New Algorithms for Efficient Mining of Association Rules*

Li Shen, Hong Shen and Ling Cheng
School of Computing and Information Technology
Griffith University, Nathan, QLD4111, Australia

Abstract

Discovery of association rules is an important data mining task. Several algorithms have been proposed to solve this problem. Most of them require repeated passes over the database, which incurs huge I/O overhead and high synchronization expense in parallel cases. There are a few algorithms trying to reduce these costs. But they contains weaknesses such as often requiring high pre-processing cost to get a vertical database layout, containing much redundant computation in parallel cases, and so on. We propose new association mining algorithms to overcome the above drawbacks, through minimizing the I/O cost and effectively controlling the computation cost. Experiments on well-known synthetic data show that our algorithms consistently outperform *Apriori*, one of the best algorithms for association mining, by factors ranging from 2 to 4 in most cases. Also, our algorithms are very easy to be parallelized, and we present a parallelization for them based on a shared-nothing architecture. We observe that the parallelism in our parallel approach is developed more sufficiently than in two of the best existing parallel algorithms.

Keywords: data mining, association rule, frequent itemset, parallel processing.

1 Introduction

Discovery of association rules is an important problem in the area of data mining. The problem is introduced in [2], and can be formalized as follows. Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of literals, called *items*. Let $\mathcal{D}$ be a collection of transactions, where each transaction d has a unique identifier and contains a set of items such that $d \subseteq \mathcal{I}$. A set of items is called an *itemset*, and an itemset with k items is called a *k-itemset*. The *support* of an itemset x in $\mathcal{D}$, denoted as $\sigma(x/\mathcal{D})$, is the ratio of the number of transactions (in $\mathcal{D}$) containing x to the total number of transactions in $\mathcal{D}$. An *association rule* is an expression $x \Rightarrow y$, where $x, y \subseteq \mathcal{I}$ and $x \cap y = \emptyset$. The *confidence* of $x \Rightarrow y$ is the ratio of $\sigma(x \cup y/\mathcal{D})$ to $\sigma(x/\mathcal{D})$. We use minsup and minconf to denote the user-specified minimum support and confidence respectively. An itemset x is *frequent* if $\sigma(x/\mathcal{D}) \geq$ minsup. An association rule $x \Rightarrow y$ is *strong* if $x \cup y$ is frequent and $\frac{\sigma(x \cup y/\mathcal{D})}{\sigma(x/\mathcal{D})} \geq$ minconf. The problem of *mining association rules* is to find all strong association rules.

*This work was partially supported by ARC Large Research Grant (1996-98) A849602031.

Since it is easy to generate all strong association rules from all frequent itemsets, almost all current studies for association discovery concentrate on *how to find all frequent itemsets efficiently*. Our paper will also focus on this key problem. Several algorithms [1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12] have been proposed in the literature to solve this problem. However, most of them are based on the *Apriori* approach [1] and require repeated passes over the database to determine the set of frequent itemsets, thus incurring high I/O overhead. In the parallel case, most algorithms perform a sum-reduction at the end of each pass to construct the global counts, also incurring high synchronization cost.

Some algorithms [11, 12] have been introduced to overcome the above drawbacks. However, they contains other weaknesses. First, they use a vertical database layout and assume that it can be available immediately. But in practice, a transaction database usually have a horizontal layout. Thus, to get a vertical layout, they need to build a new database with the same size. Clearly, this pre-processing step costs too much for a very large database and may increase their costs greatly. Second, they attach a tid-list on each candidate (potentially frequent itemset) in order to compute supports for its supersets. But, for a large database and a large amount of candidates, the space cost for storing tid-lists of all candidates is very huge and might not fit in main memory. Third, in parallel cases, they generates clusters of candidates and then partition these clusters into different processors. However, lots of these clusters are overlapping and many candidates may occur in multiple and different clusters. Thus, on one hand, these candidates will be counted for getting their supports at multiple processors so that much redundant computation is produced; on the other hand, to avoid synchronization cost, a large part of database has to be copied to multiple processors, which is not effective in both time and space costs. In short, parallelism can not be fully developed in these algorithms.

In this paper, we propose new effective association mining algorithms to overcome all the above drawbacks. First, they are more efficient than *Apriori* by reducing I/O cost greatly and controlling computation cost effectively. Second, they are based on the standard horizontal layout database and do not have any pre-processing step. Third, they are very suitable for parallelization and parallelism is easy to be fully developed for them. We present a framework of our algorithms in Section 2; describe our algorithms in Section 3; study their performances experimentally in Section 4; propose a parallelization for them in Section 5; and conclude the paper in Section 6.

2 A Framework

We use L to denote the set of all frequent itemsets, and L_k to denote the set of all frequent k-itemsets. As mentioned before, the key problem of association discovery is to generate L. Almost all current algorithms require to compute supports for a set of candidates C, and L is formed by selecting all itemsets with support exceeding minsup from C. Among them, *Apriori* [1] is one of the best because it minimizes the size of C: only those itemsets whose subsets are all frequent are included in C. However, to realize that, it has to employ a levelwise approach: at each level k, it needs a scan over data to count support for each $c \in C_k$, where C_k is a superset of L_k; next it generates L_k from C_k and then use L_k generate C_{k+1}; after that, it can start to work for the next level $k+1$. Clearly, this approach contains high I/O costs and is not suitable for parallelization.

To obtain a low I/O expense and a high parallelism, a straightforward idea is to design a method as follows: generate a candidate set C ($\supseteq L$) at one or two times instead of uncertainly multiple times like *Apriori*. If we can find such a method, we only need to use one or two scans over data to obtain all necessary supports and then generate L immediately. Clearly, this method has a high parallelism because we can partition not only the candidate set C but also the database into multiple processors to do support-computation in parallel. It is very easy to see that such a method does exist. The simplest one is to take all itemsets (i.e., all subsets of $\mathcal{I}$) as candidates. Clearly, this naive method is extremely inefficient due to the exponential size $2^{|\mathcal{I}|}$ of the candidate set C. To design efficient methods, we need to control the size of C.

Let c be an itemset. We use $\mathcal{P}_k(c)$ to denote the set of all k-subsets of c. Now we propose a faster method as follows: first find L_1 and L_2; next use L_2 to generate $C_{all} = \{c \subseteq \mathcal{I} \mid \mathcal{P}_2(c) \subseteq L_2, |c| \geq 3\}$, i.e., all k-itemsets ($k \geq 3$) whose 2-subsets are all frequent are regarded as candidates; then count supports for all $c \in C_{all}$; finally generate all frequent k-itemsets ($k \geq 3$) by checking supports for all $c \in C_{all}$. Actually, this approach is very similar to *ClusterApr* algorithm proposed in [12]. The only difference is that *ClusterApr* uses a vertical database layout.

We assume that c is the maximal candidate (in size) in C_{all}. Thus, the size of C_{all} is at least in the order of $O(2^{|c|})$, as all k-subsets ($k \geq 3$) of c must also be in C_{all}. Hence, on the condition that $|c|$ is small, the above method runs efficiently because of the limited computation cost and the minimized I/O cost. However, when $|c|$ becomes large, the computation cost increases greatly due to the huge size of C_{all} so that the algorithm might break down.

To avoid this exponential bottle neck, now we propose a framework which is based on the above method but counts support for only a small part of candidates in C_{all} when the size of C_{all} is huge. In practice, it is not wise to generate all candidates in C_{all}, as it requires not only lots of time but also huge space. An alternative method is to generate and store those maximal candidates called *item cliques*. An itemset $c \in C_{all}$ is an *item clique* if any superset of c is not in C_{all}. We always use Q to denote the set of all item cliques, Q_i to denote the set of all item cliques with a fixed size i, maxcliqsize to denote the size of the maximal item clique. Clearly, we have $Q = \bigcup_{i=3}^{\text{maxcliqsize}} Q_i$, and $C_{all} = \{c \mid c \subseteq cliq, cliq \in Q, |c| \geq 3\}$.

An *Complete Partition of Items* (CPI in short) is defined as a list of non-overlapping itemsets such that the union of them contains all items. Let $\mathcal{I}_0 \sim \mathcal{I}_{n-1}$ be a CPI. By definition, we have (1) $\mathcal{I}_i \cap \mathcal{I}_j = \emptyset$ for $i \neq j$ and $i, j \in \{0 \sim (n-1)\}$, and (2) $\bigcup_{i=0}^{n-1} \mathcal{I}_i = \mathcal{I}$. Furthermore, we say that this CPI has *size* n and *item clusters* (*clusters* in short) $\mathcal{I}_0 \sim \mathcal{I}_{n-1}$. An itemset c is called an *intra-CPI* itemset if c is a subset of any of $\mathcal{I}_0 \sim \mathcal{I}_{n-1}$; otherwise, c is an *inter-CPI* itemset.

The brief idea of our method is as follows. First, we generate L_1, L_2 and Q, and then use Q to produce an effective CPI. Second, we count supports only for all intra-CPI candidates in C_{all}, and then all frequent intra-CPI itemsets can be obtained. We use C_{intra} to denote the set of all those intra-CPI candidates, i.e., $C_{intra} = \{c \mid c \subseteq cliq, cliq \in Q, |c| \geq 3, c \text{ is intra-CPI }\}$. Third, we build a set of inter-CPI candidates $C_{inter} = \{c \mid c \subseteq cliq, cliq \in Q, |c| \geq 3, c \text{ is inter-CPI}, \text{all intra-CPI subsets of } c \text{ are frequent}\}$. It is easy to see that $C_{inter} \subseteq C_{all} \backslash C_{intra}$ contains all frequent inter-CPI k-itemsets for $k \geq 3$ because any subset of an frequent itemset must also be frequent. Therefore, finally, we can count supports for all $c \in C_{inter}$, and all frequent inter-CPI itemsets can be obtained.

Clearly, this method is not only correct but also more efficient because of $C_{intra} \cup C_{inter} \subseteq C_{all}$. Since $|C_{all}|$ is exponential (in the size of the maximal item clique), we should try to generate an effective CPI to keep both sizes of C_{intra} and C_{inter} under control. We introduce a threshold maxcansize for controlling the size of C_{intra}. We say that a candidate $c \in C_{all}$ is an *oversize* candidate if the size of c is greater than maxcansize. By setting maxcansize, we try to generate a CPI such that C_{intra} contain as few oversize candidates as possible. Since a large maxcansize may imply a very huge C_{intra} and a small one may produce a very huge C_{inter}, in practice, we often choose a medium-sized maxcansize (e.g., 8, 9 or 10) to keep both $|C_{intra}|$ and $|C_{inter}|$ reasonable. Based on the premise that $|C_{intra}|$ has been controlled not to be extremely large by using maxcansize, we should let C_{intra} contains as many candidates as possible to reduce the size of C_{inter} greatly. To realize that, we can not only keep the CPI size as small as possible but also let each item clique be related to as few clusters as possible. To sum up, we conclude the framework of our method by four phases as follows.

- **Initialization**: Generate all frequent 1-itemsets and 2-itemsets. Use all frequent 2-itemsets to generate all item cliques (potential maximal frequent itemsets).

- **CPI Generation**: Use all item cliques and a prespecified maxcansize to generate a CPI, based on the following principles: (1) let C_{intra} contain as few oversize candidates as possible; (2) keep the CPI size as small as possible; (3) let each item clique be related to as few clusters as possible.

- **Intra-CPI Mining**: Use all item cliques to generate C_{intra}. Count supports for all $c \in C_{intra}$ and generate all frequent intra-CPI itemsets.

- **Inter-CPI Mining**: Use all item cliques and frequent intra-CPI itemsets to get C_{inter}. Count supports for all $c \in C_{inter}$ and get all frequent inter-CPI itemsets.

3 Algorithms

In this section, we first design algorithms for completing the above four phases respectively and then describe our algorithms for mining frequent itemsets. For simplicity, we always assume that $\mathcal{I} = \{0, 1, \cdots, N-1\}$, and use the form of $\langle x_0, x_1, \cdots, x_{n-1}\rangle$ to denote an itemset c such that $c = \{x_0, x_1, \cdots, x_{n-1}\}$ and $x_0 < x_1 < \cdots < x_{n-1}$.

3.1 Initialization

In this phase, we need to first generate L_1 and L_2, and then use L_2 to generate all item cliques. Similar to [5], we use a one-dimensional array and a two-dimensional array to speed up the process of obtaining L_1 and L_2. This simple process runs very fast, since no searching is needed. After obtaining L_2, we can use a method presented in [12] to generate all item cliques. To give a brief description, we first introduce some useful concepts. Let x be a frequent item. The *equivalence class* of x, denoted by $[x]$, is defined as $[x] = \{y \mid \langle x, y\rangle \in L_2\}$. The *covering set* of $[x]$, denoted by $[x].cvset$, is defined as $[x].cvset = \{y \in [x] \mid [x] \cap ([y] \backslash [z]) \neq \emptyset \text{ for any } z \in [x]\}$. The algorithm is as follows, please refer to [12, 7] for more details.

Algorithm 3.1 *The item clique generation algorithm: GEN_CLQ*
Input: *(1)* $\mathcal{I} = \{0 \sim (N-1)\}$, *the set of all items; (2)* L_2, *the set of all frequent* 2*-itemsets.* **Output**: $Q_3 \sim Q_{\mathsf{maxcliqsize}}$, *where* maxcliqsize *is the size of the maximal item clique and* Q_i *is the set of all item cliques with size* i.

Algorithm GEN_CLQ
  for each $x \in \mathcal{I}$ **do** $[x] = \{y \mid \langle x, y\rangle \in L_2\}$;
  for each $x \in \mathcal{I}$ **do** $[x].cvset = \{y \in [x] \mid [x] \cap ([y] \backslash [z]) \neq \emptyset$ for any $z \in [x]\}$;
  for $(x = N-1; x \geq 0; x--)$ **do**
// *Generate all item cliques*
    $[x].cliqlist = \{\langle x\rangle\}$;
    for each $y \in [x].cvset$ **do**
      for each $cliq \in [y].cliqlist$ **do**
        if $cliq = [x]$ **then**
          mark $cliq$ with *remove*;
        if $cliq \cap [x] \neq \emptyset$ **then**
          insert $(\{x\} \cup (cliq \cap [x]))$
          into $[x].clqlist$
          such that $\nexists A, B \in [x].clqlist, A \subseteq B$;
  for each $x \in \mathcal{I}$ **do**
// *Put all item cliques in order*
    for each $cliq \in [x].cliqlist$ without mark *remove* **do**
      insert $cliq$ into Q_i, where i is the size of $cliq$;
  maxcliqsize = the size of the maximal item clique;

3.2 CPI Generation

In this phase, we need to use all item cliques and a pre-specified maxcansize to generate a CPI $\mathcal{I}_0 \sim \mathcal{I}_{n-1}$, based on the following principles: (1) let C_{intra} contain as few oversize candidates as possible; (2) keep the CPI size n as small as possible; (3) let each item clique be related to as few clusters as possible. Clearly, if maxcansize $\geq$ maxcliqsize, we can easily put all items into one cluster such that the generated CPI contains only one cluster $\mathcal{I}$. In this case, all above principles have been satisfied. Now we consider the opposite case, maxcansize $<$ maxcliqsize.

We use cpisize to denote the size of the desired CPI. Let $cliq$ be the maximal item clique. Based on the first principle, we hope that all intra-CPI subsets of $cliq$ have sizes no greater than maxcansize. To meet this requirement, we should have at least $\frac{\mathsf{maxcliqsize}-1}{\mathsf{maxcansize}} + 1$ clusters in the CPI because each cluster can contain at most maxcansize items in $cliq$. To satisfy the second principle, we simply set $\mathsf{cpisize} = \frac{\mathsf{maxcliqsize}-1}{\mathsf{maxcansize}} + 1$ in our method.

Let x be an item. We use $x.cls \in \{0 \sim (\mathsf{cpisize}-1)\}$ to indicate that x is an item in cluster $\mathcal{I}_{x.cls}$. We use $\mathcal{I}^Q$ to denote the set of all items occurring in Q. Thus, our main task is to generate $x.cls$ for each $x \in \mathcal{I}$. After initially setting all $x.cls = unknown$, we can use a scan of Q to assign a value to $x.cls$ for each $x \in \mathcal{I}^Q$. Certainly, given an item clique $cliq \in Q$, we only need to generate $x.cls$ for each $x \in cliq$ with $x.cls = unknown$. We introduce such a procedure as follows.

Procedure assigncluster(cls: an cluster,
    $cliq$: an item clique)
  $A_{unknown} = \{x \in cliq \mid x.cls = unknown\}$;
  if $A_0 = \emptyset$ **then return**; // *Each item* $x \in cliq$ *has been put in some cluster*
  for $(i = 0; i < \mathsf{cpisize}; i++)$ **do**
    $A_i = \{x \in cliq \mid x.cls = i\}$;
    Choose k from $0 \sim \mathsf{cpisize} - 1$ such that $|A_k| = max\{|A_0| \sim |A_{\mathsf{cpisize}-1}|\}$;
    if $|A_{unknown}| < |A_k|$ **then** $cls = k$;
    for each $x \in A_{unknown}$ **do** $x.cls = cls$;

In this procedure, we first partition the given $cliq$ into $A_{unknown}, A_0, \cdots, A_{\mathsf{cpisize}-1}$ based on $A_i = \{x \in cliq \mid x.cls = i\}$. Let A_k be the maximal partition (in size) among $A_0 \sim A_{\mathsf{cpisize}-1}$. If $|A_{unknown}| \geq |A_k|$, we simply set $x.cls = cls$ for each $x \in A_{unknown}$, where cls is an input parameter value. However, when $|A_{unknown}| < |A_k|$, it is easy to see that there are quite a few items in $cliq$ belonging to cluster $\mathcal{I}_k$ and the size of $A_{unknown}$ is rather small. In this case, we put all elements of $A_{unknown}$ into cluster $\mathcal{I}_k$ to maximize the number of inter-CPI subsets of $cliq$ and let $cliq$ be related to as few clusters as possible (the third principle).

Now, by a scan of Q and calling assigncluster(cls, $cliq$) for each $cliq \in Q$, we can generate $x.cls$ for all $x \in \mathcal{I}^Q$. We let cls have an initial value 0. During the scan over Q, if we call assigncluster($cliq_1$, cls) to process $cliq_1$, we always call assigncluster($cliq_2$, $(cls + 1)$ mod cpisize) to process the next clique $cliq_2$, where mod refers to the module operator. It is easy to see that our scan of Q should not start from any item

clique with size greater than maxcansize, because otherwise oversize intra-CPI candidates will be generated immediately. Our strategy is to scan Q in the following order:

$$Q_{\mathsf{maxcansize}} \rightarrow Q_{\mathsf{maxcansize}-1} \rightarrow \cdots \rightarrow Q_3 \rightarrow Q_{\mathsf{maxcansize}+1} \rightarrow \cdots \rightarrow Q_{\mathsf{maxcliqsize}},$$

where item cliques within each Q_i for all $i \in \{3 \sim \mathsf{maxcliqsize}\}$ are kept in lexicographic order. Clearly, after finishing scans for $Q_3 \sim Q_{\mathsf{maxcansize}}$, most items in $\mathcal{I}^Q$ usually have been assigned into some clusters. Thus, the following scans for $Q_{\mathsf{maxcansize}+1} \sim Q_{\mathsf{maxcliqsize}}$ do not tend to generate oversize intra-CPI candidates in the CPI. In addition, by starting from $Q_{\mathsf{maxcansize}}$, we try to let each item clique be related to as few cluster as possible to meet the third principle mentioned before.

After obtaining $x.cls$ for all $x \in \mathcal{I}^Q$, we propose two methods adjustcluster(I) and adjustcluster(II) to adjust these $x.cls$'s and try to make C_{inter} contain as few oversize candidates as possible. We will describe these methods later. When we finish all the above work, it is easy to see that we have $\mathcal{I} \backslash \mathcal{I}^Q = \{x \in \mathcal{I} \mid x.cls = unknown\}$. Note that all these items in $\mathcal{I} \backslash \mathcal{I}^Q$ are *uninteresting* for us, because they won't occur in any intra-CPI or inter-CPI candidate that concerns us. With this observation, we simply set $x.cls = 0$ for all $x \in \mathcal{I} \backslash \mathcal{I}^Q$, and thus all $\mathcal{I}_0 \sim \mathcal{I}_{\mathsf{cpisize}-1}$ can be obtained by scanning $x.cls$ for each $x \in \mathcal{I}$. Now we can present the description of our algorithms for CPI generation as follows, where GEN_CPI(X) denote the algorithm that calls adjustcluster(X) to do cluster adjustment and X may be either I or II.

Algorithm 3.2 *The CPI generation algorithms: GEN_CPI(I) and GEN_CPI(II).*

Input: *(1)* $Q = \bigcup_{i=3}^{\mathsf{maxcliqsize}} Q_i$, *the set of all item cliques; (2)* maxcansize, *a pre-specified threshold for controlling the size of* C_{intra}. **Output**: $\mathcal{I}_0 \sim \mathcal{I}_{\mathsf{cpisize}-1}$, *a CPI.*

Algorithm GEN_CPI(X) // $X \in$ {*I, II*}
  for each $x \in \mathcal{I}$ **do** $x.cls = unknown$;
  $cls = 0$; cpisize $= \frac{\mathsf{maxcliqsize}-1}{\mathsf{maxcansize}} + 1$;
  for ($i =$ maxcansize; $i \geq 3$; $i--$) **do**
    for each $cliq \in Q_i$ **do** assigncluster(cls, $cliq$) and $cls = (cls + 1)$ mod cpisize;
  for ($i =$maxcansize+1; $i \leq$maxcliqsize; $i++$) **do**
    for each $cliq \in Q_i$ **do** assigncluster(cls, $cliq$) and $cls = (cls + 1)$ mod cpisize;
  if X=I **then** adjustcluster(I)
  else adjustcluster(II);
  for ($i = 0; i <$ cpisize; $i++$) **do** $\mathcal{I}_i = \emptyset$;
  for each $x \in \mathcal{I}$ **do**
    if $x.cls = unknown$ **then** $\mathcal{I}_0 = \mathcal{I}_0 \cup \{x\}$
    else $\mathcal{I}_{x.cls} = \mathcal{I}_{x.cls} \cup \{x\}$;

Procedure adjustcluster(X) // $X \in$ {*I, II*}
  for each $x \in \mathcal{I}$ **do** $x.adj = 0$;
  for each $cliq \in Q$ **do**
    for each A satisfying A is a maximal oversize intra-CPI subsets of $cliq$ **do**
  $A_0 = \{x_1 \sim x_n\} = \{x \in A \mid x.adj = 0\}$ and $A_1 = \{x \in A \mid x.adj = 1\}$;
      for ($i = 1; i \leq n; i++$) **do**
        if ($|A_1| + i \leq$ maxcansize) **then** $x_i.adj = 1$ **else** $x_i.adj = 2$;
  if X=II **then** cpisize++;
  for each $x \in \mathcal{I}$ with $x.adj = 2$ **do**
    if X=II **then** $x.cls =$ cpisize $- 1$
    else $x.cls = (x.cls + 1)$ mod cpisize;

It is easy to see that the main procedure of our algorithm GEN_CPI(X) employs the idea mentioned before. Now we explain the adjustcluster(X) procedure, where X may be I or II. We initially set $x.adj = 0$ for all $x \in \mathcal{I}$, which means x has not been found in any oversize intra-CPI candidate. Then, we use a scan over Q to process each maximal oversize intra-CPI candidate A. For each A, we will set either $x.adj = 1$ or $x.adj = 2$ for all $x \in A$ with $x.adj = 0$, with the goal of making the size of A_1 as close to maxcansize as possible, where $A_1 = \{x \in A \mid x.adj = 1\}$. After completing the processing for all those A's, we move all $x \in \mathcal{I}$ with $x.adj = 2$ to a new cluster. If X=I, x will be moved to the next cluster by setting $x.cls = (x.cls+1)$ mod cpisize. If X=II, cpisize will be increased by 1 and then x will be moved to the new cluster $\mathcal{I}_{\mathsf{cpisize}-1}$. The difference of these two methods is as follows: the first one tries to satisfy the third principle, while the second one tries to satisfy the first principle. It is easy to see that any old oversize intra-CPI candidate can be broken into two smaller intra-CPI candidates after this adjustment.

3.3 Intra-CPI Mining

In this phase, we will first use the obtained CPI and all item cliques to generate C_{intra}; then count supports for all candidates in C_{intra}; and finally generate all frequent intra-CPI subsets. First, we study the data structure for storing candidates. The *hash-tree* structure introduced in [1] can only be used to count supports for a set of candidates with the same size, and so is not suitable for our problem. Here, we use a *prefix-tree* T to store a set of candidates with different sizes. Let r be the root of T. Each node $t \in T$ contains four fields: $t.item$, $t.cn$, $t.son[1 \sim t.cn]$ and $t.sup$. We use $t.item$ to store an item, except that $r.item$ stores nothing. We use $t.cn$ to store the number of children of t, and use $t.son[1 \sim t.cn]$ to store the addresses of all its children. We always maintain the following property: $t.item < t.son[1].item < \cdots < t.son[t.cn].item$. Furthermore, each $t_n \in T$ corresponds to a unique candidate, denoted by $t_n.can$, such that $t_n.can = \langle t_1.item, \cdots, t_n.item \rangle$ and $(r, t_1, \cdots, t_n)$ forms exactly a path (or prefix) from root r to the current node t_n. We use $t.sup$ store the support of $t.can$.

Constructing and updating such a prefix-tree is easy, and so we simply use *instree(T, c)* to denote the process of inserting an candidate c into a prefix-tree T. Note that when we insert a candidate $\langle x_1, \cdots, x_n \rangle$ into T, any its prefix $\langle x_1, \cdots, x_m \rangle$ for $m \leq n$ will also be regarded as a candidate in T. For example, Figure 1 shows a prefix-tree generated by the following candidate set: C_{exam} ={ $\langle 1,2,5 \rangle$, $\langle 1,2,6,7 \rangle$, $\langle 1,2,6,7,9 \rangle$, $\langle 1,2,6,8 \rangle$, $\langle 1,4,9 \rangle$, $\langle 1,8,9 \rangle$, $\langle 2,5,8 \rangle$, $\langle 3,5,9 \rangle$, $\langle 3,6,7 \rangle$, $\langle 3,6,7,8,9 \rangle$, $\langle 3,6,7,9 \rangle$, $\langle 3,7,9 \rangle$ }. Note that $\langle 3,6,7,8 \rangle$ is regarded as a candidate in the tree, though it is not in C_{exam}. Now we can present the following procedure to generate supports for all candidates stored in

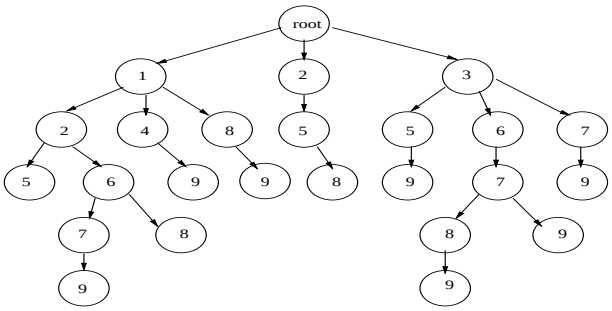

Figure 1: An example of a prefix-tree

prefix-tree T.

Procedure CountSup(T: a prefix-tree, $\mathcal{D}$: a transaction database)
Set $t.sup = 0$ for each $t \in T$ and let r be the root of T;
for each transaction $d \in \mathcal{D}$ **do**
Let $\langle x_1 \sim x_n \rangle$ be the itemset containing exactly all frequent items in d;
IncSup(r, $\langle x_1 \sim x_n \rangle$);

Procedure IncSup(t: a prefix-tree node, $\langle x_1 \sim x_n \rangle$: an itemset)
$t.sup++$; **if** $n = 0$ **then return**;
if $\exists i \in \{1 \sim t.cn\}$, $t.son[i].item = x_1$ **then**
IncSup($t.son[i]$, $\langle x_2 \sim x_n \rangle$);
for $(i = 2; i \leq n; i++)$ **do**
IncSup(t,$\langle x_i \sim x_n \rangle$);

Based on the definition of the prefix-tree, it is easy to see the correctness of the above method. Now we can present our algorithm for Intra-CPI mining as follows.

Algorithm 3.3 *The Intra-CPI mining algorithm: INTRA_MINER.*
Input: *(1) Q, the set of all item cliques; (2) $\mathcal{I}_0 \sim \mathcal{I}_{\mathsf{cpisize}-1}$, a CPI; (3) L_1 and L_2, the sets of all frequent 1-itemsets and 2-itemsets; (4)* minsup, *a support threshold; (5) $\mathcal{D}$, a transaction database.* **Output**: *L_{intra}, the set of all frequent intra-CPI itemsets.*

Algorithm INTRA_MINER
$C_{intra} = \{c \mid c \subseteq cliq, cliq \in Q,$
$|c| \geq 3, c$ is intra-CPI$\}$;
Let T be an empty prefix-tree and
for each $c \in C_{intra}$ **do** instree(T, c);
CountSup($T, \mathcal{D}$);
Collect all frequent intra-CPI k-itemsets
for $k \geq 3$ into L_{intra} by a scan of T;
$L_{intra} = L_{intra} \cup L_1 \cup \{x \in L_2 \mid$
x is intra-CPI$\}$;

This algorithm is easy to understand. First we use Q to generate C_{intra}. Next, all candidates in C_{intra} are inserted into a prefix-tree T. Then, we call CountSup($T, \mathcal{D}$) to count supports for all candidates in T. Finally, we can generate L_{intra} by a scan of T, L_1 and L_2. The correctness of the algorithm is straightforward.

3.4 Inter-CPI Mining

In this phase, we first use all item cliques and all frequent intra-CPI itemsets to generate C_{inter}, next count supports for all $c \in C^{intra}_{suball}$ and then generate all frequent inter-CPI k-subsets for $k \geq 3$. The algorithm is as follows.

Algorithm 3.4 *The Inter-CPI mining algorithm: INTER_MINER.*
Input: *(1) Q, the set of all item cliques; (2) $\mathcal{I}_0 \sim \mathcal{I}_{\mathsf{cpisize}-1}$, a CPI; (3) L_{intra}, the set of all frequent intra-CPI itemsets; (4) L_2, the set of all frequent 2-itemsets; (5)* minsup, *a support threshold; (6) $\mathcal{D}$, a transaction database.* **Output**: *L_{inter}, the set of all frequent inter-CPI itemsets.*

Algorithm INTER_MINER
$C_{inter} = \{c \mid c \subseteq cliq, cliq \in Q, |c| \geq 3,$
c is inter-CPI, all intra-CPI subsets
of c are in $L_{intra}\}$;
Let T be an empty prefix-tree and
for each $cliq \in Q$ **do** instree(T, c);
CountSup($T, \mathcal{D}$);
Collect all frequent inter-CPI k-itemsets
for $k \geq 3$ into L_{inter} by a scan of T;
$L_{inter} = L_{inter} \cup \{x \in L_2 \mid x$ is inter-CPI$\}$;

This algorithm is easy to understand. First we use Q and L_{intra} to generate C_{intra}. Next, all candidates in C_{inter} are inserted into a prefix-tree T. Then, we call CountSup($T, \mathcal{D}$) to count supports for all candidates in T. Finally, we can generate L_{inter} by a scan of T and L_2. The correctness of the algorithm is straightforward.

3.5 Algorithm Description

Now we can present our algorithms for finding all frequent itemsets as follows.

Algorithm 3.5 *The frequent itemset mining algorithms: MINER(I) and MINER(II).*
Input: *(1) $\mathcal{I}$, the set of all items; (2) $\mathcal{D}$, a transaction database; (3)* minsup, *a support threshold; (4)* maxcansize, *a pre-specified threshold for controlling the size of C_{intra}.* **Output**: *L, the set of all frequent itemsets.*

Algorithm MINER(X) // $X \in \{I, II\}$
Use array technique to generate L_1
and L_2;
GEN_CLQ; // Get $Q = \bigcup_{i=3}^{\mathsf{maxcliqsize}} Q_i$
if maxcliqsize $\leq$ maxcansize **then**
$C_{all} = \{c \mid c \subseteq cliq, cliq \in Q, |c| \geq 3\}$;
Let T be an empty prefix-tree and **for**
each $c \in C_{all}$ **do** instree(T, c);
CountSup($T, \mathcal{D}$);
Collect all frequent k-itemsets for $k \geq 3$
into L by a scan of T;
$L = L \cup L_1 \cup L_2$;
else
GEN_CPI(X); // *CPI generation*
INTRA_MINER; // *L_{intra} generation*
INTER_MINER; // *L_{inter} generation*
$L = L_{intra} \cup L_{inter}$;

This algorithm is also easy to understand. First, we use array technique to generate L_1 and L_2. Next, we call GEN_CLQ to generate all item cliques. If maxcliqsize $\leq$ maxcansize, we know that the size of C_{all} is

limited. Hence, we use a prefix-tree T to store all candidates in C_{all} and then count supports for them. After that L can be obtained immediately. If maxcliqsize $>$ maxcansize, we call GEN_CPI(X) to do CPI generation, call INTRA_MINER to obtain L_{intra}, call INTER_MINER to generate L_{inter}, and finally we have $L = L_{intra} \cup L_{inter}$. The correctness of the algorithm is straightforward.

4 Performance Study

We have implemented our algorithms *MINER(I)* and *MINER(II)* for evaluating their perfermances. In both implementations, we set maxcansize = 10. For comparison, we have also implemented two *Apriori* versions. One version denoted by *AprиHash* uses *hash-tree* structure suggested by [1] to store candidate sets. The other version denoted by *AprиPref* uses our *prefix-tree* structure to store candidate sets. To assess the performances of *MINER(I)*, *MINER(II)*, *AprиHash* and *AprиPref*, we performed several experiments on a SUN ULTRA-1 workstation with 64 MB main memory running Sun OS 5.5. To keep the comparison fair, we implemented all the algorithms using the same basic data structures, except that *AprиHash* used a different *hash-tree* structure to store candidate sets. In addition, for a fair comparison, all our test results do not contain the execution time for generating L_1 and L_2, because *Apriori* uses a much slower method to get L_1 and L_2 than our array technique.

The synthetic datasets used in our experiments were generated by a tool described in [1]. We use the same notation $Tx.Iy.Dz$ to denote a dataset in which x is the average transaction size, y is the average size of a maximal potentially frequent itemset and z is the number of transactions. In addition, we generated all datasets by setting $N = 1000$ and $|L| = 2000$, where N is the number of all items, $|L|$ is the number of maximal potentially frequent itemsets. Please refer to [1] for more details on the dataset generation.

Figure 2 shows the experimental results for four synthetic datasets to compare the performances of these algorithms. We observe that both our algorithms *MINER(I)* and *MINER(II)* outperform both *AprиHash* and *AprиPref*, by factors ranging from 2 to 4 in most cases. However, the difference between *MINER(I)* and *MINER(II)* is minor.

In theory, there are three reasons for that our methods outperform the *Apriori* approach. First, our methods only need at most 2 scans of databases for obtaining all frequent itemsets with size greater than 2, while *Apriori* requires uncertainly multiple scans. Hence, the I/O cost in our methods is usually much smaller than that in *Apriori*. Second, we design effective methods to make the size of our candidate set reasonable so that the computation cost is also well-controlled. Third, using prefix-tree structure to store candidates also has two advantages: (1) its space requirement is very low; (2) the support-increment operations can be made at each visited node to reduce the computation cost, while these operations can only be made at leaf nodes when *hash-tree* is used. To sum up, with these advantages, our algorithms are more efficient than *Apriori*.

5 Parallelization

Since mining frequent itemsets requires lots of computation power, memory and disk I/O, it is not only interesting but also necessary to develop parallel algorithms for this data mining task. Here, we study how to extend our algorithm MINER(X) for parallelization, where X $\in$ {I, II}. We assume a shared-nothing architecture, where each of n processors has a private memory and a private disk. The processors are connected by a communication network and can communicate only by passing messages.

The main idea of our parallelization is as follows. First, the transaction database $\mathcal{D}$ is evenly distributed on the disks attached to the processors. We use $\mathcal{D}_i$ to denote the set of transactions at processor P_i, for all $i \in \{1 \sim n\}$. Let c be an itemset. The *local support count* of c at processor P_i refers to the number of transactions containing c in $\mathcal{D}_i$. We generate all required candidates and count their local supports at each processor. All the obtained local support counts will be sent to all other processors. Thus, when a processor completes its local support-counting and also receives all other local support counts from other processors, the global support counts for all candidates can be accumulated and all frequent itemsets can be obtained at this processor.

Now we introduce some concepts to help our discussion. Let T_1 and T_2 be two prefix-trees. We say that (1) T_1 is a *subtree in shape* of T_2, denoted by $T_1 \sqsubseteq T_2$, if for each $t_1 \in T_1$ there exists $t_2 \in T_2$ such that $t_1.item = t_2.item$ and the position (or occurrence) of t_1 in T_1 is the same as that of t_2 in T_2; (2) T_1 and T_2 are *in the same shape* if $T_1 \sqsubseteq T_2$ and $T_2 \sqsubseteq T_1$. We introduce the following procedure to do support accumulation for T_1 and T_2 in the same shape and save the result in T_1.

Algorithm AccumSup(T_1, T_2)
  for each $t_1 \in T_1$ **do**
    Let $t_2 \in T_2$ such that the positions of t_1 in T_1 and t_2 in T_2 are the same;
    $t_1.sup = t_1.sup + t_2.sup$;

Before giving our parallel solution, we first parallelize INTRA_MINER and INTER_MINER as follows, where we assume that all required input data can be available.

Algorithm PAR_INTRA_MINER
// A parallelization of INTRA_MINER
  for $i = 1$ to n at processor P_i
  do in parallel
    $C_{intra} = \{c \mid c \subseteq cliq, cliq \in Q,$
    c is intra-CPI and $|c| \geq 3\}$;
    Let T_i be an empty prefix-tree and **for** each $c \in C_{intra}$ **do** instree(T_i, c);
    CountSup(T_i, $\mathcal{D}_i$) and send T_i to all other processors;
    Receive T_j's from all other processors P_j's; *// Synchronization point*
    for $j = 2$ to n **do** AccumSup(T_1, T_j);
// $T_1 \sim T_n$ are in the same shape
    Collect all frequent intra-CPI k-itemsets for $k \geq 3$ to L_{intra} by a scan of T_1;
    $L_{intra} = L_{intra} \cup L_1 \cup \{x \in L_2 \mid x$ is intra-CPI$\}$;

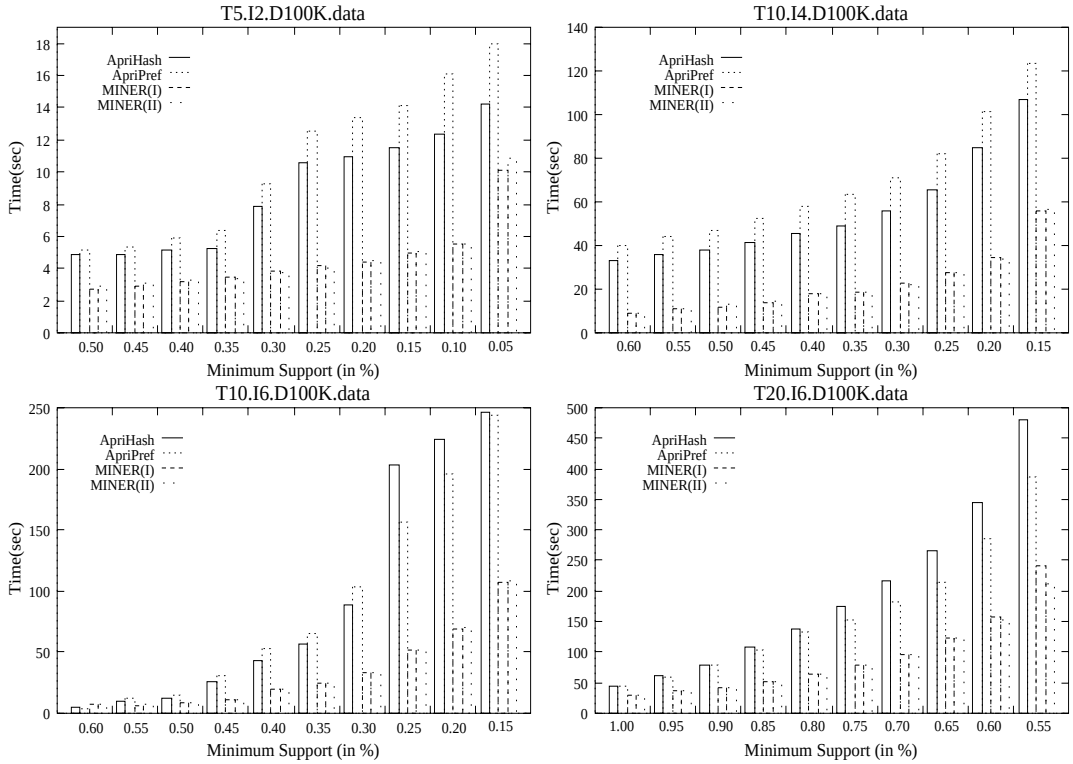

Figure 2: Performance comparison

Algorithm PAR_INTER_MINER
// A parallelization of INTRA_MINER
for $i = 1$ to n at processor P_i
do in parallel
$C_{inter} = \{c \mid c \subseteq cliq, cliq \in Q,$
$|c| \geq 3, c$ is inter-CPI, all intra-CPI subsets of c are in $L_{intra}\}$;
Let T_i be an empty prefix-tree and
for each $cliq \in Q$ **do** instree(T, c);
CountSup($T_i, \mathcal{D}_i$) and send T_i to all other processors;
Receive T_j's from all other processors P_j's; *// Synchronization point*
for $j = 2$ to n **do** AccumSup(T_1, T_j);
// $T_1 \sim T_n$ are in the same shape
Collect all frequent inter-CPI k-itemsets for $k \geq 3$ to L_{inter}
by a scan of T_1;
$L_{inter} = L_{inter} \cup \{x \in L_2 \mid x$ is inter-CPI$\}$;

Both algorithms above employ the same idea. First each processor P_i for all $i \in \{1 \sim n\}$ independently inserts all intra-CPI or inter-CPI candidates into a local prefix-tree T_i. Next, P_i uses local transaction data $\mathcal{D}_i$ to obtain the local supports for all candidates and sends the result tree T_i to all other processors. Then, it waits for receiving all T_j's from all other processors P_j's. After all $T_1 \sim T_n$ are available. P_i does support accumulation for all candidates and store the global supports into T_1. Finally, L_{intra} or L_{inter} can be generated easily at each P_i. It is not hard to see that the above procedure is complete. Thus we can present the parallelization of MINER(X) as follows.

Algorithm PAR_MINER(X) *// $X \in \{I, II\}$*
for $i = 1$ to n at processor P_i
do in parallel
Count local supports for all 1-itemsets and 2-itemsets; send the results to all other processors; receive local supports from all other processors; accumulate all local supports to get global supports; and then generate L_1 and L_2;
GEN_CLQ; *// Get $Q = \bigcup_{i=3}^{\mathsf{maxcliqsize}} Q_i$*
if maxcliqsize $\leq$ maxcansize **then**
for $i = 1$ to n at processor P_i
do in parallel
$C_{all} = \{c \mid c \subseteq cliq \in Q, |c| \geq 3\}$;
Let T_i be an empty prefix-tree and
for each $c \in C_{all}$ **do** instree(T_i, c);
CountSup(T_i, $\mathcal{D}_i$) and send T_i to all other processors;
Receive T_j's from all other processors P_j's;
// Synchronization point
for $j = 2$ to n **do**
AccumSup(T_1, T_j);
// $T_1 \sim T_n$ are in the same shape
Collect all frequent k-itemsets for $k \geq 3$ into L by a scan of T_i;
$L = L \cup L_1 \cup L_2$;
else
for $i = 1$ to n at processor P_i
do in parallel GEN_CPI(X);
// Get a CPI
PAR_INTRA_MINER;
// L_{intra} generation at $P_1 \sim P_n$
PAR_INTER_MINER;
// L_{inter} generation at $P_i \sim P_n$
for $i = 1$ to n at processor P_i
do in parallel $L = L_{intra} \cup L_{inter}$;

It is easy to see that PAR_MINER(X) is a natural parallelization of MINER(X) on a shared-nothing architecture by using our main idea mentioned before, where X may be I or II. This parallelization uses a simple principle of allowing "redundant computations in parallel on otherwise idle processors to avoid communication", which is also employed by *Count Distribution* algorithm [3]. Note that *Count Distribution* is the fastest parallelization of *Apriori*. However *Count Distribution* contains uncertainly multiple synchronization points due to the sum-reduction at the end of each pass to construct the global counts. This fact greatly limits the development of parallelism. Our PAR_MINER(X) has overcome this drawback because it contains at most 3 synchronization points. Furthermore, different from the methods in [11], all of our database partitions are non-overlapping, which guarantees there is no redundant support-counting operation in our method. Hence, our method develops the parallelism more sufficiently than both the above existing methods.

6 Conclusions

We have proposed new algorithms for efficient mining of association rules. Different from all existing algorithms, we introduce a concept of CPI (Complete Partition of Items) and divide all itemsets into two types: intra-CPI and inter-CPI. After obtaining all frequent 1-itemsets and 2-itemsets, we generate and maintain a set Q of item cliques (maximal potentially frequent itemsets). Furthermore, we have designed two methods for generating an effective CPI by using Q. Then, we can use Q and our CPI to get a set C_{intra} of intra-CPI candidates and count supports for them so that all frequent intra-CPI itemsets can be obtained. Finally, we use Q, our CPI and all frequent intra-CPI itemsets to generate a set C_{inter} of inter-CPI candidates and also count supports for them so that all frequent inter-CPI itemsets can be obtained.

Our algorithms have several advantages. First, their I/O costs are quite limited because they only require at most 3 scans over database. Second, they can make both sizes of C_{intra} and C_{inter} reasonable so that their computation costs are also effectively controlled. Third, they use a prefix-tree structure to store candidates, which can also reduce computation cost. As a result, they appear to be more efficient than *Apriori*, one of the best algorithms for association discovery. To confirm that, We have done the experiments to compare the performances of our algorithms together with *Apriori*. The test results show that our algorithms outperform *Apriori* consistently, by factors ranging from 2 to 4 in most cases.

Another advantage of our algorithms is that they are easy to be parallelized. We have also presented a possible parallelization for our algorithms based on a shared-nothing architecture. We observe that the parallelism can be developed more sufficiently in our parallelization than two of the best existing parallel algorithms.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo. Fast Discovery of Association Rules. *Advances in Knowledge Discovery and Data Mining*, Chapter 12, AAAI/MIT Press, 1996.

[2] R. Agrawal, T. Imielinski, A. Swami. Mining Associations between Sets of Items in Massive Databases. *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, Washington D.C., May 1993, 207-216.

[3] R. Agrawal, J.C. Shafer. Parallel Mining of Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 6, December 1996.

[4] D.W. Cheung and Y. Xiao. Effect of Data Skewness in Parallel Mining of Association Rules. *Proc. The Second Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD-98)*, Melbourne, Australia, April, 1998, 48-60.

[5] Dao-I Lin and Zvi M. Kedem. Pincer-Search: A New Algorithm for Discovering the Maximum Frequent Set. *EDBT'98*, March 1998.

[6] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. *AAAI Workshop on Knowledge Discovery in Databases (KDD-94)*, pages 181–192, July 1994.

[7] G. D. Mulligan and D. G. Corneil. Corrections to Bierstone's Algorithm for Generating Cliques. *J. Association of Computing Machinery*, 19(2):244-247, Apr. 1972.

[8] Jong Soo Park, Ming-Syan Chen and Philip S. Yu. An Effective Hash-Based Algorithm for Mining Association Rules. *Proc. 1995 ACM-SIGMOD Int. Conf. Management of Data*, San Jose, CA, May 1995.

[9] Jong Soo Park, Ming-Syan Chen and Philip S. Yu. Efficient Parallel Data Mining for Association Rules. *ACM CIKM '95*, Baltimore MD USA

[10] Ashok Savasere, Edward Omiecinski, Shamkant Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, 1995.

[11] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, Wei Li. New Parallel Algorithms for Fast Discovery of Association Rules. *Data Mining and Knowledge Discovery: Special Issue on Scalable High-Performance Computing for KDD*, pp 343-373, Vol. 1, No. 4, December 1997.

[12] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, Wei Li. New Algorithms for Fast Discovery of Association Rules. *Technical Report URCS TR 651*, University of Rochester, 1997.