Mining Frequent Itemsets in Large Data Warehouses: A Novel Approach Proposed for Sparse Data Sets

S.M. Fakhrahmad[1], M. Zolghadri Jahromi[2], M.H. Sadreddini[3]

[1] Faculty member in Department of Computer Eng., Islamic Azad University of Shiraz and PhD student in Shiraz University, Shiraz, Iran
[2, 3] Department of Computer Science &Engineering, Shiraz University, Shiraz, Iran
mfakhrahmad@cse.shirazu.ac.ir ,{zjahromi , sadredin}@shirazu.ac.ir

Abstract. Proposing efficient techniques for discovery of useful information and valuable knowledge from very large databases and data warehouses has attracted the attention of many researchers in the field of data mining. The well-known Association Rule Mining (ARM) algorithm, Apriori, searches for frequent itemsets (i.e., set of items with an acceptable support) by scanning the whole database repeatedly to count the frequency of each candidate itemset. Most of the methods proposed to improve the efficiency of the Apriori algorithm attempt to count the frequency of each itemset without re-scanning the database. However, these methods rarely propose any solution to reduce the complexity of the inevitable enumerations that are inherited within the problem. In this paper, we propose a new algorithm for mining frequent itemsets and also association rules. The algorithm computes the frequency of itemsets in an efficient manner. Only a single scan of the database is required in this algorithm. The data is encoded into a compressed form and stored in main memory within a suitable data structure. The proposed algorithm works in an iterative manner, and in each iteration, the time required to measure the frequency of an itemset is reduced further (i.e., checking the frequency of n-dimensional candidate itemsets is much faster than those of n-1 dimensions). The efficiency of our algorithm is evaluated using artificial and real-life datasets. Experimental results indicate that our algorithm is more efficient than existing algorithms.

Keywords: Data Mining, Frequent Itemset, Association Rule Mining, Transactional Database, Logical Operations

1 Introduction

Mining association rules (ARs) is a popular and well researched field in data mining for discovery of interesting relations between items in large databases and transaction warehouses. Their most popular applications include market basket data analysis, cross-marketing, catalog design, information retrieval, clustering and classification [1,2,3].

ARs are represented in the general form of $X \rightarrow Y$ and imply a co-occurrence relation between X and Y, where X and Y are two sets of items (called itemsets). X and Y are called antecedent (left-hand-side or LHS) and consequent (right-hand-side

or RHS) of the rule, respectively. Many evaluation measures are defined to select interesting rules from the set of all possible candidate rules. The most widely used measures for this purpose are minimum thresholds on support and confidence.

In most cases, we are just interested in ARs involving itemsets that appear frequently. For example, we cannot run a good marketing strategy involving items that are infrequently bought. Thus, most of mining methods assume that we only care about set of items that appear together in at least an acceptable percentage of the transactions, i.e., the minimum *support* threshold. The *support* of an itemset X is defined as the proportion of transactions in the data set containing X. The term *frequent itemset* is used for itemsets with high value of support.

The confidence of a rule $X \rightarrow Y$ is defined as $supp(X \cap Y)/supp(X)$, i.e., a fraction of transactions containing X, which contain Y as well. ARs must satisfy a minimum degree of support and confidence at the same time. In this paper, we use the short terms *MinSupp* and *MinConf* for minimum support and minimum confidence thresholds, respectively.

Most association rule mining (ARM) algorithms generate association rules in two steps: (1) Mining all frequent itemsets, and (2) generating all rules using these itemsets. The base of such algorithms is the fact that any subset of a frequent itemset must also be frequent, and that both the LHS and the RHS of a frequent rule must also be frequent. Thus, every frequent itemset of length *n* can result in *n* association rules with a single item on the RHS [4,5,6,10,11].

In data mining applications, the data is often too large to fit in main memory. Therefore, the first step of mining ARs is expensive in terms of computation, memory usage and I/O resources. Much of the research effort in this field has been devoted to improving the efficiency of the first step. The main factors used to evaluate these algorithms are the time needed to read data from disk and the number of times each data item has to be read. There are also some approaches, which consider the memory usage as the main factor to be minimized.

Different algorithms use some key principles and tricks to mine frequent itemsets more efficiently. Most of these algorithms try to present a solution to the problem of finding frequent itemsets by reducing the number of times the whole database has to be scanned (i.e., reduce the number of times that the occurrences of itemsets has to be counted). In the literature, many efficient solutions have already been proposed for this problem. However, one key issue, which has rarely been addressed by other researchers in this field is how to compute the frequency of itemsets in an efficient manner. Finding the frequency of an itemset is carried out by counting the number of occurrences of the itemset, which is a very time consuming process due to the large volume of data in data mining applications. In this paper, we focus our attention on how to present an efficient solution for this problem. In our approach, the database is scanned only once and the data is encoded into a compressed form and stored in main memory within a suitable data structure. The proposed algorithm works in an iterative manner, where by each iteration, the time required to measure the frequency of itemsets, is reduced further.

The rest of the paper is organized as follows: Section 2 introduces some efficient ARM algorithms from the literature. In Section 3, we describe our approach and give the detail of the algorithm, *FastARM*. Experimental results using artificial and real-life data sets are presented in Section 4. Finally, we give the conclusion in Section 5.

2 Related Work

Many algorithms that have already been proposed for ARM, use a two step process for generating ARs: 1) mining frequent itemsets, 2) generating ARs from frequent itemsets. The main focus of many of these proposed algorithms is over the first step, where they try to improve the efficiency of the mining process for finding frequent itemsets by reducing the number of read operations from disk, as much as possible. For this purpose, some methods propose solutions to compute the support of some itemsets in order to avoid a number of unnecessary data re-scans. Some others build a special data structures in main memory for this purpose.

Apriori [4] is the most well-known ARM method. The concepts and principles of this method are the basis of many other proposed algorithms. Many improved versions or efficient implementations of the primary Apriori have also been proposed by different researchers. VIPER [5] and ARMOR[1] [6] are two relatively new algorithms which use the Apriori approach, but are much more efficient. VIPER uses a similar data presentation to our proposed method, but it is not efficient because of its need for multiple data re-scans. ARMOR can be considered as the improved version of another efficient algorithm, Oracle [6]. Oracle and ARMOR use a data structure called DAG to optimize their counting operations of itemset occurrences. FP-Growth [7] is another well-known algorithm, which works differently from others. It discovers frequent itemsets without generating any candidate itemset. In this algorithm, the data is read three times from disk and a hash tree structure is built in memory. All frequent itemsets can be found by traversing the hash tree. The main problem of FP-Growth is its heavy utilization of main memory, which is very dependent on the size of database. Running this algorithm for huge data sets is almost impossible due to the limitation of main memory.

The major problems of many ARM methods are their need to read data from the disk iteratively and the time consuming operation of counting the frequency of each itemset [4,5,6,8,10]. The method proposed in this paper attempts to provide a solution for these problems.

3 The Proposed Algorithm

For ease of illustration, we assume the transaction data warehouse as a binary-valued data set having a relational scheme. Each column in this scheme stands for a possible item that can be found in any transaction of the data warehouse and each tuple represents a transaction. Each 0 or 1 value indicates the presence or absence of an itemset in a transaction, respectively. As an example the relation shown in Fig. 1.(b) is the structured form of the data set of Fig. 1.(a), which contains four transactions.

[1] Association Rule Mining based on ORacle

	Cheese	*Coke*	*Egg*	*Beer*
Cheese, Coke, Egg	1	1	1	0
Cheese, Egg	1	0	1	0
Coke, Cheese, Beer	1	1	0	1
Coke, Beer	0	1	0	1

a) A transactional data set b) A structured presentation of transactions

Fig. 1. A data set containing some transactions of a market basket

3.1 Mining Frequent Itemsets

As the first step of the algorithm, we divide the relation horizontally into some equi-size partitions, each containing *k* tuples. We comment on choosing the best value for *k* later in this section. In this relation, each column contains *k* bits in each partition, thus the group of bits in each column within each partition can be viewed as a *k*-bit binary code, which is equivalent to a decimal number between 0 and 2^k-1. These decimal numbers are the major elements of our algorithm.

The partitioned relation is scanned just once and the supports of singletons (1-itemsets) are measured to find 1-frequent itemsets. Meanwhile, for each partition, all nonzero decimal values are extracted. For any column of the data set, which represents a frequent singleton, we build a hash table in memory. Each value in this hash table, is a non-zero decimal value extracted from a partition and its access key is the number of that partition (an integer number between 1 and *m*, where *m* is the number of partitions). Since we do not insert zero values into the hash tables, then the values recorded in the hash table indicate the regions of the itemset occurrences and limits the search space for the next steps.

	A	*B*	*C*
I	1	0	0
	0	1	0
	0	1	0
	1	1	0
II	0	0	1
	1	0	0
	1	0	0
	1	0	0
III	1	0	1
	1	0	1
	1	0	0
	0	0	1
IV	0	1	1
	0	1	0
	0	0	1
	0	1	0
V	1	0	0
	0	0	1
	1	0	0
	1	0	0
VI	0	1	1
	0	1	0
	0	1	0
	0	1	0

Fig. 2. An Example data set

As an example, consider the data set *r* with 24 transactions, shown in Fig. 2, where A, B and C are three different items. Assume the value of k is set to 4. Thus, the 24 transactions are divided into 6 partitions, each containing 4 tuples. The proposed algorithm with MinSupp set to 0.4, searches for frequent itemsets as follows. The first step involves counting the occurrences of all singletons and constructing the hash tables for the frequent ones. The calculated values for the supports of *A*,*B* and *C* are 0.45, 0.41 and 0.33, respectively.

Here, only the hash tables of *A* and *B* (frequent singletons) are constructed. The hash table for *C* is not constructed because its support is less than the *MinSupp* threshold.

Hash table of *A*:

Keys	I	II	III	V
Values	9	7	14	11

$(9 = (1001)_2, 7 = (0111)_2, 14 = (1110)_2, 11 = (1011)_2)$

Hash table of *B*:

Keys	I	IV	VI
Values	7	13	15

$(7 = (111)_2, 13 = (1101)_2, 15 = (1111)_2)$

The support of a compound itemset such as AB, is easily measured by using the hash tables of its elements (i.e., A and B), instead of scanning the whole database again. In order to calculate the support of a compound itemset, we begin with the smaller hash table (i.e., the one having fewer values). For each key of this hash table, we first verify if it also exists in the other hash table. This verification does not involve any search due to the direct access structure of hash table. If a key exists in both hash tables, then we perform a logical *AND* operation between the the corresponding values related to that key.

The result of the *AND* operation is another integer value, which gives the co-occurrences of A and B in that partition. If the result is zero, it means that there is no simultaneous occurrence of A and B in that partition. We build a similar hash table for the compound itemset, AB, and insert the non-zero integer values resulted from *AND* operations in this table. The size of this hash table is at most equal to the size of the smaller hash table of the two elements. Each number stored in this hash table is equivalent to a binary number, which contains some 1's. The total number of 1's indicates the co-occurrence frequency of A and B. Thus we should just enumerate the total number of 1's for all integer values, instead of scanning the whole database. This measurement can be done using logical Shift Left (*SHL*) or Shift Right (*SHR*) operations over each value and adding up the carry bits until the result is zero (i.e., there is no other 1-bits to be counted).

The *SHR* operation is preferred to *SHL* in cases where the decimal number under investigation has a value less than $2^{k/2}$. The reason is that the equivalent binary codes for such cases do not contain any 1-bit in their left-hand side half, and selecting *SHR* will make enumeration at least two times faster than using *SHL*.

The efficiency of this structure becomes clearer for measuring the support of higher dimensional itemsets. As we proceed to higher dimensional itemsets, the size of hash tables becomes smaller due to new zeros emerging from *AND* operations. These zeros are not inserted into the result hash table.

Let us refer to the above example and continue the mining process. According to the frequent singletons found, the only candidate for 2-frequent itemsets (pairs) is the itemset *AB*. In order to build the hash table of *AB*, each value stored in the hash table of *B* (i.e., the smaller hash table) is selected for logical *AND* operation with a value having the same key stored in *A*'s hash table. The only key present in both hash tables is I, thus the result is a hash table having just one item, as follows.

Hash table of *AB*: (9 & 7 = 1)

Keys	I
Values	1

To measure the support of *AB*, the number of 1's in the value field of this hash table (in the binary form) has to be counted. Since this value is equal to 1 (i.e., 0001), just one *SHR* operation and thus one comparison is enough to count 1's. However, if we had searched all the data to find the co-occurrences of A and B, the number of required comparisons would have been 48 (for reading the value of A and B in all 24 tuples). In general, this improvement is much more apparent for itemsets of higher dimensions.

In a same way, the hash tables of 2-frequent itemsets are then used to mine 3-frequent itemsets and in general, n-frequent itemsets are mined using (n-1)-frequent itemsets. However, we do not use all combinations of frequent itemsets to get (n+1)-frequent itemsets. The Apriori principle [4] is used to avoid verifying useless combinations: "An n-dimensional itemset can be frequent if all of its (n-1)-dimensional subsets are frequent". Thus, for example if AB and AC are two frequent itemsets, their combination is ABC, but we do not combine their hash tables unless the itemset BC is also frequent. If all the n-1 subsets of an n-dimensional itemset are frequent, combining two of them is enough to get the hash table of the itemset.

4 Experimental Results

We conducted two experiments to evaluate the performance of our algorithm, *FastARM* in comparison with four well-known ARM methods, *Apriori*, *VIPER*, *ARMOR* and *FP-Growth*. We implemented the algorithms in C++ on a 3GHz Intel system with 1 GB RAM. We performed experiments on synthetic and real-life data. In all of the experiments we used $k = 32$ for the size of partitions.

4.1 Experiment 1: Synthetic Data

We used 10 data sets each containing $2*10^6$ transactions in this experiment to evaluate the performance of *Apriori*, *VIPER*, *ARMOR* and *FastARM* algorithms. We generated synthetic data sets randomly for 500 distinct items such that the probability of an item being presenct in a transaction is 0.1. In this experiment, we could not evaluate the performance of *FP-Growth* due to its heavy utilization of main memory. The reason for this is that *FP-Growth* stores the database in a condensed form in main memory (using a data structure called FP-tree).

The results are shown in Fig. 3. The x-axis in these graphs represents the MinSupp threshold values and the y-axis represents the run times of different algorithms. For each specified value of MinSupp, the average run time of each algorithm over 10 data sets is measured and displayed. In this graph, we observe that the execution time of *FastARM* is relatively less than all of the other algorithms. This relative efficiency is more sensible where the value of MinSupp is very low. We also see that there is a considerable improvement in the performance of *FastARM* with respect to both *Apriori* and *VIPER* and also a relative improvement with respect to *ARMOR*.

Table 1 shows the memory consumption of the algorithm throughout each part of the experiment. The values shown in this table represent the amount of memory required for hash tables in each case. Since the hash tables contain the whole information of the primary database (in another format), we can find out the compression rate of the algorithm by comparing these values with the size of database (which is about 100 MB).

Table 1. The memory consumption of the algorithm through each part of the experiment

Probability of 1-bit	Consumed Memory (MB)
0.005	6.8
0.01	11.3
0.05	47.6
0.1	88.7

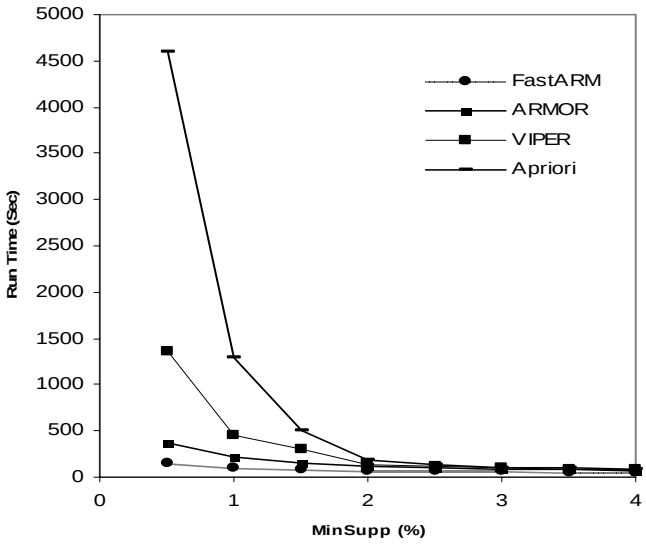


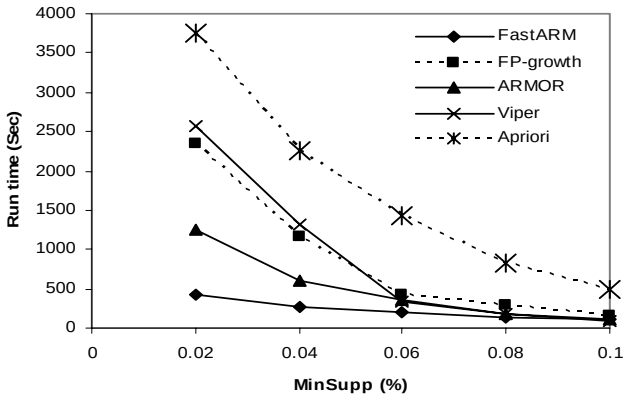
Fig. 3. Performance of different methods on synthetic data for different Minsupp values

4. 2 Experiment 2: Real Databases

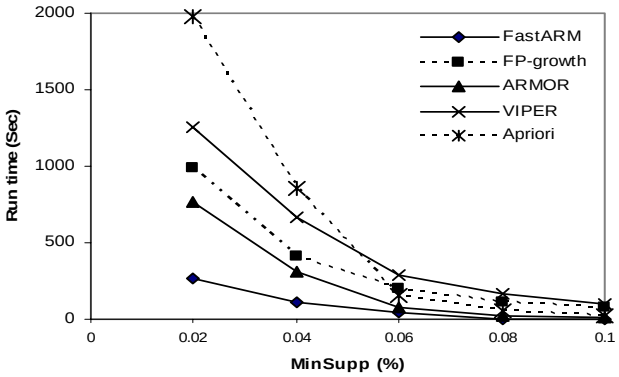
Our second set of experiments involved real data sets extracted from the Frequent Itemset Mining Dataset Repository, namely BMS-POS, BMS-WebView-1 and BMS-WebView-2.

The BMS-POS dataset contains sales data of several years from a large electronics retailer. Since this retailer has so many different products, product categories are used as items. Each transaction in this dataset is a customer's purchase transaction consisting of all product categories purchased at one time. The goal for this dataset is to find associations between product categories purchased by customers in a single visit to the retailer. This data set contains 515,597 transactions and 1,657 distinct items. The BMS-WebView-1 and BMS-WebView-2 datasets contain several months worth of clickstream data from two e-commerce web sites. Each transaction in these data sets is a web session consisting of all the product detail pages viewed in that session. That is, each product detail view is an item. The goal for both of these datasets is to find associations between products viewed by visitors in a single visit to the web site. These two data sets contain 59,602 and 77,512 transactions, respectively (with 497 and 3,340 distinct items).

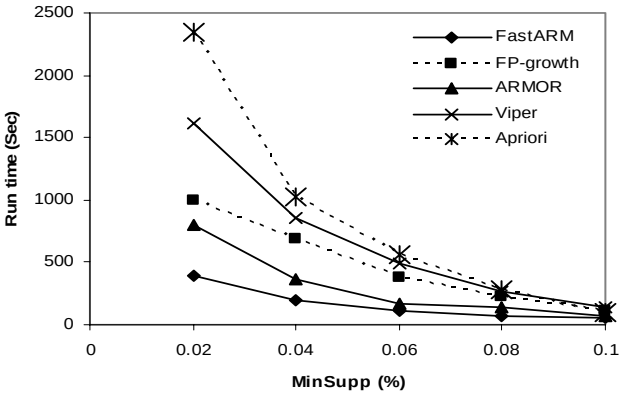
We set the MinConf threshold value to zero and evaluated the performance of different algorithms using the MinSupp value varying within the range of (0.02%–0.1%). The results of these experiments are shown in Figures 4a–c. We see in these graphs that for lower values of MinSupp, the performance of *FastARM* is significantly better than other methods.



a) Running times of various methods on the BMS-POS data set



b) Running times of various methods on the BMS-WebView-1 data set



c) Running times of various methods on the BMS-WebView-2 data set

Fig. 4. Performance of algorithms over some real-life data sets

5 Conclusion

In this paper, we proposed an efficient ARM algorithm called the *FastARM* that partitions the data and constructs hash tables to count the frequency of itemsets. Only a single scan of the database is required in this approach and all the necessary information is stored in hash tables. Frequent itemsets are computed by performing the logical AND operations on values from individual hash tables.

We used two experiments on artificial and real-life data sets to evaluate the run time of *FastARM* in comparison with *Apriori*, *FP-Growth, VIPER* and *ARMOR* as four well-known ARM algorithms proposed in the literature. The experiments were conducted to investigate the effect of MinSupp and the database size on the execution time of each algorithm. The results of these experiments clearly indicated that *FastARM* performs better specially for lower values of MinSupp. It should be noticed that as we increase the value of MinSupp, the number of frequent itemsets and generated ARs decreases rapidly. That is why *FastARM* performs similar to the other methods when higher values for MinSupp are used.

References

1. Zamiri, M. J. and A.A. Rezaei- Roodbari.: Relationship between blood physiological attributes and carcass characteristics in Iranian fat-tailed sheep, Iranian Journal of Science and Technology, Transactions A, Vol. 28, No. A, 97--06 (2004)
2. Ghassem-Sani, G. and Halavati, R.: Employing Domain Knowledge to Improve AI Planning Efficiency, Iranian Journal of Science and Technology, Transaction B, Vol. 29, No. B1 (2005)
3. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases, In: 20th International Conference on Very Large Data Bases, pp. 487--499 (1994)
4. Shenoy, P., Haritsa J., Sudarshan S., Bhalotia G., Bawa M., and Shah D.: Turbo-charging vertical mining of large databases. In: ACM SIGMOD Intl. Conf. on Management of Data, ACM Press, Vol. 29 , No. 2, pp. 22--33, 2000.
5. Pudi, V., Haritsa, J.R..: ARMOR: Association Rule Mining based on ORacle. In: ICDM Workshop on Frequent Itemset Mining Implementations, Florida, USA (2003)
6. Han, J., Pei, J., Yin Y., and Mao R.: Mining frequent patterns without candidate generation: A frequent –pattern tree approach. Data Minning and Knowledge Discovery, Vol. 8, No. 1, pp. 53—87 (2004)
7. Zheng, Z., Kohavi, R. and Mason L.: Real world performance of association rule algorithms. In: Intl. Conf. on Knowledge Discovery and Data Mining (KDD) (2001)
8. Michie, D., Spiegelhalter, D.J., and Taylor, C.C., (eds.). Machine Learning, Neural and Statistical Classification (STATLOG Project), Herfordshire: Ellis Horwood.
9. Pei, J., Han, J., and Mao, R.: CLOSET. An efficient algorithm for mining frequent closed itemsets. In: ACM_SIGMOD International Workshop on Data Mining and Knowledge Discovery (2003)
10. Webb., G.I.: OPUS: An efficient admissible algorithm for unordered search. JAIR, Vol. 3, 431—465 (2004)
11. Webb., G.I., Efficient search for association rules. In: Sixth ACM-SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY: ACM, pp. 99--107.