Binary Partition Based Algorithms for Mining Association Rules

Jianlin Feng
Department of Computer Science
Huazhong University of Science and Technology
Wuhan 430074, China
E-mail: dbinst@hust.edu.cn

Yucai Feng
Department of Computer Science
Huazhong University of Science and Technology
Wuhan 430074, China
E-mail: dbinst@hust.edu.cn

Abstract

Mining association rules is an important data mining problem. A binary partition based fast algorithm BPA for mining association rules in large databases is presented in this paper. Basically, the framework of BPA is similar to that of algorithm Apriori. In the first pass, all the frequent 1-itemsets can be divided into two disjoint parts. Accordingly in each subsequent pass k, we partition the set of all the frequent k-itemsets into three subsets. Any two different partitions are disjoint. If necessary, this partition procedure can be a recursive one. Therefore we get a binary partition tree in the first pass and a corresponding ternary partition tree in each subsequent pass k. Due to such a partition, BPA can be very easily parallelized assuming a shared-memory architecture.

1. Introduction

Data mining, also known as knowledge discovery in databases, has been recognized as a new area for database research. This area can be loosely defined as the efficient discovery of previously unknown patterns or rules in large databases. Database mining is motivated by the decision support problem faced by most large retail organizations. Progress in bar-code technology has made it possible for large supermarkets to collect and store massive amounts of sales data, referred to as the basket data. A record in such data typically consists of the customer-id, the transaction date and the items bought in the transaction. Analysis of the big amount of past transaction data can provide very valuable information on customer buying behavior, and thus we can improve the quality of business decisions. [1, 2]

The problem of mining association rules over basket data was introduced in [1]. An example of such a rule might be that 90% of customers that purchase bread and butter also purchase milk. The intuitive meaning of such a rule is when customers purchase some items how they will tend to purchase some other items too. Finding all such rules is valuable for cross-marketing and attached mailing applications. Other applications include catalog design, add-on sales, store layout, and customer segmentation based on buying patterns. The databases involved in these applications are very large. Therefore, it is imperative to design efficient algorithms to mine association rules. [1, 2, 3, 4, 8, 9, 10, 11].

In this paper, an efficient algorithm BPA (Binary Partition Based Algorithm) is presented. The rest of the paper is organized as follows. A formal description of mining association rules is first given in section 2. Then we discuss the related works in section 3. In section 4, we describe the BPA algorithm in detail, and discuss scale-up problem and parallelization at the same time. The relative performance study is discussed in section 5. Finally we conclude with a summary in section 6.

2. Mining of Association Rules

The following is a formal statement of the problem of mining association rules: [2]

Let $I = \{i_1, i_2 \cdots i_m\}$ be a set of literals, called items. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. Associated with each transaction is a unique identifier, called its TID. We say that a transaction T contains X, a set of some items in I, if $X \subseteq T$. An association rule is an implication of the form $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \bigcap Y = \varnothing$. The rule $X \Rightarrow Y$ holds in the transaction set D with confidence c if $c\%$ of transactions in D that contain X also contain Y. The rule $X \Rightarrow Y$ has support s in the transaction set D if $s\%$ of transactions in D contain $X \bigcup Y$.

Given a set of transactions D, the problem of mining association rules is to generate all association rules that have support and confidence greater than the user-specified minimum support (called *minsup*) and minimum confidence (called *minconf*) respectively.

The problem of discovering all association rules can be decomposed into two subproblems:

1. Find all sets of items (*itemsets*) whose support is greater than the user-specified *minsup*. Itemsets with minimum support are called frequent itemsets.
2. Use the frequent itemsets to generate the desired rules. The general idea is that if, say, $ABCD$ and AB are frequent itemsets, then we can determine if the $AB \Rightarrow CD$ holds by computing the ratio $conf = \text{support}(ABCD)/\text{support}(AB)$. If $conf \geq minconf$, then the rule holds.

Since the second subproblem is straightforward, much of the current research has been focussed on the first subproblem [1, 2, 3, 4].

3. Related Works

Among the many algorithms that have been proposed for discovering frequent itemsets, the Apriori algorithm presented by R.Agrawal and R.Srikant is the most successful and influential [4, 7, 11]. To discover all frequent itemsets, Apriori makes multiple passes over the data. The first pass of the algorithm simply counts item occurrences to determine the frequent 1-itemsets. A subsequent pass, say pass *k*, consists of two phases. First, the frequent itemsets L_{k-1} found in the (*k*-1)th pass are used to generate the candidate itemsets C_k, using the apriori-gen function. Next, the database is scanned and the support of candidates in C_k is counted. At the end of the pass, Apriori determines which of the candidate itemsets are actually frequent. This process continues until no new frequent itemsets are found. The key of efficiency is based on not generating and evaluating those candidate itemsets that can not be frequent. To achieve this goal, all currently known algorithms use a same basic intuition that any subset of a frequent itemset must be frequent [2].

To deal with a kind of incremental updating problem of association rules, We have proposed to partition the frequent 1-itemsets L_1 using a binary partition tree in paper [7]. In fact, when users interactively mine association rules, they may have to continually tune two thresholds, minimum support and minimum confidence, which describe the users' special interestingness. Assume the transaction database does not change, we mainly consider the case that the new minimum support is less than the old one. Our partition idea is based on the following observation:

In the first pass, all the frequent 1-itemsets can be divided into two disjoint parts, all the new frequent 1-itemsets L_1^1 and all the old frequent 1-itemsets L_1^2. And since any subset of a frequent itemset must be frequent too, then for every single item i in a frequent *k*-itemset c, its corresponding frequent 1-itemsets $\{i\}$ is either drawn from L_1^1 or L_1^2. Accordingly in each subsequent pass *k*, we can partition all the frequent *k*-itemsets into three disjoint classes:

1) frequent *k*-itemset $c = \{i_1, i_2, \ldots i_k\}$, $\forall j (1 \leq j \leq k)$, $\{i_j\} \in L_1^1$;

2) frequent *k*-itemset $c = \{i_1, i_2, \ldots, i_k\}$, $\forall j (1 \leq j \leq k)$, $\{i_j\} \in L_1^2$;

3) frequent *k*-itemset $c = \{i_1, i_2, \ldots, i_k\}$, there must be two non-empty subsets c_1 and c_2 that $c_1 \bigcup c_2 = c$, $c_1 \bigcap c_2 = \varnothing$, and $c_1 \subset L_1^1$, $c_2 \subset L_1^2$.

The above partition procedure is at single level. In the final part of that paper, we have shown we can make a generalization by doing further partition at multiple levels. In this paper, Our goal is to use the same basic intuition to solve the mining of association rules itself and discuss the implementation problems of the generalized partition procedure in detail.

4. Algorithm BPA

Given a transaction database D, the support of an itemset can be taken as the number of transactions that contain the itemset. Suppose the *minsup* is s, L_k is the corresponding set of frequent *k*-itemsets, and C_k is the corresponding set of candidate *k*-itemsets. Associated with each itemset is a count field to store the support of this itemset.

4.1. Algorithm BPA

Basically, the framework of BPA is similar to that of Apriori, it needs to make multiple passes over the database too. In the first pass, BPA generates L_1. Then we use a binary tree (called a binary partition tree) to partition L_1, and accordingly, in each subsequent pass k, we employ a ternary tree (called a ternary partition tree) to partition L_k.

Now we describe our partition procedure in detail. Let *maxl* be the maximal level number which we'd like to choose. We use the notation $L_k^{l\cdot n}$ (*l*=1, 2, …, *maxl*) to denote a subset of frequent *k*-itemsets in the *l*th level, which is labeled by *n*. *n* is a sequence of integers in the range of 1 to 3 with a length of *l*. Let L_1 be the root of the binary partition tree and at level 0; its two disjoint sons $L_1^{1\cdot 1}$ and $L_1^{1\cdot 2}$ at level 1, etc. A internal vertex $L_1^{l\cdot n}$ at the *l*th level corresponds to the union of its two disjoint sons $L_1^{(l+1)\cdot n1}$ and $L_1^{(l+1)\cdot n2}$ at the $(l+1)$th level . Likewise, In each subsequent pass *k*, let L_k be the root of the corresponding ternary partition tree and at level 0; its three mutually disjoint sons $L_k^{1\cdot 1}$, $L_k^{1\cdot 2}$ and $L_k^{1\cdot 3}$ at level 1, etc. A internal vertex $L_k^{l\cdot n}$ at the *l*th level corresponds to the union of its three mutually disjoint sons $L_k^{(l+1)\cdot n1}$, $L_k^{(l+1)\cdot n2}$ and $L_k^{(l+1)\cdot n3}$ at the $(l+1)$th level. However, the third son such as $L_k^{1\cdot 3}$ can not be further partitioned, it can only be a leaf vertex. Figure 1 illustrates an example of binary partition tree and ternary partition tree with *maxl*=2.

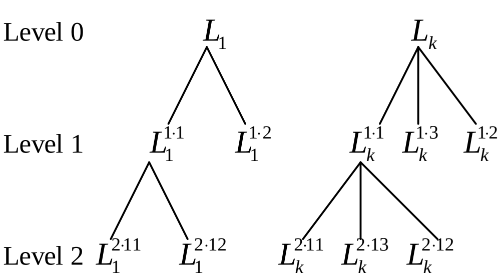

Figure 1 Example of binary partition tree and ternary partition tree

At the same level *l*, $L_1^{l\cdot p1}, L_1^{l\cdot p2}, L_k^{l\cdot p1}, L_k^{l\cdot p2}$ and $L_k^{l\cdot p3}$ (*p* is the common prefix of length $(l-1)$) must satisfy the following conditions which are similar to those of the three disjoint classes mentioned in section 3:

1) $\forall c \in L_k^{l\cdot p1}$, let $c = \{i_1, i_2, \ldots, i_k\}$, $\forall j (1 \le j \le k)$, $\{i_j\} \in L_1^{l\cdot p1}$;

2) $\forall c \in L_k^{l\cdot p2}$, let $c = \{i_1, i_2, \ldots, i_k\}$, $\forall j (1 \le j \le k)$, $\{i_j\} \in L_1^{l\cdot p2}$;

3) $\forall c \in L_k^{l\cdot p3}$, let $c = \{i_1, i_2, \ldots, i_k\}$, there must be two non-empty subsets c_1 and c_2 that $c_1 \cup c_2 = c$, $c_1 \cap c_2 = \varnothing$, and $c_1 \subset L_1^{l\cdot p1}, c_2 \subset L_1^{l\cdot p2}$.

Clearly, based on the binary partition tree, the generation of L_1 can be viewed as a procedure of 2-way merge. And in each subsequent pass *k*, it is a 3-way merge.

Suppose in each subsequent pass *k*, $C_k^{l\cdot n}$ is the corresponding candidate *k*-itemsets of $L_k^{l\cdot n}$, and $C_k^{l\cdot p1}, C_k^{l\cdot p2}$ and $C_k^{l\cdot p3}$ correspond to $L_k^{l\cdot p1}, L_k^{l\cdot p2}$ and $L_k^{l\cdot p3}$ respectively. We use function bpa-gen($L_k^{l\cdot n}$) to generate $C_k^{l\cdot n}$.The parameter $L_k^{l\cdot n}$ corresponds to a vertex in the ternary partition tree. If $L_k^{l\cdot n}$ is a leaf vertex, we simply use the apriori-gen function to generate $C_k^{l\cdot p1}$ and $C_k^{l\cdot p2}$, and to generate $C_k^{l\cdot p3}$, we use iua-gen($L_j^{l\cdot p1}$) presented in [7]. In fact, we can get every candidate *k*-itemset in $C_k^{l\cdot p3}$ by simply concatenating a frequent *j*-itemset $(1 \le j \le k-1)$ in $L_{k-1}^{l\cdot p1}$ and a frequent (*k*-*j*)-itemset in $L_{k-1}^{l\cdot p2}$, assuming neither this pair of itmsets $(L_j^{l\cdot p1}, L_{k-j}^{l\cdot p2})$ is empty. Function iua-gen($L_j^{l\cdot p1}$) takes two steps. First, in the concatenate step, we concatenate $L_j^{l\cdot p1}$ and $L_{k-j}^{l\cdot p2}$:

insert into $C_k^{l\cdot p3}$
select $p.\text{item}_1$, $p.\text{item}_2, \ldots,$ $p.\text{item}_j$, $q.\text{item}_1$, $q.\text{item}_2, \ldots,$ $q.\text{item}_{k-j}$
from $L_j^{l\cdot p1}$ p, $L_{k-j}^{l\cdot p2}$ q;

Next in the prune step, delete all itemsets $c \in C_k^{l\cdot p3}$ such that some (*k*-1)-subset of c is not in $L_{k-1}^{l\cdot p3}$. In the case of $L_k^{l\cdot n}$ is a internal vertex, the generations of $C_k^{l\cdot p1}$ and $C_k^{l\cdot p2}$ are implemented by the 3-way merges of the (*l*+1)th level, and we still use function iua-gen to generate $C_k^{l \cdot p3}$. Using a recursive style, the bpa-gen($L_k^{l \cdot n}$) function is described as follows:

```
bpa-gen( L_k^{l·n} )
{
    if ( L_k^{l·n} is a leaf vertex) then begin
        C_k^{l·p1} =apriori-gen( L_{k-1}^{l·p1} );
        C_k^{l·p2} =apriori-gen( L_{k-1}^{l·p2} );
    end
    else begin   /* L_k^{l·n} is a internal vertex */
        C_k^{l·p1} =bpa-gen( L_k^{(l+1)·n1} );
        C_k^{l·p2} =bpa-gen( L_k^{(l+1)·n2} );
    end
    C_k^{l·p3} =∅;
    for ( j = 1; j ≤ k − 1; j++ ) do begin
        C_k^{l·p3} = C_k^{l·p3} ∪ iua-gen( L_j^{l·p1} );
    end
    C_k^{l·n} = C_k^{l·p1} ∪ C_k^{l·p2} ∪ C_k^{l·p3};
    return ( C_k^{l·n} );
}
```

After generating C_k, the database is scanned and L_k is finally generated. This process continues until no new frequent itemsets are found. Figure 2 gives the primary framework of algorithm BPA:

1) $L_1 = \{\text{frequent 1-itemsets}\}$;
2) for $(k = 2; L_{k-1} \neq \varnothing; k++)$ do begin
3) C_k =bpa-gen(L_k); /* L_k can be viewed as $L_k^{0 \cdot 0}$, here l=0, n=0 */
4) forall transactions $t \in D$ do begin
5) $C_t = \text{subset}(C_k, t)$;
6) forall candidates $c \in C_t$ do
7) $c.count++$;
8) end
9) $L_k = \{c \in C_k \mid c.count \geq s\}$;
10) end
11) Answer= $\bigcup_k L_k$;

Figure 2 Algorithm BPA

4.2. Scale-up and Parallelization

In the candidate generation phase of pass *k*, we need storage for L_{k-1} and C_k. With the increase of the number of transactions, the number of frequent itemsets may greatly increase too. Then we have to do swaps between memory and disks. However, by means of the binary partition and the ternary partitions, we are always able to keep both a subset of C_k and the corresponding subset of L_{k-1} entirely available in memory. This aim can be taken as a basic partition rule. Whenever a subset of L_{k-1} does not entirely fit in memory, we do a partition of that subset, therefore the corresponding leaf vertex in the binary partition tree or the ternary partition tree becomes a internal vertex. Apparently, it is a dynamic partition procedure. Whenever a new partition is done, the binary partition tree in the first pass and the ternary tree in each subsequent pass *k* have to do a vertex split accordingly. For example, as shown in the figure 1, we can first divide L_1 into two disjoint subsets $L_1^{1 \cdot 1}$ and $L_1^{1 \cdot 2}$. If $L_1^{1 \cdot 1}$ and $L_1^{1 \cdot 2}$ can entirely fit in memory respectively, we then suspend the partition. (We can also have another choice, if $C_1^{1 \cdot 1}$ can not entirely fit in memory, we can choose to further partition $L_1^{1 \cdot 1}$.) Suppose in each subsequent pass *k*, whenever a subset of L_{k-1}, say $L_{k-1}^{1 \cdot 1}$ can not entirely fit in memory, we first partition $L_1^{1 \cdot 1}$ into $L_1^{2 \cdot 11}$ and $L_1^{2 \cdot 12}$; then in each subsequent pass k, we partition $L_k^{1 \cdot 1}$ into $L_k^{2 \cdot 11}$ and $L_k^{2 \cdot 12}$ and $L_k^{2 \cdot 13}$.

Based on the binary partition and ternary partitions, BPA can be very easily parallelized assuming a shared-memory architecture. That is algorithm PBPA. Suppose we have *n* processors which have a shard-memory. Based on multiple levels partition, we have a straightforward choice. For example, we can make the ternary partition tree have at least *n* leaf vertices. We then distribute the load equably among the *n* processors. Therefore we can take full advantage of the available processors.

5. Relative Performance

Basically, BPA is an extension of Apriori. Now we compare the relative performance of BPA with that of Apriori. The framework of BPA is similar to that of Apriori, and there is not much extra overhead for the maintenance of the binary partition tree and ternary partition trees .

BPA does better than Apriori in the generating of the candidates *k*-itemsets corresponding to the third child (such as $L_k^{1\cdot 3}$) of a internal vertex. Apriori expands two frequent (k-1)-itemsets to generate candidate k-itemsets using the apriori-gen function, it needs (k-1) join conditions to implement complex join. However, BPA only needs to concatenate two sub-itemsets using the iua-gen function. Furthermore, in the prune step. Suppose $L_k^{l\cdot n}$ is a leaf vertex, When generating $C_k^{l\cdot n}$, BPA can implement efficient prune by simply checking the corresponding $L_{k-1}^{l\cdot n}$. However, Apriori has to check the whole set L_{k-1}. Therefore, suppose a ternary partition tree has m leaf vertices, then Apriori has to check the scope to decide a prune by (m-1) times $|L_{k-1}|$ more than BPA. This can be more significant while single $L_{k-1}^{l\cdot n}$ can entirely fit in memory respectively but the whole of L_{k-1} can not. In fact, at this time Apriori can no longer prune those candidates whose subsets are not in L_{k-1} [3]. Therefore, BPA can efficiently generate less candidates than Apriori. Based on the binary partition, BPA can always gain this advantage.

It is obvious that PBPA not only has all the above virtues of BPA, but also can do efficient parallel computation.

To get detailed relative performance data, related experiments are still under construction.

6. Summary

Based on the binary partition of all the frequent 1-itemsets, this paper presents an efficient mining algorithm BPA. We suppose to use the basic intuition to solve the mining problem of the generalized association rules [5] or other kinds of rules such as sequential patterns.[6]

References

[1] R.Agrawal, et al. Mining association rules between sets of items in large databases. In Proc. of ACM SIGMOD'93, P207-216, Washington, D.C., May 1993.

[2] R.Agrawal and R.Srikant. Fast algorithms for mining association rules. In VLDB'94, P487-499, Santiago, Chile, September 1994.

[3] R.Agrawal and R.Srikant. Fast algorithms for mining association rules. In IBM Research Report, 1994.

[4] R.Agrawal and J.C.Shafer. Parallel mining of association rules: Design, implementation, and experience. In IBM Research Report, 1996.

[5] R.Srikant and R.Agrawal. Mining generalized association rules. In VLDB'95, P407-419, Zurich, Switzerland, September 1994.

[6] R.Agrawal and R.Srikant. Mining sequential patterns. In Proc. 1995 Int'l Conf. on Data Engineering, P3-14, Taipei, Taiwan, March 1995.

[7] Jianlin Feng, Yucai Feng. Incremental Updating Algorithms for Mining Association Rules. In 1997 8th Int'l Conf. on Management of Data(COMAD'97) , Chennai(Madras), India. Dec.1997.

[8] J.S.Park, et al. An effective hash based algorithm for mining of association rules. In Proc of ACM SIGMOD'95, P175-186, San Jose, California, May 1995.

[9] A.Savasere, et al. An efficient algorithms for mining association rules in large databases. In VLDB'95, P432-444, Zurich, Switzerland, September 1995.

[10] M.Houtsma and A.Swami. Set-oriented mining association rules. In Proc. 1995 Int'l Conf. on Data Engineering, P25-33, Taipei, Taiwan, March 1995.

[11] H.Toivonen. Sampling large databases for association rules. In VLDB'95, P1-12, Bombay, India,1996.