A Compress-Based Association Mining Algorithm for Large Dataset

Mafruz Zaman Ashrafi, David Taniar, and Kate Smith

School of Business Systems, Monash University, PO BOX 63B, Clayton 3800, Australia.
{Mafruz.Ashrafi,David.Taniar,Kate.Smith}@infotech.monash.edu.au

Abstract. The association mining is one of the primary sub-areas in the field of data mining. This technique had been used in numerous practical applications, including consumer market basket analysis, inferring patterns from web page access logs, network intrusion detection and pattern discovery in biological applications. Most of the traditional association-mining algorithms assume that whole dataset can be loaded in the main memory. Hence, problem arise when such algorithms is applied in large dataset. In this paper we present a new algorithm for association mining. Our algorithm is efficient when the size of dataset is huge that cannot be load in the main memory. The proposed algorithm also reduces the frequent itemsets search space, by eliminating non-frequent 1-itemsets after the first pass. Our performance evaluation shows algorithm outperforms Apriori algorithm in different datasets.

1 Introduction

Data mining is iterative and interactive processes that explore and analyze voluminous data in order to discover valid, novel and meaningful patterns, associations or rules, using computationally efficient techniques [1]. It is related to the sub area of statistics called exploratory data analysis, which has similar goals and relies on statistical measures and also closely related to the sub areas of artificial intelligence called knowledge discovery and machine learning.

Data mining has been attracted huge attention in numerous research communities due to its wide applicability in many areas such as retail industry, financial forecast, decision support and intrusion detection [2]. Data mining methods include associations clustering, classification, and prediction. One of the most important fields of the data-mining domain is the association mining. Many interesting and efficient mining of association rule algorithms have been proposed in the different mining literature [2, 3, 4, 5, 6, 8, 9, 10, 11]. In this paper we present an efficient association-mining algorithm for large dataset.

Agarwal et al first introduced discovering association rules from the market basket dataset. The Apriori algorithm is one of the most popular algorithms in the mining association rules. There are number of extensions of the Apriori algorithm such as Partition [5], DHP [11], etc.

Discovering all association rules from very large dataset, by using support-confidence based framework is not a trivial task. The search space is exponential in the number of database attributes and with millions of I/O minimizations become paramount [2].

In this paper we present an algorithm called Compress that has significant difference from the Apriori and all other algorithms those are extended from Apriori algorithm. The compress algorithm loads the dataset into main memory and organized them into different vertical partition table based on the first item of the each transaction and compresses them after a specific limit. The algorithm also removes items from the transactions if those particular items don't have user specific support level.

The rest of paper organized as follows: In Section 2 we describe background of association mining and its problem. We mention the problem of Apriori algorithm in Section 3. In Section 4 we describe our proposed algorithm. The performance evaluation and comparison studies are described in section 5 and we conclude at Section 6.

2 Association Mining Rules: A Background

Association mining can be described as a correlation of events which means, events those are frequently observed together or in other words an association algorithm creates rules that describe how often events occur together. The prime task of association mining is to discover a set of attributes shared among a large number of objects in a given database [1]. Two important measures for association rules are support (*s*) and confidence (*c*), the former can be defined as an associated measure of statistical significance of an *itemset*, and the later is an indication of the strength of the rule. A rule is frequent if its support is greater than the user specified support and strong if its confidence is greater than the user specified confidence. The following formula could measure the confidence of an association rule.

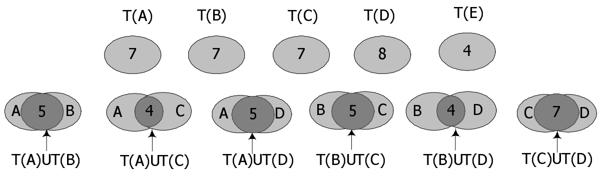
$$Confidence = \frac{Support\ (LHS \cup RHS)}{Support\ (LHS)}$$

Consider an example of supermarket sales database shown in the figure 1(a). There are few different items and few sample transactions of customers. In order to identify purchase/shopping pattern from such dataset, the association rule intend to established correlation among the different items and apply a specific level of support and confidence on those correlations and discover rules. Finally, supermarket can use those association rules knowledge for promotions, shelf placements, inventory control, customer service and etc. Figure 1(b) shows the number of time each item appears in the supermarket customer transaction and correlation (up to level 2) of items, those have minimum *50%* of support.

Items Description	
Tea	A
Milk	B
Coffee	C
Suger	D
Pen	E

Transactions	
1	A B C D
2	B C D
3	A B
4	A B C D E
5	A C D
6	A B E
7	C D E
8	A D
9	B C D E
10	A B C D

(a)



(b)

Fig. 1. (a) Supermarket dataset (b) correlation of items

Let $I = \{I_1, I_2, \ldots, I_m\}$ be a set of distinct attributes, also called *literals*. Let D be the databases of transactions, where each transaction T has a set of items such that $T \subseteq I$, and unique identifier (*tid*). The set of items also known as *itemset* and the number of items in an itemset is called the length of an *itemset*. The support of an itemset X, is the number of transactions it occur as a subset. An itemset is frequent if that itemset has a user specific support. An *association rule* is an implication of the form $X \Rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$ are *itemset*, and $X \cap Y = \phi$. Here, X is called antecedent, and Y consequent. The rule $X \Rightarrow Y$ has *support s* in transaction set D, if and only if $s\%$ of transactions in D contains $X \cup Y$ and holds with *confidence c* in transaction set D, if and only if $c\%$ of transactions in D that contains X also contains Y, and can be calculated by using the formula 1.

The problem of the association rule can be divided into two subgroups. Firstly, generating all itemsets that have user specific support i.e. *frequent itemsets*. Secondly, generating *confidence rules* from those frequent itemsets, which satisfy the user specific confidence. The formal problem is nontrivial and most crucial factor that affects the performance of the association rules. Moreover, this problem affects more adversely, if numbers of distinct attributes in database are large. For example, if a particular database has m number distinct attributes there can be 2^m numbers of possible distinct large itemsets and the problem is to find out those itemsets that have user specific support. Where as, the later problem is relatively easy and straightforward. For generating *confidence rules* from *frequent itemsets* those have user specific confidence in the $X/Y \Rightarrow Z$ form, where $Y \subset X$ and X is frequent

2.1 Apriori Algorithm

To resolve above mention problem [2] present this algorithm. This algorithm is considered as one of the most popular Association Rule Mining algorithms, however this algorithm need as many database scan as to find longest itemset from the dataset. The algorithm has the following steps:

– Large itemsets are generated through iterations and each iteration database is scanned once.

- In first iteration, algorithm determines large *1-itemsets* (i.e. itemset of length 1) by simply counting each item occurrence in transactions.
- Subsequent iteration consist two phases, firstly candidate sets C_k generation by applying *Apriori-gen* function, using the large itemsets L_{k-1} found in *(k-1)th* iteration those and store in a hash tree. A node of that hash tree either contains a list of itemsets (a leaf node) or a hash table (an interior node), secondly, the database is scanned and support of the C_k is counted.
- Pruning away those candidate subsets from C_k, those have not required support.

3 Problems of Apriori

From the above discussion, it is clear that Apriori association-mining process is an iterative process, and each iteration dataset needs to be read. Reading from disk involves I/O operation and hence computationally intensive. To read disk resident data at every pass of the association mining-algorithm resultant large number of disk I/O operations. On the other hand, it is not always feasible to keep whole dataset within the main memory if dataset is large. As a result, need multiple scans, which incur some additional computational cost and degrade overall mining performance even for small size dataset.

The performances of Apriori association-mining algorithm degrade further, if it requires to read disk resident data for every pass in order to generate itemsets then it will not able to perceive those transactions that have identical itemsets and therefore will unnecessarily occupy resources for repeatedly generating itemsets from such identical transactions.

In this work, our goal is to reduce I/O operations and computational cost of association mining by organizing each database transaction in vertical partitioned-based table (i.e. based on the first item of a transaction) and dynamically compress those tables in order to utilize main memory more efficiently. Furthermore, after first pass we prune away those non-frequent *1-itemsets* from each transaction of the dataset in order to reduce unnecessary itemsets generation.

4 Proposed Algorithm

In this section we discuss different components of our association-mining algorithm to elaborate it more precisely. Before embarking on the algorithm description, we will briefly discuss two assumptions that we made in this work. Firstly, we assume each transaction in dataset is either (*TID*, $i_1\ i_2\ i_3 \ldots \ldots i_j$) or ($i_1\ i_2\ i_3 \ldots \ldots i_j$) form. Secondly, all items in transactions are sorted in a lexicographic order. It is worth to mention that similar kinds of assumptions had been made in number of previous association rule mining research [2, 5].

Our proposed algorithm works in two different phases. The *first phase* is known as *reading* phase. In this phase, all transactions are read and based on the first item of each transaction, we construct different range of partition table as shown in the Figure 2(a) and placed each transactions to its corresponding table. Since transaction in a dataset may have thousand of items and not necessarily every transaction will begin with the same item, hence this method will create number of tables. The main benefit of this approach over the horizontal partition [5] of dataset is that, there is possibility to find transaction with same itemsets (even if we do not find transaction with the same itemsets in first pass, possibility of finding it is much higher after we prune itemsets from transaction, this method will be discussed in details in the following paragraph) in the later pass. Hence, it does not generate frequent itemsets from similar transactions (i.e. transaction with identical itemsets) for second time. Furthermore, this approach guaranteed that transactions with identical itemsets would only have single occurrence in the partition table so consumes less space in the main memory.

To understand this more precisely consider previous example at figure 1(a), where we have ten transactions consist of five different items. For each of those items we form a partition table (a hash table). Depending on the first item of a transaction we placed it to the corresponding table and set it counter into 1. Counter value will increment if any transaction with similar itemset is found later read. This phase also counts the support of each itemsets of length 1 (*1- itemset*). The size of each partition table defined in advance. When a particular partition table reached its maximum size the corresponding table is compressed and temporarily kept in the secondary memory in order to utilize the main memory more efficiently. When a partition table reaches its maximum limit a compression technique is used in order to compress that table.

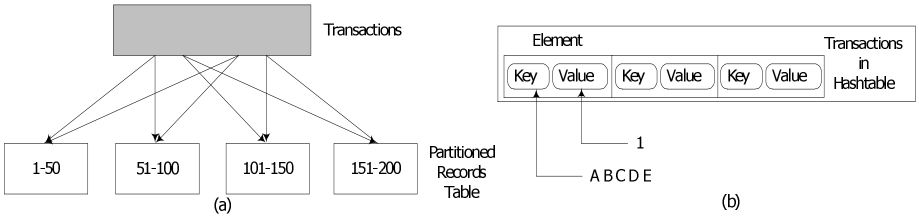


Fig. 2. (a) Different range partition table based on the first item of a transaction (b) Elements of partition table.

The *second phase* is known as *iteration* phase. At the beginning of this phase actual support of *1- itemsets* is generated. After generating all frequent *1-itemsets* each of the partition table will be iterated and corresponding itemsets (length *2,3,4, ... n*) will be generated. During iteration, elements of each partition table are examined carefully. Every non-frequent *1-itemsets* are discarded if found in any transaction because of the fact that itemset those are infrequent in initial pass cannot be able to generate frequent itemsets in the later pass and finally rehash those transactions into the corresponding partition table for a second time.

For example consider our previous example at Figure 1, after completion of the first phase all non-frequent *1-itemsets* are marked (in this case item *E* if we set *min_support = 50%*). During first pass of the *iteration* phase each transaction is examined, remove item *E* from each transaction that contains it and rehash those elements in the partition table. Figure 3 shows initially partition table has 9 elements and 4 elements contains item *E*. When we remove item *E* from each elements and rehash them for a second time we will have partition, which will have 6 elements as shown in the figure 3.

Conter	Transaction
2	A B C D
1	B C D
1	A B
1	A B C D E
1	A C D
1	A B E
1	C D E
1	A D
1	B C D E

Partition table in 1St phase

Conter	Transaction
3	A B C D
2	B C D
2	A B
1	A C D
1	C D
1	A D

Partition table in 2nd phase

Fig. 3. Elements in Partition table.

The notation used in this paper is shown in figure 4. Items of the dataset *I* will vertically be fragmented into different partitions *nP* and size *sP*. The size of a partition table *sP* specify how many transaction can be hold before compress that particular table in order to utilize main memory more efficiently. The vertical items range (i.e. which partition table will store based on the starting item if each transaction) of each partition table *vR* can be calculated by dividing the number of items *I* by the total number of partition table.

s	Minimum support;
c	Minimum confidence;
C	Candidate itemset;
N	Total number of transactions;
I	Total number of items;
nP	Number of partition tables;
sP	Size of each partition table;
vR	Vertical range of each table;
f	First item of a transaction;
F^i	Frequent itemset of length I;

Fig. 4. Notation

The algorithm is shown in the figure 5. In the first phase, algorithm will scan the whole dataset and insert each transaction in a particular partition table and count each *1-itemsets* occurrence. In the second phase each compress partition table will load into the main memory, remove every *1-itemsets* that don't have a specific support level and rehash them into the table for a second time. The *generate_itemset()* method takes each transaction as an input and generate itemsets of corresponding length in each iteration. Since the algorithm prune away elements after the first phase, *generate_itemset()* method generate less *itemsets* of various length. The pruning operation is applied after each iteration and remove *itemsets* those don't have corresponding support.

```
/*1st phase*/
for i = 1 to N do begin
  insert_into_table(i);
  count_item(i);
end;

/*2nd phase*/
generate_frequent()
for i = 1 to nP do
begin
  t = prune(i)
  if(t! = i)
    rehash(t);
  generate_itemset(i);
end;
```

```
insert_into_table(i)
begin
  f = firstItem(i)
//find partition table for f
  pT = vR(f)
  If(t < sP)
    put(i);
  else
  {
    compress(T)
    create table(T)
    put(i);
  }
end;//insert_into_table
```

Fig. 5. Algorithm

5 Performance Evaluation

In experiments we used a *1.6GHz I*ntel P4 machine with 512MB main memory. We used different synthetic datasets that have been proposed by [2] and those generated datasets were previously used for evaluating performance in various mining algorithms. We implemented our algorithm in Java 1.3. Since we implemented our algorithm in Java, we were looking for third party Apriori algorithm in Java in order to compare the performance our proposed algorithm with Apriori. But we were not able to manage any third party Apriori algorithm implemented in Java.

Let *D* denote the number of transactions, *T* the average transaction size, *I* the size of the maximal potentially frequent itemset, *L* the number of maximal potentially frequent itemsets and *N* the number of items. Experiments are conducted on datasets with different values of *T* and *I*. The parameters are shown in figure 6. The nature of datasets (sparse or dense) is depending on the *L* and *N* values. In order to generate dense dataset we keep both *L* and *N* values within 100.

Name	|T|	| I |	|D|	Size in MB
T18.I8.D10M	20	8	10M	490
T20.I12.D10M	20	12	10M	500
T10.I8.D10M	10	8	10M	321
T22.I12.D10M	22	12	10M	641

Fig. 6. Dataset with different parameters

Apriori algorithm we used in the comparison study was implemented in C[1]. Java treats every thing as an object (except few primitive data type such as *int*, *float*, *char* etc.) and object creation is considered as a costliest operation. In the past, various organizations did some basic Input/Output performance comparison between Java and C and result shows C outperforms Java. To give a clear picture of the above mention argument we did some experiment to show how much time a simple and straight forward Java program will take in order to generate frequent itemsets of length-1. Figure 7 (a) shows how much time Apriori implemented in C and a sample Java program took in order to find out all itemsets of length 1. The figure 7 (b) shows how much time took to find out itemsets of length 1, Apriori algorithm implemented in C and our proposed compress association-mining algorithm. The compression algorithm took more time due to the extra overhead that required in Java object creation.

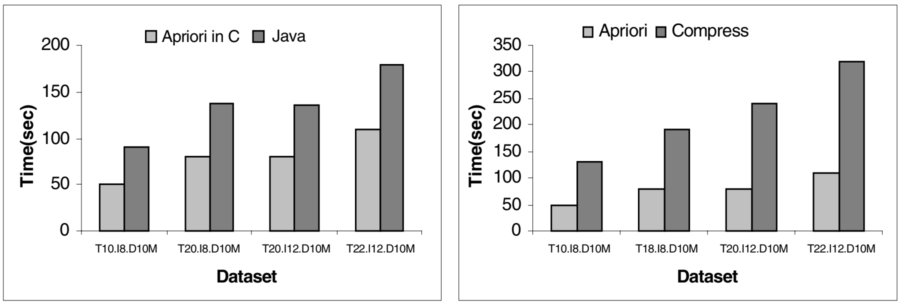


Fig. 7. Time taken for reading and generating itemsets of length 1, from different datasets.

Figure 8 (a, b, c, d) shows the performance comparison between the our proposed Compress and Apriori algorithm. The Apriori algorithm we used in this study has various options. It can store the whole dataset into the main memory during the mining task. In the beginning we tried to run Apriori with this option but we were not successful because the datasets we used in this experiment is large. Hence, we run the Apriori algorithm without loading the dataset in the main memory.

[1] The apriori algorithm was implemented by Christian Borgelt.

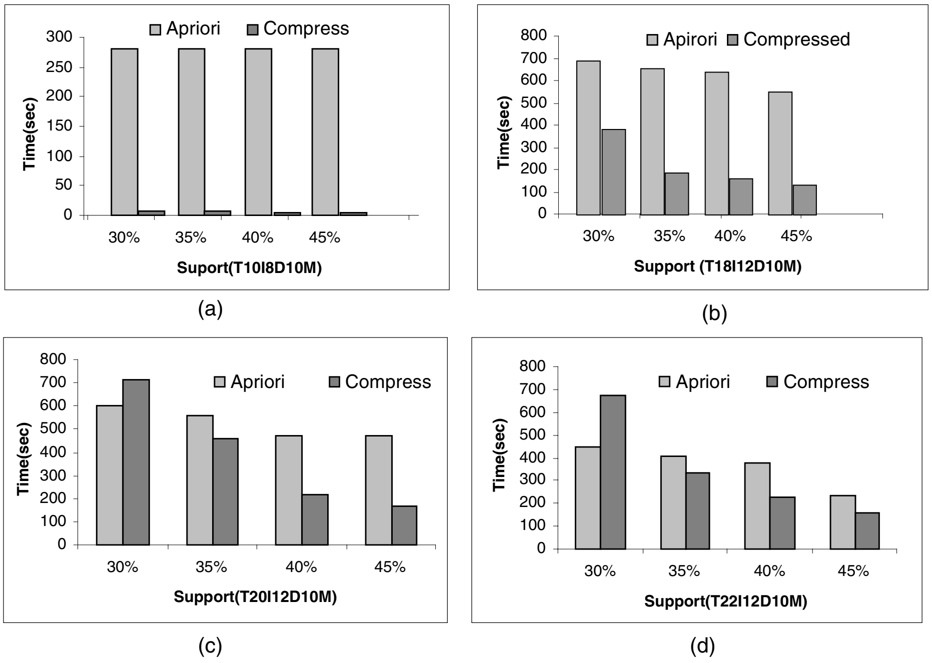


Fig. 8. Execution time.

As from the figure 8 it is clear that both Apriori and compress algorithm took more time if the size of the transaction (i.e. avg. number of items) is higher and the support level is lower. The compress algorithm prunes away those itemsets that don't have a specific support level after the first phase. It prunes away more itemsets from the transaction when support level is high. Due to this fact the proposed algorithm outperform the Apriori in most of the cases. Moreover, in our algorithm we implemented a better technique for generating candidate sets. Whereas, the Apriori algorithm uses the subset operation and hence cost of generating per itemsets is increased in the later pass [5].

Our compress algorithm performs significantly batter than Apriori algorithm in most of the cases. We found compress algorithm achieve best performance with dataset *T10.I8.D10M* in figure 8(a), because after the initial pruning transaction size become smaller. Hence the ratio of identical transactions after first phase is higher than the other datasets. Both Apriori and compress algorithm took more time as expected with the dataset *T20.I8.D10M* shown in the figure 8 (b). This is because the avg. length of size of each transaction which is larger than the previous dataset. In this dataset compress algorithm generate almost equal number of *1-itemsets* for support (35%, 40%, 45%) for this reason total execution time remain the same.

From figure 8 (c) and (d), it is clear that compress algorithm took more time as compared to Apriori algorithm when support is low and datasets have bigger transaction size. This is because, the compression algorithm finds less number of non-frequent 1-itemsets after the first phase and hence extra time required to create some extra objects. For example, in order to remove 1-itemsets from the each transaction compress algorithm create number of objects and only able to remove small number of 1-itemsets from it. We could reduce some time if implement our compress C/C++.

6 Conclusion

In this paper we examined various issues related with association mining, we also have described an algorithm, which is effectively generate association rules by discovering frequent itemsets from large datasets. An important outcome of our algorithm is to reduce the search space by eliminating *1-itemsets* from the transactions after the first pass. This feature may prove useful for finding frequent itemset from many real life dense dataset. An extensive number of experimental evaluation over various dataset showed that our proposed compress algorithm outperforms the Apriori algorithm in most of the cases.

Reference

1. Mohammed Javeed Zaki, "Parallel and Distributed Association Mining: A Survey", *IEEE Concurrency*, October-December 1999.
2. Mohammed Javeed Zaki, "Scalable Algorithms for Association Mining" *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12 No.2 pp. 372–390 (2000).
3. Rakesh Agrawal and Ramakrishnan Srikant, "Fast Algorithms for Mining Association Rules in Large Database", *In Proceedings of the 2oth International Conference on Very Large Databases*, pp. 407–419, Santiago, Chile, 1994.
4. David Wai-Lok Cheung, Vincent T. Ng, Ada Wai-Chee Fu, and Yongjian Fu, "Efficient Mining of Association Rules in Distributed Databases", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 6, pp. 911–922, 1996.
5. Ming-Syan Chen, Jiawei Han, and Philip S. Yu "Data mining: An overview from a database perspective". *IEEE Transactions on Knowledge and Data Engineering*, Vol 8, No 6, pages 866–883, 1996.
6. Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases", *In Proceedings of the 21nd International Conference on Very Large Databases*, pp. 432–444, Zurich, 1995.
7. Jiawei Han, Jian Pei, Yiwen Yin, "Mining Frequent Patterns without Candidate Generation*", In Proc. ACM SIGMOD Intl. Conference on Management of Data*, 2000.
8. Mohammed Javeed Zaki, Ya Pin, " Introduction: Recent Developments in Parallel and Distributed Data Mining" *Journal of Distributed and Parallel Databases,* Vol.11, No.2, 2002.
9. Doug Burdick, Manuel Calimlim, and Johannes Gehrke. "MAFIA: a maximal frequent itemset algorithm for transactional databases" *In Intl. Conf. on Data Engineering*, 2001.
10. Jong Soo Park, Ming-Syan Chen, and Philip S. Yu, "An Effective Hash Based Algorithm for Mining Association Rules", *In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pp. 175–186, San Jose, California, 1995.