Enhanced Mining Association Rule Algorithm with Reduced Time & Space Complexity

Punit Mundra, Amit K Maurya, and Sanjay Singh
Department of Information & Communication Technology
Manipal Institute of Technology, Manipal University, Manipal-576104, INDIA
sanjay.singh@manipal.edu

Abstract—In this paper, we have proposed a technique to improve the performance of existing mining association rule algorithm which significantly reduces the time and space complexity of independent of datasets. There are many data mining algorithms for finding association rules our contribution can be used in almost all of the algorithms independent of its variety. In this paper we are concentrating more on Apriori algorithm which is a type of candidate generation algorithm also a fundamental block of all the mining algorithms, rectifying its major limitation of consuming ample amount of time in generating the candidates.

Index Terms—Association Rule, Candidate Generation Algorithm, Data Mining

I. INTRODUCTION

Observing the present-day scenario, one of the major concerns is how to store immense amount of data in lesser amount of space, also how efficiently we can seek for particular information in definite time. In past few years, data mining has appealed a great deal of attention in the information industry [1][2]. The goal of data mining is to extract knowledge from a data set in a human-understandable structure. Association rule mining is one of the most important step of data mining.

Finding frequent patterns is a very important technology in association rule. R. Agrawal et al in 1993 [3] first proposed the famous Apriori algorithm based on frequent itemset. The task of Apriori algorithm is to extract the information from the immense amount of data but the enhancement what we are providing here is how proficiently we can perform it by controlling the number of comparisons. There are two phases in Apriori algorithm, first is to generate the candidates and, second frequent item set generation which is done after the comparisons. After adopting this method one can reduce the time complexity to $O(n^2)$ which is very high in standard algorithm. In our method we enumerate the transactions and after it we consolidate it.

The rest of the paper is organized as follows. Section II discusses the state of existing work in the related area and section III describes the concept of Apriori algorithm. Section IV discusses the proposed method. Section V explains the space and time complexity of the proposed algorithm and its comparison with the existing one. Section VI discusses about the results obtained and finally section VII concludes the paper.

II. RELATED WORK

There are several work focused on the improvement of Apriori algorithm. Ji, Zhang and Li [4] has used $(k-1)$-dimensional infrequent item sets to decrease the operation time. A dynamic item set counting method has been used in [5]. Apriori-Growth Algorithm [6] combines Apriori algorithm and FP-Growth to mine association rules.

Huan Wu et al [7] has also used counting based method to prune the candidate set, the count occurrence operation is improved by decreasing the amount of scan data using generation record. Ant colony algorithm (ACA) [8] is a new method for solving the optimal combination problem, they have applied Apriori algorithm to extract the useful information for the ACA from the large number of running data in the substation operation process.

Li Pingxiang et al [9] has suggested mining frequent item sets by evaluating their probability of supports based on association analysis. First, it gained the probability of every 1-itemset by scanning the database. Second, it evaluates the probability of every 2-itemset, every 3-itemset, and every $k$-item set from the frequent 1-itemsets. Third, it gains the entire candidate frequent item sets. Fourth, it scans the database for verifying the support of the candidate frequent item sets, last the frequent item sets are mined and association rules also do, by this method they reduces a lot of time for scanning database and shortened the time complexity of the algorithm.

On the other hand, there are many algorithms which have been devoted to perform ARM in parallel and/or distributed data mining, such as Hash Portioned Apriori (HPA) [10], weighted load-balancing parallel apriori [11], T-trees [12], Heuristic Data Distribution Schemes (HDDS) [13] implements ARM on grid environments, and some works have been done in peer-to-peer system.

Goethals and Zaki [14] has tested some existing frequent item sets mining algorithms, and performed timing and memory usage experiments for given datasets. According to the experiment report, there are no clear winners for all kind of datasets. A generating artificial data model has been proposed in [15] to produce realistic synthetic data for testing ARM algorithms. Algorithms of ARM for specific dataset are still needed.
Most of the existing work on association rule mining emphasizes either on reducing the domain of candidate set (so less frequent item set are generated) or avoid the scanning of database. All the work done is mostly applicable for candidate generation method. But the proposed method explained in this paper is applicable for both candidate generation and non-candidate generation method. Our methodology greatly reduces the number of comparison for comparing the frequent item set. Enumerating the database or transaction will help to locate the data item set. By enumeration technique we know the location of date item which helps to find the frequent patterns with fewer comparisons. If we compare our methodology to counting based method or probabilistic method we have less number of comparisons while generating the frequent set. Even the other method have less number of candidate set domain, but if we consider larger amount of data set (say 1 million data items/transactions), our methodology performs much better in term of comparisons hence reduced time complexity.

III. CONCEPT OF APRIORI ALGORITHM

Apriori algorithm [3] is the most popular algorithm to find all the frequent sets. It makes use of downward closure property. This algorithm as the name suggest is based on a bottom up search [16], moving upward level-wise in the lattice. However the beauty of the method is that before reading the database at every level, it gracefully prunes many of the sets which are unlikely to be frequent sets. Concluding, the elementary idea of Apriori algorithm is to generate a candidate set, and then frequent set. On the basis of given support, frequent item set is generated, but when number of frequent item set is huge, it results in large number of candidate generation (in millions). As the dataset size increases time required to find frequent item set also increases.

A. Limitations of Apriori Algorithm

1) Scan the same database again and again which generates options in form of frequent item sets [17].
2) As the size of the item set increases, the amount of memory requirement also increases which further increases the time to calculate required support. For calculating the support we have to consider both the parameters memory and time.

B. Progressive Approach to Overcome the Limitations of Apriori Algorithm

Every time reading from the database is a time consuming task, so to avoid the same we have proposed two solutions:

1) Read the database once and store it in a text file for permanent use and whenever we need to read the data we can read it from a text file rather than reading it from the database.
2) If the frequent item set is too large then again it requires lot of time to calculate the support count. Earlier methods like hash function [18] were proposed to avoid unnecessary comparisons, but it requires computation for each data item every time. If we can identify in advance as which data item is stored at what location then the number of comparisons can be reduced substantially while generating support count during frequent item set.

One of the ways of doing this task is to enumerate each data item, store that data item and use its enumerated value as an index. In our method enumeration procedure is used. One of the limitation of the enumeration procedure is that it consumes unnecessary space so here we have proposed a solution to that problem as well.

IV. PROPOSED METHOD

Consider the data item set where transaction is from $t_1$ to $t_{15}$. Every single transaction covers various data item $d_0$, $d_1$ etc. For every row we produce the string (enumerate the data item of each row). Enumeration of the data item will be useful while locating data item, it will make search easier and efficient.

Proposed method involves following steps:
1) In a transaction, assign the index number to every data item.
2) Generate a ‘binary string’ corresponding to each row. If data item is present allot ‘1’ to that particular index and allot ‘0’ to each index where item set is not present. In row 6 of Table I the last element is $I_{14}$ it means the length of the string is 15 i.e one more than the last element. The corresponding Table I shows how a string is generated to each corresponding row of data item set.
3) Further we produce the decimal number corresponding to string of each row of data item set. The key concept of producing a string is that, when we compute the support for 2, 3 or $n$ data item set, number of comparison reduces significantly, because the items in the string are enumerated, hence we know the position where the data item is existing. As we know that, data item $I_{05}$ will be at the 6th position in the string produced, so when we associate a particular data item of the row we search only at that place, which will subsequently evade the needless comparisons. But the problem is that if the row contains large numbers of items or only few items such as $I_0$, $I_{90}$, then in this condition our size of the string is very large and it consumes a lot of space, which results in wastage of space. To overcome the above mentioned problem we use a different concept, where we will transform these strings to an integer. Suppose row 4 contains data items $I_0, \cdots, I_{15}$, then string produced is $(1000000000000001)_2$ so the corresponding decimal equivalent will be 32769.

4) Producing the fresh array from the string corresponding to each row. This fresh array will contain decimal value of reduction table. Illustration: $T_1 = 65535$, $T_2 = 21845$ etc. With reference to step-2, we have allotted binary bit of ‘1’ or ‘0’ to each data item, for producing a string. Our objective is to decrease the space as much as possible, for that we have taken an array which is of the type ‘Long int’, which means that a single element
### Table I
**Sample Transaction**

<table>
<thead>
<tr>
<th>Transaction</th>
<th>List Of Items</th>
<th>Corresponding String</th>
<th>Decimal Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>$I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11}, I_{12}, I_{13}, I_{14}, I_{15}$</td>
<td>1111111111111111</td>
<td>65535</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$I_0, I_2, I_3, I_4, I_6, I_8, I_{10}, I_{12}, I_{14}$</td>
<td>10101010101010101</td>
<td>21845</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$I_1, I_3, I_5, I_7, I_9, I_{11}, I_{13}, I_{15}$</td>
<td>010101010101010101</td>
<td>43690</td>
</tr>
<tr>
<td>$t_4$</td>
<td>$I_0, I_{15}$</td>
<td>00000000000000001</td>
<td>32769</td>
</tr>
<tr>
<td>$t_5$</td>
<td>$I_0, I_3, I_6, I_9, I_{12}, I_{15}$</td>
<td>00010001000100010001</td>
<td>37449</td>
</tr>
<tr>
<td>$t_6$</td>
<td>$I_2, I_6, I_{10}, I_{14}$</td>
<td>00101000100010001</td>
<td>17476</td>
</tr>
<tr>
<td>$t_7$</td>
<td>$I_0, I_5, I_{10}, I_{15}$</td>
<td>00000001000000001</td>
<td>16512</td>
</tr>
<tr>
<td>$t_8$</td>
<td>$I_7, I_{14}$</td>
<td>0000000000001111</td>
<td>1052</td>
</tr>
<tr>
<td>$t_9$</td>
<td>$I_{11}, I_{13}$</td>
<td>010000000000000111</td>
<td>6400</td>
</tr>
<tr>
<td>$t_{10}$</td>
<td>$I_1, I_2, I_3, I_{13}, I_{14}, I_{15}$</td>
<td>01110000000000011</td>
<td>57358</td>
</tr>
<tr>
<td>$t_{11}$</td>
<td>$I_4, I_5, I_6, I_{10}, I_{11}, I_{14}, I_{15}$</td>
<td>000111100000110011</td>
<td>52336</td>
</tr>
<tr>
<td>$t_{12}$</td>
<td>$I_4, I_5, I_6, I_9, I_{10}, I_{11}, I_{12}$</td>
<td>0011110001101111</td>
<td>7740</td>
</tr>
<tr>
<td>$t_{13}$</td>
<td>$I_1, I_7, I_8, I_{10}, I_{11}, I_{13}, I_{14}$</td>
<td>010000001101111</td>
<td>28034</td>
</tr>
<tr>
<td>$t_{14}$</td>
<td>$I_8, I_{11}, I_{12}$</td>
<td>00000000000100011</td>
<td>6490</td>
</tr>
<tr>
<td>$t_{15}$</td>
<td>$I_{12}, I_{13}, I_{14}$</td>
<td>00000000000011011</td>
<td>28672</td>
</tr>
</tbody>
</table>

![Transformation of Data in Decimal Equivalent](image)

**Algorithm 1** Enhanced Apriori Algorithm(Transaction $T$)

1. $C_1$ ← all the 1 item sets
2. while $T \neq \emptyset$ do
   3. $C_i$ = Generating-1-ItemSet($T$) 
   4. $ED$ = GenerateEnumeratedString($T$)
   5. $CD$ = GenerateCompressedDataBase($ED$)
   6. end while
7. Initialize index = 2
8. $F_{index} = Prune(C_1)$
9. Generating-2-ItemSet
10. while $F_{index} \neq \emptyset$ do
   11. $C_{index} = GeneratingCandidateSet(F_{index})$
   12. $F_{index} = Prune(C_{index})$
   13. end while

**Algorithm 2** GeneratingCandidateSet($F_{index}$)

1. $C_{index} \leftarrow \emptyset$
2. $\forall i \in F_{index} - 1$
3. $\forall j \in F_{index} - 1$
   then
   5. $c = C_{index} \cup c$
   6. return $F_{index}$
7. end if

---

in fresh array can be point up to 63 data items (size of 'Long int' is 64 bits).

Figure 1 describes how the mapping takes place between enumerated and compacted array in our proposed algorithm.

**A. Algorithm Description**

Initially we start with the calculation of the user specified minimum support count where for every item set corresponding to every transaction we generate a binary string of length Imax i.e. item index in that particular transaction. String ‘1’ is allocated if item is present in the item set else ‘0’. Next step
Algorithm 3 Prune($C_{index}$)
1: while $C_{index} \subseteq C_{index-1}$ do
2: \hspace{1em} if ($C_{index}$-Support $\geq$ Given Support) then
3: \hspace{2em} $F_{index} = C_{index}$
4: \hspace{2em} return $F_{index}$
5: \hspace{1em} end if
6: end while

Algorithm 4 GenerateEnumeratedString(Transaction T)
1: Initialize $ED = \varnothing$, difference = 0
2: while transaction $\neq \varnothing$ do
3: \hspace{1em} difference = $T_{index} - T_{index-1}$
4: \hspace{2em} $ED = ED \cup$ AppendZero(difference)
5: \hspace{1em} return ED
6: end while

Algorithm 5 GenerateCompressedDataBase(ED)
1: Initialize = $CD = \varnothing$
2: while $(\log_{63}(\text{length} \cdot ED))$ do
3: \hspace{1em} $CD = CD \cup$ GenerateDecimalValue($ED$)
4: \hspace{1em} return CD
5: end while

is to divide the whole string in $\log_{63}$ (String Length). Here we have taken value of log base as 63 due to the simple reason that we are using data type as "Long Int" and we know that it takes 8 bytes (or 63 bits) to store the data. So we can represent a bunch of items by one Long Int data type character. Next step is to convert this binary string corresponding to every part into decimal number. For searching any item we have to search at the index corresponding to item's index, so if the item is present then its value will be '1' else '0'. This reduces our time for searching because it will significantly reduce the number of comparisons while searching. In the generation of $n$-item frequent item set we generate a binary number corresponding to the $n$-item candidate set and then convert this binary number to the decimal equivalent and match this decimal equivalent generated in the previous steps.

For generation of 2-item frequent item set if we have candidate set as $I_2, I_4$ then we use '00101' as our binary string corresponding to the candidate set and then convert this to the decimal equivalent. Next step is to compare this decimal equivalent with the decimal equivalent generated in the process of 1-item frequent item set. The algorithm works in two phase. Phase one is generation of candidate item set and phase two is generation of frequent item set. In our algorithm first we read the transaction item (data base) and generate 1-item set. Algorithm Generating-1-ItemSet generate 1-item set. While reading the transaction(data base), we generate the enumeration string simultaneously, whenever a row completes, algorithm GenerateCompressedDataBase generates compressed data base. After the generation of compressed data set the following equation is used for the generation of frequent item set.

$$f_{index+1} \leftarrow \text{Transaction} \subseteq \exists \text{CandidateSet}_{index+1}$$

The candidate set which is subset of transaction and it should be support number of times present in the database.

V. SPACE AND TIME COMPLEXITY OF THE ENHANCED ALGORITHM

In this section we are calculating and confirming the space used is getting compact, also number of comparisons has been reduced after adopting proposed method. Let the number of rows be $n$. Number of elements in each row is defined as:

First row carries $m_1$ number of elements
Second row carries $m_2$ number of elements and so on $n^{th}$ row contains $m_n$ number of elements

So total number of elements in a database is given by

$$M = \sum_{i=1}^{n} m_i$$

where \( i = 1, 2, 3, \ldots, n \).

A. Time Complexity Analysis

Fig. 2. Graph to Understand Time Complexity

Firstly for 1-item set selection, we will read the database from Row_1 to Row_n. So, the total number of comparisons will be $M$. So we can represent this as $O(M)$. It is to be noted that we are generating the string while reading the database first time. Now if we want to find 1-item set with a particular property then we have to prune this one item set on the basis of support count which requires comparisons equal to total number of item sets. Then, the complexity will be $O(1 - \text{item set})$. Here, we are generating 2-item set with the help of 1-item set. If items = 4, then 6 step count is required to generate the 2-item set. So the complexity, while generating 2-item set will be $O(1^{item-1})$ which can be approximated to $O(\text{item}^2)$.

For generating q-item set (where $q = 3$) we have to compare each phase with q-items. But again the complexity will remain unchanged and will be $O(\text{item}^q)$. While comparing the single item among the 2-item sets there will be total of $n$ comparisons since there are $n$ rows. Similarly, for other items, there will
be again $n$ comparisons. So, the total number of comparisons for 2-item set is $2n$. Now, we will be doing this for $n$ rows of 2-item set, so this will require $2n \times n$ comparisons. Hence, the time complexity will be $O(n^2)$. Time complexity performance of various version of Apriori algorithm is shown in Fig.2.

B. Space Complexity Analysis

Here, we will consider each row individually. First row is containing $m_1$ items so the space consumed will be $\log_{63} m_1$. If we have to find this in terms of bytes the space will be $8 \times \log_{63} m_1$. Now, if we consider all the rows, then total space consumption will be $(8 \times \log_{63} m_1 + 8 \times \log_{63} m_2 + \ldots + 8 \times \log_{63} m_n)$. The total space consumption is given by

$$M = \sum_{i=1}^{l} 8 \log_{63} m_i \text{ for } i = 1, 2, \ldots, l$$

where $l$ is length of the string.

VI. RESULTS AND DISCUSSION

Presented paper primarily laid emphasis on the reduction of space and time complexity of Apriori algorithm. We have considered a huge data item set and attempted to figure out the result in relation to number of comparisons as well as space.

Figure 2 depicts how the consumption of time gets reduced due to decrease in number of comparisons for generating the frequent data item set. Here we have analyzed on the frequent data item set such as frequent 2-item set or 3-item set or 4-item set represented on X-axis. Y-axis is representing the number of comparisons. Series two shows the reduced number of comparisons. Consider two 2-dimensional array, first array consist of transaction item and second array consists of candidate data items set (2-items or 3-item or 4-item etc.).

Consider the above equation which calculates the frequent item set. Now we have to compare two 2-dimensional array, traversing 2D array will require $O(n^2)$ comparison. If two 2D array to be traversed it required $O(n^2 \times m^2)$, where $n$ and $m$ are the dimension of each arrays. In our method for comparing 2-item set we require $2T \ast (total \ number \ of \ rows \ in \ 2-item \ set \ candidate \ set)$ comparison, where $T$ is the total number of transaction. Hence $O(n^2 \times m^2) >> 2T \ast (total \ number \ of \ rows \ in \ 2-item \ set)$. Similarly for 3-item set or more than that comparison increases drastically, this shows the efficacy of our algorithm.

Figure 3 shows how the space consumption is decreasing by adopting our proposed method. X-axis is denoting the number of items in data set. Y-axis is denoting the space in bytes for storing. In Fig.3 line contain diamond indicating the actual space consumed by the information. Here we have considered three scenarios namely worst case, average and the best case respectively. The difference in these cases depends on the position of data items. In best case the data item sets are present in consecutive position, while in worst case data items are present at first and last positions. To prove our method we have successfully simulated our algorithm on some random data set and figured out the efficacy and ability of the method we propose is proficient enough as compared to Apriori algorithm.

VII. CONCLUSION

The fundamental mechanism of association mining rule is to generate a candidate and frequent item set. In this paper we are generating the candidate and frequent item set in form of strings which has been enumerated in decimal form, so adopting this method reduces the number of comparisons significantly, while finding the support for frequent data item set. Considerable work has been done in past, for improving the proficiency of Apriori algorithm, but the major drawback in all those methodologies is that their performance is dependent on the type of data item set they select. The major advantage of the proposed methods is that the performance remains stable and unbiased irrespective of the data set we select.

REFERENCES


