Partitioning and Counting Inference (PCI) Approach For Mining Frequent Pattern From Large Database

*R.S. Thakur, **R.C. Jain, ***K.R. Pardasani

* Department of MCA, TIT, RGPV, Bhopal (M.P.) INDIA
** Department of Computer Applications, SATI, Vidisha, INDIA
*** Department of Mathematics, MANIT, Bhopal INDIA

Abstract

The cost of the frequent pattern discovery process comes from the reading of the database (I/O time) and generation of new candidates (CPU time). The number of candidates dominates the entire processing time. So aim of today algorithms reduces number of candidates. We present an efficient algorithm called PCI that allow perform as few support counts as possible. It follow concept of partition algorithm but not require access to all partition in all passes. Due to concept of key pattern it reduce number of candidate (CPU time) and number of passes of reading the database (I/O time). Our PCI Algorithm work well for strong correlated database and also restricted required passes for weakly correlated databases.

INTRODUCTION

Data Mining

Data Mining is the discovery of hidden information found in database and can be viewed as a step in the knowledge discovery process [by “Fayyad”]

Data mining is a process which identifies valid, novel and useful patterns from large amount of data.

 Association Rules Mining

It is most important data mining applications, used to identify relationship among a set of items in a database. These relationship are based on co-occurrence of the data items.

It is useful for
• analyze market basket transactions.
• management to increase the effectiveness associated with advertising, marketing, inventory.
• prediction of failure in telecommunications network by identify what events occur before a failure.

Mining Association rules consists two stages:

1. Find all frequent itemsets : by definition each of these itemsets will occur at least as frequently as a pre-determined minimum support count.

2. Generate association rules from the frequent itemsets: by definition these rules must satisfy minimum support and minimum confidence.

Transaction Database of grocery store

<table>
<thead>
<tr>
<th>Transaction ID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Bread, Butter, Milk</td>
</tr>
<tr>
<td>T2</td>
<td>Butter, Eggs, Milk</td>
</tr>
<tr>
<td>T3</td>
<td>Butter, Milk</td>
</tr>
<tr>
<td>T4</td>
<td>Bread, Butter</td>
</tr>
</tbody>
</table>

There are two association rules

1. When butter is purchased, bread is purchased 50% of the time.
2. 75% of time when butter is purchased so is Milk
Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items. Let $D$ be a database containing transactions $T$, such that $T \subseteq I$.

An association rule is an implication of the form $A \Rightarrow B$, where $A \cap B = \emptyset$.

The rule $A \Rightarrow B$ holds in the transaction set $D$ with support $s$, where $s$ is the percentage of transactions in $D$ that contain $A \cup B$.

The rule $A \Rightarrow B$ has confidence $c$ in the transaction set $D$ if $c$ is the percentage in $D$ containing $A$ that also contain $B$.

For disk resident databases, this requires reading the database completely for each pass resulting in a large number of disk reads. In these algorithms, the effort spent in performing just the I/O may be considerable for large databases.

The cost of the frequent pattern discovery process comes from the reading of the database (I/O time) and generation of new candidates (CPU time). The number of candidates dominates the entire processing time. So the aim of today algorithms reduces number of candidates.

In this paper, we proposed PCI (Partition with Counting Inference) approach, that is based on Partition [16] and Counting Inference method[17]. It requires at most 2 databases passes, when database is strongly correlated. In case database is weakly correlated it requires at most roughly 3 databases passes. In Contrast the Partition Algorithm, where the database scanned at most two times but CPU overhead is increased.

Our algorithm has much lower CPU overhead compared to previous Partition algorithm [16]. Because Partition method generate all frequent itemsets in each partition of each pass and itemsets contain their tidlist. If database is strongly correlated then tidlist of itemsets will be large(equal to number of records in database ) and require more memory ,and thus space use for remaining partitions in main memory will be small.
PCI Algorithm:-

Phase I
1. Find 1-itemsets with their support from each partition $P_i$
2. Merge Phase
   a) Merge and add support count of all local 1-itemset from each partition $P_i$
   b) Find frequent 1-itemset globally
3. Identify key pattern and non key pattern from global frequent 1-itemset
4. Calculate powerset $PK$ of key patterns
5. Find size $m$ of powerset (cardinality of powerset)

Phase II
1. If size of $m$ is small than
   a) Database scan second time for count support of only patterns belong to $PK$
   b) And rest k-itemsets ($K\geq 2$) calculate without accessing database, by using Pascal-Gen algorithm
2. If size of $m$ is large
   a) Access database and find 2-itemsets with their support count
   b) Simultaneously store address of records which contain key pattern // option 1
      OR
      Create New Database $ND$ of those records which contain key pattern // option 2

Phase III
a) Find large $K$-itemsets ($K\geq 3$)
   If choose option 1 then access only those records which are needed
   OR
   If choose option 2 then access new database $ND$ for find next large itemset

1. Find large 1-itemsets with their support from $CT$
2. For all $I \in CT$ do begin
3. If supp = 1 then key = true
4. If key is true then add to $PK$
5. For all $I = \emptyset$ do begin
6. For all $I \in CT$ do begin
7. $m = |I|$
8. Phase II
   a) For all $I \in CT$ do begin
   b) For all $I \in CT$ do begin
   c) $m = |I|$
   d) Phase II
   e) For all $I \in CT$ do begin
   f) If $m$ small then
   g) $PK = \text{powerset}(I)$
   h) For all $I \in CT$ do begin
   i) $PK = \text{get_support}(PK, p_i)$
   j) $PK = \text{get_support}(PK, p_i)$
   k) $PK = \text{get_support}(PK, p_i)$
   l) $PK = \text{get_support}(PK, p_i)$
   m) $PK = \text{get_support}(PK, p_i)$
   n) $PK = \text{get_support}(PK, p_i)$
   o) $PK = \text{get_support}(PK, p_i)$
   p) $PK = \text{get_support}(PK, p_i)$
   q) $PK = \text{get_support}(PK, p_i)$
   r) $PK = \text{get_support}(PK, p_i)$
   s) $PK = \text{get_support}(PK, p_i)$
   t) $PK = \text{get_support}(PK, p_i)$
   u) $PK = \text{get_support}(PK, p_i)$
   v) $PK = \text{get_support}(PK, p_i)$
   w) $PK = \text{get_support}(PK, p_i)$
   x) $PK = \text{get_support}(PK, p_i)$
   y) $PK = \text{get_support}(PK, p_i)$
   z) $PK = \text{get_support}(PK, p_i)$
   {#Phase II}
   a) For all $I \in CT$ do begin
   b) If $m$ small then
   c) $PK = \text{powerset}(I)$
   d) For all $I \in CT$ do begin
   e) $PK = \text{get_support}(PK, p_i)$
   f) $PK = \text{get_support}(PK, p_i)$
   g) $PK = \text{get_support}(PK, p_i)$
   h) $PK = \text{get_support}(PK, p_i)$
   i) $PK = \text{get_support}(PK, p_i)$
   j) $PK = \text{get_support}(PK, p_i)$
   k) $PK = \text{get_support}(PK, p_i)$
   l) $PK = \text{get_support}(PK, p_i)$
   m) $PK = \text{get_support}(PK, p_i)$
   n) $PK = \text{get_support}(PK, p_i)$
   o) $PK = \text{get_support}(PK, p_i)$
   p) $PK = \text{get_support}(PK, p_i)$
   q) $PK = \text{get_support}(PK, p_i)$
   r) $PK = \text{get_support}(PK, p_i)$
   s) $PK = \text{get_support}(PK, p_i)$
   t) $PK = \text{get_support}(PK, p_i)$
   u) $PK = \text{get_support}(PK, p_i)$
   v) $PK = \text{get_support}(PK, p_i)$
   w) $PK = \text{get_support}(PK, p_i)$
   x) $PK = \text{get_support}(PK, p_i)$
   y) $PK = \text{get_support}(PK, p_i)$
   z) $PK = \text{get_support}(PK, p_i)$
}
Experimental Results

The efficiency of PCI is analyzed with the help of experimental results of PCI in comparison with partition algorithm. It shows that PCI outperforms partition approach in terms of execution time.

We report experimental results on synthetic data set T10.I4.D100K [18], which is very large and sparse database. The number of items \( N \) is set to 1000.

<table>
<thead>
<tr>
<th>Minimum Support (%)</th>
<th>Partition Algorithm</th>
<th>PCI Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1458</td>
<td>1321</td>
</tr>
<tr>
<td>0.35</td>
<td>1022</td>
<td>871</td>
</tr>
<tr>
<td>0.5</td>
<td>735</td>
<td>614</td>
</tr>
<tr>
<td>0.6</td>
<td>681</td>
<td>568</td>
</tr>
<tr>
<td>0.7</td>
<td>629</td>
<td>548</td>
</tr>
<tr>
<td>0.8</td>
<td>606</td>
<td>510</td>
</tr>
<tr>
<td>0.9</td>
<td>546</td>
<td>498</td>
</tr>
<tr>
<td>1.0</td>
<td>541</td>
<td>485</td>
</tr>
<tr>
<td>1.5</td>
<td>378</td>
<td>342</td>
</tr>
</tbody>
</table>

See Figure 2, the best improvement occurs when minimum support is less. In this experiment, PCI run about 1.2 times faster than the Partition algorithm. The improvement came from reducing the number passes of reading the database and the number of candidates. Figure 3 shows total frequent itemsets according to minimum support. For instance, our algorithm can perform better when database is dense but when database is too sparse then number of 1-itemset key patterns (value of \( m \)) will be large, algorithm requires extra effort to create new database of particular records.
Performance Analysis and Conclusions

Our PCI algorithm based on concept of key pattern, which allow database; scan only for candidate key patterns. It also avoid concept of tidlist for itemset, which require more space. These both principal reduce I/O time for finding frequent patterns.

When database is weakly correlated then m is large (number of key pattern is large) in this situation we have two approach. In approach first store address of all records which contain key pattern accessing only these records we will find rest frequent k-itemsets. These records will be small because after second scanning the database we have only few key patterns. Best case of this approach is when all require records in one partition then need only for accessing one partition. But in worst case all records scattered in n partitions.

In first phase, PCI find locally large 1-itemset in linear order. In second phase if database is highly correlated, then size of m (cardinality of set of key pattern) is small and it access database only for few candidates, which is super set of key patterns. This process requires less time for I/O as well CPU. So we have need only two scan of databases when database is strongly correlated.

In approach second we create new database of those records, which contain key pattern. This new database access for finding rest large k-itemset (k>3). If database require roughly three pass then

Total block access = D + D + K = 2D + K

Where whole database (n blocks) are read in first and second pass and K is size of new database of records containing key patterns and is accessed in the third pass.
Best case of this approach is size of K = 0 that means require 2 database passes, average case is when K = D/2 and worst case will be when K = D that is 3 database passes will be required.

This approach will be useful for incremental mining. When new record is added in database then for finding large pattern, use this new database with new coming record rather then old whole database. This approach give speedup by reduces I/O time, but it requires secondary storage space for storing new database.

Advantages of the PCI algorithm are as follows:

• Lower CPU overhead, because itemsets do not require tidlist.
• Require less runtime for partition, as it accesses records contains key pattern.
• Our algorithm can potentially save I/O time, specially when database is strongly correlated.
• The algorithm is inherently parallel in nature and can be parallelized with minimal communication and synchronization between the processing nodes.

References: