*Nov. 2004, Vol.19, No.6, pp.876–884* *J. Comput. Sci. & Technol.*

# Efficient Incremental Maintenance of Frequent Patterns with FP-Tree

Xiu-Li Ma[1,2], Yun-Hai Tong[1,2], Shi-Wei Tang[1,2], and Dong-Qing Yang[1]

[1] *School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, P.R. China*

[2] *National Laboratory on Machine Perception, Peking University, Beijing 100871, P.R. China*

E-mail: {xlma, yhtong, ydq}@db.pku.edu.cn; tsw@pku.edu.cn

Received March 26, 2003; revised September 22, 2004.

**Abstract** Mining frequent patterns has been studied popularly in data mining area. However, little work has been done on mining patterns when the database has an influx of fresh data constantly. In these dynamic scenarios, efficient maintenance of the discovered patterns is crucial. Most existing methods need to scan the entire database repeatedly, which is an obvious disadvantage. In this paper, an efficient incremental mining algorithm, Incremental-Mining (IM), is proposed for maintenance of the frequent patterns when new incremental data come. Based on the frequent pattern tree (FP-tree) structure, IM gives a way to make the most of the things from the previous mining process, and requires scanning the original data once at most. Furthermore, IM can identify directly the differential set of frequent patterns, which may be more informative to users. Moreover, IM can deal with changing thresholds as well as changing data, thus provide a full maintenance scheme. IM has been implemented and the performance study shows it outperforms three other incremental algorithms: FUP, DB-tree and re-running frequent pattern growth (FP-growth).

**Keywords** data mining, association rule mining, frequent pattern mining, incremental mining

## 1 Introduction

Since the introduction to association mining in [1], there have been many studies on efficient and scalable frequent pattern mining algorithms[2−5], of which two typical methods are: candidate-generation-and-test Apriori[1], and divide-and-conquer frequent pattern growth (FP-growth)[3].

However, most of the works have assumed that the database being mined is static. In fact, there are more and more domains where the datasets tend to be constantly updated with fresh data, such as e-commerce, web-based or data stream domains. Response time is crucial in such an environment because lengthy time delay can disturb the flow of human perception and formation of insight. Simply re-executing mining from scratch on the whole updated database can result in an explosion in the computational and I/O resources required. What is needed is a way to process the data incrementally and update the discovered patterns.

To address this problem, Cheung and Lee first proposed FUP[6], $\text{FUP}_2$[7] and DELI[8]. However, these algorithms are all Apriori-like, mostly need to generate a large number of candidate itemsets at each level, scan the entire database several times.

To tackle this difficulty, two incremental mining algorithms are presented in [9]. One is the DB-tree algorithm, which stores all the database information in a frequent pattern tree (FP-tree) and requires no re-scan of the original database. However, it neglects that the tree would be too large to fit in the main memory. The other is the PotFP-tree algorithm, which stores items either "frequent at present" or "infrequent but potential" in an FP-tree, controlled by a tolerance value $t$. However, the choice of $t$ is very hard. Furthermore, once $t$ is broken, the tree needs to be re-constructed. In fact, it ignores the already discovered knowledge, duplicating the work already done.

Moreover, all the above incremental mining algorithms assume the minimum support threshold would not change, what if the users decide to change the threshold while the new data come?

To tackle all these challenges, we propose an efficient maintenance approach, Incremental-Mining (IM).

First, the maintenance task is based on FP-tree, from which we exploit many useful properties suitable for incremental maintenance. Its most exciting feature is the adaptiveness to incremental data and changing thresholds. In an FP-tree, the new part resulting from new data and new threshold

*Correspondence

Supported by the National Basic Research 973 Program of China under Grant No.G1999032705.

can be clearly cut from the old part. This means we can re-use the old information to the utmost extent, which will be much faster than constructing all things from scratch, especially when encountering big trees resulting from prolific or long patterns in very large databases. This is a really improvement especially when continuous increments come.

Second, we can identify the differential set of patterns directly, such as who are winners (those become frequent), or losers (those turn to infrequent). Otherwise, mining from scratch cannot tell differences and trends directly. In general, the changes to the patterns set may be more informative in dynamic environment.

Third, our approach can integrate incremental mining with re-mining. Re-mining is to find the set of frequent patterns for the same database under different thresholds[10]. Some work[10,11] has been done on this problem. However, they all assume the database remains unchanged. On the other hand, almost all the methods on incremental mining have focused on changing databases with unchanging threshold. In fact, incremental mining and re-mining should be the double sides of one whole maintenance problem, needing an integration algorithm. Our approach can deal with both at the same time. An integrated approach is proposed in [12], however, it is in the framework of candidate generation and test, whose inherent inefficiency has been analyzed above.

Fourth, we examine the general essence of IM with the lattice-theoretic guidance. According to its homogeneity with lattice partitioning, our idea can be generalized when other mining algorithms are used at first.

These features combined together form the contribution of our new algorithm, IM. A performance study has been conducted to compare IM with FUP, DB-tree, and re-running FP-growth. IM is found to outperform the other three methods.

The remaining of this paper is organized as follows. Section 2 gives a more precise description of the problem, and briefly goes through FP-tree and FP-growth on which our algorithm is based. Section 3 exploits the properties of FP-tree and the methods to extend an FP-tree. Section 4 develops the IM algorithm. Section 5 analyzes the essence of incremental mining under lattice guidance. Section 6 presents our experimental study. Section 7 summarizes our study.

## 2 Problem Definitions

### 2.1 Mining of Frequent Patterns[3]

Let $I = \{a_1, a_2, \ldots, a_m\}$ be a set of items, and a transaction database $DB = \langle T_1, T_2, \ldots, T_n \rangle$, where $T_i$ $(i \in [1, \ldots, n])$ is a transaction which contains a set of items in $I$. The support of a pattern $A$, which is a set of items, is the number of transactions containing $A$ in *DB*. The support of $A$ in *DB* is denoted to be $s_{DB}(A)$. $A$ is a frequent pattern if $A$'s support is no less than a predefined *minimum support threshold* (or *threshold* in short), $\xi$.

Given a transaction database *DB* and a threshold $\xi$, the problem of *finding the complete set of frequent patterns* is called the *frequent pattern mining problem.*

Notice that *support* is defined as *absolute* occurrence frequency. Sometimes, users may predefine *minimum support threshold* in percentage form (*percentage threshold* in short) $\sigma$. Then, a pattern $A$ is frequent in *DB* if its support satisfies: $s_{DB}(A) \geqslant \sigma \times |DB|$. In fact, *threshold* and *percentage threshold* have the same nature.

### 2.2 Incremental Mining of Frequent Patterns

Let *DB* be the original database and $\sigma$ the percentage threshold. Let $F^{DB}$ be the set of frequent patterns in *DB*. Assume an increment *db* of new transactions is added to *DB*. Let $U = DB \cup db$ be the whole database. With respect to the same percentage threshold $\sigma$, a pattern $A$ is frequent in the updated database $U$ if the support of $A$ in $U$ satisfies: $s_U(A) \geqslant \sigma \times |U|$.

Incremental mining of frequent patterns is to find $F^U$, the set of frequent patterns in the updated database $U$ with respect to the same percentage threshold $\sigma$.

Note that, for the same $\sigma$, as the database changes, a frequent pattern in *DB* may not be frequent in $U$, defined as a loser; on the other hand, a pattern not frequent in *DB*, may become a frequent pattern in $U$, defined as a winner; if a frequent pattern in *DB* remains frequent in $U$, it is called a retainer.

### 2.3 Re-Mining of Frequent Patterns

After having found some frequent patterns in *DB*, users may be unsatisfied with the results and

want to tune the *percentage thresholds*, such as from $\sigma$ to $\sigma'$. Re-mining is to find the set $F^{DB}$ of frequent patterns in $DB$ under $\sigma'$.

### 2.4 FP-Tree and FP-Growth

Let us briefly go through *FP-tree* structure and *FP-growth* algorithm in [3], on which we will base. The problem of frequent pattern mining with FP-tree can be decomposed into two sub-problems: first, constructing an FP-tree, which needs two scans of the database; then, the FP-growth algorithm is executed to mine recursively.

An FP-tree is a highly compact data structure for storing crucial information about frequent patterns. Only frequent length-1 items have nodes in the tree, and the tree nodes are arranged in such a way that more frequently occurring nodes have better chance of sharing nodes than less frequently occurring ones. For space limit, we refer the readers to [3]. Fig.1(a) illustrates the FP-tree for $DB$ in Table 2 under *threshold* 3.

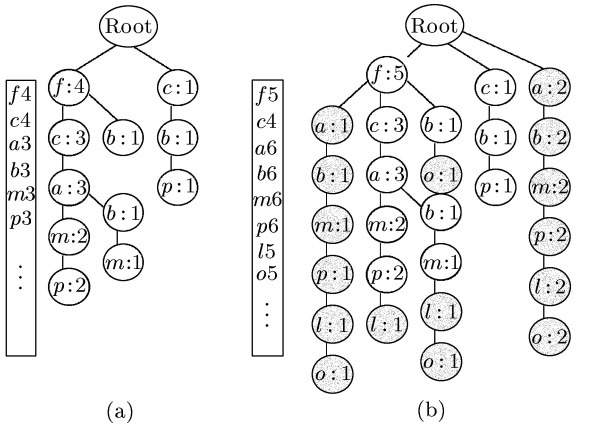



Fig.1. Comparison of FP-tree for $DB$ and for $U$. (a) FP-tree for $DB$ under threshold 3. (b) FP-tree for $U$ under threshold 5 (by extension).

FP-growth is a divide-and-conquer process of recursive mining of each frequent item in a frequency ascending order. For each frequent item $a_i$, FP-growth first derives a frequent pattern $a_i$, with support equal to $a_i$'s count in the header table, then generates a conditional pattern-base $Ba_i$ (the sub-pattern base under the condition of $a_i$'s existence) and constructs a conditional FP-tree FP-tree$|a_i$, on which FP-growth is then recursively performed. The patterns resulting from mining FP-tree$|a_i$ must be concatenated with $a_i$.

## 3 Extending FP-Tree with New Data

When new data come, re-construction of an FP-tree represents a nontrivial overhead. It could be beneficial to extend an existing FP-tree. In this section, we focus on how to extend an existing FP-tree when new data come (and the *threshold* may also change). Table 1 summarizes the notations used in this paper. Let us first consider what will happen to an FP-tree when an increment comes.

**Table 1**. List of Notations

| | |
|---|---|
| $DB$ | The original database |
| $db$ | The increment database |
| $U$ | The whole database ($U = DB \cup db$) |
| $\sigma$ | The percentage threshold |
| $\sigma'$ | The new percentage threshold after $db$ comes |
| $T$ | The FP-tree constructed for $DB$ under $\sigma$ |
| $T'$ | The FP-tree constructed for $U$ under $\sigma'$ |

*Example* 1. Table 2 shows the example transaction database (borrowed from [3]). The transactions with *TID* 100 to 500 compose the original database $DB$, while those from 600 to 800 compose the increment $db$. Let the *percentage threshold* be 60%. Thus the absolute *threshold* is $5 \times 60\% = 3$ in $DB$, and $8 \times 60\% = 5$ in $U$. For ease of explanation, the original frequent-item projections of $DB$ and $U$ are respectively shown in the third and fourth column of Table 2. Figs.1(a) and 1(b) respectively illustrate the FP-tree for $DB$ under *threshold* 3 and $U$ under 5. At the first time mining, the complete set of frequent items are $\{f : 4, c : 4, a : 3, b : 3, m : 3, p : 3\}$; After $db$ comes, the set will be $\{f : 5, a : 6, b : 6, m : 6, p : 6, l : 5, o : 5\}$.

**Table 2**. Example Transaction Database
$DB$ (*TID*: 100–500), $db$ (*TID*: 600–800)

| *TID* | Items bought | (Ordered) frequent items ($DB$, 3) | Frequent items ($U$, 5) |
|---|---|---|---|
| 100 | *f a c d g i m p* | *f c a m p* | *f a m p* |
| 200 | *a b c f l m o* | *f c a b m* | *f a b m l o* |
| 300 | *b f h j o* | *f b* | *f b o* |
| 400 | *b c k s p* | *c b p* | *b p* |
| 500 | *a f c e l p m n* | *f c a m p* | *f a m p l* |
| 600 | *f a b m p l o* | | *f a b m p l o* |
| 700 | *a b m p l o* | | *a b m p l o* |
| 800 | *a b m p l o* | | *a b m p l o* |

**Property 1.** *Any fixed ordering of frequent items in FP-tree should not affect the completeness of the result set of FP-growth.*

*Proof.* According to a fixed order, the complete set of frequent patterns in FP-growth will be divided into $n$ subsets without overlap, where $n$ is the total number of frequent items. In fact, the support-descending order in FP-tree is just for the purpose of high compactness. □

*Example* 2. Consider the incremental mining problem in Example 1. Pay attention to the orderings and the two trees. If we do not remove the loser item $c$, the original FP-tree $T$ is completely contained in the new tree $T'$ without any variance. Although this ordering may not result in a tree as compact as in support descending order, the original FP-tree $T$ can be re-used to the maximal extent.

**Lemma 1.** *Given DB, db, U, $\sigma$, $\sigma'$, T, T' as meant in Table* 1. *When db comes, we can impose an order for the list of the frequent items in T': the original list of frequent items in DB appended with the new winner items in U. T' constructed in such an order contains the complete information of U in relevance to frequent pattern mining.*

*Proof.* This lemma is based directly on Property 1. □

It is now obvious that we can get $T'$ by extending $T$. We do not remove the loser items. Just a flag is needed to identify a loser item. Moreover, it is really easy to deal with changing *threshold*. That will just be involving more or less new winner items in $T'$.

To prepare for extension, we give some adaptations to mining on *DB* as follows.

1) Once created, an FP-tree is stored into a set.

2) As all items' supports have been computed, store them in support descending order in header table. A *tail* value, which points to the last element in the list of frequent items of the header table, is also stored with the FP-tree, such as $p$ in the header table of Fig.1(a), and $o$ in Fig.1(b).

3) A seed is attached to each FP-tree. It is the pattern fragment to be concatenated with the patterns from the tree.

4) In order to extend an FP-tree, its corresponding conditional base, header table, tail, and seed are all needed. When the first-time mining ends, for each FP-tree (some are conditional FP-trees), we integrate the above things into an object and materialize it. For the whole tree, we save null as its base, because materializing the whole database will be unfair for other algorithms.

Based on the above analysis, we have the following algorithm for extending an FP-tree. Two main parts are included: one is appending the new winner items in *DB* onto the FP-tree, called "FP-tree Tuning", which is the same as "FP-tree extension" in [11]. The other is, extending the transactions in *db* onto the FP-tree, which will not cause any problem since adding those new records is equivalent to scanning additional transactions in the FP-tree construction. We call it "FP-tree Construct-Delta". Note that, we need at most one scan of the original database in order to find back new winner items.

**Algorithm 1.** FP-Tree Extension

Input: An integrated object of FP-tree $T$ for $DB$ under $\sigma$, and new incremental data $db$, new threshold $\sigma'$.

Output: The FP-tree $T'$, for $U(= DB \cup db)$ under $\sigma'$.

Method:

(1) $\xi = |DB| \times \sigma$;

(2) Compute the new support threshold $\xi' = |U| \times \sigma'$;

(3) $T'$ = FP-tree Tuning$(T, DB, \xi, \xi')$;

(4) Call FP-tree Construct-Delta$(T', db, \xi')$;

Procedure FP-tree Tuning$(T, DB, \xi, \xi')$

{(1) Sort the items after the old tail in the header table according to support descending order;

(2) Identify the new tail, which is the index of the entry in header table (along the direction "down" from the old tail) with the smallest support no less than $\xi'$;

(3) For each transaction *Trans* in *DB*, select those items in *Trans* satisfying $\xi' \leqslant$ *item.support* $\leqslant \xi$, and append into the corresponding branch;}

Procedure FP-tree Construct-Delta$(T', db, \xi')$

{(1) For each transaction *Trans* in *db*, select those items in *Trans* within the range 0 and the new tail according to the order of header table, and append them into the FP-tree $T'$;}

## 4 Incremental Mining

Obtaining an FP-tree through extension ensures that the previous FP-tree can be re-used. However, this does not guarantee an efficient maintenance if we simply execute FP-growth on this FP-tree, since one still needs to recursively construct conditional bases and FP-trees. In this section, we will study how to re-use everything of the previous mining process.

*Example* 3. Let us examine the incremental process for *DB* in Table 2 when *db* comes. As FP-growth is a partition-based method, we can split the set of items before the tail in $U$ under threshold 5 into two parts: 1) each item after the old tail, (a new winner item), such as $l$ and $o$, has never been considered; 2) each item before the old tail, such as $f$, $c$, $a$, $b$, $m$, $p$, has been considered in *DB*, not in $U$ yet.

It is easier for us to deal with the items of the first type, just construct conditional pattern base and FP-tree and execute FP-growth as in [3].

Let us concentrate on the second type of items. Let $a_i$ be such an item. With *db* coming, the change of $a_i$'s support can be further classified into three cases: I) $s_{DB}\ (a_i) \geqslant 3$, and $s_U(a_i) \leqslant 5$, such as $f$, $a$, $b$, $m$, $p$, thus put $a_i$ into the set of retainers; II) $s_{DB}(a_i) \geqslant 3$, and $s_U(a_i) \geqslant 5$, such as $c$, thus put $a_i$ into the set of losers; III) $s_{DB}(a_i) \leqslant 3$, and $s_U(a_i) \geqslant 5$, thus put $a_i$ into the set of winners. (A continuous process with multiple increments may result in this kind of items.) For each loser, set its flag "IsLoser". When the item turns frequent later, clear the flag.

After identifying the change of all the items before the tail, let us have a look at such an item's conditional base.

With *db* coming, $a_i$'s conditional base changes. According to the ordering in header table, one can get the $a_i$-projection sub-incremental of *db* as: Scan *db* and project the set of frequent items (except $a_i$) of a transaction *Trans* into the $a_i$-projection database as a transaction, where $a_i$ is in *Trans* and no other item in *Trans* ordered after $a_i$, according to the same ordering in the whole header table. For the transaction with TID 600, i.e., "*f-a-b-m-p-l-o*", we should put "*f*" into *a*-conditional base, put "*f-a*" into *b*-conditional base, etc. We can think the $a_i$-projection from *db* as an increment to $a_i$'s conditional base. With this increment coming, the incremental mining problem for $a_i$ turns into the same problem as that of the global FP-tree.

Note that *db* will be distributed level-by-level. So we must organize all the original integrated objects in a hierarchical way, i.e., for an FP-tree, its direct conditional trees are its children. All the FP-trees are organized level by level. We call this a Hyper-Tree, formed during the FP-growth process for *DB* under $\sigma$. Based on the above analysis, we can find the incremental mining problem is a procedure of incremental mining all the existing FP-trees in a hierarchical way: staring from the global FP-tree, find out the new atoms and mine them; classifying the existing items into retainers, losers and winners; distribute the increment into its children's increment. Thus we have the following algorithm for incremental mining.

**Algorithm 2. IM**

Input: Given *DB*, *U*, *db*, $\sigma$, $\sigma'$ as meant in Table 1; the HyperTree for *DB* under $\sigma$; $F^{DB}$, the complete set of frequent patterns in *DB* under $\sigma$.

Output: $F^U$, the complete set of frequent patterns in *U* under $\sigma'$; and Winners, Losers, and Retainers.

Method:

(1) $\xi' = \sigma' \times |U|$;

(2) Call FptreeIncMining(*db*, $\xi'$, *HyperTree*);

(3) $F_U = F_{DB} \cup$ Winners–Losers;

Procedure FptreeIncMining(*increment*, $\xi'$, *current*)

{(1) **if** (*current* != NULL){

(2) IncrementalMining(*increment*, $\xi'$, *current*);

(3) FptreeIncMining(*increment*, $\xi'$, *current* $\rightarrow$ *Child*);

(4) FptreeIncMining(increment, $\xi'$, *current* $\rightarrow$ *Sibling*); }

(5) **else** return; }

Procedure IncrementalMining(*increment*, $\xi'$, *current*)

{(1) Scan the increment once. Accumulate the support of the items into their original count in header table. Identify the new tail in header table;

(2) **if** (the new tail > the old tail)

(3) { FP-tree Tuning (*current*, $\xi'$);

(4) Extending-Growth(*current*, $\xi'$, *oldTail*, *current*, *null*);} // mining new winners as in [11]

(5) IdentifyWLR (*current*, $\xi'$, Winners, Losers, Retainers);

(6) FP-tree Construct-Delta (*current*, *increment*)

(7) **if** (*Child* != NULL) Distribute(*current*, *increment*);

(8) Append *increment* into the corresponding conditional base;

(9) clear the *increment*; }

Procedure IdentifyWLR (*current*, $\xi'$, Winners, Losers, Retainers)

{(1) **for** each item $a_j$ before the old tail of *current*'s header table **do** {

(2) generate pattern $\beta = a_j \cup$ *seed* with *support* = $a_j$.*support*;

(3) **if** ($a_j$.*count* $\geqslant \xi'$)

(4) { **if** ($a_j$.IsLoser) {Append $\beta$ into Winners; clear $a_j$.IsLoser;}

(5) **else** Append $\beta$ into Retainers; }

(6) **else**

(7) **if** (!$a_j$.IsLoser) { Set $a_j$.IsLoser; Append $\beta$ into Losers;} } }

Procedure Distribute(*current*, *increment*)

{ (1) **for** each transaction in *increment* **do** {

(2) Select and Sort the items before the *tail* according to the order in *current*'s header table into *Tran*;

(3) **for** each item $a_j$ in *Tran* **do** {

(4) project the set of items before $a_i$ (except $a_i$) into the $a_i$-projection database as a transaction; } }

(5) **for** each item $a_j$ **do** {append $a_j$-projection database into $a_j$'s increment; } }

*Analysis.* We traverse the whole space in a root-first order.

If it is the whole tree, increment is *db*; otherwise, it is the distributed increment. On each FP-tree, call IncrementalMining to extend the tree, mine the new winners (Extending-Growth) and classify the

original items (IdentifyWLR). The FP-trees generated in this process should be organized into HyperTree as current's children.

In the procedure Distribute, current's increment is distributed (projected) into its children's increment. Note IM assumes that the first time FP-growth has not adopted the SinglePath branch tuning in [3].

As we have distributed the incremental data, for a nested FP-tree, we can get the right new extension from its own new data, not from its parent tree. Thus we separate the calling of FP-tree tuning and the FP-tree Construct-Delta.

## 5 Lattice-Based Analysis of Incremental Mining

In this section, we will analyze the essence of incremental mining under lattice guidance[2,13]. We will include, without proof, some conclusions in [2]. We call them corollaries here.

### 5.1 Lattice-Theoretic Approach

**Corollary 1.** *For set $S$, the ordered set $P(S)$, the power set of $S$, is a complete lattice in which join and meet are given by union and intersection.*

Define a prefix-based equivalence relation $\theta_k$ on the lattice $P(I)$, so that two itemsets are in the same class if they share a common $k$ length prefix.

*Example* 4. Fig.2 shows the power-set lattice $P(I)$ of the set of items $I = \{m, b, a, c, f\}$. It also shows the lattice induced by the equivalent relation $\theta_1$ on $P(I)$, where all itemsets with a common 1 length prefix are collapsed into an equivalent class. The resulting set or lattice of equivalent classes is $\{[m], [b], [a], [c], [f]\}$. In fact, recursive class decomposition can be applied.

From the above, we can find an analogy between lattice decomposition and FP-growth.

### 5.2 Analogy Between FP-Growth and Lattice-Theoretic Approach

**Lemma 2.** *The divide-and-conquer methodology in FP-growth is homogeneous with the decomposition in lattice according to the prefix-based equivalent relation.*

*Proof.* According to FP-growth, for each frequent item $a_i$, $a_i$'s conditional pattern base contains only the items before $a_i$, according to the list of frequent items. The patterns resulting from the recursive mining on FP-tree$|a_i$ must be concatenated with $a_i$. Thus, given the same order among frequent items, the frequent patterns resulting from $a_i$ in FP-growth are just the same set of frequent patterns in $[a_i]$, the prefix-based equivalent class of $a_i$ in lattice $P(I)$. □

Next, let us examine the incremental mining process based on the lattice essence.

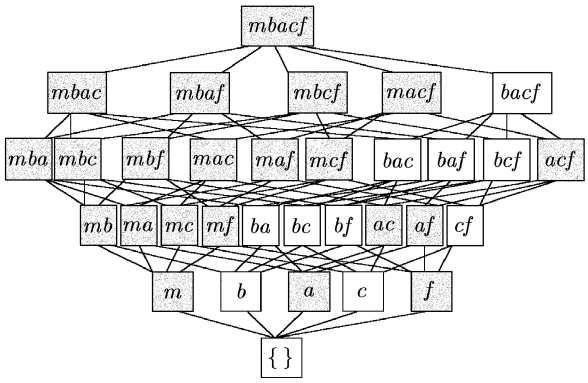



Fig.2. Complete power-set lattice $P(I)$, also, the equivalent classes of $P(I)$ induced by $\theta_1$.

### 5.3 Lattice-Based Analysis of Incremental Mining

**Lemma 3.** *Given DB, db, U as meant in Table 1. When db comes, in the lattice $P(I)$ constructed with the frequent items as atoms, items should be considered this way: for a winner item $a_i$, its prefix-based equivalence class should be constructed and searched. For a loser item $a_j$, all patterns in its prefix-based equivalence class should be losers. For a retained item $a_k$, its sub-lattice should be examined as the whole lattice has been.*

The proof is cut for space limit. Actually, the lattice-theoretic partitioning is the essence of IM. We can implement the idea in IM through other data structures, such as H-tree[4], or FPL[5], which have the same nature as FP-tree.

## 6 Experimental Evaluation and Performance Study

In this section, we present a performance comparison of IM with FUP, DB-tree and re-executing FP-growth when increment comes.

All the experiments are performed on a 1400MHz Pentium PC with 512MB main memory. All the programs are written in Microsoft/Visual C++.Net 6.0. We implement the algorithms of [3, 6, 9] in our environment.

The synthetic data sets that we used in our experiments were generated using the procedure de-

scribed in [1]. In the following, we use the notation T$x$I$y$D$m$d$n$ to denote a database in which $|D| = m$ thousands, $|d| = n$ thousands, $|T| = x$, $|I| = y$. In order to do comparison on a database of size $|D|$ with an increment of size $|d|$, a database of size $(|D| + |d|)$ is first generated and then the first $|D|$ transactions are stored in the database $DB$ and the remaining $|d|$ transactions in the increment $db$.

We report experimental results on four data sets. The first one, denoted as $D1$, is T10I4D99d1. $D2$ is T15I10D9d1. $D3$ is T25I20D99d1. $D4$ is T25I15D9d1. For each setting of synthetic data, such as $D1$, we generate 20 datasets with different random seeds. Then, the mean runtime is computed to be the final score. In this way, the accuracy and reliability of our experiments can be improved. The runtime of IM and the other two algorithms (FP for rerunning FP-growth in short, and FUP) for $D1$ and $D2$ are plotted in Fig.3, while

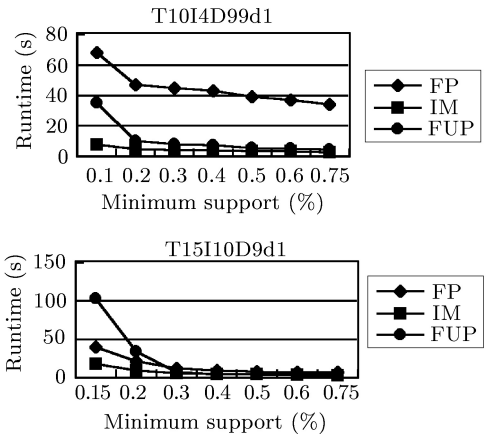



Fig.3. Scalability with threshold for $D1$ and $D2$.

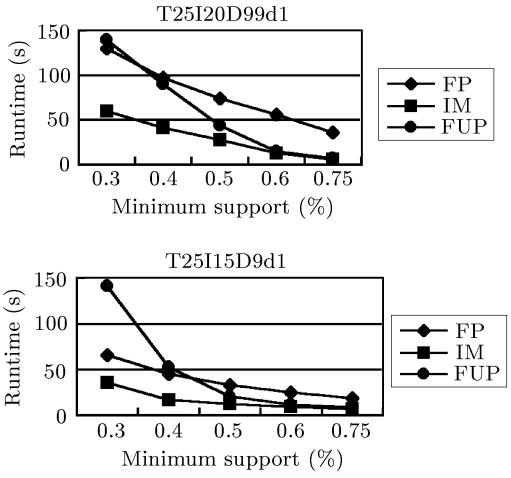



Fig.4. Scalability with threshold for $D3$ and $D4$.

for $D3$ and $D4$ in Fig.4. In Fig.5, performance ratio of IM to FP and FUP for $D1$ and $D2$, $D3$ and $D4$ are plotted.

For DB-tree, as a DB-tree can be seen as an FP-tree with a minimum support of 0, the tree is too huge to fit in memory. The system just respond "virtual memory minimum too low", so the DB-tree method is infeasible. We do not plot its runtime.

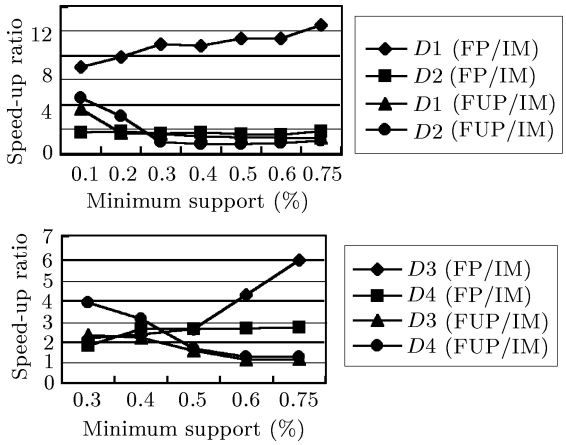



Fig.5. Speed-up ratio with threshold.

### 6.1 IM Versus FP-Growth, FUP

We start the mining each time by using our adapted FP-growth, and then IM. In Fig.3 and Fig.4, as we see, for both datasets, IM wins the other two. When the threshold is very low, the speedup ratio of IM to FUP is much higher. The main reason is that, generally speaking, the FP-tree constructed under a lower threshold is taller than that under a higher threshold. Hence, extension and incremental mining of a bigger FP-tree can save much more time than re-construction and re-executing mining from scratch. Whereas for FUP, in situations with prolific patterns, or quite low threshold, FUP suffers much from generation of a huge number of candidates and repeated scan of databases.

### 6.2 Performance of IM with Large Increment

A database T10I4D50dx with updates of 0.5K, 1K, 2K, 5K, 10K, 20K and 30K are generated, and different updates with different supports are done by IM and FP. For the same support, the speed-up ratio decreases when increment size increases. Fig.6 plots the performance ratio for $D1$ and $D4$

when the threshold is 0.75%. In the same setting of T10I4D50dx, we increase the increment size $x$ from 0.5K gradually to 50K for comparison. The fact that IM still exhibits performance gain when the increment is almost as large as the original database shows that it is efficient.

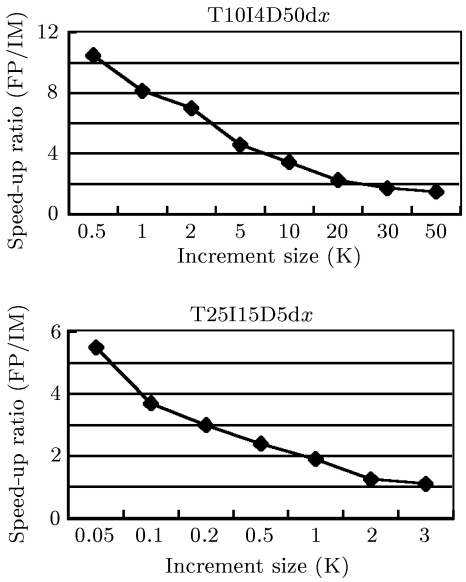



Fig.6. Speed-up ratio vs. increment size.

### 6.3 Small Overhead of IM

Our last experiment is to analyze the overhead incurred by IM. In general, if the time to compute $F^U$ from an updated database $U$ is added to the time to compute the original set $F^{DB}$ from $DB$ by a mining algorithm, the sum would be larger than that if the same mining algorithm was applied directly on $U$ to compute $F^U$. The difference (Diff) of these two runtime values is a measurement of the overhead of the update. If the overhead is small, then it indicates that the update is done very efficiently. The overhead percentage (Diff/FP) is plotted in Fig.7. IM works well even in the case of large increment.

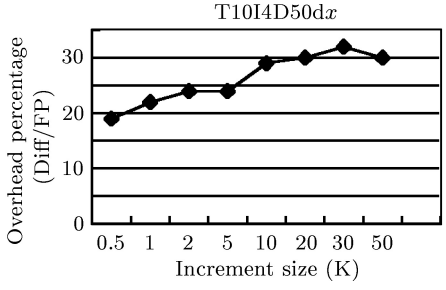



Fig.7. Overhead percentage vs. increment size.

## 7 Conclusion

We have proposed an efficient method for incremental mining frequent patterns when increment comes. Based on re-using materialized by-products of previous mining, incremental mining becomes a procedure of process the increment in a hierarchical way. As the minimum support threshold can be changed when extending an FP-tree, our approach is an integrated solution to incremental mining and re-mining. Moreover, the difference can be directly identified, which is more important and can be used to observe the changing trend each time. As FP-tree can extend continuously, IM is most suitable for continuous increments. We give out the nature of incremental mining in the framework of lattice, thus the idea in this paper is feasible even if we mine through H-Mine or FPL at the first time.

There are a lot of interesting issues related to incremental mining, such as incremental mining of maximal patterns, closed patterns. We also take efforts towards changing trend mining.

## References

[1] Agrawal R, Srikant R. Fast algorithm for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases,* Santiago de Chile, Chile, September 12–15, 1994, pp.487–499.

[2] Zaki M J. Scalable algorithms for association mining. *IEEE Trans. Knowledge and Data Engineering.* 2000, 12(3): 372–390.

[3] Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data,* Dallas, TX, USA, May 14–19, 2000, pp.1–12.

[4] Pei J, Han J, Lu H, Nishio S, Tang S, Yang D. H-Mine: Hyper-structure mining of frequent patterns in large databases. In *Proc. 2001 Int. Conf. Data Mining,* San Jose, CA, USA, Nov.29–Dec.2, 2001, pp.441–448.

[5] Tseng F, Hsu C. Generating frequent patterns with the Frequent Pattern List. *Lecture Notes in Artificial Intelligence* 2035, Cheung D, Williams G J, Li Q (eds.), Springer-Verlag, 2001, pp.376–386.

[6] Cheung D, Han J, Ng V, Wong C. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. 12th Int. Conf. Data Engineering,* New Orleans, Louisiana, Feb. 26–Mar. 1, 1996, pp.106–114.

[7] Cheung D, Lee S, Kao B. A general incremental technique for maintaining discovered association rules. In *Proc. 5th Int. Conf. Database Systems for Advanced Applications,* Melbourne, Australia, April 1–4, 1997, pp.185–194.

[8] Lee S, Cheung D. Maintenance of discovered association rules: When to update? In *Proc. 1997 SIGMOD Workshop on Research Issues on Data Mining and Knowl-*

*edge Discovery* (*DMKD'97*), Tucson, Arizona, May 11, 1997.

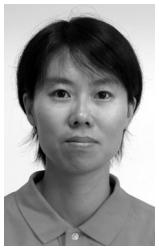
[9] Ezeife C I, Su Y. Mining incremental association rules with generalized FP-tree. *Lecture Notes in Computer Science* 2338, Cohen R, Spencer B (eds.), Springer-Verlag, 2002, pp.147–160.

[10] Liu J, Yin J. Towards efficient data re-mining (DRM). *Lecture Notes in Artificial Intelligence* 2035, Cheung D, Williams G J, Li Q (eds.), Springer-Verlag, 2001, pp.406–412.

[11] Ma X, Tang S, Yang D, Du X. Towards efficient re-mining of frequent pattern upon threshold changes. *Lecture Notes in Computer Science* 2419, Meng X, Su J, Wang Y (eds.), Springer-Verlag, 2002, pp.80–88.

[12] Du X, Tang S, Makinouchi A. Maintaining discovered frequent itemsets: Cases for changeable database and support. *Journal of Computer Science and Technology*, Sept. 2003, 18(5): 648–658.

[13] Davey B A, Priestley H A. Introduction to Lattices and Order. Cambridge Univ. Press, 1990.



**Xiu-Li Ma** received the Ph.D. degree in computer science from Peking University in 2003. She is currently a post-doctoral researcher at National Lab on Machine Perception of Peking University. Her main research interests include data warehousing, data mining, intelligent online analysis, and sensor network.

**Yun-Hai Tong** received the Ph.D. degree in computer software from Peking University in 2002. He is currently an assistant professor at School of Electronics Engineering and Computer Science of Peking University. His research interests include data warehousing, online analysis processing and data mining.

**Shi-Wei Tang** received the B.S. degree in mathematics from Peking University in 1964. Now, he is a professor and Ph.D. supervisor at School of Electronics Engineering and Computer Science of Peking University. His research interests include DBMS, information integration, data warehousing, OLAP, and data mining, database technology in specific application fields. He is the vice chair of the Database Society of China Computer Federation.

**Dong-Qing Yang** received the B.S. degree in mathematics from Peking University in 1969. Now, she is a professor and Ph.D. supervisor at School of Electronics Engineering and Computer Science of Peking University. Her research interests include database design methodology, database system implementation techniques, data warehousing and data mining, information integration and sharing in Web environment. She is a member of academic committee of Database Society of China Computer Federation.