Applying On-Line Bitmap Indexing to Reducing Counting Costs in Mining Association Rules *

Xuemin Lin and Yijun Li
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
Email: {lxue, yijun}@cse.unsw.edu.au

Chi-Ping Tsang
Department of Computer Science
University of Western Australia
Nedlands, WA 6907, Australia
Email: tsang@cs.uwa.edu.au

Abstract

Counting costs have a great impact on efficiency of mining association rules in a large database of sales transactions. In this paper, we first formly analyze the facts that determine counting costs. Secondly, we present an on-line bitmap indexing technique to speed up counting process. Besides theoretical analysis, our implementation reports suggest that this indexing technique may reduce counting costs up to 50%, and almost brings no extra costs.

Keywords: Relational Databases, Data Mining, and Algorithms.

1 Introduction

Nowadays, the competitiveness of companies is increasingly dependent on the quality of their decision making. It is no wonder that companies often try to learn the public interests from past transactions to improve the quality of decisions taken for the future. Serving for this kind of purposes, data mining emerges as a new research area in databases. In the area of data mining, the problem of discovering association rules has recently received a great deal of attention [1, 2, 4, 6, 10]. In this problem, we are given a set of items and a large collection of transactions which are subsets of these items. The task is to find relationships between the presence of various items within those transactions.

Mining association rules is one of the fundamental problems in data mining and has a number of applications. A typical application from which the problem is named is supermarketing, where the items are products and the transactions are customer purchases at the checkout. In this application, an association might be 80% of customers who purchase milk and bread also buy butter. Discovering all such rules can be very useful for planning and marketing. Other applications include spatial data mining [7] and web access patterns discovery [5, 8].

The problem of mining association rules was first investigated in [1]. In this pioneering work, it is shown that mining association rules can be decomposed into two subtasks. First, we need to identify all subsets of items - itemsets - that are contained in a sufficient number of

*The research is partially supported by a small ARC grant.

transactions above the given minimum requirement (support value); these itemsets are called "large itemsets". Once the large itemsets are obtained, the desired association rules can be generated straightforwardly.

As the costs in the first subtask are much higher than those in the second subtask, existing research work focused on the developments of efficient algorithms for mining large itemsets. Several algorithms [2, 4, 9, 10] have been proposed for this purpose. These existing algorithms follow the same framework as proposed in [2]. First, candidates of large itemsets are constructed. Then the database is scanned and candidates are counted against each transaction (tuple) to obtain the actual large itemsets. Since sizes of the database and the candidates may be too large to place every thing in the main memory, the algorithms for mining large itemsets are implemented in an iterative fashion. The large itemsets discovered in one iteration will be used as the basis to generate the candidate set for the next iteration. Apriori [2] was the first algorithm to efficiently handle the iterations.

In [9, 10], it was pointed out that the dominant costs of mining large itemsets occur in processing earlier iterations. The costs consist of three equally important components:

1. the costs of reading database from disk to main memory;
2. the costs of building an index (hash tree) on candidates for efficient counting; and
3. the costs of counting candidates against the database.

Savasere, Omiecinski, and Navathe [10] noticed unnecessary I/O overhead in Apriori due to multiple disk readings, and then proposed a partition based algorithm that guarantees at most twice disk readings. Meantime, Park, Chen, and Yu [9] also noticed some other inefficiency in Apriori. A hash based algorithm DHP has been proposed that aims at producing smaller size of candidates in the second iteration and trimming the original database to a much smaller one from the third iteration to speed up the rest computation. It has been experimentally shown [9] that the algorithm DHP is more efficient than Apriori and generates much smaller candidates in the second iteration at the costs of higher demanding for main memory.

Consider that there is no formal analysis of the performances of the existing algorithms except experiment results. In this paper, we shall present a way to carry out this. More specifically, we show the facts that determine the counting process costs. This yields an explanation of why DHP is more efficient than Apriori in most cases. This is the first contribution of the paper.

As there is no consideration in trimming a database smaller to reduce counting costs in the second iteration, in this paper we shall propose an on-line bitmap indexing technique to logically truncate the database "vertically". The indexing technique employs a projection-like operation (PLO). Our implementation shows that a combination of DHP and PLO may speed up counting process time up to 50%, comparing to DHP, at little extra costs. Consider that DHP has extra demands of available main memory and disk storage space; and some times the extra demands are not able to be accommodated due to physical limitations. We also implement a combination of Apriori and PLO. Our implementation suggests that the combination can save counting process time up to 30% comparing to Apriori. A theoretical investigation is also undertaken to show why such a saving can be made. This is the second contribution of the paper.

The rest of the paper is organized as follows. In section 2, we formally define the problem of mining association rules, and then briefly describe the framework of mining large itemsets by iterations. In section 3, we show the relationships among counting costs, database size, and candidates size. Section 4 presents PLO and its performance analysis. We then detail our experiment results in section 5. This follows by conclusions and remarks.

2 Mining Association Rules

Let $I = \{i_1, i_2, ..., i_m\}$ be a set of literals, called items; and D be a set of transactions where each transaction T is a set of items such that $T \subseteq I$ and is associated with a unique identifier TID. Following the assumptions made in [1, 2, 9, 10], we assume that the items in each transaction are sorted on their keys in an increasing order. A transaction T *contains* a set X of some items if $X \subseteq T$. An *association rule* is an implication of the form $X \Rightarrow Y$ where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ has *support* s in the database D of transactions if at least $s\%$ of transactions in D that contain $X \cup Y$. The rule $X \Rightarrow Y$ *holds* in D with *confidence* c if at least $c\%$ of transactions in D that contain X also contain Y.

The problem of mining association rules is to find all association rules with respect to a given minimum support value $s\%$ and a given confidence $c\%$. A set X of some items is a *k-itemset* if it consists of k items, and is *large* if there are at least $s\%$ of transactions in D that contain X.

As mentioned in section 1, we shall focus on computing all large itemsets. This is usually done through an iteration fashion [2, 3, 9, 10] that is briefly described as follows. Let L_k denote the set of large k-itemsets and C_k to denote the candidate set of large k-itemsets.

Mining Large Itemsets
Scan D to obtain L_1
for $(k = 2; L_{k-1} \neq \emptyset; k++)$ **do**
  { $C_k = gen(L_{k-1})$; //generate new candidates from L_{k-1}.
    if $C_k \neq \emptyset$ **then**
      { $count(C_k)$; //count candidates against D.
        $L_k = \{c \in C_k | c.count \geq s\}$; }
    else $L_k = \emptyset$ }
Return $\cup L_k$.

As we shall discuss only Apriori and DHP in this paper, now we explain these two algorithms by an example. Suppose $I = \{A, B, C, D, E, F\}$ and $L_1 = \{B, C, E, F\}$.

In Apriori, $C_2 = L_1 \times L_1 = \{(B,C), (B,E), (B,F), (C,E), (C,F), (E,F)\}$. The counting process $count(C_2)$ is implemented as follows:

- Organize an hash index - a hash tree - for C_2; then
- read the database D into main memory; and then
- for each $T \in D$, do counting against C_2.

Note that each element (x, y) in C_2 is an ordered pair such that the key of x is smaller than that of y. Here, the hash tree [2] is built in such a way that the root corresponds to a hash table that hashes first item in each element in C_2; while the children of the root store the corresponding hash tables that hashes the second item in each element in C_2. Therefore, all elements of C_2

are attached to an appropriate hash bucket of a hash table in the second level. For a tuple $T = (x_1, x_2, ..., x_l)$, the counting of C_2 against T is implemented as follows:

$\underline{count(T, C_2)}$
for $(i = 1; i \neq l - 1; i++)$ **do**
  if the hash bucket containing x_i is not empty **then**
    for $(j = i + 1; j \neq l; j++)$ **do**
      { search the elements in the hash bucket containing x_j to determine if $(x_i, x_j) \in C_2$
        if $(x_i, x_j) \in C_2$ **then**
          add 1 to the count number of (x_i, x_j); }

Suppose that L_2 is discovered to be $\{(B, C), (B, E), (B, F), (C, E), (C, F)\}$. Then, in Apriori $C_3 = \{(B, C, E), (B, E, F)\}$, and do counting of C_3, and so on. Generally, C_k is generated from L_{k-1} such that for each element $X \in C_k$, all sub $(k-1)$-itemsets of X are in L_{k_1}. To implement this efficiently, Apriori suggests that L_{k-1} is first sorted according to the lexicographical ordering, then apply the *sort-merge* join together with a prune technique. The development of a hash tree on C_k is similar to the technique on C_2.

The algorithm DHP follows the same framework as Apriori; but the following two additional operations [9] are added to reduce the size of C_2 and the size of the database from $k = 3$:

- Build a hash table H_2 while generating L_1, and then check C_2 against H_2 to reduce the size of C_2.
- Trim database while counting C_2 so that the database used for counting process from $k = 3$ is smaller than the original database.

For instance, with respect to the above example C_2 in DHP may be equal to L_2 by using H_2. From the experiments in [9], C_2 in DHP can be much smaller than that in Apriori; and the database size may also be reduced significantly. Therefore, the total execution time can be greatly reduced although extra costs are spent in building H_2. Next, we analyze the computation costs in Apriori and DHP; particularly we concentrate on counting process costs. Then, we shall be able to see what may be further improved.

3 Analysis of Counting Costs

Most existing algorithms in mining large itemsets adopt the same counting process as presented in the preceding section. In this section, we estimate the countings costs, and then present a relationship between counting costs and candidates. More specifically, we analyze the counting costs in the second iteration, because in practice it is the most expensive part. Although a rigorous analysis for the later iterations ($k \geq 3$) tends to be much more complicated and is not presented in the paper, we believe that the similar results to the second iteration should hold. This is confirmed by our experiments.

For an item i_j, we use $n(i_j)$ to denote its occurrences in D; and for a pair (i_j, i_l) of items, we use $n(i_j, i_l)$ to denote its occurrences in D. If we assume that the possibilities of the occurrences of each item in the database D are independent, then the relationship among the occurrences of a single item and a pair of items is as follows.

Lemma 1 *Suppose that the occurrences of two items i_j and i_l are independent in the database D. Then, on average $n(i_j, i_l) = \frac{n(i_j)n(i_l)}{|D|}$.*

Proof: Note that the probability for an item i_m to occur in a given tuple is $\frac{n(i_m)}{|D|}$. Since the occurrences of a item in D are independent with others, the probability for (i_j, i_l) to occur in a given tuple is $p(i_j, i_l) = \frac{n(i_j)n(i_l)}{|D|^2}$. From the fact that the average occurrence of (i_j, i_l) is $|D|p(i_j, i_l)$, the Lemma immediately follows. $\Box$

Suppose that $T = (i_{j_1}, i_{j_2}, ... i_{j_n})$ is a transaction (tuple) in a given database D. As described in the preceding section, the process *count*(T, C_2) is implemented as follows. For $1 \leq l \leq n-1$, we first hash i_{j_l} into the hash table in the first level (root) of the hash tree of C_2. If the corresponding hash bucket E_1 has a child then we have to consider all combinations $\{(i_{j_l}, i_{j_m}) : l < m \leq n\}$. This can be done by hashing each i_{j_m}, for $1 < m \leq n$, into a bucket E_2 in the second level of the hash tree. Again, if E_2 is not empty then we have to compare (i_{j_l}, i_{j_m}) with each 2-itemset included in E_2 to do counting. However, in case that E_1 is empty then there is no need to consider the combinations $\{(i_{j_l}, i_{j_m}) : l < m \leq n\}$. Similarly, if E_2 is empty then (i_{j_l}, i_{j_m}) will be skipped. In the following, we estimate the costs with respect to these operations in the counting process.

Suppose that in an execution of the counting process *count*(C_2) in the second iteration, I_{1,C_2} represents the set of items that appear as the first item in the elements in C_2, and I_{2,C_2} represents the set of items that appear as the second item in the elements in C_2. Note that a hash bucket of the hash tree of C_2 may contain additional items which are not in I_{1,C_2} nor I_{2,C_2}. We use I^h_{1,C_2} to denote the set of items hashed into the first level hash buckets containing the items in I_{1,C_2}, and I^h_{2,C_2} to denote the set of items hashed into the second level hash buckets containing the items in I_{2,C_2}. Clearly, $I_{1,C_2} \subseteq I^h_{1,C_2}$ and $I_{2,C_2} \subseteq I^h_{2,C_2}$.

Note that while building up a hash tree of C_2, we always choose the same number of hash buckets for each hash table. Consequently, we can use constant β to represent costs of an implementation of a hash to determine if the hash bucket is empty.

As C_2 is produced by sorting L_1 first and then use the sort-merge join, the elements of C_2 in a hash bucket E_2 are stored according their lexicographical ordering. Therefore, while searching if (i_j, i_m) is in a hash bucket, we can either use binary search or a linear scan. Because, in an algorithm for mining large itemsets, we normally adopt a reasonably large number of hash buckets. This implies that on average there are only a few elements attached to a hash bucket. So, a linear scan appears more efficient. Our costs analysis is based on the linear scan technique. We use $\alpha_{C_2}(i_j, i_l)$ to represent the costs of searching if (i_j, i_l) is contained in a second level hash bucket. The costs (execution time) $cost_{C_2,I_2}$ of *count*(C_2) with respect to set I_2 of the items, that are used to count C_2, can be represented as follows. Note that in the existing algorithms, I_2 is always equal to I. However, our algorithms lead to a smaller I_2.

$$\begin{aligned} cost_{C_2,I_2} &= \beta((\sum_{i_j \in I_2} n(i_j)) - |D|) + \sum_{i_{j_m} \in I^h_{2,C_2}} \sum_{i_{j_l} \in I^h_{1,C_2}, j_l < j_m} \{(\alpha_{C_2}(i_{j_l}, i_{j_m}) + \beta)\ n(i_{j_l}, i_{j_m})\} \\ &\quad + \beta \sum_{i_{j_m} \in I - I^h_{2,C_2}} \sum_{i_{j_l} \in I^h_{1,C_2}, j_l < j_m} n(i_{j_l}, i_{j_m}) \qquad (1) \end{aligned}$$

Here, the first part in the costs corresponds the summation of the costs of hashing the first $|T|-1$ items of each transaction T in D. Intuitively, the counting costs as described in (1) should get smaller while I and C_2 are reduced. This can be formally presented as the following theorem.

Theorem 1 *In two executions $\Re$ and $\Im$ of counting process, suppose that the hash trees are built in the same way, that is the same hash buckets are used. Further suppose that in $\Re$, its I_2 is represented as $I^{\Re}$ and its C_2 is represented as $C_2^{\Re}$; and in $\Im$, its I_2 is represented as $I^{\Im}$ and its C_2 is represented as $C_2^{\Im}$. Furthermore, we suppose that $C_2^{\Re} \subseteq C_2^{\Im}$ and $I^{\Re} \subseteq I^{\Im}$. Then $cost_{C_2^{\Re},I^{\Re}} \leq cost_{C_2^{\Im},I^{\Im}}$.*

Proof: Note that $C_2^{\Re} \subseteq C_2^{\Im}$. Thus, $I_{1,C_2^{\Re}} \subseteq I_{1,C_2^{\Im}}$ and $I_{2,C_2^{\Re}} \subseteq I_{2,C_2^{\Im}}$. Because $I^{\Re} \subseteq I^{\Im}$, we have $I^h_{1,C_2^{\Re}} \subseteq I^h_{1,C_2^{\Im}}$ and $I^h_{2,C_2^{\Re}} \subseteq I^h_{2,C_2^{\Im}}$. One can immediately verify that $\alpha_{C_2^{\Re}}(i_l, i_m) \leq \alpha_{C_2^{\Im}}(i_l, i_m)$ for any relevant pair (i_l, i_m) of items. Consequently,

$$
\begin{aligned}
cost_{C_2^{\Im},I^{\Im}} &= \beta((\sum_{i_j \in I^{\Im}} n(i_j)) - |D|) + \sum_{i_{j_m} \in I^h_{2,C_2^{\Im}}} \{ \sum_{i_{j_l} \in I^h_{1,C_2^{\Im}}, j_l < j_m} (\alpha_{C_2^{\Im}}(i_{j_l}, i_{j_m}) + \beta)\ n(i_{j_l}, i_{j_m})\} \\
&\quad + \beta \sum_{i_{j_m} \in I^{\Im} - I^h_{2,C_2^{\Im}}} \ \sum_{i_{j_l} \in I^h_{1,C_2^{\Im}}, j_l < j_m} n(i_{j_l}, i_{j_m}) \\
&\geq \beta((\sum_{i_j \in I^{\Re}} n(i_j)) - |D|) + \sum_{i_{j_m} \in I^h_{2,C_2^{\Im}}} \{ \sum_{i_{j_l} \in I^h_{1,C_2^{\Re}}, j_l < j_m} (\alpha_{C_2^{\Re}}(i_{j_l}, i_{j_m}) + \beta)\ n(i_{j_l}, i_{j_m})\} \\
&\quad + \beta \sum_{i_{j_m} \in I^{\Re} - I^h_{2,C_2^{\Im}}} \ \sum_{i_{j_l} \in I^h_{1,C_2^{\Re}}, j_l < j_m} n(i_{j_l}, i_{j_m}) \\
&= \beta((\sum_{i_j \in I^{\Re}} n(i_j)) - |D|) + \sum_{i_{j_m} \in I^h_{2,C_2^{\Re}}} \{ \sum_{i_{j_l} \in I^h_{1,C_2^{\Re}}, j_l < j_m} (\alpha_{C_2^{\Re}}(i_{j_l}, i_{j_m}) + \beta)\ n(i_{j_l}, i_{j_m})\} \\
&\quad + \sum_{i_{j_m} \in I^h_{2,C_2^{\Im}} - I^h_{2,C_2^{\Re}}} \ \sum_{i_{j_l} \in I^h_{1,C_2^{\Re}}, j_l < j_m} \{(\alpha_{C_2^{\Re}}(i_{j_l}, i_{j_m}) + \beta)\ n(i_{j_l}, i_{j_m})\} \\
&\quad + \beta \sum_{i_{j_m} \in I^{\Re} - I^h_{2,C_2^{\Im}}} \ \sum_{i_{j_l} \in I^h_{1,C_2^{\Re}}, j_l < j_m} n(i_{j_l}, i_{j_m}) \\
&\geq cost_{C_2^{\Re},I^{\Re}}.
\end{aligned}
$$

□

Consider Apriori and DHP. Since $C_2^{Apriori} \supseteq C_2^{DHP}$, the counting process costs in DHP is thus smaller than that in Apriori.

Corollary 1 *In the second iteration, $cost_{C_2^{Apriori},I^{Apriori}} \geq cost_{C_2^{DHP},I^{DHP}}$.*

Note that DHP is more efficient than Apriori in the second iteration because of a reduction on the size of candidates. According to Theorem 1, a reduction on the size of I_2 may also lead to a reduction on counting costs. In the next section, we present such a technique.

4 Applying Projection First while Counting

Consider the example in section 2, where in Apriori, $C_2 = \{(B,C),(B,E),(B,F),(C,E),(C,F),(E,F)\}$. Suppose that a transaction T in the database is $T = (A,B,D,E,F)$. While executing counting of the elements in C_2 against T, we first hash A using the first level hash table to see if the item A is in a non-empty hash bucket. If the hash bucket is non-empty then we have to consider all combinations: (A,B), (A,D), (A,E), and (A,F). However, it should be clear that all these combinations will not contribute a vote to the large 2-itemset, because they are not included in the candidate set C_2. Therefore, many computation costs may be wasted. Similarly, DHP also has this problem .

To resolve this problem, in this section we present a projection-like operation (PLO) to be used in the counting process to maintain an on-line bitmap index. The idea is very straightforward. In the above example, when $T = (A,B,D,E,F)$ is read from disk, we can first filter out A because it is not contained in C_2. Then, we buffer only (B,D,E,F) for the counting purpose.

Let L_1^k denote the set of items that appear in C_k. The following Lemmas can be immediately verified.

Lemma 2 *For $k \geq 2$, $L_1^k \subseteq L_1^{k-1} \subseteq L_1$.*

Proof: Note that C_k is generated from C_{k-1}. $\Box$

Lemma 3 *For each transaction (tuple) T in the database, it is equivalent that we count C_k using $T' = T \cap L_1^k$.*

Proof: Note that for each element $(i_{j_1}, i_{j_2}, ..., i_{j_k}) \in C_k$, $i_{j_l} \in L_1^k$ for $1 \leq l \leq k$. Thus, we can do counting using only T'. $\Box$

Consider that in DHP, we also have to trim a tuple T while counting C_2. The trimming process is based on a hash test to determine whether T or part of T will contribute to the counting process for C_3. Actually, this trimming test can also be based on the $T' = T \cap L_1^2$ instead of T, because C_3 is generated from C_2. In the following, we present an efficient execution of PLO.

Clearly, when a transaction T is read in, we cannot scan L_1^k to determine if an item in T is in L_1^k, because it is too expensive to scan L_1^k each time. Instead, we use an array bitmap B_k such that the binary value of the jth element $B_k[j]$ of B_k indicates if the item $i_j \in I$ is in L_1^k:

- $B_k[j] = 1$ means $i_j \in L_1^k$, and
- $B_k[j] = 0$ means that $i_j \notin L_1^k$.

In either DHP or Apriori, while scanning C_k to get L_k, we can build up B_k as well. Then for each tuple $T = \{i_{j_1}, i_{j_2}, ..., i_{j_n}\}$, before buffering T to do counting, we scan T to check the value of $B_k(j_l)$ for $1 \leq l \leq n$. We buffer only items i_{j_l} with $B_k(j_l) = 1$; that is, we buffer only T' for the counting purpose.

We call such a combination of Apriori and PLO as PLO-Apriori, and call such a combination of DHP and PLO as PLO-DHP. Because $I^{PLO-Apriori} \subseteq I^{Apriori}$ and $I^{PLO-DHP} \subseteq I^{DHP}$, by Theorem 1 we can conclude that

- $cost_{C_2^{Apriori}, I^{Apriori}} \geq cost_{C_2^{PLO-Apriori}, I^{PLO-Apriori}}$, and
- $cost_{C_2^{DHP}, I^{DHP}} \geq cost_{C_2^{PLO-DHP}, I^{PLO-DHP}}$.

Together with Corollary 1, we have:

Corollary 2 $cost_{C_2^{Apriori}, I^{Apriori}} \geq cost_{C_2^{DHP}, I^{DHP}} \geq cost_{C_2^{PLO-DHP}, I^{PLO-DHP}}$.

In the next section, our experiments will quantitatively report the reduction on execution time by applying PLO-Apriori and PLO-DHP.

5 Experimental Results

To quantitatively access the actual speed-up by using PLO, we implement the algorithms Apriori, PLO-Apriori, DHP, and PLO-DHP on a Pentium Pro Server with two CPUs (only one cpu was used in our implementation) clock rate of 180MHz, 128MB of main memory, running Window-NT 4.0. The synthetic data used in the implementations are generated by the program downloaded from the web site of IBM Almaden Research Center. The synthetic data generation program takes the parameters shown in Figure 1.

|D|	Number of transactions
|T|	Average size of transactions
|I|	Average size of the maximal potentially large itemsets
|L|	Number of potentially large itemsets
N	Number of items

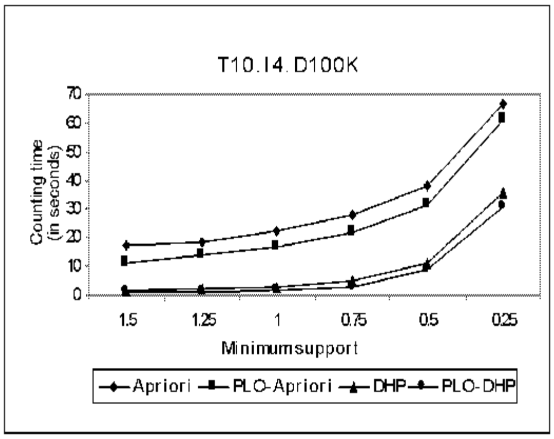
Figure 1: parameters

We choose 300 hash buckets for each node in a hash tree in our experiments. While N is chosen as 1000 and $|I|$ is chosen as 4, we vary the values of $|T|$ and $|D|$. Figure 2 shows the implementation results. There, $TX.IY.DZK$ means that $|T| = X$, $|I| = Y$, and $|D| = Z \times 1000$; we recorded all CPU time except that of disk reading. The experiment results are presented in both table and graph forms.

As shown in the experiment, PLO-DHP may speed up the CPU time of DHP up to 50%, while PLO-Apriori may speed up the CPU time of Apriori up to 33%. In fact, we believe that PLO reduce more CPU time while applied to Apriori. Comparing with DHP, Apriori takes more CPU time to build up a hash tree, and this, of course, cannot be saved by applying PLO. Thus, it appears that the saving percentage between PLO-DHP and DHP is higher. Note that when minimum support gets smaller, more items are involved in large itemsets, and thus the saving percentage drops. Furthermore, we can see that the CPU time is lower if minimum support value is lager. This is due to the expenses we spend on building a hash tree.

Now, we discuss the advantages and disadvantages of applying these algorithms. From our implementation we can see that PLO-DHP overperforms the other algorithms. However, Count time (in seconds) T10.I4.D100K

	M=1.5	M=1.25	M=1.0	M=0.75	M=0.5	M=0.25
Apriori	17.1	18.7	22.2	28.0	37.8	66.6
PLO-Apriori	11.4	13.8	17.0	21.8	31.1	61.0
DHP	1.8	2.0	2.5	5.0	11.2	35.7
PLO-DHP	1.1	1.2	1.4	2.8	9.1	30.7

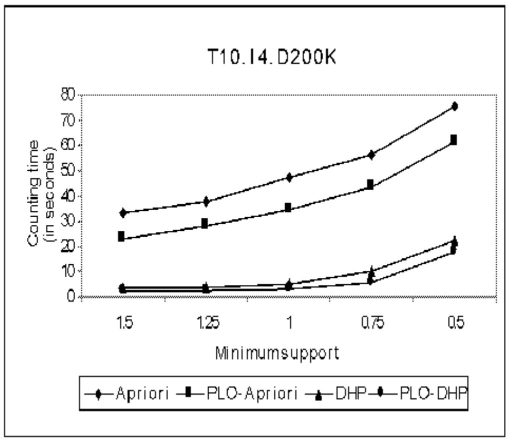


Count time (in seconds) T10.I4.D200K

	M=1.5	M=1.25	M=1.0	M=0.75	M=0.5
Apriori	33.0	37.8	47.3	56.1	75.3
PLO-Apriori	22.9	28.2	34.6	43.6	61.5
DHP	3.8	4.1	5.4	10.0	22.1
PLO-DHP	2.5	2.6	3.1	5.7	17.6

Count time (in seconds) T15.I4.D200K

	M=1.5	M=1.25	M=1.0	M=0.75	M=0.5
Apriori	100.0	115.1	135.5	175.4	267.2
PLO-Apriori	74.2	84.6	105.3	138.2	218.5
DHP	10.2	15.1	27.2	49.5	105.8
PLO-DHP	5.2	8.2	17.2	37.2	91.9



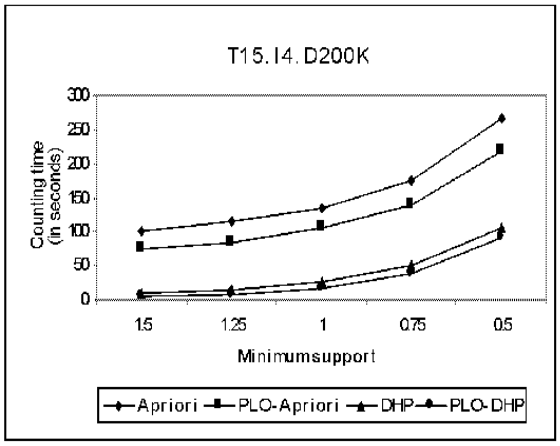


Figure 2: experiment results

PLO-DHP, just like DHP, requires extra availability of main memory. If the demand cannot be accommodated, PLO-DHP cannot be applied, neither DHP. PLO-Apriori, then, should be applied, because it is more efficient than Apriori. Comparing with Apriori, PLO-Apriori also requires some extra memory space to store the bitmap index. However, the extra space should be no more than the requirement to store a bitmap corresponding to the whole set of items, whose cardinality is just a few thousands in many applications. Theorefore, PLO-Apriori should be a better choice if we cannot use PLO-DHP due to physical limitation. On the other hand, we should note that if the minimum support value is extremely small, then there is little advantage to apply PLO.

6 Conclusions and Remarks

In this paper, we studied the problem of mining association rules. We first formally analyse the facts that determine the speed of mining algorithms. Then, we introduce an online bitmap indexing technique employing PLO. We also showed that PLO is very effective by both theoretical analysis and implementation results.

Note that mining algorithms for association rules have been applied into many different applications, such as web data mining and spatial data mining. Some applications may have huge number of items. We next will investigate how PLO technique can be applied into these applications.

References

[1] R. Agrawal, T. Imielinski, A. Swami, Mining Association Rules between Sets of Items in Large Databases, *Proceedings of 1993 ACM SIGMOD*, 207-216, 1993.

[2] R. Agrawal and R. Srikant, Fast Algorithms for Mining Association Rules, *Proceedings of the 20th VLDB Conference*, 487-499.

[3] R. Agrawal and R. Srikant, Mining Sequential Patterns, *Proceedings of the 11th International Conference on Data Engineering*, 1995.

[4] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, Dynamic Itemset Counting and Implication Rules for Market Basket Data, *Proceedings of 1997 ACM SIGMOD*, 225-264, 1997.

[5] M.-S. Chen, J. S. Park, and P. S. Yu, Data Mining for Path Traversal Patterns in a Web Environment, *Proceedings of the 16th International Conference on Distributed Computing Systems*, IEEE cs press, 385-392, 1996.

[6] M. Houtsma and A. Swami, Set-Oriented Mining for Association Rules in Relational Databases, *Proceedings of the 11th International Conference on Data Engineering*, 25-33, 1995.

[7] K. Koperski and J. Han, Discovery of Spatial Association Rules in Geographic Information Databases, *SSD'95*, LNCS, Springer-Verlage, 1995.

[8] B. Mobasher, N. Jain, E-H. Han, and J. Srivastava, Web Mining: Pattern Discovery from World Wide Web Transactions, *Tech. Report*, Department of Computer science, University of Minnesota, 1996.

[9] J. S. Park, M.-S. Chen, and P. S. Yu, An Effective Hash-Based Algorithm for Mining Association Rules, *Proceedings of 1995 ACM SIGMOD*, 175-186, 1995.

[10] A. Savasere, E. Omiecinski, and S. Navathe, An Efficient Algorithm for Mining Association Rules in Large Databases, *Proceedings of 21st VLDB Conference*, 432-443, 1995.

[11] H. Toivonen, Sampling Large Databases for Association Rules, *Proceedings of 22nd VLDB Conference*, 1996.