed by a later found itemset, and this itemset can only absorb itemset p. In this case, we will not use the later found itemset to raise support because it has already been used to raise support when we found itemset p (or p's precedents). Thus, it is safe to use p to raise *min_support*.□

To accelerate both new and old itemset checking, we introduce a two-level index header table into the pattern-tree structure. Notice that if an itemset can absorb (or be absorbed by) another itemset, the two itemsets must have same support. Thus, our first index is based on the support of an itemset. In addition, for new itemset checking, we only need to check if itemset p can be absorbed by another itemset that also contains i_j; however, for old itemset 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 itemset as the index key. At each itemset-tree node, we also record the length of the itemset, in order to judge if the corresponding itemset needs to be checked.

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

Example 4 (Closed itemset verification). Fig. 4 shows the two-level indexed result-tree structure for verification of closed itemsets in our running example. The closed itemsets in the result tree are generated in the following order: $ab:7$, $abc:6$, $abcd:3$, $ad:4$, and $ae:3$.

Based on the above 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 itemset candidate is in the tree. Assume $abcd:3$ is the newly mined itemset, by following indices of support 3 and item-id c into the result tree, we find it cannot absorb its prefix abc, this because abc has a different support (i.e., 6), although it is a closed itemset. Also, following indices of support 3 and item-id d into the result tree, we cannot find any other closed itemsets which contain item d and can absorb $abcd:3$. Thus, it is safe to use the support of $abcd:3$ to raise the minimum support.

3.4 Algorithm

ALGORITHM 1 shows the TFP algorithm that mines the set of top-*k* frequent closed itemsets. It first builds the *FP-tree* from the input database DB: If the number of frequent items in an input transaction T is no less than the minimal length, T is inserted into *FP-tree*, in the mean time, it uses the *closed node count* method to raise minimum support, *min_sup*, and uses the raised *min_sup* to prune infrequent items from the *FP-tree* (line 5). After the *FP-tree* has been constructed, the *descendant_sum* method is adopted to further raise *min_sup* and prune *FP-tree* (lines 6-8). Then, we begin the mining process: Top-down traverse each item in the global header table. If the corresponding item's *l_count* is no less than the current *min_sup* (line 10), treat this item as a frequent prefix itemset, build conditional *FP-tree* (together with its header table) for it (line 11), and call subroutine Mine_cond_FP tree() to mine the frequent closed itemsets (line 12). Finally, output the top-*k* frequent closed itemsets from the result pattern tree by traversing it in a bottom-up manner and in the support descending order (line 13).

ALGORITHM 1: $\mathbf{TFP}(DB, K, min\_l, FCI^k)$
INPUT: an input database DB, an integer K, and the minimal length threshold min_l.
OUTPUT: the complete set of top-K frequent closed itemsets, FCI^k.
01. $FCI^k = \phi$; $min\_sup = 0$; $FPtree = NULL$;
    $ResultTree = NULL$;
02. *for each transaction t in DB*
03. $t' = t$ - *set of infrequent items w.r.t. the current*

min_sup;
04. $if(length(t' \geq min\_l))$
05. $insert\_tree(t, FPtree)$; $closed\_node\_count(min\_sup)$; $pruneTree(FPtree, min\_sup)$;
06. $find\_and\_sort\_anchor\_nodes(FPtree)$; *//in count descending order*
07. *for each anchor-node N whose support is greater than min_sup do*
08. $Descendant\_sum(N, min\_sup)$; $pruneTree(FPtree, min\_sup)$;
09. *for (each item i in global header table) do //top-down traversing*
10. $if(l\_count(i) \geq min\_sup)$
11. $cond\_FPtree^i = build\_conditional\_FPtree(i, FPtree)$;
12. *call* $Mine\_cond\_FPtree(cond\_FPtree^i, i, min\_l, K, ResultTree)$;
13. FCI^k=top-K frequent closed itemsets in ResultTree;

Mine_cond_FPtree() (see SUBROUTINE 1) recursively calls itself and works as follows: For prefix I_p, it uses the item_merging technique to absorb its locally frequent items with the same support (line 14), which leads to a new prefix I'_p. If I'_p cannot pass the new_itemset_checking, we can stop mining closed itemsets with prefix I'_p according to the *prefix_itemset_skipping* technique (line 16). If I'_p can pass the old_itemset_checking, then the support of I'_p can be used to raise the *min_sup* (line 17). Finally, by bottom-up traversing, the locally frequent items in the local header table w.r.t. prefix I'_p Mine_cond_FPtree() will recursively call itself (lines 19-22).

SUBROUTINE 1: **Mine_cond_FPtree** ($cond\_FPtree^{I_p}$, I_p, $min\_l, K, ResultTree$)
INPUT: projected FPtree $cond\_FPtree^{I_p}$, prefix itemset I_p, minimal length min_l, integer K, and pattern tree ResultTree.
OUTPUT: the current set of top-K frequent closed itemsets, FCI^k.
14. $LFI^{I_p} = local\_frequent\_item(I_p, cond\_FPtree^{I_p})$; $I'_p = item\_merging(I_p, LFI^{I_p})$; $LFI^{I'_p} = LFI^{I_p} - (I'_p - I_p)$;
15. *if* $(length(I'_p) \geq min\_l)$
16. if $(!(new\_pattern\_checking(I'_p, ResultTree)))$ $return$; *//nonclosed, apply prefix_itemset_skipping method*
17. $if(old\_pattern\_checking(I'_p, ResultTree))$ $support\_raise(min\_sup, K)$;
18. $insert\_ResultTree(I'_p, ResultTree)$;
19. *for each item j in* $LFI^{I'_p}$*// bottom-up traversing* $cond\_FPtree^{I_p}$
20. $if(l\_count(j) \geq min\_sup)$
21. $I''_p = I'_p \cup \{j\}$; $cond\_FPtree^{I''_p} = build\_conditional\_FPtree(I''_p, min\_l, cond\_FPtree^{I_p})$;
22. *call* $Mine\_cond\_FPtree(cond\_FPtree^{I''_p}, I''_p, min\_l, K, ResultTree)$;

4 Experimental Evaluation

In this section, we report our performance study of *TFP* over a variety of data sets. In particular, we compared the efficiency of *TFP* with two well-known algorithms for mining frequent closed itemsets: *CHARM* [27] and *CLOSET+* [25]. However, one should note that different from these *min_support*-based methods, *TFP* represents a new class of algorithms which do not require user's knowledge of *min_support*. To give the best possible credit to *CHARM* and *CLOSET+*, our comparison was always based on assigning the best tuned *min_support* to the two algorithms so that they can generate the same top-*k* closed itemsets for a user-specified k value (under a condition of *min_l*). In practice, we should bare in mind that, although this is natural for *TFP*, it is a difficult task for *min_support*-based algorithms to speculate a proper *min_support*. In the experiments, we turned off the output for all the three algorithms. In addition, we also studied the scalability of *TFP* and evaluated the effects of two support raising methods and two search space pruning techniques employed in *TFP*.

To obtain the best possible *min_support* for *CHARM* and *CLOSET+*, our experiments were conducted as follows: For each experimental condition of *min_l* and k, *TFP* was first run to get the optimal support for the generation of required itemsets, and this support was then used to run the other two algorithms. By doing so, we can compare the running time of *TFP* with no *min_support* against that of *CHARM* and *CLOSET+* running with the optimal *min_support*.

The experiments show that

1. The running time of *TFP* is shorter than *CLOSET+* and *CHARM* in most cases when *min_l* is not too short, and is comparable in other cases.
2. *TFP* has nearly linear scalability.
3. The search space pruning techniques are very effective in enhancing the performance.
4. The support raising methods are effective in raising the minimum support.

4.1 Data Sets

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

4.1.1 Dense Data Sets that Contain Many Long Frequent Closed Itemsets

1. *pumsb* census data, which consists of 49,046 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 data sets are obtained from the UC-Irvine Machine Learning Database Repository.

4.1.2 Sparse Data Sets

1. *gazelle* 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 itemsets (obtained from BlueMartini Software Inc.) and

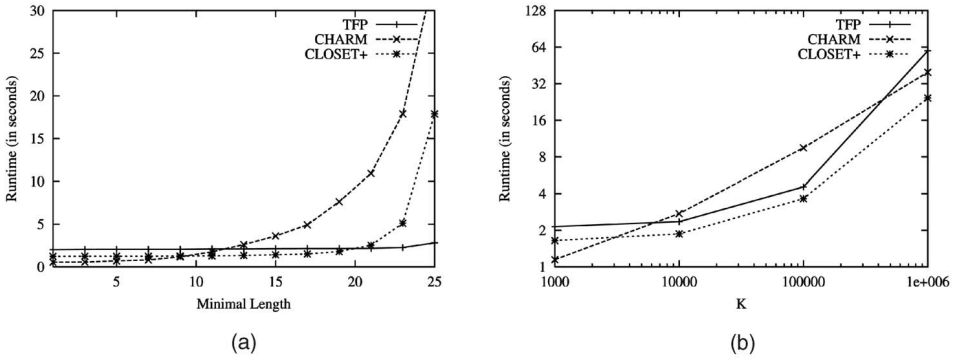


Fig. 5. Performance on Connect-4. (a) $k = 500$. (b) $min\_l = 0$.

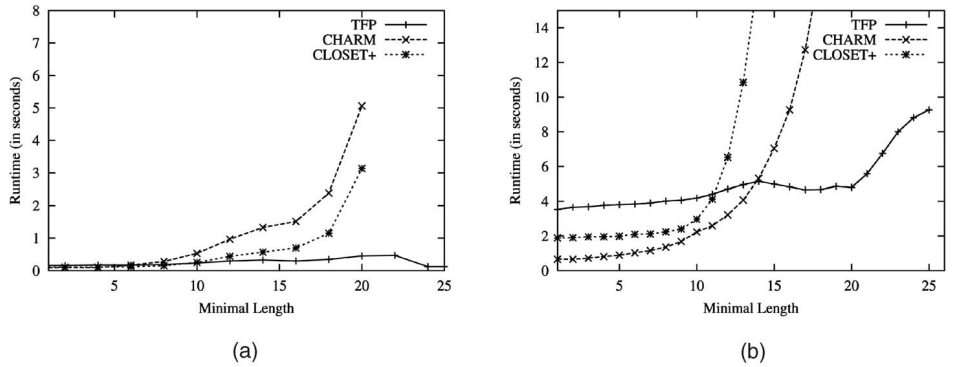


Fig. 6. Performance on (a) mushroom and (b) pumsb ($k = 500$).

2. *T10I4D100K* synthetic data from the IBM data set generator, which consists of 100,000 transactions with an average length of 10 items, and with many closed frequent itemsets having average length of 4.

4.2 Performance Results

All the experiments were conducted on a 1.7GHz Pentium-4 PC with 512MB of memory, running Windows 2000. The *CHARM* code was provided to us by its author. We compared the performance of *TFP* with *CHARM* and *CLOSET+* on the five data sets by varying *min_l* and k.

4.2.1 Dense Data Sets

For the dense data sets with many long closed itemsets, *TFP* performs consistently better than *CHARM* and *CLOSET+* for longer *min_l*. Note because the transactions in each of such data sets have the same length (43, 74, and 23 for *connect*, *pumsb*, and *mushroom*, respectively), we cannot do any preprocessing for *TFP*.

Fig. 5a shows the running time of the three algorithms on the connect-4 data set for k fixed at 500 and *min_l* ranging from 0 to 25. We observe that the running time of *TFP* remains stable over the range of *min_l*. When *min_l* reaches 20, *TFP* starts to outperform all the other three algorithms. Fig. 5b shows the running time of the three algorithms on the connect-4 data set with *min_l* set to 0 (which means *TFP* generates the same number of itemsets as the other two algorithms) and k ranging from 1,000 to 1,000,000. We can see that, even without the *min_l* constraint, *TFP* runs almost as fast as the other two algorithms for the very large k values. Because *TFP* starts with a minimum support 0, this shows that *TFP* finds the correct minimum support very fast for this dense data set.

Fig. 6 shows the running time of the three algorithms on the mushroom and pumsb data sets with k set to 500 and *min_l* ranges from 0 to 25. For the mushroom data set, 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 the other two algorithms when *min_l* is larger than 10. Pumsb has very similar results as connect-4 and mushroom data sets.

4.2.2 Sparse Data Set

For large *min_l*, our experiments show similar results with sparse data sets as those with dense ones: *TFP* outperforms both *CHARM* and *CLOSET+*. Due to the limited space, we do not report them here. Instead, we show the performance results with the *min_l* fixed at 0, in order for *TFP* to generate the same set of frequent itemsets as *CHARM* and *CLOSET+*. Experiments show that, even without the *min_l* constraint, *TFP* can gain comparable performance with *CHARM* and *CLOSET+*.

Fig. 7a shows the running times of the three algorithms on T10I4D100K data set with *min_l* fixed at 0. We first ran *CHARM* and *CLOSET+* with *min_support* ranging from 0.09 percent to 0.05 percent to get the number of frequent itemsets and use this number as the k value to run *TFP* (k is greater than 15,000 for each of these support thresholds, which is very large for a sparse data set like T10I4D100k). The result in Fig. 7a shows that *TFP* is only about 2-3 times slower than *CHARM* and *CLOSET+*, although *TFP* starts with *min_support* 0 and the k value is set to very large. For example, at *min_support* 0.05 percent, *CHARM* used about 6.6 seconds to finish and generated 49,122 frequent itemsets,

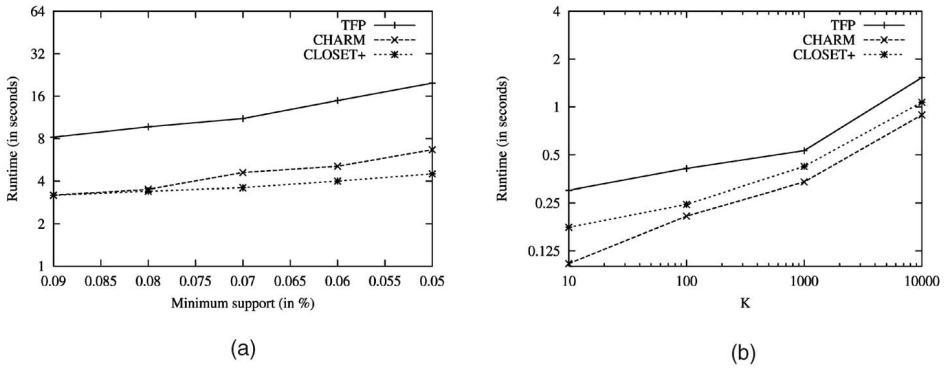


Fig. 7. Performance on (a) T10I4D100K and (b) Gazelle (*min_l* = 0).

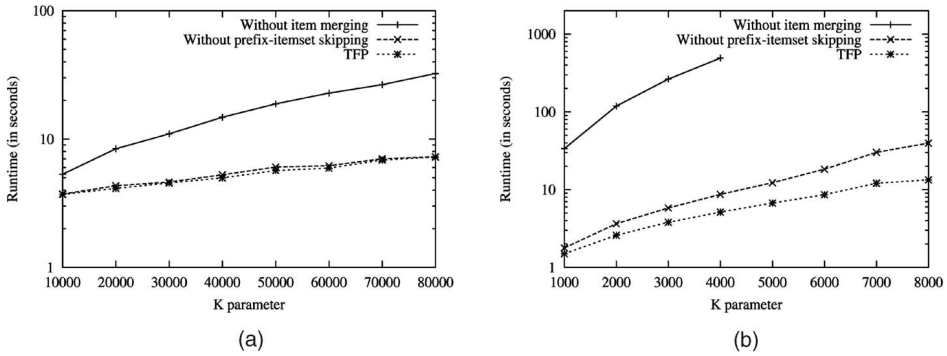


Fig. 8. Search space pruning method evaluation $(min\_l = 10)$. (a) Connect-4 data set. (b) Gazelle data set.

while the runtime of *TFP* is about 19.7 seconds with k set at 49,122.

The experiments on the gazelle data set are shown in Fig. 7b. Here, we fixed *min_l* at 0, and varied k from 10 to 10,000. We can see that all the three algorithms work well for this data set. For example, at $k = 10,000$, which corresponds to a very low *min_support*, 0.000789, the runtime of *TFP*, *CHARM*, and *CLOSET+* are 1.532 seconds, 0.892 seconds, and 1.071 seconds, respectively.

From the above performance study, we conclude that *TFP* has good overall performance for both dense and sparse data sets. Because *TFP* can push the *min_l* constraint deeply into the mining process, it outperforms two efficient closed itemset mining algorithms a lot. Some new techniques and mining strategies proposed here, like the *closed-node-count* and *descendant-sum* support raising methods, the combination of the top-down and bottom-up mining strategy, and the efficient itemset closure checking scheme, make *TFP* achieve comparable efficiency with *CHARM* and *CLOSET+* for $min\_l = 0$. In this case, *TFP* mines the same set of itemsets as the traditional closed itemset mining algorithms which can be used to generate association rules. We also noticed that in some cases, the derived top-*k* frequent closed itemsets contains much overlap, i.e., the itemsets may be clustered into groups of similar itemsets (a group of similar itemsets usually come from one region of the FP-tree and this usually happens when k is not large). However, this is intrinsic to many frequent itemset mining problem formulations. For example, all subitemsets of a long frequent itemset are also frequent, which implies that the itemsets mined by *CHARM* or *CLOSET+* also contains a lot of overlap. On the other hand, the phenomenon of the itemset "clustering" may reflect the nature the data sets and may be useful in some applications. For example, if the data set contains a set of documents with a similar topic, then each group of the similar top-*k* itemsets may form a *Micro Concept*, i.e., a potential core of one of the natural clusters in these documents. The further compression of such closely related frequent patterns into an even smaller set of core concepts will be an interesting research topic beyond this paper.

Effectiveness of Search Space Pruning Methods. We also tested the effectiveness of the *item merging* and *prefix-itemset skipping* techniques using both dense and sparse data sets. From Fig. 8a, we can see that without the *item merging* technique, *TFP* can be several times slower while the *prefix-itemset skipping* technique can only marginally improve the performance for dense data set connect-4. Fig. 8b shows that both the *item merging* and *prefix-itemset skipping* techniques are very effective at improving the *TFP* performance for sparse data set Gazelle: The *TFP* algorithm can be several times (or more than an order of magnitude) slower without the *prefix-itemset skipping* (or *item merging*).

Effectiveness of the Support Raising Methods. We also tested the effectiveness of the two support raising methods: *closed-node-count* and *descendant-sum*. Fig. 9a shows the results for data set T10I4D100K by varying the k parameter from 2 to 1,024 and *min_l* fixed at 5. The y-axis depicts the support that can be raised by both methods, the final absolute support for different k parameters, and the number of the traversed anchor nodes. Both *closed-node-count* and *descendant-sum* methods are very effective at raising the support threshold, especially when the k parameter is not very large. For example, for $k = 2$, the *closed-node-count* method can raise the minimum support to 59 and the

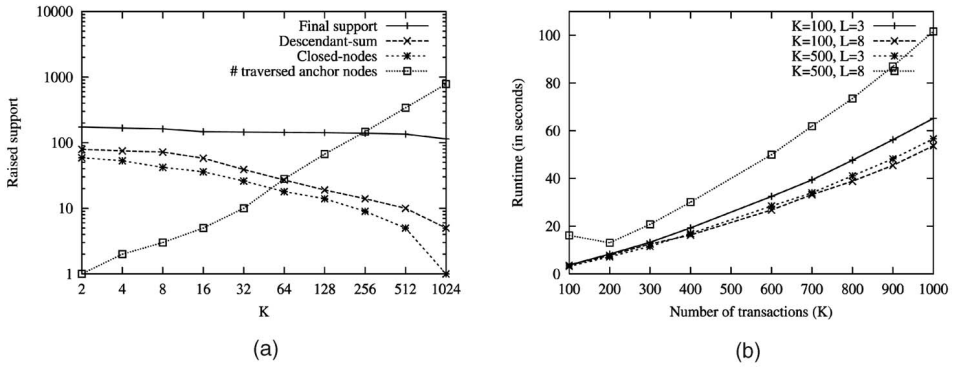


Fig. 9. (a) Support raising method evaluation and (b) scalability test (T10I4D100K data set series).

descendant-sum method can raise the minimum support to 79 by traversing only one anchor node while the final support threshold for the top-2 frequent closed itemsets is only 173. We also see that the *descendant-sum* method is a little more effective at raising the support threshold than the *closed-node-count* method because the *closed-node-count* method can be used to raise minimum support and prune the *FP-tree* during the construction of *FP-tree* and is cheaper (in implementation) than the *descendant-sum* method. Both methods are used in the *TFP* algorithm. In addition, when k becomes larger, the *descendant-sum* method needs to traverse more nodes in order to raise the support. However, because the worst case is to scan the global *FP-tree* once, this will not cause too much overhead compared with the whole mining process.

Scalability Test. In order to test the scalability of *TFP*, we generated synthetic data sets with the same characteristics as T10I4D100K, but with the size ranging from 100K to 1000K transactions. Fig. 9b shows the running time of *TFP* for four different combinations of small and big k and *min_l* values. The figure shows that the running time of *TFP* increases linearly with increased data set size in all four cases.

5 Discussion

Frequent itemset mining has been studied extensively in data mining. Recent studies [22], [27], [23], [25] have shown that it is more desirable to mine closed itemsets than the complete set of frequent itemsets. Efficient methods for mining closed itemsets, such as *CLOSET* [23], *CHARM* [27], and *CLOSET+* [25], have been developed. However, these methods all require a user-specified support threshold. Hidber presented Carma, an algorithm for online association rule mining [15], in which, a user can change the support threshold any time during the first scan of the data set (in other words, Carma still needs the user to specify the final support threshold), but its performance is worse than Apriori in general. In comparison with Carma, our algorithm does not need users to provide any minimum support and, in most cases, runs faster than two efficient algorithms, *CHARM* and *CLOSET+* (running at the best tuned *min_support* thresholds), which, in turn, outperform Apriori substantially [27], [25]. Recently, there are proposals on association rule mining without support requirement [11], [26], which are aimed at discovering confident rules instead of significant rules. As a result, they only use the confidence threshold to prune rules of small confidence. Our motivation is different because our algorithm still targets at mining significant rules, but we do not need a user to specify any *min_support* threshold.

The problem of mining top-k frequent itemsets has attracted the attention of some researchers recently. Fu et al. [12] studied mining N most interesting itemsets for every length l, which is different from our work in several aspects:

1. they mine all the itemsets instead of only the closed ones, and mining closed itemsets is not only more desirable but also more challenging;
2. they do not have minimum length constraints—since it mines itemsets 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 itemsets with length constraint, therefore, we only compare our method with two well-known efficient closed itemset mining algorithms.

From the user-interaction point of view, since our performance study shows that there is no real need to specify *min_l* if one wants to mine frequent closed itemsets of any length, and also there is no crucial need to specify k for top-k mining as long as k is a default number that fits user's expectation or application requirements, this method gives the user the minimal burden to specify mining parameters, representing a step toward *parameter-free* frequent-pattern mining.

There are extensive studies on mining frequent itemsets from many different angles, such as constraint-based mining [20], [6], [4], [19], mining generalized and quantitative rules [2], [13], and mining correlation rules [9], [18], [24], [16]. Our study on mining top-k frequent itemsets is orthogonal to these studies. Since their mining and optimization frameworks are based on a predefined *min_support* threshold, the techniques developed in this study can be extended to the scope of these studies to improve their corresponding algorithms for mining top-k frequent itemsets. We also expect that the basic principles developed here can be applied to recently developed new frequent itemset mining algorithms, such as [21], [17], when the requirement is changed to mining top-k frequent itemsets. Finally, it is expected that the philosophy developed here will influence the mining of top-k frequent

structured patterns, where a structured pattern may contain sequences, trees, lattices, and graphs.

6 Conclusions

We have studied the problem of mining top-*k* frequent closed itemsets of length no less than *min_l*. This is an interesting problem because the task arises naturally from the difficulty of specification of appropriate minimum support thresholds at mining various kinds of frequent itemsets.

In this study, we have proposed an efficient algorithm, *TFP*, which includes several techniques, such as:

1. using *closed node count array* and *descendant_sum* to raise minimum support before tree mining,
2. exploring the top-down *FP-tree* mining technique to first mine the most promising parts of the tree in order to raise minimum support and prune the unpromising part of the tree during the *FP-tree* mining process,
3. adopting several search space pruning methods to speed up the closed itemset mining, and
4. using an efficient itemset closure verification scheme to check if a frequent itemset is promising to be closed.

Our performance studies on both real and synthetical data sets show that *TFP* has high performance. In most cases, it outperforms two efficient frequent closed itemset mining algorithms, *CHARM* and *CLOSET+*, even when they are running with the best tuned minimum support. Furthermore, the method can be extended to generate association rules and to incorporate user-specified constraints.

Based on this study, we claim that for frequent itemset mining, mining top-*k* frequent closed itemsets without minimum support should be more preferable than the traditional minimum support-based mining. There are many interesting research issues along this direction, including further improvement of the performance and flexibility for mining top-*k* frequent closed itemsets, as well as mining top-*k* frequent closed itemsets in data stream environments and mining top-*k* frequent closed sequential or structured patterns.

Acknowledgements

The authors are grateful to Dr. Mohammed Zaki for providing them with the source code of *CHARM* and the vertical data conversion package, as well as promptly answering many questions related to *CHARM*. This work was supported in part by US National Science Foundation NSF IIS-02-09199 and IIS-03-08215, the University of Illinois, and Microsoft Research. This paper is a major-value added version of a conference paper that appeared in the 2002 IEEE International Conference on Data Mining (ICDM '02).

References

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. 1994 Int'l Conf. Very Large Data Bases (VLDB '94),* pp. 487-499, Sept. 1994.

[2] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proc. 1995 Int'l Conf. Data Eng. (ICDE '95),* pp. 3-14, Mar. 1995.

[3] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal, "Mining Frequent Patterns with Counting Inference," *SIGKDD Explorations,* pp. 66-75, vol. 2, 2000.

[4] S.D. Bay and M.J. Pazzani, "Detecting Change in Categorical Data: Mining Contrast Sets," *Proc. 1999 Int'l Conf. Knowledge Discovery and Data Mining (KDD '99),* pp. 302-306, Aug. 1999.

[5] R.J. Bayardo, "Efficiently Mining Long Patterns from Databases," *Proc. 1998 ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '98),* pp. 85-93, June 1998.

[6] F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi, "Exante: Anticipated Data Reduction in Constrained Pattern Mining," *Proc. Seventh European Conf. Principles and Practice of Knowledge Discovery in Databases (PKDD '03),* Sept. 2003.

[7] J.-F. Boulicaut and A. Bykowski, "Frequent Closures As a Concise Representation for Binary Data Mining," *Proc. 2000 Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD '00),* pp. 62-73, Apr. 2000.

[8] J.-F. Boulicaut, A. Bykowski, and C. Rigotti, "Free-Sets: A Condensed Representation of Boolean Data for the Approximation of Frequency Queries," *Data Mining and Knowledge Discovery,* vol. 7, pp. 5-22, 2003.

[9] S. Brin, R. Motwani, and C. Silverstein, "Beyond Market Basket: Generalizing Association Rules to Correlations," *Proc. 1997 ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '97),* pp. 265-276, May 1997.

[10] D. Burdick, M. Calimlim, and J. Gehrke, "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases," *Proc. 2001 Int'l Conf. Data Eng. (ICDE '01),* pp. 443-452, Apr. 2001.

[11] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J.D. Ullman, and C. Yang, "Finding Interesting Associations without Support Pruning"*Proc. 2000 Int'l Conf. Data Eng. (ICDE '00),* pp. 489-499, Feb. 2000.

[12] A.W.-C. Fu, R.W.-W. Kwong, and J. Tang, "Mining n-Most Interesting Itemsets," *Proc. 2000 Int'l Symp. Methodologies for Intelligent Systems (ISMIS '00),* pp. 59-67, Oct. 2000.

[13] J. Han and Y. Fu, "Discovery of Multiple-Level Association Rules from Large Databases," *Proc. 1995 Int'l Conf. Very Large Data Bases (VLDB '95),* pp. 420-431, Sept. 1995.

[14] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *Proc. 2000 ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '00),* pp. 1-12, May 2000.

[15] C. Hidber, "Online Association Rule Mining," *Proc. 1999 ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '99),* pp. 145-156, June 1999.

[16] Y.-K. Lee, W.-Y. Kim, Y.D. Cai, and J. Han, "CoMine: Efficient Mining of Correlated Patterns," *Proc. 2003 Int'l Conf. Data Mining (ICDM '03),* Nov. 2003.

[17] G. Liu, H. Lu, W. Lou, and J.X. Yu, "On Computing, Storing, and Querying Frequent Patterns," *Proc. 2003 ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '03),* Aug. 2003.

[18] S. Morishita and A. Nakaya, "Parallel Branch-and-Bound Graph Search for Correlated Association Rules," *Large-Scale Parallel Data Mining,* pp. 127-144, 1999.

[19] S. Morishita and J. Sese, "Traversing Itemset Lattice with Statistical Metric Pruning," *Proc. 2000 ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS '00),* pp. 226-236, May 2001.

[20] R. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang, "Exploratory Mining and Pruning Optimizations of Constrained Associations Rules," *Proc. 1998 ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '98),* pp. 13-24, June 1998.

[21] F. Pan, G. Cong, A.K.H. Tung, J. Yang, and M. Zaki, "CARPENTER: Finding Closed Patterns in Long Biological Datasets," *Proc. 2003 ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '03),* Aug. 2003.

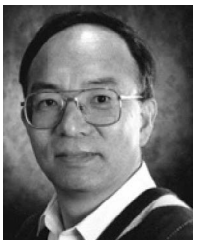
[22] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering Frequent Closed Itemsets for Association Rules," *Proc. Seventh Int'l Conf. Database Theory (ICDT '99),* pp. 398-416, Jan. 1999.

[23] J. Pei, J. Han, and R. Mao, "CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets," *Proc. 2000 ACM-SIGMOD Int'l Workshop Data Mining and Knowledge Discovery (DMKD '00),* pp. 11-20, May 2000.

[24] F. Rioult, J.-F. Boulicaut, B. Cremileux, and J. Besson, "Using Transposition for Pattern Discovery from Microarray Data," *Proc. Eighth ACM SIGMOD Workshop Research Issues in Data Mining and Knowledge Discovery,* June 2003.

[25] J. Wang, J. Han, and J. Pei, "CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets," *Proc. 2003 ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '03)*, pp. 236-245, Aug. 2003.
[26] K. Wang, Y. He, D. Cheung, and F. Chin, "Mining Confident Rules without Support Requirement," *Proc. 2001 ACM CIKM Int'l Conf. Information and Knowledge Management (CIKM '01)*, pp. 81-88, Nov. 2001.
[27] M.J. Zaki and C.J. Hsiao, "CHARM: An Efficient Algorithm for Closed Itemset Mining," *Proc. 2002 SIAM Int'l Conf. Data Mining (SDM '02)*, pp. 457-473, Apr. 2002.

Jianyong Wang received the PhD degree in computer science in 1999 from the Institute of Computing Technology, the Chinese Academy of Sciences. Since then, he has worked as an assistant professor in the Department of Computer Science and Technology, Peking (Beijing) University in the areas of distributed systems and Web search engines, and has visited the School of Computing Science at Simon Fraser University and the Department of Computer Science at the University of Illinois at Urbana-Champaign as a postdoc research fellow, mainly working in the area of data mining. He was a research associate of the Digital Technology Center at the University of Minnesota from July 2003 to November 2004. Since January 2005, he has been an associate professor in the Department of Computer Science and Technology, Tsinghua University, China.



Jiawei Han received the PhD degree in computer science from the University of Wisconsin in 1985. He is a professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. Previously, he was an Endowed University Professor at Simon Fraser University, Canada. He has been working on research into data mining, data warehousing, stream data mining, spatial and multimedia data mining, deductive and object-oriented databases, and bio-medical databases, with more than 250 journal and conference publications. He has chaired or served in many program committees of international conferences and workshops, including ACM SIGKDD Conferences (2001 best paper award chair, 2002 student award chair, 1996 PC cochair), SIAM-Data Mining Conferences (2001 and 2002 PC cochair), ACM SIGMOD Conferences (2000 exhibit program chair), ICDE Conferences (2004, 2002, and 1995 PC vice-chair), and ICDM Conferences (2005 PC cochair). He also served or is serving on the editorial boards of several journals and transactions. He is currently serving on the Board of Directors for the Executive Committee of ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD). Dr. Han has received three IBM Faculty Awards, the Outstanding Contribution Award at ICDM (2002), ACM Service Award (1999), ACM Fellow (2004), and ACM SIGKDD Innovations Award (2004). He is the first author of the textbook *Data Mining: Concepts and Techniques* (Morgan Kaufmann, 2001). He is a senior member of the IEEE and the IEEE Computer Society.

Ying Lu is currently a PhD student at the University of Illinois at Urbana-Champaign. Her research interests are in data mining, bioinformatics, and stream processing.

Petre Tzvetkov received the MS degere in computer science from the University of Illinois under the supervision of Professor Jiawei Han. He is currently an IT manager and software architect at MOST Computers Ltd. His main professional and research interests are in data mining, databases, and software engineering. He has coauthored several research papers in the area of data mining.