An Algorithm for Mining Association Rules Using Perfect Hashing and Database Pruning

S. Ayşe Özel and **H. Altay Güvenir**
Bilkent University, Department of Computer Engineering,
Ankara, Turkey.
{selma, guvenir}@cs.bilkent.edu.tr

ABSTRACT

In this paper, we propose an algorithm for finding frequent itemsets in transaction databases. The basic idea of our algorithm is inspired from the Direct Hashing and Pruning (DHP) algorithm, which is in fact a variation of the well-known Apriori algorithm. In the DHP algorithm, a hash table is used in order to reduce the size of the candidate *k*+1 itemsets generated at each step. The difference of our algorithm is that, it uses perfect hashing in order to create a hash table for the candidate *k*+1 itemsets. As perfect hashing is used, the hash table contains the actual counts of the candidate *k*+1 itemsets. Hence we do not need to make extra processing to count the occurrences of candidate *k*+1 itemsets as in the DHP algorithm. The algorithm also prunes the database at each step in order to reduce the search space. We also tested our algorithm with real datasets obtained from a large retailing company and observed that our algorithm performs better than the Apriori algorithm.

I. INTRODUCTION

Data mining has been considered as a promising field in the intersection of databases, artificial intelligence, and machine learning [1, 2]. Association rule mining has been one of the most popular data mining subjects, which can be simply defined as finding interesting rules from large collections of data. Association rule mining has a wide range of applicability. When the association rule mining was first introduced, it was developed for finding significant association rules between items in the huge sized point of sale transaction databases in order to provide valuable information to the management of the retail store such as what to put on sale, how to design coupons, how to place merchandise on shelves to maximize profit, etc. [3, 4]. Today, it is also used for building statistical thesaurus from the text databases [5], finding web access patterns from web log files [6], and also discovering associated images from huge sized image databases [7].

In the sale transaction databases domain, an example association rule may be that 90% of transactions that purchase bread and butter also purchase milk. The following is a formal statement of association rule mining for transaction databases [2, 4]: Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items and D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. Each transaction has a unique transaction identifier called its TID. We say that a transaction T contains X if $X \subseteq T$, where X is a set of some items in I. An association rule is an implication of the form $X \Rightarrow Y$, where X and Y are sets of some items in I such that they are disjoint. The rule $X \Rightarrow Y$ holds in the database 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 $X \cup Y$. Given the database D, the problem of mining association rules involves the generation of all association rules that have support and confidence greater than or equal to the user-specified *minimum support* and *minimum confidence*.

The discovery of association rules for a given dataset *D*, involves two main steps [5]: The first step is to find each set of items, called as itemsets, such that the co-occurrence rate of these items is above the minimum support, and these itemsets are called as *large itemsets* or *frequent itemsets*. The size of an itemset represents the number of items in that set. If the size of an itemset is equal to *k*, then this itemset is called as the *k-itemset*. The second step is to find association rules from the frequent itemsets that are generated in the first step. The second step of the generation of association rules is straightforward. In that step, for every frequent itemset *f*, all non-empty subsets of *f* are found. Then for every such subset *a*, a rule of the form $a \Rightarrow (f - a)$ is generated if the ratio of *support*(*f* - *a*) to *support*(*a*) is greater than or equal to the minimum confidence.

However the first step of association rule mining, finding the frequent itemsets, is very resource consuming task and for that reason, it has been one of the most popular research field in data mining. Several algorithms, AIS [3], SETM [8], Apriori [4], Direct Hashing and Pruning [5, 9], Partition [10], Sampling [11], and some other parallel algorithms [12] have been developed. In this study, a fast algorithm based on Direct Hashing and Pruning (DHP) algorithm is proposed. The DHP algorithm is described in Section II, our algorithm (Perfect Hashing and Pruning - PHP) is explained in Section III, and the results of the performance analysis are discussed in Section IV. As the experimental results show, the proposed algorithm (PHP) demonstrates significantly better performance than Apriori and DHP algorithms, when the number of distinct items in the database is not too large.

II. DIRECT HASHING AND PRUNING (DHP) ALGORITHM

The Direct Hashing and Pruning (DHP) algorithm is in fact, a variation of Apriori algorithm. Both algorithms generate candidate *k*+1-itemsets from large *k*-itemsets, and large *k*+1-itemsets are found by counting the occurrences of candidate *k*+1-itemsets in the database. The difference of the DHP algorithm is that, it uses a hashing technique to filter out unnecessary itemsets for the generation of the next set of candidate itemsets [5].

In [9] it has been showed that, the initial candidate set generation, especially for the large 2-itemsets, is the key issue to improve the performance of data mining, since in each pass, the set of large *k*-itemsets (L_k) is used to form the set of candidate *k*+1-itemsets (C_{k+1}) by joining L_k with itself on *k*-1 common items for the next pass. In general, the more itemsets in C_{k+1}, the higher the processing cost of determining L_{k+1} will be. In Apriori algorithm, $|C_2| = \binom{|L_1|}{2}$, so the step of determining L_2 from C_2 by scanning the whole database and testing each transaction against C_2 is very expensive. By constructing a significantly smaller sized C_2, the DHP algorithm performs the counting of C_2 much faster than Apriori.

In the DHP algorithm, during the support count of C_k, by scanning the database, the algorithm also accumulates information about candidate *k*+1 itemsets in advance in such a way that, all possible *k*+1 subsets of items of each transaction after some pruning are hashed to a hash table. Each entry in the hash table consists of a number of itemsets that have been hashed to this entry thus far. Then, this hash table is used to determine the C_{k+1}. In order to find C_{k+1}, the algorithm generates all possible *k*+1 itemsets from L_k as in the case of Apriori, then the algorithm adds a *k*+1 itemset into C_{k+1} only if that *k*+1 itemset passes the hash filtering, i.e. that the entry for *k*+1

itemset in the hash table is greater than or equal to the minimum support. As it has been showed in [9], such hash filtering drastically reduces the size of C_{k+1}.

The DHP algorithm is as follows [5]:

Input: Database
Output: Frequent k-itemset

/* *Database = set of transactions;*
Items = set of items;
transaction = <TID, {x ∈ Items}>;
F_1 *is a set of frequent l-itemsets* */

$F_1 = \phi$;
/* H_2 *is the hash table for 2-itemsets*
Read the transactions, and count the occurrences of each item, and generate H_2 */

for each transaction $t \in$ Database do begin
  for each item x in t do
    $x.count$ + +;
  for each 2-itemset y in t do
    $H_2.add(y)$;
end
//*Form the set of frequent 1-itemsets*

for each item $i \in$ Items do
  if $i.count / |Database| \geq \min sup$
  then $F_1 = F_1 \cup i$;
end

/**Remove the hash values without the minimum support* */

$H_2.prune(\min sup)$;
/**Find* F_k *, the set of frequent k-itemsets, where* $k \geq 2$ */

for each $(k := 2; F_{k-1} \neq \phi; k++)$ do begin

  // C_k *is the set of candidate k-itemsets*
  $C_k = \phi$;
/* $F_{k-1} * F_{k-1}$ *is a natural join of* F_{k-1} *and* F_{k-1} *on the first k - 2 items*
H_k *is the hash table for k-itemsets* */

  for each $x \in \{F_{k-1} * F_{k-1}\}$ do
    if $H_k.hassupport(x)$
    then $C_k = C_k \cup x$;
  end
  /**Scan the transactions to count candidate k-itemsets and generate* H_{k+1} */

  for each transaction $t \in$ Database do begin
    for each k-itemset x in t do
      if $x \in C_k$
      then $x.count$ + +;
    for each $(k + 1)$-itemset y in t do
      if $\neg\exists z \mid z = k -$ subset of y
        $\wedge \neg H_k.hassupport(z)$
      then $H_{k+1}.add(y)$;
  end
  // F_k *is the set of frequent k-itemsets*

  $F_k = \phi$;
  for each $x \in C_k$ do
    if $x.count / |Database| \geq \min sup$
    then $F_k = F_k \cup x$;
  end

  /* *Remove the hash values without the minimum support from* H_{k+1} */

  $H_{k+1}.prune(\min sup)$;
end
Answer = $\cup_k F_k$;

Figure 1. The Direct Hashing and Pruning Algorithm

In the initial pass, while counting the occurrences of 1-itemsets, the occurrences of the hash values of the 2-itemsets in each transaction are also counted. Then the candidate itemsets are removed if their hash entries are less than the minimum support. A k+1 itemset in a transaction is added to the hash table H_{k+1} if the hash entries of all the k-subsets of the k+1 itemset have the minimum support in H_k. Also the DHP algorithm proposed in [9] prunes the transactions, which do not have any frequent items, from the database, and trims the non-frequent items from transactions at each step.

The efficiency of the DHP algorithm in reduction of the number of candidate itemsets depends on the number of *false positives* [5]. The false positives are generated when the hash values are identical for a group of candidate itemsets whose individual frequency is less than the minimum support, but their hash entry is greater than or equal to the minimum support. The number of candidate itemsets that have the same hash value is directly related with the size of the hash table. The drawback of the DHP algorithm is that, the hash table is in competition for memory space with the hash tree used to hold the counts for the itemsets.

Experiments performed in [5] and [9] have showed that as the size of the database grows, the DHP algorithm significantly outperforms the Apriori algorithm. However, the performance of the DHP algorithm highly depends on the hash table size.

III. PERFECT HASHING AND PRUNING (PHP) ALGORITHM

In the DHP algorithm, if we can define a large hash table such that each different itemsets is mapped to different locations in the hash table, then the entries of the hash table gives the actual count of each itemset in the database. In that case, we do not have any false positives and as a result of this, an extra processing for counting the occurrences of each itemset is eliminated.

In [9], it has also been showed that, the amount of data that has to be scanned during the large itemset discovery is another performance-related issue. Reducing the number of transactions to be scanned and trimming the number of items in each transaction improves the data mining efficiency in later stages.

The proposed algorithm uses perfect hashing for the hash table generated at each pass and also, reduces the size of the database by pruning the transactions that do not contain any frequent item. So we call the algorithm as Perfect Hashing and Pruning (PHP) and the algorithm is as follows: During the first pass of our algorithm, a hash table with size equal to the distinct items in the database is created. Each distinct item in the database is mapped to different location in the hash table, and this method is called as *perfect hashing*. The *add* method of the hash table adds a new entry if an entry for item *x* does not exist in the hash table and initializes its count to 1, otherwise it increments the count of *x* in the table by 1. After the first pass, the hash table contains the exact number of occurrences of each item in the database. By only making one pass over the hash table, which is in memory, the algorithm easily generates the frequent 1-itemsets. After that operation, the *prune* method of the hash table prunes all the entries whose support is less than the minimum support.

In the subsequent passes, the algorithm prunes the database by discarding the transactions, which have no items from frequent itemsets, and also trims the items that are not frequent from the transactions. At the same time, it generates candidate *k*-itemsets and counts the occurrences of *k*-itemsets. At the end of the pass, D_k contains the pruned database, H_k contains the occurrences of candidate *k*-itemsets, and F_k is the set of frequent *k*-itemsets. This process continues until no new F_k is found. The pseudo-code of our algorithm is provided in Figure 2.

This algorithm is apparently better than the DHP algorithm, since after forming the hash table, it does not need to count the occurrences of the candidate *k*- itemsets as in the case of the DHP algorithm. Also, the proposed algorithm performs better than the Apriori algorithm, since, at each iteration, the size of the database is reduced, and this provides high performance to the algorithm when the size of the database is huge and the number of frequent itemsets is relatively small.

```
Input: Database
Output: Frequent k-itemset
/* Database = set of transactions;
   Items = set of items;
   transaction = <TID, {x ∈ Items}>;
   F1 is a set of frequent l-itemsets */

F1 = φ ;

/* H1 is the hash table for 1-itemsets
   Read the transactions, and count the occurrences of
each item, and generate H1 */

for each transaction t ∈ Database do begin
    for each item x in t do
        H1.add(x);
end;

// Form the set of frequent 1-itemset

for each itemset y in H1 do
    if H1.hassupport(y)
    then F1 = F1 ∪ y
end

/* Remove the hash values without the minimum
   support */

H1.prune(min sup);
D1 = Database ;

// Dk is the pruned database

/* Find Fk , the set of frequent k-itemsets, where k ≥ 2
and prune the database */

k = 2;
repeat
    Dk = φ;
    Fk = φ;
    for each transaction t ∈ Dk−1 do begin
        // w is k-1 subset of items in t
        if ∀w | w ∉ Fk−1
        then skip t;
        else
            items = φ ;
            for each k -itemset y in t do
                if ¬∃z | z = k −1 subset of y
                    ∧ ¬Hk-1.hassupport(z)
                then Hk.add(y);
                items=items ∪ y;
            end
            Dk = Dk ∪ t    // such that t contains
                           // items only in the set items
    end

    for each itemset y in Hk do
        if Hk.hassupport(y)
        then Fk = Fk ∪ y
    end

    /* Remove the hash values without the minimum
       support from Hk */
    Hk.prune(min sup);
    k++;
until Fk−1 = φ;
Answer = ∪k Fk ;
```

Figure 2. The Perfect Hashing and Pruning Algorithm

Additionally, after each iteration, the database D_k contains transactions with frequent items only. The algorithm forms all *k*-subsets of items in each transaction and inserts the ones whose all *k*-1 subsets are large to the hash table. For that reason the algorithm does not miss any frequent itemset. Since the algorithm makes a pruning during the insertion of the candidate *k*-itemsets to the H_k, the size of the hash table is not large and fits into memory.

IV. EXPERIMENTAL RESULTS

The algorithm proposed in Section III, the PHP algorithm, is implemented in Perl5, and the hashing facility of Perl5 is used in order to implement the hash table. The algorithm is run on Sun workstation and over the sales record data obtained from Begendik Corporation. The dataset contains the transactions that are recorded for a week, and it consists of 11,512 transactions and around 5,000 different items. A larger dataset would yield more meaningful results but it was not possible to obtain a large real dataset because of the security reasons of Begendik Corporation.

Experimentation is done to compare our algorithm with Apriori. Given that our algorithm does not produce any false positives during the candidate itemset generation, it does not perform extra processing for counting the occurrences of each itemset. For that reason, our algorithm has less number of steps than the DHP algorithm, and we do not compare it with the DHP algorithm. In our experimentation, since we could not obtain the original implementation of the Apriori algorithm [3, 4], we used an implementation of Apriori algorithm in Perl. Experimental results are shown in Table 1 and Figure 3. Both Apriori and PHP algorithms are run over the same data set, and the frequent itemsets found by the two algorithms are the same. The number of frequent itemsets for different minimum supports is given in Table 1.

Table 1. Number of Frequent Itemsets for Different Support Values

Minimum support	$\|F_1\|$	$\|F_2\|$	$\|F_3\|$	$\|F_4\|$	**Total number of large itemsets**
2.0	61	10	1	0	72
1.5	93	25	3	0	121
1.0	148	76	17	0	243
0.5	364	327	121	23	835

As it can be seen from the table, the number of large itemsets is inversely proportional to the minimum support. When the minimum support is decreased, the number of large itemsets found increases as we expect.

The memory requirement of the algorithm depends only on the number of distinct items in the database and the minimum support, so it is independent from the size of the database. The memory requirement of the algorithm decreases as the minimum support increases. When the minimum support increases, the number of frequent itemsets decreases, and as a result of this, the size of the hash table generated decreases.

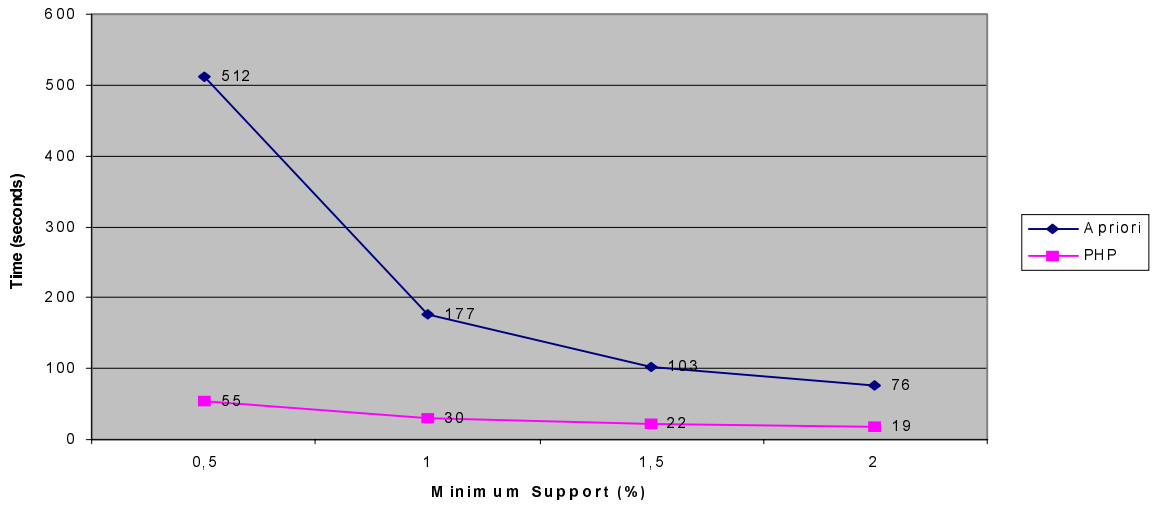

Figure 3. Performance of Apriori and PHP Algorithms

Figure 3 shows the running time comparison of the Apriori and the PHP algorithms. Figure 3 shows that our algorithm, for all minimum support values, performs better than the Apriori algorithm. As the minimum support decreases, the efficiency of our algorithm increases with respect to the Apriori algorithm, since PHP generates much smaller sized C_2 than that is generated by Apriori, and also at each step our algorithm searches on smaller sized (pruned) database and this increases run time efficiency significantly. The running time of Apriori algorithm drastically increases when the minimum support is decreased below 1.0%. For 2.0% minimum support, Apriori algorithm runs in 76 seconds, however when the minimum support decreased to 0.5%, the running time increases to 516 seconds, which is nearly 9 times slower than our algorithm. Also, as the minimum support decreases, the running time of our algorithm increases slowly.

V. CONCLUSIONS AND FUTURE WORK

In this work, we studied the problem of finding frequent itemsets for association rule mining. An algorithm called Direct Hashing and Pruning (DHP) is discussed in detail, and by using the ideas in the DHP algorithm, we propose a new algorithm PHP that employs the hashing facility of Perl5 in order to keep the actual count of occurrences of each candidate itemset of the transaction database. The proposed algorithm also prunes the transactions, which do not contain any frequent items, and trims the non-frequent items from the transactions at each step. Since our algorithm has less number of steps than the DHP algorithm, we did not compare the performance of these two algorithms. In order to test the performance of our algorithm, we compared it against an implementation of Apriori algorithm over the real dataset that was obtained from the Begendik Corporation. As the experimentation has showed, our algorithm performs better than the Apriori algorithm since at each step it reduces the database size to be scanned, and it generates much smaller sized C_2 at the initial step. As future work, our algorithm may be run over larger sets of data, and experimentation on memory requirement of the algorithm may be performed.

REFERENCES

1. H. Mannila, H. Toivonen, and A. I. Verkamo, “Efficient Algorithms for Discovering Association Rules”, Proceedings of the AAAI Workshop on Knowledge Discovery in Databases, Usama M. Fayyad and Ramasamy Uthurusamy (Eds.), Washington, pp. 181-192, (July 1994).
2. R. Srikant and R. Agrawal, “ Mining Generalized Association Rules”, Proc. of the 21st VLDB Conference, Zurich, Switzerland, (1995).
3. R. Agrawal, T. Imielinski, and A. Swami, “Mining Association Rules between Sets of Items in Large Databases”, Proc. of the ACM-SIGMOD 1993 Int'l Conference on Management of Data, Washington D.C., pp. 207-216, (May 1993).
4. R. Agrawal, and R. Srikant, “Fast Algorithms for Mining Association Rules”, Proc. of the 20th Int'l Conference on Very Large Databases, Santiago, Chile, (Sept. 1994).
5. J. D. Holt, and S. M. Chung, “Efficient Mining of Association Rules in Text Databases” CIKM'99, Kansas City, USA, pp. 234-242, (Nov. 1999).
6. B. Mobasher, N. Jain, E.-H. Han, and J. Srivastava, “Web Mining: Pattern Discovery from World Wide Web Transactions” Department of Computer Science, University of Minnesota, Technical Report TR96-050, (March, 1996).
7. C. Ordonez, and E. Omiecinski, “Discovering Association Rules Based on Image Content” IEEE Advances in Digital Libraries (ADL'99), (1999).
8. M. Houtsma and A. Swami, “Set-Oriented Mining of Association Rules”, Research Report RJ 9567, IBM Almaden Research Center, San Jose, California, (Oct. 1993).

9. J. S. Park, M. S. Chen and P. S. Yu, “Using a Hash-Based Method with Transaction Trimming for Mining Association Rules”, IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No. 5, (Sept./Oct. 1997).
10. A. Savasere, E. Omiecincki, and S. Navathe, “An Efficient Algorithm for Mining Association Rules in Large Databases”, Proc. of the 21st VLDB Conf., pp. 432-444, (1995).
11. E. M. Voorhees and D. K. Harmon (editors), The Fifth Text Retrieval Conference, National Institute of Standards and Technology, (1997).
12. M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, “New Algorithms for Fast Discovery of Association Rules”, Technical Report 651, Computer Science Department, University of Rochester, (July 1997).