Speed-up Iterative Frequent Itemset Mining with Constraint Changes

Gao Cong Bing Liu
School of Computing, National University of Singapore, Singapore 117543
E-mail: {conggao, liub}@comp.nus.edu.sg

Abstract

Mining of frequent itemsets is a fundamental data mining task. Past research has proposed many efficient algorithms for the purpose. Recent work also highlighted the importance of using constraints to focus the mining process to mine only those relevant itemsets. In practice, data mining is often an interactive and iterative process. The user typically changes constraints and runs the mining algorithm many times before satisfied with the final results. This interactive process is very time consuming. Existing mining algorithms are unable to take advantage of this iterative process to use previous mining results to speed up the current mining process. This results in enormous waste in time and in computation. In this paper, we propose an efficient technique to utilize previous mining results to improve the efficiency of current mining when constraints are changed. We first introduce the concept of tree boundary to summarize the useful information available from previous mining. We then show that the tree boundary provides an effective and efficient framework for the new mining. The proposed technique has been implemented in the contexts of two existing frequent itemset mining algorithms, FP-tree and Tree Projection. Experiment results on both synthetic and real-life datasets show that the proposed approach achieves dramatic saving in computation.

1. Introduction

Frequent itemset mining plays an essential role in mining association rules [3], correlations, sequential patterns, maximal patterns [4], etc. Although many efficient algorithms [3, 16, 1, 9] have been developed, mining of frequent itemsets remains to be a time consuming process [10], especially when the data size is large. To make the matter worse, in most practical applications, the user often needs to run the mining algorithm many times before satisfied with the final results. In each process, the user typically changes some parameters or constraints.

Considering a mining task with only the *minimum support* constraint (also called the *frequency* constraint), the user may initially set the minimum support to 5% and run a mining algorithm. After inspecting the returned results, s/he finds that 5% is too high. S/he then decides to reduce the minimum support to 3% and runs the algorithm again. Usually, this process is repeated many times before s/he is satisfied with the final mining results.

This interactive and iterative mining process is very time consuming. Mining the dataset from scratch in each iteration is clearly inefficient because a large portion of the computation from previous mining is repeated in the new mining process. This results in enormous waste in computation and time. So far, limited work has been done to address this problem, and to the best of our knowledge there is still no effective and efficient solution.

In recent years, many constraints (apart from the traditional support and confidence constraints) are introduced into frequent itemset mining in order to find only those relevant itemsets [13, 10, 12]. On one hand, these additional constraints give the user more freedom to express his/her preferences. On the other hand, however, it often prolongs the mining process because the user may want to see the results of various combinations of constraint changes by running the mining algorithm more times. This makes mining using previous results for efficiency even more important.

Constraint changes can mean *tightening constraints* and *relaxing constraints*. Let us use an example to start the discussion.

Example: Consider that one sets the constraint that the average price of the items in an itemset is less than $100 in the old mining process (for a market basket problem). After inspecting the mining results, one finds that the results are not satisfactory. There are possible two reasons: (1) $100 is too low (many useful itemsets may not be discovered), and (2) $100 is too high (too many itemsets are generated). One may wish to change the average price to $150 or to $80 for the new mining. The question is "can we make use of the results from the old mining to speed up the new mining?"

It is straightforward to answer part of the question, i.e., when constraints are *tightened* (the solution space is reduced), e.g., when the average price of frequent itemsets is decreased. To obtain the new set of frequent itemsets under the new constraints, we can simply check the frequent itemsets from the old mining to filter out those itemsets that do not satisfy the new constraints. This filtering process is sufficient because the set of new frequent itemsets is only a subset of the old set.

When constraints are *relaxed* (the solution space is

expanded), the problem becomes non-trivial as re-running the mining algorithm is needed to find those additional frequent itemsets. For instance, in the above example, when the average price of frequent itemsets is increased, more itemsets may be generated. The problem becomes even more complicated when multiple constraints are changed at the same time. The objective of this work is to study how to make use of the previous mining results to speed up re-mining when constraints are changed.

In this paper, we propose a novel technique to solve this problem. Using the relaxation of frequency constraint (the decrease of *minimum support*) as an example, we first propose the concept of *tree boundary* to summarize and to reorganize the previous mining results. We then show that the additional frequent itemsets can be generated in the new mining process by extending only the itemsets on the *tree boundary* without re-generating the frequent itemsets produced in the previous mining (note that our tree boundary based technique is quite different from the incremental mining approaches based on negative border). The proposed technique has been implemented in the contexts of two frequent itemset mining algorithms, FP-tree [9] and Tree Projection [1]. This results in two augmented itemset mining algorithms RM-FP (re-mining using FP-tree) and RM-TP (re-mining using Tree Projection). Extensive experiments on both synthetic data and real-life data show that RM-FP and RM-TP dramatically outperform FP-tree and Tree Projection algorithm respectively. Finally, we also address how the proposed technique can be applied to handle the changes of other types of constraints given in previous studies [13, 10, 12].

2. Related work

Frequent itemset mining has been studied extensively in the past e.g. in [3, 16, 1, 9, 15, 4, 5]. Most current algorithms are variations of the Apriori algorithm [3]. They use support-based generate-and-test approach to find all the frequent itemsets. Recently, some tree-based algorithms were also proposed, e.g., the FP-tree algorithm [9], which is based on the frequent pattern tree, and Tree Projection algorithm [1], which is based on the lexicographic tree. Both algorithms do not strictly follow the Apriori-like candidate generate-and-test approach and were shown to be more efficient than the Apriori algorithm [3].

Since [13] first introduced item constraints to produce only those useful itemsets, many other types of constraints have been integrated into itemset mining algorithms [10, 12]. Although many efficient algorithms for mining frequent itemsets with constraints exist, user interaction is at the minimum level. To remedy this situation, [10] proposes to establish breakpoints in the mining process to accept user feedback to guide the mining. Furthermore, online association rule mining also allows the user to increase minimum support during the mining process [2]. However, [2] does not allow decreasing of minimum support. Similarly, the support threshold used in [11] for incremental and interactive sequence pattern mining can also be increased but not decreased.

The closely related work to ours is the incremental mining, where the concept of *negative border* (proposed in [16]) is utilized to update the mining results when additional data becomes available [14, 15, 8, 11]. A negative border consists of all the itemsets that are candidates of the Apriori algorithm that do not have sufficient support. Although the methods in [14, 15, 8] only need one scan of the updated dataset, they could not avoid the disadvantage of negative border, i.e., maintaining a negative border is very memory consuming and is not well adapted for very large databases [11].

The approach in [14, 15, 8] seemingly can be adapted for handling constraint relaxation. [15] actually mentions the possibility but no detailed algorithm is proposed. However, one significant shortcoming of the approach is that generating candidates under new constraints using the negative border under old constraints usually result in over-generation of a huge number of useless candidates. This makes the approach in [14, 15, 8] impractical for our constraint relaxation problem for large datasets, especially when the minimum support is low. For example, if 10^5 frequent itemsets are obtained given minimum support of 1% and 50 1-itemsets become frequent after minimum support is reduced to 0.9%, the number of candidate itemsets generated using the above approach is $(2^{50}-1)*10^5 \approx 10^{20}$ even if we do not consider the expansions of 10^5 frequent itemsets themselves. This is clearly impractical.

FUP in [6] is another incremental mining method that follows the Apriori framework. FUP is not for mining with constraint changes. If it is applied to our task, it basically re-runs the Apriori algorithm without re-counting the supports of those itemsets generated previously (they still need to be re-generated). The computation saving is thus very limited, if any, because of some overheads (see [7] for more details).

3. Problem statement

Let I be the set of all items, and Γ be a transaction database. Each transaction in Γ consists of a subset of items in I. Let S ($\subseteq I$) be an itemset. The ***support*** of S (denoted by $Support(S)$) is defined as in [3]. Given a *minimum support MinSup*, an itemset S is ***frequent*** in Γ if $Support(S) \geq MinSup$. With a transaction set Γ and a *MinSup*, the problem of ***frequent itemset mining*** is to find the complete set of frequent itemsets in Γ.

Constraints can be imposed on both itemset S itself and its attributes (e.g., *price*, *type*, etc) in frequent itemset mining. There are many types of constraints that can be imposed on frequent itemset mining. Four categories of

constraints: *anti-monotone*, *monotone*, *succinct*, and *convertible* constraints have been effectively integrated into some mining algorithms [10, 12].

***Iterative mining of frequent itemsets with constraint changes*:** Given a transaction database T, the whole process of *iterative* (*and interactive*) *mining of frequent itemsets with constraint changes* is captured with the following iterative steps:

(1) specify the initial set of constraints *SC*.

(2) run the mining algorithm

(3) check the returned results to determine whether they are satisfactory. If so, the mining process ends. Otherwise, the user changes one or more constraints in *SC* (including deletion and addition of constraints), and the process then goes to (2).

(1) and (3) will not be discussed further in the paper as it is the user's responsibility to devise and to change constraints. Our objective is to design a framework for the mining algorithm in (2) so that it is able to leverage on the mining results from the previous mining iteration to improve the efficiency of the current mining, and consequently speed up the whole data mining process.

Constraint changes: Change of a constraint includes two cases:

(1) Tighten the constraint: The solution space is reduced. For example, when the *minimum support* is increased.

(2) Relax the constraint: The solution space is expanded. For example, the *minimum support* is decreased.

Constraint changes mean changes to one or several constraints in a set of pre-defined constraints. The changes cover deletion or addition of constraints. Adding a new constraint corresponds to tightening the constraint, while deleting an existing constraint corresponds to relaxing the constraint.

As discussed earlier, if a constraint C is tightened to C', the set of itemsets that satisfy the new constraint C' is only a subset of the itemsets that satisfy the old constraint C. Thus, the set of itemsets that satisfy C' can be obtained by filtering the set of itemsets that satisfy C. The challenge comes when a constraint C is relaxed to C'. The set of itemsets that satisfy the old constraint C is only a subset of the itemsets that satisfy the new constraint C'. The problem is how to efficiently discover the set of itemsets F_n that satisfy the new constraint C' but not the old constraint C. The rest of the paper focuses on this problem. We also study how to utilize the previous mining results to efficiently discover the set of itemsets when multiple constraints are changed at the same time.

4. The proposed technique

We use the minimum support constraint as an example to present the proposed technique for finding the set of itemsets F_n that satisfy the new but not the old *minimum support* when the *minimum support* is reduced (relaxed) from one mining process to the next. The relaxation problems of the other constraints can be solved within the proposed framework (to be discussed in Section 7), although the technical details may vary.

Let $MinSup_{old}$ be the minimum support used in the previous (or old) mining, and $MinSup_{new}$ be the relaxed (or new) minimum support. This section first introduces the useful information that can be obtained from the previous mining process using a *tree-based* itemset mining framework. The reason that we use a tree-based framework will become clear later. We then describe a method to represent the old information for the purpose of mining under $MinSup_{new}$. Next, we present a naïve approach and the proposed technique for discovering the set of itemsets F_n that are frequent under $MinSup_{new}$ but not $MinSup_{old}$.

4.1. Useful information from previous mining

After running a mining algorithm using $MinSup_{old}$, we find the set of frequent itemsets. One byproduct of the process is the set of itemsets that are checked against $MinSup_{old}$ (supports are counted) but are not frequent. Let L_f be the set of frequent itemsets under $MinSup_{old}$, and L_{if} be the set of itemsets that are counted, but found infrequent (the byproduct). Although all frequent itemset mining algorithms generate the same set L_f, the set of infrequent itemsets L_{if} checked in the process varies according to algorithms.

Algorithms, such as those in [4, 1, 9], do not strictly follow the candidate generation of Apriori-like algorithms [3, 16, 10]. Instead, they are based on some kinds of tree. We classify these algorithms as tree-based algorithms. Tree-based algorithms will count the support of an itemset $S = \{i_1, i_2, \ldots, i_k\}$ if two proper subsets of S, namely $S_1 = \{i_1, \ldots, i_{k-2}, i_{k-1}\}$ and $S_2 = \{i_1, \ldots, i_{k-2}, i_k\}$, are frequent.

We use tree-based mining algorithms as the underlying mining framework of our proposed technique because tree-based mining algorithms give us sufficient information, while Apriori-like algorithms do not (see the end of the Section). [1,9] also show that tree-based algorithms are actually more efficient in many cases.

As in [1], we use a lexicographic tree to represent the set of frequent itemsets L_f. Given the set of items I, it is assumed that a lexicographic order R exists among the items in I. The order R is important for efficiency and for the organization of mining results. We use the notation $i \leq_L j$ to denote that item i occurs lexicographically earlier than j.

Definition 4.1 (Lexicographic Tree) A node in a lexicographic tree corresponds to a frequent itemset. The root of the tree corresponds to the *null* itemset.

We extend Definition 4.1 to also represent those itemsets in L_{if} with a lexicographic tree. An example lexicographic tree is shown in Figure 1. Those nodes enclosed in circles are frequent itemsets under $MinSup_{new}$

but not $MinSup_{old}$, which are in F_n. Those nodes enclosed by dotted squares are the itemsets in L_{if} that are not frequent under either $MinSup_{old}$ or $MinSup_{new}$. The other nodes are itemsets that are frequent under both $MinSup_{old}$ and $MinSup_{new}$. Let P and Q be two itemsets and Q be the parent of P.

Definition 4.2 (Tree Extensions) A frequent 1-extension of an itemset such that the last item is the contributor to the extension is called a *tree extension*. The list of *tree extension*s of a node P is denoted by $E(P)$.

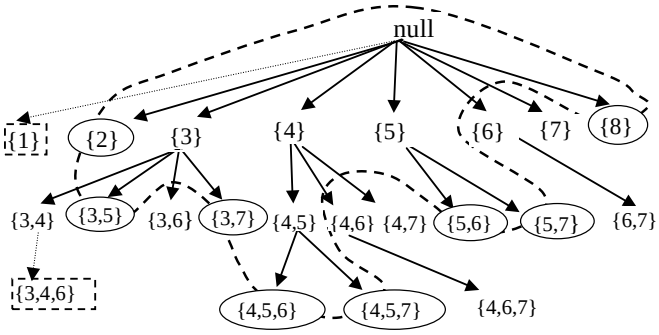


Figure 1. A lexicographic tree

In Figure 1, under $MinSup_{old}$, the list of tree extensions of node 3 $E(3) = <4, 6>$.

Definition 4.3 (Candidate Extensions) The list of candidate extensions of a node P is defined to be those items in $E(Q)$ that occur lexicographically after the node P. We denote the list by $C(P)$. Note that $E(P)$ is a subset of $C(P)$.

Items in $C(P)$ are possible frequent extensions of P. Under $MinSup_{old}$, the tree extensions of *null* node $E(null) = <3, 4, 5, 6, 7>$ (note that 2 is not frequent under $MinSup_{old}$), and the candidate extensions of node 3 $C(3) = <4, 5, 6, 7>$.

4.2. Extensions of lexicographic tree

This subsection extends the lexicographic tree with some new conceptions, which will be used in our proposed technique.

Definition 4.4 (Infrequent Borders) If a 1-extension i of itemset P is not frequent, i is called an *infrequent border*. The list of *infrequent borders* of a node P is denoted by $IB(P)$. We have the relationship: $IB(P) = C(P) - E(P)$.

In Figure 1, under $MinSup_{old}$, the infrequent borders of node 3 $IB(3) = <5, 7>$.

Definition 4.5 (New Tree Extensions) If itemset $P \cup \{i\}$, $i \in IB(P)$, becomes frequent after *MinSup* is reduced from $MinSup_{old}$ to $MinSup_{new}$, i is called a new tree extension of node P *w.r.t.* $MinSup_{new}$. The list of new tree extensions of node P *w.r.t.* $MinSup_{new}$ is denoted by $NTE(P)$.

In Figure 1, the list of new tree extensions of node 3 *w.r.t.* $MinSup_{new}$ $NTE(3) = <5, 7>$.

For any frequent itemset P (can be *null*) under $MinSup_{old}$, its *tree extensions* $E(P)$ and *infrequent borders* $IB(P)$ are stored for mining under $MinSup_{new}$. Its *new tree extensions* $NTE(P)$ *w.r.t.* $MinSup_{new}$ can be obtained by checking the list of infrequent borders of P, $IB(P)$. Under $MinSup_{old}$, the set of *tree extensions* of all frequent tree nodes makes up L_f, and the set of *infrequent borders* of all frequent nodes in the tree makes up L_{if}.

4.3. A naïve approach

With the two sets L_f and L_{if} from the mining under $MinSup_{old}$, we first look at a naïve approach to making use of previous mining results for the new mining. We then present the proposed approach based on *tree boundary*.

The naïve approach checks all itemsets in L_f and L_{if} one by one to find the change of their candidate extensions under $MinSup_{new}$, and to extend them to obtain the complete set F_n (in which itemsets are frequent under $MinSup_{new}$ but not $MinSup_{old}$). Figure 2(a) shows the children itemsets of *null* node and the children itemsets of itemset {3} in the naïve approach. To make the figure manageable, we assume that itemset {3, 8} is frequent under $MinSup_{new}$ but {4, 8}, {5, 8}, {6, 8}, and {7, 8} are not. Candidate extensions of each node are shown under the node in Figure 2(a). The only saving in the new mining is that we can utilize the count information saved previously for those itemsets in L_f and L_{if}.

However, this saving in computation is very limited in a tree-based algorithm. Thus, the computation is basically the same as re-mining from scratch. In tree-based algorithms, the main computation comes from the generation of projected transactions for each node. Project transactions for an itemset S are the set of transactions containing S. Tree-based algorithms use this sub-transaction set for counting support and for all subsequent itemset (containing S) generations. This naïve approach still requires the same computation to generate the projected transactions as running a tree-based algorithm from scratch. For instance in Figure 2(a), we still need to create projected transactions for {3} to count the support for itemset {3, 8} although the supports of its other children itemsets {3, 4}, {3, 5}, {3, 6} and {3, 7} are known previously (the projected transactions for {3} are also used to generate the projected transactions for children itemsets of {3}). Similar computation is required for creating projected transactions for {2}, {3}, {4}, {5}, {6} and {7}.

Another shortcoming of the naive approach is that it cannot avoid re-generating itemsets in L_f because they need to be extended in the new mining. For example, in Figure 2(a), itemsets {3}, {4}, {5}, {6} and {7} still need to be generated to check whether item 8 is in their tree extensions although their supports are already counted in previous mining.

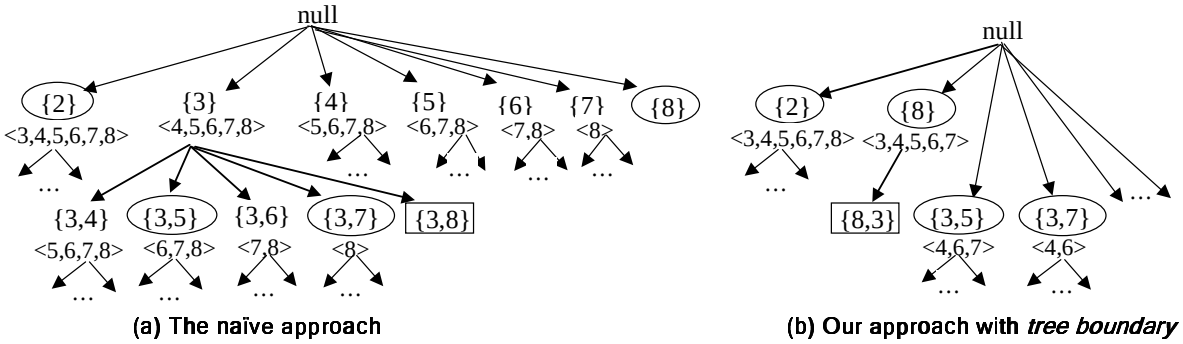


Figure 2. Part of mining results under $MinSup_{new}$

Based on the above discussion, we see that saving by the naive approach is limited. It is thus not efficient.

4.4. The proposed approach

Definition 4.6 (Tree Boundary) A tree boundary *w.r.t.* $MinSup_{new}$ is defined to be the set of itemsets $TB = \{tb \mid tb \in L_{if}, Support(tb) \geq MinSup_{new}\}$, where L_{if} is the set of counted but infrequent itemsets under $MinSup_{old}$, and $Support(tb)$ is the *support* of itemset *tb*.

For example, the itemsets on the dotted line shown in Figure 1 make up the *tree boundary w.r.t.* $MinSup_{new}$. Itemsets {1} and {3, 4, 6} are not in *TB* although they are in L_{if} because they are not frequent under $MinSup_{new}$.

Our proposed approach discovers the complete set of F_n by extending only the itemsets on the *tree boundary*. The basic idea is to eliminate the effect of *MinSup* decrease on itemsets in L_f, i.e., no itemset will be extended if it has been extended in previous mining. This is achieved by changing the order of *tree extensions* of every node (including the *null* node) in L_f (under $MinSup_{old}$).

Let S_p be *null* node or any itemset in L_f. Tree extensions of S_p under $MinSup_{new}$, denoted by $E_{new}(S_p)$, contains two parts:

- tree extensions of S_p under $MinSup_{old}$, $E_{old}(S_p)$, e.g., $E_{old}(3) = <4, 6>$, and
- new tree extensions of S_p (*w.r.t.* $MinSup_{new}$), $NTE(S_p)$, e.g., $NTE(3) = <5, 7>$.

We change the item order of $E_{new}(S_p)$ as follows: move items from the new tree extensions, $NTE(S_p)$, to the front of the (old) tree extensions of S_p under $MinSup_{old}$, $E_{old}(S_p)$. For example, in Figure 1, we change the tree extensions of *null* under $MinSup_{new}$ from <**2**, 3, 4, 5, 6, 7, **8**> to <**2**, **8**, 3, 4, 5, 6, 7>.

With the new ordering, for a child itemset of S_p such that $S_c = S_p \cup \{i\}$, where $i \in E_{old}(S_p)$ ($S_c \in L_f$), the *candidate extensions* of S_c are the same under $MinSup_{old}$ and $MinSup_{new}$. For a child itemset of S_p such that $S_n = S_p \cup \{i\}$, where $i \in NTE(S_p)$, the *candidate extensions* of S_n consists of :

(1) those items j such that $i \leq_L j$, where $j \in NTE(S_p)$, and (2) those items j, $j \in E_{old}(S_p)$.

Due to the re-ordering, candidate extensions of the itemsets in L_f are not affected. For instance, after we change the tree extensions of *null* node under $MinSup_{new}$ into <2, 8, 3, 4, 5, 6, 7>, the tree extensions of itemsets {3}, {4}, {5}, {6} and {7} under $MinSup_{new}$ are the same with those under $MinSup_{old}$. The tree extensions of itemset {8} become <3, 4, 5, 6, 7> from ∅ under $MinSup_{old}$. We compute the projected transactions for itemset {8} to decide whether items 3, 4, 5, 6, and 7 are tree extensions of {8}. There is no need to compute projected transactions for {3}, {4}, {5}, {6} and {7} (they were computed in previous mining).

Another example is given in Figure 2(b), which shows the corresponding part of Figure 2(a) in our approach. After we change the order of tree extensions of *null* node, there is no need to extend itemsets {3}, {4}, {5}, {6} and {7} with 8. We change tree extensions of itemset {3} from <4, 5, 6, 7> to <5, 7, 4, 6>. The candidate extensions of node {3, 5} are <4, 6, 7>. The candidate extensions of node {3, 7} are <4, 6>. As a result, we only need to compute projected transactions for itemsets {3, 5} and {3, 7} (which are not computed in previous mining) while the naïve approach needs to compute projected transactions for itemsets {3, 4}, {3, 5}, {3, 6} and {3, 7}.

Notice that those itemsets on the *tree boundary* whose candidate extensions are empty can be removed from the *tree boundary*, e.g., itemsets {4, 5, 7} and {5, 7} in Figure 1.

Let us summarize the advantages of our *tree boundary* based extension with ordering change.

1) Our approach is able to avoid the computation of counting the supports of itemsets in L_f and L_{if}. We do not re-generate the itemsets in L_f to extend them in the new mining process.

2) Our approach is able to avoid the generation of projected transactions that were done in previous mining while the naïve approach is unable to.

The ordering change is the key of our technique. It also brings some additional benefits when integrating tree-based algorithms with *tree boundary*. Refer to [7].

Now, let us prove the correctness and completeness of *tree boundary* approach.

Property 4.1 Given *tree boundary TB w.r.t. $MinSup_{new}$*, extending the itemsets in *TB* is able to generate the complete set of itemsets F_n (frequent under $MinSup_{new}$ but not $MinSup_{old}$).

Interested readers can refer to [7] for proof.

Remark: In Apriori-like algorithms, previous mining results under $MinSup_{old}$ do not provide sufficient information to build the *tree boundary* for re-mining under $MinSup_{new}$. Moreover, even if we could build a *tree boundary*, Apriori-like algorithms could not be easily modified to extend itemsets on *tree boundary* to discover F_n.

Interested readers can refer to [7] for proof.

5. Tree boundary based re-mining

We realized the proposed technique using the FP-tree frequent itemset mining and the Tree Projection algorithms. The algorithm using FP-tree is called Re-Mining using FP-tree (in short RM-FP), and the algorithm using Tree Projection is called RM-TP (Re-Mining using Tree Projection). Interested readers can refer to [7] for the algorithms RM-FP and RM-TP.

6. Experimental evaluation

This section presents performance comparison of FP-tree algorithm with RM-FP on both synthetic and real-life data sets. The comparison of Tree Projection algorithm with RM-TP achieves similar results, and is given in [7]. All experiments are performed on a 750-Mhz Pentium PC with 512 MB main memory, running on Microsoft Windows 2000. All the programs are written in Microsoft Visual C++ 6.0.

The synthetic datasets were generated using the procedure described in [3]. We report experiments results on two synthetic datasets: One is T25.I20.D200k [9] with 1K items, which is denoted as D1. In D1, the average transaction size and the average maximal potentially frequent itemset size are 25 and 20 respectively. The number of transactions is 200k. The other dataset is T20.I6.D100k [3] also with 1K items, denoted as D2.

We also tested our approaches on two real-life datasets obtained from the UC-Irvine Machine Learning Database Repository(http://www.ics.uci.edu/~mlearn/MLRepository.html). One is the *Connect*-4 dataset the other is the *Mushroom* dataset.

Figures 3 and 5 show the comparisons of RM-FP with FP-tree algorithm on datasets D1 and *Connect*-4. In the curves for RM-FP, the CPU time for each point (except the first point) is obtained by running RM-FP (with the value of that point as $MinSup_{new}$) based on the previous mining results under $MinSup_{old}$ just before that point. For example in Figure 3, the CPU time of RM-FP at $MinSup_{new}$ = 1.75% is based on the old mining results with $MinSup_{old}$ = 2%, and the CPU time for RM-FP at $MinSup_{new}$ = 1.5% is based on the old mining results with $MinSup_{old}$ = 1.75%, and so on. Note that when $MinSup_{new}$ of RM-FP is the same as $MinSup_{old}$ of the previous mining, e.g., at *MinSup* = 2% in Figure 3, the extra running time of RM-FP against FP-tree shows the overhead of RM-FP to output itemsets in L_{if}. The time is very small as shown in Figures 3-9. The results on D2 and *Mushroom* are not shown due to space limitations. Actually, readers can see them based on Figures 7 and 9.

From Figures 3 and 5, we observe that RM-FP is able to save more than 40% running time of FP-tree in each iteration. The saving is very significant in practice. In fact, RM-FP can achieve even better results if the decrease of *MinSup* is smaller in each iteration as shown in Figure 4. In Figure 4, the *MinSup* is reduced by 10% each time (the decrease is smaller than that in Figures 3 and 5). At each point, again RM-FP is run based on the mining results of the previous point except for 2%. In each iteration, we can save more than 70% of the running time.

More performance curves on datasets D1, D2, *Mushroom* and *Connect*-4 are given in Figure 6, 7, 8 and 9 respectively. In Figure 6, RM-FP was run based on the initial mining results of the FP-tree algorithm with $MinSup_{old}$ = 2%, 1.5% and 0.75%. In each case, a few decreased $MinSup_{new}$ values are used. In Figure 7, RM-FP was run based on the mining results of $MinSup_{old}$ = 2%, 1% and 0.5%. In Figure 8, RM-FP was run based on the mining results of $MinSup_{old}$ = 60%, 50%, and 45% (we use very high minimum support because the dataset is very dense). In Figure 9, RM-FP was run based on the mining results at $MinSup_{old}$ = 2%, 1%, and 0.5%. In each of these figures, we show results with different $MinSup_{new}$ values.

All the experiments show that RM-FP consistently outperforms the FP-tree algorithm even when *MinSup* drops to a very low level from a very high level. Using the same initial (old) mining results, we observe that the lower the $MinSup_{new}$ is in the new mining, the smaller is the percentage of saving in computation. This is clear because the number of frequent itemsets at $MinSup_{new}$ is much larger than the number of itemsets in L_f from old mining. For example, for D2, the discovered frequent itemsets at 2% is 381 while the number at 0.15% is 558,834. However, in practice, the user typically will not reduce the *MinSup* so drastically from one mining process to the next. For example, in most cases, it is quite unlikely that the user uses $MinSup_{old}$ = 2% first, and then changes it to $MinSup_{new}$ = 0.15% suddenly for the next mining. Instead, the decrease each time is usually small as in the cases of Figures 3, 4, and 5.

Note that in Figure 9, RM-FP based on 1% support takes more time than RM-FP based on 2% support at $MinSup_{new}$ = 0.75%. This is because the time used to check previous mining results offsets part of the benefit from utilizing previous mining results when the previous mining results are very large.

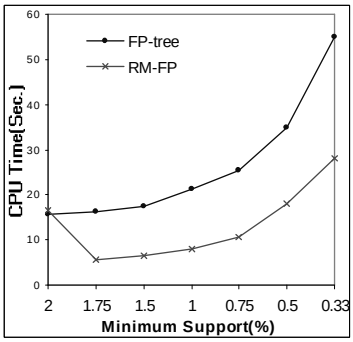


Figure 3. Interactive mining on D1

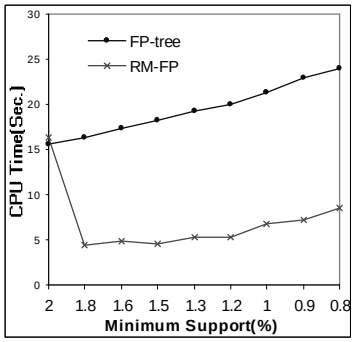


Figure 4. Interactive mining on D1(smaller decrease)

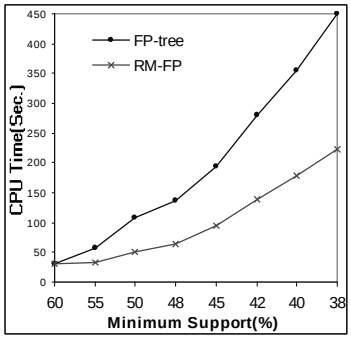


Figure 5. Interactive mining on *Connect*-4

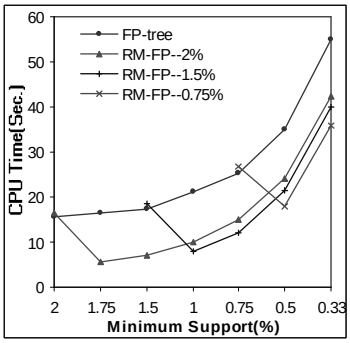


Figure 6. RM-FP performance on D1

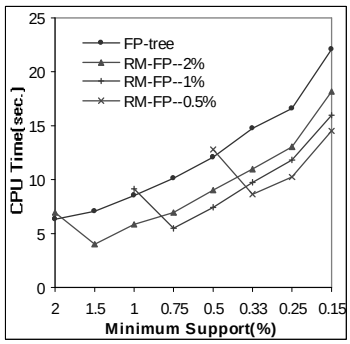


Figure 7. RM-FP performance on D2

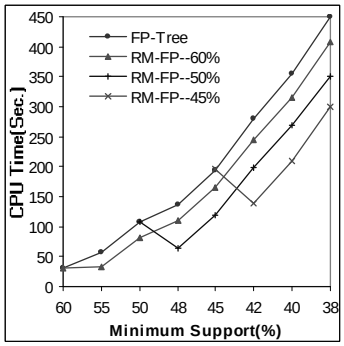


Figure 8. RM-FP performance on *Connect*-4

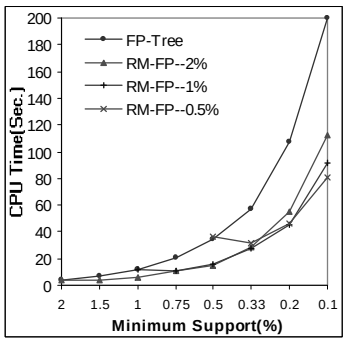
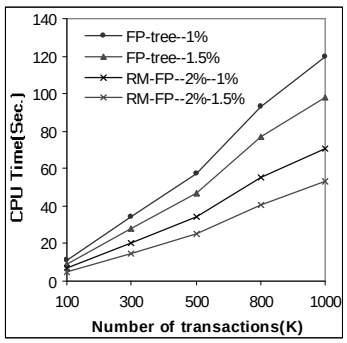


Figure 9. RM-FP performance on *Mushroom*



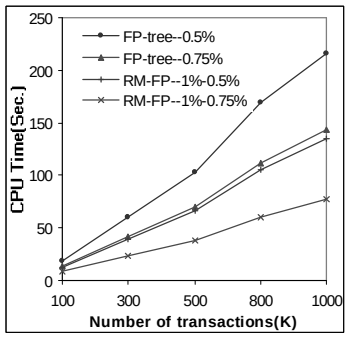


Figure 10. Scalability with the number of transactions

The scalability experiments are conducted by increasing the number of transactions on dataset D1. As shown in Figure 10, both FP-tree and RM-FP have linear scalability with the number of transactions, but RM-FP is more scalable.

7. Application to other constraints

This section shows that the proposed approach is also applicable to discovering the set F_n when any other single or multiple constraints are changed. The detailed techniques for handing changes of these constraints differ. We only present methods for dealing with the change of individual constraints and multiple constraints intuitively. Interested readers may refer to our technical report [7] for additional details and examples.

7.1. Dealing with Individual Constraint Changes

We discuss the methods for discovering the set F_n when a single constraint is changed.

Method 1: Filtering previous mining results

The set F_n can be obtained by filtering previous results in the following two cases: (1) tightening of a constraint of any kind; (2) relaxation of a convertible monotone or monotone constraint.

Method 2: Tree boundary based re-mining

This method as discussed in Section 4 applies to the relaxation of a convertible anti-monotone or anti-monotone constraint although it is a bit different when applying to anti-monotone constraint relaxation due to the special property of convertible constraints [7].

Method 3: Simpler tree boundary based re-mining

Tree boundary in this method is easier to devise than

that for Method 2 and usually contains only 1-itemsets. It applies to the relaxation of a succinct and anti-monotone constraint, or a succinct and monotone constraint. When one of such constraints is relaxed, it can be dealt with as follows: Let $E(null)$ be the list of frequent items that satisfy the old constraint. By checking the old mining results, we first find the list of frequent items $NTE(null)$ that satisfy the new constraint but not the old constraint. Itemsets made of individual items in $NTE(null)$ make up the *tree boundary*.

Constraint 1	Constraint 2	Tighten 1&2	Relax 1 tighten 2	Tighten 1 relax 2	Relax 1&2
Succinct & Anti-mono.	Succ. & Anti.	M1	M1&M3	M1&M3	M3
	Succ. & Mono.	M1	M1&adapted M3	M1&M3	adapted M3
	Anti.	M1	M1&M3	M1&M2	M2&M3
	Mono.	M1	M1&M3	M1	M1&M3
	Convert. Anti.	M1	depends	M1&M2	depends
	Convert. Mono.	M1	depends	M1	depends
Succ.&Mono.	Succ. & Mono.	M1	M1&M3	M1&M3	M3
	Anti.	M1	M1&M3	M1&M2	M2&M3
	Mono.	M1	M1&M3	M1	M1&M3
	Convert. Anti.	– \ M1	– \ M1&M3	– \ M1&M2	– \ M2&M3
	Convert. Mono.	– \ M1	– \ M1&M3	– \ M1	– \ M1&M3
Anti-mono.	Anti.	M1	M1&M2	M1&M2	adapted M2
	Mono.	M1	M1&M2	M1	M1&M2
	Convert. Anti.	M1	violates	M1&M2	violates
	Convert. Mono.	M1	violates	M1	violates
Monotone	Mono.	M1	M1	M1	M1
	Convert. Anti.	M1	M1	M1&M2	M1&M2
	Convert. Mono.	M1	M1	M1	M1
Convertible Anti-mono.	Convert. Anti.	– \ M1	– \ M1&M2	– \ M1&M2	– \ M2
	Convert. Mono.	– \ M1	– \ M1&M2	– \ M1	– \ M1&M2
Convert. Mono.	Convert. Mono.	– \ M1	– \ M1	– \ M1	– \ M1

Table 1. Handling the change of two combined constraints

7.2. Dealing with multiple constraint changes

Although users usually change one constraint at a time to see the effect of the change, it is also possible that multiple constraints are changed at the same time. Table 1 shows the methods for discovering F_n when two constraints are changed at the same time. Most of the combined cases can be handled by combining the approaches to handling the change of individual constraints. For example, tightening a succinct & anti-monotone constraint and relaxing a succinct & monotone constraint requires Method 1 (handling the tightening) and 3 (handling the relaxation). Interested readers can refer to [7] for the meanings of those exceptional cases including "Adapted", "Violates", "Depends" and "–".

Finally, when more than two constraints are changed at the same time, they can be handled by combining the methods for their respective changes in consideration of the exceptional cases in table 1.

8. Conclusions

Practical data mining is often a highly interactive and iterative process. Users change constraints and run the mining algorithm many times before satisfied with the final results. Current mining algorithms are unable to take advantage of the previous mining results to speed up the new mining process. Motivated by this problem and using the minimum support constraint as an example, this paper first proposed the concept of *tree boundary* to summarize and reorganize the previous mining results. It then presents an effective and efficient framework for re-mining under the reduced minimum support. Experiment results demonstrate that the proposed technique is highly effective. Finally, we also show that when any other individual constraint is changed or multiple constraints are changed at the same time, the new set of frequent itemsets can also be mined efficiently using the proposed technique.

References

[1] R. Agarwal, C. Aggarwal, and V. Prasad. A Tree Projection algorithm for generation of frequent itemsets. In J. Parallel and Distributed Computing, 2000.

[2] C. Aggarwal and P. Yu. Online generation of association rules. In Proc. of 14th ICDE, 1998.

[3] R. Agrawal and R. Srikant. Fast algorithm for mining association rules. In Proc. of the 20th VLDB, 1994.

[4] R. J. Bayardo. Efficiently mining long patterns from database. In Proc. of the SIGMOD, 1998.

[5] A. Bykowski, C. Rigotti. A condensed representation to find frequent patterns. In Proc. of PODS, 2001.

[6] D. W. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In Proc. of ICDE, 1996

[7] G. Cong, B. Liu, Interactive mining of frequent itemsets with constraint changes. Technical report, National Univ. of Singapore, 2002.

[8] R. Feldman, Y. Aumann, A. Amir, and H. Manila. Efficient algorithm for discovering frequent sets in incremental databases. In 2nd SIGMOD workshop DMKD, 1997.

[9] J. Han, J. Pei, and Y.Yin. Mining Frequent Patterns without Candidate Generation. In SIGMOD, 2000.

[10] R. Ng, L.V.S. Lakshmanan, J.Han, and A.Pang. Exploratory mining and pruning optimizations of constrained association rules. In Proc. of SIGMOD, 1998.

[11] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In Proc. of the 8th CIKM, Kansas City, MO, USA, November 1999.

[12] J. Pei, J. Han, and L.V.S.Lakshmanan. Mining frequent itemsets with convertible constraints. In Proc. ICDE, 2001.

[13] R. Srikant, Q, Vu, and R. Agrawal. Mining association rules with item constraints. In Proc. of KDD, CA, 1997.

[14] S. Thomas, S. Chakravarthy. Incremental mining of constrained associations. In HiPC2000.

[15] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In Proc. KDD, 1997.

[16] H. Toivonen. Sampling large databases for association rules. In Proc. of the 22th VLDB, 1996.