Improved Implementation and Performance Analysis of Association Rule Mining in Large Databases

H.R. Nagesh, M. Bharath Kumar, and B. Ravinarayana

Dept. of Computer Science & Engineering, MITE, Mangalore, India
{hodcse,bharath,ravinarayana}@mite.ac.in

Abstract. Data mining is the process of extracting interesting and previously unknown patterns and correlations from data stored in Data Base Management Systems (DBMSs). Association Rule Mining is the process of discovering items, which tend to occur together in transactions. Efficient algorithms to mine frequent patterns are crucial to many tasks in data mining. The task of mining association rules consists of two main steps. The first involves finding the set of all frequent itemsets. The second step involves testing and generating all high confidence rules among itemsets. Our paper deals with obtaining both the frequent itemsets as well as generating association rules among them.

In this paper we implement the FORC (Fully Organized Candidate Generation) algorithm, which is a constituent of the Viper algorithm for generating our candidates and subsequently our frequent itemsets. Our implementation is an improvement over Apriori, the most common algorithm used for frequent item set mining.

1 Introduction

The quantity of data to be handled by the file systems and the databases is growing at an exponential rate. Thus, users who need to access them need sophisticated information from them. The task of data mining satisfies them. To use a simple analogy, it's finding the proverbial needle in the haystack. In this case, the needle is that single piece of intelligence your business needs and the haystack is the large data warehouse you've built up over a long period of time. It is simply defined as finding the hidden information in a database. It can also be called exploratory data analysis or data driven analysis.

In essence, data mining is distinguished by the fact that it is aimed at the discovery of information, without a previously formulated hypothesis. The information discovered must be previously unknown. The new information has to be valid. Most critical, the information has to be actionable, that is, it must be possible to translate it into some business advantage and take action towards that advantage. It is "a decision support process in which we search for patterns of information in data". This search may be done just by the user, i.e. just by performing queries, or by sophisticated statistical analysis and modeling techniques, which uncover patterns and relationships hidden in organizational databases. Once found, the information needs to be presented in a suitable form, with graphs, reports, etc.

S. Unnikrishnan, S. Surve, and D. Bhoir (Eds.): ICAC3 2013, CCIS 361, pp. 94–104, 2013.

One of the reasons behind maintaining any database is to enable the user to find interesting patterns and trends in the data. The goal of database mining is to automate this process of finding interesting patterns and trends. Once this information is available, we can perhaps get rid of the original database. This goal is difficult to achieve due to the vagueness associated with the term `interesting'. The solution is to define various types of trends and to look for only those trends in the database. One such type constitutes the association rule.

Association rules attempts to predict the class given a set of conditions on the LHS. The conditions are typically attribute value pairs, also referred to as item-sets. The prediction associated with the RHS of the rule is not confined to a single class attribute; instead it can be associated with one or more attribute combinations.

Association rule induction is a powerful method for so-called *market basket analysis*, which aims at finding regularities in the shopping behavior of customers of supermarkets, mail-order companies and the like. With the induction of association rules one tries to find sets of products that are frequently bought together, so that from the presence of certain products in a shopping cart one can infer (with a high probability) that certain other products present. Such information, expressed in the form of rules, can often be used to increase the number of items sold, for instance, by appropriately arranging the products in the shelves of a supermarket (they may, for example, be placed adjacent to each other in order to invite even more customers to buy them together) or by directly suggesting items to a customer, which may be of interest for him/her.

1.1 Problem Statement

Given a database D of T transactions and a minimum support ‘s’, the problem is to find all the frequent itemsets (i.e. those itemsets whose support is greater than ‘s’). Once the frequent itemsets are found, association rules can be derived from them using confidence ‘c’.

2 Basic Concepts and Terminologies

Association rules are extracted from the database, which can be viewed as a collection of **transactions,** keeping in mind the grocery store cash register. For example, a transaction could be {Pencil, Sharpener, Eraser}, each element in it is termed as an **item.** We are not interested in quantity or cost, but only in their purchase. A collection of items is called as the **itemset**.

Here, we provide some basic concepts and terminologies used in the association rule mining.

1. Given a set of items $I = \{I_1, I_2, \dots, I_m\}$ and a database of transactions $D = \{t_1, t_2, \dots, t_n\}$ where each t_i contains different combinations of I, an **association rule** is an implication of the form $X \Rightarrow Y$ where $X, Y \in I$ are sets of items called itemsets and $X \cap Y = \phi$.

2. The **support** (s) for an association rule X=>Y is the percentage of transactions in the database that contain $X \cup Y$.

   Let T be the set of all transactions under consideration, e.g., let T be the set of all "baskets" or "carts" of products bought by the customers of a supermarket - on a given day if you like. The support of an item set S is the percentage of those transactions in T which contain S. In the supermarket example this is the number of "baskets" that contain a given set S of products, for example S = {bread, wine, cheese}. If U is the set of all transactions that contain all items in S, then

   $$\text{support}(S) = (|U| / |T|) *100\%,$$

   where |U| and |T| are the number of elements in U and T, respectively. For example, if a customer buys the set X = {milk, bread, apples, wine, sausages, cheese, onions, potatoes } then S is obviously a subset of X, hence S is in U. If there are 318 customers and 242 of them buy such a set U or a similar one that contains S, then support(S) = 76.1%.
3. The **confidence** or **strength** for an association rule X=>Y is the ratio of the number of transactions that contain $X \cup Y$ to the number of transactions that contain X.

   The confidence of a rule R="A and B->C" is the support of the set of all items that appear in the rule divided by the support of the antecedent of the rule, i.e. confidence(R) = (support({A, B, C})/support({A,B})) *100%.

   More intuitively, the confidence of a rule is the number of cases in which the rule is correct relative to the number of cases in which it is applicable. For example, let R="wine and bread -> cheese". If a customer buys wine and bread, then the rule is applicable and it says that he/she can be expected to buy cheese. If he/she does not buy wine or does not buy bread or buys neither, than the rule is not applicable and thus (obviously) does not say anything about this customer.

   If the rule is applicable, it says that the customer can be expected to buy cheese. But he/she may or may not buy cheese, that is, the rule may or may not be correct. We are interested in how good the rule is, i.e., how often its prediction that the customer buys cheese is correct. The rule confidence measures this: It states the percentage of cases in which the rule is correct. It computes the percentage relative to the number of cases in which the antecedent holds, since these are the cases in which the rule makes a prediction that can be true or false. If the antecedent does not hold, then the rule does not make a prediction, so these cases are excluded.

   With this measure a rule is selected if its confidence exceeds or is equal to a given lower limit. That is, we look for rules that have a high probability of being true, i.e., we look for "good" rules, which make correct (or very often correct) predictions.
4. If the support of an itemset is greater than or equal to a given support threshold, then it is termed a **frequent itemset,** otherwise it is infrequent.
5. Given a set of items $I = \{I_1, I_2, \ldots, I_m\}$ and a database of transactions $D = \{t_1, t_2, \ldots, t_n\}$ where each t_i contains different combinations of I, the **association rule problem** is to identify all association rules X => Y with a minimum support and confidence. These values are fed as inputs to the problem.

2.1 Data Layout Alternatives

Conceptually, a market basket database is a two dimensional matrix where the rows represent individual customer purchase transactions and the columns represent the items on sale. The implementation of this matrix is as shown in the Fig 1.

Horizontal Item Vector(HIV). The database is organized as a set of rows with each row storing a transaction identifier (tid) and a bit vector of 1's and 0's to represent for each of the items on sale, its presence or absence, respectively in the transaction is as shown in Fig. 1 a [1].

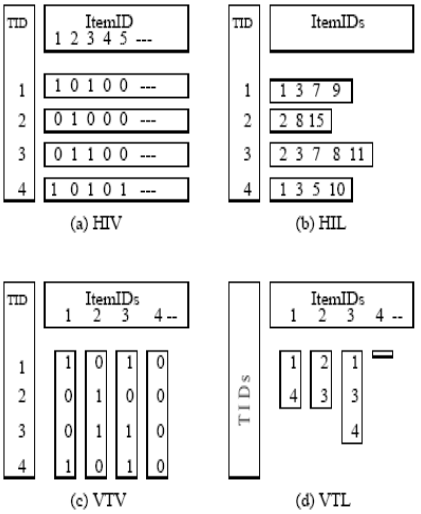

Fig. 1. Comparison of Data Layouts [1]

Horizontal Item List (HIL). This is similar to HIV, except that each row stores an ordered list of item identifiers(iid), representing only the items actually purchased in the transaction is as shown in Fig.1 b [1].

Vertical Tid Vector (VTV). The database is organized as a set of columns with each column storing an IID and a bit vector of 1's and 0's to represent the presence or absence, respectively of the item in the customer transactions is as shown in Fig 1 c [1].

Vertical Tid List (VTL). This is similar to VTV, except that each column stores an ordered list of only the tid's of the transactions in which the item was purchased is as shown in Fig 1 d [1].

2.2 The Viper Algorithm

VIPER (Vertical Itemset Partitioning for Efficient Rule Extraction) uses the vertical tid vector (VTV) format for representing an items' occurrence in the tuples of the database. The bit vector is stored in a compressed form called the 'snake'. It is a

multipass algorithm, wherein data in the form of snakes is read from and written to the disk in each pass [1]. It proceed in a bottom up manner and at the end of the data mining, the supports of all frequent itemsets are available. Each pass involves the simultaneous counting of several levels of candidates via intersections of the input snakes. To minimize the computational costs, VIPER implements a new candidate generation scheme called **FORC** (Fully organized candidate generation), partly based on the technique of equivalence class clustering. VIPER also incorporates a novel snake intersection and counting scheme called FANGS (Fast Anding Graph for Snakes). The FANGS scheme is based on a simple DAG structure that has a small footprint and efficiently supports concurrent intersection of multiple snake pairs by using a pipelined strategy [1].

Our implementation is a hybrid version wherein we generate the candidate itemsets using the FORC procedure of the VIPER algorithm, the frequent itemsets using the conventional Apriori algorithm and the association rules using the Association rule generation algorithm described before.

3 Experiments and Results

3.1 Data Generation and Formatting

Here, we generate the sample data that would be used later for testing the execution of the algorithm. The data generated is stored in a file in the form of bit vectors, i.e. 0/1 indicating the presence/absence respectively of an item in the given transaction. We have the columns depicting the items and the rows depict the transactions. This is called as a Transaction Vector, as specified earlier in the literature. If required, this data can be compressed with the help of Run Length Encoding.

3.2 The FORC Candidate Generation Algorithm

We use the algorithm called FORC (Fully Organized Candidate Generation) for efficiently generating candidate itemsets. FORC is based on the powerful technique of equivalence class clustering, but adds important new optimizations of its own.

The FORC algorithm operates as follows: Given a set S_k (which can be either a set of frequent itemsets or a set of candidate itemsets) from which to generate $C_{k+1,}$ the itemsets in S_k are first grouped into clusters called "**equivalence classes**". The grouping criterion is that all itemsets in an equivalence class should share a common prefix of length k-1. For each class, its prefix is stored in a hash table and the last element of every itemset in the class is stored in a lexicographically ordered list, called the *extList.* With this framework, the following is a straightforward method of generating candidates: For each prefix in the hash table, take the union of the prefix with all ordered pairs of items from the extList of the class (the order ensures that duplicates are not generated). For each of the potential candidates, check whether all its k-subsets are present in S_k, the necessary condition for an itemset to be a candidate. This searching is simple since the k-1 prefix of the subset that is being searched for indicates which extlist is to be searched. Finally, include those, which survive the test in C_{k+1} [1].

The FORC Algorithm under discussion is as listed below.

```
SetOfItemsets FORC(S_k) {
   Input: Set of k-itemsets(S_k)
   Output: Set of candidate k+1-itemsets(C_k+1)
   for each itemset i in S_k do
         insert (i.prefix) into hashTable;
         insert(i.lastelement) into i.prefix→extList;
         C_k+1 = ø;
         for each prefix P in the hashTable do {
         E = P→extList;
         for each element t in E do {
                  newP = P ∪ t ;
                  remList = {i|i ∪ E and i > t };
                  for each (k -1) subset subP of newP do
                  remList = remList ∩ (subP → extList);
         for each element q in remList do {
         newCand = newP ∪ {q};
         C_k+1 = C_k+1 ∪ newCand;
         }
         }
   }
   return C_k+1;
}
```

Fig. 2. FORC Algorithm [1]

3.3 Simultaneous Search Optimization

We can optimize the above-mentioned process by recognizing the fact that since the unions are taken with ordered pairs, the prefix of the subsets of the candidates thus formed will not depend on the second extension item, which in turn means that all these subsets are shared and the same for each element in the extlist. Hence, repeated searches for the same subsets can be avoided and they can be searched for simultaneously, as shown in the following example. Consider a set S3 in which we have the items ABC, ABD, ABE with prefix AB. These are grouped into the same equivalence class g, with prefix AB. The associated extlist is C, D and E. We now need to find all the candidates associated with each of the itemsets in g. We can illustrate this process by showing it for ABC. We first find the items that are lexicographically greater than C in the extlist, namely D and E. Now, the potential candidates are ABCD and ABCE and we need to check if all their 3-subsets are also present in S3. That means that we have to search for ABi, ACi, and BCi where i may be either D or E. We don't have to search for ABC, as it is already present. To search for ABD, for example, we access its group, say h, with prefix AB and then check if in the extlist D is present, which mean that ABD is present in S3. The important point is that, having come so far to check if ABD is present, we can as well check if ABE is present, as it would also be

present in the same equivalence class with prefix AB. We can do this by going further down in the extlist to check if E is present in it.

Generalizing the above example, we can overlap subset status determination of multiple candidates by ensuring that all subsets across these candidates that belong to a common group are checked for with only one access of the associated extlist. This is in marked contrast to the standard practice of subset status determination on a sequential basis, resulting in high computational cost.

3.4 Calculating Support and Frequent Items

Once the candidates are ready in the array "candidates", we process each of them to find out their support in the database of transactions. As defined earlier, support(Item) is the following:

support (item) = no occurrences (item)/ total no of transactions.

So, for each item, which is a candidate, we count from the database the number of times it is present in the set of transactions. Many optimizations can be used in this method. Some methods like FP-Tree can be used to calculate and find support without even generating candidates. However, we do the simple method of counting the existence of 1's in the transaction list. This turns out inefficient and time consuming for very large databases, of size greater than a few gigabytes. So, better methods can be used here.

Once the support is calculated for each candidate item, it is checked against the threshold given by the user. If the threshold is crossed by the candidate, it becomes a "frequent item".

We repeat this process for each candidate at the given level. The items whose support crosses the threshold are stored in the frequent item set. These items are then made the candidates of the next level. This set of candidates is subject to similar process and this process is repeated till no frequent items are found at a given level. This means that we have reached the maximum possible combination of items; having support more than the threshold, and that we cannot proceed further to the next level to find more frequent items. This process of finding maximum possible frequent itemset stops here.

3.5 Rule Generation

At the end of the previous step, we have a set of items that form the maximum possible frequent itemset. Now, we have to mine rules out of this frequent itemset. This proceeds as follows in our implementation:

The user enters a minimum threshold confidence level. Confidence, as defined before is

$$\text{Confidence } (a \text{ -> } b) = \frac{(\text{no of transactions having } a \text{ and } b)}{(\text{no of transactions having } a)}$$

This provides the assurance required to publish a rule. We generate rules as follows:

```
Generate_rules_algorithm()
for (each item in frequent itemset)
{
 find all subsets of the frequent itemset and store it;
 for(each subset S of frequent itemset under consideration)
 {
  find out the support for subset S from the storage  structure;
  find out the ratio of support(frequent item) : support(S);
  if(ratio > threshold_confidence)
  {
   publish rule subset S → (frequent itemset – S) with confidence = ratio;
  }
 }
}
```

As seen above, the algorithm takes each frequent itemset and generates all its subsets. This is stored and one by one, for each subset, we find the confidence as specified. If it crosses the threshold set, it qualifies as a strong association rule. This rule is published and this process is repeated for all subsets of all frequent itemsets.

This ends the program execution with strong association rules being published, along with their confidences.

4 Results

From the literature, we obtained the following performance statistics of Apriori Algorithm. This is compared here under similar configurations with the execution of our algorithm, a hybrid between Apriori and VIPER algorithm.

We noted the performance of our algorithm in terms of the execution time for a number of transactions as shown in Table 1. We executed the algorithm for the following configuration data.

Total Number of Items = 5
Minimum Confidence = 45%

We have then varied the total number of transactions and found the total execution time for four different levels of minimum support. Fig. 3 shows the comparison of our implementation with Apriori.

It can be seen from the above table and chart that our implementation clearly out performs the traditional Apriori implementation. Moreover, the difference in performance is more clearly pronounced as the size of the database increases.

Table 1. No. of transactions V/S execution time

No. of transactions	Execution Time (sec)	
	Apriori	Our Implementation
10,000	11	2.2
50,000	51	8.12
1,00,000	103	14.57
2,00,000	209	28.34
5,00,000	512	95.05
10,00,000	1046	137.9

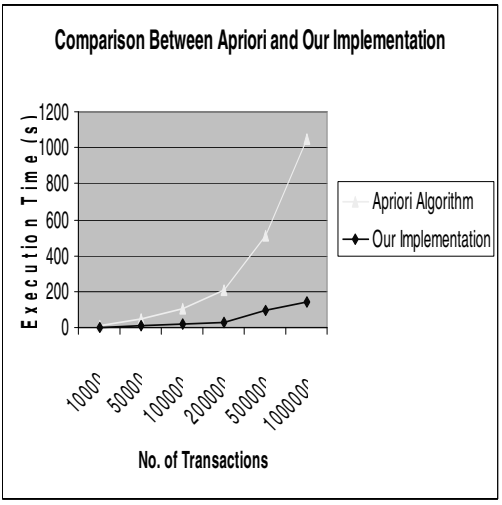

Fig. 3. Comparison of Apriori Algorithm and Our implementation

5 Conclusions

Our work "Association Rule Mining in Large Databases" has been successfully developed as a hybrid of the Viper and Apriori implementations. Based on the results of our development work, the execution time using our method is better compared to the Apriori algorithm. This is proved ever at higher number of transactions. This algorithm scales well compared to the original Apriori implementation as expected. The performance scales well with the size of the input database. Moreover, it is designed for a stand-alone system, as we have not focused on optimizing CPU cycles, which also becomes a concern in distributed databases. Some of the inherent drawbacks in the algorithm developed by us are as follows:

The algorithm reads directly from the database of transactions rather than storing it in a buffer and maintaining the buffer in main memory by periodically reading in

blocks of data. This is not required though, if the database is not read repeatedly, as in the case of prefix tree methods of mining association rules. Here, the database is read only once and a prefix tree is constructed from it, which is used for further calculations of support and confidence.

The other drawback is that we make use of a synthetic data generator for generating our test case inputs. This is a drawback of the random number generator not being "random enough" over long series of data. Data is generated by the default rand() function and as it is programmed to follow a specific probability distribution, the output also follows a trend. The data generated follows similar patterns at regular intervals, thus making the output i.e. the frequent itemsets, their support and also the association rules quite predictable when the database size is scaled. In other words, the rules obtained when the number of transactions is say 10,000 is very similar to the ones obtained when no. of transactions is say 1, 00,000, except for the scaled levels of support and confidence.

The actual efficiency of our program will be better appreciated when our program is run on real time data.

6 Future Extensions

6.1 Minimizing I/O Costs

Routines to read data in blocks from the disk into main memory have to be incorporated in our program. More efficient mechanisms to find the support and confidence from the given database should be explored into and incorporated to make the number of I/O's lesser compared to the current implementation.

6.2 Compression of Data Stored

We can use compression techniques like "Run Length Encoding" to store data in the form of compressed bit vectors called snakes in the main memory thus gaining space. This requires conversion on-the-fly when data is required for counting etc.

6.3 FANGS Implementation

We can implement the FANGS (Fast Anding of Graphs) procedure of the VIPER algorithm to generate the frequent itemsets and subsequently the association rules as it involves fewer scans of the database.

6.4 Prefix-Tree Methods

Instead of generating candidates and then checking if they are frequent, we can directly use the method of prefix trees to mine rules.

References

[1] Shenoy, P., Bawa, M., Shah, D.: Turbo-charging Vertical Mining of large databases. In: Proceedings of ACM SIGMOD Intl. Conference on Mgmt of Data (2000)
[2] Mobasher, B., Cooley, R., Srivastava, J.: Web mining: Information and pattern discovery on the World Wide Web. In: 9th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 1997 (1997)
[3] Agrawal, R., Imielinski, T., Swamy, A.: Mining association rules between sets of items in large databases. In: Proceedings of ACM SIGMOD Intl. Conference on Management of Data (1993)
[4] Bodon, F.: A Fast Apriori Implementation. Computer and Automation Research Institute. Hungarian Academy of Sciences, Budapest (2012)
[5] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proc. of 20th Intl. Conf. Very Large Databases, VLDB (1994)
[6] Golomb, S.W.: Run Length Encoding. IEEE Transactions on Information Theory 12(3) (1966)
[7] Savasere, A., Omiecinski, E., Navathe, S.: An efficient algorithm for mining association rules in large databases. In: Proc. of 21st Intl. Conf. on Very Large Databases, VLDB (1995)
[8] Chen, M., Han, J., Yu, P.S.: Data Mining: An Overview from a Database Perspective. TKDE 8(6) (December 1996)
[9] Adriaans, P., Zantinge, D.: Data Mining. Addison-Wesley (1996)
[10] Agrawal, R., Imielinski, T., Swami, A.: Database Mining: A Performance Perspective. TKDE 5(6) (December 1993)
[11] Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann (2001)
[12] Han, J., Pei, J., Yin, Y.: Mining Frequent Patterns without Candidate Generation. SIGMOD (2000)