

An Efficiently Algorithm for Mining Association Rules

Fu-zan Chen, Jian-min Hang, Qin Zhang

Abstract—Association rules mining is one of the most important topic in data mining. A new algorithm for mining association rules is proposed in this paper. In data mining, the process of counting any itemset's support requires a great I/O and computing cost. An impacted bitmap technique to speed up the counting process is employed in this paper. Nevertheless, saving the intact bitmap usually has a big space requirement. In this algorithm, each bit vector is partitioned into some blocks, and hence every bit block is encoded as a shorter symbol. Therefore the original bitmap is impacted efficiently. And then the algorithm converts the origin transaction database to an Adjacent-Itemsets-lattice (which is a directed graph) in a preprocessing, where each itemset vertex has a label to represent its support. So we can change the complicated task of mining frequent itessesets in the database to a simpler one of searching vertexes in this structure, which can speeds up greatly the mining process. At the end experimental and analytical results are presented.

Index Terms—Data mining, Association rules, Impacted Bitmap, Adjacent-Itemsets-lattice

I. INTRODUCTION

Data mining has recently attracted considerable attention from database practitioners and researchers because of its applicability in many areas such as decision support, market strategy and financial forecasting. One of the most important research topics in data mining is the discovery of association rules in large databases of sales transactions^[1,2]. It has been applied in applications such as market-basket analysis for supermarket, linkage analysis for website, sequence-pattern in bioinformatics etc.

1.1 Problem Descriptions

Association rules can be stated as follows^[1,2]. Let I be the set of items and D a transaction database in which each transaction is a subset of I and is associated with a unique identifier called its TID . A set of items is called an itemset. We say that a transaction supports an itemset X if all the items in X are contained in the transaction. The *support* of an itemset X is the percentage of transactions in D which support

X : $support(X) = \frac{\|\{t \in D | X \subseteq t\}\|}{\|\{t \in D\}\|}$. The *cover* of an itemset X consists of the set of transaction identifiers of transactions in D that support X . An association rule is a conditional implication among itemsets: $r = X \Rightarrow Y$, where $X, Y \subset I$ and $X \cap Y = \phi$. The confidence of rule r is defined as $confidence(r) = support(X \cup Y) / support(X)$, and the support is $support(r) = support(X \cup Y)$.

Given the defined thresholds for the permissible minimum support and confidence, the procedure of mining association rules can be broken into two sub-problems^[1,2]:

- Find all *frequent* itemsets X in D , i.e. itemsets with support greater or equal to *minsupport*.
- For each frequent itemset X and its proper subsets Y , generate all association rules $Y \Rightarrow X - Y$, with confidence greater or equal to *minconfidence*.

The second sub-problem can be solved in the main memory in a straightforward manner once all frequent itemsets and their support values are known. Hence the problem of mining association rules is focused on finding frequent itemsets. Major research efforts have spent on finding frequent itemsets, and many algorithms have been proposed.

1.2 Related Work

There are many association rule algorithms developed to reduce the scans of transaction database and to shrink number of candidates. These algorithms might as well be classified into two categories:

1.2.1 candidate-generation-and-test approach (such as Apriori).

The most influential algorithm Apriori developed by Rakesh .etc generates the k -candidate by combining two frequent $(k-1)$ -itemsets. The process is very time consuming. The Apriori-like algorithm employs a bottom-up, breadth-first searching that generates all frequent itemsets, which is feasible with sparse datasets such as market-basket data, where the frequent patterns are very short. However, for some application domain such as genome data where there are many, long frequent patterns, the performance of these algorithms degrades incredibly because they perform as many passes over the database as the length of the longest frequent pattern. This incurs high I/O overload for iteratively scanning large database^[1,2].

1.2.2 pattern-growth approach^[11,12].

Instead of storing the cover for every item the database, the FP-tree stores the actual transactions from the database in a

This work was supported in part by the by National Science Foundation (No. 70571057)

The authors are with the school of management in Tianjin University, Tianjin, 300072, China (Email: fzchen@tju.edu.cn)

multi-tier structure and every item has a linked list going through all transactions that contain it. Obviously, the FP-tree is just like the vertical and horizontal database layouts which constitutes a lossless representation of the complete transaction database for the generation of frequent itemsets. Indeed, the linked list starting from an item in the header table actually represents a compressed form of the cover of that item. On the other hand, a branch starting from the root node represents a compressed form of a set of transactions.

The efficiency of FP-Tree algorithm is accounted for by:

- The FP-Tree is a compressed representation of the original database. Only frequent items are used to construct the tree, and other irrelevant information are pruned. By ordering the items according to their supports, the overlapping parts appear only once with different support count.
- This algorithm only scans the database twice. The frequent patterns are generated by the FP-growth procedure. By constructing the conditional FP-Tree which contain patterns with specified suffix patterns, frequent patterns can be easily generated as shown in the above example. So the computation cost is decreased dramatically.
- The FP-Tree uses a divide-and-conquer method that considerably reduces the size of the subsequent conditional FP-Tree, longer frequent patterns are generated by adding a suffix to the shorter frequent patterns. There are examples to illustrate all the detail of this mining process in [11].

It is difficult to use the FP-Tree in an interactive mining system. During the interactive mining process, users may change the threshold of support according to the rules. However the changing of support threshold may lead to repetition of the whole mining process for the FP-Tree. Besides, the FP-Tree is not suitable for incremental mining when new datasets are inserted into the database. Pattern-growth approach adopts the divide-and-conquer strategy, and finds frequent itemsets for each sub dataset based on the FP-tree. But like the Apriori algorithm, it also needs to generate and test all frequent itemsets in all subspaces.

There remain many challenging issues for the association rule algorithms. The procedure scans the databases multiple times, which make it very hard to be scalable for large data sets. The algorithms generates and tests the huge number of itemsets to make sure that all the frequent itemsets are generated, which results in a complete set of association rules. Many novel ideas have been proposed to deal with these issues. To reduce the search space, some algorithms just mine all the closed frequent itemsets, instead of all frequent itemsets, which is a 2-3 magnitude less than the frequent itemsets. It is not necessary to discover all the association rules. Therefore a set of non-redundant association rules, which can be mined from the closed frequent itemsets, is generated. These rules can be used to infer all the association rules^[12]. In addition, most of the previous algorithms are based on the traditional horizontal database format for mining. Recently a number of vertical mining algorithms have been proposed for association mining

such as CHARM, VIPER, etc. In vertical database each item is associated with its corresponding transaction id (called tidlist). Mining algorithms using the vertical format have shown to be very effective and usually outperform horizontal approaches because frequent pattern can be counted via tidlist/granule/bitmap intersections in the vertical approach^[6,7,8,9,10].

1.3 Outline

Based on the recent advance in these areas, we propose an efficient algorithm based on the impacted bitmap and Adjacent-Itemsets-lattice structure. The remaining of this paper is organized as follows. In section 2 we analyze the problem of itemsets support counting, which is based bitmap index technology. In section 3 we give a detailed description and some theories about the Adjacent-Itemsets-lattice. The constructing and decomposition strategies are also introduced. We employ an itemsets searching strategy to find all frequent itemsets. Experimental and analytical results are discussed in section 4. Finally, we present some feasible optimizations and conclude this paper in section 5.

II. SUPPORTS COUNTING TECHNOLOGY

Three important components constitute the cost during the procedure of supports counting: 1) reading database from disk to main memory; 2) building an index(e.g. Hash Tree) on candidates for efficient counting; 3) and counting candidates against the database. Aiming at efficient calculation of the supports counting, we employ a new bitmap-based technology to optimize both I/O and CPU time, which require less disk space and memory than conventional methods.

2.2 Supports Counting Based on Bitmap

The bitmap technique was proposed in the 1960's, and has been used by a variety of products. A typical context is the modern relational DBMS (i.e. Oracle 9i), which implement bitmap indexes for accelerating join and aggregate computation. Bitmap also has been applied in association mining^[6,7,8,9,10]. The bitmap technique shows far superior query performance on unselective (low cardinality) data than traditional B-Tree indexing techniques. The bitmap represents the relational table of the database as matrixes or arrays of bits, where 1 or 0 in each relative position represents True or False.

Traditional Apriori algorithms require full table scan and multiple passes of the itemsets in order to finding association rules from a large database. We present a novel support counting algorithm based on bitmap computing approach, which avoids these time-consuming operations and relies on the fast bit operations to find the large itemsets. With the bitmap technique, sometimes referred as inverted-list, we can greatly improve the performance of the association rule algorithms with a high processing speed. Distinctively, the bitmap proposed is not the same as the one adopted in DBMS. Instead of using the initial bitmaps, we manipulate the encoded ones, which needs fewer space for the disk and main memory.

The key idea of the approach is to use a bitmap index to

determine which transactions contain which itemsets. Each transaction has one unique offset position in the bitmap. A bit vector $X.Bit = (b_1, b_2, \dots, b_n)$ is associated to each itemset X . In $X.Bit$ the i^{th} bit b_i is set to 1 if the transaction i contains the itemset X , and otherwise b_i is set to 0. It should be noted that b_i is set to null if itemsets or transaction does not exist. The ordered collection of these bit vectors composes the bitmap. In a bitmap, a line represents a transaction, while a column corresponds to a given k -itemset. The bitmap for 1-itemsets is just a classical bitmap index implemented in Oracle. Combining the process of performing a logical operation (AND/OR/NOT) on a series of bitmaps is very efficient, particularly compared with performing similar processes on lists of transaction.

For example, in the transaction database D shown in table1, bit vectors associated with 1-itemsets are such: $\{A\}.Bit=(1011100010)$, $\{B\}.Bit=(1010110111)$, $\{C\}.Bit=(1111110110)$, $\{D\}.Bit=(0101111010)$. The collected bitmap is shown in table2.

During the supports counting, algorithm intensively

TABLE I
DATABASE D

TID	Itemsets
1	ABC
2	CD
3	ABC
4	ACD
5	ABCD
6	BCD
7	D
8	BC
9	ABCD
10	B

TABLE
BITMAP OF 2-ITEMSETS

	1	2	3	4
1	1	1	1	0
2	0	0	1	1
3	1	1	1	0
4	1	0	1	1
5	1	1	1	1
6	0	1	1	1
7	0	0	0	1
8	0	1	1	0
9	1	1	1	1
10	0	1	0	0

manipulates bit vectors, and even store the intermediate ones, which requires a lot of disk space and main memory. So, it is necessary to compress the bitmap in advance. A new block strategy is proposed to encode and decode the bitmap, which is similar to the pagination technology in operating systems. In this approach, every bit vector is partitioned into fractions, called blocks, that can be encoded respectively, so that a bitmap is divided into granules. Each block should have an appropriate size, if the size is too small, the impact is not remarkable; otherwise the encoding is not straightforward. In order to take full advantage of Logical Calculation Units, the block size should be an exponential to 2. Each block is represented as $(p : W)$, p is the number of the block, and W is the block bit vector. Let l be the block size, m the number of transactions in database D . In this way each bit vector $B = (b_1, b_2, \dots, b_i, \dots, b_m)$ can be partitioned into $p = \text{INT}(m/l)$ blocks (INT is the integer function). The k^{th} block vector $W_k = (w_1, w_2, \dots, w_j, \dots, w_l)$, $j = \text{MOD}(i, l) = i - l * (k - 1)$ (MOD is the mode function).

For example, when we initialize the block size as 16, every 16 bits forms a block. Thus the total number of blocks is the

total number of transactions divided by 16. If a database contains 10K transactions, each bit vector B consists of $p = \text{INT}(10,000/16) = 625$ blocks. If the 19th bit in B $b_{i=19} = 1$, we can Figure out that the 3rd bit $w_{j=3} = 1$ in the $k = 2$ block. If the transaction set $\{1,3,4,9,11,18,20$ supports itemset X , and $\{4,11,18,24,31\}$ supports itemset Y respectively, dividing by 16 yields $X.Bit=\{(1:1011\ 0000\ 1010\ 0000, (2:0101\ 0000\ 0000\ 0000)\}$, $Y.Bit=\{(1:0001\ 0000\ 0010\ 0000, (2:0100\ 0001\ 0000\ 0010)\}$. The intersection of two bit vectors is accomplished by the logical operation AND. Thus the corresponding bit vector of their join set $X \cup Y$ is $XY.Bit=\{(1:0001\ 0000\ 0010\ 0000, (2:0100\ 0001\ 0000\ 0010)\}$. That is to say transaction set $\{4,11,18\}$ support both itemset X and Y .

Encoding each block as a shorter code can reduce the space demanding. The encoding principle which should be conformed is that each block can be represented uniquely. As a part of the initial bit vector B , each block vector W is also a binary bit vector. The conversion between binary, octal, decimal and hexadecimal can be implemented conveniently, hereby every block can be represented a binary, octal, decimal or hexadecimal code. We use a hexadecimal code, i.e. every four bits in a block encode a hexadecimal code. For instance applying this encoding, the hexadecimal vectors associated with previous itemsets are $X.Bit=\{(1:C0A0, (2:5000)\}$ and $Y.Bit=\{(1:1020, (2:4102)\}$.

As each itemset is associated to a binary bit vector, the support of a given itemset is the total number of 1 in the vector. For the sake of efficient counting the number of 1, we previously store the binary block in a bit array $Bit[1 \dots l]$ (l is the block size), and the hexadecimal blocks in an array $ABit[1 \dots p]$ (p is the number of blocks). The value in $ABit[i]$ is the hexadecimal code of the i^{th} block. The lemma that a k -itemset $X = \{i_1, \dots, i_k\}$ can be obtained by the join between its two arbitrary different $(k-1)$ -subsets $X_1 = \{i_1, \dots, i_{k-2}, i_{k-1}\}$ and $X_2 = \{i_1, \dots, i_{k-2}, i_k\}$. The corresponded bit vector can also be obtained by the intersection of these twos. Implementation of this support counting algorithm follows.

Algorithm 1 Itemsets Support Counting

```

Algorithm Countsupport( $X_1, X_2$ )
Begin
  Support=0;
  For ( $i=1; i=p; i++$ ) do
    If  $X_1.ABit[i] \neq 0$  and  $X_2.ABit[i] \neq 0$  then
      For ( $j=1; j=l; j++$ ) do
         $X.Bit[j] = X_1.Bit[j] \& X_2.Bit[j]$ ;
        Support+= X.Bit[j];
      Endfor;
       $X.ABit[i] = X_1.ABit[i] \& X_2.ABit[i]$ ;
    Else
       $X.ABit[i]=0$ ;
    endif;
  end;
end;

```

III. FREQUENT ITEMSETS SEARCH

In this section, we begin with some basic definition and notions, and then discuss how to use these techniques to represent itemsets and organize the itemsets in an Adjacent-Itemsets-lattice structure. Considering the limitation on main memory, we deal with the problem of decomposition in addition.

3.1 Adjacent-Itemsets-lattice

The lattice theory could be referred in [3,15]. We only introduce some concepts and theorems to be adopted in this paper.

Let (P, \subseteq) be an ordered set, and let A be a subset of P . The $J \in P$ is an *upper bound* of A if $J \subseteq X$ for all $X \in A$, furthermore J is the *least upper bound* of A if there is no element $Y \in P$ that $J \neq Y$ and $J \subseteq Y$. Similarly, an element $M \in P$ is a *lower bound* of A if $M \subseteq X$ for all $X \in A$, M is the *greatest lower bound* of A if there is no element $Y \in P$ that $M \neq Y$ and $Y \subseteq M$. Typically, the least upper bound is called the *join* (or top element) of A , and is denoted as $\vee A$ (or \overline{A}); the greatest lower bound is called the *meet* (or bottom element) of A , and is denoted as $\wedge A$ (or \underline{A}). For $A = \{X, Y\}$, we also write $X \cup Y$ for the join, and $X \cap Y$ for the meet. Let M be the meet of P and $X \in P$, L be the meet of P , if there is no $Y \in P$ that $M \subset Y \subset X$, then we called X is a *atom* of P . The set consisting of all of the atoms of P is denoted $A(P)$.

Definition 1 An ordered set (L, \subseteq) is a *lattice*, if for any two elements $X, Y \in L$ the join $X \vee Y = X \cup Y$ and meet $X \wedge Y = X \cap Y$ always exist. L is called a *join semi-lattice* if only the join exists. L is called a *meet semi-lattice* if only the meet exists. L is a *complete lattice* if $\wedge S$ and $\vee S$ exist for all $S \subseteq L$. Any finite lattice is complete.

It is deduced that the power set $P(I)$ on the items I derived in the transaction database D is a lattice, and also a complete one, $P(I)$ is called as *power-set-lattice*. For this $P(I)$ the join is I , the meet is ϕ , and each atom is $\{i | i \in I\}$.

Let I be the itemset derived in transaction database D , $X, Y \subseteq I$ and $X \neq Y$, if Y can be obtained from X by adding a single item, then X is said to be *adjacent* to Y . We call X is a *parent* of Y , and Y is a *child* of X . Thus, an itemset may possibly have more than one parent and more than one child. For each itemset X and each $x_k \in X$, itemset $X - \{x_k\}$ is a parent of X . In fact, the number of parents of an itemsets X is exactly equal to the cardinality of the set X ^[15].

Definition 2 Let $I = \{i_1, i_2, \dots, i_m\}$ be the itemset derived in transaction database D , the *Adjacent-Itemsets-Lattice* L on the ordered powerset $P = (P(I), \subseteq)$ of I be a DAG which each vertex is an itemset $X \subseteq I$. L is constructed as follow: For each primary itemset X , construct a graph with a vertex $v(X)$, each

vertex has a label corresponding to the value of its support, denoted as $S(X)$. For any pair of vertices corresponding to itemsets X and Y , if and only if X is a parent of Y , a directed edge exists from $v(X)$ to $v(Y)$, denoted as $E(X, Y)$. The vertex $v(X)$ is said to be the *head* of the edge $E(X, Y)$, and the vertex $v(Y)$ is said to be the *tail* of the edge $E(X, Y)$.

Thus, the search space of all itemsets can be represented by an Adjacent-Itemsets-lattice, with the empty itemset at the top and the set containing all items at the bottom.

Apparently, the fact there is a directed path from vertex $v(X)$ to vertex $v(Z)$ in the Adjacent-Itemsets-lattice L , implies itemset X is a proper subset of itemset Z , i.e. $X \subset Z$. Especially, we call X is an *ancestor* of Z , and Z is a *descendent* of X .

For any itemset, it has a property that if the itemset is frequent then all of its subsets are also frequent. Obviously, the frequent itemsets consists a meet semi-lattice. This property can be used to pruning efficiently in the lattice searching.

Considering the database D illustrated in Table1, there are five different items, i.e., $I = \{A, B, C, D\}$, The corresponding Adjacent-Itemsets-lattice L is illustrated in Figure 1. Each vertex has a label corresponding to its support value. Furthermore, the vertices with box in Figure 1 are the maximal frequent itemsets.

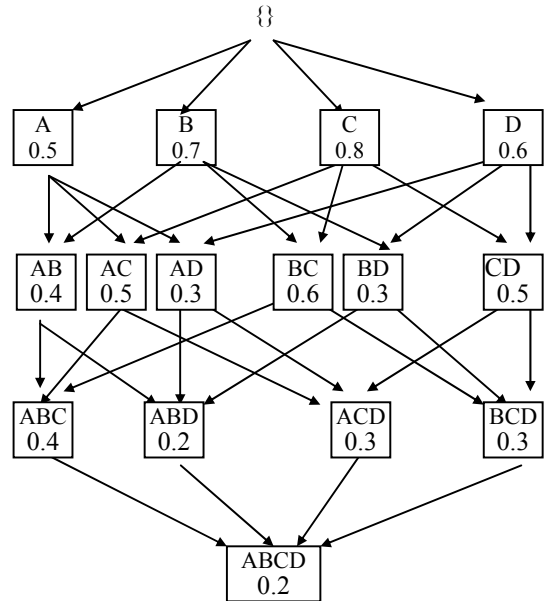


Fig.1. The itemsets-lattice $L=P(I)$

Property 1 Each vertex $X \in L$ in an Adjacent-Itemsets-lattice L can be denoted by $X = \cup \{Y \in A(L) | Y \leq X\}$.

Theorem 1 Let L be an Adjacent-Itemsets-lattice, $v(X)$ and $v(Y)$ are different vertices in L . $v(Y)$ is a descendent of $v(X)$ implies $S(Y) \leq S(X)$.

Proof: It can be derived from definition 6 and the definition of support.

Theorem 2 The number of edges in the Adjacent-Itemsets-lattice is exactly equal to the sum of the number of items in the vertex itemsets.

Proof: The number of edges may be obtained by summing the number of parents of each vertex itemset. The number of parents of a vertex itemset is exactly equal to the number of items in it. The result follows.

3.2 Adjacent-Itemsets-lattice constructing

The Adjacent-Itemsets-lattice expresses an ordered relation on the itemsets, and is an equivalent representation of original transaction database D . Instead of scanning the entire database iteratively, we employ bitmaps to count supports (as shown below). As a precondition, the signification of the group of ilk transactions is the same as an individual. Since there are large numbers of identical transaction in database D , a preprocess for clustering can be carried out to speed up the procedures before constructing Adjacent-Itemsets-lattice. The same transactions are gathered into a cluster, and then transactions selected once randomly from each cluster consist a new transaction set D' that only contains the distinct transactions from D . This clustering approach avoids lots of unnecessary repetitions, which can reduce the I/O effort efficiently.

Let D' be a transaction set derived from database D , N be the number of transactions in D' , and $I = \{i_1, i_2, \dots, i_m\}$ be the itemset contained in D , the following algorithm can construct the corresponding Adjacent-Itemsets-lattice $L = P(I)$.

Algorithm2 Adjacent-Itemsets-lattice constructing

Algorithm *ConstructItemsetsLattice*(D')

Begin

$L = \phi$;

For all transaction $t \in D'$ do

For each itemset $X = \{i_1, \dots, i_r\}$ in t do

$S(X) = CountSupport(X)$;

Add the vertex $v(X)$ to L with lable $S(X)$;

Add the edge $E(X - \{i_k\}, X)$ to L for each $k \in \{1, \dots, r\}$;

endfor;

endfor;

end;

3.3 Adjacent-Itemsets-lattice decomposition

If we had enough main memory, we could enumerate all the frequent itemsets by traversing the Adjacent-Itemsets-lattice. With only a limited amount of main memory in practice, we need to decompose the original lattice into some smaller pieces such that each one can be solved independently in main memory^[16].

Definition 3 Let L be an Adjacent-Itemsets-lattice, and $S \subset L$. For each vertex $A, B \in S$, if $A \vee B = A \cup B \in S$ and $A \wedge B = A \cap B \in S$, S is said to be a sub-lattice of L .

Definition 4 Let f be a function on ordered itemset X , which selects the first k items from X , i.e. $f(X, k) = X[1:k]$. If $X \equiv_k Y \Leftrightarrow f(X, k) = f(Y, k)$ for all $X, Y \in L$, then the binary relation \equiv_k is called an equivalence relation on X .

Theorem 3 In Adjacent-Itemsets-lattice L , each equivalent class $[X]_k$ induced by equivalence relation \equiv_k is a complete sub-lattice of L .

Proof: $\forall A, B \in [X]_k$, i.e., A and B have the same sub-itemset X . $A \cup B \supseteq X$ implies $A \cup B \in [X]_k$, and $A \cap B \supseteq X$ implies $A \cap B \in [X]_k$. That is to say there are always $A \wedge B = A \cap B \in S$ and $A \vee B = A \cup B \in S$. Then S is a sub-lattice of L following definition 4. Let $S = \{A_1, A_2, \dots, A_n\}$, meet of S is $\bigcap_{i=1}^n A_i = X$, and the join of S is $\bigcup_{i=1}^n A_i$. The result follows from definition3.

We can generate four independent sub-lattices by applying \equiv_1 to Adjacent-Itemsets-lattice shown Figure 1, the sub-lattice corresponding to $[A]_1$ and $[B]_1$ respectively is shown in Figure 2, $[C]_1$ contains two itemsets C and CD , and $[D]_1$ contains only one itemset D .

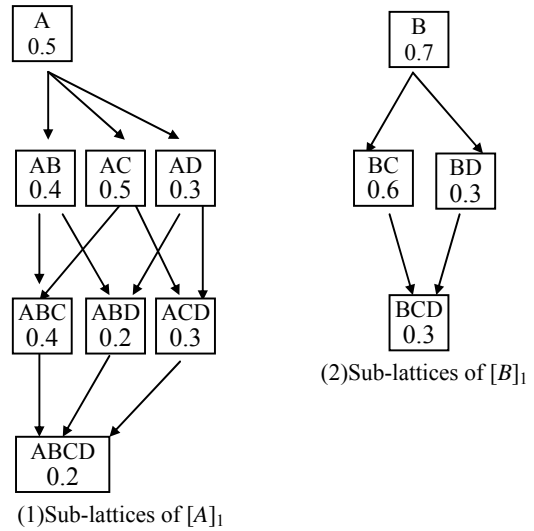


Fig.2. Sub-lattices of $L=P(I)$ induced by \equiv_1

In some cases, a sub-lattice may still be too large for the main memory. We then can apply recursive decomposition to sub-lattice until all the sub-lattices can be solved independently in main memory.

3.4 The Searching Algorithm

Let s be the minimum support, to find all frequent itemsets, we need to solve the following search problem in the Adjacent-Itemsets-lattice: for a given itemset X (i.e. bottom element, top element or some medi-vertex), find all itemsets Y such that $v(Y)$ is reachable from $v(X)$ by a directed path in the lattice or sub-lattice, and satisfies $S(Y) \geq s$.

In practice, the number of vertices reachable from a given vertex may be quite large, though the number of vertices, which

satisfy the minimum support condition, may be small. Using the lattice and sub-lattice structure can restrict the number of vertices checked. In order to restrict the number of vertices checked in the lattice, we sort the itemset vertices by their support value in inverted sequence for each level.

We discuss an efficient strategy for enumerating the frequent itemsets in lattice, denoted Breadth-First search. The Breadth-First search approach starts with the bottom element of the sub-lattice, traverses the lattice level by level. Just the same as saying, we check each vertex at the next level only after the end of checking the certain level. This approach enumerates all frequent itemsets.

Algorithm 3 Breadth-First search strategy

Algorithm Breadth-First-search(S)

Begin

$FS = \{v(B), S(B)\}$; $List = \{v(B)\}$;

While $List \neq \phi$ do

 Select $v(R)$ from $List$;

 For each unvisited child $v(T)$ of $v(X)$ do

 If $S(Y) \geq minsup$ do

$List = List \cup v(Y)$;

 Add $(v(Y), S(Y))$ to FS

 For each $i_j \in Y$ add $E(Y - \{i_j\}, Y)$ to FS

 endfor;

 enddo;

 Delete $v(R)$ from $List$

End;

Each superset of a infrequent itemset is also infrequent. Thus, we can increase searching efficiency of the algorithm by pruning the vertices such that not satisfying the minimum support condition.

3.5 Algorithm Analysis and Optimization

The performance of these candidate-generation-and-test algorithms degrades incredibly because these algorithms perform as many passes over the database as the length of the longest frequent pattern. This incurs high I/O overhead for scanning large disk-resident database many times. Furthermore, this kind of approach finds frequent itemsets according to the given minimum support. When the support changes, algorithm has to be performed on the entire database again, each itemset is also regenerated and its support is also recounting. Thus the previous computed result cannot be reused is still a side issue. We convert the original transaction database to an Adjacent-Itemsets-lattice in a preprocessing, where each itemset vertex has a support label. Since then when the given support changes, we only need traverse the Adjacent-Itemsets-lattice straightly to finding frequent itemsets, in spite of manipulating the transaction database complicatedly. Such Adjacent-Itemsets-lattice can be mined for many times without numbers. Hence, the preprocessing cost can be contributed, and the more mining task performs, the little average effort is.

The search space of all itemsets contains exactly $2^{|I|}$ different itemsets. If itemset I is large enough, then the naive approach to generate and count the supports of all itemsets over the database can not be achieved within a reasonable period of time. Instead, we could generate only those itemsets that occur at least once in the transaction database. More specifically, we generate all subsets of all transactions in the database. Of course, for large transactions, this number could still be too large. Therefore, as an optimization, we could generate only those subsets of at most a given maximum size. This technique would pay off for sparse transaction databases. For very large or dense databases, an alternative method is to generate subsets of at least a given support threshold. In such a manner those overlong itemsets would be pruned. This technique can decrease both space and compute effort consumedly. The Adjacent-Itemsets-lattice constructing is a long-time spending process. Buffer can decrease the disk I/O, and multiprocessor can parallelize the processing, these ways should accelerate the speed.

The itemset has an important property: if an itemset is infrequent, all of its superset must be infrequent. In other words all the infrequent itemsets compose a meet semi-lattice. There has no frequent itemset in the sub-lattice which bottom element is infrequent. Thus, we only need to traverse the sub-lattice which bottom element is frequent. Applying this property can minimize the I/O costs when enumerating frequent itemsets.

IV. EXPERIMENTAL AND ANALYTICAL RESULTS

We run the simulation on PC: Intel P4 2.4G CPU, 512MB main memory and Windows 2000 Server. The synthetic data set are generated using a method provided by KDD Research Group in IBM Almaden Research Center, referring to <http://www.almaden.ibm.com/cs/quest/syndata#AssocSynData>. Let D denote the number of transactions, T the average transaction size, I the size of a maximal potentially frequent itemset. A data set having $T=x$, $I=y$, $D=mK$ is denoted by $Tx.Iy.DmK$. Our approach is a three stages process, and enumerates the frequent itemsets in the third stage:

- 1) Constructing the Adjacent-Itemsets-lattice, decomposing the constructed lattice into sub-lattices if necessary.
- 2) Traversing the lattice and enumerating the frequent itemsets.

Figure 3 shows the response time variation with average transaction size during lattice constructing. The data sets are $Tx.I3.D100K$ and $Tx.I3.D100K$, the transaction size increases 1 every time. This shows that response time of lattice constructing and the average transaction size is an exponential relation. The computational effort and the scale of constructed lattice are more sensitive to the average transaction size, rather than to the number of transactions. We also find that the computational effort mushrooms when some transactions have very great length, even though the average transaction size is small.

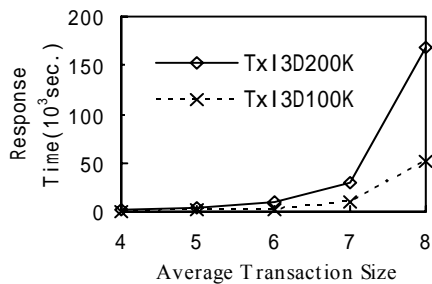


Fig.3. Response time variation with average transaction size during lattice constructing

Figure 4 shows the response time variation with the number of frequent itemsets enumerated by applying Breadth-First search strategy to Adjacent-Itemsets-lattice. The data sets are T5.I3.D100K and T5.I3.D200K. This shows that response time of frequent itemsets enumerating is more sensitive to the number of frequent itemsets searched, rather than to the average transaction size and number of transactions in the data set.

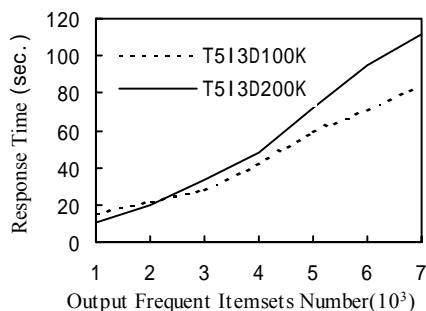


Fig.4. Response time variation with number of frequent itemsets searched

V. CONCLUSION

In this paper a new efficient approach for mining association rules is proposed based on the lattice theory and bitmap technology. These two kinds of technique have been fully researched and successfully used in a variety of fields for last decades. We convert the original transaction database to an Adjacent-Itemsets-lattice in the preprocessing, where each itemset vertex has a label to save its support. Thus the complicated task of mining frequent itemsets in the database is simplified to vertices searching in the lattice, which can speed up greatly the frequent itemsets mining process. Considering the problem that there may not have enough main memory to process the entire lattice, we discuss the solution for lattice recursive decomposition.

We use an improved compacting bitmaps database format. We count the support of itemset by means of binary bit vectors intersections, which minimizes the I/O and computing cost. To reduce the disk and main memory space demanding, we break

the bitmap down into some less blocks, which can be encoded as a shorter code. The blocks of bitmaps are fairly adaptable. Hence the additional space decreases rapidly.

REFERENCES

- [1] Qiankun Zhao, Association Rule Mining: A Survey, Technical Report, CAIS, Nanyang Technological University, Singapore, 2003, No. 2003116
- [2] Jochen Hipp, Algorithms for Association Rule Mining: A General Survey and Comparison, ACM SIGKDD, 2000, Vol.2 Iss.1 pp.58-65
- [3] J. B. Nation, Notes on Lattice Theory, <http://www.Hawaii.edu>
- [4] K.K. Loo, et al., A Lattice-based approach for I/O efficient association rule mining, Information Systems, 27(2002) pp.41-74
- [5] Huaiguo Fu, Partitioning large data to scale up lattice-based algorithm, ICTAI'03, 2003
- [6] T.Y. Lin, Xiaohua Hu, and Eric Louie, A Fast Association Rule Algorithm Based On Bitmap and Granular Computing, The Proceedings of the IEEE International Conference Fuzzy Systems, 2003, pp.678-683
- [7] Xiaohua Hu, T.Y. Lin, Eric Louie, Bitmap Techniques for Optimizing Decision Support Queries and Association. Rule Algorithms, IDEAS 2003
- [8] Mikolaj Morzy, Hierarchical Bitmap Index An Efficient and Scalable Indexing Technique for Set-Valued Attributes, ADBIS 2003, LNCS 2798, 2003, pp. 236 – 252
- [9] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, Sequential Pattern Mining Using a Bitmap Representation, Proc. of the 8th Int. Conf. on Knowledge Discovery and Data Mining, 2002, pp. 429-435
- [10] G. Gardarin, P. Pucheral, and F. Wu, Bitmap Based Algorithms for Mining Association Rules, Proc. of the 14th Bases de Donnes Avancees, 1998, pp. 157-176
- [11] J. Han, J. Pei, Y. Yin, and R. Mao, Mining Frequent Patterns without Candidate Generation, Proc. 2000 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'00), Dallas, TX, May 2000
- [12] Zaki Mohammed J., Mining Non-Redundant Association Rules, Data Mining and Knowledge Discovery, 3(2004), pp.223-248
- [13] Haixun Wang, Demand-driven frequent itemset mining using pattern structures, Knowledge and Information Systems, 2004,
- [14] Shinichi Morishita, Traversing Itemset Lattices with Statistical Metric Pruning, Symposium on Principles of Database Systems, 2000, pp. 226-236
- [15] Hu Changliu, The basic of Lattice Theory, Henan University publishing company, 1990