Knowl Inf Syst (2007) 11(3): 287–311
DOI 10.1007/s10115-006-0032-8

Knowledge and Information Systems

REGULAR PAPER

Carson Kai-Sang Leung · Quamrul I. Khan · Zhan Li · Tariqul Hoque

CanTree: a canonical-order tree for incremental frequent-pattern mining

Received: 30 November 2005 / Revised: 23 January 2006 / Accepted: 1 April 2006 / Published online: 5 October 2006

Abstract Since its introduction, frequent-pattern mining has been the subject of numerous studies, including incremental updating. Many existing incremental mining algorithms are Apriori-based, which are not easily adoptable to FP-tree-based frequent-pattern mining. In this paper, we propose a novel tree structure, called *CanTree* (*canonical-order tree*), that captures the content of the transaction database and orders tree nodes according to some canonical order. By exploiting its nice properties, the CanTree can be easily maintained when database transactions are inserted, deleted, and/or modified. For example, the CanTree does not require adjustment, merging, and/or splitting of tree nodes during maintenance. No rescan of the entire updated database or reconstruction of a new tree is needed for incremental updating. Experimental results show the effectiveness of our CanTree in the incremental mining of frequent patterns. Moreover, the applicability of CanTrees is not confined to incremental mining; CanTrees can also be applicable to other frequent-pattern mining tasks including constrained mining and interactive mining.

Keywords Knowledge discovery and data mining · Tree structure · Frequent sets · Incremental mining · Constrained mining · Interactive mining

1 Introduction

Since its introduction [1], the problem of mining association rules–and the more general problem of finding frequent patterns–from large databases has been the subject of numerous studies. These studies can be broadly divided into the

C. K.-S. Leung (✉) · Q. I. Khan · Z. Li · T. Hoque
Department of Computer Science, University of Manitoba, Winnipeg, MB, Canada R3T 2N2
E-mail: kleung@cs.umanitoba.ca

following two categories:

(a) *Functionality*: The central question considered is *what* (kind of rules or patterns) to compute. While some studies [4, 8, 13, 14, 34, 36, 37] in this category considered the data mining exercise in isolation, some others explored how data mining can best interact with (i) the database management system [33, 35] or (ii) the human user. Examples of the latter include constrained mining [6, 7, 9, 15, 22, 25, 27, 30] as well as interactive and online mining [12, 19, 22].
(b) *Performance*: The central question considered is *how* to compute the association rules or frequent patterns as efficiently as possible. Studies in this category can be further classified into several subgroups. The first subgroup consists of fast algorithms based on the level-wise Apriori framework [2]. The second subgroup focuses on performance enhancement techniques like hashing and segmentation [26, 29] for speeding up Apriori-based algorithms. The third subgroup is on *incremental updating*.

With advances in technology, one could easily collect a large amount of data. This, in turn, poses a maintenance problem. Specifically, when new transactions are inserted into an existing database DB and/or when some old transactions are deleted from DB, one may need to update the collection of frequent patterns (e.g., add to the collection those patterns that were previously infrequent in the old database DB but are frequent in the updated database DB'). Algorithms such as FUP [10], FUP$_2$ [11], and UWEP [3] were developed to solve this problem.

In general, the above-mentioned algorithms are Apriori-based, that is, they depend on a generate-and-test paradigm. They compute frequent patterns by generating candidates and checking their frequencies (i.e., support counts) against the transaction database. To improve efficiency of the mining process, Han et al. [17, 18] proposed an alternative framework, namely a tree-based framework. The algorithm they proposed in this framework constructs an extended prefix-tree structure, called *Frequent Pattern tree* (*FP-tree*), to capture the content of the transaction database. Rather than employing the generate-and-test strategy of Apriori-based algorithms, such a tree-based algorithm focuses on frequent pattern growth–which is a restricted test-only approach (i.e., does not generate candidates, and only tests for frequency).

Since the introduction of such an FP-tree-based framework, some studies have been proposed to improve functionality (e.g., interactive FP-tree-based mining [23]) and performance (e.g., FP-tree-based segmentation techniques [28]). So, how about FP-tree-based incremental mining? Recall that algorithms such as FUP [10], FUP$_2$ [11], and UWEP [3] were developed to handle incremental mining in the Apriori-based framework. They cannot be easily adoptable to FP-tree-based incremental mining. Fortunately, some tree-based incremental mining algorithms were recently developed. For example, Cheung and Zaïane [12] proposed the FELINE algorithm with the CATS tree, whereas Koh and Shieh [21] proposed the AFPIM algorithm. The former aims to make the CATS tree (a variant of the FP-tree) compact, and the FELINE algorithm is well suited for *interactive mining* where the database remains unchanged and only the minimum support threshold gets changed. So, it works well in situations that follow the “build once, mine many” principle (e.g., interactive mining), but its efficiency for *incremental mining* (where the database is changed frequently) is unclear. Unlike the FELINE algorithm, the AFPIM algorithm was proposed for incremental mining. Specifically, it was designed to produce an FP-tree for the updated database, in some cases, by adjusting the old FP-tree via the bubble sort. However, in many other cases, it requires rescanning the entire updated database in order to build the corresponding FP-tree.

To summarize, those existing Apriori-based incremental mining algorithms cannot be easily adoptable to FP-tree-based incremental mining. Among those FP-tree-based algorithms, the FELINE algorithm with the CATS tree was mainly designed for interactive mining, where the “build once, mine many” principle holds. However, such a principle does *not* necessarily hold for incremental mining. The AFPIM algorithm was proposed to reduce–but *not* to eliminate–the possibility of rescanning the updated database. Is there any algorithm that aims for incremental mining? Is there any tree structure that is simpler but yet more powerful than the CATS tree? Can we do better than the AFPIM algorithm (e.g., can we avoid rescanning the entire updated database)?

The *key contribution of this work* is the development of a simple, but yet powerful, novel tree structure that aims for incremental mining. More specifically, our proposed tree structure, called *CanTree* (canonical-order tree), captures the content of the transaction database (e.g., the original database, updated databases). When the database is updated (i.e., transactions are inserted, deleted, and/or modified), our algorithm does not need to rescan the entire updated database. Experimental results in Sect. 6 show that frequent-pattern mining with our CanTree is more efficient than that with existing algorithms or structures. Figure 1 summarizes the salient differences between our proposed CanTree and its most relevant work.

In addition to showing the efficiency of our CanTrees for incremental mining, we also discuss the applicability of CanTrees for other frequent-pattern mining tasks such as constrained mining and interactive mining. Moreover, we develop a variant of CanTree to reduce its memory space requirement.

This paper is a revised and expanded version of our ICDM paper [24]. New materials include a discussion on how our proposed CanTree deals with incremental updates involving deletions of transactions (Sect. 3.2), a proposal on a variant of CanTrees–called *CanTries*–for reducing the amount of required memory space (Sect. 4.1), descriptions on additional applicability of our CanTree in constrained mining (Sect. 5.1) and interactive mining (Sects. 5.3 and 5.4), as well as additional experimental results (Sect. 6).

This paper is organized as follows. In the next section, related work is described. Section 3 introduces our CanTree for incremental mining. In Sect. 4, we discuss efficiency and memory issues regarding our CanTrees and propose a variant of CanTrees called CanTries; in Sect. 5, we describe the additional benefits of CanTrees (e.g., for constrained mining, incremental constrained mining, interactive mining, and incremental interactive mining). Section 6 shows experimental results. Finally, conclusions are presented in Sect. 7.

FELINE/ CATS tree:	One scan on the incremental database portion *db* (i.e., inserted, deleted, and/or updated transactions) is required to maintain the CATS tree
AFPIM/ FP-tree:	In the worst case, the AFPIM algorithm requires two scans on the updated database $DB' = DB \cup db$ to update/rebuild the FP-tree
Our proposed CanTree:	Only one scan on the incremental database portion *db* is required to maintain the CanTree
FELINE/ CATS tree:	Items are arranged in descending order of *local* frequency in each path of the CATS tree
AFPIM/ FP-tree:	In the FP-tree, items are arranged in descending order of (*global*) frequency of the updated database DB'
Our proposed CanTree:	In the CanTree, items are arranged according to some *canonical* order, which is unaffected by frequency changes
FELINE/ CATS tree:	Updates to the original database DB may cause swapping and/or merging of tree nodes
AFPIM/ FP-tree:	Updates may cause swapping (via the bubble sort), splitting, and/or merging of tree nodes
Our proposed CanTree:	Updates to the original database DB does *not* lead to any swapping of tree nodes

Fig. 1 Our proposed CanTree vs. the most relevant work

2 Related work

In this section, we discuss two existing FP-tree-based algorithms that handle incremental mining, namely (i) the FELINE algorithm with the CATS tree [12] and (ii) the AFPIM algorithm [21].

2.1 The FELINE algorithm with the CATS tree

Cheung and Zaïane [12] designed the *CATS tree* (compressed and arranged transaction sequences tree) mainly for interactive mining. The CATS tree extends the idea of the FP-tree to improve storage compression, and allows frequent-pattern mining without the generation of candidate itemsets. The aim is to build a CATS tree as compact as possible.

The idea of tree construction is as follows. It requires one database scan to build the tree. New transactions are added at the root level. At each level, items of the new transaction are compared with children (or descendant) nodes. If the same items exist in both the new transaction and the children (or descendant) nodes, the transaction is merged with the node at the highest frequency level. The remainder of the transaction is then added to the merged nodes, and this process is repeated recursively until all common items are found. Any remaining items of the transaction are added as a new branch to the last merged node. If the frequency of a node becomes higher than the frequencies of its ancestors, then it has to swap with the ancestors so as to ensure that its frequency is lower than or equal to the frequencies of its ancestors. Let us consider the following example to gain a better understanding of how the CATS tree is constructed.

Example 2.1 Consider the following database:

	TID	Contents
Original database (DB)	t_1	$\{a, d, b, g, e, c\}$
	t_2	$\{d, f, b, a, e\}$
	t_3	$\{a\}$
	t_4	$\{d, a, b\}$
The first group of insertions (db_1)	t_5	$\{a, c, b\}$
	t_6	$\{c, b, a, e\}$
The second group of insertions (db_2)	t_7	$\{a, b, c\}$
	t_8	$\{a, b, c\}$

Figure 2 shows the resulting CATS tree after each transaction is added. Some important steps are highlighted as follows. Initially, the CATS tree is empty. Transaction $t_1 = \{a, d, b, g, e, c\}$ is then added as it is. When transaction $t_2 = \{d, f, b, a, e\}$ is added, common items (i.e., a, d, b, e) are merged with the existing tree. To do so, node e is swapped with its ancestor g (i.e., e is "moved up"). Since there is no other common items, the remaining item of t_2 (namely, f) is added as a new branch to e. Transactions t_3 and t_4 are added in a similar fashion. When transaction t_5 is added, it finds and merges with common items a and b. Node b is swapped with d, and "moved up". For another common item c, it cannot be swapped and merged. Otherwise, the tree property–the frequency of a node is at least as high as the sum of frequencies of all its children–would be violated. Consequently, item c is added as a new branch (the right branch) to b. Transactions t_6, t_7, and t_8 are added in a similar fashion.

It is interesting to note the following. First, CATS trees keep all items in every transaction. This is different from FP-trees, which keep only those frequent items.

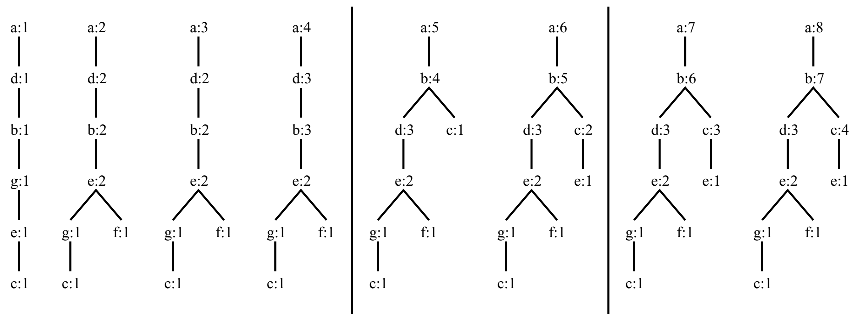


Fig. 2 The CATS tree after each transaction is added (for the FELINE algorithm)

Second, nodes in CATS trees are ordered according to *local* frequency (which may be different from global frequency) in the paths. For example, after transaction t_6 is added, item e is above c on the left branch while the opposite holds on the right branch.

Given that the above tree construction step takes only a single data scan (i.e., constructing the tree without prior knowledge of data), Cheung and Zaïane admitted that their CATS tree is not guaranteed to have the maximal compression. Moreover, the tree compression is sensitive to (a) the ordering of transactions within the database and (b) the ordering of items within each transaction.

In addition, when handling incremental updates, their FELINE algorithm (frequent/large patterns mining with CATS tree) suffers from the problems/weaknesses described below. First, tree construction could be computationally expensive, because it searches for common items and tries to merge the new transaction (the entire one or a portion of it) into an existing tree path when each transaction is added. It checks existing tree paths one by one until a mergeable one is found. Since items are arranged according to their local frequency in the path in the CATS tree, an item may appear above another item on one branch but below it on another branch (e.g., item e appears above item c on the left branch but below c on the right branch in the final tree in Fig. 2). This makes such a search-and-merge costly.

Second, a lot of computation is spent on tree construction with an expectation that the tree is "built once, mined many" (e.g., in interactive mining where the database remains unchanged and only the minimum support threshold *minsup* is changed interactively). However, such a "build once, mine many" principle does not necessarily hold for incremental mining. Specifically, for incremental mining, the database can be changed by insertions, deletions, and/or modifications of transactions. Hence, after a tree is built, it may be mined only once.

Third, extra cost is required for the swapping and/or merging of nodes. See Example 2.1.

Fourth, since items are arranged in descending *local* frequency order in the CATS tree, the FELINE algorithm needs to traverse both upwards *and downwards* in order to include frequent items when forming projected databases (during the mining process). This is different from usual FP-tree-based mining (e.g., using the FP-growth algorithm [17]) where only upward traversal is needed. Specifically, the CATS tree uses the local-frequency ordering (e.g., item e is above item c on the left branch but is below c on the right branch in the final tree in Fig. 2), the downward traversal is needed for completeness (e.g., to avoid missing item c at the leaf of the left branch). Consequently, it costs more to traverse both upwards and downwards! Due to the additional downward traversal, extra work is needed for additional checking to ensure that infrequent items are excluded and those mined items are not doubly counted when forming projected databases!

2.2 The AFPIM algorithm

Koh and Shieh [21] developed the *AFPIM algorithm* (adjusting FP-tree for incremental mining). The key idea of their algorithm can be described as follows. It uses the original notion of FP-trees, in which only "frequent" items are kept in the tree. Here, an item is "frequent" if its frequency is no less than a threshold called *preMinsup*, which is lower than the usual user-support threshold *minsup*. As usual, all the "frequent" items are arranged in descending order of their global frequency. So, insertions, deletions, and/or modifications of transactions may affect the frequency of items. This, in turn, affects the ordering of items in the tree. More specifically, when the ordering is changed, items in the tree need to be adjusted. The AFPIM algorithm does so by swapping items via the bubble sort, which recursively exchanges adjacent items. This can be computationally intensive because the bubble sort needs to apply to all the branches affected by the change in item frequency.

In addition to changes in the item ordering, incremental updating may also lead to the introduction of new items in the tree. This occurs when a previously infrequent item becomes "frequent" in the updated database. When facing this situation, the AFPIM algorithm can no longer produce an updated FP-tree by just adjusting items in the old tree. Instead, it needs to rescan the entire updated database to build a new FP-tree. This can be costly, especially for large databases. To gain a better understanding of the AFPIM algorithm, let us consider the following example.

Example 2.2 Consider the same database as in Example 2.1. Here, we set the threshold *preMinsup* be 35% (and the minimum support threshold *minsup* be 55%). Figure 3 shows the original FP-tree and trees after the first and second groups of insertions. Some important steps are highlighted as follows. The AFPIM algorithm first scans the original database DB once to obtain the global frequency of each item (i.e., $\langle a{:}4, b{:}3, d{:}3, e{:}2\rangle$). It then scans DB the second time for building an FP-tree, in which only "frequent" items are kept. Here, items having frequency at least *minsup* must be "frequent" (because $minsup \geq preMinsup$), but the converse does not hold.

Note that the FP-tree for DB contains only items a, b, d, and e. After transactions t_5 and t_6 are inserted, item c (which had a frequency of 1–i.e., infrequent–in DB) becomes "frequent" with a frequency of 3 in $DB \cup db_1$. Since not all "frequent" items in $DB \cup db_1$ are covered by the FP-tree for DB, the AFPIM algorithm needs to rescan the entire updated database (i.e., $DB \cup db_1$) twice for building a new FP-tree. This could involve a lot of I/Os, especially when the database is large.

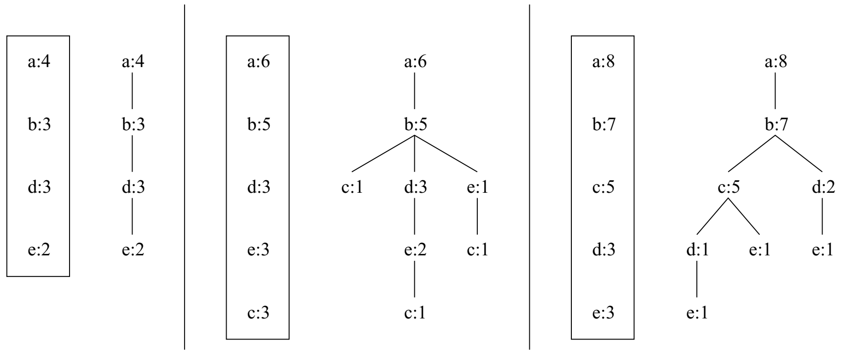


Fig. 3 The FP-trees for DB, $DB \cup db_1$, and $DB \cup db_1 \cup db_2$ (for the AFPIM algorithm)

After the second group of insertions (where transactions t_7 and t_8 are added), the frequency of items changes from $\langle a{:}6, b{:}5, d{:}3, e{:}3, c{:}3\rangle$ in $DB \cup db_1$ to $\langle a{:}8, b{:}7, c{:}5, d{:}3, e{:}3\rangle$ in $DB \cup db_1 \cup db_2$. Consequently, items d, e, and c in the middle FP-tree in Fig. 3 (i.e., the FP-tree for $DB \cup db_1$) need to be swapped using the bubble sort. Besides the swapping of nodes, the AFPIM algorithm may also require the merging and/or splitting of nodes (e.g., after the second group of insertions, c nodes are merged, but d and e nodes are split).

Like the FELINE algorithm, the AFPIM algorithm also suffers from several problems/weaknesses when handling incremental updates. A problem is the amount of computation spent on swapping, merging, and splitting tree nodes. *Swapping* is required because items are arranged according to a frequency-dependent ordering (specifically, descending order of global frequency). So, when the database is updated (e.g., by inserting and/or deleting transactions), frequencies of items may be changed. As a result, the ordering of items needs to be adjusted. This problem is more serious (than FELINE) because AFPIM uses the bubble sort to recursively exchange adjacent tree nodes. The bubble sort is known to be of $O(h^2)$ computation, where h is the number of tree nodes involved in a tree branch. There are many branches in a tree! Furthermore, the swapping of tree nodes often leads to the *merging* and *splitting* of nodes. For instance, the insertion of transactions t_7 and t_8 in Example 2.2 changes the frequency order of items in the tree. Nodes d and e need to swap with c. After swapping, nodes d and e in the path $\langle d, e, c\rangle$ are split into two (i.e., $\langle c, d, e\rangle$ and $\langle d, e\rangle$ as branches of b). At the same time, three descendants of b (i.e., c in paths $\langle c\rangle$, $\langle d, e, c\rangle$, and $\langle e, c\rangle$) are in common, and hence these c nodes are merged and resulted in the rightmost FP-tree in Fig. 3. To summarize, incremental updates to database often result in a lot of swapping, merging, and splitting of tree nodes.

Another problem of the AFPIM algorithm is its requirement for an additional mining parameter *preMinsup*, which is set to a value lower than the usual mining parameter *minsup* (the minimum support threshold). With this additional parameter, only the items whose frequency meets *preMinsup* are kept in the tree. However, it is well known that finding an appropriate value for *minsup* is challenging, which explains the call for interactive mining where the user can interactively adjust or refine *minsup*. So, finding appropriate values for both *minsup* and *preMinsup* can be even more challenging!

3 Our canonical-order tree (CanTree)

Recall from the previous section that, when handling incremental updates, the aforementioned tree-based algorithms–both the FELINE algorithm (with the CATS tree) and the AFPIM algorithm (with the FP-tree)–suffer from several problems/weaknesses. These can be summarized as follows:

1. The FELINE algorithm requires a large amount of computation for searching common items and mergeable paths during the construction of CATS trees. In addition, it needs extra downward traversals during the mining process.
2. The AFPIM algorithm requires an additional mining parameter (namely, *preMinsup*). Finding an appropriate value for this parameter is not easy; it is as challenging as finding an appropriate value for *minsup*.

3. Both FELINE and AFPIM algorithms need lots of swapping, merging, and splitting of tree nodes, because items in the trees are arranged according to a frequency-dependent ordering. So, when the database is updated, item frequencies may have changed. This may result in changes in the ordering.

3.1 An overview of our CanTree

In this section, let us describe our proposed *CanTree* (*canonical-order tree*) and show how it solves the above-mentioned problems. In general, the CanTree is designed for incremental mining. The tree captures the content of the database. In the tree, items are arranged according to some *canonical order*, which can be determined by the user prior to the mining process or at runtime during the mining process. So, the construction of the CanTree requires only one database scan. This is different from the construction of an FP-tree where two database scans are required (one scan for obtaining item frequencies and another one for arranging items in descending frequency order).

Specifically, items in our CanTree can be consistently arranged in lexicographic order or alphabetical order (as in Example 3.1). Alternatively, items can be arranged according to some specific order depending on the item properties (e.g., their price values, their validity of some constraints). For example, items can be arranged according to prefix function order $\mathcal{R}$ or membership order $\mathcal{M}$ for constrained mining. (See Sect. 5.1 for more details on incremental constrained mining.) While the above orderings are frequency-independent, items can also be arranged according to some fixed frequency-related ordering (e.g., in descending order of the global frequency of the "original" database DB). Notice that, in this case, once the ordering is determined (say, for DB), items will follow this ordering in our CanTrees for subsequent updated databases (e.g., $DB \cup db_1$, $DB \cup db_1 \cup db_2, \ldots$) even if the frequency ordering of items in these updated databases is different from DB. With this setting (the canonical ordering of items), there are some nice properties, as described below.

Property 3.1 Items are arranged according to a canonical order, which is a fixed global ordering.

Property 3.2 The ordering of items is unaffected by the changes in frequency caused by incremental updating.

Property 3.3 The frequency of a node in the CanTree is at least as high as the sum of frequencies of all its children.

By exploiting properties of our CanTree, we note the following. Although items are arranged according to a fixed global ordering, our CanTree maintains its structural integrity–for example, the frequency of a node in our CanTree is at least as high as the sum of frequencies of all its children. Due to this canonical order of items, transactions can be easily added to the CanTree without any extensive searches for mergeable paths (as those required by the FELINE algorithm with the CATS tree). As canonical order is fixed, any changes in frequency caused by incremental updating (e.g., insertions, deletions, and/or modifications of transactions) do not affect the ordering of items in the CanTree at all. Consequently, swapping of tree nodes–which often leads to merging and splitting of tree nodes–is *not* needed.

Once the CanTree is constructed, we can mine frequent patterns from the tree in a fashion similar to the FP-growth algorithm. In other words, we can employ a divide-and-conquer approach. Projected databases can be formed by traversing the paths *upwards only*. Since items are consistently arranged according to some canonical order (e.g., lexicographic order, prefix function order $\mathcal{R}$, global frequency order of DB), one can guarantee the inclusion of all *frequent* items using just upward traversals. There is no worry about possible omission or doubly counting of items. Hence, for CanTrees, there is no need for having both upward and downward traversals. This significantly reduces computation by half! For example, forming $\{X\}$-projected databases (where X is $a, b, c, \ldots, g$) requires traversals of 62 nodes in the rightmost CATS tree in Fig. 2; it needs to traverse only 27 nodes in our CanTree!

To summarize, our proposed CanTree solves the problems/weaknesses of the FELINE or AFPIM algorithms as follows:

1. For our CanTree, items are arranged according to some canonical order that is unaffected by the item frequency. Hence, searching for common items and mergeable paths during the tree construction is easy. No extra downward traversals are needed during the mining process.
2. The construction of our proposed CanTree is independent of the threshold values. Thus, it does not require such user thresholds as *preMinsup*.
3. Since items are consistently ordered in our CanTree, any insertions, deletions, and/or modifications of transactions have no effect on the ordering of items in the tree. As a result, swapping of tree nodes–which may lead to merging and splitting of tree nodes–is *not* needed.

This shows how we use our CanTree to solve the problems/weaknesses of the CATS tree/FELINE algorithm and the AFPIM algorithm. To gain a better understanding, let us consider the following example.

Example 3.1 Consider the same database as in Example 2.1. Figure 4 shows the original CanTree and the trees after the first and second groups of insertions. The

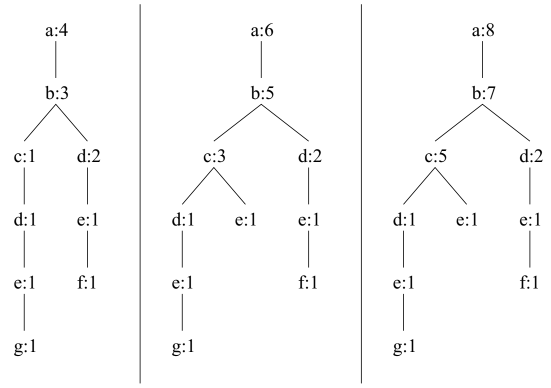


Fig. 4 The CanTree after each group of transactions is added

construction of the original CanTree only requires one database scan. This is different from the construction of an FP-tree where two database scans are required. Like the CATS tree, our CanTree also keeps all items in every transaction. In the CanTree, items are arranged according to some canonical order (say, lexicographical/alphabetical order in this example). Hence, transactions t_1–t_4 can be easily added to the tree, without any extensive searches for mergeable paths (like those in FELINE). As canonical order is unaffected by the frequency order of items at runtime, any changes in frequency caused by incremental updates do not affect the ordering of items in the CanTree at all. Consequently, swapping of tree nodes–which often leads to merging and splitting of tree nodes–is *not* needed.

Once the CanTree is constructed, we can mine frequent patterns from the tree in a divide-and-conquer fashion (similar to FP-growth). We form projected databases (solely for frequent items) by traversing the tree paths upwards only (i.e., no need for having both upward and downward traversals). During the traversals, we only include frequent items. Determination of whether an item is frequent can be easily done by a simple look-up (an O(1) operation) at the header table. There is no worry about possible omission or doubly counting of items.

3.2 Advantages of CanTrees in handling deletions of transactions

So far, we have described in detail how our proposed CanTree solves the problems/weaknesses of the FELINE or AFPIM algorithms. We have shown that transactions can be easily inserted into the CanTree (especially, during incremental updating). In general, when a database is updated, some existing transactions are deleted from the database DB–while some new transactions are inserted into DB–to form an updated database DB'. Hence, in the remaining of this section, we explain in detail how our CanTrees (and its most relevant work) handle deletions of transactions during incremental updating.

First, let us compare our CanTree with the CATS tree/FELINE algorithm. Recall from Sect. 2.1 that the CATS tree keeps items in *local* frequency order in each tree path. So, it may be quite costly and difficult for the corresponding FELINE algorithm to locate those transactions to be removed, due to a large number of possible arrangements of items in tree paths. For instance, items in transaction $t_4 = \{d, a, b\}$ (in Example 2.1) can be arranged in one the following six orderings in the CATS tree: $\langle a, b, d\rangle$, $\langle a, d, b\rangle$, $\langle b, a, d\rangle$, $\langle b, d, a\rangle$, $\langle d, a, b\rangle$, or $\langle d, b, a\rangle$. For transaction $t_6 = \{c, b, a, e\}$, there are 24 possible orderings. In general, there are $h!$ possible orderings for arranging h items contained in the transaction to be removed. Moreover, once the transactions are located and removed from the CATS tree, the FELINE algorithm may need to swap, merge, or split some tree nodes as the local frequency of some tree paths may have changed due to the deletion.

In contrast, our proposed CanTree keeps items in a canonical order. Thus, one can easily locate those transactions to be removed (because items in each tree path follow the same canonical order). Moreover, swapping of nodes–which may in turn lead to merging and splitting of nodes–is *not* needed for our proposed CanTree.

Next, let us compare our CanTree with the AFPIM algorithm. Recall from Sect. 2.2 that the AFPIM algorithm arranges items in the FP-tree according to descending (global) frequency. So, it needs to adjust the tree node via bubble sort if the frequency order of items gets changed. It may also need to rescan the entire updated database to build a new FP-tree if new items are introduced. It is not uncommon that deletions of transactions during incremental updating change the frequency of tree nodes (and thus the ordering). Deletions may also make some previously infrequent items "frequent" in the updated database (due to the decrease in the number of transactions). When facing these situations, the AFPIM algorithm requires lots of swapping, merging, and splitting of tree nodes (and even rebuilding of trees).

In contrast, the ordering of items in our proposed CanTree is unaffected by changes in frequency caused by incremental updates. As a result, no swapping is needed for updating the database!

To summarize, the key success of our proposed CanTree over the CATS tree/FELINE algorithm and the AFPIM algorithm is that the ordering of items in the CanTree is unaffected by any changes in frequency due to deletions (or insertions) of transactions.

4 Discussion on efficiency and memory issues

In this section, we discuss efficiency and memory issues of our proposed CanTrees. On the surface, it appears that our CanTree may take a large amount of memory. For instance, our CanTree may not be as compact as the CATS tree. However, it is important to note that CATS trees do not necessarily reduce computation or time (e.g., a lot of computation spent on finding mergeable paths as well as traversing paths both upwards and downwards). In contrast, our CanTrees significantly reduce computation and time, because they easily find mergeable paths and require only upward path traversals. As a result, our proposed CanTrees provide users with *efficient incremental mining*. Moreover, with modern technology, main memory space is no longer a big concern. This explains why, in this paper, we made the same realistic assumption as in many studies [12, 20, 31, 38] that *we have enough main memory space* (in the sense that the trees can fit into the memory).

Regarding the tree size, our CanTree–like FP-trees and CATS trees–is an extended prefix-tree structure that captures the content of the transaction database. With the path sharing, the number of tree nodes is *no more* than the total number of items in all transactions in the database.

4.1 CanTries: a variant of CanTrees

Although the number of tree nodes is no more than the total number of items in the database, can we reduce the size of our CanTree? The answer is yes, by incorporating the idea of Patricia tries [32] into our CanTree. Here, we propose a variant of CanTrees called *CanTries*. The key idea is as follows. Like FP-trees, CanTrees work well for dense datasets–where there is higher probability for tree nodes to share common paths. However, when datasets are sparse, such probability drops. As a result, the number of tree nodes can get as large as, but not exceeding, the total number of items in the database. When facing this situation, we can use a

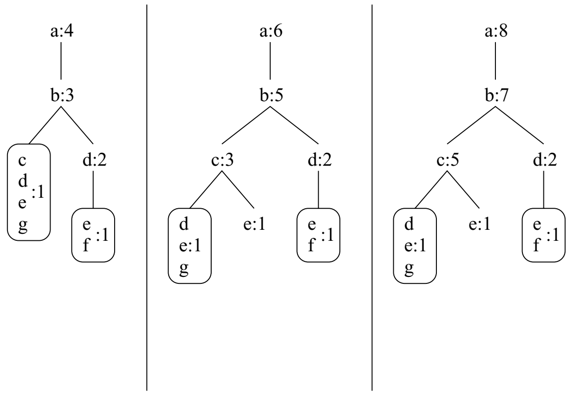


Fig. 5 The CanTries after each group of transactions is added

variant of CanTrees, called *CanTries*, to capture the content of transactions. The structure of a CanTrie is quite similar to that of the CanTree, except that nodes along the same path are coalesced into a mega-node if they have the same frequency. Figure 5 shows how CanTries capture the content of databases (cf. Fig. 4). By coalescing nodes that have the same frequency in the same path, we reduce the memory requirement. For instance, we reduce the number of nodes from 10 nodes in the rightmost CanTree in Fig. 4 to 7 nodes (i.e., 5 regular nodes plus 2 mega-nodes) in the rightmost CanTrie in Fig. 5. Both CanTrees and CanTries capture the same content of the updated database, but the latter require less memory space. For a large (dense or sparse) database, CanTries can substantially reduce the amount of memory required.

While reducing the amount of memory space, we do not sacrifice the runtime. Given that we incorporated the idea of Patricia tries into our CanTries and that the structures of both Patricia tries and CanTries are quite similar, we adopt the PatriciaMine algorithm [32]–which was designed to mine frequent patterns from Patricia tries–to mine frequent patterns from our proposed CanTries. It was known that, for both dense and sparse datasets, mining from Patricia tries (e.g., using the PatriciaMine algorithm) is more efficient than mining from FP-trees [16]. Hence, mining from CanTries can be more efficient than mining from CanTrees.

While CanTries reduce the space requirement without sacrificing efficiency, there could be situations even CanTries representing the databases do not fit into memory. In these situations, recursive projections and partitioning are required to break the CanTries into smaller pieces. As a result, additional performance overhead may incur.

5 Applications of CanTrees

So far, we have shown how efficient our proposed CanTrees are for incremental mining. However, it is important to note that CanTrees also provide us with additional functionalities. For example, CanTrees can be used for (i) *constrained mining*, (ii) *incremental constrained mining*, (iii) *interactive mining*, as well as (iv) *incremental interactive mining*.

5.1 Applicability for constrained mining

Besides incremental mining, frequent-pattern mining has been generalized to many forms since its introduction. These include *constrained mining*. The use of constraints permits user focus and guidance, enables user exploration and control, and leads to effective pruning of the search space and efficient discovery of frequent patterns satisfying the user-specified constraints. Over the past few years, several FP-tree-based constrained mining algorithms have been developed to handle various classes of constraints. For example, the $\mathcal{FIC}$ algorithms [30] handle the so-called convertible constraints (e.g., $C_{conv} \equiv avg(S.Price) \leq 7$ which finds frequent itemsets whose average item price is at most $7). As another example, the FPS algorithm [25] supports the succinct constraints (e.g., $C_{succ} \equiv max(S.Price) \geq 30$ which finds frequent itemsets whose maximum item price is at least $30). The success of these algorithms partly depends on their ability to arrange the items according to some specific order in the FP-trees. More specifically, $\mathcal{FIC}$ arranges items according to prefix function order $\mathcal{R}$ (e.g., arranges the items in ascending order of the price values for the above C_{conv}). Similarly, FPS arranges items according to order $\mathcal{M}$ specifying their membership (e.g., arranges the items in such a way that mandatory items below optional items in the tree for the aforementioned C_{succ}). For lack of space, we do not describe these algorithms further; please refer to the work of Pei et al. [30] and Leung et al. [25] for more details.

These constrained mining algorithms can use CanTrees (instead of FP-trees), and arrange tree items according to some canonical order (e.g., order $\mathcal{R}$ for the $\mathcal{FIC}$ algorithm, order $\mathcal{M}$ for the FPS algorithm). By mining from the CanTree, constrained frequent patterns can be found. Hence, the CanTree can be considered as an alternative tree structure (to the FP-tree) for capturing the content of the database in constrained mining.

5.2 Applicability for incremental constrained mining

To a further extent, our proposed CanTree provides the user with additional functionality to these algorithms, namely *incremental constrained mining*. More precisely, these algorithms can use CanTrees, and arrange tree items according to some canonical order (e.g., order $\mathcal{R}$ for the $\mathcal{FIC}$ algorithm, order $\mathcal{M}$ for the FPS algorithm). By doing so, when transactions are inserted into or deleted from the original database, the algorithms no longer need to rescan the updated database nor do they need to rebuild a new tree from scratch. In addition, no merging or splitting of tree nodes is needed.

5.3 Applicability for interactive mining

In addition to incremental mining and constrained mining, frequent-pattern mining has also been generalized to many other forms, which include *interactive mining*. It is well known that finding an appropriate value for the minimum support threshold *minsup* is challenging. On the one hand, setting *minsup* too low may result in too many frequent patterns. On the other hand, setting *minsup* too high may result in too few frequent patterns. Interactive mining provides the user an opportunity to interactively adjust or refine *minsup*. For interactive mining, the database usually remains unchanged and only *minsup* gets changed. So, it fits the "build once, mine many" principle, where the tree capturing the database content is built once and used in mining with various *minsup* values. By doing so, the tree construction cost can be amortized over several runs of the mining process.

Like CATS trees (used in the FELINE algorithm), our proposed CanTrees also keep all items (regardless whether they are frequent or not). So, once a CanTree is built, it can be mined repeatedly for frequent patterns using different *minsup* values without the need to rebuild the tree. Given that the FELINE algorithm with CATS trees can also handle interactive mining, why would users consider CanTrees? What are advantages of CanTrees? The answers are as follows. First, the CanTree can be considered as an alternative structure (other than the CATS tree) for interactive mining. Second, in terms of tree construction, our proposed CanTree requires less time to build than does the CATS tree. This is because items in the CanTree are arranged in canonical order (which is a fixed global ordering), whereas items in the CATS tree are arranged in descending local frequency order. With the aim of making the tree compact, the tree construction process of CATS trees requires extensive search for common items and mergeable paths during the construction step. This, in turn, leads to lots of node swapping, merging, and splitting. Third, in terms of mining, frequent patterns can be found more efficiently with our proposed CanTree than with the CATS tree. On the surface, it may appear to be contradictory to some readers' expectation. To elaborate, knowing that the CATS trees are usually more compact than are the CanTrees, one would normally expect mining with CATS trees be more efficient than that with CanTrees. However, a close examination reveals the secret. Although CATS trees are usually smaller than the CanTrees, *both upward and downward traversals* on CATS trees are needed to form projected databases during the mining process. Conversely, although CanTrees are usually slightly bigger, there is no need for having both upward and downward traversals on CanTrees–*only upward traversals* are needed–to form projected databases during the mining process. This significantly reduces computation! Let us give a concrete example. The formation of all projected databases for the rightmost CATS tree in Fig. 2 requires traversals of 62 nodes, whereas that for the rightmost CanTrees in Fig. 4 (which captures the same database contents) requires traversals of only 27 nodes!

To summarize, the runtime of interactive mining algorithms mainly depends on the tree construction time and the actual mining time. As shown above (and by our experimental results in Sect. 6), interactive mining with CanTrees requires less time (to construct the tree structure for capturing content of the database and to find frequent patterns with various *minsup* values) than using interactive mining algorithm FELINE with the CATS tree. Thus, the CanTree is a good alternative structure for interactive mining (where the CanTree is built once and mined many times with different *minsup* values).

Along this direction, the mining time using CanTrees can be further reduced as follows. We could cache the frequent patterns mined from the previous round, and reuse them for the current round (when a different *minsup* is used). For example, if the new *minsup* is higher than the old one used in the previous round, we could find frequent patterns satisfying the new *minsup* by using those cached patterns.

Otherwise (i.e., when the new *minsup* is lower than the old one), we could combine the cached patterns together with *"delta" patterns* to form a complete set of frequent patterns satisfying the new *minsup*. We would not need to traverse the entire CanTree, but only relevant portion of the tree, to mine those "delta" patterns. For lack of space, we do not describe this optimization technique further.

5.4 Applicability for incremental interactive mining

In Sect. 3, we showed how efficient our proposed CanTrees are for incremental mining. In the previous section (Sect. 5.3), we showed how efficient our CanTrees are for interactive mining. For both forms of frequent-pattern mining (i.e., incremental mining and interactive mining), the CanTree solves the problems/weaknesses of the CATS tree/FELINE algorithm, and requires less time in finding frequent patterns. Hence, a natural question to ask is, "Can we combine these two forms of mining and use CanTrees for the resulting form (namely, *incremental interactive mining*)?" The answer is yes. By using our CanTrees, the user can find frequent patterns with various *minsup* values from the current database in multiple runs. When such a database is updated (e.g., due to insertions and/or deletions of transactions), the CanTree can efficiently capture the content of the updated database DB' so that new sets of frequent patterns for DB' can be found using the same or a different *minsup* value.

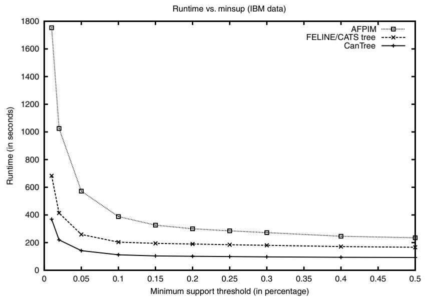
6 Experimental results

In the experiments, we used different databases including (i) several transaction databases generated by the program developed at IBM Almaden Research Center [2], (ii) some real-life databases (e.g., mushroom, connect-4, etc.) from UC Irvine Machine Learning Depository [5], and (iii) some databases from Frequent Itemset Mining Dataset Repository [16]. The results produced are consistent. So, for space consideration, we only show some experimental results in this section.

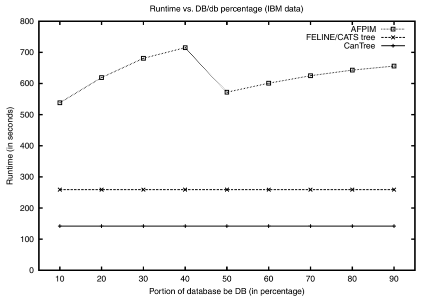
All experiments were run in a time-sharing environment in a 1 GHz machine. The reported figures are based on the average of multiple runs. Runtime includes CPU and I/Os; it includes the time for both tree construction and frequent-pattern mining steps.

6.1 Effectiveness of CanTrees for incremental mining: comparison with the most relevant work

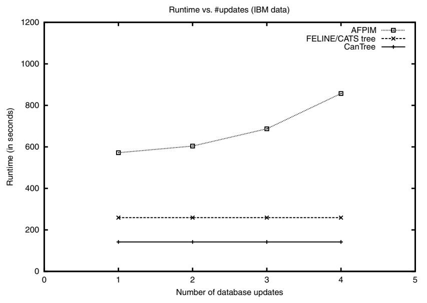
Here, we conducted several experiments by mainly comparing the following algorithms that were implemented in C: (i) the FELINE algorithms with the CATS tree, (ii) the AFPIM algorithm (with the FP-tree), and (iii) the mining algorithm with our proposed CanTree.



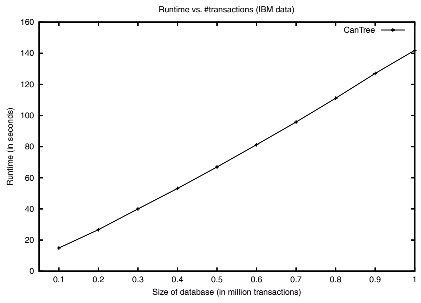
(a) Changes of *minsup*



(b) $DB = p\%$ of the database DB'



(c) Number of updates



(d) Scale-up

Fig. 6 Runtime: CanTree vs. the most relevant work for incremental mining (on the IBM transaction database)

6.1.1 Results on the IBM synthetic transaction database

We cite below the experimental results based on an IBM transaction database, which consists of 1M transactions with an average transaction length of 10 items and a domain of 1,000 distinct items.

In the first experiment, we divided the transaction database DB' into the "original database" DB and the update portion db (i.e., $DB' = DB \cup db$). We tested how the *minsup* values affect the runtime of the algorithms. The y-axis of Fig. 6a shows the runtime, and the x-axis shows *minsup*. When *minsup* decreases, the runtime increases. Note that FP-trees for the AFPIM algorithm are usually smaller than CATS trees and CanTrees, because only "frequent" items are kept in the FP-trees. When *minsup* decreases, the corresponding FP-trees become bigger and take longer to build. Moreover, the lower the *minsup*, the higher is the probability that (i) frequencies of items in the tree get changed (which in turn lead to adjustment of tree nodes) and/or (ii) new items get introduced (which in turn lead to construction of a new tree). As for both CanTrees and CATS trees, their construction is independent of *minsup* because they both keep all items in every transaction. Among them, CATS trees take more time to build than do CanTrees due to extra computation in (i) swapping, merging, and splitting of tree nodes as well as (ii) searching of common items and mergeable tree paths in CATS trees.

As for mining, both AFPIM and our proposal traverse upwards to form projected databases (for frequent items). Among the two, the AFPIM algorithm requires less traversal because the corresponding FP-trees are smaller. As for the FELINE algorithm, it takes longer because it needs to traverse the corresponding CATS trees both upwards *and downwards* when forming projected databases! Hence, although CATS trees are slightly more compact than our CanTrees (e.g., CanTrees are 1.2 times bigger than CATS trees), mining with our CanTrees can be faster (e.g., more than 1.2 times faster) than the FELINE algorithm with CATS trees.

In the second experiment, we again divided DB' into DB and db so that DB be $p\%$ of DB' and db be the remaining $(100 - p)\%$. We varied the percentage p from 10 to 90%. It was observed from Fig. 6b that both CATS trees and our proposed CanTrees are not affected by the various percentage values. However, for the AFPIM algorithm, the higher the percentage p (i.e., larger DB and smaller db), the bigger is the FP-tree for DB. This means a higher probability for the swapping, merging, and splitting of tree nodes (when the frequency order of items gets changed due to incremental updating). However, it also means a lower probability for the introduction of new items (i.e., when some infrequent items become "frequent" due to incremental updating so that the old tree does not cover these items and new tree is needed). Hence, for low $p\%$ (e.g., $p \leq 40\%$), updates caused tree rebuild; for high $p\%$ (e.g., $p \geq 50\%$), updates required node adjustment.

In the third experiment (see Fig. 6c), we divided DB' into DB and several update portions. We tested the number of incremental updates on the runtime. The larger the number of updates, the longer was the runtime for the AFPIM algorithm. This is because frequent updates lead to a higher probability that (i) the item-frequency order before and after the update is different (which leads to swapping, merging, and splitting of tree nodes) and (ii) some new items were introduced after the update (which leads to tree rebuild). This problem can be worsened when using a database with items from a larger domain (e.g., 10,000 distinct domain items).

In the fourth experiment, we tested scalability with the number of transactions. The results in Fig. 6d show that mining with our proposed CanTrees has linear scalability.

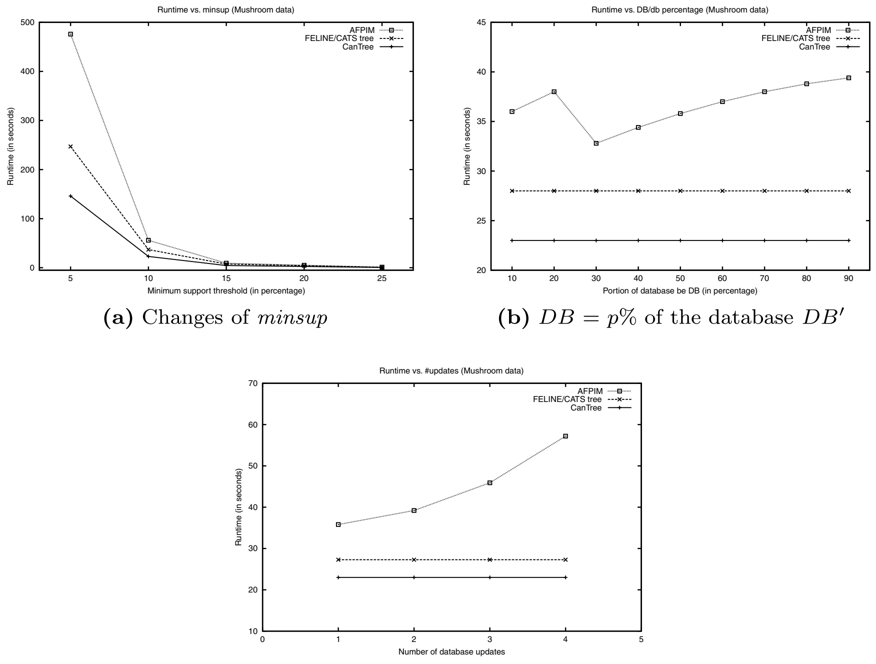
6.1.2 Results on the real-life "mushroom" database

As we conducted several experiments on various databases in addition to the IBM synthetic transaction database, we show here additional results on a different database–namely, the "mushroom" database [5, 16]. Such a database consists of 8,124 transactions with an average transaction length of 23 items and a domain of 119 distinct items.

As observed from Fig. 7, the results on the "mushroom" database is consistent with those on the IBM database. So, for lack of space, we use the IBM database for subsequent experiments.

6.1.3 Effects on the number of distinct items

The previous two sets of experiments were conducted with datasets containing 1,000 and 119 distinct items. Some readers may consider that these datasets are more manageable; they may wonder how would the results change if the number of distinct items is larger? Here, we conducted an experiment by varying the



(a) Changes of *minsup* **(b)** $DB = p\%$ of the database DB'

(c) Number of updates

Fig. 7 Runtime: CanTree vs. the most relevant work for incremental mining (on the "mushroom" database)

number of distinct items in the database but fixing both the database size (1M transactions) and *minsup* (0.05%). When the number of distinct items increases, the runtime decreases. Why? With the increasing number of distinct items, the corresponding CanTrees capturing the database become bigger and contain more tree branches. However, with more distinct items in the database, the frequency of each item drops. Consequently, the number of frequent patterns decreases. In terms of runtime, the tree-construction cost slightly increases with a bigger tree, but the mining cost drops much more with fewer frequent patterns.

6.2 Effectiveness of CanTrees for incremental mining: comparison with other work

6.2.1 Comparison with rebuild

For any incremental techniques, a natural question to ask is whether incremental techniques (in this case, incremental mining with CanTrees) are beneficial? Or, will the use of incremental techniques be worse than without it? More specifically, our proposed CanTree captures the content of the transaction database; it can be easily maintained when database transactions are inserted, deleted, and/or modified during incremental updating. What if one does not use the CanTree? Then, how to handle incremental mining? A natural, and naïve, approach is to use

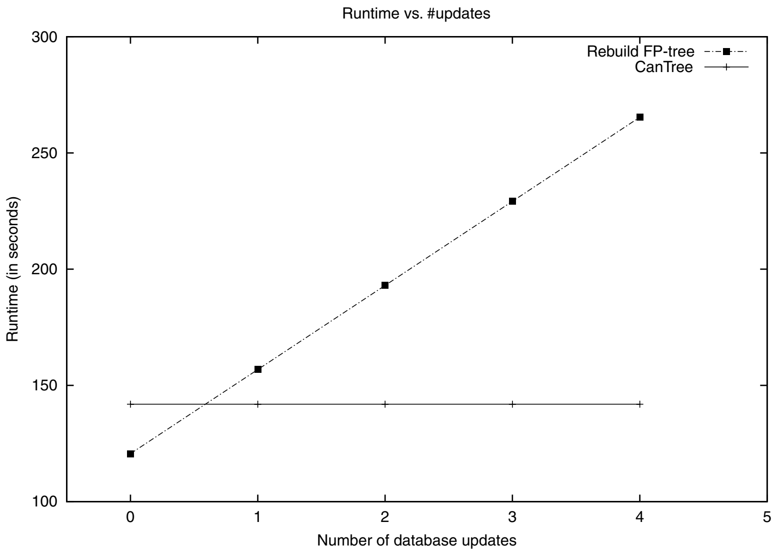


Fig. 8 Runtime for incremental mining: CanTree vs. Rebuild

an FP-tree to capture the content of the original database DB. When DB is updated, discard this FP-tree that captures the content of DB, build a new FP-tree to capture the content of the updated database DB', and use tree-based mining algorithms (e.g., the FP-growth algorithm [17]) to find those frequent patterns from this new FP-tree.

Here, we compared the following techniques that were implemented in C: (i) incremental mining with our proposed CanTree and (ii) incremental mining *without* our proposed CanTree (say, discard the old FP-tree and rebuild a new one). Figure 8 shows the results where we divided the database into DB and several update portions. We tested the effect of varying the number of incremental updates on the runtime. The y-axis of Fig. 8 shows the runtime, and the x-axis shows the number of updates. When there is no update on the database DB, mining without CanTrees wins because the FP-tree is smaller (as it only keeps frequent items). However, when there is an update (e.g., the same database is divided into DB and an update portion db), CanTrees become the winner. The reason is that CanTrees can be easily maintained when transactions are inserted or deleted, and mining can be done using the updated CanTree (for $DB' = DB \cup db$). So, the runtime is just the time to construct the original CanTree (where tree construction requires only one database scan), the time to update/maintain the CanTree (where the maintenance cost is low), plus the mining time. In contrast, for mining *without* the CanTree, the runtime is the time to construct two FP-trees (one for DB and another for DB', and two database scans are required for the construction of each tree) plus the mining time. As expected, when the number of update portions increases, the gap between the two techniques increases (with our proposed technique becomes more superior). Therefore, it is beneficial to use CanTrees, which help incremental mining.

6.2.2 Comparison with Apriori-based algorithms

So far, we have compared with FP-tree-based algorithms. Next, let us compare with FUP, FUP$_2$, and UWEP. The results showed that these Apriori-based algorithms take longer than does the AFPIM algorithm, which in turn takes longer than does mining with CanTrees. This is because Apriori-based algorithms generate lots of candidate patterns, and FP-tree-based algorithms (e.g., the AFPIM algorithm with FP-trees, mining with our CanTrees) avoid generation of these candidates. Moreover, Apriori-based algorithms easily run out of memory when *minsup* is small (e.g., 0.02%). Performance of Apriori is sensitive to *minsup* values.

6.3 Effectiveness of CanTrees for interactive mining

Recall from Sects. 5.1–5.4 that the applicability of our proposed CanTree is not confined to incremental mining, CanTrees can be applicable to various forms of mining including interactive mining. In this section, we show the effectiveness of our proposed CanTrees for interactive mining. Specifically, in this experiment, we compared the following two algorithms: (i) interactive mining with our proposed CanTree and (ii) the FELINE algorithm with the CATS tree.

The runtime of interactive mining algorithms mainly depends on the tree construction time and the actual mining time. The results show that, when comparing with the FELINE algorithm with CATS trees, interactive mining with CanTrees requires less time to (i) construct the tree structure for capturing content of the database and (ii) find frequent patterns using various *minsup* values (see Fig. 9).

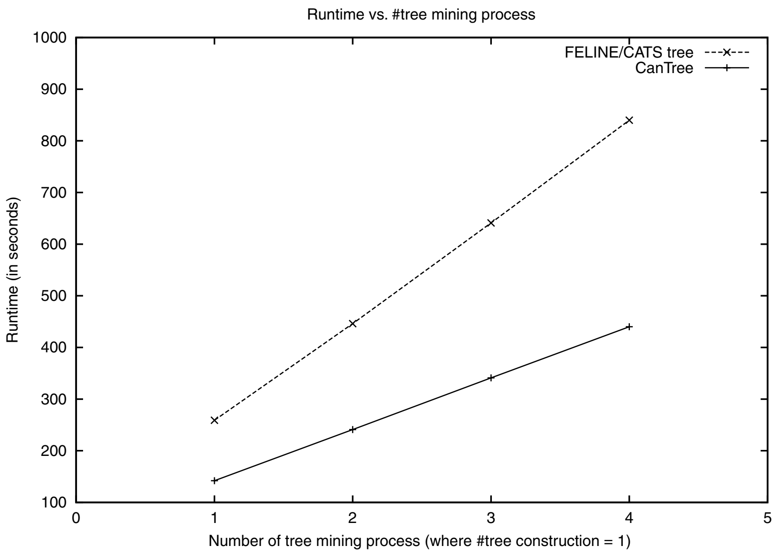


Fig. 9 Runtime for interactive mining: CanTree vs. CATS tree

Hence, the CanTree serves as a good alternative structure for interactive mining (where the CanTree is built once and mined many).

7 Conclusions

A key contribution of this paper is to provide the user with a simple, but yet powerful, tree structure for efficient FP-tree-based incremental mining. Specifically, we proposed and studied the novel structure of CanTree (canonical-order tree). The tree captures the content of the transaction database, and arranges tree nodes according to some canonical order that is unaffected by changes in item frequency. By exploiting its nice properties, the CanTree can be easily maintained when database transactions are inserted, deleted, and/or modified. Specifically, its maintenance does not require merging and/or splitting of tree nodes. It avoids the rescan of the entire updated database or the reconstruction of a new tree for incremental updating. Moreover, our proposed CanTree can also be used for efficient constrained as well as interactive mining of frequent patterns.

Acknowledgements This project is partially sponsored by Science and Engineering Research Canada (NSERC) and The University of Manitoba in the form of research grants.

References

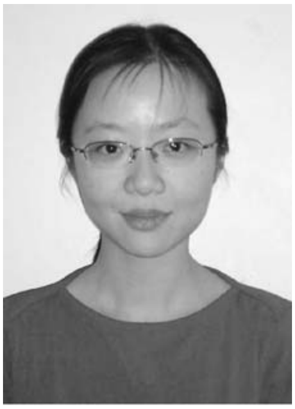
1. Agrawal R, Imielinski T, Swami A (1993) Mining association rules between sets of items in large databases. In: Buneman P, Jajodia S (eds) Proceedings of the SIGMOD 1993. ACM Press, New York, pp 207–216
2. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Bocca JB, Jarke M, Zaniolo C (eds) Proceedings of the VLDB 1994. Morgan Kaufmann, San Francisco, CA, pp 487–499
3. Ayan NF, Tansel AU, Arkun E (1999) An efficient algorithm to update large itemsets with early pruning. In: Fayyad U, Chaudhuri S, Madigan D (eds) Proceedings of the SIGKDD 1999. ACM Press, New York, pp 287–291
4. Bayardo RJ (1998) Efficiently mining long patterns from databases. In: Haas LM, Tiwary A (eds) Proceedings of the SIGMOD 1998. ACM Press, New York, pp 85–93
5. Blake CL, Merz CJ (1998) UCI repository of machine learning databases. University of California – Irvine, Irvine, CA
6. Bonchi F, Giannotti F, Mazzanti A, Pedreschi D (2005) Efficient breadth-first mining of frequent pattern with monotone constraints. KAIS 8(2):131–153
7. Bonchi F, Lucchese C (2004) On closed constrained frequent pattern mining. In: Rastogi R, Morik K, Bramer M, Wu X (eds) Proceedings of the ICDM 2004. IEEE Computer Society Press, Los Alamitos, CA, pp 35–42
8. Brin S, Motwani R, Silverstein C (1997) Beyond market baskets: generalizing association rules to correlations. In: Peckham J (ed) Proceedings of the SIGMOD 1997. ACM Press, New York, pp 265–276
9. Bucila C, Gehrke J, Kifer D, White WM (2002) DualMiner: a dual-pruning algorithm for itemsets with constraints. In: Zaïane OR, Goebel R, Hand D, et al (eds) Proceedings of the SIGKDD 2002. ACM Press, New York, pp 42–51
10. Cheung DW, Han J, Ng VT, Wong CY (1996) Maintenance of discovered association rules in large databases: an incremental updating technique. In: Su SYW (ed) Proceedings of the ICDE 1996. IEEE Computer Society Press, Los Alamitos, CA, pp 106–114
11. Cheung DW, Lee SD, Kao B (1997) A general incremental technique for maintaining discovered association rules. In: Topor RW, Tanaka K (eds) Proceedings of the DASFAA 1997. World Scientific, Singapore, pp 185–194

12. Cheung W, Zaïane OR (2003) Incremental mining of frequent patterns without candidate generation or support constraint. In: Desai BC, Ng W (eds) Proceedings of the IDEAS 2003. IEEE Computer Society Press, Los Alamitos, CA, pp 111–116
13. Coatney M, Parthasarathy S (2005) MotifMiner: efficient discovery of common substructures in biochemical molecules. KAIS 7(2):202–223
14. Fukuda T, Morimoto Y, Morishita S, Tokuyama T (1996) Data mining using two-dimensional optimized association rules: scheme, algorithms, and visualization. In: Jagadish HV, Mumick IS (eds) Proceedings of the SIGMOD 1996. ACM Press, New York, pp 13–23
15. Gade K, Wang J, Karypis G (2004) Efficient closed pattern mining in the presence of tough block constraints. In: Kim W, Kohavi R, Gehrke J, DuMouchel W (eds) Proceedings of the SIGKDD 2004. ACM Press, New York, pp 138–147
16. Goethals B, Zaki MJ (2003) Advances in frequent itemset mining implementations: introduction to FIMI'03. In: Goethals B, Zaki MJ (eds) Proceedings of the FIMI 2003. Available via CEUR-WS.org
17. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Chen W, Naughton JF, Bernstein PA (eds) Proceedings of the SIGMOD 2000. ACM Press, New York, pp 1–12
18. Han J, Pei J, Yin Y, Mao R (2004) Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Min Knowledge Dis 8(1):53–87
19. Hidber C (1999) Online association rule mining. In: Delis A, Faloutsos C, Ghandeharizadeh S (eds) Proceedings of the SIGMOD 1999. ACM Press, New York, pp 145–156
20. Huang H, Wu X, Relue R (2002) Association analysis with one scan of databases. In: Kumar V, Tsumoto S, Zhong N, et al (eds) Proceedings of the ICDM 2002. IEEE Computer Society Press, Los Alamitos, CA, pp 629–632
21. Koh J-L, Shieh S-F (2004) An efficient approach for maintaining association rules based on adjusting FP-tree structures. In: Lee Y-J, Li J, Whang K-Y, Lee D (eds) Proceedings of the DASFAA 2004. Springer-Verlag, Berlin Heidelberg New York, pp 417–424
22. Lakshmanan LVS, Leung CK-S, Ng RT (2003) Efficient dynamic mining of constrained frequent sets. ACM TODS 28(4):337–389
23. Leung CK-S (2004) Interactive constrained frequent-pattern mining system. In: Bernardino J, Desai BC (eds) Proceedings of the IDEAS 2004. IEEE Computer Society Press, Los Alamitos, CA, pp 49–58
24. Leung CK-S, Khan QI, Hoque T (2005) CanTree: a tree structure for efficient incremental mining of frequent patterns. In: Han J, Wah BW, Raghavan V, et al (eds) Proceedings of the ICDM 2005. IEEE Computer Society Press, Los Alamitos, CA, pp 274–281
25. Leung CK-S, Lakshmanan LVS, Ng RT (2002) Exploiting succinct constraints using FP-trees. SIGKDD Explorat 4(1):40–49
26. Leung CK-S, Ng RT, Mannila H (2002) OSSM: a segmentation approach to optimize frequency counting. In: Agrawal R, Dittrich K, Ngu AHH (eds) Proceedings of the ICDE 2002. IEEE Computer Society Press, Los Alamitos, CA, pp 583–592
27. Ng RT, Lakshmanan LVS, Han J, Pang A (1998) Exploratory mining and pruning optimizations of constrained associations rules. In: Haas LM, Tiwary A (eds) Proceedings of the SIGMOD 1998. ACM Press, New York, pp 13–24
28. Ong K-L, Ng WK, Lim E-P (2003) FSSM: fast construction of the optimized segment support map. In: Kambayashi Y, Mohania MK, Wöß W (eds) Proceedings of the DaWaK 2003. Springer-Verlag, Berlin Heidelberg New York, pp 257–266
29. Park JS, Chen M-S, Yu PS (1997) Using a hash-based method with transaction trimming for mining association rules. IEEE TKDE 9(5):813–825
30. Pei J, Han J, Lakshmanan LVS (2001) Mining frequent itemsets with convertible constraints. In: Buchmann A, Georgakopoulos D (eds) Proceedings of the ICDE 2001. IEEE Computer Society Press, Los Alamitos, CA, pp 433–442
31. algorithm for mining frequent closed itemsets. In: Gunopulos D, Rastogi R (eds) Proceedings of the DMKD 2000, pp 21–30 (the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery) is Available via www.cs.ucr.edu/∼dg/DMKD.html
32. Pietracaprina A, Zandolin D (2003) Mining frequent itemsets using Patricia tries. In: Goethals B, Zaki MJ (eds) Proceedings of the FIMI 2003. Available via CEUR-WS.org
33. Sarawagi S, Thomas S, Agrawal R (1998) Integrating association rule mining with relational database systems: alternatives and implications. In: Haas LM, Tiwary A (eds) Proceedings of the SIGMOD 1998. ACM Press, New York, pp 343–354

34. Teng W-G, Hsieh M-J, Chen M-S (2005) A statistical framework for mining substitution rules. KAIS 7(2):158–178
35. Tsur D, Ullman JD, Abiteboul S, et al (1998) Query flocks: a generalization of association-rule mining. In: Haas LM, Tiwary A (eds) Proceedings of the SIGMOD 1998. ACM Press, New York, pp 1–12
36. Tzvetkov P, Yan X, Han J (2005) TSP: mining top-*k* closed sequential patterns. KAIS 7(4):438–457
37. Wang W, Yang J, Yu P (2004) WAR: weighted association rules for item intensities. KAIS 6(2):203–229
38. Zaki MJ, Hsiao C-J (2002) CHARM: an efficient algorithm for closed itemset mining. In: Grossman RL, Han J, Kumar V, et al (eds) Proceedings of the SDM 2002. SIAM, Philadelphia, PA, pp 457–473

Carson K.-S. Leung received his B.Sc.(Honours), M.Sc., and Ph.D. degrees, all in computer science, from the University of British Columbia, Canada. Currently, he is an Assistant Professor at the University of Manitoba, Canada. His research interests include the areas of databases, data mining, and data warehousing. His work has been published in refereed journals and conferences such as *ACM Transactions on Database Systems (TODS)*, *IEEE International Conference on Data Engineering (ICDE)*, and *IEEE International Conference on Data Mining (ICDM)*.

Quamrul I. Khan received his B.Sc. degree in computer science from North South University, Bangladesh, in 2001. He then worked as a Test Engineer and a Software Engineer for a few years before he started his current M.Sc. degree program in computer science at the University of Manitoba under the academic supervision of Dr. C. K.-S. Leung.



Zhan Li received her B.Eng. degree in computer engineering from Harbin Engineering University, China, in 2002. Currently, she is pursuing her M.Sc. degree in computer science at the University of Manitoba under the academic supervision of Dr. C. K.-S. Leung.

Tariqul Hoque received his B.Sc. degree in computer science from North South University, Bangladesh, in 2001. Currently, he is pursuing his M.Sc. degree in computer science at the University of Manitoba under the academic supervision of Dr. C. K.-S. Leung.