PFIMII: Parallel Frequent Itemset Mining using Interval Intersection

Neelam Duhan
YMCA University of Sc. & Tech.
Faridabad

Parul Tomar
YMCA University of Sc. & Tech.
Faridabad

Amit Siwach
YMCA University of Sc. & Tech.
Faridabad

ABSTRACT

Data Mining techniques are helpful to uncover the hidden predictive patterns from large masses of data. Frequent item set mining also called Market Basket Analysis is one the most famous and widely used data mining technique for finding most recurrent itemsets in large sized transactional databases. Many methods are devised by researchers in this field to carry out this task, some of these are Apriori, Partitioning approach and Interval Intersection etc. In this paper, a new approach is being proposed to find the frequent item sets using Interval Intersection and Apriori Algorithm, which produces results in parallel on several partitions of dataset. For representing the item sets, interval sets are used and for calculating the support count, interval intersection operation is used. The experimental results indicate that the proposed approach is accurate and produces results faster than Apriori Algorithm.

Keywords

Frequent Item set mining, A-priori, Partition Algorithm, Interval Intersection, Support count.

1. INTRODUCTION

Frequent itemset mining is one of the most important tasks in data mining. Algorithms like Apriori [1] use breadth first search to find k-itemsets from (k-1)-itemsets. Apriori Algorithm takes N-scans on the dataset to generate the frequent itemsets of N size, therefore a lot of I/O cost is incurred.

In Apriori Algorithm, while generating frequent itemsets, care has to be taken to ensure that no redundant frequent itemsets are generated. Also in Apriori Algorithm, to calculate the support count of an item, whole dataset needs to be scanned. To ensure this, a new technique is used known as interval intersection [3], in which items are represented using intervals. This new technique reduces the amount of memory required and number of scans.

Apriori like algorithms take very large execution time due to large number of scans. To reduce the execution time, the task is performed in parallel on several partitions of the dataset. The results are merged in a global data structure using merge algorithm. Thus, frequent itemsets are generated using parallel technique in only two scans of the dataset.

In this paper, a new approach to find frequent itemsets is proposed which takes advantage of these three approaches. It partitions the Dataset so as to process them in parallel, deploys Apriori to find frequent Itemsets and represents the intermediate itemsets by using interval intersection approach.

The rest of paper is organized as follows. Section 2 presents some background history of frequent itemset mining. Section 3 describes the proposed algorithm. Section 4 shows the results using example and finally, Section 5 concludes the paper.

2. LITERATURE REVIEW

In this section, some of the prevalent methods for frequent itemset mining are discussed in detail.

2.1 Frequent Itemset Mining

Frequent item set mining was first studied by R. Agarwal [1]. He analyzed supermarket transactional dataset and proposed an algorithm known as Apriori Algorithm. Before defining frequent item set mining, some basic terms [2] needs to be defined such as:

Itemset $I = \{I_1, I_2 \ldots\ldots\ldots, I_m\}$ is a set of m items.

Transaction set $T = \{T_1, T_2 \ldots\ldots\ldots, T_n\}$ is a set of n transactions, where Ti is a transaction consists of items such that $T_i \subset I$. Itemsets can be any size, such as k-itemset is a set of k items which are different to each other and belonging to I. Now, frequent k-itemset can be defined as the itemset having support count greater than or equal to the user specified minimum support count (min_sup). So, the frequent itemset problem is to find all the frequent k-item sets from the dataset.

2.2 Apriori Algorithm

The Apriori Algorithm [1] is the very first and most widely used frequent item set mining algorithm. This algorithm was proposed by R. Agarwal in 1993 after analyzing the supermarket transactional dataset. The algorithm is used to find the frequent itemsets from datasets. This is a two phase algorithm. In first phase, for generating 1-itemsets the dataset is scanned. After generating the 1-itemsets, their support count is calculated and items are pruned on the basis of support count value. For k>=2, k-itemsets are generated by joining the different (k-1)-itemsets until no further itemsets can be generated. After generating all k-itemsets, their support count value is calculated by scanning the dataset and only those itemsets are retained whose support count value is greater than min_sup (minimum support) value. There are some limitations in the Apriori Algorithm such as it takes many scans of the dataset and a lot of time is wasted in calculating the support count because during each iteration dataset is need to be scanned.

2.3 Partitioning Algorithm

The main idea of this algorithm is to partition the database. Frequent items are found on the basis of dividing the dataset into N-parts depending upon the size of dataset. As database size becomes huge, so while finding the frequent item sets, the dataset does not fit into the Main Memory. Partitioning algorithm [4, 8, 9] overcomes the memory problem for large databases by partitioning the database in small units such that they fit well in main memory and processing can be done efficiently. A key objective [5] of the partitioning algorithm is to reduce the disk I/O as much as possible. A partition *p* is subset of database *D*. This is a two phase algorithm.

1. First phase:

The database is divided logically such that no two partitions p_i and p_j overlap i.e., ($p_i \cap p_j = \phi$, $i \neq j$) and local item sets for all partitions are determined. A large local item set may not be large with respect to global database. At the end of phase-1, all item sets are merged.

2. Second Phase:

Global support count is calculated for each item set and large item sets are identified. An item set can be globally frequent only if it is locally frequent in at least one partition. The partition size is chosen such that each partition fit in main memory and at least those item sets that are used for generating the new large item sets can fit in main memory.

2.4 Interval Intersection

An interval [3, 6] defines a range between two real numbers such as [a, b]. Let x be any real number in this interval then x can be represented by $a \le x \le b$ where a is starting number and b is ending number of the interval. Intersection is an operation on two intervals which is mathematically expressed as given:

For intervals X = [Xa, Xb] and Y = [Ya, Yb], the intersection operation is denoted by $X \cap Y$ and shown as: $\{Z \mid Z \in X$ and $Z \in Y\} = \{\max(Xa, Yb), \min(Xb, Yb)\}$.

Interval intersection notation is used to represent the itemsets so that minimum memory is used and least time is consumed for calculating the support count. Another advantage of this technique is that using this technique for calculating the frequent itemsets, only two scans are required, so it reduces the number of scans and make the process faster.

3. PROPOSED APPROACH

In this section, a new parallel approach to find frequent itemsets is discussed. The new approach is based on the interval intersection method and divides the dataset into various partitions to find frequent itemsets in parallel. The proposed algorithm is discussed as follows:

3.1 PFIMII: Parallel Frequent Itemset Mining Algorithm

Input to this algorithm is, dataset D and Global minimum support count which is given by user, is taken. The data structures used are, SCL i.e. Support Count List and Negative Border. SCL is used to store support count of each item set. Negative border is used to store those item sets, which are having less support count than the minimum support count. Both the data structures are initialized to null. As already discussed, the dataset size is huge so, N-partitions of the dataset are created using the partitioning algorithm and on each partition Interval Intersection algorithm is applied. Interval Intersection is used to quickly calculate the support count.

Now, as a result local frequent item sets are generated. These local frequent item sets are combined to generate global frequent item sets. After generating the global frequent item sets, a global scan is required to check the support count of frequent item sets against the value of Global minimum support count. Item sets having less support count than global minimum support count value are pruned and finally frequent item sets are generated. Algorithm executes until no further item sets can be generated. First all 1-item sets are generated and their support count is calculated by scanning the dataset. All the support counts are stored in support count list. Those items whose support count is less than that of the minimum support count value are put into negative border so that these items can be used at the time of merging. Pseudo code is given in Algorithm 1.

```
Algorithm: PFIMII(Dataset D, Gmin_sup)

Input: Dataset 'D' and Gmin_sup (Global min_sup).
Output: Frequent item set list.            //Stored in FIL
SCL=NULL //Initialize Support count List;
N_Border = Null //Initialize negative border;
P=Partition (Dataset D)     //partition dataset D into N parts.
Min_sup=Gmin_sup/N
for each partition p= 1 to N $\in$ P
{
Repeat until no further item sets are found i.e. $FIL_k = \phi$
//FIL(frequent item set list)
{
for i = 1 to k          //k length item sets;
{
scan the dataset and store the support counts in SCL i.e.
SCL=SCL$\cup$ Sup_count ($Item_i$)
while (SCL != empty)
{
        If( min_sup > SC($Item_i$))
        N_Border= N_Border $\cup$ {$Item_i$ }
        SCL= SCL-{$Item_i$}
}
for k>=2
Interval Intersection(Interval set A, interval set B, FIL)
Return $FIL_p$
}
}
}
Merge($FIL_1$, $FIL_2$,……$FIL_N$,Gmin_sup)
Return (FIL).
```

Algorithm 1: PFIMII Algorithm

After first iteration onwards, Interval Intersection is used to generate the frequent item sets. After generating the frequent item sets for each partition, merging of results of each partition is done using the merging algorithm. Finally the results are stored in a list called as FIL i.e. frequent item set list. Proposed algorithm (PFIMII) executes in parallel on several partition of the dataset and it needs only two scans of the dataset to generate the frequent item sets, thus provides faster results as compared to other algorithms.

3.2 Merge Algorithm

Merge algorithm is used to combine the results from all the partitions (partition-1, partition-2,……., partition-N) and the result is stored in FIL (frequent item set list). The results of each partition are stored in FIL1, FIL2…..FILN, corresponding to each partition. Merge algorithm works by first storing the results of partition-1 in FIL, which is the final frequent itemset list and then each item set from partition-2 to partition-N is taken and check against FIL. If the item set is present in the FIL, its support count is added to support of item set being checked and thus support count gets incremented.

```
Algorithm: Merge ($FIL_1$, $FIL_2$,……$FIL_N$, Gmin_sup)

Input: Results of all the partitions.      // $FIL_1$, $FIL_2$……$FIL_N$;
Output: List of frequent item sets.      // Final results in FIL;
FIL=NULL          //Initialize frequent item set list;
if ($Item_i \in$ FIL)
{
  SC($Item_i$)= SC($Item_i$).FIL + SC($Item_i$).$FIL_l$
 // Support count is added;
}
Else
{
  FIL= FIL ∪ {$Item_i$}          //Item is inserted in FIL
  while (FIL != empty)
 {
   if (SC($Item_i$) > Gmin_Sup) //GMin_Sup is global min_sup
     Continue;
   else if ( $item_i \in$ N_Border)
     SC ($item_i$)=SC($item_i$).$FIL_l$ + SC($Item_i$).N_Border
   if(SC($item_i$) > Gmin_Sup)
     continue
   else
     FIL= FIL – {$Item_i$} ;
 //Item is removed from the FIL;
  }
 }
```

Algorithm 2: Merge algorithm

If the value of any item set is less than the value of Global minimum support i.e. Gmin_sup, the itemset is checked in Negative Border N_Border, if found there, its support count is incremented by the value of support count of itemset present in negative border. After this, it is again checked and if it is more than the Global support count, itemset is placed in the list FIL, otherwise it is discarded. The pseudo–code is shown above in Algorithm 3.2.

3.3 Comparison Study

Proposed algorithm is compared with the Apriori Algorithm [1]. The comparison is done in terms of number of scans of dataset to generate the frequent item sets and the results i.e. frequent item sets generated in each of the iteration and is shown in Table 1.

Table 1. Comparison of Apriori and PFIMII Algorithm

Parameters	Apriori Algorithm	PFIMII Algorithm
Complexity	More complex due to many number of scans	Less complex due to only two scans.
Number of Scans	Three scans of the dataset	Two scans of the dataset
Results	Same results as that of the PFIMMI Algorithm	Same as that of Apriori Algorithm

In each of the iteration, the proposed algorithm is producing the same results as that of the Apriori Algorithm. So, it is obvious that proposed algorithm works accurately to generate the frequent itemsets. The other advantage of proposed algorithm over the Apriori Algorithm is that, Apriori needs a number of scans while proposed algorithm needs only two scans of the dataset to generate frequent item sets.

4. EXAMPLE DEMONSTRATION

To check the correctness of the proposed algorithm, in this section an example is demonstrated using Apriori Algorithm and proposed algorithm. The results of both the algorithms are compared and it is observed that proposed algorithm is correct.

In this example, a dataset, with 8 transactions and 6 items, is taken. This example simulates the real time example, in which transactions are denoted using integers and items are denoted using alphabets. First the example is solved using the proposed algorithm. For this, the dataset is divided into two partitions, p1 and p2 each having 4 transactions. The Gmin_sup (Global Minimum support count) value is 4. Value of local support count is taken as 2 i.e. (Gmin_sup/Number of partitions). First of all, the dataset is scanned to calculate the support count for 1-item set where Support count is the value for an item appearing in a number of transactions. Table 2 and Table 3 show the partitions of the whole Dataset having 8 transactions.

Table 2. Partition-1 of the Dataset

TID	Items
1	A, C, D, F
2	A, B, E
3	B, F
4	A, C, E, F

Thus for all items the values are calculated as:

A=3, B=2, C=2, D=1, E=2, F=3

After calculating the support count, pruning is done. Items which are having less support count than that of the given min_siup value are pruned. Here Support_count (D) < min_sup, so 'D' is kept in negative border NB = {D}. Now represent the items in interval set representation as:

A= [1, 2] [4, 4], B= [2, 3], C= [1, 1] [4, 4], E= [2, 2] [4, 4], F=[1, 1] [3,4]

After representing the items in interval set representation, 2-itemsets are generated from 1-itemsets using interval intersection operation. Here the items generated are as follows with their interval set representations:

AB= [2, 2], BE = [2, 2], AC= [1, 1] [4, 4], BF = [3, 3], AE=[2,2] [4, 4], CE = [4, 4], AF= [1, 1] [4, 4], CF = [1, 1][4,4],BC=[], EF = [4, 4]

Now, calculate the support count of item sets using (1)

$$Support(c) = \sum_{1}^{i} \mathrm{End_i} - \mathrm{Start_i} + 1 \qquad (1)$$

where End_i, $Start_i$ are the ending and starting points in the interval set. For example support count of AB is

Support count = (2 - 2 + 1) = 1.

In this iteration, item sets AB, BE, BF, CE and EF are having support count=1 i.e. less than min-sup, so these are placed in negative border NB= {D, AB, BE, BF, CE, EF}. In next iteration, 3-itemsets are generated using interval intersection operation of 2-itemsets. The 3-iemsets are as follows:

ACE = [4, 4], ACF = [1, 1] [4, 4], AEF = [4, 4]

Support count is calculated using (1) and item sets ACE and AEF are having less support count than min-sup, so these are placed in negative border NB= {D, AB, BE, BF, CE, EF,ACE, AEF}.

Now there is only one frequent itemset i.e., ACF, so no further itemsets can be generated because to generate next level frequent item sets, two item sets are required. So, the largest frequent itemset for partition-1 is 'ACF'. Now, after calculating frequent itemsets for partition-1, frequent item sets for partition-2 (Table III) are calculated as follows:

Table 3. Partition-2 of the Dataset

TID	ITEMS
1	A, D, E, F
2	A, B, E, F
3	A, C, F
4	A, C, E, F

Frequent itemsets for Partition-2 are generated using same procedure as used for partition-1. So, the frequent itemsets are given as:

{ACF, AEF}

Now, after finding the frequent item sets from all the partitions, results are merged using the merging algorithm and a global scan of the dataset is required to check the support count of all the frequent item sets globally. For this, a global array is taken and all the results of partition-1 are put into this array. After this, items from each partition are taken one by one and are checked against global array. If any item is present in the global array, the value of the support count is added to value of support count of item being checked; otherwise the item is inserted as it is. Now, after putting all the item sets in the global array, each item set is checked for the global support count and if support count is found to be less, the item is discarded; otherwise it is kept in the global array which is the final result. {AEF} is having support count 3, so it is checked against the negative border. If item set {AEF} is found in negative border its support count is incremented by the value of the {AEF} in negative border. Now its value is found to be 4, so the final frequent item sets are {ACF, AEF}. Results are shown below in Table 4.

Table 4. Final results

ITEMS	SUP_COUNT
A	7
C	4
E	5
F	7
AC	4
AE	5
AF	6
CF	4
EF	4
ACF	4
AEF	4

Now, after generating the frequent item sets using the proposed algorithm, to verify the results of the propose algorithm, frequent item sets using the Apriori algorithm have also been found. Apriori Algorithm [1] works in two steps. First step is to generate the frequent item sets and the second step is to prune the item sets which are having less support count than minimum support count value. First of support count of all 1-item sets is calculated by scanning the dataset. After calculating the support count items are pruned. Now, (k+1) item sets are generated from k-item sets. According to Apriori algorithm, only those k-item sets can be combined whose (k-1) items are same i.e. ABC and ABE can be combined to generate ABCE because AB i.e. length (k-1) is same in both item sets. After generating the (k+1)-item sets, dataset is scanned to calculate the support count for newly generated item sets and items sets having less support count than that of minimum support count value are pruned. This whole process is repeated until no further item sets can be generated. The process is explained using the following example (Table 5).

Table 5. Example Global dataset

TID	ITEMS
1	A, C, D, F
2	A, B, E
3	B, F
4	A, C, E, F
5	A, D, E, F
6	A, B, E, F
7	A, C, F
8	A, C, E, F

There are basically two steps in Apriori, first is scanning and pruning and second is joining. First C_k is generated by scanning the dataset and calculating the support count for each item. After that those items having support count less than min_sup are pruned and result is stored in L_k.

Now L1 is self-joined to generate 2-itemsets and C2 is returned. According to Apriori principle two item sets can only be joined if their (k-1) items are same.

In L1, each element is having length-1, so each item is joined every other item and C2 is generated. Now those item sets having less support count than min_sup are pruned and result is stored in L2. This whole process is shown below in Table 6 (a, b, c, d, e, f).

Now L2 is self-joined and C3 is generated. In self-joining procedure, only the item sets which are having same (k-1)-items, can be joined. For example, in L2 item AC and AE can be joined because (k-1)-items are same i.e., A. After joining the item sets C3 is generated and items are pruned from C3 and L3 is produced as:

Table 6. Step by Step Demonstration of Apriori

ITEMS	SUP_ COUNT
A	7
B	2
C	4
D	2
E	5
F	7

(a) C1: Candidate Generation

ITEMS	SUP_ COUNT
A	7
C	4
E	5
F	7

(b) L1: Pruned Set

ITEMS	SUP_COUNT
AC	4
AE	5
AF	6
CE	2
CF	4
EF	4

(c) C2: Candidate Generation

ITEMS	SUP_ COUNT
AC	4
AE	5
AF	6
CF	4
EF	4

(d) L2: Pruned Set

ITEMS	SUP_COUNT
ACE	2
ACF	4
AEF	4

(e) C3: Candidate Generation

ITEMS	SUP_COUNT
ACF	4
AEF	4

(f) L3: Pruned Set

After generating L3, further no item sets can be generated, so ACF and AEF are final frequent item sets. Finally the frequent item sets which are produced using Apriori algorithm are shown in Table 7.

Table 7. Frequent Item sets using Apriori

ITEMS	SUP_ COUNT
A	7
C	4
E	5
F	7
AC	4
AE	5
AF	6
CF	4
EF	4
ACF	4
AEF	4

5. EXPERIMENTAL RESULTS

In this section, the performance of proposed algorithm is compared with Apriori Algorithm on the basis of produced results and execution time. The details are discussed below.

5.1 Experimental Environment

All the experiments are performed on a personal laptop with Intel Core2Duo 1.8 GHz processor, 2GB Primary Memory and 32-bit Windows operating System. Example Dataset is created manually which is described in the next section. Apriori Algorithm and Proposed algorithm are implemented using jdk1.8.0_51 version. In the experimental study, the effects of database size on the execution time and produced results by both of the algorithms are analyzed.

5.2 Dataset

Dataset consists of item sets represented using binary digits 0 and 1. 1 represents the presence of an item and 0 represents absence of an item. Numbers of rows are considered as total number of transaction. Numbers of columns are taken as total number of items in dataset. Numbers of transactions are taken in corresponding to the configuration file. To calculate the support count, numbers of ones are counted. This is a small dataset, consisting of only thirty transactions. Large sized datasets can be created using online data generators. Dataset is shown in the Fig. 1.

```
1 0 0 0 1 1 0 1 0 1 1 0 0 0 1 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0
1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 1 0 1 0
0 1 0 1 0 0 0 0 1 0 1 0 1 0 0 0 0 1 0 1 0 1 0 1 0 0 1 0 1 0
1 0 1 1 0 0 0 1 0 0 1 0 1 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0 1 0
0 0 0 0 1 0 1 0 1 0 1 1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 1 0 1 0
0 1 0 1 1 1 0 0 0 1 1 0 1 0 0 0 0 1 0 1 0 1 0 1 0 0 1 0 1 0
1 1 1 0 0 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1
0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1 0 1 0 1 0 0 0 0 1 0 1
1 0 1 1 0 0 0 1 0 0 1 0 1 0 0 0 0 1 0 1 1 0 1 0 0 0 0 1 0 1
0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1
1 0 1 0 0 0 0 1 0 1 1 1 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0 1 0 1
1 1 1 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1 0 1 0 0 0 0 1 0 1 1
0 1 0 1 0 0 0 0 1 0 1 0 1 0 0 0 0 1 0 1 0 1 0 1 0 0 1 0 1 0
1 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 1 0 0 0 1 0 0
0 0 0 0 1 0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 1 0 0 1 0 0 0 1 0 0
0 1 0 1 1 1 0 0 0 1 1 1 1 0 0 1 0 1 0 1 1 0 1 1 0 0 0 1 0 0
1 1 1 0 0 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 1 0 0 0 1 0 0
1 0 1 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0 1 0 1 0 1 1 0 0 0 1 0 0
0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 1 1 0 1 0 1 1 0 1 1
1 0 1 0 0 0 0 1 0 1 1 1 1 0 0 0 0 0 0 1 1 1 0 1 0 1 1 0 1 1
1 0 0 0 1 1 0 1 0 1 0 0 0 0 1 0 1 0 1 0 1 1 1 1 0 1 1 0 1 1
1 0 0 0 1 1 0 1 0 1 1 0 1 1 0 0 0 1 0 0 1 0 1 1 1 0 0 0 1 1
1 1 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 1
0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0 0 0 1 1
1 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 1 1 0 0 0 1 1
0 0 0 0 1 0 1 0 1 0 1 0 1 1 0 0 0 1 0 0 1 0 0 0 1 0 0 1 0 1
0 1 0 1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 0 0 1 0 0 0 1 0 0 1 0 1
1 1 1 0 0 1 0 1 0 1 1 0 1 1 0 0 0 1 0 0 1 0 0 0 1 0 0 1 0 1
1 0 1 1 0 0 0 1 0 0 0 1 0 1 0 0 1 0 1 0 1 0 0 0 1 0 0 1 0 1
0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 1 0 0 0 1 0 0 1 0 1
```

Fig. 1: Example Dataset

5.3 Performance comparison based on Execution Time

Performance comparison is done on the basis of execution time which in turn depends on the size of the dataset. In this experiment, different sized databases are taken. The value of minimum support count is taken as 40% of the number of transactions. The comparison is shown in the Fig. 2.

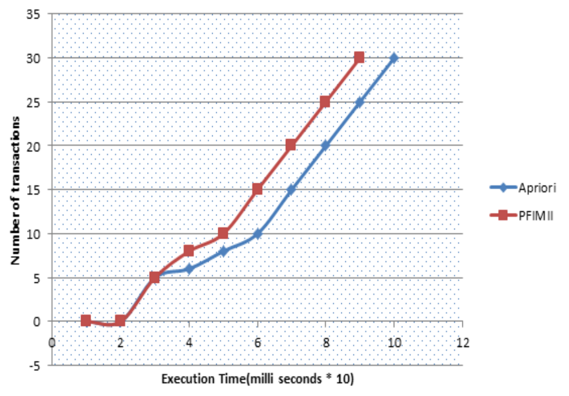

Fig. 2: Execution Time Comparison

Both the algorithms are executed on different sized datasets and execution time and results are compared. The results produced by both the algorithms are same as shown in section IV using an example. Here execution time based on different sized databases is compared. It is clearly shown that Proposed Algorithm takes less time as compared to Apriori Algorithm. The algorithms are compared on a very small dataset, but they can also be executed on large sized databases.

6. CONCLUSION AND FUTURE SCOPE

In this paper, a new algorithm is proposed, which tries to avoid identified problems and makes the process of finding frequent item sets efficient. Proposed algorithm creates many partitions of the dataset and performs the task of finding frequent item sets in parallel on each partition. Many of the previous algorithms make multiple scans of the dataset to determine the support count and frequent item sets. This makes the process time consuming and inefficient. But, proposed work takes only two scans of the dataset, thus makes the task less complex and efficient. Algorithm performs the task of frequent item sets in parallel on various partitions of the dataset which makes it faster. In future, the bit vector methods can be utilized to increase the efficiency of proposed algorithm.

7. REFERENCES

[1] Aggaraval R; Imielinski.t; Swami.A. “Mining Association Rules between Sets of Items in Large Databases”. ACM SIGMOD Conference. Washington DC, USA, 2013.

[2] Jiawei Han And Micheline kamber, “Frequent item set mining methods”, Data Mining concepts and techniques.

[3] Moore,R. E, R. Baker Kearfott and M. J. Cloud, “Introduction to interval analysis”, Siam,2009

[4] Siddharth Shah, N. C. chauhan, S. D. Bhanderi, “Incremental Mining of association rule: a survey”, International journal of computer science and information technology, vol. 3(3), 2012, 4071-4074

[5] H. Li, Yi Wang, D. Zhang. PFP: Parallel FP-Growth for Recommendation, Proceedings of the 2008 ACM conference on Recommender systems, October. 2008,pp. 23-2.

[6] Yungho-Leu, Vania Utami, “A new frequent item set mining algorithm based on interval intersection” in proceedings of Conference on machine learning and cybernatics, guangzhou 12-15 April, 2015.

[7] R. Bhaskar, S. Laxman, A. Smith, and A. Thakurta, “Discovering frequent patterns in sensitive data,” in Proc. 16th ACM SIGKDD Int. Conf. Knowl. DiscoveryData Mining, 2010, pp. 503–512.

[8] Ashok Savasere, Edward Omiecinski, Shamkant Navathe, “An efficient algorithm for mining association rules in large databases”, College of computing, Georgia Institute of Technology 2010.

[9] D. Cheung, J. Han, V. Ng, and C. Y. Wong. Large Databases: An Incremental Updating Technique. Proceedings of the 12th International Conference on Data Engineering, pages 106—114, February 1996.