Information Sciences 179 (2008) 559–583

Contents lists available at ScienceDirect

Information Sciences

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

Efficient single-pass frequent pattern mining using a prefix-tree

Syed Khairuzzaman Tanbeer, Chowdhury Farhan Ahmed, Byeong-Soo Jeong *, Young-Koo Lee

Department of Computer Engineering, Kyung Hee University, 1 Seochun-dong, Kihung-gu, Youngin-si, Kyunggi-do 446-701, Republic of Korea

ARTICLE INFO

Article history:
Received 10 March 2008
Received in revised form 24 October 2008
Accepted 27 October 2008

Keywords:
Data mining
Frequent pattern
Association rule
Incremental mining
Interactive mining

ABSTRACT

The FP-growth algorithm using the FP-tree has been widely studied for frequent pattern mining because it can dramatically improve performance compared to the candidate generation-and-test paradigm of *Apriori*. However, it still requires two database scans, which are not consistent with efficient data stream processing. In this paper, we present a novel tree structure, called CP-tree (compact pattern tree), that captures database information with one scan (*insertion phase*) and provides the same mining performance as the FP-growth method (*restructuring phase*). The CP-tree introduces the concept of dynamic tree restructuring to produce a highly compact frequency-descending tree structure at runtime. An efficient tree restructuring method, called the branch sorting method, that restructures a prefix-tree branch-by-branch, is also proposed in this paper. Moreover, the CP-tree provides full functionality for interactive and incremental mining. Extensive experimental results show that the CP-tree is efficient for frequent pattern mining, interactive, and incremental mining with a single database scan.

1. Introduction

Finding frequent patterns (or *itemsets*) plays an essential role in data mining and knowledge discovery techniques, such as association rules, classification and clustering. A large number of studies [2,3,15,12,32,22,33,1,13,24,11] have been published presenting new algorithms or improvements on existing algorithms to solve the frequent pattern mining problem more efficiently. The candidate generation-and-test methodology, called the *Apriori* technique [2], was the first technique to compute frequent patterns based on the *Apriori* principle and the anti-monotone property, i.e. if a pattern is found to be frequent then all of its non-empty subsets will be frequent. In other words, if a pattern is not frequent, then none of its supersets can be frequent. The *Apriori* technique finds the frequent patterns of length k from the set of already generated candidate patterns of length $k-1$. Therefore, this algorithm requires multiple database scans equal to the size of the maximum length of frequent patterns in the worst case. Moreover, it requires a large amount of memory to handle the candidate patterns when the number of potential frequent patterns is reasonably large.

Introduction of prefix-tree based frequent pattern mining [12,13,15,20,10,24] efficiently addressed the problems of numerous candidate patterns, time complexity, and multiple database scans for *Apriori* algorithms. Prefix-tree based techniques follow a pattern growth approach that avoids candidate generation and test costs by generating a frequent pattern from its prefix. It constructs a conditional database for each frequent pattern X, denoted as D_X. All patterns that have X as a prefix can be mined from D_X without accessing other information. The mining performance of the pattern growth approach is closely related to the number of conditional databases constructed during the whole mining process and the construction/traversal cost of each conditional database. The former depends on which order the frequent items are arranged in a

* Corresponding author. Tel.: +82 31 201 2932; fax: +82 31 202 1723.
E-mail addresses: tanbeer@khu.ac.kr (S.K. Tanbeer), farhan@khu.ac.kr (C.F. Ahmed), jeong@khu.ac.kr, jeong_khu@yahoo.com (B.-S. Jeong), yklee@khu.ac.kr (Y.-K. Lee).

0020-0255/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2008.10.027

prefix-tree. The latter depends on the representation and traversal strategy of conditional databases. If many transactions can share their prefixes in a prefix-tree, mining performance will improve. The FP-growth mining technique proposed by Han et al. [15] has been found to be an efficient algorithm when using the prefix-tree data structure. The performance gain achieved by FP-growth is mainly based on the highly compact nature of the FP-tree, where it stores only the frequent items in frequency-descending order and ensures the tree can maintain as much prefix sharing as possible among patterns in the transaction database. However, the construction of such an FP-tree requires two database scans and prior knowledge about the support threshold, which are the key limitations of applying the FP-tree in a data stream environment or for incremental and interactive mining.

To escape from the requirement for two database scans in the FP-growth mining technique, we need to devise an efficient summary structure to contain all database information with one scan, and pattern mining should be efficiently performed on this structure without an additional database scan. For the summary data structure, the prefix-tree might be a good candidate even though it may suffer from memory-size limitation when the number of transactions is too large and the length of a transaction is too long. However, as the available memory increases beyond GBytes, several prefix-tree data structures, which capture all the database information in a tree, have been proposed for mining frequent patterns with one database scan [29,22,10,24]. CanTree, proposed in [24], captures complete database information in a prefix-tree structure, while facilitating incremental and interactive mining via an FP-growth mining technique. Although CanTree requires only one database scan, it usually results in less compact tree size and poor mining performance compared to the FP-tree due to the frequency-independent canonical order of item insertion. Then the question arises, *can we efficiently construct a prefix-tree structure with one database scan that captures the full database information, inherits the compactness of the FP-tree, and yields better mining performance?* In this paper, we propose such a novel tree structure, called the CP-tree (compact pattern tree), that can construct an FP-tree like prefix-tree structure with one database scan and can provide the same mining performance as the FP-growth technique through the efficient tree restructuring process.

The main idea of our CP-tree is to periodically reorganize the prefix-tree in frequency-dependent item order after inserting some transactions into the prefix-tree using the previous item order. Through repeated reorganization, our CP-tree can maintain as much prefix sharing as possible in a prefix-tree and provide better mining performance as a result. However, as you may see, the tree reorganization overhead might not be trivial since our CP-tree size is large enough to hold complete database information. Thus, we devise several tree restructuring techniques and analyze their performance. We also suggest techniques for choosing the appropriate time to reorganize a prefix-tree for optimal performance. Our comprehensive experimental results on both real and synthetic datasets show that frequent pattern mining with our CP-tree outperforms the state-of-the-art algorithms in terms of execution time and memory requirements. Furthermore, the experimental results also show that the CP-tree provides better performance than existing algorithms for interactive and incremental mining.

In summary, our main contributions in this work are

- We propose a novel, highly compact tree structure that significantly improves the performance of frequent pattern mining with a single database scan.
- We propose a phase-by-phase tree restructuring method, which improves the degree of prefix sharing in the tree structure and provides the criteria for choosing an appropriate restructuring method.
- We observe the performance characteristics of a pattern-growth mining approach through extensive experimental study.
- We show that the CP-tree can be useful for other frequent pattern mining applications like incremental mining and interactive mining.

The rest of the paper is organized as follows: in Section 2, we summarize the existing algorithms to solve the frequent pattern mining, incremental, and interactive mining problems. Section 3 describes the problem definition of frequent pattern mining and presents preliminary knowledge. The structure and restructuring process of the CP-tree are given, in detail, in Section 4. This section also explains several performance issues regarding the pattern growth mining technique with the CP-tree. Our proposed tree restructuring method is also introduced in Section 4. Section 5 summarizes some additional applications of the CP-tree. We report our experimental results in Section 6. Finally, Section 7 concludes the paper.

2. Related work

Mining frequent patterns in a static database has been well-addressed over the last decade. The first known frequent pattern mining algorithm is *Apriori* [2], which was proposed in 1993. Since then a number of algorithms have been proposed for improving the performance of *Apriori*-based algorithms [3,8,14,32,21,27]. The main performance problem of *Apriori*-like approaches comes from the requirement of multiple database scans and a large number of candidate patterns, many of which are proved to be infrequent after scanning the database. This restricts use of these methods to incremental, interactive, and data stream mining. To overcome this problem, Han et al. [15] proposed the frequent pattern tree (FP-tree) and the FP-growth algorithm that can reduce the number of database scans by two and eliminate the requirement of candidate generation. The highly compact FP-tree structure introduced a new wing of research in mining frequent patterns with a prefix-tree structure. Many studies have been performed based on the FP-tree, either focusing on its performance improvement [12,17,18] or proposing new areas [20,31,25] of its applicability. However, the static nature of the FP-tree still limits its appli-

cability to incremental and interactive frequent pattern mining, and its two database scan requirement prevents its use in frequent pattern mining on a data stream.

Many algorithms have been proposed to discover frequent patterns in incremental databases and to maintain association rules in dynamically updated databases [6,9,16,20,26,24,33,23,34,28]. FUP2 [9] and UWEP [30] are based on the framework of the *Apriori*-algorithm. The tree-based AFPIM [20], EFPIM [26] and FUFP-tree [16] algorithms perform incremental mining by using a compact data structure, mainly by adjusting the FP-tree structure. These approaches still require two database scans for both the original (to construct the FP-tree structure) and incremented portions (to update the tree structure). Moreover, in the first two approaches an additional error bound called *pre-minimum support* is considered to maintain the potential frequent patterns. While updating the tree structure, FUFP-tree constantly maintains the initial sort order of the items. Thus it adds the new frequent items (i.e. items which were previously infrequent and become frequent due to an update of the database) at the end of a list it constructs in order to maintain the sort order and the frequency of each item, and in the tree structure as new leaves. Therefore, the resultant tree structure usually loses its frequency-descending property when more frequent items in the original database become comparatively less frequent in the updated database or when a new item becomes more frequent than the item(s) in the existing list. This problem may become more serious if the items in the initial tree structure occur rarely compared to the newly added frequent items when the database is incremented or updated. Furthermore, these three methods may require re-scanning of the updated (for AFPIM and EFPIM) or the original (for FUFP-tree) database when any infrequent pattern becomes frequent after the database is updated; otherwise the occurrence information of the pattern in the original database cannot be obtained and, as a result, they lose the property of complete mining.

The drawbacks of AFPIM and EFPIM are well-addressed by the CATS tree [10] that requires only one scan of the original database and eliminates the need to scan the whole updated database during incremental mining by scanning the incremented portion of the database only once. Even though the CATS tree shows a compact representation of a database like FP-tree, its efficiency in incremental mining is not clear because of its complex tree construction process. Rather than incremental mining, it is well suited for interactive mining where the database remains unchanged and only the minimum support threshold varies.

In [24], Leung et al. proposed the CanTree that captures the contents of a transaction database with a single-pass and stores them in a prefix-tree in canonical order. It follows the FP-growth based divide-and-conquer approach to discover frequent patterns. The simple tree construction process of CanTree enables it to outperform AFPIM and CATS tree in incremental and interactive mining. However, since it is not similar to the FP-tree in compactness (or not maintaining frequency-descending order of items during tree construction or afterward), the mining phase takes more time than the FP-tree approach when the number of frequent patterns in a database is reasonably large.

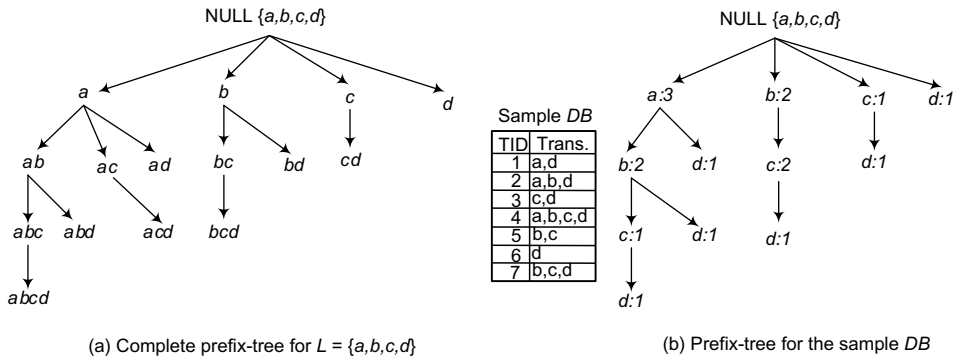
In summary, FP-tree is a highly compact tree structure that enables highly efficient mining. However, it only handles the frequent items in a database and it is a two-pass solution. On the other hand, CanTree offers a single-pass solution which maintains complete database information suitable for incremental and interactive mining. However, it incurs very high mining time due to the canonical order of its tree structure. Investigating and analyzing the above scenario, we propose the CP-tree, which efficiently resolves the drawbacks of these methods and offers a highly competent frequent pattern mining solution for incremental and interactive mining. CP-tree guarantees FP-growth mining performance on the prefix-tree of an entire database in a memory efficient manner.

3. Preliminaries

In this section, we describe the definitions of key terms that explain the concepts of frequent pattern mining more formally. Let $L = \{i_1, i_2, \ldots, i_n\}$ be a set of literals, called items, that have been used as information units in an application domain. A set $P = \{i_1, \ldots, i_k\} \subseteq L$, where $k \in [1,n]$, is called a *pattern* (or an *itemset*), or a k-itemset if it contains k items. A transaction $t = (tid, Y)$ is a tuple where *tid* is a transaction-id and Y is a pattern. If $P \subseteq Y$, it is said that t contains P or P occurs in t. A transaction database DB over L is a set of transactions and $|DB|$ is the size of DB, i.e. the total number of transactions in DB. The support of a pattern P in a DB, denoted as $Sup(P)$, is the number of transactions in DB that contain P. A pattern is called a *frequent pattern* if its support is no less than a user given minimum support threshold *min_sup* ∂, with $0 \leqslant \partial \leqslant |DB|$. The *frequent pattern mining problem*, given a ∂ and a DB, is to discover the complete set of frequent patterns in a DB having support no less than ∂. F_{DB} refers to a set of all frequent patterns in a DB under a given *min_sup*.

A *prefix-tree* is an ordered tree structure which can represent a transaction database in a highly compressed form. It is constructed by reading transactions one at a time with a predefined item order and mapping each transaction onto a path in the prefix-tree. Since different transactions can have several items in common, their paths may overlap.

The more the paths overlap with one another, the more compression we can achieve using a prefix-tree structure. More formally, given a set of items L, we put a total order $\leqslant_x$ on the items in L, where x represents item ordering. A pattern $P = \{i_{a_1}, i_{a_2}, \ldots, i_{a_m}\}$ is said to be ordered (or sorted) if $\forall j \in [1, m-1], i_{a_j} \leqslant_x i_{a_{j+1}}$. Given two ordered patterns $p_1 = \{i_{a_1}, i_{a_2}, \ldots, i_{a_m}\}$ and $p_2 = \{i_{b_1}, i_{b_2}, \ldots, i_{b_k}\} (m \leqslant k)$, if $\forall j \in [1, m], i_{a_j} = i_{b_j}$, then p_1 is called a prefix of p_2. If $k = m + 1$, then p_1 is called an immediate prefix of p_2. The *root* of the prefix-tree represents the empty set. Each node in the tree represents a *pattern* (or an *itemset*). There is an edge between a pattern and its immediate prefix. Each node at level k (the *root* being at level 0 and its children at level 1, and so on) represents a k-itemset. Fig. 1a shows the complete prefix-tree for a set of items $L = \{a, b, c, d\}$



(a) Complete prefix-tree for $L = \{a,b,c,d\}$ (b) Prefix-tree for the sample *DB*

Fig. 1. Prefix-tree search space with an example.

sorted in lexicographic order and Fig. 1b represents one example of a prefix-tree which was constructed from the sample database. For every pattern p in the tree, only the items after the last item of p can be appended to p to form a longer pattern. Such a set of items is called the *candidate extensions* of p. For instance, items c and d are candidate extensions of ab, while b is not a *candidate extension* of ac, because b is before c. If extending a frequent pattern p by one of its *candidate extensions* leads to a frequent pattern, then that *candidate extension* is called a *frequent extension* of p. The frequency count of each node represents the total occurrence of the *itemset* in the respective path in the tree. One basic property of a prefix-tree is that the frequency count of any node is always greater than or equal to the sum of frequency counts of its children.

In the worst case, the maximum number of nodes in a prefix-tree can be 2^k, where k is the number of distinct items. In other words, the maximum number of all possible patterns in a transaction database is 2^k. This size is too large to contain a transaction database in one prefix-tree even when the value of k is only 100. However, in a real scenario, very few patterns appear in the database when the value of k is large. Table 1 shows a sketch of such a picture by reporting the number of nodes (tree size) of different prefix-trees constructed from different datasets while varying the number of distinct items, the number of transactions, and the average transaction length (we assume that the item occurrence in a transaction follows the *Zipf's* distribution). The table reflects that the tree size varies significantly due to the variation in the average transaction length and the number of transactions, while the number of distinct items has less effect on the tree size. This indicates that very few of the patterns among the 2^k patterns normally appear in a database.

When we apply such a prefix-tree to store a real world transaction database, where the number of transactions is very large and the length of a transaction is relatively long, we need to carefully consider the prefix-tree structure so that it can be fitted into main memory. The size of such a prefix-tree depends on the degree of prefix sharing among all patterns in the tree. As mentioned before, the higher the prefix sharing is, the more compact the tree structure is. Again, the amount of prefix sharing depends on how specifically the tree search space is explored, i.e. the order of items. For example, if the most frequent items are arranged at the upper region of the prefix-tree, then the possibility of obtaining higher prefix sharing and compactness increase as a result. On the other hand, a prefix-tree in a canonical order (e.g. lexicographical order) fails to ensure such compactness although it might be easier to construct the tree. The size of such a tree might be unmanageably large when there are huge items with low frequency in the database.

Since the prefix-tree size will severely affect the mining performance, it is very important to keep a prefix-tree as small as possible. *Tree restructuring* is meant to reorganize a prefix-tree with items ordered so that it can have as much prefix sharing as possible. Since the frequency-descending order of items can provide higher degree of prefix sharing, during restructuring this sort order can be followed to reorder the prefix-tree. Such restructuring should not only result in a smaller prefix-tree, but also provide complete mining results. In Section 4.3, we deal with the problem of *tree restructuring*.

4. Overview of CP-tree: construction and performance issues

In this section, we first introduce our CP-tree, and then describe several issues related to its performance. We also analyze the complexity of CP-tree construction based on different tree restructuring approaches, and suggest some heuristics for deciding the time of restructuring and choosing the restructuring method.

Table 1
Tree size variation on different database parameters.

	Number of nodes			
Average transaction length	10		20	
Number of distinct items	2K	10K	2K	10K
Number of transactions = 100K	4,22,081	4,39,920	8,76,241	9,15,680
Number of transactions = 200K	8,22,303	8,62,974	17,64,077	18,09,169

4.1. CP-tree construction

The FP-growth technique, which mines frequent patterns without candidate generation, requires two *DB* scans to achieve a highly compact frequency-descending tree structure (FP-tree). During the first scan, it derives a list of frequent items in which items are ordered in frequency-descending order. According to the frequency-descending list, the second *DB* scan produces a frequent-pattern tree (FP-tree), which can store compact information on transactions involving frequent patterns. Although a requirement for two *DB* scans might not be a problem for mining over a static data set, it is a problem for incremental, interactive, and stream data mining. The CP-tree, on the contrary, builds an FP-tree like compact frequency-descending tree structure with a single-pass of a transaction database. At first, transactions are inserted into the CP-tree according to a predefined item order (e.g. lexicographical item order) one by one. The item order of a CP-tree is maintained by a list, called an *I-list*. After inserting some of the transactions, if the item order of the *I-list* deviates from the current frequency-descending item order (*I-list* maintains the current frequency value of each item) to a specified degree, the CP-tree is dynamically restructured by the current frequency-descending item order and the *I-list* updates the item order with the current list.

In summary, CP-tree construction mainly consists of two phases:

(i) *Insertion phase*: scans transaction(s), inserts them into the tree according to the current item order of *I-list* and updates the frequency count of respective items in the *I-list*.
(ii) *Restructuring phase*: rearranges the *I-list* according to frequency-descending order of items and restructures the tree nodes according to this newly rearranged *I-list*.

These two phases are dynamically executed in alternate fashion, starting with the *insertion phase* by scanning and inserting the first part of *DB*, and finishing with the *restructuring phase* at the end of *DB*. We describe our proposed tree restructuring method and compare its performance characteristics with the existing technique in the remaining part of this section. Another objective of the periodic tree restructuring technique is to construct a frequency-descending tree with reduced overall restructuring cost.

We use an example to illustrate the construction of a CP-tree from the original *DB*. Fig. 2 shows a transaction *DB* (Fig. 2a) with corresponding transaction *ID*s, and a step-by-step CP-tree construction procedure (Fig. 2c–g). For simplicity in this example, we assume that the tree is restructured after inserting every three transactions. However, in the following part of this section we propose other restructuring criteria, which may be effective depending on the size and type of dataset. We also assume that the tree construction begins by inserting the transactions of the first slot in the predefined item order (e.g. lexicographical item order). Initially, the CP-tree is empty which means it starts with a '*null*' *root* node. Like an FP-tree, a CP-tree contains nodes representing an itemset and the total number of passes (i.e. support) of that itemset in the path from the *root* up to that node. It follows the FP-tree construction technique to insert any sorted transaction into the tree. For simplicity in the figures, we do not show the node traversal pointers in the trees; however, they are maintained in a fashion similar to the FP-tree. We also denote the *I-list*s of the same items representing frequency-independent and frequency-descending item ordering by I and I_{sort}, respectively.

The construction of a CP-tree starts with an *insertion phase*. The first *insertion phase* begins by inserting the first transaction {*b*, *a*, *e*} into the tree in a lexicographical item order. Fig. 2d shows the exact structures of the tree and *I-list* after inserting

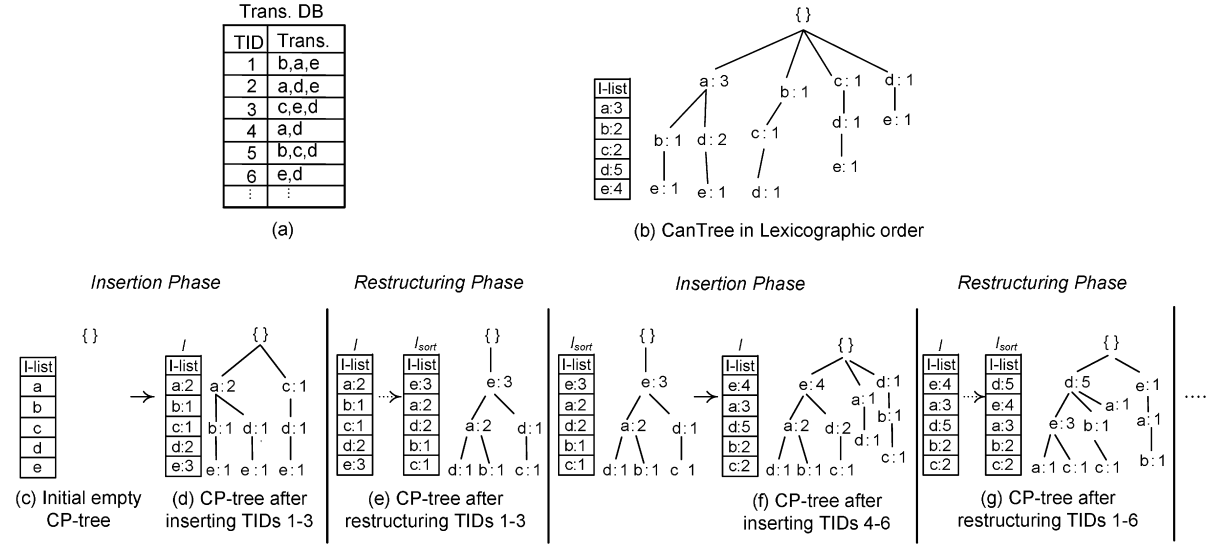


Fig. 2. Construction of CP-tree and comparison with CanTree.

transactions 1, 2, and 3, which correspond to the first *insertion phase*. Since the tree will be restructured after every three transactions, the first *insertion phase* ends here and the first *restructuring phase* starts immediately. Since items so far are not inserted in frequency-descending order, the CP-tree at this stage is like a frequency-independent tree with a lexicographical item order. To rearrange the tree structure, first, the item order of the *I-list* is rearranged in frequency-descending order, and then the tree is restructured according to the new *I-list* item order. Fig. 2e shows the change in *I-list* and restructured CP-tree after this stage. It can be noted that items having a higher count value are arranged at the upper most portion of the tree; therefore, the CP-tree at this stage is a frequency-descending tree offering higher prefix sharing among patterns in tree nodes. The first *restructuring phase* terminates when the full process of *I-list* rearrangement and tree restructuring are completed. The CP-tree construction will enter into the next *insertion phase*, thereafter.

During the next *insertion phase* all transactions will be inserted following the new sort order of the *I-list* {*e*, *a*, *d*, *b, c*} instead of previous item order {*a*, *b*, *c*, *d*, *e*}. The updated CP-tree and the modified status of the *I-list* after passing the second *insertion phase* including transactions 4, 5, and 6 are shown in Fig. 2f. The tree may again lose its property of compactness due to new transaction insertion in the second *insertion phase*. Therefore, the tree is restructured again to become a frequency-descending tree. The next *restructuring phase* will result in a new *I-list* and CP-tree structure as in Fig. 2g. The CP-tree repeats this procedure until all transactions of a database are inserted into a tree.

Similar to an FP-tree, the CP-tree constructed in the previous example can achieve a highly compact tree structure while maintaining complete *DB* information. The CP-tree improves the possibility of prefix sharing among all the patterns in *DB* with one *DB* scan. That is, more frequently occurring items are more likely to be shared and thus they are arranged closer to the *root* of the CP-tree. This item ordering enhances the compactness of the CP-tree structure. The frequency-descending tree may not always provide maximum compactness. Sometimes, the insertion order of transactions may affect the tree size. However, on the whole, for a CP-tree, the frequency-descending tree provides better compactness, which it achieves with one *DB* scan. Like the FP-tree and CanTree, the size of a CP-tree is bounded by the size of the *DB* itself because each transaction contributes at most one path of its size to the CP-tree. However, since there are usually many common prefix patterns among the transactions, the size of the tree is normally much smaller than its *DB*. Since CanTree does not guarantee frequency-descending item order in a tree, generally the size of a CP-tree will be smaller than that of a CanTree. As shown in Fig. 2b and Fig. 2g, the CP-tree contains 11 nodes, while a lexicographic CanTree contains 14 nodes for the database in Fig. 2a. Our experimental study also shows the same results as described in Section 6. Throughout the construction of the CP-tree, we observe the following property and lemmas.

Property 1. *The total frequency count of any node in the CP-tree is greater than or equal to the sum of the total frequency counts of its children.*

Lemma 1. *Given a transaction database DB, the complete set of all item projections of all transactions in the database can be derived from DB's CP-tree.*

Proof. Based on the CP-tree construction mechanism, all item projections of each transaction are mapped to only one path in the CP-tree, and any path from the *root* up to an item maintains the complete projection for exactly *n* transactions, where *n* is the difference between the count of the item itself and the count summation for all of its children nodes in the path. Therefore, CP-tree maintains a complete set of all item projections of each transaction for the transaction database only once. □

Lemma 2. *Given a transaction database DB and a CP-tree constructed on it, the search space during mining F_{DB} for any value of support threshold ∂ can be started from the bottom most item in the I-list having count value $\geqslant \partial$ to the first item upward in the I-list.*

Proof. Let *X* be the bottom most item in the *I-list* of a CP-tree having support count $\geqslant \partial$. The *I-list* of restructured CP-tree maintains the items in frequency-descending order. Therefore, the support counts of all items under *X* are less than *Sup*(*X*), assuring no item under *X* is frequent. So, mining F_{DB} for *DB* can be carried out by considering only *X* and its upper items in the *I-list*. □

Once a CP-tree is constructed, based on above property and lemmas, F_{DB} can be mined from it using an FP-growth mining technique for any value of ∂. In other words, we can employ a divide-and-conquer approach. Based on Lemma 2, projected databases can be formed by traversing the paths upwards only starting from the last frequent item in the *I-list*. Since items are dynamically and consistently arranged in frequency-descending order, one can guarantee the inclusion of all frequent items using just upward traversals. There is no scope for including *global*[1] infrequent items or double counting of items.

4.2. Tree restructuring criteria

One of the two primary factors that affect the performance of a CP-tree construction is effectively deciding when to begin the *restructuring phase*. Too many or too few restructuring operations may lead to poor performance. Therefore, to gain a

[1] Item *Y* is a *global* infrequent item if *Sup*(*Y*) in *I-list* is less than the *min_sup* value.

higher overall performance, it is required to effectively formulate the switching to the *restructuring phase*. In this subsection, we suggest three different heuristics to determine the switching to the *restructuring phase*, namely *N*% technique, Top-*K* technique, and *N*-*K* technique.

N% *technique*: the user supplies a value of *N*. The *restructuring phase* will be called after each *N*% transactions of *DB*, and at the end of *DB* if required. The example of the CP-tree constructed in the previous subsection demonstrates this technique. Although this approach is simple, it is important to choose the value of *N* in such a way that it does not lead to poor performance. From our experience we have found that $N = 10$ shows acceptable results for different types of *DB*.

Top-K technique: the user supplies a value of *K*. The combined rank displacement of the top *K* items with the highest frequency in the *I-list* is constantly (i.e. after inserting each transaction) monitored. Therefore, restructuring is performed when this value is greater than a given displacement threshold value (λ). For instance, $K = 3$ and the structure of a sorted *I-list* is I_{sort} = {*a*:5, *e*:4, *b*:4, *d*:4, *f*:4, *c*:2} (the number after ":" indicates the support) with the rank of the first item '*a*' is 1, that for the next item '*e*' is 2, and so on. For this I_{sort}, the content of the top-*K* ($K = 3$), say *K-list*, will be {*a*:5, *e*:4, *b*:4}. After inserting a new transaction {*a*, *d*, *f*} into the corresponding tree, I_{sort} becomes I = {*a*:6, *e*:4, *b*:4, *d*:5, *f*:5, *c*:2} and *K-list* becomes *K′-list* = {*a*:5, *d*:5, *f*:5}. The displacement for each item in *K′-list* is calculated by the difference of its current (in I) and previous (in I_{sort}) ranks. Therefore, the displacement for the first item '*a*' in *K′-list* is zero since its rank is unchanged, that for the second item '*d*' is +2 ($= 4 - 2$), and that for the third item '*f*' is +2 ($= 5 - 3$). Thus the combined displacement for the top-*K* items is 4 ($0 + 2 + 2$). If this value is greater than λ, the *restructuring phase* is called.

It is clear that the efficiency of this technique depends on the values of *K* and λ. Any fixed values for both may lead to poor performance, since it is a common phenomenon in almost all datasets that, as time goes by, new items are introduced in *I-list* that result in an increase of *I-list*'s size, making the previous fixed values of *K* and λ insignificant. It is important to choose suitable values for *K* and λ, which will dynamically vary with the population of the *I-list*. From our experience, we have found that if the value of *K* is chosen to be around 20% and 5% of *I-list* size for dense and sparse datasets, respectively, and λ = *I-list* size, better performance is achieved since the displacement is related to the dynamic size of *I-list*, which varies with the rate of new item introduction.

N-*K* *technique*: this is a combination of the above two approaches when the combined displacement of the top-*K* items' ranks is computed after every *N*% of transactions, and checked against the displacement threshold λ. The *restructuring phase* is called if the combined rank displacement of the top-*K* items after each *N*% of transactions is greater than λ. Like both of the previous techniques, the user is to provide the values for *N* and *K*. Moreover, the value of λ can be dynamically set in the same manner as the top-*K* technique.

4.3. Tree restructuring methods

The other performance factor of the CP-tree construction is related with the way to restructure the CP-tree. As discussed above and shown in Section 4.1, the construction of a CP-tree requires dynamic tree restructuring multiple times to obtain a frequency-descending tree. Therefore, an efficient tree restructuring mechanism is an important factor in reducing overall tree restructuring overhead. Usually a tree is restructured by rearranging the nodes of an existing prefix-tree built based on an *I-list* order to another order of items in a new *I-list*. This operation involves both rearranging the items in *I-list* and restructuring the tree nodes. In this subsection, we present an existing tree restructuring approach called the path adjusting method and propose a new technique called the branch sorting method (BSM).

4.3.1. Path adjusting method

The path adjusting method was proposed in [20] where the authors used this technique to rearrange the structure of an already constructed FP-tree, due to updating the *DB*. In this method, the paths in a prefix-tree are adjusted by recursively swapping the adjacent nodes in the path unless the path has completely achieved the new sort order. Thus, it uses a bubble sort technique to process the swapping between the two nodes. One of the basic properties of the FP-tree is that the count of a node cannot be greater than that of its parent. In order to maintain this property in the updated FP-tree, this method inserts a new node of the same name as a sibling of the parent node in the tree when the parent node needs to be exchanged with any child node having a smaller count value. Otherwise, when the support counts of both of these two nodes are equal, a simple exchange operation between them is sufficient. However, after swapping, this method may need to repeatedly merge the two siblings when they become the same due to the exchange operation. We refer to [20] for interested readers to obtain more insight about the method. However, we provide a rough sketch of the method in the following discussion.

In this method, the frequency-descending order of items in the *I-list* is followed to restructure the tree. At first, the method finds a pair of items in the *I-list* that requires exchange to adjust the tree structure according to the sort order, and then performs the exchange operation between them in the tree first and then in the *I-list*. Let us assume that the tree based on an old unsorted *I-list*, i.e. I and that based on a frequency-descending *I-list*, i.e. I_{sort} are represented by T and T_{sort} respectively. According to the basic philosophy of the path adjusting method, given T and its I, the first step is to find each pair of items in I that require exchange to construct I_{sort}. Then, not only all the nodes of these two items occurring in all respective paths in T, but also the same items in I are exchanged. If in a particular path there is any intermediate node found in between these two exchanging nodes, then the upper node is swapped down using a bubble sort technique to reach the bottom node. In each intermediate swapping, I is adjusted accordingly. Finally, T_{sort} and I_{sort} are constructed when no further pair of items in I is found to be exchanged. Fig. 3 provides the basic working principle while exchanging two nodes. As shown in the figure,

Let X, Y, and Z be three nodes in a path in a prefix-tree, where X is the parent of Y, Y is the same oi Z and nodes Y and Z are required to be exchanged to adjust the path. Consider node_name.name, node_name.count and node_name.child refer to the name, the support count (in the referred path) and a child of a node. Therefore, the path adjusting is performed according to the following algorithm:

1. *If (Y.count == Z.count) goto step 5*

***Insertion**:*

2. *Insert Y' (such that Y'.name = Y.name) to X as a new child node and Set Y'.count = Y.count - Z.count*
3. *Assign all children of Y except Z to Y'*
4. *Set Y.count = Z.count*

***Exchange**:*

5. *Exchange parent and children links of Y and Z*

***Merge**:*

6. *If C is another child node of X such that C.name = Z.name then Merge_Node(C, Z)*
7. *Delete C and its sub-tree*
8. *Repeat with next two nodes of Y and Z in another path to be exchanged and Terminate when no further node exchange is required.*
9. *Merge_Node(P, Q){*
10. *Set Q.count = Q.count + P.count*
11. *For each child of Q*
12. *For each child of P*
13. *If (Q.child = = P.child)*
14. *Merge_Node(Q.child, P.child)*
15. *Else add P.child and its sub-tree to Q children list*
16. *}*

Fig. 3. Path adjusting method algorithm.

this operation is performed with three phases: insertion, exchange, and merge. To gain a better understanding of the method, let us visit the following example.

Example 1. Fig. 4a represents a *DB* of four transactions. Consider the prefix-tree of Fig. 4b to be *T* constructed based on the *DB* using lexicographic order of items. The order of items in *I* becomes $I = \{a{:}2, b{:}3, c{:}1, d{:}3, e{:}2, f{:}1\}$, which is not in frequency-descendent order. The frequency-descendent order of items is $I_{sort} = \{b{:}3, d{:}3, a{:}2, e{:}2, c{:}1, f{:}1\}$. Therefore, the restructuring of the tree according to I_{sort} requires several exchange operations among *b*, *a*; *d*, *c; d*, *a*; and *e*, *c* in *I*, and the corresponding nodes in *T* in this sequence. Moreover, the tree *T* should be adjusted after performing each exchange operation. During the first exchange between all '*a*' and '*b*' nodes in *T*, the count of node '*a*:2' (i.e. 2) in the first path of the first branch is found to be greater than that of its child '*b*:1' (i.e. 1). Therefore, another new node named '*a*' should be inserted as a sibling of '*a*' (i.e. as a new child of the *root*). In other words, we can say node '*a*:2' is split into two nodes. The original '*a*' node will contain '*b*:1' as a child, while the new '*a*' node will be the parent of the remaining children of the original '*a*' node. In this example there is only one child '*c*:1' with this branch. The whole process of splitting and children assignment is shown in Fig. 5a. Nodes '*a*:1' and '*b*:1' are exchanged, as shown in Fig. 5b. Since this exchange operation results in the *root* of the tree obtaining two nodes of the same item '*b*', nodes '*b*:1' and '*b*:2' (both children of the *root*) are merged together (Fig. 5c) to become one node '*b*:3' with recursive merging of their children as well. After completing these splits, exchange and merging operations between all '*a*' and '*b*' nodes in *T*, items '*a*' and '*b*' are exchanged in *I*, as shown in Fig. 5c. Following the same procedure all other swapping between tree nodes will be performed, which will produce the restructured frequency-descending tree T_{sort} of Fig. 4c.

It is clear from the above discussion that the path adjusting method requires a heavy computation load for splitting, swapping, and merging tree nodes. This problem is more serious because this method uses the bubble sort method to recursively exchange adjacent tree nodes. Moreover, the sorting needs to be applied to all the branches affected by the change in item frequency. There are many branches in a tree! So, the amount of computation increases significantly as the tree becomes large. Therefore, we offer an alternate array-based tree restructuring method called the branch sorting method that can eliminate the node splitting, swapping, and merging problems of the path adjusting method. In the next subsection, we discuss our branch sorting method.

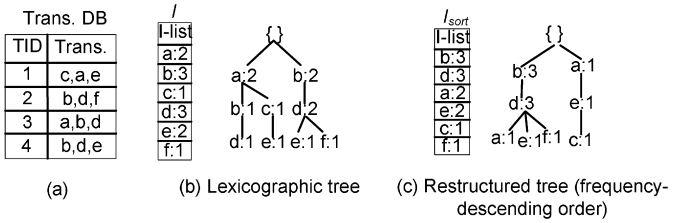


Fig. 4. Tree restructuring – before and after.

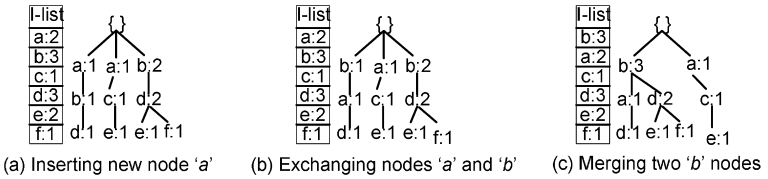


Fig. 5. Path adjusting method- exchanging nodes '*a*' and '*b*' for prefix-tree of Fig. 4.

4.3.2. Branch sorting method

We propose our tree restructuring approach, called the branch sorting method (BSM), that unlike the Path adjusting method, first obtains the I_{sort} by rearranging the items in I in a frequency-descending order and then performs the restructuring operation on T. It is an array-based technique that performs the branch-by-branch restructuring process from the *root* of T. Each sub-tree under each child of *root* can be treated as a branch. Therefore, a tree T contains as many branches as the number of children it has under the *root*. Each branch may consist of several paths. While restructuring a branch, BSM sorts each path in the branch according to the new sort order by removing it from the tree, sorting it into a temporary array, and again inserting it into the tree. However, while processing a path, if it is found to already be in sort order, the path is skipped and no sorting operation is performed. Finally, the restructuring mechanism is completed when all the branches are processed which produces the final T_{sort}. We call this approach the branch sorting method, since it performs branch-by-branch restructuring operations on T. We use a merge sort technique to process (i.e. sort) both I and any unsorted path in T. We refer to the following formal definitions before stating the tree restructuring mechanism using our BSM.

Definition 1 (*Sorted path).* Let P_1: $\{a_1, a_2, \ldots, a_n\}$ be a path in T, where a_1 and a_n are the immediate next node to the *root* and the leaf node of the path, respectively. P_1 is defined as a sorted path if all items represented by a_i, $\forall i \in [1, n]$ are found in I_{sort} order. For example, path $b{:}1 \rightarrow d{:}1 \rightarrow e{:}1$ in the second branch of prefix-tree of Fig. 4b is a sorted path, since in this path items of all nodes are already arranged in I_{sort} order.

In contrast, any path which is not a *sorted path* can be referred to as an *unsorted path*.

Definition 2 (*Branching node).* Let a be any node in T and a_{clist} refers to its children list. Let size (a_{clist}) be the size of this children list (i.e. the total number of children in the list). Node a is defined as a branching node if size (a_{clist}) > 1. For example, in Fig. 4b nodes '*a*:2' and '*d*:2' are branching nodes, since these nodes contain more than one child.

Based on the above definitions we obtain the following important corollary for handling a *sorted path* during tree restructuring:

Corollary 1. *Let P_1: $\{a_1, a_2, \ldots, a_k, a_{k+1}, \ldots, a_n\}$ be a path in T, where a_1 and a_n are the immediate next node to the root and the leaf node of the path, respectively. Let a_k be a branching node and P_2:$\{a_1, a_2, \ldots, a_l, a_{l+1}, \ldots, a_m\}$ be another path sharing the same prefix $\{a_1, a_2, \ldots, a_k\}$ with P_1 where $k = l$. If P_1 is a sorted path, then the sub-path $\{a_1, a_2, \ldots, a_l\}$ in P_2 is also sorted if and only if no item between a_{l+1} and a_m possesses an item rank (calculated in a similar fashion as discussed in top-K heuristic in Section 4.2) less than that of a_l.*

Proof. Since P_1 is a *sorted path*, item ranks of $a_1, a_2, \ldots, a_k$ nodes must be in sorted order. If any individual item between a_{l+1} and a_m in P_2 contains an item rank smaller than that of a_l, i.e. a_k then the position of that item would be anywhere between the *root* and a_l. Therefore, the sub-path $\{a_1, a_2, \ldots, a_l\}$ would not be sorted. Otherwise, the sub-path $\{a_1, a_2, \ldots, a_l\}$ is also sorted in P_2. □

Based on the above definitions and corollary, we provide the BSM algorithm in Fig. 6. As discussed above according to the basic philosophy of BSM, given I and T, I_{sort} is constructed first (line 1) in frequency-descending order of items in I. Then all the branches of T are checked one after another (line 2) for any *unsorted path* (line 4) and, if found, the path is sorted (line 6) and inserted into T (lines 19–22). Therefore, when no branch is left to be processed, restructuring of T is complete and T_{sort} is the new restructured tree (line 7).

One of the important features of BSM is handling the branches with *sorted path*(s). During tree restructuring, if any path is found to be a *sorted path* (line 4), we not only skip the sort operation for that path, but also propagate this status information about the path to all *branching nodes* in the same path of the whole branch, indicating that the path from that *branching node* up to the *root* is sorted. Therefore, while sorting the other remaining paths in the same branch, only the sub-paths from the leaf node to the *branching node* of the *sorted path* are checked to determine whether any item in the sub-path contains a rank less than that of the *branching node* (line 11). If no such item is found, according to Corollary 1, we keep the sub-path from the *root* to the *branching node*, which is the common prefix path, as it is and only process the sub-path from the *branching node* to the leaf (lines 12, 13, 14, 16). Otherwise, the whole path from the leaf to the *root* (line 15) is sorted (line 16). When all the sub-paths (line 10) from the *branching node* are adjusted, we move to the next available *branching node* (line 9). Therefore, the BSM significantly reduces the processing cost when several paths in T are found to be in sort order during tree restructuring. From the above algorithm, we can deduce the following lemma.

```
Input: T and I
Output: T_sort and I_sort
1.  Compute I_sort from I in frequency-descending order using merge sort technique
2.  For each branch B_i in T
3.    For each unprocessed path P_j in B_i
4.      If P_j is a sorted path
5.          Process_Branch(P_j)
6.      Else  Sort_Path(P_j)
7.  Terminate when all the branches are sorted and output T_sort and I_sort.
8.  Process_Branch (P){
9.      For each branching node n_b in P from the leaf_p node
10.       For each sub-path from n_b to leaf_k  with k ≠ p
11.         If item ranks of all nodes between n_b and leaf_k are greater than that of n_b
12.             P = sub-path from n_b to leaf_k
13.               if P is a sorted path
14.                     Process_Branch(P)
15.         Else   P =  path from the root to leaf_k
16.         Sort_Path (P)
17. }
18. Sort_Path (Q){
19.   Reduce the count of all nodes of Q by the value of leaf_Q count
20.   Using merge sort, sort Q in an array according to I_sort order
21.   Delete all nodes having count zero from Q
22.   Insert sorted Q into T at the location from where it was taken
23. }
```

Fig. 6. Branch sorting method algorithm.

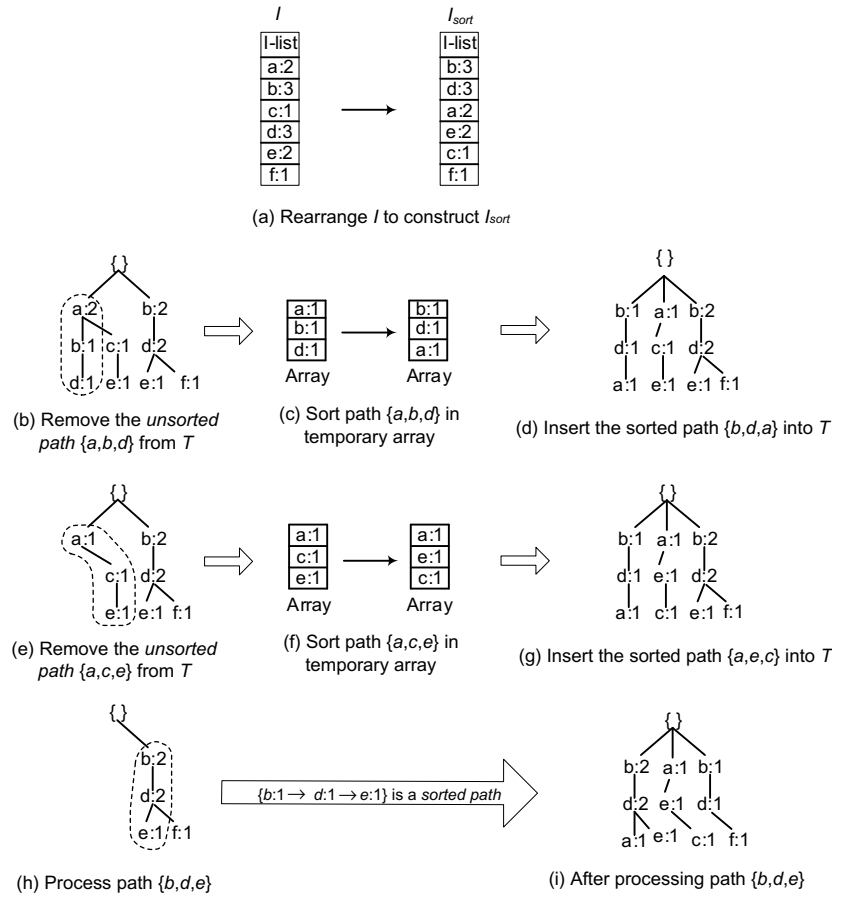


Fig. 7. Branch sorting method.

Lemma 3. *Given a transaction database DB and using the CP-tree construction mechanism, a tree T can be constructed on DB in any frequency-independent item order. If T_{sort} is a frequency-descending tree restructured from T using the BSM tree restructuring technique, then T_{sort} contains the complete information from DB.*

Proof. Based on the BSM tree restructuring mechanism, each path in *T* is inserted into T_{sort} by sorting (if required) in a different order. In other words, there is one path in T_{sort} corresponding to one path in *T*, and vice versa. Again, based on the CP-tree construction mechanism, *T* contains complete information from *DB*. Therefore, T_{sort} contains the complete information from *DB* and the lemma holds. □

To illustrate the working principle of this tree restructuring method we use the following example.

Example 2. Consider the same *DB* and trees as in Fig. 4. As shown in Example 1, *I-list*, *I*, and the prefix-tree, *T* in Fig. 4b are not arranged in frequency-descending order. Therefore, to restructure the tree using the BSM in such order, *I* is sorted first to generate a new *I-list*, I_{sort} as shown in Fig. 7a. Tree restructuring, then, starts with the first branch, which is, say the left most branch from the *root* of the tree in Fig. 4b. Since the first path $\{a{:}1 \rightarrow b{:}1 \rightarrow d{:}1\}$ of the branch is an *unsorted path*, it is removed from the tree (Fig. 7b), sorted using the merge sort technique into a temporary array to the order $\{b{:}1 \rightarrow d{:}1 \rightarrow a{:}1\}$ by satisfying I_{sort} order (Fig. 7c), and then again inserted into *T* in sorted order. Fig. 7d shows the tree structure after sorting the first path. Using the same technique, the next path $\{a{:}1 \rightarrow c{:}1 \rightarrow e{:}1\}$ is processed and inserted into *T* in $\{a{:}1 \rightarrow e{:}1 \rightarrow c{:}1\}$ order. The processing steps of this path are shown in Fig. 7e–g. Insertion of this path into the tree completes the sorting operation for the first branch of *T*. The next branch from the *root* of *T* is then processed by processing the first path $\{b{:}1 \rightarrow d{:}1 \rightarrow e{:}1\}$. This path is a *sorted path*. That is, nodes in the path are already arranged in I_{sort} order and, therefore, it is not necessary to sort it. It only needs to be merged with the previously processed common *sorted path* (if any) with necessary pointer adjustments, as shown in Fig. 7h and i. Since this path is a *sorted path*, the first *branching node* from leaf '*e*:1' in *T*, '*d*:2' maintains the information that the sub-path from itself to the *root* is sorted. Therefore, according to Corollary 1, only the remaining sub-paths from this node toward leaf nodes are required to be checked in order to determine whether any such sub-path contains any node whose item rank is less than that of the branching node '*d*:2'. In this example, we have only one sub-path containing one node '*f*:1' whose item rank in I_{sort} is greater than '*d*' node's item rank. So, we can conclude that the next path $\{b{:}1 \rightarrow d{:}1 \rightarrow f{:}1\}$ is also a *sorted path*. Thus, the path is processed in the same fashion as the previous path. The restructuring of the tree in Fig. 4b is therefore complete, since all the branches are processed. The complete frequency-descending tree obtained after completing tree restructuring using the BSM is the tree as shown in Fig. 4c. According to Lemma 3, this tree maintains complete database information in a more compact manner.

The BSM sorting method does not affect the links between the items in the *I-list* and in the tree; hence it is not necessary to readjust the link pointers of items in the *I-list*. tree traversal from the newly sorted *I-list* is not hampered and no extra costs are involved.

From the above discussion we can see that if a branch in *T* contains no or few *sorted path*s, it requires sorting for each path individually. If all or some of these paths share a common prefix, then this prefix sub-path will be processed as many times as the number of paths that share it. This may increase the processing cost of the approach due to repeated processing of a common sub-path. Therefore, to further improve the performance of BSM, we adopt an optimization technique to reduce the tree restructuring cost. We introduce a new term *pseudo branching node* as shown below.

Definition 3 (*Pseudo branching node).* Let $P_1{:}\{a_1, a_2, \ldots, a_k, a_{k+1}, \ldots, a_n\}$ be an unsorted path in *T*, where a_1 and a_n are the immediate next node to the *root* and the leaf node of the path, respectively. Let a_k be a branching node. If according to I_{sort}, a_i, $\forall i \in [1,k]$ is the node having the highest rank value among all nodes of the prefix path $\{a_1, a_2, \ldots, a_k,\}$, then node a_i is defined as a pseudo branching node in P_1.

Therefore, the basic idea of optimization is that if a branch in *T* contains several *unsorted path*s sharing a common prefix, we sort the first *unsorted path* among them first. Then we can use the same concept as the *sorted path* handling mechanism by considering the *pseudo branching node* of the common prefix as a *branching node*, which can be obtained from the sort order of all nodes in the common prefix after sorting the first path of the branch. Therefore, while sorting the next path(s), only the sub-paths from the leaf node to the first *branching node* are checked against the *pseudo branching node*. That is, if the sub-path contains any node having rank less than that of the *pseudo branching node*, only then do we perform a sort operation on the whole path from the *root* to the leaf. Otherwise, simply sorting the sub-path is enough to sort the whole path according to I_{sort}. This is because while sorting the first path, we have already obtained the sort order of all nodes with a common prefix. Therefore, using this information, we can skip some operations, while processing each remaining path. Thus, we can improve overall tree restructuring performance by reducing the amount of data in sorting operations.

It may be assumed that constructing a frequency-descending tree using the BSM restructuring technique may require double computation since it removes a path from the tree structure and inserts it into the tree again in sorted order. However, we argue that the time complexity that BSM requires is generally much less than the double computation of removing and inserting each transaction of a database for the following reasons. First, since BSM sorts each path with its support count (i.e. support count of the last node), it eventually processes as many transactions as the count value with a single operation. Thus, all of the identical transactions are processed together, which reduces the transaction-by-transaction processing cost. Second, the *sorted path* handling mechanism enables the BSM to further reduce the

processing cost when the path is already sorted, since no operation is done on such paths. Third, the information carried in each *branching node* in a *sorted path* may significantly cut down the time requirement, since we need to process only the sub-paths (from the *branching node* to the respective leaves) when the lowest rank among all the items in the sup-path is higher than that of the *branching node*. Fourth, the optimization achieved by applying the concept of *pseudo branching node* helps to further reduce the restructuring cost when several paths in a branch share common prefixes, which is very common in prefix-tree structures.

In general, there are many identical transactions and/or common prefixes among the transactions occurring in a database. Moreover, the periodic tree restructuring operation constantly facilitates the tree structure to obtain a higher degree of prefix sharing among the already inserted transactions, and reduces the overall restructuring cost compared to restructuring the tree once at the end of *DB* (the experimental results shown in Section 6.2.1 also demonstrate this fact). Therefore, through the effective utilization of the *sorted path* and the *pseudo branching node* handling mechanisms, BSM significantly reduces the computation cost of restructuring the tree with a single database scan. In contrast, if there are no identical transactions, no prefix sharing among the transactions, and no sorted path in the tree, only then does BSM need to process each path (i.e. each transaction) individually by removing it and inserting it back again into the tree structure. However, BSM can still perform better compared to scanning the database a second time, since reading a transaction from the memory-resident tree structure is faster than scanning it from the disk, while other tasks such as sorting it and inserting it into the tree are the same in both cases.

In the next subsection, we analyze the complexity of two tree restructuring approaches.

4.3.3. Complexity analysis

It is clear from the above discussion that the performance of the path adjusting method largely depends on the degree of displacement among the items between two *I-lists*, I and I_{sort}. One way to calculate this degree of displacement of I may be to compute the total rank displacement of all the items in it with respect to I_{sort}. For example, the *degree of rank displacement* of two *I-lists* $I = \{a{:}2,\ b{:}3,\ c{:}1,\ d{:}3,\ e{:}2,\ f{:}1\}$ and $I_{sort} = \{b{:}3,\ d{:}3,\ a{:}2,\ e{:}2,\ c{:}1,\ f{:}1\}$ of Examples 1 and 2 is 8 $\{=|1-3| + |2-1| + |3-5| + |4-2| + |5-4| + |6-6|\}$. However, for simplicity of discussion, for the rest of the paper we refer to the degree of displacement between two *I-lists* as the *degree of disorder*. Since the path adjusting method uses a bubble sort technique to swap two items in a path, the relative position of two items involved in a swapping operation is one of the major factors contributing to the overall performance. The swapping between two nodes in a path must consider the bubble sort running cost of $O(n^2)$, where n is the number of nodes involved in the path between them. Therefore, the cost of swapping two nodes in a path increases exponentially with the increase in the number of intermediate nodes between them in the path. Moreover, this computation becomes more time-intensive when there are more paths in the tree requiring such adjusting. Hence, this method may suffer from relatively high time requirements to adjust the tree with larger and a greater number of branches, or an *I-list* with many items in it with a high *degree of disorder*.

When both *I-lists* are in opposite order, i.e. items are ordered in reverse direction from each other, the worst case swapping complexity of this method is observed, which is $O(mn^2)$ where m is the total number of paths (each path has > 1 node) and n in the average length of all transactions. On the other hand, the best case can be achieved if the *I-lists* are almost the same, with very few (say, only two) items with different item ranks between two lists, and all transactions are identical. So, the best case complexity will be $O(mn^2)$, with values $m = 1$ and $n = 1$. In addition, in both cases this method has to consider another complexity of $O(mn^2)$, for checking the order of items to be sorted. Therefore, as mentioned above, the performance of this method also depends on the number of paths in the tree affected by the change in item frequency. In the worst case, it is the total number of paths in the tree, while it is only one path in the best case. Thus, for the case of a nearly sorted tree, the number of such paths and the *degree of disorder* may be less. As a result, this approach should perform better in such a case, since its runtime becomes closer to the best case complexity.

On the other hand, our BSM uses a merge sort approach to sort the nodes of any path and, therefore, the *degree of disorder* does not have a large effect on the performance during sorting, since irrespective of data distribution the complexity of merge sort is always $O(n\log_2 n)$, where n is the total number of items in the list. Moreover, the number of intermediate nodes to be sorted is also not an influencing parameter in BSM. One of the important features of BSM is handling the branches having *sorted path*(s) that can reduce the number of sorting operations and/or the size of data to be sorted.

The worst case of BSM occurs when it restructures a tree constructed on a *DB* in which every two transactions share no prefix with each other, and the tree contains no *sorted path*. In this case, the cost of sorting all paths in the tree is $O(mn\log_2 n)$, where n is the average length of transactions and m is the number of paths in the tree. Similar to the path adjusting method, the number of *unsorted paths* in the tree is also an influencing parameter on the time required with BSM. However, it is not as serious as in the path adjusting method because it multiplies a linear term ($n\log_2 n$) in BSM, while in the path adjusting method it is an exponential term (n^2). BSM achieves the best case complexity of $O(n\log_2 n)$ when all the transactions are identical with length n.

Therefore, in the case of the path adjusting method, if the tree is almost sorted previously it achieves better performance while the BSM performs comparatively better when the tree size and transaction length are large and two *I-lists* differ significantly. Consequently, it is not appropriate to use the path adjusting method when the *degree of disorder* is high and/or the number of nodes to be swapped is relatively large. Based on the above discussion we can draw the following conclusions.

Conclusion 1: the path adjusting method is a better choice when the degree of disorder and tree size are low.

Conclusion 2: the branch sorting method is a better choice when the degree of disorder and tree size are high.

From our experience we have found that usually the rate of new item introduction is greater early in the *DB*, i.e. the size of an *I-list* increases more rapidly during the initial stages of a *DB* scan. However, this phenomenon may not occur in huge sparse datasets containing transactions with few co-occurrences among items. From this observation on dataset characteristics and analyzing the performance characteristics of both restructuring methods, we use the phase-by-phase restructuring technique and adopt the branch sorting approach for the first restructuring operation, because of the possible high *degree of disorder* and larger tree size during this stage. For the remainder of the *DB*, unless the *degree of disorder* becomes high, we select the path adjusting method. A somewhat dynamic manner to choose the restructuring method might be applicable based on the value of *degree of disorder*. Clearly, this approach demands monitoring the status of the value of *degree of disorder* at regular intervals.

4.4. Mining using pattern growth technique

Recall from Section 1 that the primary goal of constructing the CP-tree is to obtain a significant improvement in mining performance based on its compact tree structure. So far, we have shown in detail how the CP-tree can be constructed in frequency-descending order with a dynamic tree restructuring mechanism. We have discussed the benefits we attain due to such ordering in *I-list* and the tree. We also mentioned that the CP-tree uses an FP-growth mining technique. In this subsection, we revisit the FP-growth mining approach and investigate in detail why the CP-tree with frequency-descending item ordering achieves significant improvement in mining performance.

The basic operations in the FP-growth based pattern growth mining approach are (i) counting frequent items, (ii) constructing conditional pattern-base for each frequent item, and (iii) constructing new conditional tree from each conditional pattern-base. Counting frequent items is facilitated with the help of an *I-list* that contains each distinct item along with the respective support count. In fact, this is another purpose of maintaining such a list. For each frequent item in *I-list* X_i, a small conditional pattern-base PB_{X_i} is generated, each consisting of the set of transformed prefix paths of X_i from the original tree and the frequency count of every node in PB_{X_i} carries the count of the corresponding X_i in the path. A small *I-list* consisting of only the items in PB_{X_i} for X_i is also created. We refer to such an *I-list* as *I-list*$_{X_i}$. The mining frequent patterns for all the patterns suffixing X_i is then recursively performed on PB_{X_i} by constructing a conditional tree from PB_{X_i}. Construction of a conditional tree requires eliminating all infrequent items having a count less than the *min_sup* value from the *I-list*$_{X_i}$ and corresponding nodes from the tree of PB_{X_i} and then adjusting the tree structure. Therefore, the performance of the mining phase depends on (i) how efficiently the search space is managed, in other words, how effectively all items are sorted in the *I-list*; (ii) the number of infrequent nodes to be deleted from the conditional pattern-base to construct the conditional tree, in other words, the number of nodes participating in the mining operation; (iii) how fast the deletion and successive tree adjustment are executed, (iv) the number of conditional trees constructed during the mining process, and (v) the mining cost of each individual conditional tree.

If the *I-list* is arranged in any canonical order of items, it is quite unlikely that all the frequent items for a given *min_sup* value will appear at the top part of the tree. Thus, it is rather obvious that there will be a mix-up of frequent and infrequent items in such an *I-list* and its corresponding tree. Since PB_{X_i} maintains the same item ordering as its *I-list*$_{X_i}$ which is constructed following the main *I-list* order, the tree may contain nodes representing *global* infrequent items. As a result, it not only contributes to increase the number of nodes participating in the mining process, but also incurs additional node deletion and adjustment operations on the tree.

Eliminating any node from a tree has three phases: *pointers adjustment*, *node deletion*, and *tree adjustment*. Pointers adjustment incurs extra operations of changing the node's children and parent pointers, and shifting the item traversal pointers to the next item in the tree, while node deletion is simply removing the node from the tree by freeing its physical memory. The children's pointer adjustment assigns the parent to all of its children (if any), merges the children (if any) list to the parent's children list and removes itself from that list, while the shifting item travel pointer alters the tree node traversal pointer to the next immediate node of the same item. However, completion of all these operations may not terminate the node elimination process. Rather, the whole process might be recursively repeated for all consecutive children pairs (from two children lists). This phase refers to the *tree adjusting* phase. Therefore, constructing a conditional tree of any item from its corresponding conditional pattern-base may require multiple *global* infrequent node(s)[2] deletion operations for each *global* infrequent item in the list, which obviously increases cost with respect to the situation when the list does not contain any *global* infrequent items.

An alternate approach to obtain the *global* infrequent item free conditional pattern-base is to include only the frequent items while traversing a prefix path to construct a conditional pattern-base. However, this process will require an enormous number of look-up operations to the *I-list* to check whether each item in the prefix path is frequent or not. For a prefix path of N items, N look-up operations are required irrespective of how many items are frequent or infrequent. Moreover, if this prefix path shares first N-K,$\forall K \in [1,N]$ items with one or more prefix paths, during processing these N-K look-up operations must be repeated as many times as they are present in other prefix paths. Furthermore, this approach has to consider all the node

[2] A node in the tree structure that represents a *global* infrequent item.

elimination phases discussed above except item travel pointer adjustment and node deletion operations. Therefore, this approach not only increases the number of nodes participating in the mining process, but also incurs an additional large amount of *I-list* look-up operations and node adjustment processes in the tree. Thus, from the above discussion we can deduce the following important lemma.

Lemma 4. *Let T and T_{sort} be two trees constructed in any canonical order and frequency-descending order, respectively, on a database DB. If N_T and $N_{T_{sort}}$ are the total number of global infrequent nodes participating in the mining operations, respectively, in both trees for a min_sup value = ∂, then $N_{T_{sort}} \leqslant N_T$.*

Proof. It is obvious that, if all the items in *DB* carry support $\geqslant \partial$ then $N_{T_{sort}} = N_T$. If at least one item in *DB* is infrequent, then as T_{sort} is sorted in frequency-descending order, that particular item must appear at the bottom of the *I-list* reflecting that all the nodes containing that item cannot appear in any conditional pattern-base of any of the other frequent items. Therefore, there is no scope for these nodes to participate in the mining operation. In contrast, for *T*, this infrequent item may appear at any location in the corresponding *I-list*. If it appears anywhere except the last item in the *I-list*, then all or some of the nodes in *T* containing that item may appear in any conditional pattern-base of other frequent items under it in the *I-list*. This results in an increased number of nodes participating in the mining process. The more infrequent items the *I-list* contains, the greater the probability of handling such a phenomenon. Therefore, the number of *global* infrequent nodes participating in the mining operation in T_{sort} cannot be greater than that in *T*, i.e. $N_{T_{sort}} \leqslant N_T$. □

From this lemma, we can directly derive the following important corollary.

Corollary 2. *Let T be a tree and I be its corresponding I-list. If X (an item in I) is frequent then the conditional pattern-base of X, PB_X, i.e. the prefix-tree suffixing X, does not contain any global infrequent node for any value of min_sup, if and only if I and T are arranged in frequency-descending item order.*

Proof. Let $X_1, X_2, \ldots, X_k, \ldots X_j, \ldots, X_n$ be the contents of *I*. Let X_i, $\forall i \in [1,n]$ be the item of reference. The conditional pattern-base for X_i, PB_{X_i} must contain item(s) between X_1 to X_{i-1}. If *I* is arranged in frequency-descending item order, then $Sup(X_i) \geqslant Sup(X_{i+1})$, $\forall\ i \in [1,n]$. Therefore, $Sup(X_k) \geqslant Sup(X_j)$, $\forall k \in [1,j-1]$ hence, if X_j is frequent, then X_k must be frequent. On the other hand, if *I* is not arranged in such an order, the proposition $Sup(X_i) \geqslant Sup(X_{i+1})$, $\forall i \in [1,n]$ is not satisfied and, consequently, PB_{X_i} may contain *global* infrequent node(s). □

According to Corollary 2, conditional tree construction for any item in *I-list* can always be performed by avoiding the *global* infrequent node deletion process in its conditional pattern-base if the *I-list* is arranged in frequency-descending order. Moreover, in such an order, the search space for finding the next frequent item in the *I-list* is constantly one item (with $O(1)$) starting from the bottom-most item having support $\geqslant \partial$ to the top-most item in the list. On the other hand, searching for the first frequent item to start mining in any frequency-independent *I-list* must begin from the bottom of the list. At the same time, while traversing the *I-list*, it is not guaranteed that the next immediate item is also frequent; meaning searching for the next frequent item requires applying a linear search method in the upward direction until that item is found.

The order of items in *I-list* also directly influences the total number of conditional databases constructed during the mining process. The frequency-descending order is capable of minimizing the total number of conditional databases. Let X_l be the frequent item with the lowest support and X_h be the most frequent item in the *I-list*. Then consider two situations: (1) X_l is the bottom-most item in the list to start mining; and (2) X_h is the bottom-most item in the list to start mining. Therefore, in the two cases the two items have almost the same set of *candidate extensions*. However, the frequency of X_h is usually much higher than that of X_l, i.e. X_h's conditional database contains more transactions than X_l's conditional database. This implies that it is very possible that the number of *frequent extensions* of X_h is much larger than that of X_l. Therefore, in situation (2), it is very possible that we have to construct larger and/or more conditional trees in subsequent mining. In contrast, if we sort frequent items in frequency-descending order, X_l will be the first item to start mining. The number of its *frequent extensions* cannot be very large, and we need to build smaller and/or less conditional databases in subsequent mining. With mining in progress, the items become more and more frequent, but their candidate extension sets become smaller and smaller and the transactions in their conditional databases become shorter and shorter. This ensures that the number of conditional trees constructed in subsequent mining cannot be large.

In support of the above discussions, we use the following example to demonstrate the performance gain that CP-tree achieves in mining due to its dynamic tree restructuring operation:

Example 3. Refer to CP-tree and CanTree of Fig. 2g and b, respectively. Both were constructed from the database given in Fig. 2a. The CP-tree is a frequency-descending tree, while the CanTree is built based on a lexicographic order of items. Consider the mining operation for finding frequent patterns for *min_sup* of 50% in this database using both trees individually. Any pattern having support $\geqslant 3$, therefore, would be a frequent pattern in this database. The items *d*:5, *e*:4, and *a*:3 are therefore frequent, while other items, say '*b*' and '*c*', are infrequent. In order to mine all frequent patterns with the CP-tree, we can search the *I-list* from the top for the last item with a support value $\geqslant 3$, and then start mining from that item upward. Thus, we find item *a*:3 to be the first item and we complete mining for '*a*'. After finishing mining with all patterns suffixing '*a*', we mine for other remaining items (i.e. '*e*', and '*d*') above this item. Most importantly, to obtain the next frequent item in the *I-list* we need to look-up the next item only. We never access the items after '*a*' in the list. This early pruning of infrequent

items saves both *I-list* traversal costs and condition (frequent/infrequent) checking cost. In contrast, while mining with CanTree, which has a frequency-independent *I-list* order, we need to consider the whole *I-list* as the search space to find frequent items. Since FP-growth traverses the tree in a bottom-up manner, it is a good strategy to start searching from the bottom most item '*e*' in the *I-list*. Since '*e*' is a frequent item, we can start mining with this item. After completing mining all patterns suffixing '*e*' we move to the next frequent item in the *I-list*, '*d*' which is also a frequent item. Therefore, after completing mining with '*d*' we perform the same operation for seeking the next frequent item. However, due to the frequency-independent order of items, we need to check and skip two infrequent items ('*c*' and '*b*') to obtain the next frequent item. Thus, CanTree has to consider not only traversing the whole *I-list*, but also performing the extra checking for each item in the list, which is extra computation overhead due to the frequency-independent search space arrangement.

Fig. 8 shows the conditional pattern-bases constructed for the two items, '*a*' for CP-tree and '*e*' for CanTree. Since in the CP-tree, '*a*' is the least frequent item among all the frequent items, the pattern-base of '*a*' contains few *candidate extensions*. Therefore, the size of the conditional pattern-base for '*a*' in Fig. 8a is very small. Moreover, it is not needed to construct any conditional pattern-base for the most frequent item '*d*', since it is the first item in the *I-list*. In contrast, CanTree constructs the conditional pattern-base for '*e*' (Fig. 8b) first. Since '*e*' is one of the most frequent items, the size of its conditional pattern-base would be larger due to the higher probability of having more *candidate extensions*. Therefore, the number and size of conditional trees constructed from it will be larger. Moreover, while constructing conditional trees from this conditional pattern-base, we need to remove *global* infrequent nodes shown as dotted circles in Fig. 8b from it, which is additional computation overhead for such item ordering.

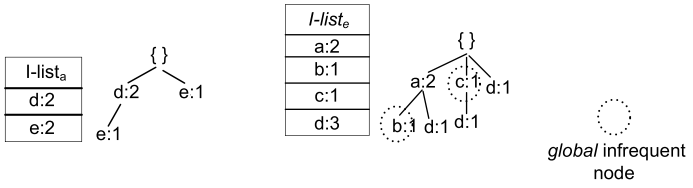
Therefore, for a particular *DB* and for any value of ∂, FP-growth mining from a frequency-descending tree would require minimal time compared to other trees constructed in frequency-independent canonical order. Motivated by this analysis, we design our CP-tree in such a way that it achieves the properties of a frequency-descending tree using one *DB* scan and requires the minimum mining time. Our experimental results given in Section 6 reflect that such mining with a frequency-descending tree is multiple orders of magnitude faster than that with a canonical ordered tree. These results also show that the performance gain we achieve in our CP-tree is primarily due to the significant gain during the mining phase that suppresses the extremely insignificant tree restructuring overhead. In the next section, we discuss some additional forms of frequent pattern mining that CP-tree can efficiently handle.

5. CP-tree and additional functionalities

Besides mining the complete set of frequent patterns, frequent pattern mining has been generalized to many forms since its introduction. There may be some applications that require not only efficient, but also effective frequent pattern mining techniques. Nevertheless, there might be some applications predominantly requiring only one of the above parameters. *Incremental mining*, *interactive mining*, *incremental interactive mining*, *constraint-based mining*, and *incremental constraint-based mining* are examples of candidate requirements for such applications. So far, we have shown that with a single-pass to *DB*, the CP-tree efficiently discovers the complete set of frequent patterns using its highly compact tree structure. It is, however, important to note that the CP-tree also provides us with additional functionalities. For instance, it can be efficiently used for *incremental mining*, *interactive mining*, *incremental interactive mining*, *constraint-based mining*, *and incremental constraint-based mining.*

5.1. CP-tree in incremental mining

The primary objective of the algorithms used in incremental mining is to provide an environment so that frequent patterns can be discovered when the *DB* is constantly updated, such that when new transactions are added and/or old transactions are removed, the patterns can be mined without restarting the mining process from scratch. Like CanTree, our CP-tree retains all items regardless of whether they are frequent or not in a tree structure. Since the CP-tree maintains the complete information for *DB* in a highly compact frequency-descending manner, it is quite easy to discover transactions to delete and/or update, and insert new transactions into the tree. Thus, when adding new transactions into the tree, the transaction should be inserted according to current *I-list* order. Consequently, by using the tree restructuring operation the final compact



(a) Conditional Pattern-base for '*a*' ($\partial = 3$) in CP-tree of Fig. 2(g)

(b) Conditional Pattern-base for '*e*' ($\partial = 3$) in CanTree of Fig. 2(b)

Fig. 8. Conditional pattern-bases on CP-tree and CanTree for $\partial = 3$ on *DB* of Fig. 2a.

CP-tree can be constructed. Such a CP-tree can be used to efficiently mine for frequent patterns for the updated *DB*. CanTree also provides incremental mining capability as mentioned in [24]. While handling incremental updates, it utilizes global item ordering to maintain the updated tree. Therefore, due to its canonical order of items, CanTree's performance is much lower than that of our CP-tree while executing the mining operation.

5.2. CP-tree in interactive mining

It is quite challenging to find an optimum value for the support threshold, especially when there are many frequent patterns that occur in *DB*. Setting *min_sup* too low may result in too many frequent patterns. In contrast, setting *min_sup* too high may result in too few frequent patterns and some important patterns of interest may be missed. Interactive mining provides the user an opportunity to interactively regulate or refine the *min_sup* value. For such mining, the *DB* usually remains unchanged and only the *min_sup* value is changed. So, this technique fits the "build-once-mine-many" principle, where the tree capturing the *DB* content is built only once and used in mining with various *min_sup* values. By doing so, the tree construction cost can be amortized over several runs of the mining process.

Similar to incremental mining, maintenance of complete *DB* information without any loss in tree structure permits the CP-tree to cope well with interactive mining. Once the CP-tree is constructed, based on Lemma 2, it can be used to mine with different support threshold values without rebuilding the tree. Unlike the CanTree, each iterative mining operation with the CP-tree can be performed on a platform of highly compact tree due to the frequency-descending order of items. The time complexity of interactive mining algorithms mainly depends on tree construction cost and the cost of actual mining. Therefore, interactive mining using a CP-tree will always be much faster than that using a CanTree because, in this case, the original tree is constructed only once while the number of mining operations varies.

Similar to other interactive mining algorithms, mining time using the CP-tree can be further reduced by caching the frequent patterns mined from the previous round, and reusing them for the current round with different *min_sup* value. For example, if the new *min_sup* is higher than that used in the previous round, we could find frequent patterns satisfying the new *min_sup* by using those cached patterns. On the other hand, when the new *min_sup* is lower than the previous, we would perform mining for the extensions of these patterns and for all other new patterns.

5.3. CP-tree in incremental interactive mining

In the previous two subsections, we showed that the CP-tree can be efficiently applied in incremental mining and interactive mining application domains. Due to its frequency-descending tree structure, for both forms of frequent pattern mining, our CP-tree outperforms the CanTree. These results reflect that the CP-tree is also effectively applicable in the combined domain of these two, i.e. incremental interactive mining. By using our CP-tree, the user can find frequent patterns with various *min_sup* values from the current database in multiple runs. When such a database is updated, due to insertions and/or deletions of transactions, the CP-tree can precisely and efficiently capture the content of the updated *DB* in a highly compact manner with the required tree restructuring operations so that new sets of frequent patterns for the new *DB* can be found for the same or different *min_sup* values.

5.4. CP-tree in constraint-based mining

In addition to incremental mining and interactive mining, frequent pattern mining has also been widely used for many other applications, which include constraint-based mining. A *constraint C* is a predicate on the powerset of the set of items *L*, i.e. $C{:}2^L \to \{true, false\}$. A pattern *P* satisfies a constraint *C* if and only if *C*(*P*) is true. The set of patterns satisfying a constraint *C* is $Sat_C(L) = \{P | P \subseteq L \wedge (P) = true\}$. We call a pattern in $Sat_C(L)$ valid. Given a transaction database *DB*, a *min_sup* value ∂, and a set of constraints *C*, the problem of mining frequent patterns with constraints is to find the complete set of frequent patterns satisfying *C*, i.e. find $F_C = \{P | P \in Sat_C(L) \wedge Sup(P) \geqslant \partial\}$.

Recent works [31,25,7] have highlighted the importance of the constraint-based mining paradigm. Over the past few years, several FP-tree-based constrained mining algorithms have been developed to handle various classes of constraints associated with frequent pattern mining. In [31], an algorithm named mining frequent itemsets with convertible constraint (*FIC*) was proposed to handle *convertible* constraints such as $C_{avg} \equiv avg(P) \geqslant 25$ which finds all frequent patterns whose average item weight is no less than 25. The success of mining with a *convertible* constraint depends on the respective algorithm's ability to arrange the item space according to some specific order *R* based on the value, weight, function, or any other parameter related to all items. More specifically, *FIC* arranges items according to a prefix function order such as descending order of weights for the above C_{avg}. Another constraint-based algorithm FP-tree based mining of succinct constraints (*FPS*), proposed in [25], arranges items according to the order *M*, specifying their membership in order to support succinct constraint-based mining such as $C_{\max} \equiv \max(P) \geqslant 25$, which finds frequent patterns whose maximum item weight is at least 25.

The CP-tree can be used as a data structure for these constraint-based mining operations. The tree *restructuring phase* in a CP-tree can be employed using the preset specific item-order of interest so that the tree provides a highly compact platform for the mining operation. Due to the compactness of the tree structure, CP-tree will significantly outperform CanTree.

Similar to CanTree, CP-tree can provide the user with further functionality in the case of constraint-based mining by performing *incremental constraint-based* mining more efficiently. By doing so, when transactions are inserted into or deleted

from the original database, both of the algorithms no longer need to rescan the updated database nor do they need to rebuild a new tree from scratch.

Most importantly, the CP-tree can handle *incremental constraint-based* mining on dynamic value, weight, function, or any other parameter related to all items. Since the CP-tree restructures itself dynamically phase-by-phase, it incurs no overhead to tackle such dynamic ordering in *R* and/or *M*. CanTree or the algorithms in [31,25] cannot handle such a situation efficiently when these values are changed.

6. Experimental results

In this section, we present a comprehensive experimental analysis and the performance results of (i) both tree restructuring methods discussed in this paper, and (ii) the CP-tree in classical frequent pattern mining problems and in additional applications.

6.1. Experimental environment

All programs are written in Microsoft Visual C++ 6.0 and run on Windows XP with a 2.66 GHz CPU and 1 GB memory. Runtime specifies the total execution time, i.e. CPU, I/Os, and it includes tree construction, tree restructuring (for CP-tree only), and mining time. The runtimes reported in figures are the average of multiple runs for each case.

The experiments were performed on several synthetic datasets developed by IBM Almaden Quest research group [19] and real datasets from the UCI Machine Learning Repository [4,5]. Table 2 shows some statistical information about the datasets used for experimental analysis. The fifth and sixth columns represent maximal and average transaction lengths, respectively. The last column shows the percentage of total distinct items that appear in each transaction of each dataset. Since the transactions in a database may vary in size, we use the average transaction length in order to calculate it. For instance, as shown in Table 2, 1.16% of total distinct items appear in each transaction (on average) in dataset *T10I4D100K* and each transaction (on average) in dataset *chess* contains 49.33% of its total distinct items. It is a measure of whether a dataset is sparse or dense. In general, a sparse dataset contains few items per transaction and many distinct items. A dense dataset, in contrast, can be represented with many items per transaction containing only a few distinct items. A dataset can be considered sparse if the value of this parameter is less than 10.0; otherwise, it is a dense dataset. The degree of sparsity or density of a dataset can also be measured using this value. The lesser the value of this parameter, the more sparse the dataset. Similarly, the greater the value of this parameter, the denser the dataset. For example, as shown in the table, *T10I4D100K*, *pumsb**, *kosarak* and *retail* are sparse and the others are dense datasets. Among them, *kosarak* and *chess* are extremely sparse and extremely dense, respectively. Therefore, these statistics provide some rough description about the characteristics of the datasets. *T10I4D100K* is a synthetic dataset generated by [19], while *chess*, *mushroom*, *pumsb**, *connect-4*, *kosarak* and *retail* are all real datasets. Except *retail*, which is provided by Brijs [5], all real datasets are obtained from [4].

We obtain consistent results for all of the above datasets. We divide the experimental analysis into three parts. In the first part we discuss our experimental results on tree structuring techniques; in the next part we show the performance of our CP-tree on frequent pattern mining; and the last part demonstrates the CP-tree's effectiveness and performance on other applications.

6.2. Experiments on tree restructuring

6.2.1. Branch sorting method and path adjusting method

In the first experiment, we compare the performance of the path adjusting method and branch sorting method (BSM) by restructuring trees constructed using sparse and dense datasets separately. The results of the experiments are shown in Figs. 9 and 10. The time in the *y*-axis in each experiment in this subsection is the total time required to construct and restructure the tree, which does not include any mining time. We find that the performance variation between the two methods primarily depends on the tree size, the tree restructuring criteria, and the *degree of disorder* of the *I-lists*. Fig. 9 shows the results on *T10I4D100K* and *chess* when tree restructuring is applied on different sizes of these datasets, i.e. increasing the number of transactions. In *T10I4D100K* dataset, for example, we used both methods individually to restructure the tree constructed

Table 2
Dataset characteristics.

Dataset	Size (MB)	#Trans	#Items	Max TL	Avg TL	AvgTL/#items (×100)
T10I4D100K	3.93	1,00,000	870	29	10.10	1.16
chess	0.34	3196	75	37	37.00	49.33
mushroom	0.83	8124	119	23	23.00	19.33
*pumsb**	10.70	49,046	2088	63	50.48	2.42
connect-4	8.82	67,557	129	43	43.00	33.33
kosarak	30.50	9,90,002	41,270	2498	8.10	0.02
retail	3.97	88,162	16,470	76	10.31	0.06

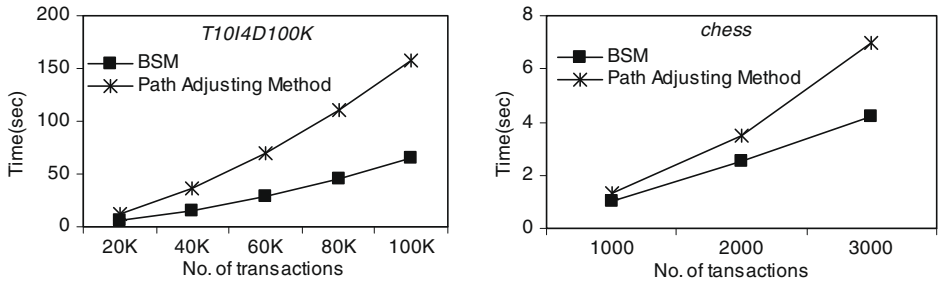


Fig. 9. Restructuring approach comparison (full).

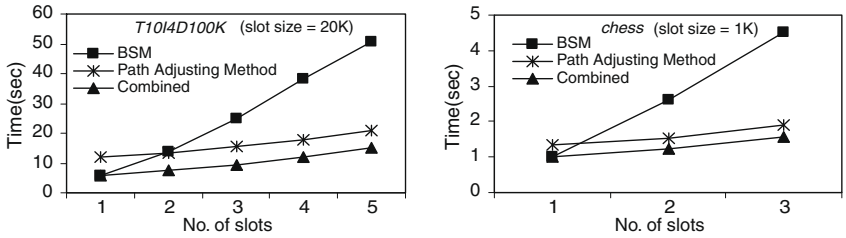


Fig. 10. Restructuring approach comparison (phase-by-phase).

on first 20K transactions, then again performed the same on the tree constructed on first 40K transactions and so on. Similar experiments were also conducted on *chess*. The results clearly reflect that the larger the datasets, the better the BSM performs compared with the path adjusting method for both dense and sparse datasets. For example, when applying the restructuring operation to 20K transactions of *T10I4D100K* and to 1K transactions of *chess*, the path adjusting method performs similarly to BSM. However, the gap increases noticeably for 100K and 3K transactions for the datasets, respectively. As the number of transactions increases, the *degree of disorder* of the *I-lists* and the tree size increase, which are more efficiently handled by the BSM.

However, when applied to an almost sorted tree, the path adjusting method performs comparatively better. To demonstrate this scenario we divided the same datasets (i.e. *T10I4D100K* and *chess*) into several fixed-sized slots (i.e. chunks of transactions) and applied the restructuring methods individually after each slot when constructing the tree on the datasets. As shown in Fig. 10, the *T10I4D100K* dataset is divided into five equal-sized slots with 20K transactions in each slot, then tree restructuring is performed at the insertion of each slot. According to the figure, the path adjusting method offers improved performance with an increasing number of slots. Since the tree is restructured after each slot, except after the first slot, for the remaining slots both of the restructuring methods restructure an almost-sorted tree with reduced tree size due to a higher degree of prefix sharing among tree paths. However, at the initial slot they restructure a tree with a higher *degree of disorder* and larger size, since the rate of new item introduction is generally higher in the first portion of the *DB*. This means that, at that portion, the size of *I-list* generally increases rapidly compared to the rest of *DB*, and the tree offers less prefix sharing among the paths. This is why during the beginning stage, BSM outperforms the path adjusting method as reflected in the figure. We obtain a similar result for the dense *chess* dataset (with slot size of 1K transactions). We also checked the results by varying the slot size.

The above setting (i.e. the restructuring on slotted dataset) can be achieved by dynamically restructuring the tree. The dynamic restructuring technique enables the tree structure to achieve higher prefix sharing at each stage of construction. As a result, the tree restructuring methods are applied to small and compact tree structures, which incur reduced overall restructuring cost compared to the cost required to restructure the tree once at the end of the *DB*. The results shown in Figs. 9 and 10 also reflect such phenomena. Guided by this outcome, we use the BSM in the first slot and the path adjusting method for the remaining slots for most of the datasets, while *N*% heuristic of tree restructuring criteria is followed. On the other hand, when using the top-*K* heuristic, we use BSM when the value of the tree restructuring parameter becomes greater than the set threshold (λ). Results for this combined approach are plotted in the graphs of Fig. 10, which shows the overall gain. Results shown in Figs. 9 and 10 clearly illustrate that for both restructuring approaches, the overall restructuring efficiency increases when applied on the *DB* phase-by-phase, rather than considering the full *DB* at a time. These experiments therefore direct us to carry out dynamic phase-by-phase restructuring by appropriately choosing the methods for restructuring in different phases and accurately selecting the restructuring slots.

6.2.2. Choosing the value of K (in top-K heuristic)

We performed experimental analysis for choosing the appropriate value of *K* while using a top-*K* heuristic as proposed in the tree restructuring criteria in Section 4.2. Fig. 11 shows the experimental results on two sparse and two dense datasets. The *x*-axes of the charts represent different values of *K* in the form of percentage of *I-list* size. For both types of datasets,

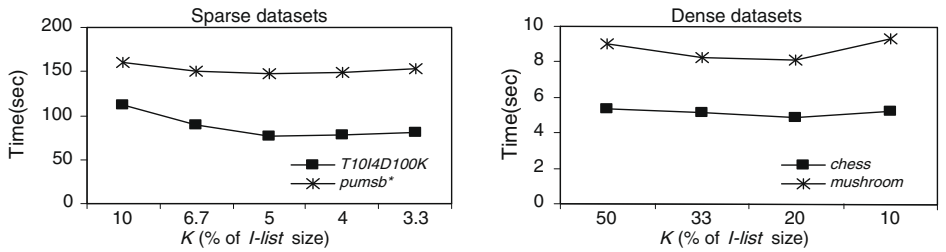


Fig. 11. Tree restructuring parameter.

larger and smaller values for K lead to comparatively poor performance. Therefore, on dense datasets (e.g. *chess* and *mushroom*) we obtain the optimum tree construction and restructuring time if we choose a value of K around 20% of the *I-list* size. A suitable value for K is around 5% for most of the sparse datasets (e.g. *T10I4D100K* and *pumsb**). In each case, the value of the restructuring threshold λ was chosen as the size of *I-list*. The reason for obtaining such different values for K in different types of datasets is that for dense datasets more items in the *I-list* are frequent and for sparse datasets the scenario is reversed.

6.3. CP-tree performance comparison

In this subsection we demonstrate the performance gain that the CP-tree achieves compared with the state-of-the-art algorithms for mining frequent patterns and other applications on datasets with different characteristics. Since it has been shown in [24] that CanTree outperforms other similar algorithms such as FP-tree, AFPIM, and CATS tree, in this paper we compare the performance of CP-tree and CanTree. Moreover, since the performance of CanTree may greatly vary with the order of items and data distribution in transactions, to generalize the performance comparison we compare CP-tree with three versions of CanTree: lexicographic order (CT_L), reverse lexicographic order (CT_R), and item appearance order (CT_A). The runtimes for these algorithms have been computed with lower time granularity of three parameters: construction time, tree restructuring time (only for the CP-tree), and mining time. We report three sets of experiments in which the memory consumption, runtime, and scalability of the CP-tree were compared with CanTrees. Subsequently, we show the effectiveness of the CP-tree in incremental and interactive mining with experimental results.

6.3.1. Compactness of the CP-tree

We conducted experiments to verify the improvement in memory consumption that CP-tree yields due to frequency-descending item ordering over the trees constructed in frequency-independent order. The results of memory consumption of the algorithms for CP-tree and all versions of CanTree on three large (*T10I4D100K*, *pumsb** and *connect-4*) and two small (*mushroom* and *chess*) datasets are shown in Fig. 12. The size of a dataset is determined based on the number of transactions it contains. The y-axes in the graphs report the total amount of memory consumed to store the whole tree structure.

The results indicate that the size of CanTree varies notably depending on data distribution in the dataset and the order of items in the tree. However, the size of the CP-tree is independent of such parameters and in most cases, it is much smaller (both in large and small datasets) than all versions of the CanTree designed in our experiments. It can be noted that for *T10I4D100K*, all trees require a similar amount of memory, but the CP-tree still requires the least. However, in the real datasets *pumsb** and *connect-4*, the CP-tree achieves good compactness compared to other datasets. The gain in memory usage obtained by the CP-tree in small datasets is also notable (as shown in Fig. 12). For both of the real small datasets, in most cases, the CP-tree shows high compactness. However, in the *chess* dataset, the tree size for CT_R is slightly smaller than that of the CP-tree. This reflects that a frequency-descending tree does not guarantee the most compact tree structure at all times. Nevertheless, the CP-tree outperforms CT_R in execution time when used in frequent pattern mining applications as shown in the next subsection. The reason is that only the frequency-descending tree guarantees that there are no *global* infrequent nodes between the *root* and any node of a frequent item in any path in the tree structure. Therefore, as discussed before,

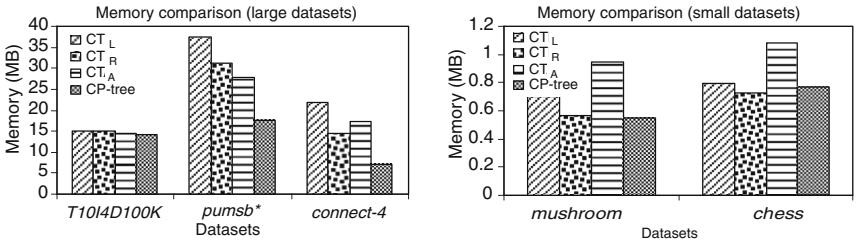


Fig. 12. Compactness of the CP-tree.

the FP-growth mining time on such a tree structure is the minimum. Among the datasets used in this experiment, *T10I4D100K* and *pumsb** are sparse while *connect-4*, *mushroom*, and *chess* are dense. Therefore, we can see that CP-tree requires a minimum amount of memory for almost all datasets irrespective of large/small and sparse/dense. In the following experiment we examine the performance improvement that CP-tree achieves compared to CanTree.

6.3.2. Performance study on execution time

We observed the effect of changing the support threshold on runtime for our CP-tree and CanTrees over different sparse and dense datasets. Fig. 13 shows the runtime for these tree structures over most of the datasets mentioned in Table 2. The *x*-axis in each graph shows the change in the support threshold value in decreasing order and the *y*-axis indicates the overall runtime. We can see from the figure that the running time of the CP-tree structure is rather stable with respect to the decreasing values of the support threshold, and it outperforms the other tree structures on all test datasets. The gain CP-tree obtains is in the mining phase due to its frequency-descending item order, which is clearly shown in Table 3. This table represents the time distribution on tree construction, tree restructuring (for CP-tree only), mining time, and total time for different datasets for the CP-tree and CanTrees. Notice that in the table, we show mining time and total time data for two *min_sup* values (∂_1 and ∂_2) for each dataset. With these records, we show that CP-tree performs better for both small and large support threshold values. It is also interpretable from the table that all the tree structures require a similar amount of time for tree construction. However, for restructuring the tree, the CP-tree needs an additional amount of time, which is not claimed in CanTrees. Most importantly, as shown in the table, for all datasets and even for a considerably larger support threshold, the tree restructuring time sacrifice is extremely negligible compared to the mining time gain. Therefore, as mentioned before, the key achievement of the CP-tree is its remarkable mining time gain with a comparatively insignificant tree restructuring cost. The last two columns in Table 3 represent the overall gain in CP-tree by showing the total time required from tree construction to mining operation for two support thresholds in all datasets.

As shown in Fig. 13 the smaller the *min_sup* value is, the greater the gain the CP-tree achieves over CanTrees for all datasets. This gain is linear for initially higher *min_sup* values, but for some datasets it is multiple orders of magnitude at lower values. For example, in the *chess* dataset the required runtime for the CP-tree with *min_sup* values of 60% and 55% are 131.998 and 546.843 seconds, respectively. For the same *min_sup* values, on the other hand, CT_L, CT_A, and CT_R need 569.75 and 1920.04; 532.047 and 1879.89; 285.094 and 1576.39 s, respectively. The reason for such a performance improvement is that the number of frequent items and the length of frequent patterns increases for smaller values of *min_sup*, which the CP-tree can handle more efficiently and completely.

Notice that with the difference in item ordering, CanTree performs inconsistently in mining time for different datasets. CT_A, for instance, shows better performance in *T10I4D100K*, *kosarak*, and *mushroom* among all CanTrees; however, in *chess*

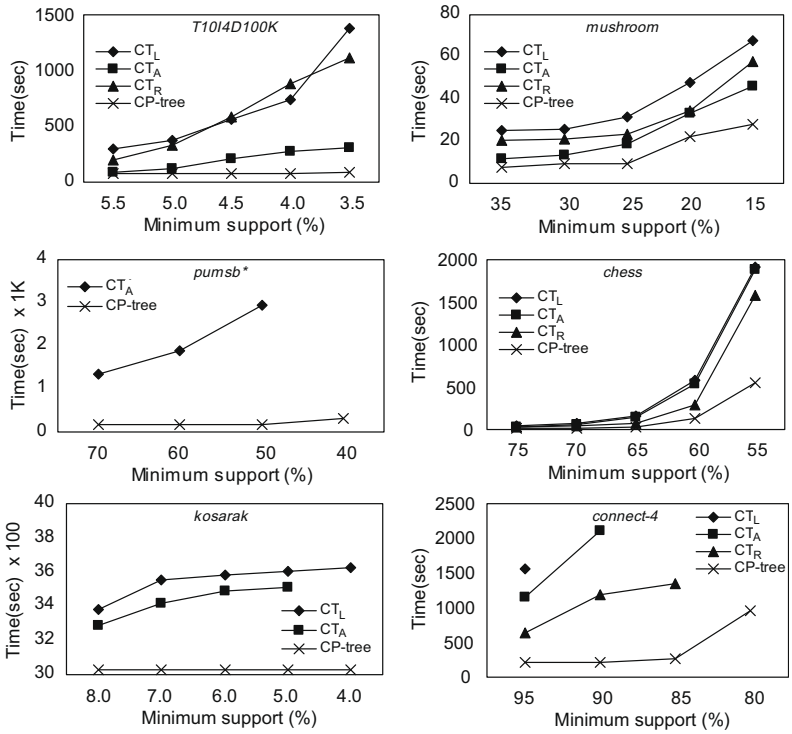


Fig. 13. Performance study on the CP-tree.

Table 3
Runtime distribution.

		Tree construction time (s)	Tree restructuring time (s)	Mining time (s)		Total time (s)	
				∂_1	∂_2	∂_1	∂_2
T10I4D100K	CT_L	61.671	–	240.156	1321.94	301.827	1383.611
∂_1 = 5.5%	CT_A	58.875	–	30.562	253.297	89.437	312.172
∂_2 = 3.5%	CT_R	57.093	–	141.656	1057.8	198.749	1114.893
	CP-tree	61.859	19.111	0.001	2.016	80.971	82.986
mushroom	CT_L	4.828	–	20.282	62.765	25.11	67.593
∂_1 = 35%	CT_A	5.656	–	5.906	40.532	11.562	46.188
∂_2 = 15%	CT_R	4.578	–	15.547	53.188	20.125	57.766
	CP-tree	5.718	1.89	0.25	20.672	7.858	28.28
chess	CT_L	2.797	–	41.64	1917.25	44.437	1920.047
∂_1 = 75%	CT_A	3.359	–	32.687	1876.53	36.046	1879.889
∂_2 = 55%	CT_R	2.796	–	25.719	1573.59	28.515	1576.386
	CP-tree	3.312	1.797	2.797	541.734	7.906	546.843
connect-4	CT_L	75.219	–	1491.73	–	1566.95	–
∂_1 = 95%	CT_A	85.844	–	1071.58	–	1157.42	–
∂_2 = 85%	CT_R	74.546	–	561.453	1278.89	635.999	1353.44
	CP-tree	87.265	129.222	0.219	54.421	216.706	270.908
*pumsb**	CT_A	113.703	–	1218.11	2816.33	1331.813	2930.033
∂_1 = 70%	CP-tree	110.203	59.017	0.016	15.937	169.236	185.157
∂_2 = 50%							
kosarak	CT_L	2910.19	–	467.781	693.984	3377.971	3604.174
∂_1 = 8.0%	CT_A	2837.89	–	452.422	672.187	3290.312	3510.077
∂_2 = 5.0%	CP-tree	2851.73	178.781	0.056	0.187	3030.567	3030.698

and *connect-4* CT_R is the winner. Also note that for some datasets (e.g. *pumsb**, *kosarak*, and *connect-4*) we could not show the complete set of results (Fig. 13 and Table 3) for all versions of CanTree because in these cases the runtimes recorded in experiments were too high to plot. Therefore, based on these experimental observations, we believe that the size and structure of frequency-independent trees depend on data distribution in datasets, their size, and characteristics. Furthermore, these parameters have a significant influence on the performances of these trees. In some cases it was found to be unmanageable to handle smaller support thresholds (e.g. 40% in *pumsb**, 55% in *chess*, 4% in *kosarak*, and 80% in *connect-4*) and/or large datasets (e.g. *pumsb**, *kosarak*, and *connect-4*) with these tree structures. The CP-tree, in contrast, shows consistent performance with the datasets and it is exceptionally effective with lower support thresholds, which is also demonstrated during the scalability study of the CP-tree in the next subsection.

The introduction rate of new frequent patterns is low in sparse datasets on dropping of the support threshold value. This results in a small change in mining time for CP-tree in such datasets. So, we obtain curves for CP-tree that are almost horizontal with respect to the *x*-axis for *T10I4D100K*, *pumsb**, and *kosarak* datasets in Fig. 13. In most cases for such datasets, CanTrees also show similar characteristics, but with a comparatively high mining time compared with CP-tree. In dense datasets *mushroom*, *chess*, and *connect-4*, although for higher support thresholds CP-tree's performance is similar to its performance in sparse datasets, for lower support threshold it requires higher mining times that increase the rate of change in overall runtime. However, in each case, CP-tree still dramatically outperforms all CanTrees because of its frequency-descending tree structure.

In summary, the highly compact frequency-descending CP-tree reduces frequent pattern mining time with FP-growth mining by multiple orders of magnitude with one database scan. We achieve such a structure by dynamically restructuring the tree. In the next experiment, we study the scalability of our CP-tree and compare it with other tree structures considered so far.

6.3.3. Scalability of the CP-tree

We study the scalability of the CP-tree by varying the number of transactions, support threshold value, and number of distinct items in the database. The experimental results are shown in Figs. 14–16. To test the scalability of CP-tree with varying the number of transactions, we use *kosarak*, since it is a huge sparse dataset with a large number of distinct items and transactions (Table 2). We divided this dataset into ten portions of 0.1 million transactions. Then we tested the performance of the CP-tree after accumulating each portion with previous parts and performing frequent pattern mining each time. We fixed the *min_sup* value at 5%. The time in the *y*-axis of Fig. 14 specifies the total time required to build the tree plus corresponding mining time with increasing database size. Tree building time includes tree construction time and tree restructuring time (for CP-tree only) from the first transaction to the number of transactions as labeled on the *x*-axis. Clearly, as the size of the database increases, the overall tree construction and mining time increases. We do not plot the results for CT_R, since for 5% support threshold over *kosarak*, we found this tree structure requires a large amount of execution time. All other

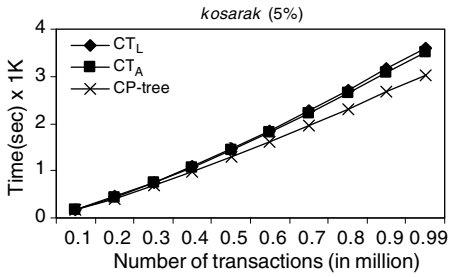


Fig. 14. Scalability with No. of transactions.

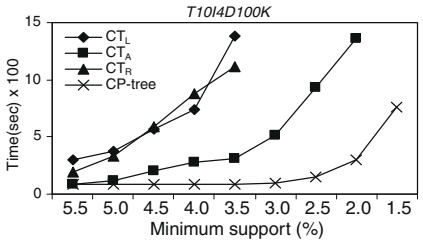
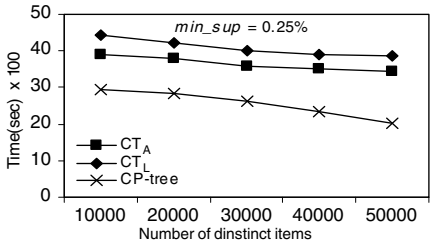
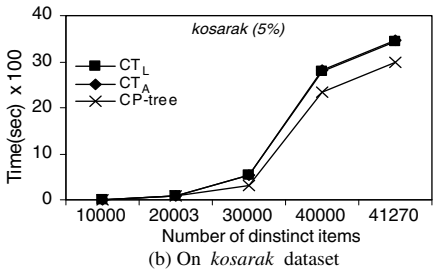


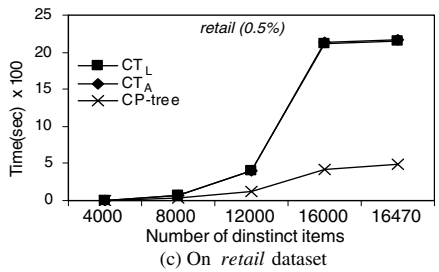
Fig. 15. Scalability with support threshold.



(a) On generated dataset with number of transactions = 1M



(b) On *kosarak* dataset



(c) On *retail* dataset

Fig. 16. Scalability with number of distinct items.

CanTrees and CP-tree show stable performance with a linear increase in runtime as database size increases. However, CP-tree is the fastest of the tree structures.

The runtime of CP-tree and other CanTrees on the synthetic dataset *T10I4D100K* as the support threshold decreases from 5.5% to 1.5% is shown in Fig. 15. Since the dataset is sparse, when the support threshold is high, the frequent patterns are short and the set of such patterns is not large, the advantages of CP-tree over CanTrees are not impressive. However, as the support threshold decreases, the gap becomes wider. Moreover, only the CP-tree can finish the computation for a support threshold of 1.5% within a reasonable amount of time. None of the CanTrees can complete it due to the high mining times required for their frequency-independent item ordering. Among all the CanTrees, CT_A shows the best performance with results for a support threshold as low as 2%. However, it still requires much more time than CP-tree.

Since the CP-tree keeps all items in the tree structure, we also study its scalability with respect to the number of distinct items in the database. Using the dataset generator obtained from [19], we generate a dataset with one million transactions of average transaction length 10. Then, we perform frequent pattern mining on this dataset by varying the number of distinct items (from 10,000 to 50,000) but keeping the *min_sup* fixed at 0.25%. The results on CP-tree, CT_A and CT_L are shown in Fig. 16a where the *y*-axis represents the total runtime. As shown in the figure, as the number of distinct items increases, the overall time in CP-tree decreases. With an increasing number of distinct items, the corresponding tree structure becomes larger but the frequency of each item goes down. Accordingly, the size of F_{DB} decreases. As a result, the mining time drops sharply although there is a slight increase in tree construction and restructuring costs. On the other hand, since the compactness of the CanTrees is much lower than that of the CP-tree, the mining time gain is not as high as that of CP-tree.

CP-tree's scalability on the number of distinct items has also been examined on real datasets *kosarak* and *retail*, since they contain a large number of distinct items (Table 2) compared to almost all real and synthetic datasets commonly used in frequent pattern mining experiments. We choose reasonably low values of support thresholds, i.e. 5% for *kosarak* and 0.5% for *retail*. We report the results for CT_L, CT_A and CP-tree for *kosarak* and *retail* in Fig. 16b and c, respectively. The *x*-axes of the graphs show the increase in distinct items for each dataset and the *y*-axes report the total tree construction time from the beginning of the dataset up to the transaction at which the *m*th distinct item appears (where *m* is a data point on *x*-axis), tree restructuring time (only for CP-tree), and mining time. For both datasets, the increases in runtime for all three tree structures are not linear with the increase in number of distinct items. The reason is that, the increase in dataset size (in number of transactions) with the increase in number of distinct items is not directly correlated. For example, in *kosarak* the dataset size varies 49,502 transactions between the appearances of the 10,000th and the 20,003rd distinct items, and 6,10,167 transactions between the 30,000th and the 40,000th distinct items. In *retail*, the number of transactions between the appearances of the 4000th and the 8000th distinct items, and the 12,000th and the 16,000th distinct items are respectively 6070 and 51,139. Therefore, both CanTrees and the CP-tree require comparatively higher runtime when the database size increases rapidly. Even though CP-tree performs similar to CanTrees for a small number of distinct items in both *kosarak* and *retail*, it is much more efficient as the number of items increases. This is because CP-tree stores the complete set of items in a highly compact fashion, which enables it to create a compact tree structure and results in high mining time gain. On the other hand, the canonical order tree structure restricts the tree size and mining time efficiency of CanTrees. Although CP-tree keeps a complete set of distinct items in the tree structure, it is highly scalable with the size of the distinct item set due to its frequency-descending tree structure.

From the above scalability tests we observe that CP-tree shows much better scalability compared to CanTree with increasing database size, number of distinct items, and decreasing support threshold.

6.4. CP-tree in other applications

Recall from the previous section that the applicability of our CP-tree can be extended to various forms of mining. In this subsection, we examine the effectiveness of CP-tree for incremental mining and interactive mining. It has been shown in [20,10] that AFPIM and CATS tree, respectively, outperform FP-tree in both incremental and interactive mining. The experimental results reported in [24] suggest that CanTree performs better than the FP-tree, AFPIM, and CATS tree in such applications. Therefore, similar to previous experiments, we compare (i) incremental and incremental mining with our proposed CP-tree and (ii) those with CanTrees.

6.4.1. CP-tree in incremental mining

We have tested the effectiveness of the CP-tree in incremental mining with the help of the *kosarak* dataset. We chose this dataset because of its extreme characteristics regarding the number of transactions, number of distinct items, and degree of sparsity as discussed earlier and shown in Table 2. Similar to the scalability study, we divided the *kosarak* dataset into several update portions of 0.1 million transactions in each part. Then we tested the effect of varying the number of incremental updates on the runtime by accumulating each update portion with previous parts by performing frequent pattern mining in each case. The cumulated times reflect the time from the addition of the first transaction until the end of the mining phase. For each mining operation we choose a support threshold value of 5%, which is relatively small for the *kosarak* dataset. We again fail to plot the results of CT_R because the algorithm took too long to complete.

Fig. 17 describes the incremental mining results for CP-tree, CT_L, and CT_A. The *x*-axis in the figure shows the number of database updates and the *y*-axis shows the runtime including time required to update the tree with new data and corresponding mining time. As shown in the figure, during the initial updates, the CP-tree and CanTrees perform similarly with a small performance gain observed in CP-tree. However, this gap becomes wider with an increasing number of updates. During the update, all tree structures have similar maintenance costs, while CP-tree requires significantly less mining time due to restructuring it in frequency-descending item order. The more the updates, the greater the mining time gain CP-tree achieves over CanTrees. For instance, in order to find the complete set of frequent patterns, which is as small as 33 patterns, after the ninth update, the mining times required by CT_L and CT_A are 693.984 and 672.187 s, respectively. On the other hand, CP-tree mines the same set of frequent patterns at this stage in only 0.187 s. The total mining times of CT_L and CT_A from the first update to the last update are 2872.93 and 2429.75 s, respectively. In contrast, the total mining time needed for CP-tree during this period is only 1.5 s. However, CP-tree requires 359.062 s in total tree restructuring overhead. Notice that this overhead is quite insignificant with respect to the gain in the mining phase. Therefore, the CP-tree is highly effective for incremental mining.

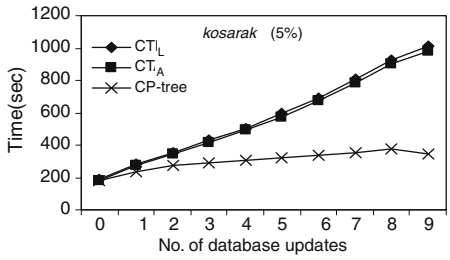


Fig. 17. Incremental mining of the CP-tree.

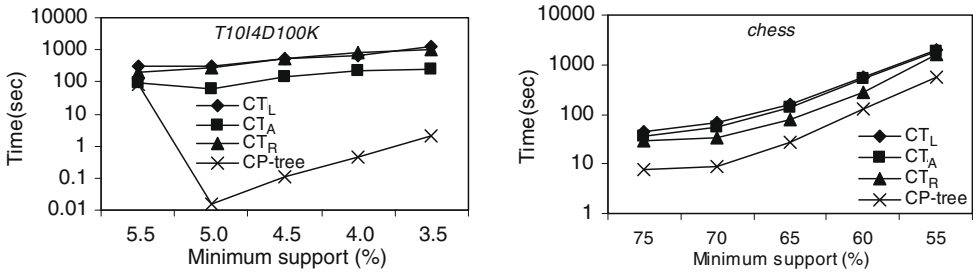


Fig. 18. Interactive mining (number of tree construction = 1).

6.4.2. CP-tree in interactive mining

CP-tree is also suitable for handling interactive mining where the user needs to perform the mining operation on a fixed database with several support thresholds. In this subsection, we present the test results of CP-tree in interactive mining over one sparse (*T10I4D100K*) and one dense (*chess*) dataset. We observed the total execution time required for CP-tree and CanTrees by varying the support threshold, *min_sup*. We only report the results when values of *min_sup* are in decreasing order. For the opposite case (i.e. when values of *min_sup* are in increasing order) the set of frequent patterns can trivially be achieved from the result set of the previous run with a lower *min_sup* value, as discussed in Section 5.2. Therefore, the main issue in interactive mining algorithms is how efficiently they discover the results when support thresholds are in decreasing order. Fig. 18 shows the performance of CP-tree in such a scenario. The *x*-axes indicate the decreasing values of *min_sup*. Like CanTree, CP-tree needs to construct the tree only once. It can then be used for all consecutive mining. We build the trees for the first *min_sup* (5.5% for *T10I4D100K* and 75% for *chess*) value in each case. Other values for the next data points report only mining time with associated support threshold. Notice that CP-tree significantly outperforms all CanTrees in both datasets. This is because when the tree is constructed, consecutive mining operations in the CP-tree with different values of support threshold are always faster than those in CanTrees. For example, over *T10I4D100K* the mining time varies from 0.001 to 2.016 s in the CP-tree while *min_sup* varies from 5.5% to 3.5%. On the other hand, for the same range of *min_sup*, these variations are from 240.156 to 1321.94 s in CT_L, from 30.562 to 253.297 s in CT_A, and from 141.656 to 1057.8 s in CT_R over the same dataset. We also obtain a significant improvement over the *chess* dataset as shown in the figure.

All the experiments demonstrate that despite tree restructuring time overhead, our CP-tree consistently outperforms other state-of-the-art algorithms on classical frequent pattern mining problem and other applications due to its highly compact tree structure. The CP-tree also shows a high degree of scalability for *DB* size, support threshold, and number of distinct item parameters. Moreover, with repeated tree restructuring operations, CP-tree becomes a highly compact tree; therefore, in most cases, it requires less memory compared to frequency-independent trees.

7. Conclusions

In this paper we introduce the dynamic tree restructuring concept. We have proposed CP-tree which, by dynamically applying a tree restructuring technique, achieves a frequency-descending prefix-tree structure with a single-pass and considerably reduces the mining time to discover frequent patterns from a dataset. We have demonstrated that with relatively small tree restructuring overhead, CP-tree achieves a remarkable performance gain in terms of overall runtime. We investigate the reasons behind CP-tree's significant performance gain as well. Moreover, we have shown that the periodic (phase-by-phase) tree restructuring mechanism of the CP-tree significantly reduces the overall tree restructuring cost compared to restructuring the tree for the entire database at one time. Besides, we have proposed the branch sorting method, which is a new efficient tree restructuring technique, and presented guidelines for choosing appropriate values for tree restructuring parameters. In addition, the easy maintenance and constant access to full database information in a highly compact fashion facilitate CP-tree's applicability in interactive, incremental, and other frequent pattern mining applications.

Acknowledgements

We would like to express our deep gratitude to the anonymous reviewers of this paper. Their useful and constructive comments have played a significant role in improving the quality of this work.

This research was supported by the Kyung Hee University Research Fund in 2008. KHU-20080637.

References

[1] R.C. Agarwal, C.C. Aggarwal, V.V.V. Prasad, Depth first generation of long patterns, in: Proceedings of the Sixth ACM SIGKDD Conference, 2000, pp. 108–118.
[2] R. Agrawal, T. Imielinski, A.N. Swami, Mining association rules between sets of items in large databases, in: Proceedings of the ACM SIGMOD Conference on Management of Data, 1993, pp. 207–216.
[3] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings of the 20th International Conference on Vary Large Data Bases, 1994, pp. 487–499.
[4] C.L. Blake, C.J. Merz, UCI Repository of Machine Learning Databases, University of California – Irvine, Irvine, CA, 1998.
[5] T. Brijs, G. Swinnen, K. Vanhoof, G. Wets, Using association rules for product assortment decisions: a case study, in: Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining, San Diego (USA), 1999, pp. 254–260.
[6] C.H. Chang, S.H. Yang, Enhancing SWF for incremental association mining by itemset maintenance, in: Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2003.
[7] E. Chen, H. Cao, Q. Li, T. Qian, Efficient strategies for tough aggregate constraint-based sequential pattern mining, Information Sciences 178 (2008) 1498–1518.
[8] G. Chen, Q. Wei, Fuzzy association rules and the extended mining algorithms, Information Sciences 147 (1–4) (2002) 201–228.
[9] D.W. Cheung, S.D. Lee, B. Kao, A general incremental technique for maintaining discovered association rules, in: Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, 1997, pp. 185–194.
[10] W. Cheung, O.R. Za, Incremental mining of frequent patterns without candidate generation or support constraint, in: Proceedings of the Seventh International Database Engineering and Applications Symposium (IDEAS), 2003.
[11] G. Cormode, S. Muthukrishnan, What's hot and what's not: tracking most frequent items dynamically, ACM Transactions on Database Systems 30 (1) (2005) 249–278.
[12] G. Grahne, J. Zhu, Fast algorithms for frequent itemset mining using FP-Trees, IEEE Transactions on Knowledge and Data Engineering 17 (10) (2005) 1347–1362.
[13] J. Han, H. Cheng, D. Xin, X. Yan, Frequent pattern mining: current status and future directions, Data Mining and Knowledge Discovery (2007). 10th Anniversary Issue.
[14] J. Han, G. Dong, G. Yin, Efficient mining of partial periodic patterns in time series database, in: Proceedings of IEEE International Conference on Data Mining, 1999.
[15] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 2000, pp. 1–12.
[16] T.-P. Hong, C.-W. Lin, Y.-L. Wu, Incrementally fast updated frequent pattern trees, Expert Systems with Applications 34 (4) (2008) 2424–2435.
[17] T. Hu, S.Y. Sung, H. Xiong, Q. Fu, Discovery of maximum length frequent itemsets, Information Sciences 178 (2008) 69–87.
[18] H. Huang, X. Wu, R. Relue, Association analysis with one scan of databases, in: Proceedings of the IEEE International Conference on Data Mining, 2002, pp. 629–632.
[19] IBM, QUEST Data Mining Project, <http://www.almaden.ibm.com/cs/quest>.
[20] J.-L. Koh, S.-F. Shieh, An efficient approach for maintaining association rules based on adjusting FP-tree structures, in: Y.-J. Lee, J. Li, K.-Y. Whang, D. Lee (Eds.), Proceedings of the DASFAA, Springer-Verlag, Berlin Heidelberg, New York, 2004, pp. 417–424.
[21] A.J.T. Lee, R.W. Hong, W.M. Ko, W.K. Tsao, H.H. Lin, Mining spatial association rules in image databases, Information Sciences 177 (7) (2007) 1593–1608.
[22] A.J.T. Lee, C.-S. Wang, An efficient algorithm for mining frequent inter-transaction patterns, Information Sciences 177 (2007) 3453–3476.
[23] Y.-S. Lee, S.-J. Yen, Incremental and interactive mining of web traversal patterns, Information Sciences 178 (2008) 287–306.
[24] C.K. Leung, Q.I. Khan, Z. Li, T. Hoque, CanTree: a canonical-order tree for incremental frequent-pattern mining, Knowledge and Information Systems 11 (3) (2007) 287–311.
[25] C.K.-S. Leung, L.V.S. Lakshmanan, R.T. Ng, Exploiting succinct constraints using FP-trees, SIGKDD Explorer 4 (1) (2002) 40–49.
[26] X. Li, X. Deng, S. Tang, A fast algorithm for maintenance of association rules in incremental databases, in: ADMA, 2006, pp. 56–63.
[27] Q. Li, L. Feng, A. Wong, From intra-transaction to generalized inter-transaction: landscaping multidimensional contexts in association rule mining, Information Sciences 172 (2005) 361–395.
[28] M.-Y. Lin, S.-Y. Lee, Interactive sequence discovery by incremental mining, Information Sciences 165 (2004) 187–205.
[29] G. Liua, H. Lua, J.X. Yub, CFP-tree: a compact disk-based structure for storing and querying frequent itemsets, Information Systems 32 (2007) 295–319.
[30] N.F. Ayan A.U. Tansel E. Akrun, An efficient algorithm to update large itemsets with early pruning, in: Proceedings of the fifty ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 1999, pp. 287 – 291.
[31] J. Pei, J. Han, L.V.S. Lakshmanan, Mining frequent itemsets with convertible constraints, in: A. Buchmann, D. Georgakopoulos (Eds.), Proceedings of the International Conference on Data Engineering, IEEE Computer Society Press, Los Alamitos, CA, 2001, pp. 433–442.
[32] Y.J. Tsay, J.Y. Chiang, An efficient cluster and decomposition algorithm for mining association rules, Information Sciences 160 (1-4) (2004) 161–171.
[33] F. Wua, S.-W. Chiang, J.-R. Linb, A new approach to mine frequent patterns using item-transformation methods, Information Systems 32 (2007) 1056–1072.
[34] S. Zhang, J. Zhang, C. Zhang, EDUA: an efficient algorithm for dynamic database mining, Information Sciences 177 (2007) 2756–2767.