# CanTree: A Tree Structure for Efficient Incremental Mining of Frequent Patterns

Carson Kai-Sang Leung* Quamrul I. Khan Tariqul Hoque
*The University of Manitoba, Winnipeg, MB, Canada*
{*kleung, qikhan, thoque*}*@cs.umanitoba.ca*

## 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 (**Can**onical-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.*

## 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 following two categories.

(a) *Functionality*: The central question considered is *what* (kind of rules or patterns) to compute. While some studies [4, 6, 11] 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 [27, 28] or (ii) the human user. Examples of the latter include constrained mining [5, 7, 12, 18, 20, 22, 25] and interactive and online mining [10, 15, 18].

(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 levelwise Apriori framework [2]. The second subgroup focuses on performance enhancement techniques like hashing and segmentation [21, 24] for speeding up Apriori-based algorithms. The third subgroup is on *incremental updating*.

*Person handling correspondence: C.K.-S. Leung.

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 [8], $FUP_2$ [9], 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. [13, 14] 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 [19]) and performance (e.g., FP-tree based segmentation techniques [23]). So, how about FP-tree based incremental mining? Recall that algorithms such as FUP [8], $FUP_2$ [9], 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 [10] proposed the FELINE algorithm with the CATS tree, whereas Koh and Shieh [17] 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 thresh-

Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM'05)
1550-4786/05 $20.00 © 2005 IEEE

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

**Figure 1. Our proposed CanTree vs. the most relevant work.**

old 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 the FP-tree of the updated database, in some cases, by adjusting the old tree via the bubble sort. However, in many other cases, it requires rescanning the entire updated database $DB'$ 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 (i.e., 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 for maintaining frequent patterns found in the updated database. More specifically, we propose a novel tree structure, called *CanTree* (CANonical-order TREE), that aims for incremental mining. The tree captures the content of the transaction database. 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 Section 5 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.

This paper is organized as follows. In the next section, related work is described. Section 3 introduces our CanTree for incremental mining. In Section 4, we discuss the additional benefits of CanTrees (e.g., for incremental *constrained* mining). Section 5 shows experimental results. Finally, conclusions are presented in Section 6.

## 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 [10] and (ii) the AFPIM algorithm [17].

### 2.1. The FELINE Algorithm with the CATS Tree

Cheung and Zaïane [10] 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 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 1** *Consider the following database:*

| | | *TID* | *Contents* |
|---|---|---|---|
| $DB$ | *Original DB* | $t_1$ | $\{a, d, b, g, e, c\}$ |
| | | $t_2$ | $\{d, f, b, a, e\}$ |
| | | $t_3$ | $\{a\}$ |
| | | $t_4$ | $\{d, a, b\}$ |
| $db_1$ | *1st group of insertions* | $t_5$ | $\{a, c, b\}$ |
| | | $t_6$ | $\{c, b, a, e\}$ |
| $db_2$ | *2nd group of insertions* | $t_7$ | $\{a, b, c\}$ |
| | | $t_8$ | $\{a, b, c\}$ |

*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,*

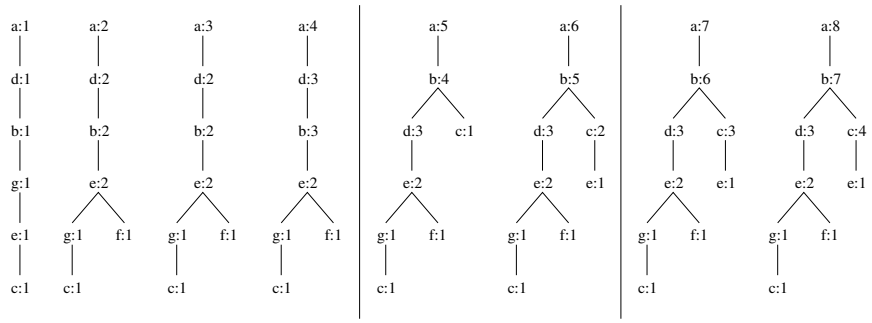


**Figure 2. The CATS tree after each transaction is added (for the FELINE algorithm).**

*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 its children—would be violated. Consequently, $c$ 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. Second, nodes in CATS trees are ordered according to* local *frequency in the paths. For example, after $t_6$ is added, $e$ 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 (e.g., $e$ in Example 1) may appear above another item (e.g., $c$) on one branch, but below it on another branch. 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 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 1.

Fourth, since items are arranged in descending *local* frequency order in the CATS tree. So, when forming projected databases (during the mining process), the FELINE algorithm needs to traverse both upwards *and downwards* to include frequent items. This is different from usual FP-tree mining (e.g., using the FP-growth algorithm [13]) where only upward traversal is needed. Specifically, the CATS tree uses the local-frequency ordering (e.g., item $e$ is above $c$ on the left branch but is below $c$ on the right branch in the final tree in Example 1), the downward traversal is needed for completeness (e.g., to avoid missing item $c$ at the leave of the left branch). Consequently, it costs more to traverse both upwards and downwards! Due to the additional downward traversal, extra work is need for additional checking to ensure that infrequent items as well as those mined items are not doubly-counted when forming projected databases!

## 2.2. The AFPIM Algorithm

Koh and Shieh [17] 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

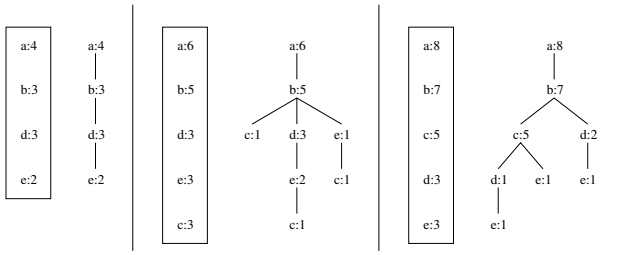


**Figure 3. The FP-trees for $DB$, $DB \cup db_1$, and $DB \cup db_1 \cup db_2$ (for the AFPIM algorithm).**

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** *Consider the same database as in Example 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 the 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 of $DB$, the AFPIM algorithm needs to rescan the entire updated database (i.e., $DB \cup db_1$) twice for building a new tree. This could involve a lot of I/Os, especially when the database is large.*

*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 tree in Figure 3 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 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 it 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 changes the frequency order of items in the tree. Nodes $c$ need to swap with nodes $d$ and $e$. After swapping, nodes $d$ and $e$ in 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 children of $b$ (i.e., $c$ in paths $\langle c\rangle$, $\langle c, d, e\rangle$, and $\langle c, e\rangle$) are in common, and hence the $c$ nodes are merged and resulted in the rightmost FP-tree in Figure 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:

(i) 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.

(ii) 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 that for *minsup*.

(iii) 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 results in changes in the ordering.

In the remaining of 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 construction of the CanTree only requires 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). In our CanTree, 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. Specifically, items can be consistently arranged in lexicographic order or alphabetical order (as in Example 3). 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 Section 4 for more details on incremental constrained mining.) While the above orderings are frequency-independent, items can also 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 subsequently updated databases (e.g. $DB \cup db_1$, $DB \cup db_1 \cup db_2$, ...) even 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 1** *The ordering of items is unaffected by the changes in frequency caused by incremental updates.* ■

**Property 2** *The frequency of a node in the CanTree is at least as high as the sum of frequencies of its children.* ■

By exploiting properties of our CanTree, we note the following. Transactions can be easily added to the CanTree without any extensive searches for mergeable paths (like those in FELINE). As canonical order is fixed, any changes in frequency caused by incremental updates (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 FP-growth. In other words, we employ a divide-and-conquer approach. We form projected databases 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 $a, b, c, ..., g$) requires traversals of 62 nodes in the rightmost CATS tree in Figure 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:

(i) 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.

(ii) The construction of our proposed CanTree is independent of the threshold values. Thus, it does not require such user thresholds as *preMinsup*.

(iii) 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.

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

**Example 3** *Consider the same database as in Example 1. Figure 4 shows the original tree and the trees after the first and second groups of insertions. The 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 tree, items are arranged according to some canonical order (say, lexicographical/alphabetical order in this example). Hence, transactions $t_1$ to $t_4$ 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 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)*

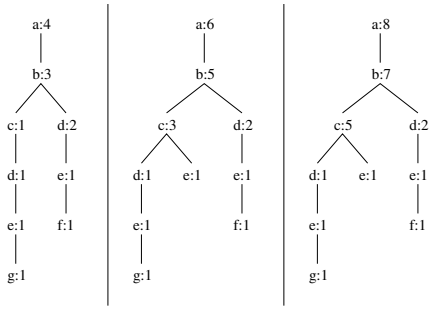


**Figure 4. The CanTree after each group of transactions is added.**

*at the header table. There is no worry about possible omission or doubly-counting of items.* ■

# 4. Discussion

In this section, we discuss two issues: (i) the applicability of the proposed CanTree for incremental *constrained* mining and (ii) efficiency and memory issues regarding our CanTrees.

## 4.1. Applicability for Constrained Mining

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 **incremental *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 [25] 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 [20] 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 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. [25] and Leung et al. [20] for more details.

Our proposed CanTree provides the user with additional functionality to these algorithms, namely *incremental* constrained mining. More precisely, these 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 so doing, 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.

## 4.2. Efficiency and Memory Issues

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 corresponding 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 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 (constrained or unconstrained) 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 [10, 16, 26, 29] 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 number of items in the database.

For situations where the CanTree representing $DB'$ does not fit into memory, recursive projections and partitioning are required to break $DB'$ into smaller pieces. As a result, additional performance overhead may incur.

# 5. Experimental Results

In the experiments, we used (i) several transaction databases generated by the program developed at IBM Almaden Research Center [2] and (ii) some real-life databases from UC Irvine Machine Learning Depository. The results produced are consistent. So, for lack of space, we cite below the experimental results based on an IBM transaction database, which consists of 1M records with an average transaction length of 10 items and a domain of 1000 items.

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. In the experiments, we mainly compared the following algorithms that were implemented in C:

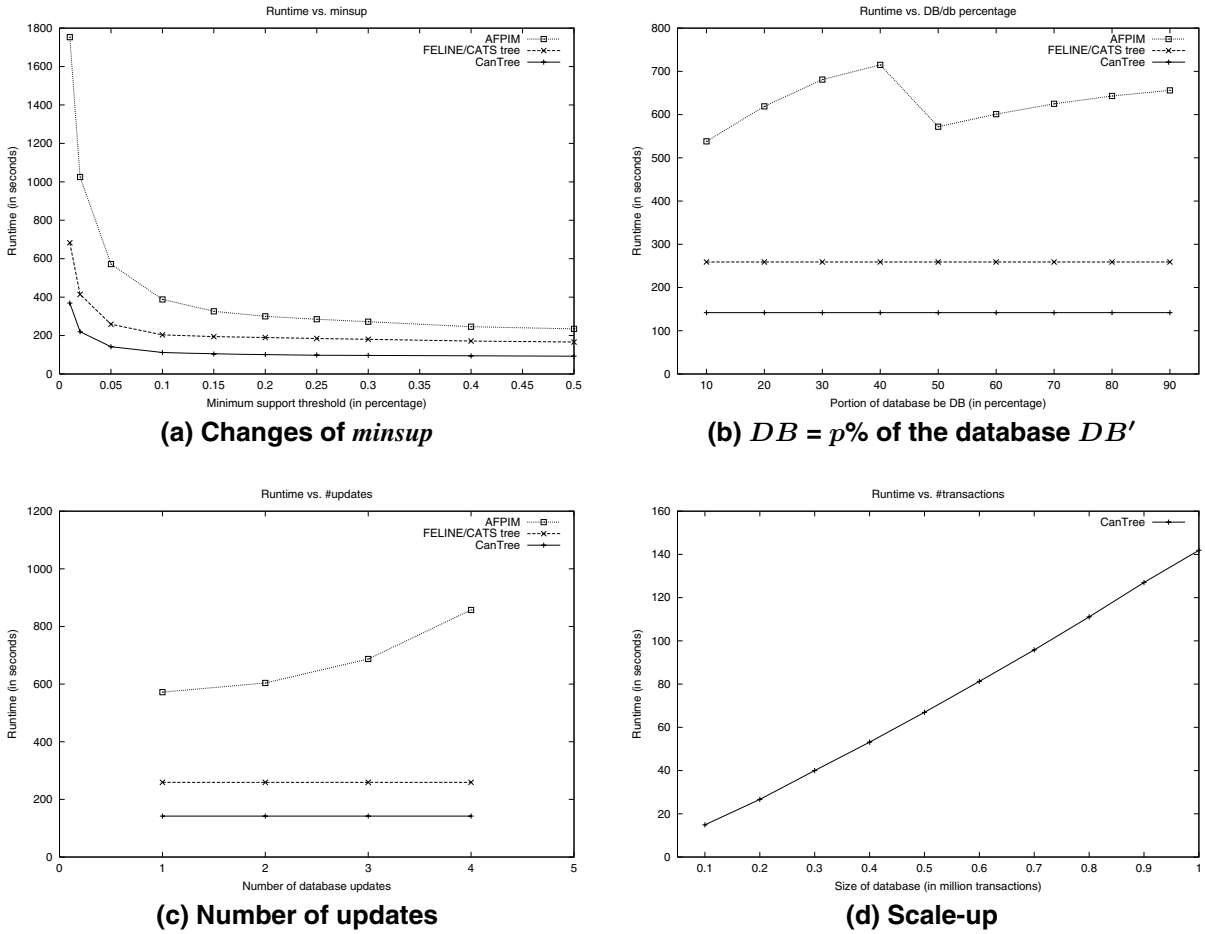


**Figure 5. Runtime: CanTree vs. the most relevant work for incremental mining.**

(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.

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 Figure 5(a) shows the runtime, and the x-axis shows *minsup*. When *minsup* decreased, the runtime increased. Note that FP-trees for the AFPIM algorithm were usually smaller than CATS trees and CanTrees, because only "frequent" items were kept in the FP-trees. When *minsup* decreased, the corresponding FP-trees became bigger and took longer to build. Moreover, the lower the *minsup*, the higher was the probability that (i) frequency order of items in the tree got changed (which, in turns, led to adjustment of tree nodes) and/or (ii) new items got introduced (which, in turns, led to construction of a new tree). As for both CanTrees and CATS trees, their construction was independent of *minsup* because they both kept all items in every transaction. Among them, CATS trees took more time to build than 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 traversed upwards to form projected databases (for frequent items). Among the two, the AFPIM algorithm required less traversal because the corresponding FP-trees were smaller. As for the FELINE algorithm, it took longer because it needed to traverse the corresponding CATS trees both upwards *and downwards* when forming projected databases! Hence, although CATS trees were slightly more compact (e.g., our CanTree was 1.2 times bigger than CATS trees) than our CanTrees, mining with our CanTrees could 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 Figure 5(b) that both CATS trees and our proposed CanTrees were not affected by the varying percentage values. However, for the AFPIM algorithm, the higher the percentage $p\%$ (i.e., larger $DB$ and smaller $db$), the bigger was 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 got changed due to incremental updates). However, it also means a lower probability for the introduction of new items (i.e., when some infrequent items became "frequent" due to incremental updates in such a way that the old tree did not cover these items and new tree was 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 Figure 5(c)), we divided $DB'$ into $DB$ and several update portions. We tested the number of incremental updates on the runtime. The higher the number of updates, the longer was the runtime for the AFPIM algorithm. This was because frequent updates led to a higher probability that (i) the item-frequency order before and after the update was different (i.e., 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 Figure 5(d) show that mining with our proposed CanTrees had linear scalability.

## 6. 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 incremental constrained mining of frequent patterns.

## Acknowledgement

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] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. SIGMOD 1993*, pp. 207–216.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. VLDB 1994*, pp. 487–499.

[3] N.F. Ayan, A.U. Tansel, and E. Arkun. An efficient algorithm to update large itemsets with early pruning. In *Proc. SIGKDD 1999*, pp. 287–291.

[4] R.J. Bayardo. Efficiently mining long patterns from databases. In *Proc. SIGMOD 1998*, pp. 85–93.

[5] F. Bonchi and C. Lucchese. On closed constrained frequent pattern mining. In *Proc. ICDM 2004*, pp. 35–42.

[6] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: generalizing association rules to correlations. In *Proc. SIGMOD 1997*, pp. 265–276.

[7] C. Bucila, J. Gehrke, D. Kifer, and W.M. White. DualMiner: a dual-pruning algorithm for itemsets with constraints. In *Proc. SIGKDD 2002*, pp. 42–51.

[8] D.W. Cheung, J. Han, V.T. Ng, and C.Y. Wong. Maintenance of discovered association rules in large databases: an incremental updating technique. In *Proc. ICDE 1996*, pp. 106–114.

[9] D.W. Cheung, S. D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proc. DASFAA 1997*, pp. 185–194.

[10] W. Cheung and O.R. Zaïane. Incremental mining of frequent patterns without candidate generation or support constraint. In *Proc. IDEAS 2003*, pp. 111–116.

[11] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining using two-dimensional optimized association rules: scheme, algorithms, and visualization. In *Proc. SIGMOD 1996*, pp. 13–23.

[12] K. Gade, J. Wang, and G. Karypis. Efficient closed pattern mining in the presence of tough block constraints. In *Proc. SIGKDD 2004*, pp. 138–147.

[13] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. SIGMOD 2000*, pp. 1–12.

[14] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: a frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, **8**(1), pp. 53–87, Jan. 2004.

[15] C. Hidber. Online association rule mining. In *Proc. SIGMOD 1999*, pp. 145–156.

[16] H. Huang, X. Wu, and R. Relue. Association analysis with one scan of databases. In *Proc. ICDM 2002*, pp. 629–632.

[17] J.-L. Koh and S.-F. Shieh. An efficient approach for maintaining association rules based on adjusting FP-tree structures. In *Proc. DASFAA 2004*, pp. 417–424.

[18] L.V.S. Lakshmanan, C.K.-S. Leung, and R.T. Ng. Efficient dynamic mining of constrained frequent sets. *ACM TODS*, **28**(4), pp. 337–389, Dec. 2003.

[19] C.K.-S. Leung. Interactive constrained frequent-pattern mining system. In *Proc. IDEAS 2004*, pp. 49–58.

[20] C.K.-S. Leung, L.V.S. Lakshmanan, and R.T. Ng. Exploiting succinct constraints using FP-trees. *SIGKDD Explorations*, **4**(1), pp. 40–49, June 2002.

[21] C.K.-S. Leung, R.T. Ng, and H. Mannila. OSSM: a segmentation approach to optimize frequency counting. In *Proc. ICDE 2002*, pp. 583–592.

[22] R.T. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations Rules. In *Proc. SIGMOD 1998*, pp. 13–24.

[23] K.-L. Ong, W.K. Ng, and E.-P. Lim. FSSM: fast construction of the optimized segment support map. In *Proc. DaWaK 2003*, pp. 257–266.

[24] J.S. Park, M.-S. Chen, and P.S. Yu. Using a hash-based method with transaction trimming for mining association rules. *IEEE TKDE*, **9**(5), pp. 813–825, Sept./Oct. 1997.

[25] J. Pei, J. Han, and L.V.S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proc. ICDE 2001*, pp. 433–442.

[26] J. Pei, J. Han, and R. Mao. CLOSET: an efficient algorithm for mining frequent closed itemsets. In *Proc. DMKD 2000*, pp. 21–30.

[27] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. In *Proc. SIGMOD 1998*, pp. 343–354.

[28] D. Tsur, J.D. Ullman, S. Abiteboul, C. Clifton, R. Motwani, S. Nestorov, and A. Rosenthal. Query flocks: a generalization of association-rule mining. In *Proc. SIGMOD 1998*, pp. 1–12.

[29] M.J. Zaki and C.-J. Hsiao. CHARM: an efficient algorithm for closed itemset mining. In *Proc. SDM 2002*.